THE UNIVERSITY *of* EDINBURGH

Semantics and Provenance of Configuration Programming Language μPuppet

Weili Fu

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2019

Abstract

Nowadays computing infrastructures have grown bigger in scale and more complex. Automated configuration management tools have taken the place of traditional approaches of configuration tasks, such as manual configuration and writing simple scripts, which become tedious, inefficient and error-prone to human mistakes. These tools typically contain their own specific configuration languages. In general, users write configuration specifications of the system and the tools compile them to configuration files that are deployed on each machine to change its configuration. Configuration specifications tend to be large and involve thousands of parameters and the relations between them. However automating configuration management does not prevent configuration errors that might cause system failures that are costly and time-consuming to remedy. Configuration languages normally support separation of concerns of different users, but there is limited work on access control and work flow to guarantee the correctness of the configurations when there are changes in the specifications, or the analysis tools for root causes for configuration errors. Configuration errors remains one of the dominant reasons for system failures (Oppenheimer et al., 2003).

In this thesis, we argue that provenance techniques, a dynamic technique of tracking data history developed in the database field, is a natural solution to providing configuration management tools with the ability of analysing configuration errors and identifying their root causes in the configuration specifications. For this purpose, we first choose a popular configuration management tool Puppet and present μPuppet which models the operational semantics of a core subset of Puppet language. In addition, we prototype an interpreter and a parser for μPuppet and compare their validation results by using more 50 Puppet manifests against the real Puppet language. Based on the formal semantics of μPuppet, we formalise where-, expression- and dependency-provenance which track the original inputs of an output data value, the primitive operation procedure that derives an output data value and all the inputs on which a generated output depended respectively in the process of compiling the configuration specifications. Further more, we establish the static correctness properties of three forms of provenance. We prove that where- and expression-provenance satisfy these properties. We find the limitation of dependency-provenance we define that does not record the full dependency information for some unusual value-overriding semantics, which

reveals the complexity of the Puppet language. We prove the partial correctness of dependency-provenance confined to a subset of μPuppet that excludes the unusual cases and propose possible solutions to deal with these unusual cases.

Acknowledgements

I would like to express my gratitude to both of my supervisors James Cheney and Paul Anderson. They have been fully supporting and encouraging in the whole progress of doing my PhD project, while providing their constant help and understanding. From their guide, I have learned a lot of knowledge, known gradually what doing research is and how to do research, especially acknowledge the right attitude of doing research. Their professional attitude towards doing research has been always enlightening and inspiring for me on the way of doing my Phd. In spite of their constant supports, I have become more independent through their guide, for which I feel most grateful.

Moreover, I also want to thank LFCS group, Informatics of UoE where I have been studying my PhD. All the staffs and colleagues have made a enjoyable, inspiring and professional environment to study in. Particularly, they hold regular seminars, group meetings and workshops with interesting topics, which broaden my foundational understanding of the knowledge in a larger programming-language field than my project topic and supply me with precious insights about doing research. I have been enjoying and feeling informed talking with any of them. They made a great excitement in the period of doing my PhD.

I want to express my gratefulness to my colleagues and friend Simon Fowler for giving his kind moral support in the difficult time and providing immediate helps.

Finally, I want to thank to my parents who provide their greatest support and understanding from far away back in China.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Weili Fu*)

Table of Contents

1 Introduction **1**

2 Background **7**

2.1 System Configurations and Configuration Management Tools . . . 7

2.1.1 Configuration Management Tools 8

2.1.2 Configuration Languages 9

2.1.3 Puppet 10

2.2 Operational Semantics 11

2.2.1 Syntax 12

2.2.2 Semantics 13

2.3 Provenance 17

2.3.1 Lineage, Why-provenance and Where provenance 17

2.3.2 How Provenance 18

2.3.3 Dependency Provenance 19

2.4 Language-based and Information Flow Security 20

2.5 Synthesis-Based System Repair 21

3 Semantics of μPuppet **23**

3.1 Introduction 23

3.2 Puppet 25

3.2.1 Puppet: key concepts 25

3.3 μPuppet 33

3.3.1 Abstract syntax 33

3.3.2 Operational Semantics 34

3.4 Metatheory 46

3.5 Implementation and Evaluation 47

3.5.1 Test cases and results 48

3.5.2 Unsupported features 50
3.6 Related work 53
3.7 Conclusions 54

4 Where and Expression Provenance **57**
4.1 Methodology 58
4.2 Where-Provenance 59
4.2.1 Annotated Compact Grammar 60
4.2.2 Propagation of Evaluation in the Evaluation 61
4.2.3 Correctness of Where-provenance 69
4.3 Expression-Provenance 80
4.3.1 Annotated Compact Grammar 81
4.3.2 Propagation of Annotations in the Evaluation 82
4.3.3 Correctness of Expression-Provenance 87

5 Dependency Provenance **97**
5.1 Background 97
5.2 Annotations and Propagation of Annotations 99
5.2.1 Annotated Compact Grammar 99
5.2.2 Propagation of Annotations in μPuppet Evaluation 102
5.3 Correctness of Dependency-Provenance 113
5.3.1 Limitation of Method 114
5.3.2 Equivalence Relation 116
5.3.3 Correctness of Dependency Provenance 120

6 Conclusion **137**

Bibliography **141**

A Semantics of μPuppet **147**
A.1 Glossary 147
A.2 Features supported 149
A.3 Operational semantics 151
A.3.1 Environment operations 151
A.3.2 Expressions $(\sigma, \kappa, v_C, e \xrightarrow{\alpha} e')$ 153
A.3.3 Resources $(\sigma, \kappa, H \xrightarrow{\alpha}_{\mathsf{H}} H'$ and $\sigma, \kappa, e : H \xrightarrow{\alpha}_{\mathsf{R}} e' : H')$ 156
A.3.4 Statements $(\sigma, \kappa, v_C, s \xrightarrow{\alpha}_{\mathsf{s}} \sigma', \kappa', v_C', s')$ 156

A.3.5 Manifests $(\sigma, \kappa, v_{\mathrm{C}}, m \xrightarrow{N}_{\mathsf{m}} \sigma', \kappa', v'_{\mathrm{C}}, m')$ 160

B Where Provenance **163**

B.1 Propagation of Annotations in the Evaluation 163

B.1.1 Expressions 163

B.1.2 Resources 163

B.1.3 Statements 164

B.1.4 Manifests 167

B.2 Proofs of the invariant in Expression Evaluation 168

B.3 Proofs of Invariant in Statement Evaluation 170

B.3.1 Proofs of the invariant in Manifest Evaluation 179

C Expression Provenance **183**

C.1 Propagation of Annotations in the Evaluation 183

C.1.1 Expressions 183

C.1.2 Resources 183

C.1.3 Statements 184

C.1.4 Manifests 184

C.2 Proofs for Invariants in Expression Evaluation 184

C.2.1 Proofs for Invariants in Resource Evaluation 188

C.3 Proof of Invariant in Statement Evaluation 189

C.4 Proof of Invariant in Manifest Evaluation 198

D Dependency Provenance **203**

D.1 Propagation of Annotations in the Evaluation 203

D.1.1 Expressions 203

D.1.2 Statements 204

D.1.3 Manifests 206

Chapter 1

Introduction

Nowadays computing infrastructures have grown bigger in scale and more complex. These systems constitute many devices and machines of different kinds spread over different locations. Each of the devices or machines could involve hundreds to thousands of parameters to configure. Besides, it is common that there exist relations and dependencies between the internal parameters in a machine and the external parameters to be taken care of, which is the most difficult aspect in the configuration task. Due to the complexity of the systems, traditional configuration approaches, such as manual management and writing low-level scripts, become infeasible. They are ineffective to configure many machines that could share lots of common parameters and error-prone due to human errors. Automated configuration tools becomes prevailing for configuration management. The popular tools include Puppet (Puppet, 2016), Chef (Chef, 2009), Ansible (DeHaan, 2012) and Saltstack (Hatch, 2011). Such tools improve the efficiency and help to avoid human-mistakes to some extent. However, they do not prevent configuration errors that cause system failures that are costly and time-consuming to remedy. Configuration errors are one of the main causes of large scale systems' failures (Oppenheimer et al., 2003).

These tools bring their own problems due to their inherent characterizations that will cause misconfigurations. In turn, they make troubleshooting the configuration errors difficult. They typically provide their own configuration languages that can specify the configurations in a high level. These languages can be declarative or imperative. The declarative languages allow users to describe the desired states of the systems as the configuration specification. The specification is gathered and compiled centrally by the server machine. Then the tool will transfer

the result to the actions in the individual machine to modify its configuration accordingly. Configuration tasks normally take a collaborative effort of many administrators. Each of them takes charge of a part of the system or an aspect of the configuration. Thus the configuration specification is produced by many users with different concerns. These languages take into account the separate concerns of many users by providing language features to support. They share the properties of the inheritance of classes that gives modularity and the value-overriding of a chain of users. However, these tools suffer the lack of access control on the users and work flow techniques to confine the inheritance behaviour. Instead they only use the access control techniques defined for software management. Then there is the case that many users can influence one parameter in the configuration. Due to the configuration under constant changes, when the users change their specifications on these parameters, its final result may not reflect the desired status, which might cause system failures. Anderson and Cheney (2012) illustrated this situation by an example. I use Puppet language to illustrate and adapt it in Puppet language. Programs in Puppet language is typically contributed by multi-users and centred to execute.

```
1 class a { %class a is defined by Alice
2  ts = ts.reliable.com
3  ...
4 }
5 class b inherit a { %class b is defined by Bob
6  ts = sales.widget.com
7  ...
8 }
9 include b
```

In this example, Alice is an external upper-level user who defined the class a and Bob is a local sub-system administrator who defined the class b. Alice specified some default configuration in the class a including one parameter ts which is an address for a "timeserver" against which to synchronise the operating system time. Bob can inherit class a and override the default parameters in it. He overrode and assign ts with a different address "sales.widget.com". The class specifications Alice and Bob wrote might be in different files or modules among many others, so that the over-riding or shadowing of the parameter ts might not be easy to notice. At some point, Alice changes ts's value to another server "unreliable.com" while Bob remains unchanged as the following. Bob's overriding on ts will mask any change made by Alice.

```
1  class a { %class a is defined by Alice
2   ts =  ts.unreliable.com
3   ...
4  }
5  class b inherit a { %class b is defined by Bob
6   ts = sales.widget.com
7   ...
8  }
9  include b
```

Later on, Bob removes his overriding on ts. Then the system Bob takes charge of will inherit the default values defined by Alice. If it causes system failure, Bob will be blamed for the misconfiguration.

```
1  class a { %class a is defined by Alice
2   ts =  ts.unreliable.com
3   ...
4  }
5  class b inherit a { %class b is defined by Bob
6  %ts = sales.widget.com
7   ...
8  }
9  include b
```

Although the changes on the configuration specification are made as users' desired, the final result may not reflect these changes. They might be masked in the process of assigning parameters in the configuration languages.

Since there may be thousands of parameters in one machine and hundreds or thousands of machines in one system, it is hard to identify the responsible party for a result value in the final configuration. Unfortunately, there is little work on auditing of configuration changes. Vanbrabant et al. (2009) presented the tool ACHEL where access control was designed in the tool Puppet to decide whether a change should be made or not. This was realised by extracting semantic meanings of configuration changes in the level of configuration specifications. However the changes it concerns are only the last ones made in the configuration specification. In the case of the last example, Bob will still be blamed. It does not consider the changes made before. So it does not help in this case when Alice is the real cause of the error.

Attariyan and Flinn (2008, 2010) presented a tool ConfAid which can pinpoint the root cause of a configuration problem. This tool uses dynamic information analysis to monitor the causal dependencies introduced by control and data flow. Their analysis is performed in the processes of deploying the configurations in a bottom-up matter, by rebuilding the error on a reference computer and obtain-

ing the causal dependencies of the test case execution to detect the similarity between a configuration state on a sick computer and the reference computer. However this dependency analysis method does not rely on the execution process of the configuration specifications and track the error back to the configuration specifications. Thus it cannot reason about the root cause of a problem in the configuration specification.

Our research hypothesis is that formal semantics and provenance tracking can provide a principled foundation for understanding configuration errors. Due to the lack of information on a configuration error, we want to target the place in the configuration specification where a configuration error came from and know how it has been generated in the specification. We propose to build on provenance that has been developed in the database field. Provenance is a technique to record the historical origin of data in databases. Since the databases grow larger in scale and undertakes changes constantly, it is hard to analyse the source of an error in the query results. Provenance has been designed in terms of the semantics of the query languages so that it is realised along the execution of the queries. It collects the relevant inputs of a query result and its computational process dynamically along the execution. The application of provenance in databases is comparable to it in the configuration languages since the query errors can be related to the configuration errors and the query languages are related to the configuration languages. There have been several forms of provenance presented that provide different aspect of history information about a query result. Where-provenance (Cheney et al., 2009) provides the original input data of an output in query result; why- and how-provenance (Cheney et al., 2009) explains how an output has been generated from the inputs; dependency-provenance (Cheney et al., 2007) records all the inputs that could influence the generation of an output. We will adapt them for the information in the configuration specifications to explain the configuration errors.

To realise this goal, we will continue to use the method of establishing provenance in databases that is to annotate the inputs and modify the semantics of configuration languages so as to propagate the labels on the inputs and compute the desired provenance. We choose a popular configuration management tool Puppet and build provenance on the semantics of its language *Puppet language*. Puppet language is a mostly declarative configuration language built in an ad-hoc style. It is undertaking frequent addition and changes. The lack of a formal

semantics is an obstacle to formalise the analysis on the language. Thus as the first step toward defining provenance, we will formalise Puppet language. We first identify a core subset of this language and formalise its syntax and the semantics using operational semantics (Plotkin, 2004) in Chapter 3. We call this formalised language *μPuppet*. We also build an interpreter and a parser for μPuppet based on its formalisation. We then will evaluate them by more than 50 examples that cover all the features of μPuppet and compare the results against that of the real Puppet language. Then we establish *where-* and *expression-provenance* in Chapter 4. Where-provenance locates the input data value of an output value back in the μPuppet specification. Expression-provenance further records the primitive computations in the execution of the specifications that derive an output data value. We also define the static characterisation of the correctness of where- and expression-provenance and prove they are correct according to this characterisation. Then we formalise *dependency-provenance* in Chapter 5 that is to collect all the data values in the specification that could influence an output in the configuration file. To do it, the conditional constructs in μPuppet are taken into account for computing provenance, besides the primitive operators. We again define the static correctness characterization of dependency-provenance. However, due to the complexity of μPuppet language, dependency-provenance does not track the full dependency information of the unusual cases introduced by the value overriding. We prove that it satisfies the correctness with the limitation of excluding the unusual semantics from μPuppet. We believe there are several possible solutions of this problem that can be achieved. One is to modify the annotation propagation in the semantics of μPuppet by including more relevant labels and adjusting the correctness characterization accordingly. However this will enlarge the size of dependency-information and the overhead of computing it. The increased information will make it more difficult for users to understand. We need to find a balance between the usefulness of the provenance and the cost. The other possibility is to constrain μPuppet specifications to eliminate the unusual cases in a formal way. We leave it for further work.

Our main contributions in this thesis are an operational semantics of μPuppet, establishing where-, expression- and dependency-provenance on the top of this semantics to track different forms of information about output data values in the inputs, and characterising their correctness and proving they are correct or partially correct. The formal semantics of μPuppet is an important step for in-

troducing analysis in the Puppet language. Beside building provenance, it might be useful for other analysis purposes. The limitation of dependency-provenance exposes the complexity of the Puppet language, and gives evidence for programming designers to improve the language.

Chapter 2

Background

2.1 System Configurations and Configuration Management Tools

Computing infrastructures consist of hardware and software, which have many parameters that need to be configured in order to perform in conformance to each other and the desired requirements. The infrastructures are typically subject to changes due to updating the hardware or software in them or the changed requirements on them. The term *configuration* has been used in many contexts with different meanings, such as hardware configuration, software configuration management, network configuration management and distributed application configuration. However there is no generally accepted definition for the term *system configuration*. System configuration is a task to configure the whole infrastructure including the hardware and software to make the system perform the desired functionalities. Moreover, when the requirements on the infrastructures change or the environment changes happen such as hardware or software updates, system configuration should be maintained to conform with these changes in order to keep the system performing the desired functionality. *Configuration specification* is the description how the entire system should be configured. The specification could be in different detail levels. For example, it could specify the common setting of all the machines in the system and leave the differences among them to further configuration procedure, or could describe in detail how every machine should be configured.

System configuration faces many difficulties. First of all, the hardware of an

infrastructure could include diverse kinds of machines, from laptops to database servers. Their configurations have different requirements. Secondly, as we have mentioned, an infrastructure undertakes constant changes. These changes could be due to critical hardware or software system failure, the new functionality from upgraded software, configuration refactoring and new or upgraded hardware and so on (Anderson, 2006). Thirdly, configuring a system is not only to configure the individual machines but also to take care of the corresponding relationships between the configurations in machines. Such relationships could exist either between different machines or between different components in one machine. For example, a Web service needs to configure the web server. The entire DNS (domain name space) should be configured with the information about the web server, as well as the firewall and so on. Fourthly, a configuration specification of a large infrastructure could involve many people. Each of them manges the configuration of a part of an infrastructure. Or the configuration of some aspect of the system, such as the mail system, the network and the security functionalities, can be assigned to a specialist. These people may have different aspects of knowledge and may not have sufficient contacts between them. Finally, large systems are typically distributed. They share all the difficulties of distributed programming, such as communication, failure recovery and latency (Anderson, 2006).

2.1.1 Configuration Management Tools

Automating system configuration or configuration management becomes more and more appealing when the infrastructures become larger and more complex. System configuration was traditionally dealt with manually. However, when an infrastructure becomes larger, manual configuration is less efficient and error-prone. Then automated techniques have taken into part of system configuration gradually from the low level of automation techniques such as cloning, scripting programming to the high level one, i.e. programming languages. Many automated configuration tools have arisen in these years, which are based on some *model* for structuring configurations in the high level. Such models work as standards of interoperability between devices in an automatic way. Simple approaches for configurations such as basic scripting do not involve any underlying model, while the configuration management tools based on models include their own language

for configuration specifications.

LCFG and Puppet are configuration management tools based on some intermediate models. These models provide the frameworks of the components that are used to describe the configuration in a declarative way. Then these components are composed and instantiated for different machines.

There have been little academic works on presenting system configurations that are widely accepted so we do not refer any of them here to avoid misunderstanding for people with different background.

2.1.2 Configuration Languages

All the configuration frameworks lie somewhere on the spectrum between "imperative" and "declarative". At the imperative end, developers use conventional scripting languages to automate common tasks. It is left to the developer to make sure that steps are performed in the right order, and that any unnecessary tasks are not (potentially harmfully) executed anyway. At the declarative end of the spectrum, the desired system configuration is *specified* in some higher-level way and it is up to the configuration framework to determine how to *realise* the specification: that is, how to generate a compliant configuration, or adapt an already-configured system to match a new desired specification.

Most existing frameworks have both imperative and declarative aspects, such as Chef (Marschall, 2013), CFEngine (Zamboni, 2012). Chef, CFEngine, and Ansible (Geerling, 2015) are imperative in relation to dependency management; the order in which tasks are run must be specified. Chef and CFEngine are declarative in that a configuration is specified as a desired target state, and only the actions necessary to end up in a compliant state are executed. (This is called *convergence* in configuration management speak.) The Puppet framework Turnbull (2008) lies more towards the declarative end, in that the order in which configuration tasks are carried out is also left mostly to the framework. Puppet also provides a self-contained *configuration language* in which specifications are written, in contrast to some other systems. (Chef specifications are written in Ruby, for example, whereas Ansible is YAML-based.)

Configuration languages often have features in common with general-purpose programming languages, such as variables, expressions, assignment, and conditionals. One aim of configuration languages is to model different kind of services

in the configuration tasks in a high level. Some, including Puppet, also include "object-oriented" features such as classes and inheritance. However, (declarative) configuration languages differ from regular programming or scripting languages in that they mainly provide mechanisms for specifying, rather than realising, configurations. While some "imperative" features that can directly mutate system state are available in Puppet, their use is generally discouraged.

2.1.3 Puppet

Puppet is one of the most popular open source configuration management tools used by a big community. According to the company, Puppet has been chosen and used by 40,000 organizations.

Puppet uses several terms – especially *compile*, *declare*, and *class* – in ways that differ from standard usage in programming languages and semantics. We introduce these terms with their Puppet meanings in this section, and use those meanings for the rest of the chapter. To aid the reader, we include a glossary of Puppet terms as Appendix A.1.

The basic workflow for configuring a single machine (*node*) using Puppet is shown in Figure 2.1. A *Puppet agent* running on the node to be configured contacts the *Puppet master* running on a server, and sends a check-in request containing local information, technically called *facts*, such as the name of the operating system running on the client node. Using this information, along with a centrally maintained configuration specification called the *manifest*, the Puppet master *compiles* a *catalog* specific to that node. The manifest is written in a high-level language, the Puppet programming language (often referred to simply as Puppet), and consists of *declarations* of *resources*, along with other program constructs used to define resources and specify how they are assigned to nodes. A resource is simply a collection of key-value pairs, along with a *title*, of a particular *resource type*; "declaring" a resource means specifying that a resource of that type exists in the target configuration. The catalog resulting from compilation is the set of resources computed for the target node, along with other metadata such as ordering information among resources. The agent receives the compiled catalog and *applies* it to reconfigure the client machine, ideally producing a compliant state. Finally, it sends a status report back to the master indicating success or failure.

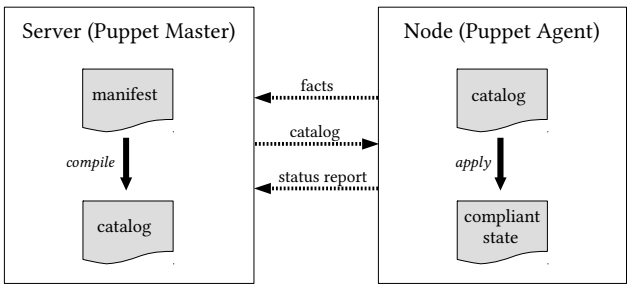

Figure 2.1: Puppet overview

Figure 2.1 depicts the interaction between a single agent and master. In a large-scale system, there may be hundreds or thousands of nodes configured by a single master. The manifest can describe how to configure all of the machines in the system, and parameters that need to be coordinated among machines can be specified in one place. A given run of the Puppet manifest compiler considers only a single node at a time.

2.2 Operational Semantics

Any programming language has two linguistic aspects, its syntax and semantics. The syntax in general consists of a set of classes representing different phrases in the language. The semantic meaning of this syntax describes how they behave so to reach the final result of a program in this language.

Three kinds of semantics for programming languages have been established. They are operational semantics, denotational semantics and axiomatic semantics. We use operational semantics in this thesis, so we focus on introducing it as background. *Operational* means that the semantics or the behaviour of a programming language are defined in terms of the syntactic transformations of programs and simple operations on discrete data (Plotkin, 2004).

There have been some machines proposed to interpret a programming language, such as Plotkin's $\langle S, M, C \rangle$ machine. It gives a flavour of operational semantics but is still not direct. Such machines are based on *transition systems*.

Definition 2.2.1. (Transition System) A *Transition System* is a structure $\langle \Gamma, \rightarrow \rangle$ where Γ is a set of elements, γ, called *configurations* and $\rightarrow \subseteq \Gamma \times \Gamma$ is a binary relation, called the transition relation. Read $\gamma \rightarrow \gamma'$ as saying that there is a transition from the configuration γ to the configuration γ'. (Plotkin, 2004)

$$
\begin{aligned}
\langle \texttt{boolean} \rangle &::= \texttt{true} \mid \texttt{false} \\
\langle \texttt{value} \rangle &::= \texttt{Integer} \mid \langle \texttt{boolean} \rangle \\
\langle \texttt{binary\_oper} \rangle &::= + \mid - \mid \ldots
\end{aligned}
$$

Figure 2.2: Concrete Syntax of Values in L

$$
\begin{aligned}
\langle \texttt{value\_exp} \rangle &::= \texttt{true} \mid \texttt{false} \\
&\mid \langle \texttt{value\_exp} \rangle \, \langle \texttt{binary\_oper} \rangle \, \langle \texttt{value\_exp} \rangle
\end{aligned}
$$

Figure 2.3: Concrete Syntax of Value Expressions in L

The $\langle S, M, C \rangle$ machine works as transitions based on the expressions or commands C on the top of the control stacks. It is not defined in the intuition for operational semantics since the transitions are not *syntax-directed*. For example, the transitions for a sequence of command $c; c'$ cannot be defined by the transitions for each c and c' respectively.

2.2.1 Syntax

We use an example to illustrate operational semantics. For the consistency of content, we choose a subset of our μPuppet, called L, as our initial language. The syntax of a language is first written as *concrete syntax* which is usually defined using a context-free grammar. It represents how the language is written. The concrete syntax of values and value expressions is shown in Figure 2.2 and 2.3. *Abstract Syntax* is abstracted from the concrete syntax of a language and represents the real structure of a language. The process is accomplished by a parser.

The abstract syntax for arithmetic expressions is shown in Figure 2.4.

As we mentioned there are different syntactic classes in a language. They can be differentiated again by *basic syntactic sets* and *derived syntactic sets*. Basic syntactic sets include all the basic data of different types and variables. In

$$e \;::=\; i \mid e_1 + e_2 \mid e_1 - e_2$$

Figure 2.4: Abstract Syntax of Arithmetic Expressions in L

$$\begin{aligned} e \;::=\;& i \mid \texttt{true} \mid \texttt{false} \mid v \\ \mid\;& e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\ \mid\;& e_1 = e_2 \mid e_1 \,\texttt{and}\, e_2 \mid e_1 \,\texttt{or}\, e_2 \mid \texttt{!}e \end{aligned}$$

Figure 2.5: Arithmetic Expressions and Boolean Expressions

our example, these data are truth values, natural numbers and variables. In our language L we take truth values `true` as truth and `false` as false, natural numbers as i and variables represented as v. Derived syntactic sets cover expressions and commands in a language. In L we include arithmetic expressions, boolean expressions and commands as shown in Figure 2.5 and 2.6.

2.2.2 Semantics

2.2.2.1 Semantics of Expressions

As in $\langle S, M, C\rangle$ machine, an expression is transformed into its final result in steps or transitions. If we skip some uncrucial transitions which put the elements from control stack to S, the steps left here are the addition steps. For example, $(1+(2+3))+(4+5)$ would be computed as the following transitions. (Plotkin, 2004)

$$(1+(2+3))+(4+5) \to (1+5)+(4+5) \to 6+(4+5) \to 6+9 \to 15$$

$$s \;::=\; e \mid s_1 \_ s_2 \mid x = e \mid \texttt{if}\ e\ \{s\}\ \texttt{else}\ \{s\}$$

Figure 2.6: Commands

These transition sequences of expressions are called *reduction* sequences or *derivations*. They constitute the procedure of expression *evaluation*. This procedure processes the expression from left to right and can be summarised by a specification as the following:

2.2.2.1.1 To evaluate the expression $e_1 + e_2$

Constants: if e_1 or e_2 are constants, do not do anything.

Sum:

(1) If e_1 is not a constant, evaluate it to a number i_1 as result.

(2) If e_2 is not a constant, evaluate it to a number i_2 as result.

(3) Add i_1 to i_2, get the finally result $i_1 +_N i_2 = i_3$.

This specification is *syntax-directed* since it works for any expression $e_1 + e_2$. From this specification, a formal set of rules for sum expressions is formalised on which the transitions of evaluating the expressions are based. The procedure of the evaluation makes the derivation of the form

$$e = e_1 \to e_2 \to \ldots \to e_{n-1} \to e_n = i$$

If we look at the first step of the specification above, we get the following

(1) If e_1 is not a constant, then evaluate it to a new expression e_1' as result.

(2) If e_1 is a constant and e_2 is not, then evaluate e_2 to a new expression e_2'.

(3) If both e_1 and e_2 are constants, then add i_1 to i_2 to give a number.

This specification can be summarised as the following rules

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2}\ (1) \qquad \frac{e_2 \to e_2'}{i + e_2 \to i + e_2'}\ (2) \qquad \frac{}{i_1 + i_2 \to i_1 +_N i_2}\ (3)$$

These rules state the specification above in the sense that in rule (1) if e_1 is not a constant and evaluated to e_1', then evaluating $e_1 + e_2$ to $e_1' + e_2$ is the first step. Rule (2) describes the second option in the specification is the similar way. Rule (3) is the same as the third option in the specification.

To extend these rules for dealing with variables, we need an extra construct *memory* as in $\langle S, M, C\rangle$ machine. We name memory as σ. Then the configuration of the transition for the evaluation of the full set of expressions becomes the combination of an expression e and the current memory σ the evaluation is based on. The transition relation becomes

$$\sigma, e \to e'$$

The transition of evaluating a variable v involves accessing the memory for the value of this variable. It is written as

$$\sigma, v \to \sigma(v)$$

where $\sigma(v)$ can be seen as a function $\sigma : Variable \to Natural$. The original rules can be extended as follows under the new relations of transitions

$$\frac{\sigma, e_1 \to e_1'}{\sigma, e_1 + e_2 \to e_1 + e_2} \qquad \frac{\sigma, e_2 \to e_2'}{\sigma, i + e_2 \to i + e_2} \qquad \frac{}{\sigma, i_1 + i_2 \to i_1 +_N i_2}$$

2.2.2.2 Semantics of Commands

As we discussed for critical transitions for expressions, we also consider the important transitions for executing commands in $\langle S, M, C\rangle$ machines. We take the transition steps that change system memory M but ignore the steps that move symbols in control C to the stack S. As for expressions, we define the configuration for our transitions for commands as σ, c. Let us see an example for the crucial transition we are concerned with. We take the commands as a sequence of commands of assignments $z := x; x := y; y := z$ and the store σ as $\{x := a, y := b, z := c\}$. The transitions for a sequence of commands of assignments $z := x; x := y; y := z$ are as the following

$$\begin{aligned} & \{x := a, y := b, z := c\}, z := x; x := y; y := z \\ \to\ & \{x := a, y := b, z := a\}, x := y; y := z \\ \to\ & \{x := b, y := b, z := a\}, y := z \\ \to\ & \{x := b, y := a, z := a\}, \varepsilon \end{aligned}$$

Then the transitions for commands can be seen to according to the following specifications

2.2.2.2.1 To evaluate a command c in a store σ

Empty command If c is empty, then the transition leads to the current store σ.

Assignment If c is an assignment $x := e$, then evaluate e in store σ. If the result is i, then change the store σ so that $\sigma(x) = i$.

Composition If c is the sequence of commands $c_1; c_2$, then

(1) evaluate the command c_1 in the store σ and get the new store σ_1

(2) evaluate the command c_2 in the store σ_1 and get the new store σ_2

This specification for transition is syntax-directed again. Considering the termination of execution for commands, we take the configurations as $\sigma, c \cup \sigma$ where σ is the termination configuration. The transitions for executing commands are in the form $\sigma, c \to \sigma', c'$. The transitions for the configuration σ, c is the following

$$\sigma, c \to \sigma_1, c_1 \to \sigma_2, c_2 \to \ldots \sigma_{n-1}, c_{n-1} \to \sigma_n$$

It describes that to execute c, first execute c in store σ and derive c_1 and σ_1. The rest of the execution of c starts from executing c_1 in store σ_1. Then consider the first transition of executing command c, we get the following rules for the transitions

$$\frac{}{\sigma, \varepsilon \to \sigma}\ (\textsc{Empty}) \qquad \frac{\sigma, e \to e'}{\sigma, x := e \to \sigma, x := e'}\ \textsc{AssI} \qquad \frac{}{\sigma, x := i \to \sigma[i/x]}\ \textsc{AssII}$$

$$\frac{\sigma, c_1 \to \sigma', c_1'}{\sigma, c_1; c_2 \to \sigma', c_1'; c_2}\ \textsc{CompI} \qquad \frac{\sigma, c_1 \to \sigma'}{\sigma, c_1; c_2 \to \sigma', c_2}\ \textsc{CompII}$$

We extend the specification to deal with the conditional command `if` b c_1 `else` c_2.

2.2.2.2.2 To evaluate a conditional command `if` b c_1 `else` c_2.in a store σ

Evaluate the boolean expression b in store σ.

(1) If the result is `true`, then evaluate c_1 in store σ.

(2) If the result is `false`, then evaluate c_2 in store σ.

The rules for the first step of transitions for conditional commands are

$$\frac{\sigma, b \to \sigma, b'}{\sigma, \texttt{if } b\ c_1\ \texttt{else } c_2 \to \sigma, \texttt{if } b'\ c_1\ \texttt{else } c_2}\ \textsc{CondB}$$

$$\frac{\sigma, b \to \sigma, \texttt{true}}{\sigma, \texttt{if } b\ c_1\ \texttt{else } c_2 \to \sigma, c_1}\ \textsc{CondI} \qquad \frac{\sigma, b \to \sigma, \texttt{false}}{\sigma, \texttt{if } b\ c_1\ \texttt{else } c_2 \to \sigma, c_2}\ \textsc{CondII}$$

2.3 Provenance

Provenance has been investigated in the database field. It was designed to record the relevant history information of data. It has become more important over the last years since data now comes not only from trusted sources but also from the Internet. The integrity of data is not ensured if one does not know the provenance information of the result data. Furthermore, some applications access a collection of databases. For the users of an application, it is important to know the provenance of data. In Cheney et al.'s survey (2009), the three most common forms of provenance in the relational database setting which were *where-*, *why-* and *how-* provenance have been summarized and compared to each other. Where provenance provided the information of the locations of the input data of some parts of the output of a query. Why provenance collected the set of input records that might contribute to the output of a query. How provenance recorded the full procedure of how the output is produced.

2.3.1 Lineage, Why-provenance and Where provenance

Cui et al. (2000) were among the first to introduce a notion of provenance of data for relational databases, called *lineage*. It relates one output entry to all the input entries that contribute to generating this output. According to Cui et al., each entry in the database is labelled with a symbol called an *annotation*. For each entry in the output, it is labelled by a set of annotations that correspond to all the entries in the database that contribute the entry in this output entry. Such a set of labels is called a "proof" or "witness". However such sets do not necessarily correspond to the minimal set of entries that could produce an output entry.

Buneman et al. (2001) introduced *why-provenance* for a semi-structured data model that is more general than the relational model. Why-provenance can cap-

ture different witnesses for an output entry. They introduced the concept of *witness basis* where one of the tuples has to appear in why-provenance, which formalised the idea of multiple witnesses for one output tuple.

The authors modified a *deterministic tree model*, an edge-labelled tree model, in which each node is uniquely determined by a path of edge labels from root node to that node. They then proposed a query language named *Deterministic QL*. The authors also put restrictions on both syntax and interpretation of queries so as to make them sound.

Based on the deterministic model and the query language, two meanings of provenance and how to compute provenance were formulated. The authors described the difference between why- and where-provenance for the first time. Why-provenance is to address the relevant inputs as witnesses that influence one output in a query result; while where-provenance indicates the location in the input where a value in the output tuple is copied from. Why-provenance has been studied for relational models while in this paper a syntactic approach has been used to explore why-provenance and where-provenance for more general models.

2.3.2 How Provenance

Green et al. (2007a) proposed commutative semirings as a model for annotations in relational databases which describes how the output tuple was computed. They first identified the similarity in query answering in four extensions of annotated relations, such as incomplete databases generalising RA to conditional tables where relations are annotated with Boolean formulas, probabilistic databases generalising relational algebra to event tables which is a form of annotated relations, computing lineage for the output tuples of queries in data warehousing, and finally RA on bag semantics. The bag semantics means a multiset that is a set where an element can appear more than one time. They proposed to use commutative semirings as a general algebraic structure for these models. The authors argued that the polynomials of commutative semirings formed a symbolic representation of semiring calculations that indeed captures the computation information of RA querying from inputs to outputs.

This paper gave examples of annotated RA applied to incomplete databases, multisets and probabilistic databases. In every case, the tuples in databases are labeled in some way. For incomplete databases, Imieliński and Lipski (1984) cre-

ated c-tables where the tuples in databases are labelled with boolean variables. In multiset or bag applications, numbers represent the multiplicity of the tuple in the multiset. In probabilistic databases, the tuples are labelled with a value between 0 and 1 to represent their probabilistics. In these applications, query answering is also extended to compute the annotations for the outputs of queries. The authors observed the similarity among these relational algebras and generalised them to a common algebra model. They first generalised the tags (annotations) with various kinds of information and then the *positive algebra* noted as RA^+ for such annotated-tuple relations.

This paper also pointed out that computing why-provenance for queries in RA^+ would not provide enough information if the tags of two output tuples are the same while they were computed from different tuples, i.e. different tags. They found out using the different operations of the semiring, positive algebra can fully record how an output tuple is produced. Thus they introduced *positive algebra provenance semiring* which was the semiring of polynomials. They show that the annotation semantics given by polynomial semirings is as general as (or more general than) the semantics for any other semiring.

2.3.3 Dependency Provenance

Cheney et al. (2007) tried to understand the mathematical or semantic foundations of data provenance. Inspired by the motivations in some very different fields such as program slicing, the authors built up a foundation for a new form of provenance *dependency-provenance*. They proposed that dependency analysis and slicing techniques familiar from programming languages provide a suitable foundation for an interesting class of provenance techniques. They gave an example for static and dynamic slicing. This slicing technique means to identify a subset of a program that contributes to some result of the output. It has been used in the process of debugging. The authors tried to explain how to understand data dependency provenance in databases in the light of programming slicing as *data slice*. It used three queries in relational database to illustrate dependency provenance.

The authors considered annotation propagating semantics for the queries of *nested relational calculus* (NRC) which is a core database query language closely related to *monad algebra*. Then they defined the property *dependency-correctness*

which means that the provenance annotation produced by queries shows how the output is influenced if any input is changed.

They also addressed the weakness of the previous research on provenance on databases. The definitions of provenance were *elusive* so that they are ambiguous and hard to be formalised based on the semantic descriptions. In particular, many forms of provenance defined before are sensitive to query rewriting in the sense that equal queries may have different provenance behaviours. These approaches are difficult to extend beyond monotone relational queries. In a word, there is a lack of the formal definition and foundations for provenance.

2.4 Language-based and Information Flow Security

Language-based security is the line of research that uses programming language techniques, such as type systems, dynamic monitoring, or a combination of static and dynamic methods, to ensure the security properties of the programs such as confidentiality. Language-based techniques have also taken part into the line of research of information flows. These works are related to provenance since all of them try to characterise what happens in the execution so as to verify some security property or help to understand the errors after a failure.

Pottier and Conchon (2000) presented a framework that can extend an arbitrary type system with dependency information and how soundness and non-interference proofs for the new system may rely on. This work enriches any of the type system today with information flow analysis. They pointed out information flow analysis is nothing but a *dependency analysis*. They realised this by proposing a translation scheme which translated the programs written in a source labelled ML to a target calculus that is equipped with a type system. Then the type system of the source language was built by the translation and the type system of the target calculus. The non-interference property was stated based on the soundness properties of the type systems of the target and the source calculus. This revealed the type system contains useful dependency information.

Indeed, Abadi et al. (1999) found that many program analysis techniques, such as program slicing, call tracking, binding-time analysis, are actually dependency analysis. They presented a unifying dependency calculus, a small extension of Moggi's computational lambda calculus, called *Dependency Core Calculus*(DCC). Simple non-interference proofs can be derived for each case.

Dependency-provenance in databases is also a dynamic dependency tracking. However more detailed techniques in these lines of research are less comparable to that of provenance, so we will not present them here.

2.5 Synthesis-Based System Repair

Besides configuration management tools, there is a line of research employing synthesis-based techniques to facilitate system changes.

Weiss et al. (2017) addressed the problem that when changes need to make to system configurations, how to benefit from both the configuration tools and shell commands. Change requirements on system configurations happen often. Using configuration tools is not always the easiest way to realise changes since it is hard to find where and how to make changes in the configuration specification. In some cases, shell commands remain the simple way for administrators to apply the changes and detect the states of system immediately. This work proposed to combine the strength of both configuration languages and shell commands for system configuration updates. While configuration languages are used for machine states, shell commands can be employed for small changes. However changing the system by shell commands when a manifest is in use can cause *configuration shift*. To solve this, the authors realised this approach in a tool Tortoise that automatically synthesizes repairs to system configurations so as to change Puppet manifests consistent with the changes made by shell commands. Tortoise includes a imperative modelling language ΔP. It can translate system manifests and shell commands used by administrators to the models in this languages, then find a repair space for manifests.

Chugh et al. (2016) focused on the problem how to combine programmatic systems and direct manipulation. Either of these methods has their own strength. Direct manipulation provides visual feedback and enable rapid prototyping, whereas programmatic method allows reusing complex abstractions. This work proposed to integrate these two modes to use, a combination called *prodirect manipulation*. It was realised in the specific domain of Scalable Vector Graphics (SVG). The authors proposed the approach, a workflow called *live synchronization*, which means the system will synthesizes the updates to a program immediately when users make changes to the output on the interface. The authors realised this approach in the system Sketch-n-Sketch. A framework in it to infer program

updates is called *trace-based synthesis*. The programming language in Sketch-n-Sketch is a core, untyped functional language called `little`. All the data are labelled uniquely by their locations in the AST of the program. Trace-based synthesis instruments the execution of the programs so that it produces a trace for each output values. Traces are formed in the run-time. Then the program and output are related by value-trace equations that are used for inferring program updates. When the user manipulate a value on the interface,Sketch-n-Sketch updates the values at program locations in real-time.

Synthesis-Based system repair focuses on updating systems or programs automatically conforming to the changes caused in other ways by using language relevant techniques. It does not mean to provide information on the reason of errors for users so to fix them as provenance does.

Chapter 3

Semantics of μPuppet

Puppet is a popular declarative framework for specifying and managing complex system configurations. The Puppet framework includes a domain-specific language with several advanced features inspired by object-oriented programming, including user-defined resource types, 'classes' with a form of inheritance, and dependency management. Like most real-world languages, the language has evolved in an ad hoc fashion, resulting in a design with numerous features, some of which are complex, hard to understand, and difficult to use correctly.

We present an operational semantics for μPuppet, a representative subset of the Puppet language that covers the distinctive features of Puppet, while excluding features that are either deprecated or work-in-progress. Formalising the semantics sheds light on difficult parts of the language, identifies opportunities for future improvements, and provides a foundation for provenance tracking in the following chapters. Our semantics leads straightforwardly to a reference implementation in Haskell. We also discuss some of Puppet's idiosyncrasies, particularly its handling of classes and scope, and present an initial corpus of test cases supported by our formal semantics. The content of this chapter is based on the ECOOP paper (Fu et al., 2017).

3.1 Introduction

Managing a large-scale data centre consisting of hundreds or thousands of machines is a major challenge. Manual installation and configuration is simply impractical, given that each machine hosts numerous software components, such as databases, web servers, and middleware. Hand-coded configuration scripts are

difficult to manage and debug when multiple target configurations are needed. Moreover, misconfigurations can potentially affect millions of users. Recent empirical studies (Yin et al., 2011; Gunawi et al., 2014) attribute a significant proportion of system failures to misconfiguration rather than bugs in the software itself. Thus better support for specifying, debugging and verifying software configurations is essential to future improvements in reliability (Xu and Zhou, 2015).

A variety of *configuration frameworks* have been developed to increase the level of automation and reliability. They typically contain their own configuration languages. Like most real-world languages, configuration languages have largely evolved in an ad hoc fashion, with little attention paid to their semantics. Given their infrastructural significance, this makes them an important (although challenging) target for formal study: a formal model can clarify difficult or counterintuitive aspects of the language, identify opportunities for improvements and bug-fixes, and provide a foundation for static or dynamic analysis techniques, such as typechecking, provenance tracking and execution monitoring. In this chapter, we investigate the semantics of the configuration language used by the Puppet framework. Puppet is a natural choice because of its DSL-based approach, and the fact that it has seen widespread adoption. The 2018 PuppetConf conference was packed with over 80 sessions presented by more than 100 of the industries top leaders and had more than 30 sponsors from around the world.

An additional challenge for the formalisation of real-world languages is that they tend to be moving targets. For example, Puppet 4.0, released in March 2015, introduced several changes that are not backwards-compatible with Puppet 3, along with a number of non-trivial new features. In this chapter, we take Puppet 4.8 (the version included with Puppet Enterprise 2016.5) as the baseline version of the language, and define a subset called μPuppet that includes the established features of the language that appear most important and distinctive; in particular, it includes the constructs `node`, `class`, and `define`. These are used in almost all Puppet programs (called *manifests*). We chose to exclude some features that are either deprecated or not yet in widespread use, or whose formalisation would add complication without being particularly enlightening, such as regular expressions and string interpolation.

The main contributions of this chapter are:

1. a formalisation of μPuppet, a subset of Puppet 4.8;

2. a discussion of simple metatheoretic properties of μPuppet such as determinism, monotonicity and (non-)termination;

3. a reference implementation of μPuppet in Haskell;

4. a corpus of test cases accepted by our implementation;

5. a discussion of the more complex features not handled by μPuppet.

We first give an overview of the language via some examples (Section 3.2.1), covering some of the more counterintuitive and surprising parts of the language. Next we define the abstract syntax and a small-step operational semantics of μPuppet (Section 3.3). We believe ours to be the first formal semantics of a representative subset of Puppet; although recent work by Shambaugh et al. (2016) handles some features of Puppet, they focus on analysis of the "realisation" phase and do not present a semantics for the `node` or `class` constructs or for inheritance (although their implementation does handle some of these features). We use a small-step operational semantics (as opposed to large-step or denotational semantics) because it is better suited to modelling some of the idiosyncratic aspects of Puppet, particularly the sensitivity of scoping to evaluation order. We focus on unusual or novel aspects of the language in the main body of the chapter; the full set of rules are given in Appendix A.3. Section 3.4 discusses some properties of μPuppet. Section 3.5 describes our implementation and how we validated our rules against the actual behavior of Puppet, and discusses some of the omitted features. Sections 3.6 and 3.7 discuss related work and present our conclusions.

3.2 Puppet

3.2.1 Puppet: key concepts

We now introduce the basic concepts of the Puppet language – manifests, catalogs, resources, and classes – with reference to various examples. We also discuss some behaviours which may seem surprising or unintuitive; clarifying such issues is one reason for pursuing a formal definition of the language. The full Puppet 4.8 language has many more features than presented here. Appendix A.2 contains a complete list of features and the subset supported by μPuppet.

3.2.1.1 Manifests and catalogs

Figure 3.1 shows a typical manifest, consisting of a *node definition* and various *classes* declaring resources, which will be explained in § 3.2.1.4 below. Node definitions, such as the one starting on line 1, specify how a single machine or group of machines should be configured. Single machines can be specified by giving a single hostname, and groups of machines by giving a list of hostnames, a regular expression, or `default` (as in this example). The `default` node definition is used if no other definition applies.

In this case the only node definition is `default`, and so compiling this manifest for any node results in the catalog on the right of Figure 3.1. In this case the catalog is a set of resources of type `file` with titles `config1`, `config2` and `config3`, each with a collection of attribute-value pairs. Puppet supports several persistence formats for catalogs, including YAML; here we present the catalog using an abstract syntax which is essentially a sub-language of the language of manifests. The `file` resource type is one of Puppet's many built-in resource types, which include other common configuration management concepts such as `user`, `service` and `package`.

3.2.1.2 Resource declarations

Line 11 of the manifest in Figure 3.1 shows how the `config1` resource in the catalog was originally declared. The `path` attribute was specified explicitly as a string literal; the other attributes were given as variable references of the form $\$x$. Since a resource with a given title and type is global to the entire catalog, it may be declared only once during a given compilation. A *compilation error* results if a given resource is declared more than once. Note that what Puppet calls a "compilation error" is a purely dynamic condition, and so is really a runtime error in conventional terms.

The ordering of attributes within a resource is not significant; by default they appear in the catalog in the order in which they were declared. Optionally they can be sorted (by specifying ordering constraints) or randomised. Sorting is usually recommended over relying on declaration order (Rhett, 2016).

```
node default {
  $source = '/source'
  include service1
}

class service1 {
  $mode = '123'

  include service2

  file { 'config1':
    path => 'path1',
    source => $source,
    mode => $mode,
    checksum => $checksum,
    provider => $provider,
    recurse => $recurse
  }

  $checksum = md5
}

class service2 inherits service3 {
  $recurse = true

  file { 'config2':
    path => 'path2',
    source => $source,
    mode => $mode,
    checksum => $checksum,
    provider => $provider,
    recurse => $recurse
  }
}

class service3 {
  $provider = posix

  file { 'config3':
    path => 'path3',
    mode => $mode,
    checksum => $checksum,
    recurse => $recurse
  }
}
```

```
file { 'config3':
    path => 'path3'
}
file { 'config2':
    path => 'path2',
    source => '/source',
    provider => 'posix',
    recurse => true
}
file { 'config1':
    path => 'path1',
    source => '/source',
    mode => '123'
}
```

Figure 3.1: Example manifest (left); compiled catalog (right)

3.2.1.3 Variables and strict mode

Puppet lacks variable declarations in the usual sense; instead variables are implicitly declared when they are assigned to. A compilation error results if a given variable is assigned to more than once in the same scope. As we saw above, unqualified variables, whether being read or assigned to, are written in "scripting language" style $\$x$.

Puppet allows variables to be used before they are assigned, in which case their value is a special "undefined" value `undef`, analogous to Ruby's `nil` or JavaScript's `undefined`. By default, attributes only appear in the compiled output if their values are defined. Consider the variables `$mode` and `$checksum` introduced by the assignments at lines 7 and 20 in the manifest in Figure 3.1. The ordering of these variables relative to the file resource `config1` is significant, because it affects whether they are in scope. Since `$mode` is defined *before* `config1`, its value can be read and assigned to the attribute `mode`. In the compiled catalog, `mode` thus appears as an attribute of `config1`. On the other hand `$checksum` is assigned *after* `config1`, and is therefore undefined when read by the code which initialises the `checksum` attribute. Thus `checksum` is omitted from the compiled version of `config1`.

Since relying on the values of undefined variables is often considered poor practice, Puppet provides a *strict* mode which treats the use of undefined variables as an error. For similar reasons, and also to keep the formal model simple, μPuppet always operates in strict mode. We discuss the possibility of relaxing this in Section 3.5.2.

3.2.1.4 Classes and includes

Resource declarations may be grouped into *classes*. However, Puppet "classes" are quite different from the usual concept of classes in object-oriented programming – they define collections of resources which can be declared together by *including* the class. This is sometimes called *declaring* the class, although there is a subtle but important distinction between "declaring" and "including" which we will return to shortly.

In Figure 3.1, it is the inclusion into the node definition of class `service1` which explains the appearance of `config1` in the catalog, and in turn the inclusion into `service1` of class `service2` which explains the appearance of `config2`.

(The fact that `config3` also appears in the output relates to inheritance, and is discussed in §3.2.1.6 below.) Inclusion is idempotent: the same class may be included multiple times, but doing so only generates a single copy of the resources in the catalog. This allows a set of resources to be included into all locations in the manifest which depend on them, without causing errors due to duplicate declarations of the same resource.

To a first approximation, including a class into another class obeys a lexical scope discipline, meaning names in the including class are not visible in the included class. However inclusion into a node definition has a quite different behaviour: it introduces a containment relation between the node definition and the class, meaning that names scoped to the node definition are visible in the body of the included class. Thus in Figure 3.1, although the variable `$mode` defined in `service1` is not in scope inside the included class `service2` (as per lexical scoping), the `$source` variable defined in the node definition *is* in scope in `service1`, because `service1` is included into the node scope.

This is similar to the situation in Java where a class asserts its membership of a package using a package declaration, except here the node definition pulls *in* the classes it requires. The subtlety is that it is actually when a class is *declared* (included for the first time, dynamically speaking) that any names in the body of the class are resolved. If the *usage* of a class happens to change so that it ends up being declared in so-called *top* scope (the root namespace usually determine at check-in time), it may pick up a different set of bindings. Thus including a class, although idempotent, has a "side effect" – binding the names in the class – making Puppet programs potentially fragile. More of the details of scoping are given in the language reference manual (Puppet, 2016).

3.2.1.5 Qualified names

A definition which is not in scope can be accessed using a *qualified* name, using a syntax reminiscent of C++ and Java, with atomic names separated by the token `::`. For example, in Figure 3.2 above, `$::osfamily` refers to a variable in the top scope, while `$::ssh::params::sshd_package` is an absolute reference to the `$sshd_package` variable of class `ssh::params`.

Less conventionally, Puppet also allows the name of a class to be a qualified name, such as `ssh::params` in Figure 3.2. Despite the suggestive syntax, which resembles a C++ member declaration, this is mostly just a convention used to

indicate related classes. In particular, qualified names used in this way do not require any of the qualifying prefixes to denote an actual namespace. (Although see the discussion in Section 3.5.2 for an interaction between this feature and nested classes, which μPuppet does not support.)

3.2.1.6 Inheritance and class parameters

Classes may *inherit* from other classes; the inheriting class inherits the variables of the parent class, including their values. In the earlier example (Figure 3.1), `service2` inherits the value of `$provider` from `service3`. Including a derived class implicitly includes the inherited class, potentially causing the inherited class to be declared (in the Puppet sense of the word) when the derived class is declared:

> *When you declare a derived class whose base class hasn't already been declared, the base class is immediately declared in the current scope, and its parent assigned accordingly. This effectively "inserts" the base class between the derived class and the current scope. (If the base class has already been declared elsewhere, its existing parent scope is not changed.)*

This explains why `config3` appears in the compiled catalog for Figure 3.1.

Since the scope in which a class is eventually declared determines the meaning of the names in the class (§ 3.2.1.4 above), inheritance may have surprising (and non-local) consequences. At any rate, the use of inheritance for most use cases is now discouraged.[1] The main exception is the use of inheritance to specify default values; this is the scenario illustrated in Figure 3.2.

Line 1 of Figure 3.2 introduces class `ssh::params`, which assigns to variable `$sshd_package` a value conditional on the operating system name `$::osfamily` (line 2). The class `ssh` (line 8) inherits from `ssh::params`. It also defines a *class parameter* `$ssh_pkg` (before the `inherits` clause), and uses the value of the `$sshd_package` variable in the inherited class as the default value for the parameter. Because an inherited class is processed before a derived class, the value of `$sshd_package` is available at this point.

The value of the parameter `$ssh_pkg` is then used as the title of the `package` resource declared in the `ssh` class (line 9) specifying that the relevant software package exists in the target configuration. The last construct is a node definition specifying how to configure the machine with hostname `ssh.example.com`. If

[1] `https://docs.puppet.com/puppet/latest/style_guide.html`, section 11.1.

```
1  class ssh::params {
2    case $::osfamily {
3    'Debian': { $sshd_package  = 'ssh' }
4    'RedHat': { $sshd_package  = 'openssh-server' }
5    default:  { fail("SSH class not supported") }
6    }
7  }
8  class ssh ($ssh_pkg = $::ssh::params::sshd_package) inherits ssh::params {
9    package { $ssh_pkg:
10     ensure => installed
11   }
12 }
13 node 'ssh.example.com' {
14   include ssh
15 }
```

Figure 3.2: Example manifest showing recommended use of inheritance for setting default parameters

host `ssh.example.com` is a Debian machine, the result of compiling this manifest is a catalog containing the following `package` resource:

```
1  package { 'ssh' :  ensure => installed }
```

3.2.1.7 Class statements

Figure 3.3 defines a class `c` with three parameters. The `class` statement (line 31) can be used to include a class and provide values for (some of) the parameters. In the resulting catalog, the `from_class` resource has `backup` set to `true` (from the explicit argument), `mode` set to `123` (because no `mode` argument is specified), and `source` set to `'/default'` (because the `path` variable is undefined at the point where the class is declared (line 31)).

However, the potential for conflicting parameter values means that multiple declarations with parameters are not permitted, and the `class` statement must be used instead (which only allows a single declaration).

3.2.1.8 Defined resource types

Defined resource types are similarly to classes, but provide a more flexible way of introducing a user-defined set of resources. Definition `d` (line 14) in Figure 3.3 introduces a defined resource type. The definition looks very similar to a class definition, but the body is a macro which can be instantiated (line 36) multiple

```
1  class c (
2    $backupArg = false,
3    $pathArg = '/default',
4    $modeArg = '123' ) {
5
6    file { 'from_class':
7      backup => $backupArg,
8      source => $pathArg,
9      path => $path,
10     mode => $modeArg
11   }
12 }
13
14 define d (
15   $backupArg = false,
16   $pathArg = '/default',
17   $modeArg = '123' ) {
18
19   file { 'from_define':
20     backup => $backupArg,
21     source => $pathArg,
22     path => $path,
23     mode => $modeArg
24   }
25 }
26
27 node default {
28
29   $backup = true
30
31   class { c:
32     backupArg => $backup,
33     pathArg => $path
34   }
35
36   d { "service3":
37     backupArg => $backup,
38     pathArg => $path
39   }
40
41   $path = '/path'
42 }
```

```
1  file { 'from_class':
2      backup => true,
3      source => '/default',
4      mode => '123'
5  }
6  file { 'from_define':
7      path => '/path',
8      backup => true,
9      source => '/default',
10     mode => '123'
11 }
```

Figure 3.3: Manifest with class parameters and defined resource types (left); catalog (right)

times with different parameters.

Interestingly, the `path` attribute in the `from_class` file is undefined in the result, apparently because the assignment `$path = '/path'` follows the declara-

tion of the class — however, in the `from_define` file, `path` *is* defined as `'/path'`! The reason appears to be that defined resources are added to the catalog and re-processed after other manifest constructs.[2] We will not model this behaviour of the assignments as `$path = '/path'` since we want to model a monotonic sub-language. Monotonicity will be explained later in detail.

3.3 μPuppet

We now formalise μPuppet, a language which captures many of the essential features of Puppet. Our goal is not to model all of Puppet's idiosyncrasies, but instead to attempt to capture the 'declarative' core of Puppet, as a starting point for future study. As we discuss later, Puppet also contains several non-declarative features whose behaviour can be counterintuitive and surprising; their use tends to be discouraged in Puppet's documentation and by other authors.

3.3.1 Abstract syntax

The syntax of μPuppet manifests m is defined in Figure 4.3, including expressions e and statements s. Constant expressions in μPuppet can be integer literals i, string literals w, or boolean literals `true` or `false`. Other expressions include arithmetic and boolean operations, variable forms $\$x$, $\$::x$ and $\$::a::x$. Here, x stands for variable names and a stands for class names. *Selectors* e ? $\{M\}$ are conditional expressions that evaluate e and then conditionally evaluate the first matching branch in M. Arrays are written $[e_1, \ldots, e_n]$ and hashes (dictionaries) are written $\{k \Rightarrow e, \ldots\}$ where k is a key (either a constant number or string). Full Puppet includes many more built-in functions that we omit here.

Statements s include expressions e (whose value is discarded), composite statements $s_1 \llcorner s_2$, assignments $\$x = e$, and conditionals `unless`, `if`, `case`, which are mostly standard. (Full Puppet includes an `elsif` construct that we omit from μPuppet.) Statements also include resource declarations $t\ \{e : H\}$ for built-in resource types t, resource declarations $u\ \{e : H\}$ for defined resource types u, and class declarations `class` $\{a : H\}$ and `include` a.

Manifests m can be statements s; composite manifests $m_1 \llcorner m_2$, class definitions `class` $a\ \{s\}$ with or without parameters ρ and inheritance clauses `inherits` b;

[2] `http://puppet-on-the-edge.blogspot.co.uk/2014/04/getting-your-puppet-ducks-in-row.html`

Expression	e	::=	$i \mid w \mid \texttt{true} \mid \texttt{false} \mid \$x \mid \$::x \mid \$::a::x$
		|	$e_1 + e_2 \mid e_1 - e_2 \mid e_1 / e_2 \mid e_1 > e_2 \mid e_1 = e_2 \mid e_1 \,\texttt{and}\, e_2 \mid e_1 \,\texttt{or}\, e_2 \mid \texttt{!}e \mid \ldots$
		|	$\{H\} \mid [e_1, \ldots, e_n] \mid e_1[e_2] \mid e \;?\; \{M\}$
Array	A	::=	$\varepsilon \mid e, A$
Hash	H	::=	$\varepsilon \mid k \Rightarrow e, H$
Case	c	::=	$e \mid \texttt{default}$
Matches	M	::=	$\varepsilon \mid c \Rightarrow e, M$
Statement	s	::=	$e \mid s_1 \,␣\, s_2 \mid \$x = e \mid \texttt{unless}\ e\ \{s\} \mid \texttt{if}\ e\ \{s\}\ \texttt{else}\ \{s\} \mid \texttt{case}\ e\ \{C\} \mid D$
Cases	C	::=	$\varepsilon \mid c : \{s\} \,␣\, C$
Declaration	D	::=	$t\ \{e : H\} \mid u\ \{e : H\} \mid \texttt{class}\ \{a : H\} \mid \texttt{include}\ a$
Manifest	m	::=	$s \mid m_1 \,␣\, m_2 \mid \texttt{node}\ Q\ \{s\} \mid \texttt{define}\ u\ (\rho)\ \{s\} \mid \texttt{class}\ a\ \{s\} \mid \texttt{class}\ a\ (\rho)\ \{s\}$
		|	$\texttt{class}\ a\ \texttt{inherits}\ b\ \{s\} \mid \texttt{class}\ a\ (\rho)\ \texttt{inherits}\ b\ \{s\}$
Node spec	Q	::=	$N \mid \texttt{default} \mid (N_1, \ldots, N_k) \mid r \in \mathit{RegExp}$
Parameters	ρ	::=	$\varepsilon \mid x, \rho \mid x = e, \rho$

Figure 3.4: Abstract syntax of μPuppet

node definitions `node` Q $\{s\}$; or defined resource type definitions `define` u (ρ) $\{s\}$. Node specifications Q include literal node names N, `default`, lists of node names, and regular expressions r (which we do not model explicitly).

Sequences of statements, cases, or manifest items can be written by writing one statement after the other, separated by whitespace, and we write ␣ when necessary to emphasise that this whitespace is significant. The symbol ε denotes the empty string.

3.3.2 Operational Semantics

We now define a small-step operational semantics for μPuppet. This is a considered choice: although Puppet is advertised as a declarative language, it is not *a priori* clear that manifest compilation is a terminating or even deterministic process. Using small-step semantics allows us to translate the (often) procedural descriptions of Puppet's constructs directly from the documentation.

The operational semantics relies on auxiliary notions of catalogs v_{C}, scopes α, variable environments σ, and definition environments κ explained in more detail below. We employ three main judgements, for processing expressions, statements,

and manifests:

$$\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e' \qquad \sigma, \kappa, v_{\mathrm{C}}, s \xrightarrow{\alpha}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, s' \qquad \sigma, \kappa, v_{\mathrm{C}}, m \xrightarrow{N}_{\mathsf{m}} \sigma', \kappa', v'_{\mathrm{C}}, m'$$

Here, σ, κ, and v_{C} are the variable environment, definition environment, and catalog beforehand, and their primed versions are the corresponding components after one compilation step. The parameter α for expressions and statements represents the ambient scope; the parameter N for manifests is the target node name.

The main judgement is $\rightarrow_{\mathsf{m}}$, which takes a μPuppet manifest m and a node name N and compiles it to a catalog v_{C}, that is, a list of resource values v_{R} for that node. Given initial variable environments σ (representing data provided by the client) and κ (containing any predefined classes or resource definitions), execution of manifest m begins with an empty catalog and terminates with catalog v_{C} when the manifest equals `skip`, i.e. $\sigma, \kappa, \varepsilon, m \xrightarrow{N}_{\mathsf{m}} \cdots \xrightarrow{N}_{\mathsf{m}} \sigma', \kappa', v_{\mathrm{C}}, \texttt{skip}$.

3.3.2.1 Auxiliary definitions: catalogs, scopes and environments

Before defining compilation formally, we first define catalogs (§3.3.2.1.1), the result of compiling manifests; scopes (§3.3.2.1.2), which explicitly represent the ambient scope used to resolve unqualified variable references; variable environments (§3.3.2.1.3), which store variable bindings; and definition environments (§3.3.2.1.4), which store class and resource definitions.

3.3.2.1.1 Catalogs The syntax of catalogs is given in Figure 3.5. A *catalog* v_{C} is a sequence of resource values, separated by whitespace; a *resource value* $v_{\mathrm{R}} = t\,\{w : v_{\mathrm{H}}\}$ is a resource whose title is a string value and whose content is a hash value; a hash value v_{H} is an attribute-value sequence in which all expressions are values; and finally a *value* v is either an integer literal i, string literal w, boolean literal `true` or `false`, hash $\{v_{\mathrm{H}}\}$, or array $[v_1, \ldots, v_n]$. In a well-formed catalog, there is at most one resource with a given type and title; attempting to add a resource with the same type and title as one already in the catalog is an error.

3.3.2.1.2 Scopes As discussed in Section 3.2.1, Puppet variables can be assigned in one scope and referenced in a different scope. For example, in Figure 3.2, the parent scope of class scope `ssh` is class scope `ssh::params`. To model this,

Catalog	v_{C}	::=	$\varepsilon \mid v_{\mathrm{R}} \multimap v_{\mathrm{C}}$
Value	v	::=	$i \mid w \mid \texttt{true} \mid \texttt{false} \mid \{v_{\mathrm{H}}\} \mid [v_1, \ldots, v_n] \mid t[v]$
Hash value	v_{H}	::=	$\varepsilon \mid k \Rightarrow v, v_{\mathrm{H}}$
Resource value	v_{R}	::=	$t\,\{w : v_{\mathrm{H}}\}$
Scope	α	::=	$:: \mid ::a \mid ::\texttt{nd} \mid \alpha\,\texttt{def}$
Statement	s	::=	$\ldots \mid \texttt{scope}\,\alpha\,s \mid \texttt{skip}$

Figure 3.5: Auxiliary constructs: catalogs and scopes

we model scopes and parent-child relations between scopes explicitly. Scope :: represents the top scope, $::a$ is the scope of class a, ::nd is the active node scope, and $\alpha\,\texttt{def}$ is the scope of a resource definition that is executed in ambient scope α.

The scope for defined resources takes another scope parameter α in order to model resource definitions that call other resource definitions. The top-level, class, and node scopes are persistent, while $\alpha\,\texttt{def}$ is cleared at the end of the corresponding resource definition; thus these scopes can be thought of as names for stack frames. The special statement form $\texttt{scope}\ \alpha\ s$ is used internally in the semantics to model scope changes. An additional internal statement form skip, unrelated to scopes, represents the empty statement. Neither of these forms are allowed in Puppet manifests.

As discussed earlier, there is an ancestry relation on scopes, which governs the order in which scopes are checked when dereferencing an unqualified variable reference. We use mutually recursive auxiliary judgments $\alpha\,\mathsf{parentof}_\kappa\,\beta$ to indicate that α is the parent scope of β in the context of κ and $\alpha\,\mathsf{baseof}_\kappa\,\beta$ to indicate that α is the *base* scope of β. The base scope is either ::, indicating that the scope is the top scope, or ::nd, indicating that the scope is being processed inside a node definition. We first discuss the rules for $\mathsf{parentof}_\kappa$:

$$\frac{}{:: \,\mathsf{parentof}_\kappa\, ::\texttt{nd}}\ \textsc{PNode} \qquad \frac{\beta\,\mathsf{baseof}_\kappa\,\alpha\,\texttt{def}}{\beta\,\mathsf{parentof}_\kappa\,\alpha\,\texttt{def}}\ \textsc{PDefRes}$$

$$\frac{\kappa(a) = \mathsf{DeclaredClass}(\alpha)}{\alpha\,\mathsf{parentof}_\kappa\,::a}\ \textsc{PClass}$$

The PNode rules says that the top-level scope is the parent scope of node scope.

The PDefRes rule says that the parent scope of the defined resource type scope is its base scope. Thus, a resource definition being declared in the toplevel will have parent ::, while one being declared inside a node definition will have parent scope ::nd. The PClass rule defines the scope of the (declared) parent class b to be the scope α that is recorded in the $\mathsf{DeclaredClass}(\alpha)$ entry. The rules for class inclusion and declaration in the next section show how the $\mathsf{DeclaredClass}(\alpha)$ entry is initialised; this also uses the baseof_κ relation. The rules defining baseof_κ are as follows:

$$\frac{}{:: \mathsf{baseof}_\kappa ::}\ \textsc{BTop} \qquad \frac{}{::\mathtt{nd}\,\mathsf{baseof}_\kappa ::\mathtt{nd}}\ \textsc{BNode} \qquad \frac{\alpha\,\mathsf{baseof}_\kappa\,\beta}{\alpha\,\mathsf{baseof}_\kappa\,\beta\ \mathtt{def}}\ \textsc{BDefRes}$$

$$\frac{\kappa(a) = \mathsf{DeclaredClass}(\beta) \qquad \alpha\,\mathsf{baseof}_\kappa\,\beta}{\alpha\,\mathsf{baseof}_\kappa ::a}\ \textsc{BClass}$$

These rules determine whether the ambient scope α in which the class is declared is *inside* or *outside* a node declaration. The base scope of toplevel or node scope is toplevel or node scope respectively. The base scope of $\beta\,\mathtt{def}$ is the base scope of β, while the base scope of a class scope $::a$ is the base scope of its parent scope as defined in the definition environment κ. (We will only try to obtain the base scope of a class that has already been declared.)

3.3.2.1.3 Variable environments During the compilation of a manifest, the values of variables are recorded in *variable environments* σ which are then accessed when these variables are referenced in the manifest. (We call these *variable* environments, rather than plain environments, since "environment" has a specific technical meaning in Puppet; see the glossary in Appendix A.1.)

Formally, a variable environment is defined as a partial function $\sigma : \mathit{Scope} \times \mathit{Var} \to \mathit{Value}$ which maps pairs of scopes and variables to values. The scope component indicates the scope in which the variable was assigned. We sometimes write $\sigma_\alpha(x)$ for $\sigma(\alpha, x)$. Updating a variable environment σ with new binding (α, x) to v is written $\sigma[(\alpha, x) : v]$, and clearing an environment (removing all bindings in scope α) is written $\mathit{clear}(\sigma, \alpha)$.

3.3.2.1.4 Definition environments Some components in Puppet, like classes and defined resource types, introduce *definitions* which can be declared elsewhere. To model this, we record such definitions in *definition environments* κ. Formally,

a definition environment is a partial function $\kappa : \mathit{Identifier} \to \mathit{Definition}$ mapping each identifier to a definition D. Evaluation of the definition only begins when a resource is declared which uses that definition.

Definitions are of the following forms:

$$\begin{array}{rcl} D & ::= & \mathsf{ClassDef}(c_{opt}, \rho, s) \mid \mathsf{DeclaredClass}(\alpha) \mid \mathsf{ResourceDef}(\rho, s) \\ c_{opt} & ::= & c \mid \bot \end{array}$$

The definition form $\mathsf{ClassDef}(c_{opt}, \rho, s)$ represents the definition of a class (before it has been declared); c_{opt} is the optional name of the class's parent, ρ is the list of parameters of the class (with optional default values), and s is the body of the class. The definition form $\mathsf{DeclaredClass}(\alpha)$ represents a class that has been declared; α is the class's *parent scope* and ρ and s are no longer needed. In Puppet, a class can be declared only once, and when it is declared its definition environment entry is changed to $\mathsf{DeclaredClass}(c_{opt})$. Finally, the definition form $\mathsf{ResourceDef}(\rho, s)$ represents the definition of a new *resource type*, where ρ and s are as above.

3.3.2.2 Expression evaluation

Expressions are the basic computational components of μPuppet. The rules for expression forms such as primitive operations are standard. The rules for selector expressions are also straightforward. Since variable accessibility depends on scope, the variable evaluation rules are a little more involved:

$$\frac{x \in dom(\sigma_\alpha)}{\sigma, \kappa, v_C, \$x \xrightarrow{\alpha} \sigma_\alpha(x)}\ \textsc{LVar}$$

$$\frac{x \notin dom(\sigma_\alpha) \qquad \sigma, \kappa, v_C, \$x \xrightarrow{\beta} v \qquad \beta\ \mathsf{parentof}_\kappa\ \alpha}{\sigma, \kappa, v_C, \$x \xrightarrow{\alpha} v}\ \textsc{PVar}$$

$$\frac{x \in dom(\sigma_{::})}{\sigma, \kappa, v_C, \${::}x \xrightarrow{\alpha} \sigma_{::}(x)}\ \textsc{TVar} \qquad \frac{x \in dom(\sigma_{::a})}{\sigma, \kappa, v_C, \${::}a :: x \xrightarrow{\alpha} \sigma_{::a}(x)}\ \textsc{QVar}$$

The LVar rule looks up the value of an unqualified variable in the current scope, if present. The PVar rule handles the case of an unqualified variable that is not defined in the current scope; its value is the value of the variable in the parent scope. The TVar and QVar rules look up fully-qualified variables in top

scope or class scope, respectively. (There is no qualified syntax for referencing variables in node scope from other scopes.)

μPuppet also includes array and hash expressions. An array is a list of expressions in brackets and a hash is a list of keys and their expression assignments in braces. When the expressions are values, an array or a hash is also a value. Each expression in them can be dereferenced by the array or hash followed by its index or key in brackets. The rules for constructing and evaluating arrays and hashes are straightforward, and included in Appendix A.

Resource references of the form $t[v]$ are allowed as values, where t is a built-in resource name and v is a (string) value. Such references can be used as parameters in other resources and to express ordering relationships between resources. Resource references can be used to extend resources or override inherited resource parameters; we do not model this behaviour. A resource reference can also (as of Puppet 4) be used to access the values of the resource's parameters. This is supported in μPuppet as shown in the following example.

```
1  file {"foo.txt":
2    owner => "alice"
3  }
4  $y = "foo.txt"
5  $x = File[$y]
6  file {"bar.txt":
7    owner => $x["owner"]
8  }
```

In this example, we first declare a file resource, with an `owner` parameter `"alice"`, then we assign y the filename and $\$x$ a resource reference (value) `File["foo.txt"]`. Then in defining a second file resource we refer to the `"owner"` parameter of the already-declared file resource via the reference `File["foo.txt"]`. This declaration results in a second file resource with the same owner as the first.

The rules for dereferencing arrays, hashes, and resource references are as fol-

lows:

$$\frac{\sigma, \kappa, v_C, d \xrightarrow{\alpha} d'}{\sigma, \kappa, v_C, d[e] \xrightarrow{\alpha} d'[e]}\ \textsc{DeRefExp} \qquad \frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, v[e] \xrightarrow{\alpha} v[e']}\ \textsc{DeRefIndex}$$

$$\frac{}{\sigma, \kappa, v_C, [v_0, \ldots, v_n, \ldots, v_m][n] \xrightarrow{\alpha} v_n}\ \textsc{DeRefArray}$$

$$\frac{k = k_n}{\sigma, \kappa, v_C, \{k_1 = v_1, \ldots, k_n = v_n, \ldots, k_m = v_m\}[k] \xrightarrow{\alpha} v_n}\ \textsc{DeRefHash}$$

$$\frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, t[e] \xrightarrow{\alpha} t[e']}\ \textsc{RefRes} \qquad \frac{\mathit{lookup}_C(v_C, t, w, k) = v}{\sigma, \kappa, v_C, t[w][k] \xrightarrow{\alpha} v}\ \textsc{DeRefRes}$$

In the rule DeRefExp the expression e is evaluated to an array or a hash value. The rule DeRefIndex evaluates the index inside the brackets to a value. Rule DeRefArray accesses the value in an array at the index n while rule DeRefHash accesses the hash value by searching its key k. There could be a sequence of reference indexes in a reference. As we can see, such reference is evaluated in the left-to-right order of the index list. Rule ResRef evaluates the index and in the DeRefRec rule, the function lookup_C looks up the catalog for the value of the attribute k of the resource $t[v]$.

3.3.2.3 Statement evaluation

As with expressions, some of the statement forms, such as sequential composition, conditionals (`if`, `unless`), and `case` statements have a conventional operational semantics, shown in the appendix. An expression can occur as a statement; its value is ignored. Assignments, like variable references, are a little more complex. When storing the value of a variable in an assignment in σ, the compilation rule binds the value to x in the scope α:

$$\frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, \$x = e \xrightarrow{\alpha}_s \sigma, \kappa, v_C, \$x = e'}\ \textsc{AssignStep}$$

$$\frac{x \notin \mathit{dom}(\sigma_\alpha)}{\sigma, \kappa, v_C, \$x = v \xrightarrow{\alpha}_s \sigma[(\alpha, x) : v], \kappa, v_C, \texttt{skip}}\ \textsc{Assign}$$

Notice that Puppet does not allow assignment into any other scopes, only the current scope α.

We now consider $\texttt{scope}\ \alpha\ s$ statements, which are internal constructs (not part of the Puppet source language) we have introduced to track the scope that is in effect in different parts of the manifest during execution. The following rules handle compilation inside `scope` statements and cleanup when execution inside such a statement finally terminates.

$$\frac{\alpha \in \{::, ::a, ::\mathtt{nd}\} \qquad \sigma, \kappa, v_\mathrm{C}, s \xrightarrow{\alpha}_\mathrm{s} \sigma', \kappa', v'_\mathrm{C}, s'}{\sigma, \kappa, v_\mathrm{C}, \texttt{scope}\ \alpha\ s \xrightarrow{\alpha'}_\mathsf{s} \sigma', \kappa', v'_\mathrm{C}, \texttt{scope}\ \alpha\ s'}\ \textsc{ScopeStep}$$

$$\frac{\sigma, \kappa, v_\mathrm{C}, s \xrightarrow{\alpha\,\mathtt{def}}_\mathsf{s} \sigma', \kappa', v'_\mathrm{C}, s'}{\sigma, \kappa, v_\mathrm{C}, \texttt{scope}\ (\alpha\ \mathtt{def})\ s \xrightarrow{\alpha}_\mathsf{s} \sigma', \kappa', v'_\mathrm{C}, \texttt{scope}\ (\alpha\ \mathtt{def})\ s'}\ \textsc{DefScopeStep}$$

$$\frac{\alpha \in \{::, ::a, ::\mathtt{nd}\}}{\sigma, \kappa, v_\mathrm{C}, \texttt{scope}\ \alpha\ \texttt{skip} \xrightarrow{\beta}_\mathsf{s} \sigma, \kappa, v_\mathrm{C}, \texttt{skip}}\ \textsc{ScopeDone}$$

$$\frac{}{\sigma, \kappa, v_\mathrm{C}, \texttt{scope}\ (\alpha\ \mathtt{def})\ \texttt{skip} \xrightarrow{\alpha}_\mathsf{s} \mathit{clear}(\sigma, \alpha\ \mathtt{def}), \kappa, v_\mathrm{C}, \texttt{skip}}\ \textsc{DefScopeDone}$$

The ScopeStep and DefScopeStep rules handle compilation inside a scope; in the ScopeStep rule the ambient scope α' is overridden and the scope parameter α is used instead; similarly in the DefScopeStep rule the local scope $\alpha\ \mathtt{def}$ of s replaces the scope α. The ScopeDone rule handles the end of compilation inside a "persistent" scope, such as top-level, node or class scope, whose variables persist throughout execution, and the DefScopeDone rule handles the temporary scope of defined resources, whose locally-defined variables and parameters become unbound at the end of the definition. (In contrast, variables defined in toplevel, node, or class scopes remain visible throughout compilation.)

Resource declarations are compiled in a straightforward way; the title expression is evaluated, then all the expressions in attribute-value pairs in the hash component are evaluated. Once a resource is fully evaluated, it is appended to the catalog:

$$\frac{\sigma, \kappa, v_\mathrm{C}, e : H \xrightarrow{\alpha}_\mathsf{R} e' : H'}{\sigma, \kappa, v_\mathrm{C}, t\ \{e : H\} \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa, v_\mathrm{C}, t\ \{e' : H'\}}\ \textsc{ResStep}$$

$$\frac{}{\sigma, \kappa, v_\mathrm{C}, v_\mathrm{R} \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa, v_\mathrm{C} \llcorner v_\mathrm{R}, \texttt{skip}}\ \textsc{ResDecl}$$

Defined resource declarations look much like built-in resources:

```
1 apache::vhost {'homepages':
2   port    => 8081,
3   docroot => '/var/www-testhost',
4 }
```

When a defined resource type declaration is fully evaluated, it is expanded (much like a function call).

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, \{e : H\} \xrightarrow{\alpha}_{\mathsf{R}} \{e' : H'\}}{\sigma, \kappa, v_{\mathrm{C}}, u\ \{e : H\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, u\ \{e' : H'\}}\ \textsc{DefStep}$$

$$\frac{\kappa(u) = \mathsf{ResourceDef}(\rho, s) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}})}{\sigma, \kappa, v_{\mathrm{C}}, u\ \{w : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{scope}\ (\alpha\ \texttt{def})\ \{\$\mathit{title} = w \llcorner s' \llcorner s\}}\ \textsc{Def}$$

The *merge* function returns a statement s' assigning the parameters to their default values in ρ or overridden values from v_{H}. Notice that we also add the special parameter binding $\$\mathit{title} = w$; this is because in Puppet, the title of a defined resource is made available in the body of the resource using the parameter $\$\mathit{title}$. The body of the resource definition s is processed in scope α `def`. Class declarations take two forms: *include-like* and *resource-like declarations*.

The statement `include` a is an include-like declaration of a class a. (Puppet includes some additional include-like declaration forms such as `contain` and `require`). Intuitively, this means that the class body is processed (declaring any ancestors and resources inside the class), and the class is marked as declared; a class can be declared at most once. The simplest case is when a class has no parent, covered by the first two rules below:

$$\frac{\kappa(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = \mathit{merge}(\rho, \varepsilon) \qquad \beta\ \mathsf{baseof}_{\kappa}\ \alpha}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa[a : \mathsf{DeclaredClass}(\beta)], v_{\mathrm{C}}, \texttt{scope}\ (::a)\ s' \llcorner s}\ \textsc{IncU}$$

$$\frac{\kappa(a) = \mathsf{DeclaredClass}(\beta)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{skip}}\ \textsc{IncD}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{include}\ b\ \texttt{include}\ a}\ \textsc{IncPU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{DeclaredClass}(\beta) \qquad s' = \mathit{merge}(\rho, \varepsilon)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa[a : \mathsf{DeclaredClass}(::b)], v_{\mathrm{C}}, \texttt{scope}\ (::a)\ \{s' \llcorner s\}}\ \textsc{IncPD}$$

In the IncU rule, the class has not been declared yet, so we look up its body and default parameters and process the body in the appropriate scope. (We use the *merge* function again here to obtain a statement initialising all parameters which have default values.) In addition, we modify the class's entry in κ to $\mathsf{DeclaredClass}(\beta)$, where $\beta\,\mathsf{baseof}_\kappa\,\alpha$. As described in Section 3.2.1, this aspect of Puppet scoping is dynamic: if a base class is defined outside a node definition then its parent scope is ::, whereas if it is declared during the processing of a node definition then its parent scope is ::nd. (As discussed below, if a class inherits from another, however, the parent scope is the scope of the parent class no matter what). If this sounds confusing, this is because it is; this is the trickiest aspect of Puppet scope that is correctly handled by μPuppet. This complexity appears to be one reason that the use of node-scoped variables is discouraged by some experts (Rhett, 2016).

In the IncD rule, the class a is already declared, so no action needs to be taken. In the IncPU rule, we include the parent class so that it (and any ancestors) will be processed first. If there is an inheritance cycle, this process loops; we have confirmed experimentally that Puppet does not check for such cycles and instead fails with a stack overflow. In the IncPD rule, the parent class is already declared, so we proceed just as in the case where there is no parent class.

The rules for *resource-like class declarations* are similar:

$$\frac{\kappa(a) = \mathsf{ClassDef}(c_{opt}, \rho, S) \qquad \sigma, \kappa, v_\mathrm{C}, H \xrightarrow{\alpha}_\mathsf{H} H'}{\sigma, \kappa, v_\mathrm{C}, \texttt{class}\ \{a : H\} \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa, v_\mathrm{C}, \texttt{class}\ \{a : H'\}}\ \textsc{CDecStep}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = \mathit{merge}(\rho, v_\mathrm{H}) \qquad \beta\,\mathsf{baseof}_\kappa\,\alpha}{\sigma, \kappa, v_\mathrm{C}, \texttt{class}\ \{a : v_\mathrm{H}\} \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa[a : \mathsf{DeclaredClass}(\beta)], v_\mathrm{C}, \texttt{scope}\ (::a)\ s' \llcorner s}\ \textsc{CDecU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\sigma, \kappa, v_\mathrm{C}, \texttt{class}\ \{a : v_\mathrm{H}\} \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa, v_\mathrm{C}, \texttt{include}\ b\ \texttt{class}\ \{a : v_\mathrm{H}\}}\ \textsc{CDecPU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{DeclaredClass}(\beta) \qquad s' = \mathit{merge}(\rho, v_\mathrm{H})}{\sigma, \kappa, v_\mathrm{C}, \texttt{class}\ \{a : v_\mathrm{H}\} \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa[a : \mathsf{DeclaredClass}(::b)], v_\mathrm{C}, \texttt{scope}\ (::a)\ \{s' \llcorner s\}}\ \textsc{CDecPD}$$

There are two differences. First, because resource-like class declarations provide parameters, the rule CDecStep provides for evaluation of these parameters. Second, there is no rule analogous to IncD that ignores re-declaration of an already-declared class. Instead, this is an error. (As with multiple definitions of

variables and other constructs, however, we do not explicitly model errors in our rules.)

3.3.2.4 Manifest compilation

At the top level, manifests can contain statements, node definitions, resource type definitions, and class definitions. To compile statements at the top level, we use the following rule:

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, s \xrightarrow{::}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'}{\sigma, \kappa, v_{\mathrm{C}}, s \xrightarrow{N}_{\mathsf{m}} \sigma', \kappa', v'_{\mathrm{C}}, s'} \textsc{TopScope}$$

The main point of interest here is that we change from the manifest judgement (with the node name parameter N) to the statement judgement (with toplevel scope parameter ::). The node name parameter is not needed for processing statements, and we (initially) process statements in the toplevel scope. Of course, the statement s may well itself be a `scope` statement which immediately changes the scope.

A manifest in Puppet can configure all the machines (nodes) in a system. A node definition describes the configuration of one node (or type of nodes) in the system. The node declaration includes a specifier Q used to match against the node's hostname. We abstract this matching process as a function $\mathsf{nodeMatch}(N, Q)$ that checks if the name N of the requesting computer matches the specifier Q. If so (NodeMatch) we will compile the statement body of N. Otherwise (NodeNoMatch) we will skip this definition and process the rest of the manifest.

$$\frac{\mathsf{nodeMatch}(N, Q)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{node}\ Q\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{scope}\ (::\texttt{nd})\ s} \textsc{NodeMatch}$$

$$\frac{\neg\mathsf{nodeMatch}(N, Q)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{node}\ Q\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{skip}} \textsc{NodeNoMatch}$$

Resource type definitions in Puppet are used to design new, high-level resource types, possibly by declaring other built-in resource types, defined resource types, or classes. Such a definition includes Puppet code to be executed when a resource of the defined type is declared. Defined resource types can be declared multiple times with different parameters, so resource type definitions are loosely analogous to procedure calls. The following is an example of a defined resource type:

```
1 define apache::vhost (Integer $port) {
2   include apache
3   file { "host":
4     content => template('apache/vhost-default.conf.erb'),
5     owner   => 'www'
6   }
7 }
```

The compilation rule for defining a defined resource type is:

$$\frac{u \notin dom(\kappa)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{define}\ u\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[u : \mathsf{ResourceDef}(\rho, s)], v_{\mathrm{C}}, \texttt{skip}}\ \textsc{RDef}$$

The definition environment is updated to map u to $\mathsf{ResourceDef}(\rho, s)$ containing the parameters and statements in the definition of u. The manifest then becomes `skip`.

A class definition is used for specifying a particular service that could include a set of resources and other statements. Classes are defined at the top level and are declared as part of statements, as described earlier. Classes can be parameterised; the parameters are passed in at declaration time using the resource-like declaration syntax. The parameters can be referenced as variables in the class body. A class can also inherit directly from one other class. The following rules handle the four possible cases:

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ a\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[a : \mathsf{ClassDef}(\bot, \varepsilon, s)], v_{\mathrm{C}}, \texttt{skip}}\ \textsc{CDef}$$

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ a\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[a : \mathsf{ClassDef}(b, \varepsilon, s)], v_{\mathrm{C}}, \texttt{skip}}\ \textsc{CDefI}$$

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ a\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[a : \mathsf{ClassDef}(\bot, \rho, s)], v_{\mathrm{C}}, \texttt{skip}}\ \textsc{CDefP}$$

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ a\ (\rho)\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[a : \mathsf{ClassDef}(b, \rho, s)], v_{\mathrm{C}}, \texttt{skip}}\ \textsc{CDefPI}$$

In the simplest case (CDef) we add the class definition to the definition environment with no parent and no parameters. The other three rules handle the cases with inheritance, with parameters, or with both.

3.4 Metatheory

Because Puppet has not been designed with formal properties in mind, there is relatively little we can say formally about the "correctness" of μPuppet. Instead, the main measure of correctness is the degree to which μPuppet agrees with the behaviour of the main Puppet implementation, which is the topic of the next section. Here, we summarise two properties of μPuppet that guided our design of the rules. First, evaluation is deterministic: a given manifest can evaluate in at most one way.

Theorem 3.4.1 (Determinism)**.** All of the evaluation relations of μPuppet are deterministic:

- If $\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e'$ and $\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e''$ then $e' = e''$.
- If $\sigma, \kappa, v_{\mathrm{C}}, s \xrightarrow{\alpha}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'$ and $\sigma, \kappa, v_{\mathrm{C}}, s \xrightarrow{\alpha}_{\mathsf{s}} \sigma'', \kappa'', v''_{\mathrm{C}}, s''$ then $\sigma' = \sigma''$, $\kappa' = \kappa''$, $v'_{\mathrm{C}} = v''_{\mathrm{C}}$ and $s' = s''$.
- If $\sigma, \kappa, v_{\mathrm{C}}, m \xrightarrow{N}_{\mathsf{m}} \sigma', \kappa', v'_{\mathrm{C}}, m'$ and $\sigma, \kappa, v_{\mathrm{C}}, m \xrightarrow{N}_{\mathsf{m}} \sigma'', \kappa'', v''_{\mathrm{C}}, m''$ then $\sigma' = \sigma''$, $\kappa' = \kappa''$, $v'_{\mathrm{C}} = v''_{\mathrm{C}}$ and $m' = m''$.

Proof. Straightforward by induction on derivations. □

Second, in μPuppet, evaluation is monotonic in the sense that:

- Once a variable binding is defined in σ, its value never changes, and it remains bound until the end of the scope in which it was bound.
- Once a class or resource definition is defined in κ, its definition never changes, except that a class's definition may change from $\mathsf{ClassDef}(c_{opt}, \rho, s)$ to $\mathsf{DeclaredClass}(\beta)$.
- Once a resource is declared in v_{C}, its title, properties and values never change.

We can formalise this as follows.

Definition 3.4.1. We define orderings $\sqsubseteq$ on variable environments, definition environments and catalogs as follows:

- $\sigma \sqsubseteq \sigma'$ when $x \in dom(\sigma_\alpha)$ implies that either $\sigma_\alpha(x) = \sigma'_\alpha(x)$ or $\alpha = \beta\ \mathtt{def}$ for some β and $x \notin dom(\sigma'_\alpha)$.

- $\kappa \sqsubseteq \kappa'$ when $a \in dom(\kappa)$ implies either $\kappa(a) = \kappa'(a)$ or $\kappa(a) = \mathsf{ClassDef}(c_{opt}, \rho, s)$ and $\kappa'(a) = \mathsf{DeclaredClass}(\beta)$.

- $v_{\mathrm{C}} \sqsubseteq v'_{\mathrm{C}}$ when there exists v''_{C} such that $v_{\mathrm{C}} \sqcup v''_{\mathrm{C}} = v'_{\mathrm{C}}$.

- $(\sigma, \kappa, v_{\mathrm{C}}) \sqsubseteq (\sigma', \kappa', v'_{\mathrm{C}})$ when $\sigma \sqsubseteq \sigma'$, $\kappa \sqsubseteq \kappa'$ and $v_{\mathrm{C}} \sqsubseteq v'_{\mathrm{C}}$.

Theorem 3.4.2 (Monotonicity)**.** The statement and manifest evaluation relations of μPuppet are monotonic in $\sigma, \kappa, v_{\mathrm{C}}$:

- If $\sigma, \kappa, v_{\mathrm{C}}, s \xrightarrow{\alpha}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'$ then $(\sigma, \kappa, v_{\mathrm{C}}) \sqsubseteq (\sigma', \kappa', v'_{\mathrm{C}})$.

- If $\sigma, \kappa, v_{\mathrm{C}}, m \xrightarrow{N}_{\mathsf{m}} \sigma', \kappa', v'_{\mathrm{C}}, m'$ then $(\sigma, \kappa, v_{\mathrm{C}}) \sqsubseteq (\sigma', \kappa', v'_{\mathrm{C}})$.

Proof. Straightforward by induction. The only interesting cases are the rules in which σ, κ or v_{C} change; in each case the conclusion is immediate. □

These properties are not especially surprising or difficult to prove; nevertheless, they provide some justification for calling μPuppet a 'declarative' language. However, μPuppet does not satisfy some other desirable properties. For example, as we have seen, the order in which variable definitions or resource or class declarations appear can affect the final result. Likewise, there is no notion of 'well-formedness' that guarantees that a μPuppet program terminates successfully: compilation may diverge or encounter a run-time error. Furthermore, full Puppet does not satisfy monotonicity, because of other non-declarative features that we have chosen not to model. Further work is needed to identify and prove desirable properties of the full Puppet language, and identify subsets of (or modifications to) Puppet that make such properties valid.

3.5 Implementation and Evaluation

We implemented a prototype parser and evaluator for μPuppet in Haskell (GHC 8.0.1). The parser accepts source syntax for features of μPuppet as described in the Puppet documentation and produces abstract syntax trees as described in Section 3.3.2. The evaluator implements μPuppet compilation based on the rules shown in Appendix A.3. The implementation constitutes roughly 1300 lines of Haskell code. The evaluator itself is roughly 400 lines of code, most of which are line-by-line translations of the inference rules.

We also implemented a test framework that creates an Ubuntu 16.04.1 (x86_64) virtual machine with Puppet installed, and scripts which run each example using both μPuppet and Puppet and compare the resulting messages and catalog output.

3.5.1 Test cases and results

During our early investigations with Puppet, we constructed a test set of 52 manifests that illustrate Puppet's more unusual features, including resources, classes, inheritance, and resource type definitions. The tests include successful examples (where Puppet produces a catalog) and failing examples (where Puppet fails with an error); we found both kinds of tests valuable for understanding what is possible in cases where the documentation was unspecific.

We used these test cases to guide the design of μPuppet, and developed 16 additional test cases along the way to test corner cases or clarify behavior that our rules did not originally capture correctly. We developed further tests during debugging and to check the behavior of Puppet's (relatively) standard features, such as conditionals and case statements, arrays, and hashes. We did not encounter any surprises there so we do not present these results in detail.

We summarise the test cases and their results in Table 3.1. The "Feature" column describes the classification of features present in our test set. The "#Tests" and "#Pass" columns show the number of tests in each category and the number of them that pass. A test that is intended to succeed passes if both Puppet and μPuppet succeed and produce the same catalog (up to reordering of resources); a test that is intended to fail passes if both Puppet and μPuppet fail. The "#Unsupported" column shows the number of test cases that involve features that μPuppet does not handle. All of the tests either pass or use features that are not supported by μPuppet. Features that μPuppet (by design) does not support are italicised.

All of the examples listed in the above table are included in the repository on github[3], together with the resulting catalogs and error messages provided by Puppet.

[3]https://github.com/dcspaul/uPuppet

Feature	#Tests	#Pass	#Unsupported
Statements	11	11	0
Assignment	2	2	0
Case	1	1	0
If	4	4	0
Unless	4	4	0
Resources	18	11	7
Basics	2	2	0
Variables	3	3	0
User defined resource types	5	5	0
Virtual resources	1	0	1
Default values	1	0	1
Resource extension	4	0	4
Ordering Constraints	2	1	1
Classes	32	22	10
Basics	4	4	0
Inheritance	3	3	0
Scope	2	2	0
Variables & classes	6	6	0
Class Parameters	6	6	0
Overriding	5	0	5
Nesting and redefinition	6	1	5
Nodes	8	8	0
Resource Collectors	9	0	9
Basics	1	0	1
Collectors, references & variables	3	0	3
Application order	5	0	5

Table 3.1: Summary of test cases. Features in *italics* are not supported in μPuppet.

3.5.2 Unsupported features

Our formalisation handles some but not all of the distinctive features of Puppet. As mentioned in the introduction, we chose to focus effort on the well-established features of Puppet that appear closest to its declarative aspirations. In this section we discuss the features we chose not to support and how they might be supported in the future, in increasing order of complexity.

String interpolation. Puppet supports a rich set of string operations including string interpolation (i.e. variables and other expression forms embedded in strings). For example, writing `"Hello ${planet['earth']}!"` produces `"Hello world!"` if variable `planet` is a hash whose `'earth'` key is bound to `'world'`. String interpolation is not conceptually difficult but it is widely used and desugaring it correctly to plain string append operations is an engineering challenge.

Dynamic data types. Puppet 4 also supports type annotations, which are checked dynamically and can be used for automatic validation of parameters. For example, writing `Integer $x = 5` in a parameter list says that x is required to be an integer and its default value is 5. Types can also express constraints on the allowed values: for example, `5 =~ Integer[1,10]` is a valid expression that evaluates to `true` because 5 is an integer between 1 and 10. Data types are themselves values and there is a type `Type` of data types.

Undefined values and strict mode. By default, Puppet treats an undefined variable as having a special "undefined value" `undef`. Puppet provides a "strict" mode that treats an attempt to dereference an undefined variable as an error. We have focused on modelling strict semantics, so our rules get stuck if an attempt is made to dereference an undefined variable; handling explicit undefined values seems straightforward, by changing the definitions of lookup and related operations to return `undef` instead of failing.

Functions, iteration and lambdas. As of version 4, Puppet allows function definitions to be written in Puppet and also includes support for iteration functions (`each`, `slice`, `filter`, `map`, `reduce`, `with`) which take lambda blocks as arguments. The latter can only be used as function arguments, and cannot be assigned to variables, so Puppet does not yet have true first-class functions. We do see no immediate obstacle to handling these features, using standard techniques.

Nested constructs and multiple definitions. We chose to consider only

top-level definitions of classes and defined resources, but Puppet allows nesting of these constructs, which also makes it possible for classes to be defined more than once. For example:

```
1  class a {
2    $x1 = "a"
3    class b {
4      $y1 = "b"
5    }
6  }
7
8  class a::b {
9    $y2 = "ab"
10 }
11 include a
12 include a::b
```

Surprisingly, *both* line 4 and line 9 are executed (in unspecified order) when `a::b` is declared, so both `$::a::b::y1` and `$::a::b::y2` are in scope at the end. Our impression is that it would be better to simply reject Puppet manifests that employ either nested classes or multiple definitions, since nesting of class and resource definitions is explicitly discouraged by the Puppet documentation.

Dynamically-scoped resource defaults. Puppet also allows setting resource defaults. For example one can write (using the capitalised resource type `File`):

```
1    File { owner => 'alice' }
```

to indicate that the default owner of all files is `alice`. Defaults can be declared in classes, but unlike variables, resourced defaults are dynamically scoped; for this reason, the documentation and some authors both recommend using resource defaults sparingly. Puppet 4 provides an alternative way to specify defaults as part of the resource declaration.

Resource extension and overriding. In Puppet, attributes can be added to a resource which has been previously defined by using a *reference* to the resource, or removed by setting them to `undef`.

```
1  class main {
2    file { 'file': owner => 'alice' }
3    File['file'] { mode => '0755' }
4  }
```

However, it is an error to attempt to change the value of an already-defined resource, unless the updating operation is performed in a *subclass* of the class in which the resource was originally declared. For example:

```
1  class main::parent {
2    file { 'file':
3      owner => 'bob',
4      source => 'the source'
5    }
6  }
7  class main inherits main::parent {
8    File['file'] {
9      owner => 'alice',
10     source => undef
11   }
12 }
```

This illustrates that code in the derived class is given special permission to override any resource attributes that were set in the base class. Handling this behaviour seems to require (at least) tracking the classes in which resources are declared.

Resource collectors and virtual resources. *Resource collectors* allow for selecting, and also updating, groups of resources specified via predicates. For example, the following code declares a resource and then immediately uses the collector `File <|title == 'file'|>` to modify its parameters.

```
1  class main {
2          file { 'file': owner => 'alice' }
3          File <| title == 'file' |> {
4                  owner => 'bob',
5                  group => 'the group',
6          }
7  }
```

Updates based on resource collectors are unrestricted, and the scope of the modification is also unrestricted: so for example the resource collector `File<|owner='root'|>` will select all files owned by root, and potentially update their parameters in arbitrary ways. The Puppet documentation recommends using resource collectors only in idiomatic ways, e.g. using the title of a known resource as part of the predicate. Puppet also supports *virtual resources*, that is, resources with parameter values that are not added to the catalog until declared or referenced elsewhere. Virtual resources allow a resource to be declared in one place without the resource being included in the catalog. The resource can then be *realised* in one or more other places to include it in the catalog. Notice that you can realise virtual resources before declaring them:

```
1  class main {
2          realize User["alice"]
```

```
3          @user { "alice": uid => 100 }
4          @user { "bob": uid => 101 }
5          realize User["alice"]
6  }
```

As Shambaugh et al. (2016) observe, these features can have global side-effects and make separate compilation impossible; the Puppet documentation also recommends avoiding them if possible. We have not attempted to model these features formally, and doing so appears to be a challenging future direction.

Ordering constraints. By default, Puppet does not guarantee to process the resources in the catalog in a fixed order. To provide finer-grained (and arguably more declarative) control over ordering, Puppet provides several features: special *metaparameters* such as `ensure`, `require`, `notify`, and `subscribe`, *chaining arrows* `->` and `~>` that declare dependencies among resources, and the *require function* that includes a class and creates dependencies on its resources. From the point of view of our semantics, all of these amount to ways to define dependency edges among resources, making the catalog into a *resource graph*. Puppet represents the chaining arrow dependencies using metaparameters, so we believe this behavior can be handled using techniques similar to those for resource parameter overrides or resource collectors. The rules for translating the different ordering constraints to resource graph edges can be expressed using Datalog rules (Shambaugh et al., 2016) and this approach may be adaptable to our semantics too.

3.6 Related work

Other declarative configuration frameworks include LCFG (Anderson, 2008), a configuration management system for Unix, and SmartFrog (Goldsack et al., 2009), a configuration language developed by HP Labs. Of these, only SmartFrog has been formally specified; Anderson and Herry (2016) propose a formal semantics and identify some complications, including potential termination problems exhibited by the SmartFrog interpreter. Their semantics is presented in a denotational style, in contrast to the small-step operational semantics presented here for Puppet. Other systems, such as Ponder (Damianou, 2002), adopt an operational approach to policies for distributed systems.

Beyond this, there are relatively few formal studies of configuration languages, and we are aware of only two papers on Puppet specifically. Vanbrabant et al.

(2011) propose an access control technique for an early version of Puppet based on checking whether the changes to the catalog resulting from a change to the manifest are allowed by a policy. Catalogs are represented as XML files and allowed changes are described using path expressions. Shambaugh et al. (2016) present a configuration verification tool for Puppet called Rehearsal. Their tool is concerned primarily with the "realisation" stage of a Puppet configuration, and focuses on the problem of determinacy analysis: that is, determining whether a proposed reconfiguration leads to a unique result state. Shambaugh et al. (2016) consider a subset of Puppet as a source language, including resources, defined resources, and dependencies; they do not formalise the semantics of classes, inheritance or scope, though their implementation supports some of these features.

The present work continues a line of recent efforts to study the semantics of programming and scripting languages "in the wild". There have been efforts to define semantics for JavaScript (Maffeis et al., 2008; Guha et al., 2010), R (Morandat et al., 2012), PHP (Filaretti and Maffeis, 2014), and Python (Politz et al., 2013). Work on formal techniques for Ruby (Ueno et al., 2014) may be especially relevant to Puppet: Puppet is implemented in Ruby, and plugins can be written in Ruby, so modelling the behaviour of Puppet as a whole may require modelling both the Puppet configuration language and the Ruby code used to implement plugins, as well as other tools such as Hiera[4] that are an increasingly important component of the Puppet toolchain. Hiera supports reuse of data values by storing default values, which facilitates the modularity of Puppet. However, Puppet itself differs significantly from Ruby, and Puppet "classes" in particular bear little relation to classes in Ruby or other object-oriented languages. As for data-tracking by provenance, it would not be difficult to extend the tracking from Puppet to Hiera to find the source of data values.

3.7 Conclusions

In this chapter, we identified the main features of the Puppet language and their semantic behaviour. We named these main features as μPuppet. We modelled their semantics by operational semantics. These features includes resource, node, class and defined resource types. We also presented some meta results that justified some basic characterisation of μPuppet as a 'declarative' subset of Puppet.

[4] `https://docs.puppet.com/hiera/`.

Besides the theoretical results, we implemented an interpreter and a parser in Haskell for μPuppet based on its operational semantics and compared them with the Puppet 4.8 implementation on a number of examples. We also identified idiosyncrasies concerning evaluation order and scope where our initial approach differed from Puppet's actual behaviour. In the next two chapters, we will investigate *provenance* techniques developed in the database field in the context of operational semantics that explain where the output values in the catalog come from in the manifest, how the output values have been derived and what input values in the manifest are responsible for the output values in the catalog.

Chapter 4

Where and Expression Provenance

After defining the formal semantics of μPuppet, we now start to introduce provenance from the database field in the context of configuration language in this chapter. After the file is compiled to a catalog, it will be deployed to the node who requested to make a change to its configuration. The configuration on the node will be changed according to the catalog. When configuration failures appear, there must be mistakes in the configuration of the node changed according to the catalog. Configuration failures cause the lost of services. For example, misconfigured backup DNS (used upon attacks) made LinkedIn inaccessible for half a day. The administrators need to find the wrong parameters in the configurations. To make the configuration correct, the administrators find the solution to mend the manifest according to the possible reasons for the mistakes. Since these kind of mistakes are not bugs due to compilation failure, it is not obvious for the administrators to find out the part of the manifest that is responsible for the mistake. Moreover the manifest is normally written by many parties and is a large file, understanding the manifest and locating the problematic part in it is even harder. Provenance, as a technique to record the history of data changing over time, will naturally be adapted here to provide the information for the output data in the catalog about what happened over the compilation so the data was derived. With this provenance, the administrator will get more information besides the mistake data in the catalog, so to locate the problematic part in the manifest that caused this mistake.

The data in the catalog have been derived by the compilation process from the input data in the manifest. There are several possible ways through which the compilation procedure decides how an output data value was derived. The

first is that the output data value is copied from the input data from the manifest without going through any change. The second is that the data is generated by primitive computations, i.e. arithmetic operations on some input data in the manifest. The third is that the conditional constructs in the manifest influences the control flow where the output was derived in the catalog. In this chapter, we will focus on the first and second effects for which we formalise where- and expression-provenance for output data values. Where-provenance describes the location of the original input in the manifest where an output data value was copied from, while expression-provenance provides the information on how a data in the manifest has been generated by primitive computations and from what input data it is derived from. To make the first step, provenance will be built on μPuppet except array, hash and resource reference constructs. We believe it should be easy to extend these provenance to these constructs and explore it in the future.

4.1 Methodology

Since the data in the manifest goes through changes in the compilation process, provenance in the context of a configuration language means to record the changes of input data in the procedure of compilation. Since provenance is a formal specification, it is natural to build it based on the formal semantics of the configuration languages, i.e. μPuppet in our case.

It is natural for us to introduce a new construct in μPuppet to get involved in the compilation procedure so to record the desired information of changes on input data. As provenance in database field, the new construct could be a set of labels of mutually disjoint single elements. We have formalized the semantic behaviour of μPuppet using operational semantics in last chapter. With these labels, we could record the relevant change of the data in every step of evaluation, i.e. every semantic rule of μPuppet. In particular we annotate each data with a label in the manifest. When a labeled data gets through every step of evolution, its label will be evolved to another form representing the respective change depending on the form of provenance we define. When the compilation of a manifest stops, the data in the generated catalog will carry their labels. These labels will be provenance intended to record the relevant information of an output data value.

Later when we define a model of provenance, we will propose a form of labels

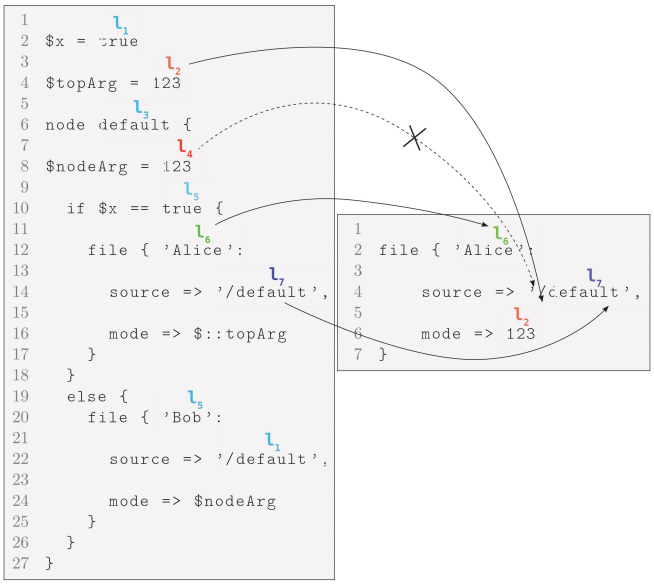

Figure 4.1: Where-provenance in Puppet

for recording the form of provenance. Then we will illustrate in every semantic rule of μPuppet how the label will be evolved in the rule.

4.2 Where-Provenance

We will define *where-provenance* that means to indicate the original location in the manifest for an output data in the catalog. Where-provenance can indicate the location of an input data value because the label of every input data is unique. So we know where an output data value was copied from by the correspondence of the labels of the input data and the output data. Such output data values are the ones that have not been changed and were copied as-is by the compilation. When the administrator is interested in such an output data value, where-provenance will inform the location of its input data in the manifest. As we mentioned, the manifest is normally large and contributed by many parties. It is hard to locate a particular data in the manifest, especially when there are many data of the same value. Furthermore, knowing the location of the input data of a data in the manifest, the administrator will know the contributor of this data. An example in Puppet below illustrates how where-provenance should work.

In Figure 4.1, all the input data values in the manifest are annotated with unique labels l_i. After the manifest is complied to a catalog, the inputs that have not been going through any changes remain in the catalog, as well as their labels.

For the multiple appearances of values, such as 123 and `default`, each appearance is labelled uniquely. In the catalog, we can know which data values exactly were copied from by the annotations. For example, the labels of the output data values 123 and `default` indicate which 123 and `default` in the inputs were copied from.

4.2.1 Annotated Compact Grammar

Where-provenance is defined to provide the location information of the input data value for an output data value. For this purpose, we will label every data value in a manifest with a unique label. In the process of every evaluation rule, the data value will carry its label. Depending on the specific evaluation rule, the labels on the data values will either remain if these data values are changed after the rule is applied, or be changed to an empty label on the result data value if these data values go through primitive operations in the rule. To formalising this behaviour of labels, we introduce first an infinite set of different elements $\{l_1, l_2, \ldots, \}$ as *labels* and an empty label as $\perp$ in the abstract syntax of μPuppet. A label that is either a l_i or $\perp$ is represented as β. The abstract syntax for labels is formalised as in Figure 4.2.

We will use these labels to annotate the compact syntax of μPuppet, catalogs and the environments helping the formal semantics of evaluation.

4.2.1.1 Annotations on Data Values

We will annotate the raw values in manifests in μPuppet. We start with annotating the primitive value types in μPuppet, which are integer i, strings w and boolean values `true` and `false`. We simply associate each value in manifests with a unique label l, represented as v^l called *labelled data value*. Such annotated raw data value is represented as v^β, i.e. $\widehat{v}$.

Besides the raw input data values in manifests, there are also other forms of data values related to μPuppet. After a manifest has been compiled, the generated catalog is a set of resource values. Resource values are a structured data type of primitive data values. We define the annotated resource values and then catalog in Figure 4.2.

$$
\begin{array}{lrcl}
\text{Label} & \beta & ::= & l \mid \bot \\
\text{Value} & v & ::= & i \mid w \mid \texttt{true} \mid \texttt{false} \\
\text{Annotated value} & \widehat{v} & ::= & v^{\beta} \\
\text{Catalog} & \widehat{v_{\mathrm{C}}} & ::= & \varepsilon \mid \widehat{v_{\mathrm{R}}} \widehat{\sqcup} \widehat{v_{\mathrm{C}}} \\
\text{Hash value} & \widehat{v_{\mathrm{H}}} & ::= & \varepsilon \mid k \Rightarrow \widehat{v}, \widehat{v_{\mathrm{H}}} \\
\text{Resource value} & \widehat{v_{\mathrm{R}}} & ::= & t\,\{w^{\beta} : \widehat{v_{\mathrm{H}}}\} \\
\text{Scope} & \alpha & ::= & :: \mid ::a \mid ::\texttt{nd} \mid \alpha\,\texttt{def} \\
\text{Statement} & s & ::= & \ldots \mid \texttt{scope}\,\alpha\,s \mid \texttt{skip}
\end{array}
$$

Figure 4.2: Annotated auxiliary constructs: catalogs and scopes

4.2.1.2 Annotating μPuppet

The constructs in μPuppet, such as expression e and statements s includes premier data values too and are labelled inductively on their structures. The annotated expressions and all kinds of statements are defined in Figure 4.2. We retain the representations e and s in the annotated compact syntax for simplicity.

4.2.1.3 Annotating evaluation environments

We will also annotate the environment constructs where the data values or statements are carried to help the evaluation procedure. Labelling these constructs can be made by induction on their structure. In store σ, we label every data value bound with a variable in it. The annotated store is represented as $\widehat{\sigma}$. κ is annotated if every definition associated with a class name in it is labelled. An annotated κ is represented as κ^{β}. Every definition is annotated if the statement set S or the parameter set ρ in this definition is labelled. Their compact syntax is in Figure 4.4.

4.2.2 Propagation of Evaluation in the Evaluation

Having the annotated compact syntax of μPuppet in hand, we will see how an annotation on a raw data value in a manifest should evolve under evaluation. We will define the evolution of the annotations step by step in semantic rules. We intend to get the evolved annotations on data values in the catalog that indicate the location of a data value back in the manifest. These *annotations* on data

$$
\begin{array}{lrcl}
\text{Expression} & e & ::= & \widehat{v} \mid \$x \mid \$::x \mid \$::a::x \\
& & \mid & e_1 + e_2 \mid e_1 - e_2 \mid e_1 / e_2 \mid e_1 > e_2 \mid e_1 = e_2 \mid e_1 \,\texttt{and}\, e_2 \mid e_1 \,\texttt{or}\, e_2 \mid \texttt{!}e \mid \ldots \\
& & \mid & \{H\} \mid [e_1, \ldots, e_n] \mid e_1[e_2] \mid e \;?\; \{M\} \\
\text{Hash} & H & ::= & \varepsilon \mid k \Rightarrow e, H \\
\text{Case} & c & ::= & e \mid \texttt{default} \\
\text{Matches} & M & ::= & \varepsilon \mid c \Rightarrow e, M \\
\text{Statement} & s & ::= & e \mid s_1 \_ s_2 \mid \$x = e \mid \texttt{unless}\ e\ \{s\} \mid \texttt{if}\ e\ \{s\}\ \texttt{else}\ \{s\} \mid \texttt{case}\ e\ \{C\} \mid D \\
\text{Cases} & C & ::= & \varepsilon \mid c : \{s\} \_ C \\
\text{Declaration} & D & ::= & t\ \{e : H\} \mid u\ \{e : H\} \mid \texttt{class}\ \{a : H\} \mid \texttt{include}\ a \\
\text{Manifest} & m & ::= & s \mid m_1 \_ m_2 \mid \texttt{node}\ Q\ \{s\} \mid \texttt{define}\ u\ (\rho)\ \{s\} \mid \texttt{class}\ a\ \{s\} \mid \texttt{class}\ a\ (\rho)\ \{s\} \\
& & \mid & \texttt{class}\ a\ \texttt{inherits}\ b\ \{s\} \mid \texttt{class}\ a\ (\rho)\ \texttt{inherits}\ b\ \{s\} \\
\text{Node spec} & Q & ::= & N \mid \texttt{default} \mid (N_1, \ldots, N_k) \mid r \in \mathit{RegExp} \\
\text{Parameters} & \rho & ::= & \varepsilon \mid x, \rho \mid x = e, \rho
\end{array}
$$

Figure 4.3: Annotated abstract syntax of μPuppet

$$
\begin{array}{lrcl}
\text{Store} & \widehat{\sigma} & : & \mathit{Scope} \times \mathit{Var} \rightarrow \widehat{\mathit{Value}} \\
\kappa & \widehat{\kappa} & ::= & \varepsilon \mid a \rightarrow \hat{d} \mid a \rightarrow \hat{d}, \hat{k} \\
\text{Definition} & \widehat{d} & ::= & \bot \mid \mathsf{ClassDef}(\mathit{optc}, \rho, s) \mid \mathsf{DeclaredClass}(\mathit{optc}) \mid \mathsf{ResourceDef}(\rho, s)
\end{array}
$$

Figure 4.4: Annotated abstract syntax of environments

values are *where-provenance* for these data values we want to define. For this purpose, we will define the evolution of the annotations on data values in each evaluation rule according to the following rules.

(1) If there is an operation in an evaluation rule where using data values as inputs, the result data value after this evaluation step will have an empty annotation $\bot$ as its annotation.

(2) If a conditional statement if is evaluated, the control expression will be omitted after its evaluation; the annotated data values in any branch will remain if this branch is chosen.

(3) If a case statement is evaluated, the control expression will be omitted after its evaluation; the data values in the branch chosen after its evaluation will

be kept in the result of its evaluation.

(4) If a selector expression is evaluated, the control expression will be omitted after its evaluation; the branch chosen after its evaluation will be kept in the result of its evaluation.

(5) The annotations of the data values not falling in the cases above will remain in the evaluation.

4.2.2.1 Annotated Judgements

We defined three judgements for evaluating manifests in Chapter 3. For annotated manifests, we will need to change these judgements to the forms with annotations. Since the annotations are just decorations on the syntax of μPuppet, the changes on the annotations made by semantic rules are only shown in the annotations after the evaluation, but on the output elements of the evaluations. So when considering the annotations on the constructs in the evaluation judgements, we only need to add the annotations on the constructs as inputs and outputs.

$$\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e' \qquad \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s' \qquad \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, m \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m'$$

We will illustrate next the evaluation rules for the constructs in μPuppet where there are crucial evolutions on the annotations on data values.

4.2.2.2 Annotation Propagation in Expression Evaluation

We will see how the annotations in expression evaluations change. We will look at different constructs of expressions respectively. We will have a sense of how the annotations on data values should change in the evaluation rules for the basic operations.

Arithmetic expressions For the rules for evaluating arithmetic operations on two operands, there are three cases. First, if the two operands are both annotated data values $i_1^{\beta_1}$ and $i_2^{\beta_2}$, the result data value $(i_1 +_{\mathbb{Z}} i_2)$ after the evaluation will be labelled with an empty label $\bot$. The labels β_1 and β_2 will be discarded since the two input data values were consumed in the + operation. Second, if the left hand side operand of an arithmetic operation is an annotated data value i^{β} and its right hand side is an expression e, the evaluate e will be evaluated in the

next step. The data value i with its label β will be preserved in this evaluation step. Last, if both operands are two expressions e_1 and e_2, the first expression e_1 will be evaluated first using the right rule. The evaluation rules for annotated expressions below shows how the labels on the data values change.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 + e_2 \xrightarrow{\alpha} e_1' + e_2}\ \textsc{ArithLeft} \qquad \frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, i^{\beta} + e \xrightarrow{\alpha} i^{\beta} + e'}\ \textsc{ArithRight}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, i_1^{\beta_1} + i_2^{\beta_2} \xrightarrow{\alpha} (i_1 +_{\mathbb{Z}} i_2)^{\perp}}\ \textsc{ArithValue}$$

The evaluation rules for annotated constructs for other arithmetic operators $-, *, /$ are similar as these rules for $+$.

Comparison expressions Similarly we will define how to evaluate annotated comparison expressions. There are three cases too. We make the comparison operation on the operator $>$ as an example. When the two operands of $>$ operator are annotated data values $v_1^{\beta_1}$ and $v_2^{\beta_2}$, the evaluation result will be either `true` or `false` depending on if $v_1 >_{\mathbb{Z}} v_2$ or $v_1 <=_{\mathbb{Z}} v_2$. Its annotation will be an empty label $\perp$ since $v_1^{\beta_1}$ and $v_2^{\beta_2}$ have been consumed as inputs of $>$ operation. When the left hand side operand is an annotated data value v_1^{β} and the right hand side is an annotated expression e, the next step will evaluate e and v_1^{β} will be carried in the result of the step. If both sides of $>$ operation are expressions e_1 and e_2, the left hand expression e_1 will be evaluated next.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 > e_2 \xrightarrow{\alpha} e_1' > e_2}\ \textsc{CompLeft} \qquad \frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta} > e \xrightarrow{\alpha} v^{\beta} > e'}\ \textsc{CompRight}$$

$$\frac{v_1 >_{\mathbb{Z}} v_2}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v_1^{\beta_1} > v_2^{\beta_2} \xrightarrow{\alpha} \texttt{true}^{\perp}}\ \textsc{CompValueI}$$

$$\frac{v_1 <=_{\mathbb{Z}} v_2}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v_1^{\beta_1} > v_2^{\beta_2} \xrightarrow{\alpha} \texttt{false}^{\perp}}\ \textsc{CompValueII}$$

The annotation evolution in the rules for other comparison operators $<$, $>=$, $<=$, $==$ and $!=$ are similar and we omit them.

Boolean expressions We now will see how to evaluate annotated "and" and "!" expressions. When the expression on and operator is on two true values $\texttt{true}^{\beta_1}$ and $\texttt{true}^{\beta_2}$, the result will be $\texttt{true}^{\perp}$. The annotations β_1 and β_2 on the two inputs will not carried in the annotation of the result since the inputs have been consumed. Similarly, when the left hand operator is an annotated false value $\texttt{false}^{\beta}$, the result will be $\texttt{false}^{\perp}$ where β will be not recorded.

For ! expressions, when the operator ! is applied on an annotated truth value $\texttt{true}^{\beta}$ or $\texttt{false}^{\beta}$, the result will be $\texttt{false}^{\perp}$ or $\texttt{true}^{\perp}$ where the label β will not be recorded for the same reason.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \,\texttt{and}\, e_2 \xrightarrow{\alpha} e_1' \,\texttt{and}\, e_2} \textsc{AndLeft}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{false}^{\beta} \,\texttt{and}\, e \xrightarrow{\alpha} \texttt{false}^{\perp}} \textsc{AndRightI}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{true}^{\beta} \,\texttt{and}\, e \xrightarrow{\alpha} \texttt{true}^{\beta} \,\texttt{and}\, e'} \textsc{AndRightII}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{true}^{\beta_1} \,\texttt{and}\, \texttt{true}^{\beta_2} \xrightarrow{\alpha} \texttt{true}^{\perp}} \textsc{AndValueI}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{true}^{\beta_1} \,\texttt{and}\, \texttt{false}^{\beta_2} \xrightarrow{\alpha} \texttt{false}^{\perp}} \textsc{AndValueII}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, !\, e \xrightarrow{\alpha} !\, e'} \textsc{NotSTEP} \qquad \frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, !\,\texttt{true}^{\beta} \xrightarrow{\alpha} \texttt{false}^{\perp}} \textsc{NotValueI}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, !\,\texttt{false}^{\beta} \xrightarrow{\alpha} \texttt{true}^{\perp}} \textsc{NotValueII}$$

The rules for disjunction are similar and are omitted.

Selector In the rule SChoose, the current branch in the selector expression is chosen since the two control data values v^{β} and $v_1^{\beta_1}$ match. The statements in the body of this branch will retain in the result. The annotations β and β_1 will retain in the result of this evaluation step since the data values they labelled do

not retain. Similarly, in the rule SDefault, the default branch is chosen to be evaluated continuously, while the label β on the control data value v^{β} will not be carried in the result of this step evaluation. In the rule SChooseI, the current branch is not chosen since the two control data values do not match. Then the current branch is omitted for later evaluation. The label on the control data value $v_1^{\beta_1}$ will not be preserved in any form for later evaluation.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e\ ?\{M\} \xrightarrow{\alpha} e'\ ?\{M\}}\ \textsc{SCon}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta}\ ?\{e_1 \Rightarrow e, M\} \xrightarrow{\alpha} v^{\beta}\ ?\{e_1' \Rightarrow e, M\}}\ \textsc{SEle}$$

$$\frac{\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\} \xrightarrow{\alpha} e}\ \textsc{SChoose}$$

$$\frac{\neg\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\} \xrightarrow{\alpha} v^{\beta}\ ?\{M\}}\ \textsc{SChooseI}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta}\ ?\{\texttt{default} \Rightarrow e, M\} \xrightarrow{\alpha} e}\ \textsc{SDefault}$$

Variables The annotations in the evaluation rules for variables will not change during the evaluation procedure. So we omit them here and are shown in Appendix B.

4.2.2.3 Annotation Propagation in Statement Evaluation

After defining the annotation evolution in expression evaluation, we will define now how the annotations on data values appearing in a statement evolve in its evaluation rule. Similarly, the evaluation for annotated statements will happen under annotated environments $\widehat{\sigma}$, $\widehat{\kappa}$ and $\widehat{v_{\mathrm{C}}}$. All the evaluation rules for statements will follow the annotated judgement for statements. Since there are no primitive operations happening to data values, there is no direct change on the annotations from some label l to an empty label $\perp$ in one evaluation step. Instead, there are conditional statements `if`, `unless` and `case` that will choose one branch of the statement according to the control expressions.

In the following, we will see the evaluation rule for storing an annotated data value v^β in an assignment to the annotated store $\widehat{\sigma}$. We will also define how the annotations in the control expressions will evolve in the evaluation of the conditional statements. The evaluation of other statements involve either the propagation to the expression evaluation, or the operation of looking up or changing environments. Each does not bring changes on annotations in the evaluation of the statements. So the form of the evaluation rules for these annotated statements will remain the same as the original rules, except all the constructs in the rules are annotated. We will omit to list these rules here but put them in the appendix.

Assignment The rule AssignStep propagates the evaluation of the annotated expression e in an assignment statement to the evaluation for the expression e. When the expression is evaluated to an annotated data value v^β, the rule Assign will put it into the annotated store $\widehat{\sigma}$ associating the variable x in the assignment and the scope this assignment belongs to.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x = e \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x = e'} \textsc{AssignStep}$$

$$\frac{x \notin dom(\sigma_\alpha)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x = v^\beta \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}[(\alpha, x) : v^\beta], \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}} \textsc{Assign}$$

If To evaluate an if statement, if e is not an annotated truth value, the rule IfStep will lift the evaluation of the control expression e to the rules of expression evaluation. If the control expression is an annotated truth value $\texttt{true}^\beta$ or $\texttt{false}^\beta$, the rule IfT or IfF will choose s_1 or s_2 to be conditionally evaluated, while the annotation β on the control data value will be discarded in the result of IfT or IfF evaluation.

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{if}\ e\ \{s_1\}\ \texttt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{if}\ e'\ \{s_1\}\ \texttt{else}\ \{s_2\}}\ \textsc{IfStep}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{if}\ \texttt{true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},s_1}\ \textsc{IfT}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{if}\ \texttt{false}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},s_2}\ \textsc{IfF}$$

Case After the control expressions have been evaluated to data values by the rule CaseStep1 and CaseStep2, the other rules for `case` will be applied. If the two control data values v and v_1 match, the body of the current branch of `case` statement will be chosen to be evaluated later. The annotations on these two values β and β_1 will not retain in this branch for further evaluation. If they do not match, this branch will not be chosen and be omitted. The remaining `case` statement will be evaluated further. The annotation on the control data value of the omitted branch will not be carried in the remaining statement. In the rule CaseDefault, the evaluation will choose the default branch for further evaluation. Similarly, the annotation on the control data value will retain in the result of the evaluation.

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{case}\ e\ \{C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{case}\ e'\ \{C\}}\ \textsc{CaseStep1}$$

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{case}\ v^{\beta}\ \{e : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{case}\ v^{\beta}\ \{e' : \{s\} \llcorner C\}}\ \textsc{CaseStep2}$$

$$\frac{\mathsf{caseMatch}(v,v_1)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},s}\ \textsc{CaseMatch}$$

$$\frac{\neg\mathsf{caseMatch}(v,v_1)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{case}\ v^{\beta}\ \{C\}}\ \textsc{CaseNoMatch}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{case}\ v^{\beta}\ \{\varepsilon\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{skip}}\ \textsc{CaseDone}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{case}\ v^{\beta}\ \{\texttt{default} : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},s\}}\ \textsc{CaseDefault}$$

4.2.2.4 Annotation Propagation in the Evaluation of Resources and Manifests

Since the evaluation for resources and manifests does not involve direct annotation changes, the form of their evaluation rules are the same as their original evaluation rules, except the constructs in the rules are all annotated. We will omit these rules here and show them in Appendix B.

4.2.3 Correctness of Where-provenance

We have defined where-provenance for the output data values in catalogs. We will check in this section whether the semantics of where-provenance we defined meets the intuitive requirement of where-provenance. We started defining where-provenance from annotating the data values with unique labels. It is easy to notice that in every evaluation step by an evaluation rule the annotations are either copied to the result of the evaluation or omitted with their data values. The uniqueness of the labels on input data values in the manifest should remain in the annotations of the output data values in the catalogs. Thus by the annotation of an output data value, we can find the location of its input data value where it was copied in the manifest. In the following we will formalise the semantic correctness of where-provenance first. Then we will prove it holds for our definition of where-provenance.

4.2.3.1 Extraction Functions

To show where-provenance is correct, the uniqueness of annotations on the output values in the catalogs is crucial to hold after the evaluation. For this purpose, we could prove the preservation of the uniqueness of annotations to establish the semantic correctness of where-provenance. To examine the preservation of the uniqueness of annotations in the evaluation, we could check if the set of the annotated data values in the catalog after the evaluation is subsumed in that of the manifest. (We want to show that the annotations that appear in the catalogs are from some initial annotations in the manifests.) To facilitate this purpose, we will define *extraction functions* that will take all the annotated data values from all the constructs of μPuppet and the evaluation environments.

In the following, we will first define the extract function $labels(e)$ for the expressions in μPuppet. Then the extract functions for statements, manifests

$$
\begin{array}{lcl}
labels(v^{\perp}) & = & \emptyset \\
labels(v^{l}) & = & \{v^{l}\} \\
labels(\varepsilon) & = & \emptyset \\
labels(\texttt{default}) & = & \emptyset \\
labels(\$x) & = & \emptyset \\
labels(\$ :: x) & = & \emptyset \\
labels(\$ :: a :: x) & = & \emptyset \\
labels(e_1 + e_2) & = & labels(e_1) \cup labels(e_2) \\
labels(e_1 - e_2) & = & labels(e_1) \cup labels(e_2) \\
labels(e_1 > e_2) & = & labels(e_1) \cup labels(e_2) \\
labels(e_1 = e_2) & = & labels(e_1) \cup labels(e_2) \\
labels(e_1 \,\texttt{and}\, e_2) & = & labels(e_1) \cup labels(e_2) \\
labels(e_1 \,\texttt{or}\, e_2) & = & labels(e_1) \cup labels(e_2) \\
labels(!e) & = & labels(e) \\
labels(e?\ \{M\}) & = & labels(e) \cup labels(M) \\
labels(k \Rightarrow e, H) & = & labels(e) \cup labels(H) \\
labels(c \Rightarrow e, M) & = & labels(e) \cup labels(M)
\end{array}
$$

Figure 4.5: Extraction Function $labels(e)$ for Annotated Expressions

and the environment will be defined inductively on their structure.

$labels(e)$ Extraction function $labels(e)$ takes an expression in μPuppet as its input and returns a set of annotated data values in the form v^{β}. For any annotated data value v^{l}, $labels(e)(v^{l})$ gives back a singleton element set $\{v^{l}\}$. While for the data value annotated with $\perp$, $labels(v^{\perp})$ gives back an empty set. In other words, we will omit the data values annotated with $\perp$. The function on any variable gives an empty set. After defining the primitive constructs of expressions in $labels(e)$, the function on other expressions will be defined inductively on the structure of these expressions. For example, the result of $labels(e)$ for expressions of binary operations is defined as the union of the results of the functions on both operands. Similarly, the application of $labels(e)$ on the other expressions gives the union of the results of this function on each construct in the expression. This complete definition of function $labels(e)$ is in Figure 4.5.

$$
\begin{array}{lcl}
labels(\$x = e]) & = & labels(e) \\
labels(s_1 \llcorner s_2) & = & labels(s_1) \cup labels(s_2) \\
labels(\texttt{unless } e\ \{s\}) & = & labels(e) \cup labels(s) \\
labels(\texttt{if } e\ \{s_1\}\ \texttt{else}\ \{s_2\}) & = & labels(e) \cup labels(s_1) \cup labels(s_2) \\
labels(\texttt{case } e\ \{C\}) & = & labels(e) \cup labels(C) \\
labels(c : \{s\} \llcorner C) & = & labels(c) \cup labels(s) \cup labels(C) \\
labels(\varepsilon) & = & \emptyset \\
labels(t\ \{e : H\}) & = & labels(e) \cup labels(H) \\
labels(u\ \{e : H\}) & = & labels(e) \cup labels(H) \\
labels(\texttt{class}\ \{a : H\}) & = & labels(H) \\
labels(\texttt{include } a) & = & \emptyset \\
labels(\texttt{skip}) & = & \emptyset \\
labels(\texttt{scope } \alpha\ s) & = & labels(s)
\end{array}
$$

Figure 4.6: Extraction Function $labels(s)$ for Annotated Statements

$labels(s)$ Function $labels(s)$ takes a statement s and returns the set of the annotated data values v^l in the statement as result. It is also defined inductively on the structure of every statement. Similarly, it is also defined for the run-time statements. The complete definition of $labels(s)$ for the annotated statements and run-time statements in μPuppet syntax is in Figure 4.6.

$labels(m)$ Function $labels(m)$ takes an annotated manifest element m and returns a set of the annotated data values in it. It is also defined inductively on the structure of each element in m. To define $labels(m)$, we will define the extraction function for an annotated parameter set ρ. Function $labels(\rho)$ takes a set of annotated parameter set ρ and returns a set of annotated data values appearing in ρ. The default values of parameters are given as inputs in the manifest. So they are annotated with some unique labels l. The definition of $labels(\rho)$ is as below.

$$labels(\{x_1 = v_1^{l_1}, \ldots, x_n = v_n^{l_n}, y_1, \ldots, y_m\}) = \{v_1^{l_1}, \ldots, v_n^{l_n}\}$$

The complete definition of $labels(m)$ is in Figure 4.7.

$labels(\hat{\sigma})$**,** $labels(\hat{\kappa})$ **and** $labels(\widehat{v_C})$ These functions are defined for the environments in the evaluation. Function $labels(\hat{\sigma})$ takes an annotated store $\hat{\sigma}$ and gets a set of annotated data values appearing in $\hat{\sigma}$ whose annotations are not $\perp$ as

$$
\begin{array}{lcl}
labels(m_1 \llcorner m_2) & = & labels(m_1) \cup labels(m_2) \\
labels(\texttt{node}\ Q\ \{s\}) & = & labels(s) \\
labels(\texttt{define}\ u\ (\rho)\ \{s\}) & = & labels(\rho) \cup labels(s) \\
labels(\texttt{class}\ a\ \{s\}) & = & labels(s) \\
labels(\texttt{class}\ a\ (\rho)\ \{s\}) & = & labels(\rho) \cup labels(s) \\
labels(\texttt{class}\ a\ \texttt{inherits}\ b\ \{s\}) & = & labels(s) \\
labels(\texttt{class}\ a\ (\rho)\ \texttt{inherits}\ b\ \{s\}) & = & labels(\rho) \cup labels(s)
\end{array}
$$

Figure 4.7: Extraction Function $labels(m)$ for Annotated Manifests

$$
\begin{array}{lcl}
labels(\hat{\sigma}) & = & labels(\{(\alpha_1 x_1, v_1^{\beta_1}), \ldots, (\alpha_n x_n, v_n^{\beta_n})\}) = \{labels(v_1^{\beta_1}), \ldots, labels(v_n^{\beta_n})\} \\
labels(\widehat{\sigma_\alpha}) & = & labels(\{(\$\alpha' x, v^\beta) \mid (\$\alpha' x, v^\beta) \in \hat{\sigma}, \alpha' = \alpha\})
\end{array}
$$

Figure 4.8: Extraction Function $labels(\hat{\sigma})$ for Annotated Stores

result. We then extend this function to the functions $\hat{\sigma}[(\alpha, x) : v^a]$, $clear(\hat{\sigma}, \alpha\texttt{def})$ and σ_α. The complete definition is in Figure 4.8.

Function $labels(\hat{\kappa})$ takes an annotated definition environment $\hat{\kappa}$ and gives back the union of the function $labels(\hat{d})$ on every definition $\widehat{d_i}$ in $\hat{\kappa}$. The definition of $labels(\hat{\kappa})$ is in Figure 4.9.

Function $labels(\widehat{v_{\mathrm{C}}})$ takes an annotated catalog. It applies $labels(\widehat{v_{\mathrm{R}}})$ to each $\widehat{v_{\mathrm{R}}}$ in $\widehat{v_{\mathrm{C}}}$ and returns the union of the results. Its definition is as below.

$$labels(\widehat{v_{\mathrm{R}}} \llcorner \widehat{v_{\mathrm{C}}}) = labels(\widehat{v_{\mathrm{C}}}) \cup labels(\widehat{v_{\mathrm{R}}})$$

$$
\begin{array}{lcl}
labels(\{a_1 \to \widehat{d_1}, \ldots, a_n \to \widehat{d_n}\}) & = & \bigcup_{i=1}^{n} labels(\widehat{d_i}) \\
labels(\bot) & = & \emptyset \\
labels(\mathsf{ClassDef}(optc, \rho, s)) & = & labels(\rho) \cup labels(s) \\
labels(\mathsf{DeclaredClass}(optc)) & = & \emptyset \\
labels(\mathsf{ResourceDef}(\rho, s)) & = & labels(\rho) \cup labels(s)
\end{array}
$$

Figure 4.9: Extraction Function $labels(\hat{\kappa})$ for Annotated Definition Environments

4.2.3.2 Invariants of the Evaluation

We have defined the annotated syntax of μPuppet and how the annotations are propagated along the evaluation rules so that where-provenance of the data values in the catalog is defined. Given an annotated manifest, its evaluation will generate an annotated catalog. These annotations are where-provenance of the data values they annotate. The purpose of where-provenance we desire is to trace back the input data value in the manifest from which the data value in the catalog has been copied in the process of evaluation. We want to check if the semantics of where-provenance we defined indeed satisfies this purpose.

As we mentioned, where-provenance means to identify the input data value in the manifest that an output data value in the catalog has been copied from. Since the initial data values in the manifest are uniquely labelled by some l, to identify the input data value for an output data value, we only need to locate the input data value that has the same annotation as this output data value. To formalise this correctness semantics, we use the extraction functions to get all the annotated data values from the manifest and the catalog after evaluation. If the set of the output data values is a subset of that of the input data values, we know that all the annotated output data values must come from some input data value and we can identify them back in the manifest by their annotations.

We will formalise this correctness in the context of the semantics of μPuppet. We have defined three judgements for the evaluation of μPuppet. To formalise the correctness we will first define an *invariant* that specifies this correctness meaning for evaluating the elements in the expression, statement or manifest category. With these invariants, we can define the correctness when a manifest is evaluated. In this section, we will define three invariants in expression, statement and manifest evaluation and prove that they hold for every evaluation rule in the evaluation category respectively.

Invariant of Expression Valuation We want to describe the meaning of correctness when evaluating expressions by an invariant that states that when evaluating an expression e, the set of all annotated data values in the input subsumes that of the output. Recall that the annotated evaluation judgement for expressions we defined is as below.

$$\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'$$

The input elements that take part in the evaluation are the expression e and the environments $\widehat{\sigma}$, $\widehat{\kappa}$ and $\widehat{v_{\mathrm{C}}}$. The output of the evaluation is a new expression e'. The evaluation of expressions does not generate new environments but might read from some environment. That is, when generating a new expression e', the evaluation might need to look up some environment and read some data value in the environment. Thus the environment could supply some data values to e', the output of the evaluation. We formalise the invariant of expression evaluation as below.

Theorem 4.2.1 (Invariant of Expression Evaluation)**.** When evaluating an expression e in μPuppet such that $\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'$, the subsumption on the sets of annotated data values

$$labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \cup labels(e) \supseteq labels(e')$$

holds.

We will prove this invariant holds for all the rules of expression evaluation. We choose the proofs below for some rules where the annotations in the input would be changed by the evaluation rules. The proofs for the rest of the rules are shown in the appendix.

Proof. Prove by induction.

1 Arithmetic expressions

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, i_1^{\beta_1} + i_2^{\beta_2} \xrightarrow{\alpha} (i_1 +_{\mathbb{Z}} i_2)^{\perp}} \textsc{ArithValue}$$

By definition, we have $labels(i_1^{\beta_1} + i_2^{\beta_2}) = labels(i_1^{\beta_1}) \cup labels(i_2^{\beta_2})$. If both β_1 and β_1 are some concrete labels l_1 and l_2, then $labels(i_1^{\beta_1}) \cup labels(i_2^{\beta_2}) = \{i_1^{\beta_1}, i_2^{\beta_2}\} = \{i_1^{l_1}, i_2^{l_2}\}$. If β_1 or β_1 is $\perp$, then $labels(i_1^{\beta_1}) \cup labels(i_2^{\beta_2})$ is equal to $\{i_2^{\beta_2}\}$ or $\{i_1^{\beta_1}\}$. We know $labels((i_1 +_{\mathbb{Z}} i_2)^{\perp}) = \emptyset$. Since $\emptyset \subseteq labels(i_1^{\beta_1}) \cup labels(i_2^{\beta_2}) \cup labels(\hat{\sigma})$, we have $labels(i_1 +_{\mathbb{Z}} i_2^{\perp}) \subseteq labels(i_1^{\beta_1} + i_2^{\beta_2}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. The invariant holds for this evaluation step.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, i^{\beta} + e \xrightarrow{\alpha} i^{\beta} + e'} \textsc{ArithRight}$$

By definition, we have $labels(i^{\beta} + e) = labels(i^{\beta}) \cup labels(e)$ and similarly $labels(i^{\beta} + e') = labels(i^{\beta}) \cup labels(e')$. By induction hypothesis, we know

the invariant holds for $\hat{\sigma}, \hat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'$ then $labels(e') \subseteq labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \cup labels(e)$, i.e. $labels(e') \subseteq labels(e) \cup labels(\hat{\sigma})$. Then $labels(i^{\beta}) \cup labels(e') \subseteq labels(i^{\beta}) \cup labels(e) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, i.e. $labels(i^{\beta} + e') \subseteq labels(i^{\beta} + e) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. The invariant holds for this evaluation step.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 + e_2 \xrightarrow{\alpha} e_1' + e_2} \textsc{ArithLeft}$$

By definition, we have $labels(e_1 + e_2) = labels(e_1) \cup labels(e_2)$ and $labels(e_1 + e_2') = labels(e_1) \cup labels(e_2')$. By induction hypothesis, we know the invariant holds for $\hat{\sigma}, \hat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'$ then $labels(e') \subseteq labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \cup labels(e)$, i.e. $labels(e_1') \subseteq labels(e_1) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. Then $labels(e_1) \cup labels(e_2) \subseteq labels(e_1') \cup labels(e_2) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, i.e. $labels(e_1 + e_2) \subseteq labels(e_1' + e_2) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. The invariant holds for this evaluation step.

2 Variables

$$\frac{x \in dom(\sigma_\alpha)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x \xrightarrow{\alpha} \sigma_\alpha(x)} \textsc{LVar}$$

Since $x \in dom(\hat{\sigma}_\alpha)$ for some α, $labels(\hat{\sigma}_\alpha(x)) \subseteq labels(\hat{\sigma})$ by definition. We also have $labels(\$x) = \emptyset$ and $labels(\hat{\sigma}_\alpha(x)) = labels(\hat{\sigma}_\alpha(x))$. Since $labels(\hat{\sigma}_\alpha(x)) \subseteq labels(\hat{\sigma})$, we have $labels(\hat{\sigma}_\alpha(x)) \subseteq labels(\$x) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. Then the invariant holds for this evaluation step.

□

Invariant of Statement Evaluation Having defined and proved the invariant in expression evaluation, we could state the similar correctness meaning as invariant for the statement evaluation, one level up than the expression. To remind, the annotated judgement of evaluating statements is as below.

$$\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha}_s \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v_{\mathrm{C}}'}, s'$$

As for the inputs of evaluating a statement, a statement s and the environments $\widehat{\sigma}$, $\widehat{\kappa}$ and $\widehat{v_{\mathrm{C}}}$ could take part in the evaluation. The output of the evaluation would be a new statement s' and updated environments $\widehat{\sigma'}$, $\widehat{\kappa'}$ and $\widehat{v'_{\mathrm{C}}}$. The input data values can be in s, $\widehat{\sigma}$, $\widehat{\kappa}$ or $\widehat{v_{\mathrm{C}}}$. They can flow from any one of them into another component among them to form the output data values. After multiple steps, annotated data values could go into any of these constructs to make these constructs updated. Therefore, the invariant in statement evaluation will be stated as the following.

Theorem 4.2.2 (Invariant of Statement Evaluation)**.** When evaluating a statement s in μPuppet such that $\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha}_s \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'$, the subsumption on the sets of annotated data values

$$labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \cup labels(s) \supseteq$$
$$labels(\widehat{\sigma'}) \cup labels(\widehat{\kappa'}) \cup labels(\widehat{v'_{\mathrm{C}}}) \cup labels(s')$$

holds.

We will prove this invariant for all the evaluation rules for annotated statements. We will choose the proofs of some rules that will change the annotations in the input. The proofs for the rest of the rules are in the appendix.

Proof. Prove by induction.

1 Evaluation of If statement

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s_1}\ \textsc{IfT}$$

By definition, $labels(\texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\}) = labels(\texttt{true}^{\beta}) \cup labels(s_1) \cup labels(s_2)$. We have $\{\texttt{true}^{\beta}\} \cup labels(s_1) \cup labels(s_2) \supseteq labels(s_1)$, i.e. $labels(\texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\}) \supseteq labels(s_1)$. Since $\widehat{\sigma}$, $\widehat{\kappa}$ and $\widehat{v_{\mathrm{C}}}$ do not change after this evaluation step, we have $labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \cup labels(\texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\}) \supseteq labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \cup labels(s_1)$. This induction rule holds w.r.t. the invariant.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{if false}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s_2}\ \textsc{IfF}$$

The proof is similar to the rule IfT

2 Evaluation of Case statement

$$\frac{\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s}\ \textsc{CaseMatch}$$

By definition, we have $labels(\mathtt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \llcorner C\}) = labels(v^{\beta}) \cup labels(v_1^{\beta_1}) \cup labels(s) \cup labels(C)$. Since $labels(v^{\beta}) \cup labels(v_1^{\beta_1}) \cup labels(s) \cup labels(C) \supseteq labels(s)$, we have $labels(\widehat{v_{\mathrm{C}}}) \cup labels(\mathtt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \llcorner C) \supseteq labels(s)$. Then $labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \cup labels(\mathtt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \llcorner C) \subseteq labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \cup labels(s)$. This induction rule holds w.r.t. the invariant.

$$\frac{\neg\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{case}\ v^{l}\ \{v_1^{l_1} : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{case}\ v^{l}\ \{C\}}\ \textsc{CaseNoMatch}$$

The proof is similar as the rule above.

3 Evaluation of Resource-like class declarations

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s)\quad \widehat{\kappa}(b) = \mathsf{DeclaredClass}(\beta)\quad s' = merge(\rho, v_{\mathrm{H}})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}[a : \mathsf{DeclaredClass}(::b)], \widehat{v_{\mathrm{C}}}, \mathtt{scope}\ (::a)\ \{s' \llcorner s\}}\ \textsc{CDecPD}$$

We have $labels(\mathtt{class}\ \{a : v_{\mathrm{H}}\}) = labels(v_{\mathrm{H}})$ by definition. Since $s' = merge(\rho, v_{\mathrm{H}})$, we have $labels(s') \subseteq labels(\rho) \cup labels(v_{\mathrm{H}})$. We have $labels(\mathtt{scope}\ (::a)\ \{s' \llcorner s\}) = labels(s') \cup labels(s)$ by definition. We also know $labels(\widehat{\kappa}(a)) = labels(\mathsf{ClassDef}(b, \rho, s)) = labels(s) \cup labels(\rho)$. Since $labels(\mathsf{DeclaredClass}(b)) = \emptyset$ we have $labels(\widehat{\kappa}[a : \mathsf{DeclaredClass}(::b)]) \subseteq labels(\widehat{\kappa})$. Thus we have $labels(v_{\mathrm{H}}) \cup labels(\widehat{\kappa}) \subseteq labels(s') \cup labels(s) \cup labels(\widehat{\kappa}[a : \mathsf{DeclaredClass}(::b)])$. Then $labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \cup labels(\mathtt{class}\ \{a : v_{\mathrm{H}}\}) \supseteq labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}[a : \mathsf{DeclaredClass}(::b)]) \cup labels(\widehat{v_{\mathrm{C}}}) \cup labels(s') \cup labels(s)$. This rule holds w.r.t. the invariant.

The proofs for other evaluation rules of statements are in Appendix B.

□

Invariant of Manifest Evaluation We will build up the invariant of manifest evaluation given the invariants for the expression and statement level evaluation.

The judgement of the annotated manifest evaluation is as below.

$$\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, m \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m'$$

The inputs of every rule evaluating a manifest element are an annotated manifest element m and the environments $\widehat{\sigma}$, $\widehat{\kappa}$ and $\widehat{v_{\mathrm{C}}}$. The output of the evaluation are a new manifest element m' and updated environments $\widehat{\sigma'}$, $\widehat{\kappa'}$ and $\widehat{v'_{\mathrm{C}}}$. Thus the invariant for the sets of annotated data values before and after an evaluation step is stated as the following.

Theorem 4.2.3 (Invariant of Manifest Evaluation)**.** When evaluating a manifest element m in μPuppet such that $\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, m \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m'$, the subsumption on the sets of annotated data values

$$\begin{aligned} &labels(m) \cup labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \supseteq \\ &labels(m') \cup labels(\widehat{\sigma'}) \cup labels(\widehat{\kappa'}) \cup labels(\widehat{v'_{\mathrm{C}}}) \end{aligned}$$

holds.

We will prove this invariant for all the rules of evaluating manifests. We will make an example for a rule where the definition will be changed in the evaluation. The proofs for the other rules are in the appendix.

Proof. Prove by induction.

Evaluation of Defined resource types

$$\frac{u \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{define}\ u\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}[u : \mathsf{ResourceDef}(\rho, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{RDef}$$

By definition, we know $labels(\texttt{define}\ u\ (\rho)\ \{s\}) = labels(\rho) \cup labels(s)$, and $labels(\texttt{skip}) = labels(\texttt{skip}) = \emptyset$. Then $labels(\widehat{\kappa}[u : \mathsf{ResourceDef}(\rho, s)]) = labels(\widehat{\kappa}) \cup labels(\mathsf{ResourceDef}(\rho, s)) = labels(\widehat{\kappa}) \cup labels(s) \cup labels(\rho)$. Thus we have $labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \cup labels(s) \cup labels(\rho) \supseteq labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}[u : \mathsf{ResourceDef}(\rho, s)]) \cup labels(\widehat{v_{\mathrm{C}}})$. That is $labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \cup labels(\texttt{define}\ u\ (\rho)\ \{s\}) \supseteq labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}[u : \mathsf{ResourceDef}(\rho, s)]) \cup labels(\widehat{v_{\mathrm{C}}})$. This rule holds w.r.t. the invariant.

□

4.2.3.3 Correctness of Where-Provenance

With these invariants in expression, statement and manifest evaluation, we can now state the relation between the data values in the input and the output when a manifest in μPuppet is evaluated. The correctness property states that given a μPuppet manifest m, the annotated data values in the final catalog resulting from evaluating m exist in the input, i.e. m. In the context of the semantics of μPuppet, the evaluation starts from m and the empty evaluation environments. The output of the evaluation will be the `skip` statement and the environments $\hat{\sigma}$, $\hat{\kappa}$ and $\hat{v_{\mathrm{C}}}$. Since Where-provenance means to define the origin of the data values in the catalog, we specify the correctness of where-provenance as all the annotated data values in the catalog $\widehat{v_{\mathrm{C}}}$ come from the input m. Where-provenance can be described in different ways. The correctness we characterise could imply the case that there is no labels on input data values as where-provenance. However, the labels in where-provenance we defined help to identify the original input data values of outputs by the annotations. If where-provenance is described in other ways, these descriptions could possess other properties that are not captured in our correctness.

The correctness we address is partial, since we exclude the manifests that do not terminate in this property.

Theorem 4.2.4 (Correctness of Where-Provenance)**.** When evaluating an annotated μPuppet manifest m such that $m \rightarrow^* \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}$, the subsumption on the sets of the annotated data values $labels(\widehat{v_{\mathrm{C}}}) \subseteq labels(m)$ holds.

Proof. Since $m \rightarrow^* \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}$, we know, for some n, there is an evaluation chain $m \rightarrow \widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{v_{\mathrm{C}1}}, m_1 \ldots \rightarrow \widehat{\sigma_n}, \widehat{\kappa_n}, \widehat{v_{\mathrm{C}n}}, m_n \rightarrow \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}$. Every evaluation step applies an evaluation rule. By Theorems 4.2.1, 4.2.2 and 4.2.3, we have $labels(m) \supseteq labels(m_1) \cup labels(\widehat{\sigma_1}) \cup labels(\widehat{\kappa_1}) \cup labels(\widehat{v_{\mathrm{C}1}}) \supseteq \ldots \supseteq labels(m_i) \cup labels(\widehat{\sigma_i}) \cup labels(\widehat{\kappa_i}) \cup labels(\widehat{v_{\mathrm{C}i}}) \supseteq \ldots \supseteq labels(m_n) \cup labels(\widehat{\sigma_n}) \cup labels(\widehat{\kappa_n}) \cup labels(\widehat{v_{\mathrm{C}n}}) \supseteq labels(\mathtt{skip}) \cup labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}})$. That is $labels(m) \supseteq labels(\mathtt{skip}) \cup labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}})$, i.e. $labels(m) \supseteq labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}})$. Then $labels(m) \supseteq labels(\widehat{v_{\mathrm{C}}})$. □

4.3 Expression-Provenance

Using where-provenance, the origin of the data values in the catalog can be identified. However the input data values that would take part in the primitive operations were not tracked so as to relate to the output data values due to these operations in where-provenance. The data values in the manifest that have empty annotations $\perp$ may be the results of the primitive operations in the evaluation. Where-provenance tells us nothing about how these results were produced, i.e. from which inputs in the manifest and by what computation procedure they have been generated. There have been such provenance models in the database field that explain how every tuple in the result of a query has been generated, called *how-provenance* as we introduced in the background chapter. The notion of how-provenance has been first introduced by Green et al. (2007b). In this model, a K-relation means a relational algebra with an annotation from the set K on every tuple in the relations. Green et al. found the relation between the generalised relational algebra and the algebraic structure $(K, 0, 1, +, \cdot)$. In this algebraic structure, every operator in the generalised relational algebra is mapped into computations in the algebraic structure. When the query is applied in the relational algebra, the computation on the annotations is represented as polynomials in the algebraic structure. The intuitive meaning of the polynomials is the explanation of how a tuple in the query result is generated, i.e. from which source tuples and by what operations. This algebraic structure is a semiring.

Acar et al. proposed a general provenance model for where-, how- and why-provenance built on a call-by-value, higher order, pure, functional language called Transparent ML(TML) (Acar et al., 2013). This language includes sums, products, and recursive types and functions. This language has been extended with traces. The tracing semantics was built in the big step semantics of TML such that the traces are closely corresponding to the semantics of the language. The traces can be seen as a general form of provenance. Then the other forms of provenance such as where-, how- and why- provenance can be extracted from the traces by functions. These functions defined the propagation of annotations in traces.

Inspired by these works, we will define expression-provenance in the context of configuration languages, in particular μPuppet. Expression-provenance means to provide the information of how a data value in the catalog has been generated

by operations. Since we care about the operation history of data values in the catalogs, we choose to start to label the data values in μPuppet in a similar way to it in how-provenance for databases. As the context of provenance in Acar et al.'s work, the context of our expression provenance is a configuration language μPuppet. We will build up our provenance in the semantics of μPuppet similarly to how traces have been defined. While TML is a pure functional programming language, μPuppet is mainly a declarative programming language with some imperative programming properties. So in TML, all the constructs are expressions, while in μPuppet, there are conditional constructs and the constructs for inheritance structure besides expressions. Traces record the computation information of all the expressions in TML. In contrast, for the expression-provenance in μPuppet, we only want to capture the primitive operations in the expressions. For this purpose, we will define the syntax of expression-provenance that will correspond to the expressions in μPuppet so as to reflect the respective computation. Then we will build up the semantics of expression-provenance in the semantics of μPuppet as how traces have been defined. After the definition of expression-provenance, we will establish the correctness of expression-provenance that will describe the purpose of expression-provenance in our context and then we will prove this correctness holds in our definition.

4.3.1 Annotated Compact Grammar

To define the annotated compact grammar of μPuppet for expression-provenance, we will start to look at what the forms of annotations for expressions-provenance should be in order to record the operations on the data values. The purpose of expression-provenance is to record the operations that would apply to the input data values in the process of generating output data values. The annotations should be able to record these operations. We will extend the form of annotations to the forms that will reflect the operations that would happen on the data values. Then we will define the annotated constructs in μPuppet inductively on their structures.

4.3.1.1 Syntax of Annotations

We will define the syntax of annotations in expression-provenance first. Since we want to record the operations on the data values, we will annotate the data

values with unique labels l as in where-provenance and then extend the form of annotations respecting to the operations in the expressions in μPuppet. Beside the primitive operations, there are selector expressions. A selector expression is executed as a conditional statement such that it chooses one of the branches in the selector body to be executed in the evaluation. Since there is no computation applied to the data values, we will not define a form of annotation that records the evaluation of selector expressions. The syntax of annotations for expression-provenance is as below.

$$\beta ::= l \mid \beta + \beta \mid \beta - \beta \mid \beta > \beta \mid \beta = \beta \mid \beta \leq \beta \mid \beta \vee \beta \mid \ ! \ \beta \mid \ldots$$

4.3.1.2 Annotated syntax of μPuppet

With the extended annotations β, we will label the constructs in μPuppet. We will continue to use the form of annotated syntax of μPuppet in where-provenance. The data values v will be annotated first, noted as $\hat{v}$. The other expressions including the annotated data values will be annotated inductively on their structures. The notations of these expressions are the same as those in where-provenance. Then the statements and the manifests will be defined accordingly and their notations are the same as in where-provenance too. All the environments σ, κ and v_{C} in the evaluation rules will be defined similarly and their notations are the same as in where-provenance. We omit their syntax here.

4.3.2 Propagation of Annotations in the Evaluation

With the annotated compact grammar of μPuppet for expression-provenance, we will define the semantics of the annotations. Similarly to the semantics of the traces in Acar et al.'s work, we will define the semantics of annotations by how they are propagated in the evaluation. So we will define in every evaluation rule how the annotations in the context of evaluation will be propagated.

4.3.2.1 Propagation of Annotations in Expression Evaluation

The annotations are designed to record the operations on the data values. These operations happen in the expressions. We will use an example to show how the annotations on the data value record an operation. Take a binary operation $+$ in an expression $i_1 + i_2$ as an example. In the annotated syntax, i_1 and i_2 will be

labelled with β_1 and β_2. When evaluating the expression $i_1^{\beta_1} + i_2^{\beta_2}$, we will assign the annotation $\beta_1 + \beta_2$ to the result of the expression $i_1 +_{\mathbb{Z}} i_2$. So the annotated output data value will be $(i_1 +_{\mathbb{Z}} i_2)^{\beta_1+\beta_2}$. The annotated evaluation rule is as below.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, i_1^{\beta_1} + i_2^{\beta_2} \xrightarrow{\alpha} (i_1 +_{\mathbb{Z}} i_2)^{\beta_1+\beta_2}} \textsc{ArithValue}$$

As for the other generation forms of + expressions, their annotated evaluation rules are as below. To evaluate the expression in the form $i^{\beta} + e$, the rule ArithRight will evaluate the expression e to e'. The annotated data value i^{β} will remain in the result of the evaluation.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 + e_2 \xrightarrow{\alpha} e_1' + e_2} \textsc{ArithLeft} \qquad \frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, i^{\beta} + e \xrightarrow{\alpha} i^{\beta} + e'} \textsc{ArithRight}$$

The propagation of annotations in the rules evaluating $-$,$*$ and / expressions is similar to those for + expressions. We omit to show these propagation rules here.

As another example in comparison expressions, we take > expressions. When evaluating an annotated > expression $i_1^{\beta_1} > i_2^{\beta_2}$, we will assign the annotation $\beta_1 > \beta_2$ to the result data value `true` or `false`. Notice that if $i_1 > i_2$ is false, the annotation on the result `false` will be the same as if the result is true since the annotation means to record the operation on the input values that derives a result data value, not the computational relation on the input data values.

$$\frac{i_1 >_{\mathbb{Z}} i_2}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, i_1^{\beta_1} > i_2^{\beta_2} \xrightarrow{\alpha} \texttt{true}^{\beta_1 > \beta_2}} \textsc{CompValueI}$$

$$\frac{i_1 \leq_{\mathbb{Z}} i_2}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, i_1^{\beta_1} > i_2^{\beta_2} \xrightarrow{\alpha} \texttt{false}^{\beta_1 > \beta_2}} \textsc{CompValueII}$$

The evaluation rules for the other forms of > expressions will propagate the annotations on the data values as the following. The evaluation on the expression $i^{\beta} > e$ will evaluate e to e' in the rule CompRight. The annotated data value i^{β} will be retained after the rule is applied.

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e_1 > e_2 \xrightarrow{\alpha} e_1' > e_2}\ \textsc{CompLeft} \qquad \frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},i^{\beta} > e \xrightarrow{\alpha} i^{\beta} > e'}\ \textsc{CompRight}$$

The evaluation rules for other comparison operators will propagate the annotations in a similar way. We will omit them here.

As for the selector expressions, there will be no immediate computation on the data values to get an output data value. Instead, there is a control flow that will choose or omit the current branch depending on the control expressions. So there is no annotation brought up when evaluating a selector expression. Since the control data values take part in deciding the part of the manifests to evaluate further, but do not resolve to some output data values in the catalogs, expression-provenance does not track these control data values. So the annotations on the control data values will not be carried in the further evaluation. The annotated evaluation rules of choosing or omitting a branch in a selector expression are as below. The other rules are in the appendix.

$$\frac{\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\} \xrightarrow{\alpha} e}\ \textsc{SChoose}$$

$$\frac{\neg\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\} \xrightarrow{\alpha} v^{\beta}\ ?\{M\}}\ \textsc{SChooseI}$$

4.3.2.2 Propagation of Annotations in Resource Evaluation

Evaluating a resource involves the evaluation of expressions appearing as resource title and the assignments of the attributes. The evaluation of expressions will be lifted up to the expression level of evaluation. There is no direct operation brought by the resource construct to the data values. So there is no annotation generation caused by the resource construct directly. Moreover there is also no control flow in the evaluation of a resource. A resource will evolve to a resource value in the catalog. The form of annotated evaluation rules is the same as the where-provenance. We omit them here.

4.3.2.3 Propagation of Annotations in Statement Evaluation

When evaluating a statement s, there is no direct operation that applies on the data values. Instead, the rule will lift the evaluation of an expression appearing in s to the evaluation of the expression level. In some statements, there is a control flow involved that will choose one branch in the body of the statements. Such statements are if and case statements. So the evaluation rules for statements will not generate the annotations for expression-provenance that means to records operations on the data values. In the rules for the statements involving control flows, the annotations on the control expressions and in the branches that would not be chosen will be not remained in the further evaluation as in where-provenance. So the forms of the evaluation rules for statement will be the same as them in where-provenance. We will omit them here. We will see the example of annotation propagation in if and case evaluation rules.

If statement After the control expression e is evaluated to $\texttt{true}^\beta$ or $\texttt{false}^\beta$, either the rule IFT or IFF will be applied. Either the branch s_1 or s_2 will be chosen to remain in the evaluation, while the control truth value with its annotation will not be discarded in the evaluation.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{if } e \texttt{ } \{s_1\} \texttt{ else } \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{if } e' \texttt{ } \{s_1\} \texttt{ else } \{s_2\}} \textsc{IfStep}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{if } \texttt{true}^\beta \texttt{ } \{s_1\} \texttt{ else } \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s_1} \textsc{IfT}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{if } \texttt{false}^\beta \texttt{ } \{s_1\} \texttt{ else } \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s_2} \textsc{IfF}$$

Case Similarly to if statements, the evaluation of case statements involves control flows. After the two control expressions have been evaluated to the data values, the current branch in the case body will be retained or omitted, depending on whether the two control data values match or not. The annotation on the control data value in the current branch will be always omitted with the data value. After one branch has been chosen, the control data value v^β of the case statement will be omitted too.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{case}\ e\ \{C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{case}\ e'\ \{C\}}\ \textsc{CaseStep1}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{case}\ v^{\beta}\ \{e : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{case}\ v^{\beta}\ \{e' : \{s\} \llcorner C\}}\ \textsc{CaseStep2}$$

$$\frac{\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s}\ \textsc{CaseMatch}$$

$$\frac{\neg\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{case}\ v^{\beta}\ \{C\}}\ \textsc{CaseNoMatch}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{case}\ v^{\beta}\ \{\varepsilon\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{CaseDone}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{case}\ v^{\beta}\ \{\texttt{default} : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s\}}\ \textsc{CaseDefault}$$

Similarly, the annotation propagation in the rules for other constructs at the statement level such as sequential composition, expressions, `unless`, include, resource-like class declaration, resource, defined resource types and scope statements are the same in where-provenance. So the annotated evaluation rules have the same form as in where-provenance.

4.3.2.4 Propagation of Annotations in Manifest Evaluation

Evaluating a construct at the manifest level, such as *statements*, *class*, *node*, *defined resource types* and *sequential* elements, will not bring new operations on the data values, except the expressions inside the construct. There are also no control flows in the evaluation. So the evaluation of expressions in the manifest construct will be lifted up to that at expression level. So there are no immediate changes on annotations on the data values in the evaluation rules for manifest constructs. The annotated evaluation rules will be the same as in the where-provenance. We omit these rules here.

4.3.3 Correctness of Expression-Provenance

We will adapt the definition of the correctness of expression provenance in Acar et al's work. In this work, a function h was defined to map the locations to values. The locations annotate the values in every expression construct in TML. For any annotation term t, $h(t)$ means to evaluate t to a value by substituting the locations appearing in t with the corresponding values by h. There is a function $occ()$ similar as $labels()$, taking an annotated expression and returning the set of annotated constants appearing in the expression. It is called that h is *consistent* with $\hat{v}$ if whenever $w^t \in occ^{\perp}(\hat{v})$, $h(t) = |w|$ (not $\perp$ here) holds. The correctness of expression-provenance stated that if h is consistent with the annotated environment $\hat{\gamma}$ then h is consistent with the expression-provenance on a trace T and $\hat{\gamma}$. With the help of function h, the correctness shows the expression-provenance indeed represents the operation history of an output data value.

In the light of the method of defining the correctness in this work, we will define our correctness for our syntax and semantics of expression-provenance for μPuppet, the expression-provenance for μPuppet is defined to record the operations that have happened so to derive an output data. So we want to show that expression-provenance indeed reflects the history of operations on data values. This correctness is comparable to that in TML setting in Acar et al.'s work. By using function h that determines a data value by a location in the original input data value, the correctness of expression-provenance can be formalized. We will adapt this definition in our setting of expression-provenance as a relation that could define an annotation and a data value uniquely. Then we will use this relation to define the correctness for our expression-provenance.

4.3.3.1 Consistency Relation

We define a *labelling function* μ that relates a label l from a set of labels to a data value in the manifest as below. By labelling the data values in a manifest by a function μ, every data value is differentiated by an unique label l.

$$\mu : labels \rightarrow values$$

Then we will build up the unique relation between the data values and their annotations by function μ. By the definition of expression provenance in the

$$\frac{\mu(l) = v}{\mu \models l \backsim v}\ \textsc{Label} \qquad \frac{}{\mu \models \bot \backsim v}\ \textsc{Bottom} \qquad \frac{\mu \models \beta_1 \backsim v_1 \qquad \mu \models \beta_1 \backsim v_1}{\mu \models \beta_1 + \beta_2 \backsim v_1 +_{\mathbb{Z}} v_2}\ \textsc{Add}$$

$$\frac{\mu \models \beta_1 \backsim v_1 \qquad \mu \models \beta_2 \backsim v_2}{\mu \models \beta_1 > \beta_2 \backsim v_1 >_{\mathbb{Z}} v_2}\ \textsc{CompLar} \qquad \frac{\mu \models \beta_1 \backsim b_1 \qquad \mu \models \beta_2 \backsim b_2}{\mu \models \beta_1 \wedge \beta_2 \backsim b_1 \wedge_{\mathbb{B}} b_2}\ \textsc{And}$$

$$\frac{\mu \models \beta \backsim \texttt{true}}{\mu \models\ !\,\beta \backsim \texttt{false}}\ \textsc{Not1} \qquad \frac{\mu \models \beta \backsim \texttt{false}}{\mu \models\ !\,\beta \backsim \texttt{true}}\ \textsc{Not2}$$

Figure 4.10: Consistency Relation

last section, the annotation on a data value records the operations that have happened. We know that the annotation on a data value should also uniquely correspond to the input data values and the operation applies to them. We then can relate the data value and its annotation under the assumption of a function μ. We define the form of this relation as $\mu \models \beta \backsim v$, called *consistency relation*. We can also say an annotation β is *consistent with* a data value v. We will define this relation by induction rules that correspond to all the operations on data values in μPuppet. The definition of consistency relation is as figure 4.10.

These induction rules have premises above the line and a conclusion under the line, which indicates that given the premises hold, the conclusion is derived. Rule Label says the consistency relation between a label l and a value v under function μ is defined given $\mu(l) = v$. Rule Bottom means a bottom label and any data value hold for consistency relation. Rule Addition shows if β_1 is consistent with v_1 and β_2 is consistent with v_2 then the annotation $\beta_1 + \beta_2$ is consistent with the result data value of addition operation on v_1 and v_2. Rule CompLar defines the consistency relation derived from $>$ operation. In CompLar, if β_1 is consistent with v_1, β_2 is consistent with v_2, then the consistency relation between the annotation $\beta_1 > \beta_2$ and the result data value $v_1 >_{\mathbb{Z}} v_2$ (or the truth value) is defined. Similarly, rule Not1 and Not2 define the consistency relation for the annotations of the form $!\beta$. Rule Not1 tells that if an annotation β is consistent with `true` then the annotation $!\beta$ is consistent with the result of !`true`, i.e. a new data value `false`. Then rule Not2 defines the consistency relation between $!\beta$ and the result of !`false`, i.e. a new data value `true`. The induction rules to

$$
\begin{aligned}
\mu \vDash \quad e \quad &\Leftrightarrow \forall v^{\beta} \in labels(e),\ \mu \models \beta \backsim v \\
\mu \vDash \quad s \quad &\Leftrightarrow \forall v^{\beta} \in labels(s),\ \mu \models \beta \backsim v \\
\mu \vDash \quad m \quad &\Leftrightarrow \forall v^{\beta} \in labels(m),\ \mu \models \beta \backsim v \\
\mu \vDash \quad \widehat{\sigma} \quad &\Leftrightarrow \forall v^{\beta} \in labels(\widehat{\sigma}),\ \mu \models \beta \backsim v \\
\mu \vDash \quad \widehat{\kappa} \quad &\Leftrightarrow \forall v^{\beta} \in labels(\widehat{\kappa}),\ \mu \models \beta \backsim v \\
\mu \vDash \quad \widehat{v_{\mathrm{C}}} \quad &\Leftrightarrow \forall v^{\beta} \in labels(\widehat{v_{\mathrm{C}}}),\ \mu \models \beta \backsim v
\end{aligned}
$$

Figure 4.11: Extended Consistency Relation

define the consistency relation on the form of annotations with other operations such as -, $*$, /, = or, $\vee$, with their corresponding result data values are defined similarly. We omit them here.

Extension of Consistency Relation We have defined the relation between the generated data values by operations and their annotations. Since the expressions of the operations on data values are carried in the constructs in μPuppet, we will extend the consistency relation $\mu \models \beta \backsim v$ to all the other constructs in μPuppet. We will use the extraction functions defined in where-provenance to extract all the data values appearing in a construct. Then we will establish the consistency relation for these data values and their annotations so that the consistency relation for this construct is established. The extended consistency relation for expressions, statements, manifest elements and environments is in figure 4.11.

4.3.3.2 Invariants in Evaluation with Annotation Propagation and Correctness

The correctness of expression-provenance should state that the expression-provenance records all the operations through which the data values have been in the catalog. Since the consistency relation defines a data value and its annotation according to the operation applied to the input data values, we will use it to formalise the correctness. To show the correctness of expression-provenance is to prove if an annotation or label l is consistent with a data value in a manifest, after evaluating the manifest, the data values and their annotations in the catalog also hold for the

consistency relation. If this statement holds, it would justify that the semantics of expression-provenance we have defined indeed reflects the operations on the data values over the evaluation. For this purpose, as in where-provenance, we will present this property as invariants in three levels of evaluation for μPuppet. Then we will establish the correctness of expression-provenance by these invariants.

Invariant in Expression Evaluation The evaluation of expression follows the judgement as below.

$$\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'$$

The operations on data values expression-provenance should record happen in this level of evaluation.

Expression-provenance means to record the operations that have happened to the data values. The operations on data values are carried in expressions. So the evaluation of expressions with operations and the annotation evolution to record these operations happen in the level of expression evaluation. We would like to show that if the input data values of an operation in an expression and their annotations follow the consistency relation, the result data value and its annotation after evaluating an expression also follow this relation. This would prove the correctness property of expression-provenance. We would instead prove this property holds in every evaluation rule for expressions. For a variable in the manifest, its assignment is from the annotated store $\widehat{\sigma}$. Since the input data values of expressions could come from the environment store and catalog, we need to show the data values and their annotations in the environments also follow the consistency relation. The invariant in every rule for expression evaluation is formalised as below.

Theorem 4.3.1 (Invariant in Expression Evaluation)**.** For any $\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'$, if $\mu \models \widehat{\sigma}$, $\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models e$, then $\mu \models e'$.

This invariant has been proved for all the semantic rules for expressions.

Proof. The invariant is proved by induction.

1 Arithmetic expressions

Case (1) $\dfrac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, i_1^{\beta_1} + i_2^{\beta_2} \xrightarrow{\alpha} (i_1 +_{\mathbb{Z}} i_2)^{\beta_1+\beta_2}}$ ArithValue

We have $labels(i_1^{\beta_1} + i_2^{\beta_2}) = \{i_1^{\beta_1}, i_2^{\beta_2}\}$ and $labels((i_1 +_{\mathbb{Z}} i_2)^{\beta_1+\beta_2}) = \{(i_1 +_{\mathbb{Z}} i_2)^{\beta_1+\beta_2}\}$ by definition. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models i_1^{\beta_1} + i_2^{\beta_2}$. That is $\forall i^{\beta} \in labels(i_1^{\beta_1} + i_2^{\beta_2})\ \mu \models \beta \backsim i$. Then $\forall i^{\beta} \in \{i_1^{\beta_1}, i_2^{\beta_2}\}$, $\mu \models \beta \backsim i$, i.e. $\mu \models \beta_1 \backsim i_1$ and $\mu \models \beta_2 \backsim i_2$. By the inference rule, we have

$$\frac{\mu \models \beta_1 \backsim i_1 \quad \mu \models \beta_2 \backsim i_2}{\mu \models \beta_1 + \beta_2 \backsim i_1 +_{\mathbb{Z}} i_2}$$

. Then $\forall i^{\beta} \in \{(i_1 +_{\mathbb{Z}} i_2)^{\beta_1+\beta_2}\}$, $\mu \models (i_1 +_{\mathbb{Z}} i_2)^{\beta_1+\beta_2}$, i.e. $\mu \models (i_1 +_{\mathbb{Z}} i_2)^{\beta_1+\beta_2}$. Then we have if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models i_1^{\beta_1} + i_2^{\beta_2}$, then $\mu \models (i_1 +_{\mathbb{Z}} i_2)^{\beta_1+\beta_2}$.

Case (2) $\dfrac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, i^{\beta} + e \xrightarrow{\alpha} i^{\beta} + e'}$ AritHRight

By induction hypothesis, we know if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models e$, then $\mu \models e'$. That is if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$, and $\forall i^{\beta} \in labels(e)\ \mu \models \beta \backsim i$ then $\forall i'^{\beta'} \in labels(e')\ \mu \models \beta' \backsim i'$. We know $labels(i^{\beta} + e) = \{i^{\beta}\} \cup labels(e)$ and $labels(i^{\beta} + e') = \{i^{\beta}\} \cup labels(e')$ by definition. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models i^{\beta} + e$. That is $\forall i^{\beta} \in labels(i^{\beta} + e)\ \mu \models \beta \backsim i$, i.e. $\forall i^{\beta} \in \{i^{\beta}\} \cup labels(e)\ \mu \models \beta \backsim i$. We have if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$, and $\forall i^{\beta} \in labels(e) \cup \{i^{\beta}\}\ \mu \models \beta \backsim i$ then $\forall i^{\beta} \in labels(e') \cup \{i^{\beta}\}\ \mu \models \beta \backsim i$, i.e. $\forall i^{\beta} \in labels(e' + i^{\beta})\ \mu \models \beta \backsim i$. That is if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models i^{\beta} + e$ then $\mu \models i^{\beta} + e'$.

2 Variables

$$\frac{x \in dom(\sigma_{\alpha})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x \xrightarrow{\alpha} \sigma_{\alpha}(x)}\ \textsc{LVar}$$

Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \$x$. We know by definition $labels(\hat{\sigma}_{\alpha}(x)) \subseteq labels(\hat{\sigma})$ for some x. Then if $\mu \models \hat{\sigma}$, i.e. $\forall v^{\beta} \in labels(\hat{\sigma})\ \mu \models \beta \backsim v$, then $\forall v^{\beta} \in labels(\hat{\sigma}_{\alpha}(x))\ \mu \models \beta \backsim v$, i.e. $\mu \models \hat{\sigma}_{\alpha}(x)$. Then we have if $\mu \models \$x$, $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$ and $\mu \models \hat{v_{\mathrm{C}}}$ then $\mu \models \hat{\sigma}_{\alpha}(x)$.

3 Selector

$$\frac{\mathsf{caseMatch}(v, v_1)}{\sigma, \kappa, v_{\mathrm{C}}, v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\} \xrightarrow{\alpha} e}\ \textsc{SChoose}$$

Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\}$. We have $\forall v'^{\beta'} \in labels(v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\})\ \mu \models \beta' \backsim v'$ by definition. We know $labels(v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\}) = labels(v^{\beta}) \cup labels(v_1^{\beta_1}) \cup labels(e) \cup labels(M)$ by definition. Since $labels(v^{\beta}) \cup labels(v_1^{\beta_1}) \cup labels(e) \cup labels(M) \supseteq labels(e)$, we have if $\forall v'^{\beta'} \in labels(v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\})\ \mu \models \beta' \backsim v'$ then $\forall v'^{\beta'} \in labels(e) \mu \models \beta' \backsim v'$. That is if $\mu \models v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\}$ then $\mu \models e$. Then we have if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\}$ then $\mu \models e$.

□

Invariant in Statement Evaluation In the evaluation of a statement, a data value in the statement or any environment might join the evaluation. Meanwhile, a new statement s' will be generated and the environments might be changed to new environments where new data values might be included. The output of an evaluation step will be the input of the next step evaluation. So we will consider the data values in all the inputs and outputs for consistency relation to establish the invariant in the evaluation. The invariant for any rule in statement evaluation for expression-provenance will be stated as below.

Theorem 4.3.2 (Invariant in Statement Evaluation)**.** For any $\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha}_s \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}} \vdash s'$, if $\mu \models \widehat{\sigma}$, $\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models s$ then $\mu \models \widehat{\sigma'}$, $\mu \models \widehat{\kappa'}$, $\mu \models \widehat{v'_{\mathrm{C}}}$ and $\mu \models s'$.

We will prove this theorem for every rule of statement evaluation. The observation is that there is no evaluation on expression level that will happen directly in the rules for statement evaluation. The expressions included in the statements will be lifted up to the evaluation of expression level. There will be conditional statements that includes few branches in these statements. The result of the evaluation will only choose one of the branches. Since the expression-provenance means to remember the operations in expressions on data values, the annotations appearing in the branches that are not chosen after evaluating the conditional statements beside the annotations on the control data values will not remain in the evaluation to shape the expression-provenance. We will choose some evaluation rules for conditional statements that illustrate our observation to prove this

invariant. We will also prove the invariant for some rule that involves updating environment. Its proofs for other rules are shown in appendix.

Proof. Prove by induction.

1 Evaluation of If statement

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},s_1}\ \textsc{IfT}$$

Suppose $\mu \models \hat{\sigma}$, $\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models \texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\}$. By definition, we have $labels(\texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\}) = labels(\texttt{true}^{\beta}) \cup labels(s_1) \cup labels(s_2) = \{\texttt{true}^{\beta}\} \cup labels(s_1) \cup labels(s_2)$. We have $\forall v^a \in labels(\texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\})\ \mu \models \beta \backsim v$ by definition. That is $\forall v^{\beta} \in \{\texttt{true}^{\beta}\} \cup labels(s_1) \cup labels(s_2)\ \mu \models \beta \backsim v$. Then we have $\forall v^{\beta} \in labels(s_1),\ \mu \models \beta \backsim v$, i.e. $\mu \models s_1$. Then we have if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models \texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\}$ then $\mu \models s_1$.

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{if false}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},s_2}\ \textsc{IfF}$$

Suppose $\mu \models \hat{\sigma}$, $\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models \texttt{if false}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\}$. By definition, we have $labels(\texttt{if false}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\}) = labels(\texttt{false}^{\beta}) \cup labels(s_1) \cup labels(s_2) = \{\texttt{false}^{\beta}\} \cup labels(s_1) \cup labels(s_2)$. We have $\forall v^a \in labels(\texttt{if false}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\})\ \mu \models \beta \backsim v$ by definition. That is $\forall v^{\beta} \in \{\texttt{false}^{\beta}\} \cup labels(s_1) \cup labels(s_2)\ \mu \models \beta \backsim v$. Then we have $\forall v^{\beta} \in labels(s_2),\ \mu \models \beta \backsim v$, i.e. $\mu \models s_2$. Then we have if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models \texttt{if false}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\}$ then $\mu \models s_2$.

2 Evaluation of Case statement

$$\frac{\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \_ C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},s}\ \textsc{CaseMatch}$$

Suppose $\mu \models \hat{\sigma}$, $\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models \texttt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \_ C\}$. By definition, we have $labels(\texttt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \_ C\}) = labels(v^{\beta}) \cup labels(v_1^{\beta_1}) \cup labels(s) \cup labels(C)$. Then we have $\forall v'^{\beta'} \in labels(\texttt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \_ C\})\ \mu \models \beta' \backsim v'$ by definition, i.e. $\forall v'^{\beta'} \in labels(v^{\beta}) \cup labels(v_1^{\beta_1}) \cup labels(s) \cup labels(C)\ \mu \models \beta' \backsim v'$. We have $\forall v'^{\beta'} \in labels(s)\ \mu \models \beta' \backsim v'$, i.e. $\mu \models s$. Then if $\mu \models \hat{\sigma}$,

$\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models \mathtt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \llcorner C\}$ then $\mu \models s$.

$$\frac{\neg \mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{case}\ v^{l}\ \{C\}}\ \textsc{CaseNoMatch}$$

The proof is similar as case above.

3 Evaluation of Resource-like class declarations

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \quad \widehat{\kappa}(b) = \mathsf{DeclaredClass}(\beta) \quad s' = \mathit{merge}(\rho, v_{\mathrm{H}})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}[a : \mathsf{DeclaredClass}(::b)], \widehat{v_{\mathrm{C}}}, \mathtt{scope}\ (::a)\ \{s' \llcorner s\}}\ \textsc{CDecPD}$$

Suppose $\mu \models \hat{\sigma}$, $\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models \mathtt{class}\ \{a : v_{\mathrm{H}}\}$. We have $\mathit{labels}(\mathtt{class}\ \{a : v_{\mathrm{H}}\}) = \mathit{labels}(v_{\mathrm{H}})$ by definition. We have $\forall v^{\beta} \in \mathit{labels}(v_{\mathrm{H}}), \mu \models \beta \backsim v$. Since $\mu \models \kappa$, we have $\forall v^{\beta} \in \mathit{labels}(\widehat{\kappa}), \mu \models \beta \backsim v$. Then we have $\forall v^{\beta} \in \mathit{labels}(\widehat{\kappa}(a)), \mu \models \beta \backsim v$. That is $\forall v^{\beta} \in \mathit{labels}(\rho) \cup \mathit{labels}(s), \mu \models \beta \backsim v$ by definition. We have $\forall v^{\beta} \in \mathit{labels}(v_{\mathrm{H}}) \cup \mathit{labels}(\rho) \cup \mathit{labels}(s), \mu \models \beta \backsim v$. Then $\forall v^{\beta} \in \mathit{labels}(s') \cup \mathit{labels}(s), \mu \models \beta \backsim v$. Since we know $\mathit{labels}(\mathtt{scope}\ (::a)\ \{s' \llcorner s\}) = \mathit{labels}(s') \cup \mathit{labels}(s)$ by definition, we have $\mu \models \mathtt{scope}\ (::a)\ \{s' \llcorner s\}$. We also know if $\forall v^{\beta} \in \mathit{labels}(\widehat{\kappa}), \mu \models \beta \backsim v$ and $\mathit{labels}(\mathsf{DeclaredClass}(b)) = \emptyset$ then $\forall v^{\beta} \in \mathit{labels}(\widehat{\kappa}[a : \mathsf{DeclaredClass}(::b)]), \mu \models \beta \backsim v$. Then we have if $\mu \models \hat{\sigma}$, $\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models \mathtt{class}\ \{a : v_{\mathrm{H}}\}$ then $\mu \models \hat{\sigma}$, $\mu \models \widehat{\kappa}[a : \mathsf{DeclaredClass}(::b)]$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models \mathtt{scope}\ (::a)\ \{s' \llcorner s\}$.

The proofs for other evaluation rules for statements are similar and shown in Appendix C.

□

Invariant in Manifest Evaluation Similarly to statement evaluation, evaluating a manifest element m might involve employing the data values from the input constructs of the evaluation and the updating of data values in them, i.e. m and the environments $\widehat{\sigma}$,$\widehat{\kappa}$ and $\widehat{v_{\mathrm{C}}}$. So to establish the invariant of expression-provenance in every step in manifest evaluation, we will consider the data values in all these constructs contributing to the manifest evaluation. The invariant is formalised as below.

Theorem 4.3.3 (Invariant in Manifest Evaluation)**.** For any $\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, m \xrightarrow{N}_{m} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}} \vdash m'$, if $\mu \models \widehat{\sigma}$, $\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models m$ then $\mu \models \widehat{\sigma'}$, $\mu \models \widehat{\kappa'}$, $\mu \models \widehat{v'_{\mathrm{C}}}$ and $\mu \models m'$.

We will prove this invariant for every rule in manifest evaluation. Similarly to statement evaluation, we have the observation that there are no operations on data values directly in the evaluation of manifests. There are statements and the definitions of classes, nodes and resource defined types as manifest elements. The evaluation of statements will be lifted up to the evaluation at the statement level. The evaluations of the definitions will involve the environment update. We will make an example of proving this invariant in evaluating such a definition. The proofs of the invariant for other rules are in appendix.

Proof. Prove by induction.

Evaluation of Defined resource types

$$\frac{u \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{define}\ u\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}[u : \mathsf{ResourceDef}(\rho, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{RDef}$$

Suppose $\mu \models \hat{\sigma}$, $\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models \texttt{define}\ u\ (\rho)\ \{s\}$. We have $labels(\texttt{define}\ u\ (\rho)\ \{s\}) = labels(\rho) \cup labels(s)$ and $labels(\texttt{skip}) = \emptyset$ by definition. Since $\widehat{\kappa}[u : \mathsf{ResourceDef}(\rho, s)] = \widehat{\kappa} \cup \{u \to \mathsf{ResourceDef}(\rho, s)\}$, $labels(\widehat{\kappa}[u : \mathsf{ResourceDef}(\rho, s)]) = labels(\widehat{\kappa}) \cup labels(\mathsf{ResourceDef}(\rho, s))$. Since $\mu \models \widehat{\kappa}$ and $\mu \models \texttt{define}\ u\ (\rho)\ \{s\}$, we have $\forall v^{\beta} \in labels(\widehat{\kappa}), \mu \models \beta \backsim v$ and $\forall v^{\beta} \in labels(\rho) \cup labels(s), \mu \models \beta \backsim v$. Then $\forall v^{\beta} \in labels(\widehat{\kappa}) \cup labels(\mathsf{ResourceDef}(\rho, s))$, i.e. $\mu \models \widehat{\kappa}[u : \mathsf{ResourceDef}(\rho, s)]$. Then if $\mu \models \hat{\sigma}$, $\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models \texttt{define}\ u\ (\rho)\ \{s\}$ then $\mu \models \hat{\sigma}$, $\mu \models \widehat{\kappa}[u : \mathsf{ResourceDef}(\rho, s)]$, $\mu \models \widehat{v_{\mathrm{C}}}$.

We omit the proofs of other manifest rules here and show them in Appendix C.

□

4.3.3.3 Correctness of Expression-Provenance

Having defined the three invariants for expression-provenance in expression, statement and manifest evaluation, we will specify the correctness property of expression-provenance. The correctness property states the data values in the catalog will

record the operations experienced in the evaluation of the manifest. To show it, we will use the consistency relation that relates a data value and its annotation according to the operations uniquely. Given a labelling function μ on the data values in a manifest, if the data values and their annotations after the evaluation in the catalog still keep the consistency relation, we know the annotations record the operations through the evaluation. Similar as the correctness of where-provenance, we only consider the correctness of expression-provenance of the manifests that terminate. The correctness we address here is partial. We formalise it as below.

Theorem 4.3.4 (Correctness of Expression-Provenance)**.** Given a labelling function μ, if $m \to^* \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}$ and $\mu \models m$ then $\mu \models \widehat{v_{\mathrm{C}}}$.

Proof. Since $m \to^* \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}$, we know that for some n there is an evaluation train $m \to \widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{v_{\mathrm{C}1}}, m_1 \to \ldots \to \widehat{\sigma_n}, \widehat{\kappa_n}, \widehat{v_{\mathrm{C}n}}, m_n \to \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}$. Given a function μ, we have $\mu \models m$ by assumption. By theorem 4.3.1, 4.3.2 and 4.3.3 we have $\mu \models m \to (\mu \models \widehat{\sigma_1} \wedge \mu \models \widehat{\kappa_1} \wedge \mu \models \widehat{v_{\mathrm{C}1}} \wedge \mu \models m_1) \to \ldots \to (\mu \models \widehat{\sigma_n} \wedge \mu \models \widehat{\kappa_n} \wedge \mu \models \widehat{v_{\mathrm{C}n}} \wedge \mu \models m_n) \to (\mu \models \hat{\sigma} \wedge \mu \models \widehat{\kappa} \wedge \mu \models \widehat{v_{\mathrm{C}}} \wedge \mu \models \mathtt{skip})$. Thus we have $\mu \models \widehat{v_{\mathrm{C}}}$. □

Chapter 5

Dependency Provenance

In this chapter, we will define another form of provenance which will record the dependency information. We call such provenance *dependency provenance*. While where-provenance records the locations of input data values that the output data values copy from and expression-provenance provides the computational information on primitive operations about the output data value, there are other computational factors such as conditional constructs in μPuppet that influence how an output data value is resolved. Dependency provenance intends to collect all the input data that could influence a final output data value in the catalog after compilation in all possible ways. In this chapter we will first review the notion of dependency and its correctness technique established in the provenance field. Then we will build the syntax and semantics of dependency provenance. Moreover we will propose our characterization of its correctness and prove the correctness of dependency-provenance. However due to the complexity of μPuppet language, dependency provenance does not record all the dependency information of a data value in some unusual manifests. We will also discuss the limitation of our method and propose the possible solutions for the unusual cases.

5.1 Background

Dependency analysis technique has been explored in program slicing and information flow analysis. As Abadi et al. (1999) pointed out, program slicing, information flow and several other programming techniques can be made uniform by considering them as dependence. Cheney et al. (2011) argued that these dependency techniques also provide a suitable foundation for a class of prove-

nance. They proposed a model of dependency provenance in the context of a core database query language. In contrast to program slicing that separates the part of a program responsible for an output result, their dependency provenance model established *data dependence* that extracted the input data in the database that could influence an output data in the result of a query.

In their work, the dependency provenance model was made on the nested relational calculus (NRC) (Buneman et al., 1995). NRC is a typed functional language. Its types include collection types. Dependency provenance was defined to provide what input data in the database are responsible for one part of the output data in the query result. The provenance model used sets of *colors* as annotations for the data. This technique of annotations on data models has been developed in the work on program analysis (Nielson et al., 1999) and on provenance (Buneman et al., 2008; Wang and Madnick, 1990). Different operations in NRC were treated as functions. To define provenance information, functions have been generated operating on annotated data values. Provenance semantics then was formed by defining annotation propagations through the generalised functions. Having the semantics of dependency provenance, the work further proposed the correctness criterion of dependency provenance. This criterion characterised whether the annotations generated by these functions describe the dependency information introduced in any ordinary function in NRC. This correctness was established in the way that the annotations captured the input data of which any change might influence some part of the output of a query. This work proposed two correctness criteria for both dynamic and static provenance tracking.

As we discussed, where- and expression- provenance as we have defined them do not cover all the computational behaviours that could happen in μPuppet. There are conditional constructs by which more than one branch of computation could be chosen. Where- and expression- provenance only tell us the provenance information of the output data that might have been chosen by conditional constructs under the current input data values but do not inform us why this has happened. We also want to capture such conditional computational information that could describe an output in the catalog. Intuitively, such information could be provided by dependency provenance that captures all the input data values that could affect an output data value. Due to the computational similarity between configuration languages and queries in the database, we will adapt the

technique of defining dependency provenance in Cheney et al.'s work to design our dependency provenance and characterise its correctness.

Due to the complexity of μPuppet, we will consider of a subset of μPuppet to formalise dependency-provenance and prove its correctness. The subset does not include the data structures array, hash and resource reference. We believe it will be easy to extend our formalization to them.

5.2 Annotations and Propagation of Annotations

5.2.1 Annotated Compact Grammar

To track dependency information in the computation, we choose to only annotate the data values as in Cheney et al.'s work. We are concerned with the input data values that might influence the output data but not computational process. Starting from annotating the input data values with a single label, the annotations on the output data values should collect all the annotations of the influential input data values so to represent the dependency information. Thus we define the annotations for dependency provenance as a set of unique labels. The syntax of the annotations is as below.

$$\beta ::= \varepsilon \mid \{l\} \mid \{l\} \cup \beta$$

5.2.1.1 Annotated Values

With the annotations, we will annotate all the data values in any constructs in μPuppet. An annotated primitive data value is denoted as v^{β}. The structured data values as compilation results such as hash values, resource values and catalogs are defined in figure 5.1. An annotated hash value is defined as the data values of the attributes are annotated. An annotated resource value is defined when its hash-value body is annotated. A resource value can also be annotated as a whole with an annotation that records the dependency on the conditionals that have influenced the control flow so to derive this resource value. We call an annotation of a whole construct as an *outer annotation*. We use an operation $+$ and β to denote it. We will introduce the $+$ operation on annotations and explain how the outer annotation is derived in detail when introducing the propagation of

$$
\begin{array}{lrcl}
\text{Value} & v & ::= & i \mid w \mid \texttt{true} \mid \texttt{false} \\
\text{Annotated Value} & \widehat{v} & ::= & v^{\beta} \\
\text{Hash value} & \widehat{v_{\mathrm{H}}} & ::= & \varepsilon \mid k \Rightarrow \widehat{v}, \widehat{v_{\mathrm{H}}} \\
\text{Resource value} & \widehat{v_{\mathrm{R}}} & ::= & (t\,\{\widehat{w} : \widehat{v_{\mathrm{H}}}\})^{+\beta} \\
\text{Catalog} & \widehat{v_{\mathrm{C}}} & ::= & \varepsilon \mid \widehat{v_{\mathrm{R}}} \_ \widehat{v_{\mathrm{C}}}
\end{array}
$$

Figure 5.1: Annotated Values

annotations in the evaluation later. At last, an annotated catalog is the collection of the annotated resource values.

5.2.1.2 Annotated μPuppet

The annotated μPuppet constructs will be defined inductively on their structures. Then these constructs are annotated if the data values appearing inside are annotated. The full syntax of annotated μPuppet is as in figure 5.2. Not only the data values inside a construct can be annotated, every construct can also have its outer annotation, similar to the annotated resource value. It is needed to deal with the conditionals so that we keep track of the annotations on the parts of manifests that affected the control flow. In the expressions and statements, there are conditional constructs such as selector expressions, if and case statements. Thus the expressions and statements can carry outer annotations to record how they depend on some conditionals. While in the level of manifests, there are no conditional constructs that will generate the dependency in some control flow. The manifest elements will not carry any outer annotation. Thus we only label the expressions and statements with outer annotations denoted as $e^{+\beta}$ and $s^{+\beta}$.

We introduce a hole as a primitive construct in the abstract syntax of μPuppet, denoted as $\_$. It is designed to formalise the semantics of the correctness of dependency provenance. We will explain it in detail in the section of correctness.

5.2.1.3 Annotated Environments

We have seen how $\widehat{v_{\mathrm{C}}}$ is defined. In the evaluation process, the other environments will also carry the annotated constructs of μPuppet. The annotated environments $\widehat{\sigma}$ and $\widehat{\kappa}$ are defined by annotating all the data values appearing in them. Their definitions are as figure 5.3. In the annotated store $\widehat{\sigma}$, the variables and their

$$
\begin{array}{llll}
\text{Expression} & e & ::= & \_ \mid \widehat{v} \mid e^{+\beta} \mid \$x \mid \$::x \mid \$::a::x \\
& & \mid & e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\
& & \mid & e_1 > e_2 \mid e_1 = e_2 \mid e_1 \,\texttt{and}\, e_2 \mid e_1 \,\texttt{or}\, e_2 \mid \texttt{!}e \mid \ldots \\
& & \mid & e \;?\; \{M\} \\
\text{Hash} & H & ::= & \varepsilon \mid k \Rightarrow e, H \\
\text{Case} & c & ::= & e \mid \texttt{default}^{\beta} \\
\text{Matches} & M & ::= & \varepsilon \mid c \Rightarrow e, M \\
\text{Statement} & s & ::= & e \mid s^{+\beta} \mid s_1 \llcorner s_2 \mid \$x = e \mid \texttt{unless}\ e\ \{s\} \mid \texttt{if}\ e\ \{s\}\ \texttt{else}\ \{s\} \\
& & \mid & \texttt{case}\ e\ \{C\} \mid D \mid \texttt{skip}^{\beta} \\
\text{Scope} & \alpha & ::= & \texttt{scope}\ \alpha\ s \\
\text{Cases} & C & ::= & \varepsilon \mid c : \{s\} \llcorner C \\
\text{Declaration} & D & ::= & t\ \{e : H\} \mid u\ \{e : H\} \mid \texttt{class}\ \{a : H\} \mid \texttt{include}\ a \\
\text{Manifest} & m & ::= & s \mid m_1 \llcorner m_2 \mid \texttt{node}\ Q\ \{s\} \mid \texttt{define}\ u\ (\rho)\ \{s\} \\
& & \mid & \texttt{class}\ a\ \{s\} \mid \texttt{class}\ a\ (\rho)\ \{s\} \mid \texttt{class}\ a\ \texttt{inherits}\ b\ \{s\} \\
& & \mid & \texttt{class}\ a\ (\rho)\ \texttt{inherits}\ b\ \{s\} \\
\text{Node spec} & Q & ::= & N \mid \texttt{default} \mid (N_1, \ldots, N_k) \mid r \in \mathit{RegExp} \\
\text{Parameters} & \rho & ::= & \varepsilon \mid x, \rho \mid x = e, \rho
\end{array}
$$

Figure 5.2: Annotated Abstract Syntax of μPuppet

data values can be annotated, but not the namespaces. $\widehat{Var}$ denotes annotated variables. We only consider the variables with the outer annotations which we will illustrate later so $\widehat{Var}$ represents the form $(\$x)^{+\beta}$, $(\$::x)^{+\beta}$ or $(\$::a::x)^{+\beta}$. In the annotated definition $\widehat{\kappa}$, every definition appearing in $\widehat{\kappa}$ is labelled. Every definition is annotated if all the components in it are labelled.

$$
\begin{array}{rlll}
\text{Store} & \widehat{\sigma} & : & \mathit{Scope} \times \widehat{Var} \to \widehat{Value} \\
\text{Definition environment} & \widehat{\kappa} & ::= & \varepsilon \mid a \to \hat{d} \mid a \to \hat{d}, \widehat{\kappa} \\
\text{Definition} & \widehat{d} & ::= & \bot \mid \mathsf{ClassDef}(c_{optc}, \rho, s) \mid \mathsf{DeclaredClass}(\alpha) \\
& & & \mid \mathsf{ResourceDef}(\rho, s)
\end{array}
$$

Figure 5.3: Annotated Abstract Syntax of Environments

5.2.2 Propagation of Annotations in μPuppet Evaluation

In the dependency provenance model proposed in Cheney et al.'s work, the provenance-tracking semantics was defined as annotated-functions that extended ordinary functions to propagate the annotations on data so as to track dependency information in the functions. We will record the dependency information in the process of evaluation. In μPuppet, there are conditional constructs that lead to control flows in the evaluation. Control flow will decide to evaluate one branch among others according to the conditional in the construct. Then a result value of the evaluation could either depend on the operations on data values or conditional constructs that choose one evaluation path. The dependency information for an output value should be able to provide the input data values in the evaluation that decides the output value. To do that, we will propagate annotations on data values in the evaluation when there is dependency involved in every evaluation step. We will define the annotation propagation in every rule evaluating expressions, statements and manifests.

5.2.2.1 Propagation of Annotation in Evaluating Expressions

Expressions in μPuppet include values, variables, all the primitive operations on data values and a selector operation. To propagate the annotations in the expressions in the evaluation to record dependency, we observe that all the operands of an operation contribute to its result. If one of operands changes, its result might change. Then the result of a primitive operation on the input data values depends on each input data value. So the dependency information about the result should be recorded as its annotation which includes the annotations on all the input data values. With this observation, the evaluation rule for the expression on each operator will propagate the annotations by this principle. Selector is a special expression that has the structure of a conditional statement. The conditional data values influence which branch in the selector body is chosen or not chosen. The dependency information in the evaluation should reflect the conditionals in the annotation of the evaluation result.

Variables When looking up the value of a variable in the annotated store $\hat{\sigma}$ under the scope of the variable, the value of this variable should be an annotated value v^a from the store.

$$\frac{x \in dom(\sigma_\alpha)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x^\beta \xrightarrow{\alpha} \sigma_\alpha(x)}\ \textsc{LVar}$$

$$\frac{x \notin dom(\sigma_\alpha) \qquad \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x^\beta \xrightarrow{\alpha'} v^\beta \qquad \alpha' \,\mathsf{parentof}_\kappa\, \alpha}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x^\beta \xrightarrow{\alpha} v^\beta}\ \textsc{PVar}$$

$$\frac{x \in dom(\sigma_{::})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \${::}x^\beta \xrightarrow{\alpha} \sigma_{::}(x)}\ \textsc{TVar} \qquad \frac{x \in dom(\sigma_{::a})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \${::}a :: x^\beta \xrightarrow{\alpha} \sigma_{::a}(x)}\ \textsc{QVar}$$

Arithmetic Expressions We choose + expressions as an example to illustrate how the annotations will be propagated in the evaluation according to the principle we introduced. When a + expression is $e_1 + e_2$, e_1 will be evaluated first. When it is the case $i^\beta + e$, e will be evaluated and i^β will remain in the result. In the case $i_1^{\beta_1} + i_2^{\beta_2}$, the evaluation result will be $i_1 +_{\mathbb{Z}} i_2$ and its annotation will be the union of β_1 and β_2. This annotation implies that the evaluation result $i_1 +_{\mathbb{Z}} i_2$ depends on both inputs i_1 and i_2.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 + e_2 \xrightarrow{\alpha} e_1' + e_2}\ \textsc{ArithLeft} \qquad \frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, i^\beta + e \xrightarrow{\alpha} i^\beta + e'}\ \textsc{ArithRight}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, i_1^{\beta_1} + i_2^{\beta_2} \xrightarrow{\alpha} (i_1 +_{\mathbb{Z}} i_2)^{\beta_1 \cup \beta_2}}\ \textsc{ArithValue}$$

The annotation propagation in the evaluation rules for the expressions on other arithmetic operators are similar.

Comparison Expressions Similarly, we show how the annotations will be propagated in the evaluation of comparison expressions. We use $>$ expressions as an example. In the case to evaluate $e_1 > e_2$, e_1 will be evaluated. In the case of $i^\beta > e$, e will be evaluated and i^β will remain in the result. For the case $i_1^{\beta_1} > i_2^{\beta_2}$, the result will be `true` or `false` while their annotations would be $\beta_1 \cup \beta_2$ since the truth value depends on both input data values i_1 and i_2 that β_1 and β_2 label respectively .

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e_1 > e_2 \xrightarrow{\alpha} e_1' > e_2}\ \textsc{CompLeft} \qquad \frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},i^{\beta} > e \xrightarrow{\alpha} i^{\beta} > e'}\ \textsc{CompRight}$$

$$\frac{i_1 >_{\mathbb{Z}} i_2}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},i_1^{\beta_1} > i_2^{\beta_2} \xrightarrow{\alpha} \texttt{true}^{\beta_1 \cup \beta_2}}\ \textsc{CompValueI}$$

$$\frac{i_1 <=_{\mathbb{Z}} i_2}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},i_1^{\beta_1} > i_2^{\beta_2} \xrightarrow{\alpha} \texttt{false}^{\beta_1 \cup \beta_2}}\ \textsc{CompValueII}$$

The rules for other comparison operators are similar and omitted.

Boolean Expressions We will show the annotation propagation in the evaluation for negation expressions. In the rule to evaluate $!e$, e will be evaluated. For the expressions $!\texttt{true}^{\beta}$ and $!\texttt{false}^{\beta}$, the result truth values `false` and `true` will be annotated with β since the results depend on the input truth values.

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},!e \xrightarrow{\alpha} !e'}\ \textsc{NotSTEP} \qquad \frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},!\texttt{true}^{\beta} \xrightarrow{\alpha} \texttt{false}^{\beta}}\ \textsc{NotValueI}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},!\texttt{false}^{\beta} \xrightarrow{\alpha} \texttt{true}^{\beta}}\ \textsc{NotValueII}$$

The rules for disjunction are similar and omitted. In Puppet, side-effecting expressions do not appear to be allowed inside Boolean expressions so we assume eager semantics for Boolean operations (both sides are fully evaluated).

Selector Expressions The rule SCon and SEle will evaluate the control expressions e and e_1 to data values respectively. For the rule SChoose and SChooseI, the evaluation will resolve different results when $v = v_1$ and $v \neq v_1$. Since their evaluation results depends on the comparison result of v and v_1, both of them influence whether they are equal or not. Then either of the results e and $v^{\beta}\,?\{M\}$ will be annotated with the union of the annotations on v and v_1, i.e. $\beta \cup \beta_1$. Since both evaluation results are expressions, their annotations should be an outer annotation $+(\beta \cup \beta_1)$.

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e\ ?\{M\} \xrightarrow{\alpha} e'\ ?\{M\}}\ \textsc{SCon}$$

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},v^{\beta}\ ?\{e_1 \Rightarrow e, M\} \xrightarrow{\alpha} v^{\beta}\ ?\{e_1' \Rightarrow e, M\}}\ \textsc{SEle}$$

$$\frac{\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\} \xrightarrow{\alpha} e^{+(\beta\cup\beta_1)}}\ \textsc{SChoose}$$

$$\frac{\neg\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\} \xrightarrow{\alpha} (v^{\beta}\ ?\{M\})^{+(\beta\cup\beta_1)}}\ \textsc{SChooseI}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},v^{\beta}\ ?\{\texttt{default} \Rightarrow e, M\} \xrightarrow{\alpha} e^{+\beta}}\ \textsc{SDefault}$$

5.2.2.2 Propagation of Annotations in Evaluating Resources

As we already have shown that a resource would be evaluated to a resource value which would in turn be stored in the catalog v_{C}. Every expression appearing in a resource will be evaluated by employing different evaluation rules. There are no conditionals or direct operations in a resource construct that decide the evaluation result. So there is no immediate annotation propagation that would describe the dependency associated with the level of evaluation of a resource. So the evaluation rules will work the same as the original rules except the constructs in the evaluation are annotated.

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},x \Rightarrow e, H \xrightarrow{\alpha}_{\mathsf{H}} x \Rightarrow e', H}\ \textsc{RbodyExp}$$

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},x \Rightarrow v^{\beta}, H \xrightarrow{\alpha}_{\mathsf{H}} x \Rightarrow v^{\beta}, H'}\ \textsc{RbodyStep}$$

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e : H \xrightarrow{\alpha}_{\mathsf{R}} e' : H}\ \textsc{RName} \qquad \frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},v^{\beta} : H \xrightarrow{\alpha}_{\mathsf{R}} v^{\beta} : H'}\ \textsc{Rbody}$$

5.2.2.3 Propagation of Annotations in Evaluating Statements

In the statements of μPuppet, there are no direct operations on data values involved. However, there are conditional statements if, unless and cases. As we mentioned, when there is a conditional in the construct, one of the branches in the construct body will be chosen or not chosen depending on the data value of the conditional. The remaining branch depends on the data value of the conditional. Thus there is dependency that should be recorded in the annotation on the evaluation result. We will define the annotation propagation in the evaluation of these statements to reflect the dependency which happened in the evaluation by collecting the annotations of relevant input data values. In the following, we will use the evaluation of if and case statements to illustrate the annotation propagation in it to show how the annotations on the results describe the dependency information.

Assignment Statement

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\$x = e \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\$x = e'}\ \textsc{AssignStep}$$

$$\frac{x \notin dom(\sigma_\alpha)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\$x = v^{\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}[(\alpha,x) : v^{\beta}],\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{skip}}\ \textsc{Assign}$$

If Statement To evaluate an if statement, the conditional expression e is first evaluated to an annotated truth value v^{β}. Depending on whether v is `true` or `false`, in the rule IfT and IfF, one of the two branches s_1 and s_2 will be chosen to remain in the evaluation respectively and the other branch will be discarded. That is their evaluation results s_1 and s_2 depend on the conditional truth value v^{β}. To record this dependency, we propagate the annotation β of v as the annotation of the results s_1 and s_2.

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{if}\ e'\ \{s_1\}\ \mathtt{else}\ \{s_2\}}\ \textsc{IfStep}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{if}\ \mathtt{true}^{\beta}\ \{s_1\}\ \mathtt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},(s_1)^{+\beta}}\ \textsc{IfT}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{if}\ \mathtt{false}^{\beta}\ \{s_1\}\ \mathtt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},(s_2)^{+\beta}}\ \textsc{IfF}$$

The annotation propagation in the evaluation of unless statements is similar as in if statements. We show the rules for unless statements in the appendix.

Case Statement The conditional in a case statement is to compare the control data value with that of the current branch. If they are equal, this branch remains as the result of the evaluation and the other branches will be discarded. If not, this branch will be omitted and the other branches will subsequently be checked by the conditional. Thus according to the result of the conditional, a different evaluation result will be derived. That is, if one of the control data values changes, the comparison of them will lead to a different truth value, thus a different evaluation result. Thus any of the two evaluation results depends on two control data values. The dependency information in the annotation on the evaluation results should reflect this by including the annotations of these two control data values. As shown in the rules CaseMatch and CaseNoMatch below, the evaluation results have the union of β and β_1 as their annotation.

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ e\ \{C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ e'\ \{C\}}\ \textsc{CaseStep1}$$

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ v^{\beta}\ \{e:\{s\}\_C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ v^{\beta}\ \{e':\{s\}\_C\}}\ \textsc{CaseStep2}$$

$$\frac{\mathsf{caseMatch}(v,v_1)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ v^{\beta}\ \{v_1^{\beta_1}:\{s\}\_C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},s^{\beta\cup\beta_1}}\ \textsc{CaseMatch}$$

$$\frac{\neg\mathsf{caseMatch}(v,v_1)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ v^{\beta}\ \{v_1^{\beta_1}:\{s\}\_C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},(\mathtt{case}\ v^{\beta}\ \{C\})^{+(\beta\cup\beta_1)}}\ \textsc{CaseNoMatch}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ v^{\beta}\ \{\varepsilon^{\beta_1}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{skip}^{\beta\cup\beta_1}}\ \textsc{CaseDone}$$

The other statements such as expressions, assignments, include, resource-like class declaration, resources, defined resource types declaration, scope statements and sequential statements do not involve immediate operations on data values or conditionals. Their evaluation on the statement level does not involve the dependency on the input data values so there is no annotation propagation to define to record dependency information. The evaluation of these statements is

about to be lifted to the level of expression evaluation, read or change stores, or derive the evaluation result constructs in different structures. The evaluation rules for these statements are omitted here and shown in the appendix.

5.2.2.4 Propagation of Annotation in Evaluating Manifests

Manifest elements consist of statements, sequential manifest elements, class definitions, node definitions and defined resource type definitions. They do not relate to the operations on data values or control flows directly. Their evaluation does not concern the dependency on the inputs so to derive the results. Then there is no annotation propagation involved in their evaluation rules. The rules will be the same as the ordinary rules, except the components in them are annotated. We omit these rules here and show them in the appendix.

5.2.2.5 Propagation of the Outer Annotation $+\beta$ in the Evaluation

As we noticed, when evaluating a construct involving conditionals, its evaluation result will carry an outer annotation of the form $+\beta$ that collects all the annotations of the data values in the conditional. That is, the evaluation result of conditional statements will be in the form of $e^{+\beta}$ and $s^{+\beta}$. The manifest m will not carry outer annotations since there are no conditional constructs at its level. Using an outer annotation on the output construct indicates the whole annotated construct has been derived due to some conditional in the previous evaluation. The propagation of outer annotations implies that the dependency information should be carried along the evaluation. That is, every sub-construct in the construct must depend on this information.

Operation + on Annotations In an outer annotation $+\beta$, $+$ can be seen as an operation on annotations, which is to push the annotation β into the construct and unions it with the inner annotation on each substructure inside the construct. As we knew, since the outer annotation of a construct records the dependency information in the past evaluation, every sub-construct in the construct must depend on it. We define the operation of $+\beta$ applied on expressions and statements respectively in the figure 5.4 (P.109) and figure 5.5 (P.110). In this definition, the operation of $+\beta$ applied on variables, assignments, resource-like class declaration and defined resource type will remain these constructs with $+\beta$ as they are. That is because these constructs with outer annotation should be treated differently

Value	$(v^{\beta_1})^{+\beta_2}$	$=$	$v^{\beta_1 \cup \beta_2}$
Variable	$(\$x)^{+\beta}$	$=$	$(\$x)^{+\beta}$
	$(\$::x)^{+\beta}$	$=$	$(\$::x)^{+\beta}$
	$(\$::a::x)^{+\beta}$	$=$	$(\$::a::x)^{+\beta}$
	$(e_1 + e_2)^{+\beta}$	$=$	$e_1^{+\beta} + e_2^{+\beta}$
	$(e_1 - e_2)^{+\beta}$	$=$	$e_1^{+\beta} - e_2^{+\beta}$
	$(e_1 * e_2)^{+\beta}$	$=$	$e_1^{+\beta} * e_2^{+\beta}$
	$(e_1 / e_2)^{+\beta}$	$=$	$e_1^{+\beta} / e_2^{+\beta}$
	$(e_1 > e_2)^{+\beta}$	$=$	$e_1^{+\beta} > e_2^{+\beta}$
	$(e_1 = e_2)^{+\beta}$	$=$	$e_1^{+\beta} = e_2^{+\beta}$
	$(e_1 \,\texttt{and}\, e_2)^{+\beta}$	$=$	$e_1^{+\beta} \,\texttt{and}\, e_2^{+\beta}$
	$(e_1 \,\texttt{or}\, e_2)^{+\beta}$	$=$	$e_1^{+\beta} \,\texttt{or}\, e_2^{+\beta}$
	$(!\, e)^{+\beta}$	$=$	$!\, e^{+\beta}$
	$(e \;?\; \{M\})^{+\beta}$	$=$	$e^{+\beta} \;?\; \{M^{+\beta}\}$
Hash	$(k \Rightarrow e, H)^{+\beta}$	$=$	$k \Rightarrow e^{+\beta}, H^{+\beta}$
Matches	$(c \Rightarrow e, M)^{+\beta}$	$=$	$(c \Rightarrow e)^{+\beta}, M^{+\beta}$
	$(c \Rightarrow e)^{+\beta}$	$=$	$c^{+\beta} \Rightarrow e^{+\beta}$
	$(\texttt{default}^{\beta})^{+\beta'}$	$=$	$\texttt{default}^{\beta \cup \beta'}$
	$(e^{+\beta})^{+\beta'}$	$=$	$e^{+(\beta \cup \beta')}$

Figure 5.4: $+\beta$ Operation on e

in the evaluation we will illustrate later. In the figure 5.4, when an annotated value v^{β_1} carries an outer annotation $+\beta_2$, v will be labelled with the union of β_1 and β_2 as $v^{\beta_1 \cup \beta_2}$. For the structured constructs, $+\beta$ will push this operation to every sub-construct. Then $+\beta$ operation is defined inductively on the structure of the constructs. For example, $+\beta$ on the expression $e_1 + e_2$ will resolve to the expression $e_1^{+\beta} + e_2^{+\beta}$. In the figure 5.5, for a sequence of statements, $+\beta$ is pushed inside to be the outer annotation of each statement in the sequence. This is especially useful when the statements in the sequence are all the specific constructs we mentioned above whose outer annotations will be treated differently in the evaluation.

Moreover, we extend the + operation on the outer annotation itself. If an expression (a statement) has two layers of outer annotations, i.e. $(e^{+\beta})^{+\beta'}$ $((s^{+\beta})^{+\beta'})$, this operation will union the two outer annotations as one outer annotation, i.e. $e^{+(\beta \cup \beta')}$ $(s^{+(\beta \cup \beta')})$.

$$
\begin{array}{lrcl}
\text{Statement} & (\$x = e)^{+\beta} & = & (\$x = e)^{+\beta} \\
 & (s_1 \llcorner s_2)^{+\beta} & = & s_1^{+\beta} \llcorner s_2^{+\beta} \\
 & (\texttt{if } e \ \{s\} \texttt{ else } \{s\})^{+\beta} & = & \texttt{if } e^{+\beta} \ \{s^{+\beta}\} \texttt{ else } \{s^{+\beta}\} \\
 & (\texttt{unless } e \ \{s\})^{+\beta} & = & \texttt{unless } e^{+\beta} \ \{s^{+\beta}\} \\
 & (\texttt{case } e \ \{C\})^{+\beta} & = & \texttt{case } e^{+\beta} \ \{C^{+\beta}\} \\
\text{Cases} & (c : \{s\} \llcorner C)^{+\beta} & = & c^{+\beta} : \{s^{+\beta}\} \llcorner C^{+\beta} \\
\text{Declaration} & (t \ \{e : H\})^{+\beta} & = & (t \ \{e : H\})^{+\beta} \\
 & (u \ \{e : H\})^{+\beta} & = & (u \ \{e : H\})^{+\beta} \\
 & (\texttt{class } \{a : H\})^{+\beta} & = & (\texttt{class } \{a : H\})^{+\beta} \\
 & (\texttt{include } a)^{+\beta} & = & (\texttt{include } a)^{+\beta} \\
\text{Scope} & (\texttt{scope } \alpha \ s)^{+\beta} & = & \texttt{scope } \alpha \ s^{+\beta} \\
 & (s^{+\beta})^{+\beta'} & = & s^{+(\beta \cup \beta')}
\end{array}
$$

Figure 5.5: $+\beta$ Operation on s

There are two ways to evaluate the constructs with outer annotations. One is to push the outer annotation into the construct and then evaluate the result using the rules introduced in the previous sections. The other is to evaluate the inside construct as usual and then propagate its outer annotation as that of the evaluation result. We choose the second way for the evaluation unless some special cases for which we will define the evaluation rules for them. In other case, we can express the propagation of outer annotations uniformly in the judgements of the evaluation for expressions and statements as below.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e^{+\beta} \xrightarrow{\alpha} e'^{+\beta}} \textsc{Expre} \qquad \frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s^{+\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'^{+\beta}} \textsc{Statement}$$

In the following, we will define the evaluation rules for some specific constructs with outer annotations that include variable, assignment, include, resource-like class declaration and defined resource type.

Variables When a variable $\$x$ ($\$::x$, $\$::a :: x$) has its dependency information, it carries some outer annotation $+\beta$. The evaluation rule looks up the store and get its value as usual. Meanwhile the outer annotation $+\beta$ of the variable will be propagated to its value. The dependency information in the outer annotation

is passed from the variable to its value. The propagation of outer annotations in the rules for the variables in different scopes are as below.

$$\frac{x \in dom(\sigma_\alpha)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\$x)^{+\beta} \xrightarrow{\alpha} (\sigma_\alpha(x))^{+\beta}} \textsc{LVar}$$

$$\frac{x \notin dom(\sigma_\alpha) \qquad \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\$x)^{+\beta'} \xrightarrow{\alpha'} (v^{\beta})^{+\beta'} \qquad \alpha' \,\mathsf{parentof}_\kappa\, \alpha}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\$x)^{+\beta'} \xrightarrow{\alpha} (v^{\beta})^{+\beta'}} \textsc{PVar}$$

$$\frac{x \in dom(\sigma_{::})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\$::x)^{+\beta} \xrightarrow{\alpha} (\sigma_{::}(x))^{+\beta}} \textsc{TVar}$$

$$\frac{x \in dom(\sigma_{::a})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\$::a :: x)^{+\beta} \xrightarrow{\alpha} (\sigma_{::a}(x))^{+\beta}} \textsc{QVar}$$

Assignment Statements When an assignment statement inherits an outer annotation, the outer annotation can be pushed inside to both the variable and the expression. The evaluation can be conducted in the following way. If the expression is not a value, it will be evaluated first as usual, as shown in the rule AssignStepPlus. If the expression is a value, the value will carry the union of its original annotation and the outer annotation same as its variable, as shown in the rule AssignPlus. When updating the store with this variable, its value will have this union as its annotation.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\$x = e)^{+\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\$x = e')^{+\beta}} \textsc{AssignStepPlus}$$

$$\frac{x \notin dom(\sigma_\alpha)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\$x = v^{\beta})^{+\beta'} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}[(\alpha, x^{+\beta'}) : (v^{\beta})^{+\beta'}], \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}} \textsc{AssignPlus}$$

Include Statements An interesting case is the statements of the declaration of classes. A class-declaration statement could be made in any place in μPuppet, including in the body of a conditional construct. So it can be the result of the evaluation of such construct so that it carries an outer annotation. While evaluating an annotated class declaration, the evaluation will execute as the ordinary

evaluation rules and then annotate the evaluation result with the outer annotation of the declaration statement.

In the rule IncU, the class a has not been declared. So this rule declares this class and resolve to the content of this class stored in the definition. Meanwhile this result carries the outer annotation $+\beta$ as its outer annotation. If a class has been declared, the rule IncD will omit this statement and resolve it to `skip` that also carries the outer annotation of this statement. The rule IncPu deals with the case when the class a has a parent class b that has not been declared. The result is the declaration statement of class b together with the declaration of class a. At this step, class b has not been declared. The definition of class b in $\widehat{\kappa}$ remain as $\mathsf{ClassDef}(c_{opt}, \rho', s')$. $(\texttt{include}\ b)^{+\beta}$ is as notification with no real content to evaluate. The evaluation result is not enlarged considering the whole environments. It will introduce the content of class b from its definition in $\widehat{\kappa}$ when $(\texttt{include}\ b)^{+\beta}$ is evaluated. In the rule IncPd, class a has a parent class b which has been declared. Then the content of class a will replace $(\texttt{include}\ a)^{+\beta}$ as the result of the evaluation.

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = merge(\rho, \varepsilon) \qquad \alpha' \,\mathsf{baseof}_{\widehat{\kappa}}\, \alpha}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\texttt{include}\ a)^{+\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \kappa[a : \mathsf{DeclaredClass}(\alpha')], \widehat{v_{\mathrm{C}}}, (\texttt{scope}\ (::a)\ s' \llcorner s)^{+\beta}}\ \textsc{IncU}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{DeclaredClass}(\alpha')}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\texttt{include}\ a)^{+\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}^{+\beta}}\ \textsc{IncD}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \widehat{\kappa}(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\texttt{include}\ a)^{+\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\texttt{include}\ b)^{+\beta}\ (\texttt{include}\ a)^{+\beta}}\ \textsc{IncPU}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \widehat{\kappa}(b) = \mathsf{DeclaredClass}(\alpha') \qquad s' = merge(\rho, \varepsilon)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\texttt{include}\ a)^{+\beta} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa[a : \mathsf{DeclaredClass}(::b)], \widehat{v_{\mathrm{C}}}, (\texttt{scope}\ (::a)\ \{s' \llcorner s\})^{+\beta}}\ \textsc{IncPD}$$

Resource-like Class Declaration Statements A resource-like class declaration statement with an outer annotation will be evaluated similarly. The body of the statement will be evaluated to values by the rule CDecStepPlus. Then when the declaration is evaluated to the other statement, this statement will carry the outer annotation.

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(c_{opt}, \rho, S) \qquad \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\texttt{class}\ \{a : H\})^{+\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\texttt{class}\ \{a : H'\})^{+\beta}}\ \textsc{CDecStepPlus}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}}) \qquad \alpha' \,\mathsf{baseof}_{\kappa}\, \alpha}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\texttt{class}\ \{a : v_{\mathrm{H}}\})^{+\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \kappa[a : \mathsf{DeclaredClass}(\alpha')], \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ s'^{+\beta} \llcorner s^{+\beta}}\ \textsc{CDecUPlus}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \widehat{\kappa}(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\texttt{class}\ \{a : v_{\mathrm{H}}\})^{+\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\texttt{include}\ b)^{+\beta}\ (\texttt{class}\ \{a : v_{\mathrm{H}}\})^{+\beta}}\ \textsc{CDecPUPlus}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \widehat{\kappa}(b) = \mathsf{DeclaredClass}(\alpha) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (\texttt{class}\ \{a : v_{\mathrm{H}}\})^{+\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \kappa[a : \mathsf{DeclaredClass}(::b)], \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ \{s'^{+\beta} \llcorner s^{+\beta}\}}\ \textsc{CDecPDl}$$

Defined Resource Type Statements While a defined resource type statement carries an outer annotation, the statement will be evaluated first until the defined resource type name and the body become values by the rule DefStepPlus. Then in the rule DefPlus, the name and the statements inside the scope statement will carry the outer annotation.

The cases of evaluating $(\texttt{class}\ \{a : v_{\mathrm{H}}\})^{+\beta}$ are similar as for $(\texttt{include}\ a)^{+\beta}$. In the rule CDecPUPlus, the parent of class a has been not declared. The result includes the declaration of the parent class to denote that it will be evaluated before class a in the further evaluation.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \{e : H\} \xrightarrow{\alpha}_{\mathsf{R}} \{e' : H'\}}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (u\ \{e : H\})^{+\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (u\ \{e' : H'\})^{+\beta}}\ \textsc{DefStepPlus}$$

$$\frac{\widehat{\kappa}(u) = \mathsf{ResourceDef}(\rho, s) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, (u\ \{w^{\beta} : v_{\mathrm{H}}\})^{+\beta'} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (\alpha\ \texttt{def})\ \{\$\mathit{title} = w^{\beta \cup \beta'} \llcorner s'^{+\beta'} \llcorner s^{+\beta'}\}}\ \textsc{DefPlus}$$

x

5.3 Correctness of Dependency-Provenance

Dependency provenance we have defined is the dependency information abstracted dynamically. The propagation of annotations in the evaluation of a manifest

records and transfers the dependency information of each data value until the end of evaluation is reached. Thus every data value in the catalog carries an annotation which records the dependency information about all the data values that have contributed to derive this data value in the process of the evaluation. In other words, the dependency information of an output data value specifies any change of which inputs might influence this output and the change of the other inputs definitely will not. In Cheney et al's work (2007), they captured this idea so as to establish a dependency correctness for the annotation behaviour of annotation-functions. The annotation behaviour of functions on annotated values (a.k.a. a-values) was characterised using a relation *"equal except at c"* on a-values. It tries to capture the changes in the data by relating two data that are only different at some place annotated by a specific label c. So it is defined in the way that two a-values are equal except at some label c if they have the same structure or if they are labelled with c and do not necessarily have the same structure. This relation was used to examine the influence of two input data equal except at c on the output data labelled with c. We will adapt this relation to capture two data values in μPuppet only different at some annotations including some label l. Then similarly we will use this relation to formalise the correctness characterisation of dependency-provenance by identifying whether two data values satisfying this relation might influence the specific output data value.

5.3.1 Limitation of Method

Before presenting the correctness characterisation and the proof of the correctness of our dependency-provenance, we want to show their limitations. Due to the complexity of μPuppet language, it turns out dependency-provenance we defined cannot capture the full dependency information of data values in some unusual manifests. The correctness characterisation is also limited to verify such manifests. We will use examples to illustrate where our definition works and where it does not. Then we will propose some possible approaches to solve this limitation.

5.3.1.1 Positive Examples

Dependency-provenance tracks the dependency due to the primitive computations and the conditionals. As in the example below, the declaration of class a depends on the value of $\$x$. According to the propagation of the annotations we defined,

when $\$x = 1$ is true, the statement "include a" will carry the annotation of $\$x$ as its outer annotation.

```
1  $x=1
2  Class a {
3    $y= 2
4  }
5  If  $x=1 {include a}
6      else {$z=2}
```

Dependency-provenance respects class inheritance in that the variables have their correct final values according to scope levels as well as the annotations of these values. In the example below, $\$y$ is defined both in the parent class b and its child class a. When class a is declared, $\$y$ is read in class a so it should get the assignment in class a, i.e. 'Tomorrow is another day', and its annotation.

```
1  Class b {
2    $y=  'Nice day'
3  }
4
5  Class a ($x) inherits b {
6    $y= 'Tomorrow is another day'
7    file { 'motd':
8    path    => $x,
9    content => $y,
10   }
11 }
12
13 class {a: x=>2}
```

5.3.1.2 Counter Examples

We have known that the assignment of a variable in a higher level of scope can be overridden by its assignment in a lower level. In the correctness property we define, the overriding is tracked by storing every assignment of this variable in different scope in the store. However, in the unusual case where there is a conditional also taking part in deciding the final value of an overriding variable, the dependency of the value of this variable on the conditional will not be recorded. Let us see an example as below.

```
1  $y=1
2  Class a ($x) {
3  if $x=1 {$y=42}
4  $z=$y
5  }
6  class {a: x=>2}
7  $w=$a::z
```

In this example, the variable $\$y$ is declared and assigned by 1 in the top level of its manifest. Then a class a is defined where $\$y$ is declared and assigned again in the body of an if statement. y will be overridden by 42 if the conditional is satisfied, i.e. $\$x = 1$. Then class a is declared meanwhile $\$x$ is passed with the value 2. In this case, the inner assignment of $\$y$ will not be executed. The variable $\$z$ will get 1 as its value, so as the variable $\$w$. The dependency-provenance we have defined records the dependency information of conditionals only inside the statement itself, but is not passed to the statements out of it. The value of the variable $\$x$ decides whether the final value of $\$y$ is 42 or 1. However the dependency on the data value of $\$x$ will only be recorded by propagating its annotation to $\$y$ when $\$x = 1$. In the case of the example, the dependency of $\$y = 1$ on $\$x$ is lost. Variable overriding makes the dependency tracking of a data value not explicit.

5.3.1.3 Solutions

We claim that dependency-provenance we defined in the last section is correct for most manifests where the variables to be looked up should have their namespaces in front to identify the scopes where they have been declared. Under this constraint, $\$z = \$y$ in the last example should be changed to $\$z = \$::y$ or $\$z = \$::a::y$. $\$y = 42$ declared in the body of if statement is not allowed since it has been declared in the scope of class a and will be undefined if $\$x = 1$ fails. The correctness characterisation we define also can only verify the correctness of dependency-provenance of the manifests within this semantic behaviour.

Besides adding proper constraints that limit the manifests with unusual semantics, we can also modify dependency-provenance to include more dependency informations by changing the propagation of the annotations in the evaluation rules.

In the following presentation, we constrain the semantics of μPuppet to the sub-language μPuppet' where the variables to look up have their namespaces in front.

5.3.2 Equivalence Relation

We will define an equivalence relation between two constructs in μPuppet. To help with defining this relation, we borrow the concept of a hole "_" in the syntax

in Pottier and Conchon's work (Pottier and Conchon, 2000). This work introduced the hole in the syntax of core ML to remove the irrelevant dependency information in the expressions in terms of an output expression. The hole was treated as a free variable. We will use the hole in μPuppet that expresses the contrary meaning in terms of dependency. Since we want to describe the influence of changing a data value on the output data values, we replace the data value in our observation with a hole in the constructs and leave the other information unchanged in the constructs. Therefore, after employing the hole in the syntax, the equivalence relation on a label l on two constructs will express that either they are syntactically equal or they are labelled with annotations containing l no matter whether they share the same structure or not. We will define a *floor function on* l to introduce the hole $\_$ in the annotated μPuppet syntax that replaces the data values whose annotations include the label l. Its definition is as below.

$$\begin{aligned} \lfloor v^{\beta} \rfloor_l &= \_ \text{ if } l \in \beta \\ \lfloor v^{\beta} \rfloor_l &= v^{\beta} \text{ if } l \notin \beta \end{aligned}$$

The floor function on l $\lfloor\rfloor_l$ is defined on annotated data values. It takes an annotated data value v^{β} and returns a hole $\_$ if the label l belongs to the annotation β, or returns v^{β} itself otherwise. Intuitively, the label l indicates the data value whose annotation includes l we want to observe about whether it influences the output data values in the catalog.

With the floor function on l in hand, we are ready to express two constructs are equal except at some label l, i.e. the equivalence relation except l, denoted as $\equiv_l$. We define this relation for expressions, statements and manifests in the figure 5.7, figure 5.8, figure 5.9 respectively. Moreover it is also defined for the constructs with an outer annotation $+\beta$ in the figure 5.10. It is also extended to the environments in the run-time syntax in the figure 5.11.

In the figure 5.7, the equivalence relation except at l is first defined for two annotated data values $v_1^{\beta_1}$ and $v_2^{\beta_2}$ in the rule Value. They are equal except at l if the results of the floor function on l to them are equal. This rule includes two cases such that $v_1^{\beta_1} \equiv_l v_2^{\beta_2}$. One is that l is in both β_1 and β_2. Then $\lfloor v_1^{\beta_1} \rfloor_l = \lfloor v_2^{\beta_2} \rfloor_l = \_$. The other case is that l is not in β_1 or β_2. Then to satisfy the condition $\lfloor v_1^{\beta_1} \rfloor_l = \lfloor v_2^{\beta_2} \rfloor_l$, $v_1^{\beta_1} = v_2^{\beta_2}$. The other expressions are structured. The equivalence relation except at l on them is defined on the expressions having the same structure. For two + expressions $e_1 + e_2$ and $e_3 + e_4$, they are equal except at l up to commutativity if two operands are equal except at l respectively, i.e.

```
1  if $x=3 { $y=2 } else { }
```

Figure 5.6: Assignment in the if statement

$e_1 \equiv_l e_3$ and $e_2 \equiv_l e_4$. The relation on the expressions of other operations is defined similarly. For the expressions Hash, Match and Selector, the relation is defined as two expressions are equivalent except at l if the two sub-constructs of every corresponding part in them are equal except at l. The equivalence relation on statements, manifests and environments in the figure 5.8, figure 5.9 and figure 5.11 is also defined on two constructs with the same structure such that the two sub-constructs of every corresponding part are equal except at l.

The equivalence relation on stores, definitions and catalogs is defined in the figure 5.11. The store $\widehat{\sigma}$ is to keep the values of the variable by processing an assignment statement. While an assignment can appear as the body in the conditional, its appearance in the store depends on the control values in the conditional. See an example as in the figure 5.6. When $\$x = 3^{\{l\}}$ or $\$x = 1^{\{l\}}$, the evaluation result of this statement would be $\$y = 2$ or empty statement. In the first case, y and its value $2^{\{l,l'\}}$ will be stored in $\widehat{\sigma}$. In the second case, y will be not in $\widehat{\sigma}$. However, in two cases, the dependency on $\$x$ in terms of the label l, two stores should follow the equivalence relation. For defining the equivalence relation on stores, we generalise the floor function $\lfloor\rfloor_l$ on annotation data values to stores. Similar to how it deals with annotation data values, it will leave out the triples where the variables depend on the data value represented by the label l, i.e. their annotations include l. The floor function on stores is defined as below.

$$
\begin{array}{rcl}
\lfloor\{(x_1^{+\beta_1}, \alpha_1, v_1^{\beta_{11}}), \ldots, (x_n^{+\beta_n}, \alpha_n, v_n^{\beta_{nn}})\}\rfloor_l & = & \{\lfloor(x_1^{+\beta_1}, \alpha_1, v_1^{\beta_{11}})\rfloor_l, \ldots, \lfloor(x_n^{+\beta_n}, \alpha_n, v_n^{\beta_{nn}})\rfloor_l\} \\
\lfloor(x_i^{+\beta_i}, \alpha_i, v_i^{\beta_{ii}})\rfloor_l & = & \_ \text{ if } l \in \beta_i \\
\lfloor(x_i^{+\beta_i}, \alpha_i, v_i^{\beta_{ii}})\rfloor_l & = & (x_i^{+\beta_i}, \alpha_i, v_i^{\beta_{ii}}) \text{ if } l \notin \beta_i
\end{array}
$$

The function removes the triples that depend on the information represented by l while remain the others. In this way, we can define the equivalence relation on the stores in this example as if the results of the floor functions on them are equal they are equal except at l, as shown in the rule Sigma in the figure 5.11.

Similarly to the assignments, the resource values could also depend on the conditionals. To define the equivalence relations on catalog in terms of some label l, we will need to remove the resource values that depend on l. We extend

the floor function to catalogs too. The function is defined as below.

$$\begin{aligned}
\lfloor (v_{\mathrm{R}})_1, \ldots, (v_{\mathrm{R}})_n \rfloor_l &= \{\lfloor (v_{\mathrm{R}})_1 \rfloor_l, \ldots, \lfloor (v_{\mathrm{R}})_n \rfloor_l\} \\
\lfloor (t\,\{\widehat{w} : \widehat{v_{\mathrm{H}}}\})^{+\beta} \rfloor_l &= _ \text{ if } l \in \beta \\
\lfloor (t\,\{\widehat{w} : \widehat{v_{\mathrm{H}}}\})^{+\beta} \rfloor_l &= (t\,\{\widehat{w} : \widehat{v_{\mathrm{H}}}\})^{+\beta} \text{ if } l \notin \beta
\end{aligned}$$

The function on annotated catalogs is defined as if the outer annotation of a resource value includes the label l this resource value will be removed. Otherwise it keeps this resource value. Then the equivalence relation on catalogs will be defined as two catalogs are equal except at l if the results of the floor function on them are equal as shown by the rule Catalog in the figure 5.11.

Since any construct could carry outer annotations, we will define the equivalence relation for such constructs too. As we notice, the outer annotations of the constructs represent the dependency information on which these constructs depended. If the outer annotations contain the label l of some data value in our observation, these constructs may depend on this data value. Thus these constructs satisfy the condition of the equivalence relation since this relation requires the constructs not labelled by l are equal but consider the constructs of which the annotations contains l the same.

In the figure 5.10, we will define the equivalence relation for expressions, statements and manifests with outer annotations respectively. There are two possibilities for each of them to satisfy this relation. One is that the two constructs on which the outer annotations are labelled are equal except at l themselves. In this possibility, whether l is contained in β_1 or β_2 is irrelevant since both two constructs are equal in terms of the dependency on the data value l is labelled on. The other possibility is that l is contained in both β_1 and β_2. As we notice, the outer annotations of the constructs represent the dependency information on which these constructs have depended on. If the outer annotations includes the label l, these constructs must depend on the data value l is labelled on. Thus these constructs should be equal in terms of the dependency on this data value, while whether they have the same structures or not is not relevant. The rules TopAnnoEI, TopAnnoSI and TopAnnoMI formalise this possibility for expressions, statements and manifests respectively. Notice more that expressions are statements and statements are manifests meanwhile. These rules imply that any two constructs in μPuppet could be related by this relation if the condition on their outer annotations holds for this possibility.

$$\frac{\lfloor v_1^{\beta_1} \rfloor_l = \lfloor v_2^{\beta_2} \rfloor_l}{v_1^{\beta_1} \equiv_l v_2^{\beta_2}}\ \textsc{Value} \qquad \frac{x_1 = x_2}{\$x_1 \equiv_l \$x_2}\ \textsc{Var} \qquad \frac{x_1 = x_2}{\$::x_1 \equiv_l \$::x_2}\ \textsc{VarSco}$$

$$\frac{a_1 = a_2 \qquad x_1 = x_2}{\$::a_1::x_1 \equiv_l \$::a_2::x_2}\ \textsc{VarClaSco} \qquad \frac{e_1 \equiv_l e_3 \qquad e_2 \equiv_l e_4}{e_1 + e_2 \equiv_l e_3 + e_4}\ \textsc{Add}$$

$$\frac{e_1 \equiv_l e_3 \qquad e_2 \equiv_l e_4}{e_1 > e_2 \equiv_l e_3 > e_4}\ \textsc{CompL} \qquad \frac{e_1 \equiv_l e_3 \qquad e_2 \equiv_l e_4}{e_1 \wedge e_2 \equiv_l e_3 \wedge e_4}\ \textsc{And} \qquad \frac{e_1 \equiv_l e_2}{!e_1 \equiv_l !e_2}\ \textsc{Not}$$

$$\frac{k_1 = k_1' \qquad e_1 \equiv_l e_1' \qquad H_1 \equiv_l H_2}{k_1 \Rightarrow e_1, H_1 \equiv_l k_1' \Rightarrow e_1', H_2}\ \textsc{Hash}$$

$$\frac{c_1 \equiv_l c_2 \qquad e_1 \equiv_l e_2 \qquad M_1 \equiv_l M_2}{c_1 \Rightarrow e_1, M_1 \equiv_l c_2 \Rightarrow e_2, M_2}\ \textsc{Match} \qquad \frac{e_1 \equiv_l e_2 \qquad M_1 \equiv_l M_2}{e_1\ ?\ \{M_1\} \equiv_l e_2\ ?\ \{M_2\}}\ \textsc{Selec}$$

$$\frac{l \in \beta_1 \cap \beta_2}{\texttt{default}^{\beta_1} \equiv_l \texttt{default}^{\beta_2}}\ \textsc{Default} \qquad \frac{l \notin \beta_1 \cup \beta_2}{\texttt{default}^{\beta_1} \equiv_l \texttt{default}^{\beta_2}}\ \textsc{DefaultI}$$

Figure 5.7: Equivalence Relation Except at l on Expressions

5.3.3 Correctness of Dependency Provenance

The equivalence relation describes that two constructs are equal except where their sub-constructs are labelled with annotations including l. It provides the observation that two constructs are the same except the places labelled with l. This gives the way of describing changing some part of the inputs identified by its label. Thus we can capture the correctness of dependency provenance by examining the changes of the output data values when part of the dependency information of output data values in our observation is changed. The equivalence relation allows us to ignore the information labelled with l. Thus we can relate the input data values before and after change at some label l. They hold for the equivalence relation. After the evaluation, we will examine again the output data values with the output before whether they hold for the equivalence relation. It means that if the dependency provenance we define is correct, the output data values before and after the changes of the input data values should hold for the equivalence relation. The outputs not labelled with l should be unchanged. Thus

$$\frac{}{s \equiv_l s}\ \textsc{SymS} \qquad \frac{s_1 \equiv_l s_3 \qquad s_2 \equiv_l s_4}{s_1 \llcorner s_2 \equiv_l s_3 \llcorner s_4}\ \textsc{SeqS} \qquad \frac{x_1 = x_2 \qquad e_1 \equiv_l e_2}{\$x_1 = e_1 \equiv_l \$x_2 = e_2}\ \textsc{Assign}$$

$$\frac{e_1 \equiv_l e_2 \qquad s_1 \equiv_l s_2 \qquad s_1' \equiv_l s_2'}{\texttt{if } e_1 \ \{s_1\} \texttt{ else } \{s_1'\} \equiv_l \texttt{if } e_2 \ \{s_2\} \texttt{ else } \{s_2'\}}\ \textsc{If}$$

$$\frac{e_1 \equiv_l e_2 \qquad s_1 \equiv_l s_2}{\texttt{unless } e_1 \ \{s_1\} \equiv_l \texttt{unless } e_2 \ \{s_2\}}\ \textsc{Unless} \qquad \frac{e_1 \equiv_l e_2 \qquad C_1 \equiv_l C_2}{\texttt{case } e_1 \ \{C_1\} \equiv_l \texttt{case } e_2 \ \{C_2\}}\ \textsc{Case}$$

$$\frac{c_1 \equiv_l c_2 \qquad s_1 \equiv_l s_2 \qquad C_1 \equiv_l C_2}{c_1 : \{s_1\} \llcorner C_1 \equiv_l c_2 : \{s_2\} \llcorner C_2}\ \textsc{CaseBody}$$

$$\frac{t_1 = t_2 \qquad e_1 \equiv_l e_2 \qquad H_1 \equiv_l H_2}{t_1 \ \{e_1 : H_1\} \equiv_l t_2 \ \{e_2 : H_2\}}\ \textsc{ResDec}$$

$$\frac{u_1 = u_2 \qquad e_1 \equiv_l e_2 \qquad H_1 \equiv_l H_2}{u_1 \ \{e_1 : H_1\} \equiv_l u_2 \ \{e_2 : H_2\}}\ \textsc{DefResTDec}$$

$$\frac{a = b}{\texttt{include } a \equiv_l \texttt{include } b}\ \textsc{ClassDel}$$

$$\frac{a_1 = a_2 \qquad H_1 \equiv_l H_2}{\texttt{class } \{a_1 : H_1\} \equiv_l \texttt{class } \{a_2 : H_2\}}\ \textsc{ClassDelI} \qquad \frac{\alpha_1 = \alpha_2 \qquad s_1 \equiv_l s_2}{\texttt{scope } \alpha_1 \ s_1 \equiv_l \texttt{scope } \alpha_2 \ s_2}\ \textsc{Scope}$$

Figure 5.8: Equivalence Relation Except at l on Statements

the outputs before and after the change of the inputs should be equal. The equivalence relation on them also holds. Those outputs labelled with l might be changed. Since the equivalence relation ignores the information labelled with l, those outputs should also satisfy the equivalence relation.

Thus, if the outputs hold for the equivalence relation up to the changes in the inputs, the dependency information abstracted in the annotations is correct. We will use the equivalence relation to express the changes and relate them so to formalise the correctness of dependency provenance. We only concern the manifests that terminate so the correctness we will address is partial.

$$\frac{}{m \equiv_l m}\ \textsc{SymM} \qquad \frac{m_1 \equiv_l m_3 \qquad m_2 \equiv_l m_4}{m_1 \llcorner m_2 \equiv_l m_3 \llcorner m_4}\ \textsc{SeqM}$$

$$\frac{Q_1 = Q_2 \qquad s_1 \equiv_l s_2}{\texttt{node}\ Q_1\ \{s_1\} \equiv_l \texttt{node}\ Q_2\ \{s_2\}}\ \textsc{NodeDef}$$

$$\frac{u_1 = u_2 \qquad \rho_1 \equiv_l \rho_2 \qquad s_1 \equiv_l s_2}{\texttt{define}\ u_1\ (\rho_1)\ \{s_1\} \equiv_l \texttt{define}\ u_2\ (\rho_2)\ \{s_2\}}\ \textsc{DefResTDef}$$

$$\frac{a_1 = a_2 \qquad s_1 \equiv_l s_2}{\texttt{class}\ a_1\ \{s_1\} \equiv_l \texttt{class}\ a_2\ \{s_2\}}\ \textsc{ClassDefI}$$

$$\frac{a_1 = a_2 \qquad \rho_1 \equiv_l \rho_2 \qquad s_1 \equiv_l s_2}{\texttt{class}\ a_1\ (\rho_1)\ \{s_1\} \equiv_l \texttt{class}\ a_2\ (\rho_2)\ \{s_2\}}\ \textsc{ClassDefII}$$

$$\frac{a_1 = a_2 \qquad b_1 = b_2 \qquad s_1 \equiv_l s_2}{\texttt{class}\ a_1\ \texttt{inherits}\ b_1\ \{s_1\} \equiv_l \texttt{class}\ a_2\ \texttt{inherits}\ b_2\ \{s_2\}}\ \textsc{ClassDefIII}$$

$$\frac{a_1 = a_2 \qquad b_1 = b_2 \qquad \rho_1 \equiv_l \rho_2 \qquad s_1 \equiv_l s_2}{\texttt{class}\ a_1\ (\rho_1)\ \texttt{inherits}\ b_1\ \{s_1\} \equiv_l \texttt{class}\ a_2\ (\rho_2)\ \texttt{inherits}\ b_2\ \{s_2\}}\ \textsc{ClassDefIV}$$

$$\frac{x = x' \qquad e \equiv_l e' \qquad \rho \equiv_l \rho'}{x = e, \rho \equiv_l x' = e', \rho'}\ \textsc{Param}$$

Figure 5.9: Equivalence Relation Except at l on Manifests

5.3.3.1 Invariants of the Evaluation

Since the evaluation and the propagation of annotations is defined in the expression, statement and manifest levels, we will first show the invariants about the correctness of dependency provenance in these evaluations respectively. Then we will state the correctness of dependency provenance when evaluating a manifest.

Invariant of Expression Evaluation To examine the correctness of the dependency information generated by the annotation propagation in the expression evaluation, we will show the evaluation of two expressions which are equal except at some label l and check whether their output expressions still follow the equiv-

alence relation or not. In the evaluation of expressions, the environments could be visited for the values and scopes of the variables. So we require the environments of the evaluation of two expressions also follow the equivalence relation. The correctness of dependency provenance in the expression evaluation is stated as an invariant as below.

Theorem 5.3.1 (Invariant of Expression Evaluation)**.** Given $\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_C)_1}, e_1 \xrightarrow{\alpha_1} e_1'$ and $\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_C)_2}, e_2 \xrightarrow{\alpha_2} e_2'$ in μPuppet', if we have $\widehat{\sigma_1} \equiv_l \widehat{\sigma_2}$, $\widehat{\kappa_1} \equiv_l \widehat{\kappa_2}$, $\widehat{(v_C)_1} \equiv_l \widehat{(v_C)_2}$, $\alpha_1 \equiv_l \alpha_2$ and $e_1 \equiv_l e_2$ at some label l, then $e_1' \equiv_l e_2'$.

Proof. Assume $\widehat{\sigma_1} \equiv_l \widehat{\sigma_2}$, $\widehat{\kappa_1} \equiv_l \widehat{\kappa_2}$, $\widehat{(v_C)_1} \equiv_l \widehat{(v_C)_2}$. To have two expressions that are equal except at l, i.e. $e_1 \equiv_l e_2$, they must either have the same structure or have outer annotations including l. In the first case, we will need to prove $e_1' \equiv_l e_2'$. In the second case, we assume that e_1 is in the form $e_{01}^{+\beta_1}$ and e_2 is in the form $e_{02}^{+\beta_2}$. As we defined the judgement of the propagation of outer annotations in the section 5.2.2, the outer annotations of the inputs will be transferred as the outer annotations of their outputs shown as below.

$$\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_C)_1}, e_{01}^{+\beta_1} \xrightarrow{\alpha_1} (e_{01}')^{+\beta_1}$$

$$\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_C)_2}, e_{02}^{+\beta_2} \xrightarrow{\alpha_2} (e_{02}')^{+\beta_2}$$

Then if $e_{01}^{+\beta_1} \equiv_l e_{02}^{+\beta_2}$, then either $e_{01} \equiv_l e_{02}$ or $l \in \beta_1 \cap \beta_2$ by definition. For the first possibility, we should show $e_{01}' \equiv_l e_{02}'$. It falls to prove the same as in the first case, i.e. $e_1' \equiv_l e_2'$. For the second possibility, the two output expressions $(e_{01}')^{+\beta_1}$ and $(e_{02}')^{+\beta_2}$ will have their outer annotations including l. According to the definition of the equivalence relation we have $(e_{01}')^{+\beta_1} \equiv_l (e_{02}')^{+\beta_2}$. In the following we will prove the first case which is to show $e_1' \equiv_l e_2'$.

As mentioned, in the first case e_1 and e_2 have the same structure. Then we can prove this theorem for each kind of expression.

1 Variables

Suppose two variables $\$\alpha_1 x_1^{\beta_1}$ and $\$\alpha_2 x_2^{\beta_2}$. There are four rules possibly applicable for their evaluations. When α_1 (α_2) is ε, i.e. $\$\alpha_1 x_1^{\beta_1}$ ($\$\alpha_2 x_2^{\beta_2}$) is of the form $\$x_1^{\beta_1}(\$x_2^{\beta_2})$, the rule LVar or PVar could apply depending on if $x_1 \in dom(\sigma_{\alpha_1'})$ or not where α_1' is the current evaluation scope. However since $\widehat{\sigma_1} \equiv_l \widehat{\sigma_2}$, we observe in its definition that if two variables are equal except at l their values also follow this relation and their annotations are

the same as the variables. Then no matter in which scope $\$x_1^{\beta_1}$ ($\$x_2^{\beta_2}$) was stored, $\widehat{\sigma_1}_{\alpha_{12}}(x_1^{\beta_1}) \equiv_l \widehat{\sigma_2}_{\alpha_{22}}(x_1^{\beta_2})$ for some α_{12} and α_{22}. If α_1 (α_2) is the top scope :: or the class scope :: a ::, the similar analysis will apply too.

2 Arithmetic expressions

We choose the rules for evaluating + expressions as an example of proving the theorem.

Suppose two + expressions $e_1 + e_2$ and $e_3 + e_4$ such that $e_1 + e_2 \equiv_l e_3 + e_4$. Since e_1, e_2, e_3 and e_4 could be values or other expressions, we need to consider double derivations of the evaluation of two expressions $e_1 + e_2$ and $e_3 + e_4$. However, by definition, we know $e_1 \equiv_l e_3$ and $e_2 \equiv_l e_4$. For the sub-constructs in the expressions, we only consider they are the constructs without outer annotations for the same reason we mentioned. Thus by definition, e_1 and e_3 have the same structure, so as e_2 and e_4. Each of the three rules will be employed depending on whether e_1, e_2, e_3 and e_4 are values or not.

Case (1) e_1, e_2, e_3 and e_4 are values. Then the rule ArithValue is employed.

We suppose $e_1 + e_2 = i_1^{\beta_1} + i_2^{\beta_2}$ and $e_3 + e_4 = i_3^{\beta_3} + i_4^{\beta_4}$. Then the evaluation of them will be as below.

$$\frac{}{\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, i_1^{\beta_1} + i_2^{\beta_2} \xrightarrow{\alpha} (i_1 +_{\mathbb{Z}} i_2)^{\beta_1 \cup \beta_2}}$$

$$\frac{}{\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, i_3^{\beta_3} + i_4^{\beta_4} \xrightarrow{\alpha} (i_3 +_{\mathbb{Z}} i_4)^{\beta_3 \cup \beta_4}}$$

We will prove $(i_1 +_{\mathbb{Z}} i_2)^{\beta_1 \cup \beta_2} \equiv_l (i_3 +_{\mathbb{Z}} i_4)^{\beta_3 \cup \beta_4}$. We know that $i_1^{\beta_1} \equiv_l i_3^{\beta_3}$ and $i_2^{\beta_2} \equiv_l i_4^{\beta_4}$. We know by definition either $i_1 = i_3$ and $l \notin \beta_1 \cup \beta_3$, or $l \in \beta_1 \cap \beta_3$. Similarly we have either $i_2 = i_4$ and $l \notin \beta_2 \cup \beta_4$, or $l \in \beta_2 \cap \beta_4$. Since $i_1 +_{\mathbb{Z}} i_2$ and $i_3 +_{\mathbb{Z}} i_4$ are values, we will check the condition of the rule Value in the definition, i.e. whether $\lfloor (i_1 +_{\mathbb{Z}} i_2)^{\beta_1 \cup \beta_2} \rfloor_l = \lfloor (i_3 +_{\mathbb{Z}} i_4)^{\beta_3 \cup \beta_4} \rfloor_l$ or not. Then we need to check whether l is in $\beta_1 \cup \beta_2$ and $\beta_3 \cup \beta_4$ or not. There are two cases which can be deduced.

(a) $i_1 = i_3$, $i_2 = i_4$ and $l \notin \beta_1 \cup \beta_2$ and $l \notin \beta_3 \cup \beta_4$

(b) $l \in \beta_1 \cup \beta_2$ and $l \in \beta_3 \cup \beta_4$

Then $\lfloor (i_1 +_{\mathbb{Z}} i_2)^{\beta_1 \cup \beta_2} \rfloor_l = \lfloor (i_3 +_{\mathbb{Z}} i_4)^{\beta_3 \cup \beta_4} \rfloor_l$. Then we have $(i_1 +_{\mathbb{Z}} i_2)^{\beta_1 \cup \beta_2} \equiv_l (i_3 +_{\mathbb{Z}} i_4)^{\beta_3 \cup \beta_4}$ by definition.

Case (2) e_1 and e_3 are not values and are expressions of the same structure. Then the rule ArithLeft is employed. The evaluation of the two expressions will be as below.

$$\frac{\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, e_1 + e_2 \xrightarrow{\alpha} e_1' + e_2}$$

$$\frac{\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, e_3 \xrightarrow{\alpha} e_3'}{\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, e_3 + e_4 \xrightarrow{\alpha} e_3' + e_4}$$

.

Since $\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, e_1 \xrightarrow{\alpha} e_1'$ and $\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, e_3 \xrightarrow{\alpha} e_3'$, and $e_1 \equiv_l e_3$, by induction hypothesis, we have $e_1' \equiv_l e_3'$. Since $e_2 \equiv_l e_4$, we have $e_1' + e_2 \equiv_l e_3' + e_4$ by definition.

Case (3) e_1 and e_3 are values. e_2 and e_4 are expressions of the same structure. Then the rule ArithRight is employed.

Suppose $e_1 + e_2 = i_1^{\beta_1} + e_2$ and $e_3 + e_4 = i_3^{\beta_3} + e_4$. Their evaluation will be as below.

$$\frac{\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, e_2 \xrightarrow{\alpha} e_2'}{\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, i_1^{\beta_1} + e_2 \xrightarrow{\alpha} i_1^{\beta_1} + e_2'}$$

$$\frac{\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, e_4 \xrightarrow{\alpha} e_4'}{\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, i_3^{\beta_3} + e_4 \xrightarrow{\alpha} i_3^{\beta_3} + e_4'}$$

.

Since $\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, e_2 \xrightarrow{\alpha} e_2'$ and $\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, e_4 \xrightarrow{\alpha} e_4'$, and $e_2 \equiv_l e_4$, by induction hypothesis, we have $e_2' \equiv_l e_4'$. Since $i_1^{\beta_1} \equiv_l i_3^{\beta_3}$, we have $i_1^{\beta_1} + e_2' \equiv_l i_3^{\beta_3} + e_4'$ by definition.

The proofs of the rules for the expressions of other primitive operations are similar and omitted here.

3 Selector expressions

Suppose two selector expressions $e_1\ ?\{e_2 \Rightarrow e_3, M_1\}$ and $e_4\ ?\{e_5 \Rightarrow e_6, M_2\}$ such that they are equal except at some label l. We will need to consider double derivations of these two expressions to prove the theorem. However, we have by definition that $e_1 \equiv_l e_4$, $e_2 \equiv_l e_5$, $e_3 \equiv_l e_6$ and $M_1 \equiv_l M_2$. Then we know e_1 and e_4 either have the same structure or they have outer annotations, so as for e_2 and e_5, e_3 and e_6, and M_1 and M_2. We only consider the first case as usual. Then the control expressions e_1 and e_4 (e_2 and e_5) are either both values or both not. When they are not values, the rule SCon and SEle will be employed. The proofs are trivial we omitted here. We will prove the case when they are values.

We suppose the two expressions will be $v_1^{\beta_1}\ ?\{v_2^{\beta_2} \Rightarrow e_3, M_1\}$ and $v_4^{\beta_4}\ ?\{v_5^{\beta_5} \Rightarrow e_6, M_2\}$. Whether the rule SChoose or SChooseI is employable depends on whether $v_1 = v_2$ and $v_4 = v_5$ or not. The evaluation of two rules will propagate the outer annotations $+(\beta_1 \cup \beta_2)$ and $+(\beta_4 \cup \beta_5)$ to the result expressions. To show whether the evaluation result expressions are equal except at l or not, we will check whether l is in the outer annotations or not.

Since $v_1^{\beta_1} \equiv_l v_4^{\beta_4}$, $v_2^{\beta_2} \equiv_l v_5^{\beta_5}$, we have $\lfloor v_1^{\beta_1} \rfloor_l = \lfloor v_4^{\beta_4} \rfloor_l$ and $\lfloor v_2^{\beta_2} \rfloor_l = \lfloor v_5^{\beta_5} \rfloor_l$ by definition. Then we can infer

(a) $l \in \beta_1 \cap \beta_4$, or $v_1 = v_4$ and $l \notin \beta_1 \cup \beta_4$

(b) $l \in \beta_2 \cap \beta_5$, or $v_2 = v_5$ and $l \notin \beta_2 \cup \beta_5$

Then there are four combinations of these conditions which derive the following.

(a) $l \in \beta_1 \cap \beta_4$ and $l \in \beta_2 \cap \beta_5$.

(b) $l \in \beta_1 \cap \beta_4$, $v_2 = v_5$ and $l \notin \beta_2 \cup \beta_5$.

(c) $v_1 = v_4$, $l \notin \beta_1 \cup \beta_4$, and $l \in \beta_2 \cap \beta_5$.

In these three cases, we would not know whether $v_1 = v_2$ and $v_4 = v_5$ or not. The rule SChoose or SChooseI could be employed for each expression. However we notice l will be in the outer annotations $+(\beta_1 \cup \beta_2)$ and $+(\beta_4 \cup \beta_5)$ of the evaluation results no matter which rule is employed. Then by definition, the results will be equal except at l.

(d) $v_1 = v_4$, $v_2 = v_5$, $l \notin \beta_1 \cup \beta_4$ and $l \notin \beta_2 \cup \beta_5$.

Then we have $v_1 = v_2$ and $v_4 = v_5$, or $v_1 \neq v_2$ and $v_4 \neq v_5$. In the first case, the rule SChoose is employed for both selector expressions. The results will be $e_3^{+(\beta_1 \cup \beta_2)}$ and $e_6^{+(\beta_4 \cup \beta_6)}$. But since $e_3 \equiv_l e_6$ by assumption, we have $e_3^{+(\beta_1 \cup \beta_2)} \equiv_l e_6^{+(\beta_4 \cup \beta_6)}$ by definition. In the second case, the rule SChooseI is employed for both selector expressions. The results will be $(v_1^{\beta_1}\ ?\{M_1\})^{+(\beta_1 \cup \beta_2)}$ and $(v_4^{\beta_4}\ ?\{M_2\})^{+(\beta_4 \cup \beta_5)}$. Since $M_1 \equiv_l M_2$ by assumption, we have $(v_1^{\beta_1}\ ?\{M_1\})^{+(\beta_1 \cup \beta_2)} \equiv_l (v_4^{\beta_4}\ ?\{M_2\})^{+(\beta_4 \cup \beta_5)}$ by definition.

There are other cases when the current branch in the selector body is a default branch. Since the annotated $\texttt{default}^{\beta_1}$ could only be equal except at l with some $\texttt{default}^{\beta_2}$ if their annotations satisfy the conditions in the definition by definition. The selector expressions can be equal except at l only when their current branches are default branches. The proof of this case will be similar to above when the current branches are expressions. We omit it here.

The proofs for other expressions are similar and omitted here.

□

Invariant of Resource Evaluation

Theorem 5.3.2 (Invariant of Resource Evaluation)**.** Given $\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_\mathrm{C})_1}, D_1 \xrightarrow{\alpha}_\mathsf{R} D_1'$ and $\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_\mathrm{C})_2}, D_2 \xrightarrow{\alpha}_\mathsf{R} D_2'$ in μPuppet', if $\widehat{\sigma_1} \equiv_l \widehat{\sigma_2}$, and $\widehat{\kappa_1} \equiv_l \widehat{\kappa_2}$, $\widehat{(v_\mathrm{C})_1} \equiv_l \widehat{(v_\mathrm{C})_2}$ and $D_1 \equiv_l D_2$, then $D_1' \equiv_l D_2'$.

We consider two resources for the proof without outer annotations for the same reason as for expressions. They must have the same structure as well. Thus we could prove this theorem for each rule of resource evaluation. These proofs can be done by induction on the derivation. We omit them here.

Invariant of Statement Evaluation The evaluation of statements may change the environments. Given two statements and their environments such that they are equal except at l, after the evaluation on them, we want to show the result statements as well as the updated environments still respect the equivalence relation. We state it as invariant of statement evaluation as below.

Theorem 5.3.3 (Invariant of Statement Evaluation)**.** Given $\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_C)_1}, s_1 \xrightarrow{\alpha} \widehat{\sigma_1'}, \widehat{\kappa_1'}, \widehat{(v_C)_1'}, s_1'$ and $\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_C)_2}, s_2 \xrightarrow{\alpha} \widehat{\sigma_2'}, \widehat{\kappa_2'}, \widehat{(v_C)_2'}, s_2'$ in μPuppet', if $\widehat{\sigma_1} \equiv_l \widehat{\sigma_2}$, and $\widehat{\kappa_1} \equiv_l \widehat{\kappa_2}$, $\widehat{(v_C)_1} \equiv_l \widehat{(v_C)_2}$ and $s_1 \equiv_l s_2$, then $\widehat{\sigma_1'} \equiv_l \widehat{\sigma_2'}$, $\widehat{\kappa_1'} \equiv_l \widehat{\kappa_2'}$, $\widehat{(v_C)_1'} \equiv_l \widehat{(v_C)_2'}$ and $s_1' \equiv_l s_2'$.

Proof. Statements can carry outer annotations. As the analysis of expression evaluation, two such statements are equal except at some l if either they follow the equivalence relation themselves or the label l is included in their outer annotations. In the first case, we need to show the evaluation result of two statements are still equal except at l. In the second case, their evaluation results will carry the same outer annotations so they are equal except at l as well. We will prove the invariant in the following for the statements without outer annotations. Then by definition, the given two statements have the same structure. We will prove the invariant for each kind of statements. We assume in the following proof the two environments for two statements $\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_C)_1}$ and $\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_C)_2}$ such that $\widehat{\sigma_1} \equiv_l \widehat{\sigma_2}$, $\widehat{\kappa_1} \equiv_l \widehat{\kappa_2}$, $\widehat{(v_C)_1} \equiv_l \widehat{(v_C)_2}$. The current evaluation scopes of them are α_1 and α_2 such that $\alpha_1 \equiv_l \alpha_2$.

1 Assignment statement

Suppose two assignment statements $\$x_1 = e_1$ and $\$x_2 = e_2$ such that $\$x_1 = e_1 \equiv_l \$x_2 = e_2$. By definition, we know $x_1 \equiv_l x_2$ and $e_1 \equiv_l e_2$. Then we have $x_1 = x_2$ by definition. Suppose e_1 and e_2 are in the form $e_{01}^{+\beta_{01}} \equiv_l e_{02}^{+\beta_{02}}$. Since $e_1 \equiv_l e_2$, by definition, we know either they have the same structure or both e_1 and e_2 have outer annotations. We consider the first case as usual.

There are two rules applicable to assignments. One is to store the values of the variables and the other is to evaluate the expressions in them. But since e_1 and e_2 have the same structure, their evaluation will employ the same rule.

When e_1 and e_2 are not values, the rule AssignStep is applied. Suppose their evaluations are as below.

$$\frac{\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_C)_1}, e_1 \xrightarrow{\alpha_1} e_1'}{\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_C)_1}, \$x = e_1 \xrightarrow{\alpha_1}_{\mathsf{s}} \widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_C)_1}, \$x = e_1'}$$

$$\frac{\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_C)_2}, e_2 \xrightarrow{\alpha_2} e_2'}{\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_C)_2}, \$x = e_2 \xrightarrow{\alpha_2}_{\mathsf{s}} \widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_C)_2}, \$x = e_2'}$$

Since the environments have been not changed after the evaluation, we only need to prove $\$x_1 = e'_1 \equiv_l \$x_2 = e'_2$. Since $\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, e_1 \xrightarrow{\alpha} e'_1, \widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, e_2 \xrightarrow{\alpha} e'_2$, and $e_1 \equiv_l e_2$, by induction hypothesis, we have $e'_1 \equiv_l e'_2$. Since $x_1 = x_2$, we have $\$x_1 = e'_1 \equiv_l \$x_2 = e'_2$.

When e_1 and e_2 are values, the rule Assign is employed. Suppose $e_1 = v_1^{\beta_1}$ and $e_2 = v_2^{\beta_2}$. Their evaluations will be as below.

$$\frac{x \notin dom(\sigma_{\alpha_1})}{\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, \$x_1 = v_1^{\beta_1} \xrightarrow{\alpha_1}_{\mathsf{s}} \widehat{\sigma_1}[(\alpha_1, x_1) : v_1^{\beta_1}], \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, \mathtt{skip}}$$

$$\frac{x \notin dom(\sigma_{\alpha_2})}{\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, \$x_2 = v_2^{\beta_2} \xrightarrow{\alpha_2}_{\mathsf{s}} \widehat{\sigma_2}[(\alpha_2, x_2) : v_2^{\beta_2}], \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, \mathtt{skip}}$$

We have $\mathtt{skip} \equiv_l \mathtt{skip}$ by definition. We want to prove $\widehat{\sigma_1}[(\alpha_1, x_1) : v_1^{\beta_1}] \equiv_l \widehat{\sigma_2}[(\alpha_2, x_2) : v_2^{\beta_2}]$. Since $x_1 = v_1^{\beta_1} \equiv_l x_2 = v_2^{\beta_2}$, we have $x_1 = x_2$ and $v_1^{\beta_1} \equiv_l v_2^{\beta_2}$ by definition. We have $\alpha_1 = \alpha_2$ by assumption. Then we have the extensions of $\widehat{\sigma_1}$ and $\widehat{\sigma_2}$ on $\$x_1$ and $\$x_2$ follow the equivalence relation, i.e. $\widehat{\sigma_1}[(\alpha_1, x_1) : v_1^{\beta_1}] \equiv_l \widehat{\sigma_2}[(\alpha_2, x_2) : v_2^{\beta_2}]$.

2 If statement

Suppose two if statements s_{if1} and s_{if2} such that $s_{if1} \equiv_l s_{if2}$. We consider the case when they do not have outer annotations. Suppose $s_{if1} = \mathtt{if}\ e_1\ \{s_1\}\ \mathtt{else}\ \{s'_1\}$ and $s_{if1} = \mathtt{if}\ e_2\ \{s_2\}\ \mathtt{else}\ \{s'_2\}$. There are three rules applicable to each of them. However since $s_{if1} \equiv_l s_{if2}$, we have $e_1 \equiv_l e_2$, $s_1 \equiv_l s_2$ and $s'_1 \equiv_l s'_2$. e_1 and e_2 must be both values or both not.

When e_1 and e_2 are not values, the rule IfStep is applied to them. Their evaluations are as below.

$$\frac{\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, e_1 \xrightarrow{\alpha_1} e'_1}{\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, \mathtt{if}\ e_1\ \{s_1\}\ \mathtt{else}\ \{s'_1\} \xrightarrow{\alpha_1}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa_1}, \widehat{v_{\mathrm{C}}}, \mathtt{if}\ e'_1\ \{s_1\}\ \mathtt{else}\ \{s_2\}}$$

$$\frac{\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, e \xrightarrow{\alpha_2} e'_2}{\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, \mathtt{if}\ e_2\ \{s_2\}\ \mathtt{else}\ \{s'_2\} \xrightarrow{\alpha_2}_{\mathsf{s}} \widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, \mathtt{if}\ e'_2\ \{s_3\}\ \mathtt{else}\ \{s_4\}}$$

Since the environments have not been changed by the evaluation, we only need to prove $\mathtt{if}\ e'_1\ \{s_1\}\ \mathtt{else}\ \{s'_1\} \equiv_l \mathtt{if}\ e'_2\ \{s_2\}\ \mathtt{else}\ \{s'_2\}$.

By assumption, we have $\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_C)_1}, e_1 \xrightarrow{\alpha} e_1'$ and $\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_C)_2}, e_2 \xrightarrow{\alpha} e_2'$. Since $\widehat{\sigma_1} \equiv_l \widehat{\sigma_2}$, $\widehat{\kappa_1} \equiv_l \widehat{\kappa_2}$, $\widehat{(v_C)_1} \equiv_l \widehat{(v_C)_2}$, and $e_1 \equiv_l e_2$, by induction hypothesis, we have $e_1' \equiv_l e_2'$. Since $s_1 \equiv_l s_2$ and $s_1' \equiv_l s_2'$, we have $\texttt{if}\ e_1'\ \{s_1\}\ \texttt{else}\ \{s_1'\} \equiv_l \texttt{if}\ e_2'\ \{s_2\}\ \texttt{else}\ \{s_2'\}$.

When e_1 and e_2 are values, the rule IfT or IfF could apply to them depending on whether they are `true` or `false`. Suppose $e_1 = v_1^{\beta_1}$ and $e_2 = v_2^{\beta_2}$. Since $v_1^{\beta_1} \equiv_l v_2^{\beta_2}$, we have either $l \in \beta_1 \cap \beta_2$, or $v_1 = v_2$ and $l \notin \beta_1 \cup \beta_2$ by definition.

In the first case, either of the rules IfT and IfF could apply to the statements. The evaluation result of each statement will be $s_1^{+\beta_1}$ or $s_1'^{+\beta_1}$ ($s_2^{+\beta_2}$ or $s_2'^{+\beta_2}$) if the rule IfT or IfF is applied. However, the results of two if statements carries the outer annotations $+\beta_1$ and $+\beta_2$ no matter which rule is employed. Moreover since $l \in \beta_1 \cap \beta_2$, we have the evaluation results follow $s_i'^{\beta_1} \equiv_l s_j'^{\beta_2}$ $(i, j \in \{1, 2\})$ by definition.

In the second case, since $v_1 = v_2$, the same rule IfT or IfF will be applied to both if statements. Their evaluation results will be $s_1^{+\beta_1}$ and $s_2^{+\beta_2}$, or $s_1'^{+\beta_1}$ and $s_2'^{+\beta_2}$ if the rule IfT or IfF is applied. Since $s_1 \equiv_l s_2$ and $s_1' \equiv_l s_2'$, we have $s_1^{+\beta_1} \equiv_l s_2^{+\beta_2}$ or $s_1^{+\beta_1} \equiv_l s_2^{+\beta_2}$ in each case.

3 Case statement

Suppose two case statements s_{c1} and s_{c2} such that $s_{c1} \equiv_l s_{c2}$. We consider the case when they do not have outer annotations as usual. Then they have the same structure. Suppose $s_{c1} = \texttt{case}\ e_1\ \{C_1\}$ and $s_{c2} = \texttt{case}\ e_2\ \{C_2\}$. Since $s_{c1} \equiv_l s_{c2}$, we know $e_1 \equiv_l e_2$ and $C_1 \equiv_l C_2$ by definition. Then we know e_1 and e_2 have the same structure.

If they are not values, the rule CaseStep1 will be applicable. The theorem will be proved by induction on the derivation of the evaluation. We omit it here. If they are values, we assume the statements are $\texttt{case}\ v_1^{\beta_1}\ \{e_1 : \{s_1\} \llcorner C_1\}$ and $\texttt{case}\ v_2^{\beta_2}\ \{e_2 : \{s_2\} \llcorner C_2\}$. Since $e_1 : \{s_1\} \llcorner C_1 \equiv_l e_2 : \{s_2\} \llcorner C_2$, we have $e_1 \equiv_l e_2$, $s_1 \equiv_l s_2$ and $C_1 \equiv_l C_2$. Then the rule CaseStep2 will be applied to both statements. The proof will be constructed by induction on the evaluation derivation. We omit it here. If e_1 and e_2 are values, we assume the two statements are $\texttt{case}\ v_1^{\beta_1}\ \{v_1'^{\beta_1'} : \{s_1\} \llcorner C_1\}$ and $\texttt{case}\ v_2^{\beta_2}\ \{v_2'^{\beta_2'} : \{s_2\} \llcorner C_2\}$. Then two rules are applicable to them which are CaseMatch and CaseNoMatch depending whether $v_1 = v_1'$ and $v_2 = v_2'$ or not. The

evaluation results of each rule will carry the outer annotations $+(\beta_1 \cup \beta_1')$ and $+(\beta_2 \cup \beta_2')$. We know that $v_1^{\beta_1} \equiv_l v_2^{\beta_2}$ and $v_1'^{\beta_1'} \equiv_l v_2'^{\beta_2'}$. Then we can infer the following.

(a) $l \in \beta_1 \cap \beta_2$, or $v_1 = v_2$ and $l \notin \beta_1 \cup \beta_2$

(b) $l \in \beta_1' \cap \beta_2'$, or $v_1' = v_2'$ and $l \notin \beta_1' \cup \beta_2'$

Then four possible cases can be derived.

(a) $l \in \beta_1 \cap \beta_2$ and $l \in \beta_1' \cap \beta_2'$.

(b) $l \in \beta_1 \cap \beta_2$, $v_1' = v_2'$ and $l \notin \beta_1' \cup \beta_2'$.

(c) $v_1 = v_2$, $l \notin \beta_1 \cup \beta_2$, and $l \in \beta_1' \cap \beta_2'$.

In these three cases, we would not know whether $v_1 = v_1'$ or $v_2 = v_2'$ or not so to know which of the rules CaseMatch and CaseNoMatch will be applicable. We notice that l will be in the outer annotations $+(\beta_1 \cup \beta_2)$ and $+(\beta_4 \cup \beta_5)$ of the evaluation results no matter which rule is employed to the statements. Then by definition, their evaluation results will be equal except at l.

(d) $v_1 = v_2$, $v_1' = v_2'$, $l \notin \beta_1 \cup \beta_2$ and $l \notin \beta_1' \cup \beta_2'$. Then we have $v_1 = v_1'$ and $v_2 = v_2'$, or $v_1 \neq v_1'$ and $v_2 \neq v_2'$. In the first case, the rule CaseMatch will be employed to both statements. Their evaluation results will be $s_1^{+(\beta_1 \cup \beta_1')}$ and $s_2^{+(\beta_2 \cup \beta_2')}$. But since $s_1 \equiv_l s_2$, we have $s_1^{+(\beta_1 \cup \beta_1')} \equiv_l s_2^{+(\beta_2 \cup \beta_2')}$ by definition. In the second case, the rule CaseNoMatch will be employed to both statements. Their evaluation results will be $(\texttt{case}\ v_1^{\beta_1}\ \{C_1\})^{+(\beta_1 \cup \beta_1')}$ and $(\texttt{case}\ v_2^{\beta_2}\ \{C_2\})^{+(\beta_2 \cup \beta_2')}$. Since $C_1 \equiv_l C_2$, we have
$(\texttt{case}\ v_1^{\beta_1}\ \{C_1\})^{+(\beta_1 \cup \beta_2)} \equiv_l (\texttt{case}\ v_2^{\beta_2}\ \{C_2\})^{+(\beta_2 \cup \beta_2')}$ by definition.

4 Include statement

Assume two declarations of classes `include` a_1 and `include` a_2 such that `include` $a_1 \equiv_l$ `include` a_2. Then $a_1 = a_2$ by definition. Since we know their definition environments $\widehat{\kappa}_1$ and $\widehat{\kappa}_2$ are also equal except at l, we have $\widehat{\kappa}_1(a_1) \equiv_l \widehat{\kappa}_2(a_2)$. The definitions of two classes are either both $\mathsf{ClassDef}((c_{optc})_1, \rho_1, s_1)$ and $\mathsf{ClassDef}((c_{optc})_2, \rho_2, s_2)$ or $\mathsf{DeclaredClass}(\alpha_1)$ and $\mathsf{DeclaredClass}(\alpha_2)$. They satisfy

$\mathsf{ClassDef}((c_{optc})_1, \rho_1, s_1) \equiv_l \mathsf{ClassDef}((c_{optc})_2, \rho_2, s_2)$ or $\mathsf{DeclaredClass}(\alpha_1) \equiv_l \mathsf{DeclaredClass}(\alpha_2)$. Then $(c_{optc})_1 = (c_{optc})_2$, $\rho_1 \equiv_l \rho_2$ and $s_1 \equiv_l s_2$, or $\alpha_1 = \alpha_2$ by definition. We know that both class declarations will employ the same evaluation rule.

In the first case, when $(c_{optc})_1 = (c_{optc})_2 = \bot$, the rule IncU will apply to both class declarations. The applications are as below.

$$\frac{\widehat{\kappa_1}(a_1) = \mathsf{ClassDef}(\bot, \rho_1, s_1) \qquad s_1' = merge(\rho_1, \varepsilon) \qquad \beta_1 \,\mathsf{baseof}_{\widehat{\kappa_1}}\, \alpha_1)}{\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, \texttt{include}\ a_1 \xrightarrow{\alpha_1}{}'_{\mathsf{s}}\ \widehat{\sigma_1}, \kappa_1[a_1 : \mathsf{DeclaredClass}(\beta_1)], \widehat{(v_{\mathrm{C}})_1}, \texttt{scope}\ (::a_1)\ s_1' \llcorner s_1} \text{ INCLUDE}$$

$$\frac{\widehat{\kappa_2}(a_2) = \mathsf{ClassDef}(\bot, \rho_2, s_2) \qquad s_2' = merge(\rho_2, \varepsilon) \qquad \beta_2 \,\mathsf{baseof}_{\widehat{\kappa_2}}\, \alpha_2}{\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, \texttt{include}\ a_2 \xrightarrow{\alpha_2}{}'_{\mathsf{s}}\ \widehat{\sigma_2}, \kappa_2[a_2 : \mathsf{DeclaredClass}(\beta_2)], \widehat{(v_{\mathrm{C}})_2}, \texttt{scope}\ (::a_2)\ s_2' \llcorner s_2} \text{ INCLUDE1}$$

Since $\rho_1 \equiv_l \rho_2$, we have $s_1' \equiv_l s_2'$. By induction hypothesis, we have $\beta_1 \,\mathsf{baseof}_{\widehat{\kappa_1}}\, \alpha_1$ and $\beta_2 \,\mathsf{baseof}_{\widehat{\kappa_2}}\, \alpha_2$ and we know they are equal except at l. Then we have $\beta_1 = \beta_2$ and $\alpha_1 = \alpha_2$. Then the updates of $\widehat{\kappa_1}$ and $\widehat{\kappa_2}$ with $\widehat{\kappa_1}(a_1) = \mathsf{DeclaredClass}(\beta_1)$ and $\widehat{\kappa_2}(a_2) = \mathsf{DeclaredClass}(\beta_2)$ are equal except at l, i.e. $\kappa_1[a_1 : \mathsf{DeclaredClass}(\beta_1)] \equiv_l \kappa_2[a_2 : \mathsf{DeclaredClass}(\beta_2)]$. Since $s_1 \equiv_l s_2$, $s_1' \equiv_l s_2'$ and $::a_1 = ::a_2$, we have $\texttt{scope}\ (::a_1)\ s_1' \llcorner s_1$ and $\texttt{scope}\ (::a_2)\ s_2' \llcorner s_2$.

When the class a_1 and a_1 have parent classes, the rule IncPU or i.e. $(c_{optc})_1 = b_1$ and $(c_{optc})_2 = \bot$, the rule IncU or IncPD could apply. However since $\widehat{\kappa_1}(a_1) \equiv_l \widehat{\kappa_2}(a_2)$ and $a_1 = a_2$, we have $\widehat{\kappa_1}(a_1) \equiv_l \widehat{\kappa_2}(a_2)$. Then either IncU or IncPD will apply to both class declarations. The proofs are similar as that using the rule IncU. We omit them here.

The second case when both classes have been declared, the rule IncD will apply. The result `skip` is equal except at l to itself.

5 Scope statement

Assume two scope statements $\texttt{scope}\ \alpha_1'\ s_1$ and $\texttt{scope}\ \alpha_2'\ s_2$ such that they are equal except at l. Then by definition, we have $\alpha_1' \equiv_l \alpha_2'$ and $s_1 \equiv_l s_2$. Then they must employ the same rule. When the statements s_1 and s_2 are not `skip` and the scope α_1' and α_2' are one of ::, ::nd and ::, the rule ScopeStep is applicable. The evaluations of two statements are as below.

$$\frac{\alpha_1' \in \{::, ::a, ::\mathrm{nd}\} \qquad \widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_\mathrm{C})_1}, s_1 \xrightarrow{\alpha_1'}_{\mathsf{s}_1'} \widehat{\sigma_1}, \widehat{\kappa_1'}, \widehat{(v_\mathrm{C})_1'}, s_1'}{\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_\mathrm{C})_1}, \mathtt{scope}\ \alpha_1'\ s_1 \xrightarrow{\alpha_1}_{\mathsf{s}} \widehat{\sigma_1'}, \widehat{\kappa_1'}, \widehat{(v_\mathrm{C})_1'}, \mathtt{scope}\ \alpha_1'\ s_1'}\ \textsc{scope}$$

$$\frac{\alpha_2' \in \{::, ::a, ::\mathrm{nd}\} \qquad \widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_\mathrm{C})_2}, s_2 \xrightarrow{\alpha_2'}_{\mathsf{s}_1} \widehat{\sigma_2}, \widehat{\kappa_2'}, \widehat{(v_\mathrm{C})_2'}, s_2'}{\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_\mathrm{C})_2}, \mathtt{scope}\ \alpha_2'\ s_2 \xrightarrow{\alpha_2}_{\mathsf{s}} \widehat{\sigma_2'}, \widehat{\kappa_2'}, \widehat{(v_\mathrm{C})_2'}, \mathtt{scope}\ \alpha_2'\ s_2'}\ \textsc{scope1}$$

By induction hypothesis, we have $\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_\mathrm{C})_1}, s_1 \xrightarrow{\alpha_1'}_{\mathsf{s}_1'} \widehat{\sigma_1}, \widehat{\kappa_1'}, \widehat{(v_\mathrm{C})_1'}, s_1'$ and $\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_\mathrm{C})_2}, s_2 \xrightarrow{\alpha_2'}_{\mathsf{s}_1} \widehat{\sigma_2}, \widehat{\kappa_2'}, \widehat{(v_\mathrm{C})_2'}, s_2'$. Moreover we know $\widehat{\sigma_1'} \equiv_l \widehat{\sigma_2'}$, $\widehat{\kappa_1'} \equiv_l \widehat{\kappa_2'}$, $\widehat{(v_\mathrm{C})_1'} \equiv_l \widehat{(v_\mathrm{C})_2'}$ and $s_1' = s_2'$. Then we have $\mathtt{scope}\ \alpha_1'\ s_1' \equiv_l \mathtt{scope}\ \alpha_2'\ s_2'$.

When s_1 and s_1 are `skip` statements, the rule ScopeDone is applied. The environments would not be updated. The results of the evaluations are `skip` statements and they are equal except at l.

When the scope α_1' and α_2' are the scopes of the defined resource types, the rule DefScopeStep or DefScopeDone will be applicable. The proofs are similar as above. We omit them here.

□

Invariant of Manifest Evaluation

Theorem 5.3.4 (Invariant in Manifest Evaluation)**.** Given $\widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_\mathrm{C})_1}, m_1 \xrightarrow{\alpha_1} \widehat{\sigma_1'}, \widehat{\kappa_1'}, \widehat{(v_\mathrm{C})_1'}, m_1'$ and $\widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_\mathrm{C})_2}, m_2 \xrightarrow{\alpha_2} \widehat{\sigma_2'}, \widehat{\kappa_2'}, \widehat{(v_\mathrm{C})_2'}, m_2'$ in μPuppet', if $\widehat{\sigma_1} \equiv_l \widehat{\sigma_2}$, $\widehat{\kappa_1} \equiv_l \widehat{\kappa_2}$, $\widehat{(v_\mathrm{C})_1} \equiv_l \widehat{(v_\mathrm{C})_2}$, $\alpha_1 \equiv_l \alpha_2$ and $m_1 \equiv_l m_2$ hold, then $\widehat{\sigma_1'} \equiv_l \widehat{\sigma_2'}$, $\widehat{\kappa_1'} \equiv_l \widehat{\kappa_2'}$, $\widehat{(v_\mathrm{C})_1'} \equiv_l \widehat{(v_\mathrm{C})_2'}$ and $m_1' \equiv_l m_2'$.

The proof of this theorem is also conducted by induction on the evaluation of every kind of manifest element. Since there are no conditional or primitive constructs, there will be no double derivations in the proofs. We omit them here.

Invariants of Evaluating Constructs with Outer Annotations

5.3.3.2 Correctness of Dependence-Provenance

Theorem 5.3.5 (Correctness of Dependence-Provenance)**.** Given two manifests m_1 and m_2 such that $m_1 \equiv_l m_2$, and $m_1 \to^* \widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_\mathrm{C})_1}, \mathtt{skip}$ and $m_2 \to^* \widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_\mathrm{C})_2}, \mathtt{skip}$, then $\widehat{\sigma_1} \equiv_l \widehat{\sigma_2}$, $\widehat{\kappa_1} \equiv_l \widehat{\kappa_2}$, $\widehat{(v_\mathrm{C})_1} \equiv_l \widehat{(v_\mathrm{C})_2}$.

Proof. Assume two manifests m_1 and m_2 such that $m_1 \equiv_l m_2$. Suppose the evaluation trains on them as below.

$$m_1 \rightarrow^* \widehat{\sigma_1}, \widehat{\kappa_1}, \widehat{(v_{\mathrm{C}})_1}, \texttt{skip}$$

$$m_2 \rightarrow^* \widehat{\sigma_2}, \widehat{\kappa_2}, \widehat{(v_{\mathrm{C}})_2}, \texttt{skip}$$

In every evaluation step we can prove the evaluation outputs follow Theorem 5.3.1, 5.3.2, 5.3.3 and 5.3.4. That is the result manifests respect the equivalence relation as well as the updated stores, definitions and catalogs. Thus when the evaluation steps stop at `skip`, the environments satisfy the equivalence relation. In particular, $\widehat{(v_{\mathrm{C}})_1} \equiv_l \widehat{(v_{\mathrm{C}})_2}$.

□

$$\frac{l \in \beta_1 \cap \beta_2}{e_1^{+\beta_1} \equiv_l e_2^{+\beta_2}}\ \textsc{TopAnnoE} \qquad \frac{e_1 \equiv_l e_2}{e_1^{+\beta_1} \equiv_l e_2^{+\beta_2}}\ \textsc{TopAnnoEI}$$

$$\frac{l \in \beta_1 \cap \beta_2}{s_1^{+\beta_1} \equiv_l s_2^{+\beta_2}}\ \textsc{TopAnnoS} \qquad \frac{s_1 \equiv_l s_2}{s_1^{+\beta_1} \equiv_l s_2^{+\beta_2}}\ \textsc{TopAnnoSI} \qquad \frac{l \in \beta_1 \cap \beta_2}{\$x_1^{+\beta_1} \equiv_l \$x_2^{+\beta_2}}\ \textsc{VarI}$$

$$\frac{x_1 = x_2}{\$x_1^{+\beta_1} \equiv_l \$x_2^{+\beta_2}}\ \textsc{VarII} \qquad \frac{l \in \beta_1 \cap \beta_2}{(\$::x_1)^{+\beta_1} \equiv_l (\$::x_2)^{+\beta_2}}\ \textsc{VarScoI}$$

$$\frac{x_1 = x_2}{(\$::x_1)^{+\beta_1} \equiv_l (\$::x_2)^{+\beta_2}}\ \textsc{VarScoII} \qquad \frac{l \in \beta_1 \cap \beta_2}{(\$::a_1::x_1)^{+\beta_1} \equiv_l (\$::a_2::x_2)^{+\beta_2}}\ \textsc{VarClaScoI}$$

$$\frac{a_1 = a_2 \qquad x_1 = x_2}{(\$::a_1::x_1)^{+\beta_1} \equiv_l (\$::a_2::x_2)^{+\beta_2}}\ \textsc{VarClaScoII}$$

$$\frac{l \in \beta_1 \cap \beta_2}{(\texttt{include}\ a)^{+\beta_1} \equiv_l (\texttt{include}\ b)^{+\beta_2}}\ \textsc{ClassDelPlus}$$

$$\frac{a = b}{(\texttt{include}\ a)^{+\beta_1} \equiv_l (\texttt{include}\ b)^{+\beta_2}}\ \textsc{ClassDelPlusI}$$

$$\frac{l \in \beta_1 \cap \beta_2}{(\texttt{class}\ \{a_1 : H_1\})^{+\beta_1} \equiv_l (\texttt{class}\ \{a_2 : H_2\})^{+\beta_2}}\ \textsc{ClassDelIPlus}$$

$$\frac{a_1 = a_2 \qquad H_1 \equiv_l H_2}{(\texttt{class}\ \{a_1 : H_1\})^{+\beta_1} \equiv_l (\texttt{class}\ \{a_2 : H_2\})^{+\beta_2}}\ \textsc{ClassDelIPlusI}$$

$$\frac{l \in \beta_1 \cap \beta_2}{(u_1\ \{e_1 : H_1\})^{+\beta_1} \equiv_l (u_2\ \{e_2 : H_2\})^{+\beta_2}}\ \textsc{DefResTDecPlus}$$

$$\frac{u_1 = u_2 \qquad e_1 \equiv_l e_2 \qquad H_1 \equiv_l H_2}{(u_1\ \{e_1 : H_1\})^{+\beta_1} \equiv_l (u_2\ \{e_2 : H_2\})^{+\beta_2}}\ \textsc{DefResTDecPlusI}$$

Figure 5.10: Equivalence Relation Except at l on the constructs with an outer annotation $+\beta$

$$\frac{\lfloor\{(x_1^{+\beta_1},\alpha_1,v_1^{\beta_{11}}),\ldots,(x_n^{+\beta_n},\alpha_n,v_n^{\beta_{nn}})\}\rfloor_l = \lfloor\{(x_1'^{+\beta_1'},\alpha_1',v_1'^{\beta_{11}'}),\ldots,(x_n'^{+\beta_n'},\alpha_n',v_n'^{\beta_{nn}'})\}\rfloor_l}{\{(x_1^{+\beta_1},\alpha_1,v_1^{\beta_{11}}),\ldots,(x_n^{+\beta_n},\alpha_n,v_n^{\beta_{nn}})\} \equiv_l \{(x_1'^{+\beta_1'},\alpha_1',v_1'^{\beta_{11}'}),\ldots,(x_n'^{+\beta_n'},\alpha_n',v_n'^{\beta_{nn}'})\}}\ \textsc{Sigma}$$

$$\frac{a = a' \qquad \widehat{d} \equiv_l \widehat{d'} \qquad \widehat{\kappa} \equiv_l \widehat{\kappa'}}{a \to \widehat{d}, \widehat{\kappa} \equiv_l a' \to \widehat{d'}, \widehat{\kappa'}}\ \textsc{Kappa}$$

$$\frac{c_{optc} = c'_{optc} \qquad \rho \equiv_l \rho' \qquad s \equiv_l s'}{\mathsf{ClassDef}(c_{optc}, \rho, s) \equiv_l \mathsf{ClassDef}(c'_{optc}, \rho', s')}\ \textsc{ClassDef}$$

$$\frac{\alpha = \alpha'}{\mathsf{DeclaredClass}(\alpha) \equiv_l \mathsf{DeclaredClass}(\alpha')}\ \textsc{Declared}$$

$$\frac{\rho \equiv_l \rho' \qquad s \equiv_l s'}{\mathsf{ResourceDef}(\rho, s) \equiv_l \mathsf{ResourceDef}(\rho', s')}\ \textsc{ResourceDef}$$

$$\frac{\lfloor(v_{\mathrm{C}})_1\rfloor_l = \lfloor(v_{\mathrm{C}})_2\rfloor_l}{(v_{\mathrm{C}})_1 \equiv_l (v_{\mathrm{C}})_2}\ \textsc{Catalog} \qquad \frac{\alpha_1 = \alpha_2 \qquad \beta_1 = \beta_2}{\alpha_1\ \mathsf{parentof}_{\kappa_1}\ \beta_1 \equiv_l \alpha_2\ \mathsf{parentof}_{\kappa_2}\ \beta_2}\ \textsc{ParentOf}$$

$$\frac{\alpha_1 = \alpha_2 \qquad \beta_1 = \beta_2}{\alpha_1\ \mathsf{baseof}_{\kappa_1}\ \beta_1 \equiv_l \alpha_2\ \mathsf{baseof}_{\kappa_2}\ \beta_2}\ \textsc{BaseOf} \qquad \frac{}{:: \equiv_l ::}\ \textsc{TopScope}$$

$$\frac{}{::\mathrm{nd} \equiv_l ::\mathrm{nd}}\ \textsc{NodeScope} \qquad \frac{a_1 = a_2}{::a_1 \equiv_l ::a_2}\ \textsc{ClassScope}$$

$$\frac{\alpha_1 \equiv_l \alpha_2}{\alpha_1\ \mathtt{def} \equiv_l \alpha_2\ \mathtt{def}}\ \textsc{DefResScope}$$

Figure 5.11: Equivalence Relation Except at l on Environments

Chapter 6

Conclusion

Automated configuration management tools have become ubiquitous in configuring large infrastructures. However the disadvantage is that these tools cannot help to troubleshoot the configuration errors that are one of the main causes of system failures. The large scale of configuration specifications and the distributed and complex systems make identifying configuration errors more difficult. There is a lack of analysis tools that target the root causes in the configuration specifications of configuration errors. Provenance is a dynamic technique developed in databases to track the origin of the data in query outputs, which is natural to introduce in the context of configuration languages to explain configuration errors by the relevant information in the configuration specifications. Rigorous foundations for configuration languages are needed to improve the reliability of configurations for critical systems and realise the analysis tools for configuration errors on configuration management tools. Puppet is a popular configuration framework, and is already being used in safety-critical domains such as air traffic control.[1]

The contributions of this thesis consist of two main parts. The first part is an operational semantics for a subset of Puppet, called μPuppet, and the interpreter and the parser for it to verify our definition of μPuppet. The second part is the formalisation of where-, expression and dependency-provenance, their correctness characterization and the proofs of their correctness or partial correctness.

μPuppet we identified as a core subset of Puppet language covers the distinctive features of Puppet that are used in most Puppet configurations, including resource, node, class, and defined resource constructs. Our rules also model

[1] `https://archive.fosdem.org/2011/schedule/event/puppetairtraffic.html`.

Puppet's idiosyncratic treatment of classes, scope, and inheritance, including the dynamic treatment of node scope. We presented some simple metatheoretic properties that justify our characterisation of μPuppet as a 'declarative' subset of Puppet, and we compared μPuppet with the Puppet 4.8 implementation on a number of examples. We also identified idiosyncrasies concerning evaluation order and scope where our initial approach differed from Puppet's actual behaviour. Because Puppet is a work in progress, we hope that these observations will contribute to the evolution and improvement of the Puppet language.

Based on the semantics of μPuppet, we have established three forms of provenance in μPuppet that track the different aspects of historical information of data values in the catalog back to the manifest. Among them, where-provenance provides the input location from where an output data value was copied from; expression-provenance tells the primitive operations by which an output data value was derived; dependency-provenance collects all the input data values that could influence the generation of the outputs. Then the correctness properties of three forms of provenance have been characterised statically. We have proved the correctness of where- and expression-provenance and partially that of dependency-provenance against these characterisations. Dependency-provenance we designed does not satisfy the intended correctness property for the complete semantics of the language. It does not track the full dependency information for some unusual value overriding cases. We illustrated the scope of language semantics our dependency-provenance applies and that does not. We have proposed possible solutions such as constraining the manifests so as to exclude these cases, or extending dependency-provenance to carry more information in the annotation propagation. This discovery shows the complexity of Puppet language and provides evidence that could help programming language designers to improve the language.

The formal foundation of Puppet language we modelled builds up a basis of explaining the errors of manifests and further developing analysis tools on it. Three forms of provenance provide specific information about data values in the catalog, which helps to understand the reasons of configuration errors if some value is suspected. Besides, these provenance can identify the responsible parties for the source data values while targeting the relevant input data values. Further more, provenance is an important step toward specifying security properties or constraints as the requirements on the configurations since it analyses the relevant

raw input information related to an output data value.

To the best of our knowledge, this work is the first attempt of employing provenance techniques developed in the data base field for improving configuration language security. Due to the limited works on troubleshooting techniques built for configuration tools, this work presents a solution to targeting the root causes in the configuration specifications using programming language techniques.

In future work, there are several directions we could investigate. Firstly, we can refine dependency-provenance to cover the unusual cases, either extending the dependency information or limiting these cases out for dependency tracking in a formal way. The correctness definition and the proof of dependency-provenance should be modified accordingly. The three forms of provenance we have defined cover most of μPuppet constructs, except hash and array. We can extend them and the correctness proofs to these constructs. The other direction is to investigate more advanced features of Puppet and extend our three forms of provenance to those features. Puppet has been equipped with a type-system. One future direction could be looking into it and studying its relation with three forms of provenance we defined. We can also implement these forms of provenance and validate it against real Puppet manifests together with μPuppet interpreter and parser.

Bibliography

Abadi, M., Banerjee, A., Heintze, N., and Riecke, J. G. (1999). A core calculus of dependency. In *POPL*, pages 147–160. ACM Press.

Acar, U. A., Ahmed, A., Cheney, J., and Perera, R. (2013). A core calculus for provenance. *CoRR*, abs/1310.6299.

Anderson, P. (2006). *System Configuration*, volume 14 of *Short Topics in System Administration*. SAGE.

Anderson, P. (2008). *LCFG: a Practical Tool for System Configuration*, volume 17 of *Short Topics in System Administration*. Usenix Association.

Anderson, P. and Cheney, J. (2012). Toward provenance-based security for configuration languages. In *TaPP*. USENIX. Online proceedings: `http://www.usenix.org/system/files/conference/-tapp12/tapp12-final15.pdf`.

Anderson, P. and Herry, H. (2016). A formal semantics for the SmartFrog configuration language. *J. Network Syst. Manage.*, 24(2):309–345.

Attariyan, M. and Flinn, J. (2008). Using causality to diagnose configuration bugs. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 281–286, Berkeley, CA, USA. USENIX Association.

Attariyan, M. and Flinn, J. (2010). Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 237–250, Berkeley, CA, USA. USENIX Association.

Buneman, P., Cheney, J., and Vansummeren, S. (2008). On the expressiveness of implicit provenance in query and update languages. *ACM Trans. Database Syst.*, 33(4):28:1–28:47.

Buneman, P., Khanna, S., and Tan, W. (2001). Why and where: A characterization of data provenance. In *ICDT*, number 1973 in LNCS, pages 316–330. Springer.

Buneman, P., Naqvi, S. A., Tannen, V., and Wong, L. (1995). Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48.

Chef (2009). `https://www.chef.io/`.

Cheney, J., Ahmed, A., and Acar, U. A. (2007). Provenance as dependency analysis. In *Proceedings of the 11th International Conference on Database Programming Languages*, DBPL'07, pages 138–152, Berlin, Heidelberg. Springer-Verlag.

Cheney, J., Ahmed, A., and Acar, U. a. (2011). Provenance as dependency analysis. *Mathematical. Structures in Comp. Sci.*, 21(6):1301–1337.

Cheney, J., Chiticariu, L., and Tan, W. C. (2009). Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474.

Chugh, R., Hempel, B., Spradlin, M., and Albers, J. (2016). Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 341–354, New York, NY, USA. ACM.

Cui, Y., Widom, J., and Wiener, J. L. (2000). Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227.

Damianou, N. (2002). *A policy framework for management of distributed systems.* PhD thesis, Imperial College.

DeHaan, M. (2012). Ansible. `https://www.ansible.com/`.

Filaretti, D. and Maffeis, S. (2014). An executable formal semantics of PHP. In *ECOOP*, pages 567–592.

Fu, W., Perera, R., Anderson, P., and Cheney, J. (2017). mupuppet: A declarative subset of the puppet configuration language. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, pages 12:1–12:27.

Geerling, J. (2015). *Ansible for DevOps: Server and configuration management for humans*. Midwestern Mac, LLC.

Goldsack, P., Guijarro, J., Loughran, S., Coles, A., Farrell, A., Lain, A., Murray, P., and Toft, P. (2009). The SmartFrog configuration management framework. *SIGOPS Oper. Syst. Rev.*, 43(1):16–25.

Green, T. J., Karvounarakis, G., and Tannen, V. (2007a). Provenance semirings. In *PODS*, pages 31–40. ACM.

Green, T. J., Karvounarakis, G., and Tannen, V. (2007b). Provenance semirings. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, pages 31–40, New York, NY, USA. ACM.

Guha, A., Saftoiu, C., and Krishnamurthi, S. (2010). The essence of JavaScript. In *ECOOP*, pages 126–150, Berlin, Heidelberg. Springer-Verlag.

Gunawi, H. S., Hao, M., Leesatapornwongsa, T., Patana-anake, T., Do, T., Adityatama, J., Eliazar, K. J., Laksono, A., Lukman, J. F., Martin, V., and Satria, A. D. (2014). What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *SOCC*, pages 7:1–7:14, New York, NY, USA. ACM.

Hatch, T. S. (2011). Saltstack. `https://www.ansible.com/`.

Imieliński, T. and Lipski, Jr., W. (1984). Incomplete information in relational databases. *J. ACM*, 31(4):761–791.

Maffeis, S., Mitchell, J. C., and Taly, A. (2008). An operational semantics for JavaScript. In *APLAS*, pages 307–325, Berlin, Heidelberg. Springer-Verlag.

Marschall, M. (2013). *Chef Infrastructure Automation Cookbook*. Packt Publishing.

Morandat, F., Hill, B., Osvald, L., and Vitek, J. (2012). Evaluating the design of the R language - objects and functions for data analysis. In *ECOOP*, pages 104–131.

Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg.

Oppenheimer, D., Ganapathi, A., and Patterson, D. A. (2003). Why do internet services fail, and what can be done about it? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 1–1, Berkeley, CA, USA. USENIX Association.

Plotkin, G. D. (2004). A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139.

Politz, J. G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., and Krishnamurthi, S. (2013). Python: The full monty. In *OOPSLA*, pages 217–232, New York, NY, USA. ACM.

Pottier, F. and Conchon, S. (2000). Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 46–57.

Puppet (2016). Puppet 4.8 reference manual. `https://docs.puppet.com/puppet/4.8/index.html`.

Rhett, J. (2016). *Learning Puppet 4: A guide to configuration management and automation*. O'Reilly Media.

Shambaugh, R., Weiss, A., and Guha, A. (2016). Rehearsal: a configuration verification tool for Puppet. In *PLDI*, pages 416–430.

Turnbull, J. (2008). *Pulling Strings with Puppet: Configuration Management Made Easy*. Apress.

Ueno, K., Fukasawa, Y., Morihata, A., and Ohori, A. (2014). The essence of Ruby. In *APLAS*, pages 78–98.

Vanbrabant, B., Delaet, T., and Joosen, W. (2009). Federated access control and workflow enforcement in systems configuration. In *Proceedings of the 23rd Conference on Large Installation System Administration*, LISA'09, pages 10–10, Berkeley, CA, USA. USENIX Association.

Vanbrabant, B., Peeraer, J., and Joosen, W. (2011). Fine-grained access control for the Puppet configuration language. In *LISA*.

Wang, Y. R. and Madnick, S. E. (1990). A polygon model for heterogeneous database systems: The source tagging perspective. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 519–533, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Weiss, A., Guha, A., and Brun, Y. (2017). Tortoise: Interactive system configuration repair. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 625–636, Piscataway, NJ, USA. IEEE Press.

Xu, T. and Zhou, Y. (2015). Systems approaches to tackling configuration errors: A survey. *ACM Comput. Surv.*, 47(4):70:1–70:41.

Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L. N., and Pasupathy, S. (2011). An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 159–172, New York, NY, USA. ACM.

Zamboni, D. (2012). *Learning CFEngine 3: Automated system administration for sites of any size*. O'Reilly Media.

Appendix A

Semantics of μPuppet

A.1 Glossary

In this section we provide short definitions of certain terms as they are used in Puppet, for reference while reading this thesis.

agent Client software (Puppet agent) running on each node, responsible for liaising with the Puppet master.

catalog A collection of *resources*, together with *ordering constraints* forming a directed acyclic graph among the resources.

class A named collection of resources. Classes can take parameters and their contents can be processed as a result of an *include-like declaration* (`include a`) or a *resource-like declaration* (`class { a : ...}`). Declaring a class more than once has no effect; however, once a class has been declared with parameters (using a resource-like declaration), it cannot be redeclared (with possibly different parameters) again.

compile To process a *manifest* in order to construct the *catalog* for a given *node*. Compilation in Puppet is closer is what is usually called *evaluation* in programming languages terminology, because an expression-like form (the manifest) is "normalised" into a value-like form (the catalog).

compilation error An error that arises during the compilation of a Puppet manifest; a compilation error in Puppet is what would normally be described as a *runtime* error, since only errors that arise in code that is actually executed are reported.

data type An expression such as `Integer` or `Integer[1,10]` that can be used for run-time type checking or parameter validation.

declare To assert the existence of, and describe the parameters of, a resource. When a built-in resource is declared, it is added to the *catalog*. When a user-defined resource or class is declared, its body is processed.

define The definition of class or defined resource type is the program construct that names, lists the parameters, and gives the body of the construct.

environment Group of Puppet nodes that share configuration settings. Useful for testing.

fact(s) Information about the *node* that the *agent* collects, which is passed to the manifest compiler as the value of a special hash-valued variable `$facts`.

hash A dictionary consisting of key-value pair bindings; keys may be 'scalar' values such as integers and strings.

include-like declaration Syntax for declaring a *class* using a special function such as `include`, `contain`, `require`, or `hiera_include`.

manifest The Puppet source code file(s) containing code in the Puppet configuration language, which is *compiled* to produce a catalog for a given *node*.

node A computer system (physical or virtual) that is managed by Puppet, identified by its hostname. Communicates with Puppet master via Puppet agent software.

node scope The scope of the active node definition, typically containing node-specific overrides of global definitions.

ordering constraints Constaints on the order in which resources are processed. These constraints can be defined explicitly using `->`, or implicitly using *metaparameters* such as `require` and `before`. Constraints can also be annotated as conveying information among resources, using `~>` or the *metaparameters* `notify` and `subscribe`.

package A particular Puppet resource type, representing a package on a NODE, abstracting over the particular operating system or package manager in use on that system.

resource A representation of a system component of a *node* that is managed by Puppet; for example, a user account, file, package, or webserver. A resource has a *type* (such as `file` or `user`) and *title* string. A *catalog* can contain at most one resource with a given type and title.

resource collector An expression `T<|pred|>` that collects all resources of a given type `T` whose parameters satisfy a predicate `pred`. Resource collectors can also be used to override resource parameter values. Sometimes called the *spaceship operator*.

resource-like declaration Syntax for declaring a *class* that resembles the syntax used to declare resources. Resource-like declarations are the only way to override the default values of parameters.

resource type The type of a resource. Puppet 4.8 also contains *data types* that classify values of expressions.

top scope The root namespace of a Puppet manifests; parent of any node scope.

A.2 Features supported

Figure A.1 and A.2 summarises our coverage of the Puppet 4.8 language.

Puppet 4.8 feature	Modelled?
Built-in operators	
Comparison operators (`==`, `!=`, `>=`, `<=`, `>`, `and` `<`)	✓
Boolean operators (`and`, `or`, `!`)	✓
Arithmetic operators (`+`, `-`, `*`, `/`, `%`, `<<`, `>>`)	✓
Array and hash operators (`*`, `<<`, `+`, `-`)	✗
Assignment (`=`)	✓
Other core features	
Conditional forms (`if`, `unless`, `case`, `?`)	✓
Node definitions (`node`)	✓
Comments	✓
Facts and pre-set variables	✗
Data types	
Strings, Numbers & Booleans	✓
Arrays	✓
Hashes	✓
Regular expressions	✗
`Sensitive`	✗
`undef`	✗
Resource references	✓
Resource data types	✗
`default`	✓
Data type annotations and values of type `Type`	✗
Resources	
Built-in resource types	✓
Defined resource types (`define`)	✓
Multiple resource bodies; per-expression default attributes	✗
Set attributes from hash (`* =>`)	✗
Abstract resource types (`Resource[...]`)	✗
Multiple resource titles	✗
Add attributes to existing resources	✗

Figure A.1: Summary of Puppet 4.8 language coverage

Puppet 4.8 feature	Modelled?
Classes	
Class parameters	✓
Inheritance	✓
`include` statement	✓
Other multiple-usage mechanisms (`require`, `contain`, `hiera_include`)	✗
Resource-like declarations	✓
Relationships and ordering	
Relationship metaparameters (`before, require, notify, subscribe`)	✗
Chaining arrows (`->`)	✗
`require function`	✗
Advanced constructs	
Plugins (Ruby-level functions)	✗
Puppet-level functions	✗
Templates	✗
Iteration functions (`each, slice, filter, map, reduce`)	✗
Lambdas (`|...| {...}`)	✗
Advanced resource features (defaults, collectors, virtual resources, exported resources)	✗
Tags	✗
Run stages	✗

Figure A.2: Summary of Puppet 4.8 language coverage cont.

A.3 Operational semantics

A.3.1 Environment operations

- The operation *lookup* returns the first value associated with a parameter in a hash, or $\bot$ if the parameter is not a key in the hash.

$$\begin{aligned} \mathit{lookup} &: \mathit{Var} \times \mathit{HashValue} \to \mathit{Value} \uplus \{\bot\} \\ \mathit{lookup}(x, (y \Rightarrow v, v_{\mathrm{H}})) &= \mathit{lookup}(x, v_{\mathrm{H}}) \qquad \text{if } x \neq y \\ \mathit{lookup}(x, (x \Rightarrow v, v_{\mathrm{H}})) &= v \\ \mathit{lookup}(x, \varepsilon) &= \bot \end{aligned}$$

- The function $\mathit{lookup}_{\mathrm{C}}$ returns an attribute value of a resource value in a

catalog v_{C} if the resource has the type t, the title w and the attribute x. It returns $\bot$ if there is no such a resource value.

$$
\begin{array}{rcl}
\mathit{lookup}_{\mathrm{C}} & : & v_{\mathrm{C}} \times \mathit{String} \times \mathit{String} \times \mathit{String} \to \mathit{Value} \uplus \{\bot\} \\
\mathit{lookup}_{\mathrm{C}}((t : \{w : y \Rightarrow v, v_{\mathrm{H}}\} \llcorner v_{\mathrm{C}}), t', w', x) & = & \mathit{lookup}_{\mathrm{C}}(v_{\mathrm{C}}, t', w', x) \\
 & & \text{if } t \neq t' \text{or } w \neq w' \\
\mathit{lookup}_{\mathrm{C}}((t : \{w : y \Rightarrow v, v_{\mathrm{H}}\} \llcorner v_{\mathrm{C}}), t, w, x) & = & v \\
\mathit{lookup}_{\mathrm{C}}(\varepsilon, t, w, x) & = & \bot
\end{array}
$$

- The partial function *merge* takes a list of parameters with optional default expressions, and a hash mapping parameter names to overriding values, and returns a statement initialising each parameter to its default or overridden value.

$$
\begin{array}{rcll}
\mathit{merge} & : & \mathit{Params} \times \mathit{HashValue} \to \mathit{Stmt} & \\
\mathit{merge}(\varepsilon, v_{\mathrm{H}}) & = & \texttt{skip} & \\
\mathit{merge}((\$x, \rho), v_{\mathrm{H}}) & = & \$x = v \llcorner \mathit{merge}(\rho, v_{\mathrm{H}}) & \text{if } v = \mathit{lookup}(x, v_{\mathrm{H}}) \neq \bot \\
\mathit{merge}((\$x = e, \rho), v_{\mathrm{H}}) & = & \$x = v \llcorner \mathit{merge}(\rho, v_{\mathrm{H}}) & \text{if } v = \mathit{lookup}(x, v_{\mathrm{H}}) \neq \bot \\
\mathit{merge}((\$x = e, \rho), v_{\mathrm{H}}) & = & \$x = e \llcorner \mathit{merge}(\rho, v_{\mathrm{H}}) & \text{if } \mathit{lookup}(x, v_{\mathrm{H}}) = \bot
\end{array}
$$

  Note that $\mathit{merge}((\$x, \rho), v_{\mathrm{H}})$ is undefined unless x is a key of v_{H}.

- Update $\sigma[(\alpha, x) : v]$ add a new variable-value pair to a variable environment. It is defined as follows:

$$
\begin{array}{rcl}
-[(-,-) : -] & : & \mathit{Env} \times \mathit{Scope} \times \mathit{Var} \times \mathit{Value} \to \mathit{Env} \\
\sigma[(\alpha, x) : v] & = & \lambda(\alpha', y).\ \text{if } \alpha = \alpha' \text{ and } x = y \text{ then } v \text{ else } \sigma(\alpha', y)
\end{array}
$$

  where $\mathit{Env} = \mathit{Scope} \times \mathit{Var} \to \mathit{Value}$.

- Specialisation σ_α is the variable environment σ specialised to a particular scope α, defined as follows:

$$
\begin{array}{rcl}
-_{-} & : & \mathit{Var} \to \mathit{Value} \\
\sigma_\alpha & = & \lambda x.\sigma(\alpha, x)
\end{array}
$$

- Clearing $clear(\sigma, \alpha)$ makes all variables in scope α undefined, leaving all other variable bindings unchanged. It is used to clean up $\alpha\,\texttt{def}$ scopes at the end of their lifetime. This operation is defined as follows:

$$\begin{array}{rcl} clear & : & Env \times Scope \to Env \\ clear(\sigma,\alpha) & = & \lambda(\alpha', y).\begin{cases} \sigma(\alpha', y) & \text{if } \alpha \neq \alpha' \\ \text{undefined} & \text{otherwise} \end{cases} \end{array}$$

- The parent of a scope α in the context of a definition environment κ is defined as follows:

$$\frac{}{::\,\mathsf{parentof}_\kappa\,::\texttt{nd}}\ \textsc{PNode} \qquad \frac{\beta\,\mathsf{baseof}_\kappa\,\alpha\,\texttt{def}}{\beta\,\mathsf{parentof}_\kappa\,\alpha\,\texttt{def}}\ \textsc{PDefRes}$$

$$\frac{\kappa(a) = \mathsf{DeclaredClass}(\alpha)}{\alpha\,\mathsf{parentof}_\kappa\,::a}\ \textsc{PClass}$$

- The *base scope* of a scope α in the context of a definition environment κ is defined as follows:

$$\frac{}{::\,\mathsf{baseof}_\kappa\,::}\ \textsc{BTop} \qquad \frac{}{::\texttt{nd}\,\mathsf{baseof}_\kappa\,::\texttt{nd}}\ \textsc{BNode} \qquad \frac{\alpha\,\mathsf{baseof}_\kappa\,\beta}{\alpha\,\mathsf{baseof}_\kappa\,\beta\,\texttt{def}}\ \textsc{BDefRes}$$

$$\frac{\kappa(a) = \mathsf{DeclaredClass}(\beta) \qquad \alpha\,\mathsf{baseof}_\kappa\,\beta}{\alpha\,\mathsf{baseof}_\kappa\,::a}\ \textsc{BClass}$$

A.3.2 Expressions ($\sigma, \kappa, v_\mathrm{C}, e \xrightarrow{\alpha} e'$)

A.3.2.1 Variables

$$\frac{x \in dom(\sigma_\alpha)}{\sigma, \kappa, v_\mathrm{C}, \$x \xrightarrow{\alpha} \sigma_\alpha(x)}\ \textsc{LVar}$$

$$\frac{x \notin dom(\sigma_\alpha) \qquad \sigma, \kappa, v_\mathrm{C}, \$x \xrightarrow{\beta} v \qquad \beta\,\mathsf{parentof}_\kappa\,\alpha}{\sigma, \kappa, v_\mathrm{C}, \$x \xrightarrow{\alpha} v}\ \textsc{PVar}$$

$$\frac{x \in dom(\sigma_{::})}{\sigma, \kappa, v_\mathrm{C}, \$::x \xrightarrow{\alpha} \sigma_{::}(x)}\ \textsc{TVar} \qquad \frac{x \in dom(\sigma_{::a})}{\sigma, \kappa, v_\mathrm{C}, \$::a :: x \xrightarrow{\alpha} \sigma_{::a}(x)}\ \textsc{QVar}$$

A.3.2.2 Arithmetic expressions

$$\frac{\sigma, \kappa, v_C, e_1 \xrightarrow{\alpha} e_1'}{\sigma, \kappa, v_C, e_1 + e_2 \xrightarrow{\alpha} e_1' + e_2}\ \textsc{ArithLeft} \qquad \frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, i + e \xrightarrow{\alpha} i + e'}\ \textsc{ArithRight}$$

$$\frac{}{\sigma, \kappa, v_C, i_1 + i_2 \xrightarrow{\alpha} i_1 +_{\mathbb{Z}} i_2}\ \textsc{ArithValue}$$

Here, $+_{\mathbb{Z}}$ stands for the usual integer addition function. The evaluation rules for $-, *, /$ are similar to those for $+$, and omitted.

A.3.2.3 Comparison expressions

$$\frac{\sigma, \kappa, v_C, e_1 \xrightarrow{\alpha} e_1'}{\sigma, \kappa, v_C, e_1 > e_2 \xrightarrow{\alpha} e_1' > e_2}\ \textsc{CompLeft} \qquad \frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, v > e \xrightarrow{\alpha} v > e'}\ \textsc{CompRight}$$

$$\frac{v_1 >_{\mathbb{Z}} v_2}{\sigma, \kappa, v_C, v_1 > v_2 \xrightarrow{\alpha} \texttt{true}}\ \textsc{CompValueI}$$

$$\frac{v_1 <=_{\mathbb{Z}} v_2}{\sigma, \kappa, v_C, v_1 > v_2 \xrightarrow{\alpha} \texttt{false}}\ \textsc{CompValueII}$$

The rules for other comparison operators are similar and omitted.

A.3.2.4 Boolean expressions

$$\frac{\sigma, \kappa, v_C, e_1 \xrightarrow{\alpha} e_1'}{\sigma, \kappa, v_C, e_1 \,\texttt{and}\, e_2 \xrightarrow{\alpha} e_1' \,\texttt{and}\, e_2}\ \textsc{AndLeft}$$

$$\frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, \texttt{false}\,\texttt{and}\, e \xrightarrow{\alpha} \texttt{false}}\ \textsc{AndRightI}$$

$$\frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, \texttt{true}\,\texttt{and}\, e \xrightarrow{\alpha} \texttt{true}\,\texttt{and}\, e'}\ \textsc{AndRightII}$$

$$\frac{}{\sigma, \kappa, v_C, \texttt{true}\,\texttt{and}\,\texttt{true} \xrightarrow{\alpha} \texttt{true}}\ \textsc{AndValue} \qquad \frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, !e \xrightarrow{\alpha} !e'}\ \textsc{NotSTEP}$$

$$\frac{}{\sigma, \kappa, v_C, !\texttt{true} \xrightarrow{\alpha} \texttt{false}}\ \textsc{NotValueI} \qquad \frac{}{\sigma, \kappa, v_C, !\texttt{false} \xrightarrow{\alpha} \texttt{true}}\ \textsc{NotValueII}$$

The rules for disjunction are analogous and are omitted.

A.3.2.5 Array

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, A \xrightarrow{\alpha} A'}{\sigma, \kappa, v_{\mathrm{C}}, [A] \xrightarrow{\alpha} [A']}\ \textsc{ArrExp} \qquad \frac{\sigma, \kappa, v_{\mathrm{C}}, A \xrightarrow{\alpha} A'}{\sigma, \kappa, v_{\mathrm{C}}, v, A \xrightarrow{\alpha} v, A}\ \textsc{ArrEle}$$

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e_1 \xrightarrow{\alpha} e_1'}{\sigma, \kappa, v_{\mathrm{C}}, e_1, A \xrightarrow{\alpha} e_1', A}\ \textsc{ArrEleI}$$

A.3.2.6 Hash

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, H \xrightarrow{\alpha} H'}{\sigma, \kappa, v_{\mathrm{C}}, \{H\} \xrightarrow{\alpha} \{H\}}\ \textsc{HExp} \qquad \frac{\sigma, \kappa, v_{\mathrm{C}}, H \xrightarrow{\alpha} H'}{\sigma, \kappa, v_{\mathrm{C}}, \mathtt{k} \Rightarrow v, H \xrightarrow{\alpha} \mathtt{k} \Rightarrow v, H}\ \textsc{HEle}$$

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_{\mathrm{C}}, \mathtt{k} \Rightarrow e, H \xrightarrow{\alpha} \mathtt{k} \Rightarrow e', H}\ \textsc{HEleI}$$

A.3.2.7 Selectors

The predicate caseMatch abstracts over the details of matching values against selector cases. Real Puppet checks any `default` clause last, failing with an exception if there is no case which matches; for simplicity we omit these details from the formalism.

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_{\mathrm{C}}, e\ ?\{M\} \xrightarrow{\alpha} e'\ ?\{M\}}\ \textsc{SCon}$$

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e_1 \xrightarrow{\alpha} e_1'}{\sigma, \kappa, v_{\mathrm{C}}, v\ ?\{e_1 \Rightarrow e, M\} \xrightarrow{\alpha} v\ ?\{e_1' \Rightarrow e, M\}}\ \textsc{SEle}$$

$$\frac{\mathsf{caseMatch}(v, v_1)}{\sigma, \kappa, v_{\mathrm{C}}, v\ ?\{v_1 \Rightarrow e, M\} \xrightarrow{\alpha} e}\ \textsc{SChoose}$$

$$\frac{\neg\mathsf{caseMatch}(v, v_1)}{\sigma, \kappa, v_{\mathrm{C}}, v\ ?\{v_1 \Rightarrow e, M\} \xrightarrow{\alpha} v\ ?\{M\}}\ \textsc{SChooseI}$$

$$\frac{}{\sigma, \kappa, v_{\mathrm{C}}, v\ ?\{\mathtt{default} \Rightarrow e, M\} \xrightarrow{\alpha} e}\ \textsc{SDefault}$$

A.3.2.8 Dereference

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, d \xrightarrow{\alpha} d'}{\sigma, \kappa, v_{\mathrm{C}}, d[e] \xrightarrow{\alpha} d'[e]}\ \textsc{DeRefExp} \qquad \frac{\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_{\mathrm{C}}, v[e] \xrightarrow{\alpha} v[e']}\ \textsc{DeRefIndex}$$

$$\frac{}{\sigma, \kappa, v_{\mathrm{C}}, [v_0, \ldots, v_n, \ldots, v_m][n] \xrightarrow{\alpha} v_n}\ \textsc{DeRefArray}$$

$$\frac{k = k_n}{\sigma, \kappa, v_{\mathrm{C}}, \{k_1 = v_1, \ldots, k_n = v_n, \ldots, k_m = v_m\}[k] \xrightarrow{\alpha} v_n}\ \textsc{DeRefHash}$$

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_{\mathrm{C}}, t[e] \xrightarrow{\alpha} t[e']}\ \textsc{RefRes} \qquad \frac{\mathit{lookup}_{\mathrm{C}}(v_{\mathrm{C}}, t, w, k) = v}{\sigma, \kappa, v_{\mathrm{C}}, t[w][k] \xrightarrow{\alpha} v}\ \textsc{DeRefRes}$$

A.3.3 Resources ($\sigma, \kappa, H \xrightarrow{\alpha}_{\mathsf{H}} H'$ and $\sigma, \kappa, e : H \xrightarrow{\alpha}_{\mathsf{R}} e' : H'$)

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_{\mathrm{C}}, x \Rightarrow e, H \xrightarrow{\alpha}_{\mathsf{H}} x \Rightarrow e', H} \qquad \frac{\sigma, \kappa, v_{\mathrm{C}}, H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\sigma, \kappa, v_{\mathrm{C}}, x \Rightarrow v, H \xrightarrow{\alpha}_{\mathsf{H}} x \Rightarrow v, H'}$$

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_{\mathrm{C}}, e : H \xrightarrow{\alpha}_{\mathsf{R}} e' : H} \qquad \frac{\sigma, \kappa, v_{\mathrm{C}}, H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\sigma, \kappa, v_{\mathrm{C}}, v : H \xrightarrow{\alpha}_{\mathsf{R}} v : H'}$$

A.3.4 Statements ($\sigma, \kappa, v_{\mathrm{C}}, s \xrightarrow{\alpha}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'$)

A.3.4.1 Expression statements

An expression can occur as a statement. Its value is ignored.

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, e'}\ \textsc{ExprStep} \qquad \frac{}{\sigma, \kappa, v_{\mathrm{C}}, v \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{skip}}\ \textsc{Expr}$$

A.3.4.2 Sequential composition

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, s_1 \xrightarrow{\alpha}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'_1}{\sigma, \kappa, v_{\mathrm{C}}, s_1 \texttt{;} s_2 \xrightarrow{\alpha}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'_1 \texttt{;} s_2}\ \textsc{SeqStep}$$

$$\frac{}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{skip;}\, s \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, s}\ \textsc{SeqSkip}$$

A.3.4.3 Assignment

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_{\mathrm{C}}, \$x = e \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \$x = e'}\ \textsc{AssignStep}$$

$$\frac{x \notin dom(\sigma_\alpha)}{\sigma, \kappa, v_{\mathrm{C}}, \$x = v \xrightarrow{\alpha}_{\mathsf{s}} \sigma[(\alpha, x) : v], \kappa, v_{\mathrm{C}}, \texttt{skip}}\ \textsc{Assign}$$

A.3.4.4 If

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{if}\ e\ \{s_1\}\ \texttt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{if}\ e'\ \{s_1\}\ \texttt{else}\ \{s_2\}}\ \textsc{IfStep}$$

$$\frac{}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{if true}\ \{s_1\}\ \texttt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, s_1}\ \textsc{IfT}$$

$$\frac{}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{if false}\ \{s_1\}\ \texttt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, s_2}\ \textsc{IfF}$$

A.3.4.5 Unless

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{unless}\ e\ \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{unless}\ e'\ \{s\}}\ \textsc{UnlessStep}$$

$$\frac{}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{unless true}\ \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}},\ \texttt{skip}}\ \textsc{UnlessT}$$

$$\frac{}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{unless false}\ \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}},\ s}\ \textsc{UnlessF}$$

A.3.4.6 Case

As with selectors, the predicate caseMatch abstracts over the details of pattern-matching. Again, in real Puppet the `default` case is always checked last, regardless of the order of the cases.

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{case}\ e\ \{C\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{case}\ e'\ \{C\}}\ \textsc{CaseStep1}$$

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{case}\ v\ \{e : \{s\} \sqcup C\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{case}\ v\ \{e' : \{s\} \sqcup C\}}\ \textsc{CaseStep2}$$

$$\frac{\mathsf{caseMatch}(v, v_1)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{case}\ v\ \{v_1 : \{s\} \sqcup C\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, s}\ \textsc{CaseMatch}$$

$$\frac{\neg\mathsf{caseMatch}(v, v_1)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{case}\ v\ \{v_1 : \{s\} \sqcup C\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{case}\ v\ \{C\}}\ \textsc{CaseNoMatch}$$

$$\frac{}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{case}\ v\ \{\varepsilon\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{skip}}\ \textsc{CaseDone}$$

A.3.4.7 Resource declarations

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, e : H \xrightarrow{\alpha}_{\mathsf{R}} e' : H'}{\sigma, \kappa, v_{\mathrm{C}}, t\ \{e : H\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, t\ \{e' : H'\}}\ \textsc{ResStep}$$

$$\frac{}{\sigma, \kappa, v_{\mathrm{C}}, v_{\mathrm{R}} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}} \sqcup v_{\mathrm{R}}, \texttt{skip}}\ \textsc{ResDecl}$$

A.3.4.8 Defined resource types

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, \{e : H\} \xrightarrow{\alpha}_{\mathsf{R}} \{e' : H'\}}{\sigma, \kappa, v_{\mathrm{C}}, u\ \{e : H\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, u\ \{e' : H'\}}\ \textsc{DefStep}$$

$$\frac{\kappa(u) = \mathsf{ResourceDef}(\rho, s) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}})}{\sigma, \kappa, v_{\mathrm{C}}, u\ \{w : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{scope}\ (\alpha\ \texttt{def})\ \{\$\mathit{title} = w \sqcup s' \sqcup s\}}\ \textsc{Def}$$

A.3.4.9 Include

$$\frac{\kappa(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = merge(\rho, \varepsilon) \qquad \beta \,\mathsf{baseof}_\kappa\, \alpha}{\sigma, \kappa, v_\mathrm{C}, \texttt{include}\ a \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa[a : \mathsf{DeclaredClass}(\beta)], v_\mathrm{C}, \texttt{scope}\ (::a)\ s' \llcorner s} \textsc{IncU}$$

$$\frac{\kappa(a) = \mathsf{DeclaredClass}(\beta)}{\sigma, \kappa, v_\mathrm{C}, \texttt{include}\ a \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa, v_\mathrm{C}, \texttt{skip}} \textsc{IncD}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\sigma, \kappa, v_\mathrm{C}, \texttt{include}\ a \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa, v_\mathrm{C}, \texttt{include}\ b\ \texttt{include}\ a} \textsc{IncPU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{DeclaredClass}(\beta) \qquad s' = merge(\rho, \varepsilon)}{\sigma, \kappa, v_\mathrm{C}, \texttt{include}\ a \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa[a : \mathsf{DeclaredClass}(::b)], v_\mathrm{C}, \texttt{scope}\ (::a)\ \{s' \llcorner s\}} \textsc{IncPD}$$

A.3.4.10 Resource-like class declarations

$$\frac{\kappa(a) = \mathsf{ClassDef}(c_{opt}, \rho, S) \qquad \sigma, \kappa, v_\mathrm{C}, H \xrightarrow{\alpha}_\mathsf{H} H'}{\sigma, \kappa, v_\mathrm{C}, \texttt{class}\ \{a : H\} \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa, v_\mathrm{C}, \texttt{class}\ \{a : H'\}} \textsc{CDecStep}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = merge(\rho, v_\mathrm{H}) \qquad \beta \,\mathsf{baseof}_\kappa\, \alpha}{\sigma, \kappa, v_\mathrm{C}, \texttt{class}\ \{a : v_\mathrm{H}\} \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa[a : \mathsf{DeclaredClass}(\beta)], v_\mathrm{C}, \texttt{scope}\ (::a)\ s' \llcorner s} \textsc{CDecU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\sigma, \kappa, v_\mathrm{C}, \texttt{class}\ \{a : v_\mathrm{H}\} \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa, v_\mathrm{C}, \texttt{include}\ b\ \texttt{class}\ \{a : v_\mathrm{H}\}} \textsc{CDecPU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{DeclaredClass}(\beta) \qquad s' = merge(\rho, v_\mathrm{H})}{\sigma, \kappa, v_\mathrm{C}, \texttt{class}\ \{a : v_\mathrm{H}\} \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa[a : \mathsf{DeclaredClass}(::b)], v_\mathrm{C}, \texttt{scope}\ (::a)\ \{s' \llcorner s\}} \textsc{CDecPD}$$

A.3.4.11 Scope

$$\frac{\alpha \in \{::, ::a, ::\texttt{nd}\} \qquad \sigma, \kappa, v_\text{C}, s \xrightarrow{\alpha}_\text{s} \sigma', \kappa', v'_\text{C}, s'}{\sigma, \kappa, v_\text{C}, \texttt{scope}\ \alpha\ s \xrightarrow{\alpha'}_\text{s} \sigma', \kappa', v'_\text{C}, \texttt{scope}\ \alpha\ s'}\ \textsc{ScopeStep}$$

$$\frac{\sigma, \kappa, v_\text{C}, s \xrightarrow{\alpha\ \texttt{def}}_\text{s} \sigma', \kappa', v'_\text{C}, s'}{\sigma, \kappa, v_\text{C}, \texttt{scope}\ (\alpha\ \texttt{def})\ s \xrightarrow{\alpha}_\text{s} \sigma', \kappa', v'_\text{C}, \texttt{scope}\ (\alpha\ \texttt{def})\ s'}\ \textsc{DefScopeStep}$$

$$\frac{\alpha \in \{::, ::a, ::\texttt{nd}\}}{\sigma, \kappa, v_\text{C}, \texttt{scope}\ \alpha\ \texttt{skip} \xrightarrow{\beta}_\text{s} \sigma, \kappa, v_\text{C}, \texttt{skip}}\ \textsc{ScopeDone}$$

$$\frac{}{\sigma, \kappa, v_\text{C}, \texttt{scope}\ (\alpha\ \texttt{def})\ \texttt{skip} \xrightarrow{\alpha}_\text{s} \mathit{clear}(\sigma, \alpha\ \texttt{def}), \kappa, v_\text{C}, \texttt{skip}}\ \textsc{DefScopeDone}$$

A.3.5 Manifests ($\sigma, \kappa, v_\text{C}, m \xrightarrow{N}_\text{m} \sigma', \kappa', v'_\text{C}, m'$)

A.3.5.1 Top-Level Statements

$$\frac{\sigma, \kappa, v_\text{C}, s \xrightarrow{::}_\text{s} \sigma', \kappa', v'_\text{C}, s'}{\sigma, \kappa, v_\text{C}, s \xrightarrow{N}_\text{m} \sigma', \kappa', v'_\text{C}, s'}\ \textsc{TopScope}$$

A.3.5.2 Sequential Composition

$$\frac{\sigma, \kappa, v_\text{C}, m_1 \xrightarrow{N}_\text{m} \sigma', \kappa', v'_\text{C}, m'_1}{\sigma, \kappa, v_\text{C}, m_1 \llcorner m_2 \xrightarrow{N}_\text{m} \sigma', \kappa', v'_\text{C}, m'_1 \llcorner m_2}\ \textsc{MSeqStep}$$

$$\frac{}{\sigma, \kappa, v_\text{C}, \texttt{skip} \llcorner m \xrightarrow{N}_\text{m} \sigma, \kappa, v_\text{C}, m}\ \textsc{MSeqSkip}$$

A.3.5.3 Node Definitions

The predicate nodeMatch abstracts over the details of matching values against node specifications.

$$\frac{\mathsf{nodeMatch}(N, Q)}{\sigma, \kappa, v_\text{C}, \texttt{node}\ Q\ \{s\} \xrightarrow{N}_\text{m} \sigma, \kappa, v_\text{C}, \texttt{scope}\ (::\texttt{nd})\ s}\ \textsc{NodeMatch}$$

$$\frac{\neg\mathsf{nodeMatch}(N, Q)}{\sigma, \kappa, v_\text{C}, \texttt{node}\ Q\ \{s\} \xrightarrow{N}_\text{m} \sigma, \kappa, v_\text{C}, \texttt{skip}}\ \textsc{NodeNoMatch}$$

A.3.5.4 Defined resource types

$$\frac{u \notin dom(\kappa)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{define}\ u\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[u : \mathsf{ResourceDef}(\rho, s)], v_{\mathrm{C}}, \texttt{skip}}\ \textsc{RDef}$$

A.3.5.5 Class Definitions

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ a\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[a : \mathsf{ClassDef}(\bot, \varepsilon, s)], v_{\mathrm{C}}, \texttt{skip}}\ \textsc{CDef}$$

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ a\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[a : \mathsf{ClassDef}(b, \varepsilon, s)], v_{\mathrm{C}}, \texttt{skip}}\ \textsc{CDefI}$$

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ a\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[a : \mathsf{ClassDef}(\bot, \rho, s)], v_{\mathrm{C}}, \texttt{skip}}\ \textsc{CDefP}$$

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ a\ (\rho)\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[a : \mathsf{ClassDef}(b, \rho, s)], v_{\mathrm{C}}, \texttt{skip}}\ \textsc{CDefPI}$$

Appendix B

Where Provenance

B.1 Propagation of Annotations in the Evaluation

B.1.1 Expressions

Variables

$$\frac{x \in dom(\sigma_\alpha)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x \xrightarrow{\alpha} \sigma_\alpha(x)}\ \textsc{LVar}$$

$$\frac{x \notin dom(\sigma_\alpha) \qquad \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x \xrightarrow{\beta} v^\beta \qquad \beta\ \mathsf{parentof}_\kappa\ \alpha}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x \xrightarrow{\alpha} v^\beta}\ \textsc{PVar}$$

$$\frac{x \in dom(\sigma_{::})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \${::}x \xrightarrow{\alpha} \sigma_{::}(x)}\ \textsc{TVar} \qquad \frac{x \in dom(\sigma_{::a})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \${::}a :: x \xrightarrow{\alpha} \sigma_{::a}(x)}\ \textsc{QVar}$$

B.1.2 Resources

The rule RName propagates the evaluation of a resource name expression e to the rule evaluating expression e. Then e will be evaluated to an annotated data value v^β. For every attribute and expression pair in a resource, the rule Rbody-Exp evaluate e. When e is evaluated to an annotated data value v^β, the rule RbodyStep will evaluate the next attribute and expression pair in the resource body.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, x \Rightarrow e, H \xrightarrow{\alpha}_{\mathsf{H}} x \Rightarrow e', H} \textsc{RbodyExp}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, x \Rightarrow v^{\beta}, H \xrightarrow{\alpha}_{\mathsf{H}} x \Rightarrow v^{\beta}, H'} \textsc{RbodyStep}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e : H \xrightarrow{\alpha}_{\mathsf{R}} e' : H} \textsc{RName} \qquad \frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta} : H \xrightarrow{\alpha}_{\mathsf{R}} v^{\beta} : H'} \textsc{Rbody}$$

B.1.3 Statements

Expression statements An expression can occur as a statement. Its value is ignored.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e'} \textsc{ExprStep} \qquad \frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}} \textsc{Expr}$$

Sequential composition

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s_1 \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'_1}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s_1 \mathbin{␣} s_2 \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'_1 \mathbin{␣} s_2} \textsc{SeqStep}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip} \mathbin{␣} s \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s} \textsc{SeqSkip}$$

Assignment

Unless

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{unless}\ e\ \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{unless}\ e'\ \{s\}} \textsc{UnlessStep}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{unless}\ \mathtt{true}^{\beta}\ \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}},\ \mathtt{skip}} \textsc{UnlessT}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{unless}\ \mathtt{false}^{\beta}\ \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}},\ s} \textsc{UnlessF}$$

Case As with selectors, the predicate caseMatch abstracts over the details of pattern-matching. Again, in real Puppet the `default` case is always checked last, regardless of the order of the cases.

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ e\ \{C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ e'\ \{C\}}\ \textsc{CaseStep1}$$

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ v^{\beta}\ \{e : \{s\}\_C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ v^{\beta}\ \{e' : \{s\}\_C\}}\ \textsc{CaseStep2}$$

$$\frac{\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\}\_C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},s}\ \textsc{CaseMatch}$$

$$\frac{\neg\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\}\_C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ v^{\beta}\ \{C\}}\ \textsc{CaseNoMatch}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ v^{\beta}\ \{\varepsilon\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{skip}}\ \textsc{CaseDone}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{case}\ v^{\beta}\ \{\mathtt{default} : \{s\}\_C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},s\}}\ \textsc{CaseDefault}$$

Resource declarations

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e : H \xrightarrow{\alpha}_{\mathsf{R}} e' : H'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},t\ \{e : H\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},t\ \{e' : H'\}}\ \textsc{ResStep}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\widehat{v_{\mathrm{R}}} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}}\_\widehat{v_{\mathrm{R}}},\mathtt{skip}}\ \textsc{ResDecl}$$

Defined resource types

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\{e : H\} \xrightarrow{\alpha}_{\mathsf{R}} \{e' : H'\}}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},u\ \{e : H\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},u\ \{e' : H'\}}\ \textsc{DefStep}$$

$$\frac{\widehat{\kappa}(u) = \mathsf{ResourceDef}(\rho, s) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}})}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},u\ \{w^{\beta} : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{scope}\ (\alpha\ \mathtt{def})\ \{\$\mathit{title} = w^{\beta}\_s'\_s\}}\ \textsc{Def}$$

Include

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = \mathit{merge}(\rho, \varepsilon) \qquad \beta \,\mathsf{baseof}_{\widehat{\kappa}}\, \alpha}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}{}'_{\mathsf{s}} \widehat{\sigma}, \kappa[a : \mathsf{DeclaredClass}(\beta)], \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ s' \llcorner s} \textsc{IncU}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{DeclaredClass}(\beta)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}} \textsc{IncD}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \widehat{\kappa}(b) = \mathsf{ClassDef}(c_{\mathit{opt}}, \rho', s')}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ b\ \texttt{include}\ a} \textsc{IncPU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{DeclaredClass}(\beta) \qquad s' = \mathit{merge}(\rho, \varepsilon)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa[a : \mathsf{DeclaredClass}(::b)], \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ \{s' \llcorner s\}} \textsc{IncPD}$$

B.1.3.0.1 Resource-like class declarations

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(c_{\mathit{opt}}, \rho, S) \qquad \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ \{a : H\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ \{a : H'\}} \textsc{CDecStep}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}}) \qquad \beta \,\mathsf{baseof}_{\widehat{\kappa}}\, \alpha}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \kappa[a : \mathsf{DeclaredClass}(\beta)], \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ s' \llcorner s} \textsc{CDecU}$$

$$\frac{\widehat{\kappa(a)} = \mathsf{ClassDef}(b, \rho, s) \qquad \widehat{\kappa(b)} = \mathsf{ClassDef}(c_{\mathit{opt}}, \rho', s')}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ b\ \texttt{class}\ \{a : v_{\mathrm{H}}\}} \textsc{CDecPU}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{DeclaredClass}(\beta) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \kappa[a : \mathsf{DeclaredClass}(::b)], \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ \{s' \llcorner s\}} \textsc{CDecPD}$$

Scope

$$\frac{\alpha \in \{::, ::a, ::\mathtt{nd}\} \qquad \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{scope}\ \alpha\ s \xrightarrow{\alpha'}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, \mathtt{scope}\ \alpha\ s'} \textsc{ScopeStep}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha\ \mathtt{def}}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{scope}\ (\alpha\ \mathtt{def})\ s \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, \mathtt{scope}\ (\alpha\ \mathtt{def})\ s'} \textsc{DefScopeStep}$$

$$\frac{\alpha \in \{::, ::a, ::\mathtt{nd}\}}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{scope}\ \alpha\ \mathtt{skip} \xrightarrow{\beta}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}} \textsc{ScopeDone}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{scope}\ (\alpha\ \mathtt{def})\ \mathtt{skip} \xrightarrow{\alpha}_{\mathsf{s}} \mathit{clear}(\widehat{\sigma}, \alpha\ \mathtt{def}), \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}} \textsc{DefScopeDone}$$

B.1.4 Manifests

Top-Level Statements

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{::}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'} \textsc{TopScope}$$

Sequential Composition

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, m_1 \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m'_1}{\widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m_1 \_ m_2 \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m'_1 \_ m_2} \textsc{MSeqStep}$$

$$\frac{}{\widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, \mathtt{skip} \_ m \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m} \textsc{MSeqSkip}$$

Node Definitions The predicate nodeMatch abstracts over the details of matching values against node specifications.

$$\frac{\mathsf{nodeMatch}(N, Q)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{node}\ Q\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{scope}\ (::\mathtt{nd})\ s} \textsc{NodeMatch}$$

$$\frac{\neg\mathsf{nodeMatch}(N, Q)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{node}\ Q\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}} \textsc{NodeNoMatch}$$

Defined resource types

$$\frac{u \notin dom(\widehat{\kappa})}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{define}\ u\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma},\widehat{\kappa}[u : \mathsf{ResourceDef}(\rho,s)],\widehat{v_{\mathrm{C}}},\texttt{skip}}\ \textsc{RDef}$$

Class Definitions

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{class}\ a\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma,\widehat{\kappa}[a : \mathsf{ClassDef}(\bot,\varepsilon,s)],\widehat{v_{\mathrm{C}}},\texttt{skip}}\ \textsc{CDef}$$

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{class}\ a\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma,\widehat{\kappa}[a : \mathsf{ClassDef}(b,\varepsilon,s)],\widehat{v_{\mathrm{C}}},\texttt{skip}}\ \textsc{CDefI}$$

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{class}\ a\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma,\widehat{\kappa}[a : \mathsf{ClassDef}(\bot,\rho,s)],\widehat{v_{\mathrm{C}}},\texttt{skip}}\ \textsc{CDefP}$$

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\texttt{class}\ a\ (\rho)\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma},\widehat{\kappa}[a : \mathsf{ClassDef}(b,\rho,s)],\widehat{v_{\mathrm{C}}},\texttt{skip}}\ \textsc{CDefPI}$$

B.2 Proofs of the invariant in Expression Evaluation

Theorem B.2.1 (Invariant for expressions)**.** If $\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'$ then $labels(e') \subseteq labels(e) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$.

Proof. Prove by induction.

1 Variables

$$\frac{x \in dom(\sigma_\alpha)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\$x \xrightarrow{\alpha} \sigma_\alpha(x)}\ \textsc{LVar}$$

$$\frac{x \notin dom(\sigma_\alpha)\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\$x \xrightarrow{\beta} v^l\beta\,\mathsf{parentof}_\kappa\,\alpha}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\$x \xrightarrow{\alpha} v^l}\ \textsc{PVar}$$

$$\frac{x \in dom(\sigma_{::})}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\$::x \xrightarrow{\alpha} \sigma_{::}(x)}\ \textsc{TVar}$$

The proof is similar as case LVar.

$$\frac{x \in dom(\sigma_{::a})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$::a :: x \xrightarrow{\alpha} \sigma_{::a}(x)}\ \textsc{QVar}$$

The proof is similar as case LVar.

2 Comparison expressions

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 > e_2 \xrightarrow{\alpha} e_1' > e_2}\ \textsc{CompLeft}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta} > e \xrightarrow{\alpha} v^{\beta} > e'}\ \textsc{CompRight}$$

$$\frac{v_1 >_{\mathbb{Z}} v_2}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v_1^{\beta_1} > v_2^{\beta_2} \xrightarrow{\alpha} \mathtt{true}^{\perp}}\ \textsc{CompValueI}$$

By definition, we have $labels(v_1^{\beta_1} > v_2^{\beta_2}) = labels(v_1^{\beta_1}) \cup labels(v_2^{\beta_2}) = \{v_1^{\beta_1}, v_2^{\beta_2}\}$ and $labels(\mathtt{true}^{\perp}) = \emptyset$. Then $\emptyset \subseteq \{v_1^{\beta_1}, v_2^{\beta_2}\} \cup labels(\hat{\sigma})$, i.e. $labels(\mathtt{true}^{\perp}) \subseteq labels(v_1^{\beta_1} > v_2^{\beta_2}) \cup labels(\hat{\sigma})$. The invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, v_1^{a_1} > v_2^{\beta_2} \xrightarrow{\alpha} \mathtt{true}^{\perp}$.

$$\frac{v_1 <=_{\mathbb{Z}} v_2}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v_1^{\beta_1} > v_2^{\beta_2} \xrightarrow{\alpha} \mathtt{false}^{\perp}}\ \textsc{CompValueII}$$

The proof is similar as the rule CompValueI.

3 Boolean expressions

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1\,\mathtt{and}\,e_2 \xrightarrow{\alpha} e_1'\,\mathtt{and}\,e_2}\ \textsc{AndLeft}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{false}^{\beta}\,\mathtt{and}\,e \xrightarrow{\alpha} \mathtt{false}^{\perp}}\ \textsc{AndRightI}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{true}^{l}\,\mathtt{and}\,e \xrightarrow{\alpha} \mathtt{true}^{\beta}\,\mathtt{and}\,e'}\ \textsc{AndRightII}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{true}^{\beta_1}\,\mathtt{and}\,\mathtt{true}^{\beta_2} \xrightarrow{\alpha} \mathtt{true}^{\perp}}\ \textsc{AndValue}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, !\,e \xrightarrow{\alpha} !\,e'}\ \textsc{NotSTEP}$$

By definition, we have $labels(!e) = labels(e)$ and $labels(!e') = labels(e')$. By induction hypothesis, we know the invariant holds for $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'$, i.e.

$labels(e') \subseteq labels(e) \cup labels(\hat{\sigma})$. Then $labels(!e') \subseteq labels(!e) \cup labels(\hat{\sigma})$. That is the invariant holds for this evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, !e \xrightarrow{\alpha} !e'$.

B.3 Proofs of Invariant in Statement Evaluation

Theorem B.3.1 (Invariant of Statement Evaluation)**.** When evaluating a statement s in μPuppet such that $\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha}_s \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'$, the subsumption on the sets of annotated data values

$$labels(\widehat{\sigma}) \cup labels(\widehat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \cup labels(s) \supseteq$$
$$labels(\widehat{\sigma'}) \cup labels(\widehat{\kappa'}) \cup labels(\widehat{v'_{\mathrm{C}}}) \cup labels(s')$$

holds.

Proof. Prove by induction.

1 Expression statements

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e'} \textsc{ExprStep}$$

By induction hypothesis $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'$, we know that the invariant holds on $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'$, i.e. $labels(e') \subseteq labels(e) \cup labels(\hat{\sigma})$. Then we have $labels(e') \subseteq labels(e) \cup labels(\hat{\sigma})$. Since $\hat{\sigma}$, $\hat{\kappa}$ and $\hat{v_{\mathrm{C}}}$, have not changed before and after the evaluation step, we have $labels(e') \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(e) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. That is the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, e \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, e'$.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}} \textsc{Expr}$$

2 Sequential composition

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s_1 \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'_1}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s_1 \texttt{;} s_2 \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'_1 \texttt{;} s_2} \textsc{SeqStep}$$

By induction hypothesis, the invariant holds for this evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, s_1 \xrightarrow{\alpha}_s \hat{\sigma'}, \hat{\kappa'}, \hat{v'_{\mathrm{C}}}, s'_1$, i.e. $labels(s'_1) \cup labels(\hat{\sigma'}) \cup labels(\hat{\sigma'}) \cup labels(\hat{v_{\mathrm{C}}}') \subseteq labels(s_1) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. By definition, we have

$labels(s_1 s_2) = labels(s_1) \cup labels(s_2)$ and $labels(s_1' s_2) = labels(s_1') \cup labels(s_2)$. Then we have $labels(s_1') \cup labels(s_2) \subseteq labels(s_1) \cup labels(\hat{\sigma}) \cup labels(s_2) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C})$, i.e. $labels(s_1' s_2) \cup labels(\hat{\sigma'}) \cup labels(\hat{\kappa'}) \cup labels(\hat{v_C}') \subseteq labels(s_1 s_2) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C})$. The invariant for the evaluation $\hat{\sigma}, \hat{\kappa}, \hat{v_C}, s_1\ s_2 \xrightarrow{\alpha}_s \hat{\sigma'}, \hat{\kappa'}, \hat{v_C'}, s_1'\ s_2$ holds. This rule is proved with respected to the invariant.

$$\frac{}{\sigma, \kappa, v_C, \texttt{skip}\_s \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_C, s} \textsc{SeqSkip}$$

By definition, $labels(\texttt{skip}\ s) = labels(\texttt{skip}) \cup labels(s) = \emptyset \cup labels(s) = labels(s)$. Then we have $labels(s) \subseteq labels(s)$, i.e. $labels(s) \subseteq labels(\texttt{skip}\ s)$. Since $\hat{\sigma}$,$\hat{\kappa}$ and $\hat{v_C}$ have not changed before and after the evaluation, we have $labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C}) \subseteq labels(\texttt{skip}\ s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C})$. Then the invariant holds for the evaluation $\hat{\sigma}, \hat{\kappa}, \hat{v_C}, \texttt{skip}\ s \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_C}, s$. This rule holds with respected to the invariant.

3 Assignment

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, \$x = e \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, \$x = e'} \textsc{AssignStep}$$

By induction hypothesis, we know the invariant holds for $\hat{\sigma}, \hat{\kappa}, \hat{v_C}, e \xrightarrow{\alpha} e'$, i.e. $labels(e') \subseteq labels(e) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C})$. By definition, we have $labels(\$x = e) = labels(e)$ and $labels(\$x = e') = labels(e')$. Since $labels(e') \subseteq labels(e) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C})$ and the environments have not changed before and after the evaluation, we have $labels(e') \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C}) \subseteq labels(e) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C})$, i.e. $labels(\$x = e') \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C}) \subseteq labels(\$x = e) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C})$. Then the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_C}, \$x = e \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_C}, \$x = e'$. This induction rule holds w.r.t. the invariant.

$$\frac{x \notin dom(\sigma_\alpha)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, \$x = v^\beta \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}[(\alpha, x) : v^\beta], \widehat{\kappa}, \widehat{v_C}, \texttt{skip}} \textsc{Assign}$$

By definition, $labels(\$x = v^\beta) = labels(\$x) \cup labels(v^\beta) = \emptyset \cup \{v^\beta\} = \{v^\beta\}$, $labels(\texttt{skip}) = \emptyset$ and $labels(\hat{\sigma}[(\alpha, x) : v^\beta]) = labels(\hat{\sigma}) \cup \{v^\beta\}$. Since $\hat{\kappa}, \hat{v_C}$ did not change before and after the evaluation, we have $labels(\hat{\sigma}) \cup \{v^\beta\} \cup labels(\hat{\kappa}) \cup labels(\hat{v_C}) \subseteq labels(\hat{\sigma}) \cup \{v^\beta\} \cup labels(\hat{\kappa}) \cup labels(\hat{v_C})$, i.e. $labels(\hat{\sigma}[(\alpha, x) : v^\beta]) \cup labels(\texttt{skip}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C}) \subseteq$

$labels(\hat{\sigma}) \cup labels(\$x = v^{\beta}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. Thus the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \$x = v^{\beta} \xrightarrow{\alpha}_{s} \hat{\sigma}[(\alpha, x) : v^{\beta}], \hat{\kappa}, \hat{v_{\mathrm{C}}}, \mathtt{skip}$. This rule holds w.r.t the invariant.

4 If

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{if}\ e'\ \{s_1\}\ \mathtt{else}\ \{s_2\}}\ \textsc{IfStep}$$

By induction hypothesis, we know the invariant holds for $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'$, i.e. $labels(e') \subseteq labels(e) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa})$. By the definition, $labels(\mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\}) = labels(e) \cup labels(s_1) \cup labels(s_2)$ and $labels(\mathtt{if}\ e'\ \{s_1\}\ \mathtt{else}\ \{s_2\}) = labels(e') \cup labels(s_1) \cup labels(s_2)$. We have $labels(s') \subseteq labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. Since $\hat{\sigma}$, $\hat{\kappa}$ and $\hat{v_{\mathrm{C}}}$ have not changed before and after the evaluation, $labels(e') \cup labels(s_1) \cup labels(s_2) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(e) \cup labels(s_1) \cup labels(s_2) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, i.e. $labels(\mathtt{if}\ e'\ \{s_1\}\ \mathtt{else}\ \{s_2\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. Then the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\} \xrightarrow{\alpha}_{s} \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \mathtt{if}\ e'\ \{s_1\}\ \mathtt{else}\ \{s_2\}$. Then the induction rule holds w.r.t. the invariant.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{if}\ \mathtt{true}^{\beta}\ \{s_1\}\ \mathtt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s_1}\ \textsc{IfT}$$

By definition, we have $labels(\mathtt{if}\ \mathtt{true}^{\beta}\ \{s_1\}\ \mathtt{else}\ \{s_2\}) = labels(\mathtt{true}^{\beta}) \cup labels(s_1) \cup labels(\mathtt{else}\ \{s_2\}) = \{\mathtt{true}^{\beta}\} \cup labels(s_1) \cup labels(s_2)$. Then $labels(s_2) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq \{\mathtt{true}^{\beta}\} \cup labels(s_1) \cup labels(s_2) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, i.e. $labels(s_2) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\mathtt{if}\ \mathtt{true}^{\beta}\ \{s_1\}\ \mathtt{else}\ \{s_2\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. Then the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \mathtt{if}\ \mathtt{true}^{\beta}\ \{s_1\}\ \mathtt{else}\ \{s_2\} \xrightarrow{\alpha}_{s} \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, s_2$. Then the induction rule holds w.r.t. the invariant.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{if}\ \mathtt{false}^{\beta}\ \{s_1\}\ \mathtt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s_2}\ \textsc{IfF}$$

The proof is similar as above.

5 Unless

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{unless}\ e\ \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{unless}\ e'\ \{s\}}\ \textsc{UnlessStep}$$

By induction hypothesis, we know the invariant holds for $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'$, i.e. $labels(e') \subseteq labels(e) \cup labels(\hat{\sigma}) cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. By definition, we have $labels(\texttt{unless}\, e\, \{s\}) = labels(e) \cup labels(s)$ and $labels(\texttt{unless}\, e'\, \{s\}) = labels(e') \cup labels(s)$. We have $labels(e') \subseteq labels(e) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. Since $\hat{\sigma}$, $\hat{\kappa}$, and $\hat{v_{\mathrm{C}}}$, have not changed before and after the evaluation, we have $labels(e') \cup labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(e) \cup labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, i.e. $labels(\texttt{unless}\, e'\, \{s\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\texttt{unless}\, e\, \{s\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. Then the invariant holds for the evaluation $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{unless}\, e\, \{s\} \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{unless}\, e'\, \{s\}$. Then the induction rule holds w.r.t. the invariant.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{unless}\, \texttt{true}^{\beta}\, \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}},\, \texttt{skip}}\ \textsc{UnlessT}$$

By definition, we know $labels(\texttt{unless}\, \texttt{true}^{\beta}\, \{s\}) = labels(\texttt{true}^{a}) \cup labels(s) = \{\texttt{true}^{\beta}\} \cup labels(s)$ and $labels(\texttt{skip}) = \emptyset$. Then we have $\emptyset \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq \{\texttt{true}^{\beta}\} \cup labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$ i.e. $labels(\texttt{skip}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\texttt{unless}\, \texttt{true}^{\beta}\, \{s\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, i.e. the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{unless}\, \texttt{true}^{\beta}\, \{s\} \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}},\, \texttt{skip}$. Then this induction rule holds w.r.t. the invariant.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{unless}\, \texttt{false}^{\beta}\, \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}},\, s}\ \textsc{UnlessF}$$

The proof is similar as above.

6 Case

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{case}\, e\, \{C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{case}\, e'\, \{C\}}\ \textsc{CaseStep1}$$

By inductive hypothesis, we know the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'$, i.e. $labels(e') \subseteq labels(e) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. By the definition of $labels(\texttt{case}\, e\, \{C\})$, we know $labels(\texttt{case}\, e\, \{C\}) = labels(e) \cup labels(C)$ and similarly $labels(\texttt{case}\, e'\, \{C\}) = labels(e') \cup labels(C)$. We have $labels(e') \subseteq labels(e) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. Then $labels(e') \cup labels(C) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(e) \cup labels(C) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, i.e. $labels(\texttt{case}\, e'\, \{C\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup$

$labels(\hat{v_C}) \subseteq labels(\texttt{case}\ e\ \{C\}) \cup \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C})$. Then the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_C}, \texttt{case}\, e\, \{C\} \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_C}, \texttt{case}\, e'\, \{C\}$. The induction rule holds w.r.t. the invariant.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, \texttt{case}\ v^\beta\ \{e : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, \texttt{case}\ v^\beta\ \{e' : \{s\} \llcorner C\}}\ \textsc{CaseStep2}$$

$$\frac{\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, \texttt{case}\ v^\beta\ \{v_1^{\beta_1} : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, s}\ \textsc{CaseMatch}$$

By definition, we have $labels(\texttt{case}\ v^\beta\ \{v_1^{a'} : \{S\}\ C\}) = labels(v^a) \cup labels(v_1^{\beta'}) \cup labels(s) \cup labels(C)$. Since $labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C}) \subseteq labels(v^\beta) \cup labels(v_1^{\beta'}) \cup labels(s) \cup labels(C) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C})$, i.e. $labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C}) \subseteq labels(v^\beta) \cup labels(\texttt{case}\, v^\beta\, \{v_1^{a'} : \{S\}\, C\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C})$, the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_C}, \texttt{case}\, v^\beta\, \{v_1^{\beta'} : \{S\}\, C\} \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_C}, S$. This induction rule holds w.r.t. the invariant.

$$\frac{\neg\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, \texttt{case}\ v^\beta\ \{v_1^{\beta_1} : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, \texttt{case}\ v^\beta\ \{C\}}\ \textsc{CaseNoMatch}$$

Omitted.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, \texttt{case}\ v^\beta\ \{\varepsilon\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, \texttt{skip}}\ \textsc{CaseDone}$$

By definition, we know $labels(\texttt{case}\, v^\beta\, \{\}) = labels(v^\beta)$ and $labels(\texttt{skip}) = \emptyset$. Since $labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C}) \cup \emptyset \subseteq labels(v^\beta) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C})$, we have $labels(\texttt{skip}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C}) \subseteq labels(\texttt{case}\, v^\beta\, \{\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_C})$. Then the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_C}, \texttt{case}\ v^\beta\ \{\ \} \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_C}, \texttt{skip}$. This induction rule holds w.r.t. the invariant.

7 Resource declarations

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, e : H \xrightarrow{\alpha}_{\mathsf{R}} e' : H'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, t\ \{e : H\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, t\ \{e' : H'\}}\ \textsc{ResStep}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, \widehat{v_R} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_C} \llcorner \widehat{v_R}, \texttt{skip}}\ \textsc{ResDecl}$$

8 Defined resource types

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, \{e : H\} \xrightarrow{\alpha}_{\mathsf{R}} \{e' : H'\}}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, u\ \{e : H\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_C}, u\ \{e' : H'\}}\ \textsc{DefStep}$$

Omitted.

$$\frac{\widehat{\kappa}(u) = \mathsf{ResourceDef}(\rho, s) s' = merge(\rho, v_{\mathrm{H}})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, u\ \{w : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{scope}\ (\alpha\ \mathtt{def})\ \{\$title = w \llcorner s' \llcorner s\}}\ \textsc{Def}$$

By definition, $labels(u\ \{w : v_{\mathrm{H}}\}) = \{w\} \cup labels(v_{\mathrm{H}})$ and $labels(\mathtt{scope}\ (\alpha\ \mathtt{def})\ \{\$title = w \llcorner s' \llcorner s) = labels(\widehat{v_{\mathrm{H}}}) \cup labels(s) \cup labels(s')$. We know $labels(\mathsf{ResourceDef}(\rho, s)) \subseteq labels(\hat{\kappa})$, i.e. $labels(\rho) \cup labels(s) \subseteq labels(\hat{\kappa})$. We also have $labels(s') \subseteq labels(\rho) \cup labels(v_{\mathrm{H}})$ by the definition of function $merge$. Then $labels(s') \cup labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \subseteq labels(v_{\mathrm{H}}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}})$, i.e. $labels(\mathtt{scope}\ \mathtt{def}\ s'\ s) \cup labels(\hat{\sigma}) \cup labels(\widehat{v_{\mathrm{C}}}) \cup labels(\hat{\kappa}) \subseteq labels(u\ \{w : v_{\mathrm{H}}\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}})$. Then the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \widehat{v_{\mathrm{C}}}, u\ \{w : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{s} \hat{\sigma}, \hat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{scope}\ (\alpha\ \mathtt{def})\ \{\$title = w \llcorner s' \llcorner s\}$. Then this rule holds w.r.t. the invariant.

9 Include

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(\bot, \rho, s) s' = merge(\rho, \varepsilon) \beta\ \mathsf{baseof}_{\widehat{\kappa}}\ \alpha}{\hat{\sigma}, \hat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{include}\ a \xrightarrow{\alpha}{}'_{\mathsf{s}} \widehat{\sigma}, \hat{\kappa}[a : \mathsf{DeclaredClass}(\beta)], \widehat{v_{\mathrm{C}}}, \mathtt{scope}\ (::a)\ s' \llcorner s}\ \textsc{IncU}$$

By definition, we have $labels(\mathtt{include}\ a) = \emptyset$ and $labels(\mathtt{scope}\ (::a)\ s' \llcorner s) = labels(s) \cup labels(s')$. By the definition of $labels(\hat{\kappa})$, we have $labels(\mathsf{ClassDef}(\bot, \rho, s)) \subseteq labels(\hat{\kappa})$, i.e. $labels(\rho) \cup labels(s) \subseteq labels(\hat{\kappa})$, and $labels(\hat{\kappa}(a \to \mathsf{DeclaredClass}(\bot))) = labels(\hat{\kappa}) \setminus labels(\mathsf{ClassDef}(\bot, \rho, S)) = labels(\hat{\kappa}) \setminus (labels(\rho) \cup labels(s))$. Then $labels(\hat{\kappa}) \setminus (labels(\rho) \cup labels(s)) \cup labels(\rho) \cup labels(s) \cup labels(\hat{\sigma}) \cup labels(\widehat{v_{\mathrm{C}}}) \subseteq \emptyset \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}})$, i.e. $labels(a \to \mathsf{DeclaredClass}(\bot)) \cup labels(\mathtt{scope}\ (::a)\ \rho\ s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \subseteq labels(\mathtt{include}\ a) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}})$. Then the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{include}\ a \xrightarrow{\alpha}{}'_{\mathsf{s}} \hat{\sigma}, \hat{\kappa}[a : \mathsf{DeclaredClass}(\beta)], \widehat{v_{\mathrm{C}}}, \mathtt{scope}\ (::a)\ s' \llcorner s$. Then this inference rule holds w.r.t. the invariant.

$$\frac{\widehat{\kappa}(a) = \mathsf{DeclaredClass}(\beta)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}}\ \textsc{IncD}$$

By definition, we know $labels(\mathtt{include}\ a) = \emptyset$ and $labels(\mathtt{skip}) = \emptyset$. Since $\emptyset \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \subseteq \emptyset \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}})$, we have $labels(\mathtt{skip}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}}) \subseteq labels(\mathtt{include}\ a) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\widehat{v_{\mathrm{C}}})$. Then the invari-

ant holds for the evaluation step
$\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{skip}$. This induction rule holds w.r.t. the invariant.

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \widehat{\kappa}(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ b\ \texttt{include}\ a} \textsc{IncPU}$$

By definition, we have $labels(\texttt{include}\ a) = \emptyset$ and $labels(\texttt{include}\ b\ \texttt{include}\ a) = \emptyset$. Since $\emptyset \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq \emptyset \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, we have $labels(\texttt{include}\ b\ \texttt{include}\ a) \cup labels(\hat{\sigma}) \cup labels(\kappa) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\texttt{include}\ a) \cup labels(\hat{\sigma}) \cup labels(\kappa) \cup labels(\hat{v_{\mathrm{C}}})$, i.e. the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{include}\ b\ \texttt{include}\ a$. Then this inference rule holds w.r.t. the invariant.

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \kappa(b) = \mathsf{DeclaredClass}(\beta) s' = merge(\rho, \varepsilon)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa[a : \mathsf{DeclaredClass}(::b)], \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ \{s' \llcorner s\}} \textsc{IncPD}$$

By definition, we have $labels(\texttt{include}\ a) = \emptyset$ and $labels(\texttt{scope}\ (::a)\ \{s' \llcorner s\}) = labels(s) \cup labels(s')$. By the definition of $labels(\hat{\kappa})$, we know $labels(\mathsf{ClassDef}(b, \rho, s)) \subseteq labels(\hat{\kappa})$, i.e. $labels(\rho) \cup labels(s) \subseteq labels(\hat{\kappa})$, and $labels(\hat{\kappa}(a \to \mathsf{DeclaredClass}(b))) = labels(\hat{\kappa}) \setminus labels(\mathsf{ClassDef}(b, \rho, s)) = labels(\hat{\kappa}) \setminus (labels(\rho) \cup labels(s))$. Then $labels(\hat{\kappa}) \setminus (labels(\rho) \cup labels(s)) \cup labels(s') \cup labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq \emptyset \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, i.e.
$labels(\hat{\kappa}(a \to \mathsf{DeclaredClass}(b))) \cup labels(\texttt{scope}\ (::a)\ \{s' \llcorner s\}) \cup labels(\hat{\sigma}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\texttt{include}\ a) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. Then the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}(a \to \mathsf{DeclaredClass}(b)), \hat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ \{s' \llcorner s\}$. Then this rule holds w.r.t. the invariant.

10 Resource-like class declarations

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(c_{opt}, \rho, s) \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ \{a : H\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ \{a : H'\}} \textsc{CDecStep}$$

By induction hypothesis, the invariable $labels(H') \subseteq labels(H) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa})$ holds. By definition, $labels(\texttt{class}\ \{a :\ H\}) = labels(H)$ and $labels(\texttt{class}\ \{a :\ H'\}) = labels(H')$. We have $labels(H') \subseteq labels(H) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa})$. Then $labels(\texttt{class}\ \{a :\ H'\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\texttt{class}\ \{a :\ H\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, i.e. the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{class}\ \{a :$

$H\} \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{class}\ \{a : H'\}$. Then this rule holds w.r.t. the invariant.

$$\frac{\hat{\kappa}(a) = \mathsf{ClassDef}(\bot, \rho, s) s' = merge(\rho, v_{\mathrm{H}}) \beta\, \mathsf{baseof}_{\hat{\kappa}}\, \alpha}{\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \hat{\sigma}, \kappa[a : \mathsf{DeclaredClass}(\beta)], \hat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ s' \llcorner s} \text{CDecU}$$

By the definition, $labels(\texttt{class}\ \{a :\ v_{\mathrm{H}}\}) = labels(v_{\mathrm{H}})$ and $labels(\texttt{scope}\ (::a)\ s'\ s) = labels(s') \cup labels(s)$. we know $labels(\mathsf{ClassDef}(\bot, \rho, s)) \subseteq labels(\hat{\kappa})$, i.e. $labels(\rho) \cup labels(s) \subseteq labels(\hat{\kappa})$,
and $labels(\hat{\kappa}(a \to \mathsf{DeclaredClass}(\bot))) = labels(\hat{\kappa}) \setminus labels(\mathsf{ClassDef}(\bot, \rho, s)) = labels(\hat{\kappa}) \setminus (labels(\rho) \cup labels(s))$. We also have $labels(s') \subseteq labels(\rho) \cup labels(\hat{v_{\mathrm{H}}})$ by the definition of function $merge$. Then $(labels(\hat{\kappa}) \setminus (labels(\rho) \cup labels(s))) \cup labels(s') \cup labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(v_{\mathrm{H}}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, i.e. $labels(a \to \mathsf{DeclaredClass}(\bot)) \cup labels(\texttt{scope}\ (::a)\ s'\ s) \cup labels(\hat{\sigma}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\texttt{class}\ \{a : v_{\mathrm{H}}\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. Then the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}} \vdash \texttt{class}\ \{a :\ v_{\mathrm{H}}\} \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}(a \to \mathsf{DeclaredClass}(\bot)), \hat{v_{\mathrm{C}}} \vdash \texttt{scope}\ (::a)\ s'\ s$. This rule holds w.r.t. the invariant.

$$\frac{\widehat{\kappa(a)} = \mathsf{ClassDef}(b, \rho, s) \widehat{\kappa(b)} = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{include}\ b\ \texttt{class}\ \{a : v_{\mathrm{H}}\}} \text{CDecPU}$$

By definition, $labels(\texttt{class}\ \{a :\ v_{\mathrm{H}}\}) = labels(v_{\mathrm{H}})$ and $labels(\texttt{include}\ b\ \texttt{class}\ \{a :\ v_{\mathrm{H}}\}) = \emptyset \cup labels(v_{\mathrm{H}}) = labels(v_{\mathrm{H}})$. Since $labels(v_{\mathrm{H}}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(v_{\mathrm{H}}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, $labels(\texttt{include}\ b\ \texttt{class}\ \{a :\ v_{\mathrm{H}}\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup \hat{v_{\mathrm{C}}} \subseteq$
$labels(\texttt{class}\ \{a :\ v_{\mathrm{H}}\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, i.e. the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{class}\ \{a :\ v_{\mathrm{H}}\} \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{include}\ b\ \texttt{class}\ \{a :\ v_{\mathrm{H}}\}$. Then this rule holds w.r.t. the invariant.

$$\frac{\hat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \kappa(b) = \mathsf{DeclaredClass}(\beta) s' = merge(\rho, v_{\mathrm{H}})}{\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \hat{\sigma}, \kappa[a : \mathsf{DeclaredClass}(::b)], \hat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ \{s' \llcorner s\}} \text{CDecPI}$$

The proof is similar as case above.

11 Scope

$$\frac{\alpha \in \{::, ::a, ::\texttt{nd}\} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha}_{\mathrm{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ \alpha\ s \xrightarrow{\alpha'}_{\mathtt{s}} \sigma', \kappa', v'_{\mathrm{C}}, \texttt{scope}\ \alpha\ s'} \textsc{ScopeStep}$$

By induction hypothesis, we know the invariant for this evaluation $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, s \xrightarrow{\alpha}_{s} \hat{\sigma}', \hat{\kappa}', \hat{v'_{\mathrm{C}}}, s'$ holds, i.e. $labels(s') \cup labels(\hat{\sigma}') \cup labels(\kappa') \cup labels(v'_{\mathrm{C}}) \subseteq labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. By definition $labels(\texttt{scope}\ \alpha\ s) = labels(s)$ and $labels(\texttt{scope}\ \alpha\ s') = labels(s')$. We have $labels(\texttt{scope}\ \alpha\ s') \cup labels(\hat{\sigma}') \cup labels(\kappa') \cup labels(v'_{\mathrm{C}}) \subseteq labels(\texttt{scope}\ \alpha\ s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. That is the invariant holds for the evaluation $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{scope}\ \alpha\ s \xrightarrow{\alpha'}_{s} \hat{\sigma}', \hat{\kappa}', \hat{v'_{\mathrm{C}}}, \texttt{scope}\ \alpha\ s'$. Then the induction rule holds w.r.t. the invariant.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha\ \texttt{def}}_{\mathtt{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (\alpha\ \texttt{def})\ s \xrightarrow{\alpha}_{\mathtt{s}} \widehat{\sigma'}, \texttt{scope}'\ (\alpha\ \texttt{def})\ s'} \textsc{DefScopeStep}$$

The proof is similar as case above.

$$\frac{\alpha \in \{::, ::a, ::\texttt{nd}\}}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ \alpha\ \texttt{skip} \xrightarrow{\beta}_{\mathtt{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}} \textsc{ScopeDone}$$

By definition, $labels(\texttt{scope}\ \alpha\ \texttt{skip}) = labels(\texttt{skip}) = \emptyset$ and $labels(\texttt{skip}) = \emptyset$. Since $\emptyset \cup labels(\hat{\sigma'}) \cup labels(\hat{\kappa'}) \cup labels(\hat{v'_{\mathrm{C}}}) \subseteq \emptyset \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, $labels(\texttt{skip}) \cup labels(\hat{\sigma'}) \cup labels(\hat{\kappa'}) \cup labels(\hat{v'_{\mathrm{C}}}) \subseteq labels(\texttt{scope}\ \alpha\ \texttt{skip}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. The invariant holds for the evaluation $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{scope}\ \alpha\ \texttt{skip} \xrightarrow{\beta}_{s} \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{skip}$. Then this induction rule holds w.r.t. the invariant.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (\alpha\ \texttt{def})\ \texttt{skip} \xrightarrow{\alpha}_{\mathtt{s}} clear(\widehat{\sigma}, \alpha\ \texttt{def}), \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}} \textsc{DefScopeDone}$$

By definition, $labels(\texttt{scope def skip}) = labels(\texttt{skip}) = \emptyset$, $labels(\texttt{skip}) = \emptyset$ and $labels(clear(\hat{\sigma}, \alpha\ \texttt{def})) = labels(\hat{\sigma}) \setminus labels(\alpha\ \texttt{def} x, v)$ for any x. Then $labels(clear(\hat{\sigma}, \alpha\ \texttt{def})) \subseteq labels(\hat{\sigma})$. Then we have $labels(clear(\hat{\sigma}, \alpha\ \texttt{def})) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \cup \emptyset \subseteq labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \cup \emptyset$, i.e. $labels(clear(\hat{\sigma}, \alpha\ \texttt{def})) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \cup labels(\texttt{skip}) \subseteq labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup \{\hat{v_{\mathrm{C}}} \cup labels(\texttt{scope def skip})$. Then the invariant holds for the evaluation $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{scope def skip} \xrightarrow{\alpha}_{s}$

$clear(\hat{\sigma}, \alpha\,\texttt{def}), \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{skip}$. This induction rule holds w.r.t. the invariant.

□

B.3.1 Proofs of the invariant in Manifest Evaluation

Proof. Prove by induction.

1 Top-Level Statements

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\cdot\cdot}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, , s'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, , s'} \textsc{TopScope}$$

By induction hypothesis, we have the invariant $labels(s') \cup labels(\hat{\sigma'}) \cup labels(\hat{\kappa'}) \cup labels(\hat{v'_{\mathrm{C}}}) \subseteq labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$ holds. We have $labels(s') \cup labels(\hat{\sigma'}) \cup labels(\hat{\kappa'}) \cup labels(\hat{v'_{\mathrm{C}}}) \subseteq labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. Then the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, s \xrightarrow{N}_{p} \hat{\sigma'}, \hat{\kappa'}, \hat{v'_{\mathrm{C}}}, s'$. Then this rule holds w.r.t. the invariant.

2 Sequential Composition

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, m_1 \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m'_1}{\widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m_1 \_ m_2 \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m'_1 \_ m_2} \textsc{MSeqStep}$$

By induction hypothesis, we have the invariant $labels(m'_1) \subseteq labels(m_1) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa})$ holds. By definition, $labels(m_1 m_2) = labels(m_1) \cup labels(m_2)$ and $labels(m'_1 m_2) = labels(m'_1) \cup labels(m_2)$. Since $labels(m'_1) \cup labels(\hat{\sigma'}) \cup labels(\hat{\kappa'}) \cup labels(\hat{v'_{\mathrm{C}}}) \subseteq labels(m_1) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, $labels(m'_1) \cup labels(m_2) \cup labels(\hat{\sigma'}) \cup labels(\hat{\kappa'}) \cup labels(\hat{v'_{\mathrm{C}}}) \subseteq labels(m_1) \cup labels(m_2) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, i.e. $labels(m'_1 m_2) \cup labels(\hat{\sigma'}) \cup labels(\hat{\kappa'}) \cup labels(\hat{v'_{\mathrm{C}}}) \subseteq labels(m_1 m_2) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. Then the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, m_1\ m_2 \xrightarrow{N}_{m} \hat{\sigma'}, \hat{\kappa'}, \hat{v'_{\mathrm{C}}}, m'_1\ m_2$. Then this rule holds w.r.t. the invariant.

$$\frac{}{\widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, \texttt{skip}\llcorner m \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m} \textsc{MSeqSkip}$$

By definition, $labels(\texttt{skip}\ m) = labels(\texttt{skip}) \cup labels(m) = labels(m)$. Since $labels(m) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(m) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, we have $labels(m) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\texttt{skip}\ m) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. That is the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{skip}\ m \xrightarrow{N}_{m} \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, m$. This rule holds w.r.t. the invariant.

3 Node Definitions

$$\frac{\mathsf{nodeMatch}(N, Q)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{node}\ Q\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (\texttt{::nd})\ s} \textsc{NodeMatch}$$

By definition, $labels(\texttt{node}\ Q\ \{s\}) = labels(s)$ and $labels(\texttt{scope}\ (\texttt{::}\ \texttt{nd})\ s) = labels(s)$. Since $labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, we have $labels(\texttt{node}\ m\ \{s\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\texttt{scope}\ (\texttt{::}\ \texttt{nd})\ s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. That is the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{node}\ m\ \{s\} \xrightarrow{N}_{p} \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{scope}\ (\texttt{::}\ \texttt{nd})\ s$. This rule holds w.r.t. the invariant.

$$\frac{\neg\mathsf{nodeMatch}(N, Q)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{node}\ Q\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}} \textsc{NodeNoMatch}$$

By definition, $labels(\texttt{node}\ Q\ \{s\}) = labels(s)$ and $labels(\texttt{skip}) = \emptyset$. Since $labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, we have $labels(\texttt{skip}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\texttt{node}\ Q\ \{s\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. That is the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}$, $\texttt{node}\ Q\ \{s\} \xrightarrow{N}_{\mathsf{m}} \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{scope}\ (\texttt{::}\ \texttt{nd})\ s$. This rule holds w.r.t. the invariant.

4 Defined resource types

$$\frac{u \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{define}\ u\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}[u : \mathsf{ResourceDef}(\rho, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{RDef}$$

5 Class Definitions

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ a\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \widehat{\kappa}[a : \mathsf{ClassDef}(\bot, \varepsilon, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{CDef}$$

By definition, $labels(\texttt{class}\ a\ \{s\}) = labels(s)$ and $labels(\texttt{skip}) = \emptyset$. Since $\hat{\kappa}(a) = \bot$, $labels(\bot) = \emptyset$. Then $labels(\hat{\kappa}(a \to \mathsf{ClassDef}(\bot, \varepsilon, s))) = labels(\hat{\kappa}) \cup labels(\mathsf{ClassDef}(\bot, \varepsilon, s)) = labels(\hat{\kappa}) \cup labels(s)$. Since $labels(\hat{\kappa}) \cup labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, we have $labels(\texttt{skip}) \cup labels(\hat{\kappa}(a \to \mathsf{ClassDef}(\bot, \varepsilon, s))) \cup labels(\hat{\sigma}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\texttt{class}\ a\ \{s\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. That is the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{class}\ a\ \{s\} \xrightarrow{N}_{p} \hat{\sigma}, \hat{\kappa}(a \to \mathsf{ClassDef}(\bot, \varepsilon, s)), v_{\mathrm{C}}, \texttt{skip}$. This rule holds w.r.t. the invariant.

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ a\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \widehat{\kappa}[a : \mathsf{ClassDef}(b, \varepsilon, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{CDefI}$$

By definition, we have $labels(\texttt{class}\ a\ \texttt{inherits}\ b\ \{s\}) = labels(s)$ and $labels(\texttt{skip}) = \emptyset$. Since $\hat{\kappa}(a) = \bot$, $labels(\bot) = \emptyset$. Then $labels(\hat{\kappa}(a \to \mathsf{ClassDef}(b, \varepsilon, s))) = labels(\hat{\kappa}) \cup labels(\mathsf{ClassDef}(b, \varepsilon, s))) = labels(\hat{\kappa}) \cup labels(s)$. Since $labels(\hat{\kappa}) \cup labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(s) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, we have $labels(\texttt{skip}) \cup labels(\hat{\kappa}(a \to \mathsf{ClassDef}(b, \varepsilon, s))) \cup labels(\hat{\sigma}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\texttt{class}\ a\ \texttt{inherits}\ b\ \{s\}) \cup labels(\hat{\sigma}) \cup labels(\kappa) \cup labels(\hat{v_{\mathrm{C}}})$. That is the invariant holds for the evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{class}\ a\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_{p} \hat{\sigma}, \hat{\kappa}(a \to \mathsf{ClassDef}(b, \varepsilon, s)), v_{\mathrm{C}}, \texttt{skip}$. This rule holds w.r.t. the invariant.

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ a\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \widehat{\kappa}[a : \mathsf{ClassDef}(\bot, \rho, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{CDefP}$$

By definition, $labels(\texttt{class}\ a\ (\rho)\ \{s\}) = labels(\rho) \cup labels(s)$ and $labels(\texttt{skip}) = \emptyset$. Since $\hat{\kappa}(a) = \bot$, $labels(\bot) = \emptyset$. Then $labels(\hat{\kappa}(a \to \mathsf{ClassDef}(\bot, \rho, s))) =$

$labels(\hat{\kappa}) \cup labels(\mathsf{ClassDef}(\bot, \rho, s)) = labels(\hat{\kappa}) \cup labels(s) \cup labels(\rho)$.
Since $labels(\hat{\kappa}) \cup labels(s) \cup labels(\rho) \cup labels(\hat{\sigma}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(s) \cup labels(\rho) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$, we have $labels(\texttt{skip}) \cup labels(\hat{\kappa}(a \to \mathsf{ClassDef}(\bot, \rho, s))) \cup labels(\hat{\sigma}) \cup labels(\hat{v_{\mathrm{C}}}) \subseteq labels(\texttt{class}\ a\ (\rho)\ \{s\}) \cup labels(\hat{\sigma}) \cup labels(\hat{\kappa}) \cup labels(\hat{v_{\mathrm{C}}})$. That is the invariant holds for the evaluation step
$\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{class}\ a\ (\rho)\ \{s\} \xrightarrow{N}_p \hat{\sigma}, \hat{\kappa}(a \to \mathsf{ClassDef}(\bot, \rho, s)), v_{\mathrm{C}}, \texttt{skip}$. This rule holds w.r.t. the invariant.

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ a\ (\rho)\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}[a : \mathsf{ClassDef}(b, \rho, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{CDefPI}$$

The proof is similar as case above.

□

Appendix C

Expression Provenance

C.1 Propagation of Annotations in the Evaluation

C.1.1 Expressions

C.1.1.1 Selector

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e\ ?\{M\} \xrightarrow{\alpha} e'\ ?\{M\}}\ \textsc{SCon}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta}\ ?\{e_1 \Rightarrow e, M\} \xrightarrow{\alpha} v^{\beta}\ ?\{e_1' \Rightarrow e, M\}}\ \textsc{SEle}$$

$$\frac{\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\} \xrightarrow{\alpha} e}\ \textsc{SChoose}$$

$$\frac{\neg\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\} \xrightarrow{\alpha} v^{\beta}\ ?\{M\}}\ \textsc{SChooseI}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta}\ ?\{\texttt{default} \Rightarrow e, M\} \xrightarrow{\alpha} e}\ \textsc{SDefault}$$

C.1.2 Resources

A resource is evaluated in the context of a variable environment. Evaluation of a resource may not change the environment.

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},x \Rightarrow e,H \xrightarrow{\alpha}_{\mathsf{H}} x \Rightarrow e',H}\ \textsc{RbodyExp}$$

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},x \Rightarrow v^{\beta},H \xrightarrow{\alpha}_{\mathsf{H}} x \Rightarrow v^{\beta},H'}\ \textsc{RbodyStep}$$

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e : H \xrightarrow{\alpha}_{\mathsf{R}} e' : H}\ \textsc{RName} \qquad \frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},v^{\beta} : H \xrightarrow{\alpha}_{\mathsf{R}} v^{\beta} : H'}\ \textsc{Rbody}$$

C.1.3 Statements

The form of the rules of evaluating annotated statements is the same as in where-provenance. We omit them here.

C.1.4 Manifests

The form of the rules of evaluating annotated manifests is the same as in where-provenance. We omit them here.

C.2 Proofs for Invariants in Expression Evaluation

Theorem C.2.1 (Invariant in Expression Evaluation)**.** For any $\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'$, if $\mu \models \widehat{\sigma}$, $\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models e$, then $\mu \models e'$.

Proof. Prove by induction.

1 Variables

$$\frac{x \in dom(\sigma_\alpha)}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\$x \xrightarrow{\alpha} \sigma_\alpha(x)}\ \textsc{LVar}$$

We know by definition $labels(\hat{\sigma}_\alpha(x)) \subseteq labels(\hat{\sigma})$ for some x. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \$x$. Then $\forall v^{\beta} \in labels(\hat{\sigma})\ \mu \models \beta \backsim v \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \$x \rightarrow \forall v'^{\beta'} \in labels(\hat{\sigma}_\alpha(x))\ \mu \models \beta' \backsim v'$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \$x \rightarrow \mu \models \hat{\sigma}_\alpha(x)$. Then if $\mu \models \$x$, $\mu \models \hat{\sigma}$,

$\mu \models \hat{\kappa}$ and $\mu \models \hat{v_{\mathrm{C}}}$ then $\mu \models \hat{\sigma}_\alpha(x)$.

$$\frac{x \notin dom(\sigma_\alpha)\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x \xrightarrow{\beta} v^\beta \beta\, \mathsf{parentof}_\kappa\, \alpha}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x \xrightarrow{\alpha} v^\beta} \textsc{PVar}$$

$$\frac{x \in dom(\sigma_{::})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$::x \xrightarrow{\alpha} \sigma_{::}(x)} \textsc{TVar}$$

The proof is similar as LVar.

$$\frac{x \in dom(\sigma_{::a})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$::a :: x \xrightarrow{\alpha} \sigma_{::a}(x)} \textsc{QVar}$$

The proof is similar as LVar.

2 Arithmetic expressions

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 + e_2 \xrightarrow{\alpha} e_1' + e_2} \textsc{ArithLeft}$$

The proof is similar as the rule ArithRight.

3 Comparison expressions

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 > e_2 \xrightarrow{\alpha} e_1' > e_2} \textsc{CompLeft}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^l > e \xrightarrow{\alpha} v^l > e'} \textsc{CompRight}$$

$$\frac{v_1 >_{\mathbb{Z}} v_2}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v_1^{\beta_1} > v_2^{\beta_2} \xrightarrow{\alpha} \mathtt{true}^\perp} \textsc{CompValueI}$$

We know $labels(v_1^{\beta_1} > v_2^{\beta_2}) = labels(v_1^{\beta_1}) \cup labels(v_1^{\beta_1}) = \{v_1^{\beta_1}, v_2^{\beta_2}\}$ and $labels(\mathtt{true}^{\beta_1 > \beta_2}) = \{\mathtt{true}^{\beta_1 > \beta_2}\}$ by definition. Suppose $\mu \models \hat{\sigma}$ and $\mu \models v_1^{\beta_1} > v_2^{\beta_2}$, i.e. $\forall v^\beta \in \{v_1^{\beta_1}, v_2^{\beta_2}\}\ \mu \models \beta \backsim v$. That is $\mu \models \beta_1 \backsim v_1$

and $\mu \models \beta_2 \backsim v_2$. Since we know $v_1 > v_2$, by inference rule we have

$$\frac{\mu \models \beta_1 \backsim v_1 \quad \mu \models \beta_2 \backsim v_2 \quad v_1 > v_2}{\mu \models \beta_1 > \beta_2 \backsim v_1 >_{\mathbb{N}} v_2}$$

. Then we have $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \kappa \wedge \mu \models \beta_1 \backsim v_1 \wedge \mu \models \beta_2 \backsim v_2 \rightarrow \mu \models \beta_1 > \beta_2 \backsim v_1 >_{\mathbb{N}} v_2$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \{v_1^{\beta_1}, v_2^{\beta_2}\}\ \mu \models a \backsim v \rightarrow \forall v^{\beta} \in \mathit{labels}(\mathtt{true}^{\beta_1 > \beta_2})\ \mu \models \beta_1 > \beta_2 \backsim \mathtt{true}$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta'} \in \mathit{labels}(v_1^{\beta_1} > v_2^{\beta_2})\ \mu \models \beta \backsim v \rightarrow \forall v^{\beta} \in \mathit{labels}(\mathtt{true}^{\beta_1 > \beta_2})\ \mu \models \beta_1 > \beta_2 \backsim \mathtt{true}$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models v_1^{B_1} > v_2^{\beta_2} \rightarrow \mu \models \mathtt{true}^{\beta_1 > \beta_2}$. Then if $\mu \models \hat{\sigma}$ and $\mu \models v_1^{\beta_1} > v_2^{\beta_2}$ then $\mu \models \mathtt{true}^{\beta_1 > \beta_2}$.

$$\frac{v_1 <=_{\mathbb{Z}} v_2}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v_1^{\beta_1} > v_2^{\beta_2} \xrightarrow{\alpha} \mathtt{false}^{\bot}}\ \textsc{CompValueII}$$

4 Boolean expressions

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \,\mathtt{and}\, e_2 \xrightarrow{\alpha} e_1' \,\mathtt{and}\, e_2}\ \textsc{AndLeft}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{false}^{\beta} \,\mathtt{and}\, e \xrightarrow{\alpha} \mathtt{false}^{\bot}}\ \textsc{AndRightI}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{true}^{\beta} \,\mathtt{and}\, e \xrightarrow{\alpha} \mathtt{true}^{l} \,\mathtt{and}\, e'}\ \textsc{AndRightII}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{true}^{\beta_1} \,\mathtt{and}\, \mathtt{true}^{\beta_2} \xrightarrow{\alpha} \mathtt{true}^{\bot}}\ \textsc{AndValue}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, !\, e \xrightarrow{\alpha} !\, e'}\ \textsc{NotSTEP}$$

By induction hypothesis, we have if $\mu \models e$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \hat{\sigma}$ then $\mu \models e'$. We have $\mathit{labels}(!e) = \mathit{labels}(e)$ and $\mathit{labels}(!e') = \mathit{labels}(e')$ by definition. Since if $\mu \models e \wedge \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}}$ then $\mu \models e'$, we have $\forall v^{\beta} \in \mathit{labels}(e)\ \mu \models \beta \backsim v \wedge \mu \models \hat{\sigma} \rightarrow \forall v'^{\beta'} \in \mathit{labels}(e')\ \mu \models \beta' \backsim v'$.

Then $\forall v^{\beta} \in labels(!e)\ \mu \models \beta \backsim v \wedge \mu \models \hat{\sigma} \rightarrow \forall v'^{\beta'} \in labels(!e')\ \mu \models \beta' \backsim v'$, i.e. $\mu \models !e \wedge \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \rightarrow \mu \models !e'$. That is if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models !e$ then $\mu \models !e'$.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, !\mathtt{true}^{l} \xrightarrow{\alpha} \mathtt{false}^{\bot}}\ \textsc{NotValueI}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, !\mathtt{false}^{l} \xrightarrow{\alpha} \mathtt{true}^{\bot}}\ \textsc{NotValueII}$$

5 Selectors

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e\ ?\{M\} \xrightarrow{\alpha} e'\ ?\{M\}}\ \textsc{SCon}$$

The proof is similar as SEle.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta}\ ?\{e_1 \Rightarrow e, M\} \xrightarrow{\alpha} v^{\beta}\ ?\{e_1' \Rightarrow e, M\}}\ \textsc{SEle}$$

By induction hypothesis, we have if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models e_1$ then $\mu \models e_1'$. That is $\mu \models \hat{\sigma} \wedge \forall v^{\beta} \in labels(e)\ \mu \models \beta \backsim v \rightarrow \forall v'^{\beta'} \in labels(e')\ \mu \models \beta' \backsim v'$. We know $labels(v^{\beta}\ ?\ \{e_1 \Rightarrow e, M\}) = labels(v^{\beta}) \cup labels(e_1) \cup labels(e) \cup labels(M)$ by definition. Similarly, $labels(v^{\beta}\ ?\ \{e_1' \Rightarrow e, M\}) = labels(v^{\beta}) \cup labels(e_1') \cup labels(e) \cup labels(M)$. Suppose $\mu \models v^{\beta}\ ?\{e_1 \Rightarrow e, M\}$. That is $\forall v'^{\beta'} \in labels(v^{\beta}\ ?\{e_1 \Rightarrow e, M\})\ \mu \models \beta' \backsim v'$, i.e. $\forall v'^{\beta'} \in labels(v^{\beta}) \cup labels(e_1) \cup labels(e) \cup labels(M)\ \mu \models \beta' \backsim v'$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(v^{\beta}) \cup labels(e_1) \cup labels(e) \cup labels(M)\ \mu \models \beta \backsim v \rightarrow \forall v^{\beta} \in labels(v^{\beta}) \cup labels(e_1) \cup labels(e') \cup labels(M)\ \mu \models \beta \backsim v$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(v^{\beta}\ ?\ \{e_1 \Rightarrow e, M\})\ \mu \models \beta \backsim v \rightarrow \forall v^{\beta} \in labels(v^{\beta}\ ?\ \{e_1' \Rightarrow e, M\})\ \mu \models \beta \backsim v$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models v^{\beta}\ ?\ \{e_1 \Rightarrow e, M\} \rightarrow \mu \models v^{\beta}\ ?\ \{e_1' \Rightarrow e, M\}$.

$$\frac{\neg \mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta}\ ?\{v_1^{\beta_1} \Rightarrow e, M\} \xrightarrow{\alpha} v^{\beta}\ ?\{M\}}\ \textsc{SChooseI}$$

The proof is similar as the rule SChoose.

$$\frac{}{\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, v^{\beta}\ ?\{\texttt{default} \Rightarrow e, M\} \xrightarrow{\alpha} e}\ \textsc{SDefault}$$

Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models v^{\beta}\ ?\{\texttt{default} \Rightarrow e, M\}$. That is $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\forall v_1^{\beta_1} \in \mathit{labels}(v^{\beta}\ ?\{\texttt{default} \Rightarrow e, M\})\ \mu \models \beta_1 \backsim v_1$. We know $\mathit{labels}(v^{\beta}\ ?\{\texttt{default} \Rightarrow e, M\}) = \mathit{labels}(v^{\beta}) \cup \mathit{labels}(e) \cup \mathit{labels}(M)$ by definition. Then we have if $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v_1^{\beta_1} \in \mathit{labels}(v^{\beta}) \cup \mathit{labels}(e) \cup \mathit{labels}(M)\ \mu \models \beta_1 \backsim v_1$ then $\forall v_2^{\beta_2} \in \mathit{labels}(e)\ \mu \models \beta_2 \backsim v_2$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v_1^{\beta_1} \in \mathit{labels}(v^{\beta}\ ?\{\texttt{default} \Rightarrow e, M\}) \rightarrow \forall v_2^{\beta_2} \in \mathit{labels}(e)\ \mu \models \beta_2 \backsim v_2$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge v^{\beta}\ ?\{\texttt{default} \Rightarrow e, M\} \rightarrow \mu \models e$. We have if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models v^{\beta}\ ?\{\texttt{default} \Rightarrow e, M\}$ then $\mu \models e$.

□

C.2.1 Proofs for Invariants in Resource Evaluation

Proof. Prove by induction.

Case 1 $$\frac{\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, R \xrightarrow{\alpha} R'}{\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, R \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, R'}$$

By induction hypothesis, we know if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$ and $\mu \models \hat{v_{\mathrm{C}}}$ then $\mu \models R'$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \rightarrow \mu \models R'$, i.e. $\mu \models \hat{\sigma} \wedge \forall v^{\beta} \in \mathit{labels}(R)\ \mu \models \beta \backsim v \rightarrow \forall v'^{\beta'} \in \mathit{labels}(R')\ \mu \models \beta' \backsim v'$. Suppose $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models R$, i.e. $\forall v^{\beta} \in \mathit{labels}(R)\ \mu \models \beta \backsim v$. Then we have if $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(R)\ \mu \models \beta \backsim v \wedge \mu \models \kappa$ then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(R)\ \mu \models \beta \backsim v \wedge \mu \models \kappa \rightarrow \forall v'^{\beta'} \in \mathit{labels}(R')\ \mu \models \beta' \backsim v' \rightarrow \forall v'^{\beta'} \in \mathit{labels}(R')\ \mu \models \beta' \backsim v'$. Then we have if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models R$ then $\mu \models R'$.

Case 2 $$\frac{}{\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \hat{v_{\mathrm{R}}} \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{R}}}\ \hat{v_{\mathrm{C}}}, \texttt{skip}}$$

We have $\mathit{labels}(\texttt{skip}) = \emptyset$ by definition. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \hat{v_{\mathrm{R}}}$. We know $\forall v^{\beta} \in \emptyset\ \mu \models \beta \backsim v$ is true. Then we have $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \hat{v_{\mathrm{R}}} \rightarrow \forall v^{\beta} \in \emptyset\ \mu \models \beta \backsim v$, i.e. $\mu \models \hat{\sigma} \wedge$

$\mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \hat{v_{\mathrm{R}}} \rightarrow \forall v^{\beta} \in \mathit{labels}(\mathtt{skip})\ \mu \models \beta \backsim v$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \hat{v_{\mathrm{R}}} \rightarrow \forall v^{\beta} \in \mathit{labels}(\mathtt{skip})\ \mu \models \beta \backsim v$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \hat{v_{\mathrm{R}}} \rightarrow \mu \models \mathtt{skip}$. Then if $\mu \models \hat{\sigma}$, $\wedge \mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \hat{v_{\mathrm{R}}}$ then $\mu \models \mathtt{skip}$

□

C.3 Proof of Invariant in Statement Evaluation

Theorem C.3.1 (Invariant in Statement Evaluation)**.** For any $\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha}_{s} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}} \vdash s'$, if $\mu \models \widehat{\sigma}$, $\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models s$ then $\mu \models \widehat{\sigma'}$, $\mu \models \widehat{\kappa'}$, $\mu \models \widehat{v'_{\mathrm{C}}}$ and $\mu \models s'$.

Proof. Prove by induction.

1 Expression statements

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e'}\ \textsc{ExprStep}$$

By induction hypothesis, we know if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models e$ then $\mu \models e'$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(e)\ \mu \models \beta \backsim v \rightarrow \forall v'^{\beta'} \in \mathit{labels}(e')\ \mu \models \beta' \backsim v'$. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models e$, i.e. $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$ and $\forall v^{\beta} \in \mathit{labels}(e)\ \mu \models \beta \backsim v$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(e)\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(e)\ \mu \models \beta \backsim v \rightarrow \forall v'^{\beta'} \in \mathit{labels}(e')\ \mu \models \beta' \backsim v' \rightarrow \forall v'^{\beta'} \in \mathit{labels}(e')\ \mu \models \beta' \backsim v'$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models e \rightarrow \mu \models e$. Since $\hat{\sigma}$, $\hat{\kappa}$ and $\hat{v_{\mathrm{C}}}$ have not changed before and after the evaluation, we have $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models e \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models e$. That is if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models e$ then $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models e'$.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}}\ \textsc{Expr}$$

2 Sequential composition

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, s_1 \xrightarrow{\alpha}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'_1}{\sigma, \kappa, v_{\mathrm{C}}, s_1 \_ s_2 \xrightarrow{\alpha}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'_1 \_ s_2}\ \textsc{SeqStep}$$

By induction hypothesis, we know if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models s_1$ then $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models s'_1$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(s_1)\ \mu \models \beta \rightsquigarrow v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in \mathit{labels}(s'_1)\ \mu \models \beta' \rightsquigarrow v'$. By definition, we have $\mathit{labels}(s_1 s_2) = \mathit{labels}(s_1) \cup \mathit{labels}(s_2)$ and similarly $\mathit{labels}(s'_1 s_2) = \mathit{labels}(s'_1) \cup \mathit{labels}(s_2)$. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models s_1 s_2$, i.e. $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\forall v^{\beta} \in \mathit{labels}(s_1 s_2)\ \mu \models \beta \rightsquigarrow v$.

Then we have $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(s_1 s_2)\ \mu \models \beta \rightsquigarrow v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in (\mathit{labels}(s_1) \cup \mathit{labels}(s_2))\ \mu \models \beta \rightsquigarrow v \rightarrow \mu \models \hat{\sigma}' \wedge \mu \models \hat{\kappa}' \wedge \mu \models \hat{v_{\mathrm{C}}}' \wedge \forall v'^{\beta'} \in (\mathit{labels}(s'_1) \cup \mathit{labels}(s_2))\ \mu \models \beta' \rightsquigarrow v' \rightarrow \mu \models \hat{\sigma} \wedge' \mu \models \hat{\kappa}' \wedge \mu \models \hat{v_{\mathrm{C}}}' \wedge \forall v'^{\beta'} \in \mathit{labels}(s'_1 s_2)\ \mu \models \beta' \rightsquigarrow v'$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models s_1 s_2 \rightarrow \mu \models s'_1 s_2$. That is if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models s_1 s_2$ then $\mu \models \hat{\sigma'}$, $\mu \models \hat{\kappa'}$, $\mu \models \hat{v'_{\mathrm{C}}}$ and $\mu \models s'_1 s_2$.

$$\frac{}{\sigma, \kappa, v_{\mathrm{C}}, \mathtt{skip} \_ s \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, s}\ \textsc{SeqSkip}$$

By definition, we have $\mathit{labels}(\mathtt{skip}\ s) = \mathit{labels}(\mathtt{skip}) \cup \mathit{labels}(s) = \emptyset \cup \mathit{labels}(s) = \mathit{labels}(s)$. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \mathtt{skip}\ s$, i.e. $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\forall v^{\beta} \in \mathit{labels}(\mathtt{skip}\ s)\ \mu \models \beta \rightsquigarrow v$.

Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(\mathtt{skip}\ s)\ \mu \models \beta \rightsquigarrow v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(s)\ \mu \models \beta \rightsquigarrow v$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \mathtt{skip}\ s \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models s$.

That is if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \mathtt{skip}\ s$ then $\mu \models \hat{\sigma'}$, $\mu \models \hat{\kappa'}$, $\mu \models \hat{v'_{\mathrm{C}}}$ and $\mu \models s$. The invariant holds for this evaluation step $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \mathtt{skip}\ s \xrightarrow{\alpha}_{s} \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, s$.

3 Assignment

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x = e \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x = e'}\ \textsc{AssignStep}$$

By induction hypothesis, we know if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models e$

then $\mu \models e'$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(e)\ \mu \models \beta \backsim v \rightarrow \forall v'^{\beta'} \in labels(e')\ \mu \models \beta' \backsim v'$. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \$x = e$. By definition, we have $labels(\$x = e) = labels(e) = labels(e)$ and $labels(\$x = e') = labels(e') = labels(e')$. Then we have $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(\$x = e)\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(e)\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(e')\ \mu \models \beta' \backsim v' \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(\$x = e')\ \mu \models \beta' \backsim v'$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \$x = e \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \$x = e'$.

$$\frac{x \notin dom(\sigma_{\alpha})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x = v^{\beta} \xrightarrow{\alpha}_{\mathtt{s}} \widehat{\sigma}[(\alpha, x) : v^{\beta}], \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}}\ \textsc{Assign}$$

By definition, we have $labels(\$x = v^{\beta}) = labels(\$x) \cup labels(v^{\beta}) = \emptyset \cup \{v^{\beta}\} = \{v^{\beta}\}$, $labels(\mathtt{skip}) = \emptyset$ and $lva\hat{\sigma}[(\alpha, x) : v^{\beta}] = labels(\hat{\sigma}) \cup \{v^{\beta}\}$. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \$x = v^{\beta}$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(\$x = v^{\beta})\ \mu \models \beta \backsim v \rightarrow \forall v'^{\beta'} \in labels(\hat{\sigma})\ \mu \models \beta' \backsim v' \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(v^{\beta})\ \mu \models \beta \backsim v \rightarrow \forall v^{\beta} \in (labels(\hat{\sigma} \cup labels(v^{\beta})))\ \mu \models \beta' \backsim v' \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mathtt{true}$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \$x = v^{\beta} \rightarrow \mu \models \hat{\sigma}[(\alpha, x) : v^{\beta}] \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \mathtt{skip}$.

4 If

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathtt{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{if}\ e'\ \{s_1\}\ \mathtt{else}\ \{s_2\}}\ \textsc{IfStep}$$

By induction hypothesis, we know if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models e$ then $\mu \models e'$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(e)\ \mu \models \beta \backsim v \rightarrow \forall v'^{\beta'} \in labels(e')\ \mu \models \beta' \backsim v'$.

By definition, we have $labels(\mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\}) = labels(e) \cup labels(s_1) \cup labels(s_2)$ and $labels(\mathtt{if}\ e'\ \{s_1\}\ \mathtt{else}\ \{s_2\}) = labels(e') \cup labels(s_1) \cup labels(s_2)$.

Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\}$, i.e. $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\forall v'^{\beta'} \in labels(\mu \models \mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\})\ \mu \models \beta' \backsim v'$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(\mathtt{if}\ e'\ \{s_1\}\ \mathtt{else}\ \{s_2\})\ \mu \models$

$\beta' \backsim v' \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(e) \cup labels(s_1) \cup labels(s_2)\ \mu \models$
$\beta' \backsim v' \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(e') \cup labels(s_1) \cup$
$labels(s_2)\ \mu \models \beta' \backsim v'$. i.e. if $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{if}\ e\ \{s_1\}\ \texttt{else}\ \{s_2\}$
then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{if}\ e'\ \{s_1\}\ \texttt{else}\ \{s_2\}$.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s_1}\ \textsc{IfT}$$

By definition, we have $labels(\texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\}) = labels(\texttt{true}^{\beta}) \cup$
$labels(s_1) \cup labels(s_2)$. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\}$.
Then that is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(\texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\}) \mu \models$
$\beta \backsim v \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \{\texttt{true}^{\beta}\} \cup labels(s_1) \cup labels(s_2)\ \mu \models$
$\beta \backsim v \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(s_1)\ \mu \models \beta \backsim v$. That is
$\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{if true}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\} \to \mu \models \hat{\sigma} \wedge \mu \models$
$\hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models s_1$.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{if false}^{\beta}\ \{s_1\}\ \texttt{else}\ \{s_2\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s_2}\ \textsc{IfF}$$

Similar as above.

5 Unless

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{unless}\ e\ \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{unless}\ e'\ \{s\}}\ \textsc{UnlessStep}$$

The proof is similar as IfStep.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{unless true}^{\beta}\ \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}},\ \texttt{skip}}\ \textsc{UnlessT}$$

The proof is similar as IfT.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{unless false}^{\beta}\ \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}},\ s}\ \textsc{UnlessF}$$

The proof is similar as IfT.

6 Case

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{case}\ e\ \{C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{case}\ e'\ \{C\}}\ \textsc{CaseStep1}$$

Similar as IfStep.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{case}\ v^{\beta}\ \{e : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{case}\ v^{\beta}\ \{e' : \{s\} \llcorner C\}}\ \textsc{CaseStep2}$$

$$\frac{\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s}\ \textsc{CaseMatch}$$

By definition, we have $labels(\mathtt{case}\ v^{\beta}\ \{v_1^{\beta'} : \{s\}\ C\}) = labels(v^{\beta}) \cup labels(v_1^{\beta'}) \cup labels(s) \cup labels(C)$. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \mathtt{case}\ v^{\beta}\ \{v_1^{\beta'} : \{s\}\ C\}$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(\mathtt{case}\ v^{\beta}\ \{v_1^{\beta'} : \{s\}\ C\})\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(v^{\beta}) \cup labels(v_1^{\beta_1}) \cup labels(s) \cup labels(C)\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(s)\ \mu \models \beta \backsim v$ i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \mathtt{case}\ v^{\beta}\ \{v_1^{\beta_1} : \{s\}\ C\} \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models s$.

$$\frac{\neg\mathsf{caseMatch}(v, v_1)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{case}\ v^{\beta}\ \{v_1^{l_1} : \{s\} \llcorner C\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{case}\ v^{\beta}\ \{C\}}\ \textsc{CaseNoMatch}$$

The proof is similar as case above.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{case}\ v^{\beta}\ \{\varepsilon\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}}\ \textsc{CaseDone}$$

We have $labels(\mathtt{case}\ v^{\beta}\ \{\varepsilon\}) = labels(v^{\beta})$ and $labels(\mathtt{skip}) = \emptyset$ and $labels(\mathtt{skip}) = \emptyset$ by definition. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \mathtt{case}\ v^{\beta}\ \{\}$. Since $\forall v^{\beta} \in labels(\mathtt{skip})\ \mu \models \beta \backsim v$, we have $\forall v^{\beta} \in \emptyset\ \mu \models \beta \backsim v = \mathtt{true}$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(v^{\beta})\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(\mathtt{skip})\ \mu \models \beta' \backsim v'$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \mathtt{case}\ v^{\beta}\ \{\varepsilon\} \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \mathtt{skip}$.

7 Resource declarations

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e : H \xrightarrow{\alpha}_{\mathsf{R}} e' : H'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, t\ \{e : H\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, t\ \{e' : H'\}}\ \textsc{ResStep}$$

Omitted.

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\widehat{v_{\mathrm{R}}} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}}\llcorner\widehat{v_{\mathrm{R}}},\mathtt{skip}}\ \textsc{ResDecl}$$

Omitted.

8 Defined resource types

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\{e:H\} \xrightarrow{\alpha}_{\mathsf{R}} \{e':H'\}}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},u\ \{e:H\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},u\ \{e':H'\}}\ \textsc{DefStep}$$

Omitted

$$\frac{\widehat{\kappa}(u) = \mathsf{ResourceDef}(\rho,s) s' = \mathit{merge}(\rho, v_{\mathrm{H}})}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},u\ \{w:v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{scope}\ (\alpha\ \mathtt{def})\ \{\$\mathit{title} = w\llcorner s'\llcorner s\}}\ \textsc{Def}$$

We have $\mathit{labels}(u\ \{w:v_{\mathrm{H}}\}) = \mathit{labels}(w) \cup \mathit{labels}(v_{\mathrm{H}})$ and $\mathit{labels}(\mathtt{scope}\ (\alpha\ \mathtt{def})\ \{\$\mathit{title} = w\llcorner s'\llcorner s) = \mathit{labels}(w) \cup \mathit{labels}(s) \cup \mathit{labels}(s')$ by definition. We know also $\mathit{labels}(\mathsf{ResourceDef}(\rho,s)) = \mathit{labels}(\rho) \cup \mathit{labels}(s) \subseteq \mathit{labels}(\kappa)$ by definition. By the definition of function merge, we know $\mathit{labels}(s) \subseteq \mathit{labels}(\rho) \cup \mathit{labels}(v_{\mathrm{H}})$. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models u\ \{w:v_{\mathrm{H}}\}$.

Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(u\ \{w:v_{\mathrm{H}}\})\ \mu \models \beta \backsim v \to \mu \models \hat{\sigma} \wedge \forall v^{\beta} \in \mathit{labels}(\hat{\kappa})\ \mu \models \beta \backsim v \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in \mathit{labels}(v_{\mathrm{H}}) \cup \{w\}\ \mu \models \beta' \backsim v' \to \mu \models \hat{\sigma} \wedge \forall v^{\beta} \in \mathit{labels}(\kappa) \setminus \mathit{labels}(\mathsf{ResourceDef}(\rho,s)) \cup \mathit{labels}(\mathsf{ResourceDef}(\rho,s))\ \mu \models \beta \backsim v \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in \mathit{labels}(v_{\mathrm{H}})\ \mu \models \beta' \backsim v' \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in \mathit{labels}(s) \cup \mathit{labels}(s)\ \mu \models \beta' \backsim v' \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in \mathit{labels}(\mathtt{scope}\ (\alpha\ \mathtt{def})\ \{\$\mathit{title} = w\llcorner s'\llcorner s\})\ \mu \models \beta' \backsim v'$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models u\ \{w : v_{\mathrm{H}}\} \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \mathtt{scope}\ (\alpha\ \mathtt{def})\ \{\$\mathit{title} = w\llcorner s'\llcorner s\}$.

9 Include

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(\bot,\rho,s) s' = \mathit{merge}(\rho,\varepsilon)\beta\, \mathsf{baseof}_{\widehat{\kappa}}\, \alpha}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{include}\ a \xrightarrow{\alpha}{}'_{\mathsf{s}} \widehat{\sigma},\kappa[a:\mathsf{DeclaredClass}(\beta)],\widehat{v_{\mathrm{C}}},\mathtt{scope}\ (::a)\ s'\llcorner s}\ \textsc{IncU}$$

By definition, we have $\mathit{labels}(\mathtt{include}\ a) = \emptyset$ and $\mathit{labels}(\mathtt{scope}\ (::a)\ s'\llcorner s) = \mathit{labels}(s) \cup \mathit{labels}(s')$. We also know that $\mathit{labels}(\kappa(a)) =$

$labels(\mathsf{ClassDef}(\bot, \rho, s)) = labels(\rho) \cup labels(s)$,
$labels(\mathsf{DeclaredClass}(\bot)) = \emptyset$ and $labels(\hat{\kappa}(a \to \mathsf{DeclaredClass}(\bot))) = labels(\hat{\kappa}) \setminus labels(\mathsf{ClassDef}(\bot, \rho, s))$ by definition.
Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \texttt{include}\ a$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(\texttt{include}\ a)\ \mu \models \beta \backsim v \to \mu \models \hat{\sigma} \wedge \forall v^{\beta} \in labels(\hat{\kappa})\ \mu \models \beta \backsim v \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(\texttt{include}\ a)\ \mu \models \beta' \backsim v' \to \mu \models \hat{\sigma} \wedge \forall v^{\beta} \in labels(\hat{\kappa}(a \to \mathsf{DeclaredClass}(\bot))) \cup labels(\mathsf{ClassDef}(\bot, \rho, s)) \mu \models \beta \backsim v \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in \emptyset\ \mu \models \beta' \backsim v' \to \mu \models \hat{\sigma} \wedge \forall v^{\beta} \in labels(\hat{\kappa}(a \to \mathsf{DeclaredClass}(\bot))) \cup labels(s) \cup labels(s') \mu \models \beta \backsim v \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \texttt{true} \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa}(a \to \mathsf{DeclaredClass}(\bot)) \wedge \mu \models \texttt{scope}\ (::a)\ s' \llcorner s \wedge \mu \models \hat{v_{\mathrm{C}}}$.
That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{include}\ a \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa}(a \to \mathsf{DeclaredClass}(\bot)) \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{scope}\ (::a)\ s' \llcorner s$.

$$\frac{\hat{\kappa}(a) = \mathsf{DeclaredClass}(\beta)}{\hat{\sigma}, \hat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \hat{\sigma}, \hat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{IncD}$$

By definition, we have $labels(\texttt{include}\ a) = \emptyset$ and $labels(\texttt{skip}) = \emptyset$. Since $\forall v^{\beta} \in labels(\texttt{include}\ a)\ \mu \models \beta \backsim v$, we have $\forall v^{\beta} \in \emptyset\ \mu \models \beta \backsim v = \texttt{true}$. Similarly, $\forall v^{\beta} \in labels(\texttt{skip})\ \mu \models \beta \backsim v = \texttt{true}$. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \texttt{include}\ a$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(\texttt{include}\ a)\ \mu \models \beta \backsim v \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \texttt{true} \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(\texttt{skip})\ \mu \models \beta' \backsim v'$.
That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{include}\ a \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{skip}$.

$$\frac{\hat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \hat{\kappa}(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\hat{\sigma}, \hat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \hat{\sigma}, \hat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ b\ \texttt{include}\ a}\ \textsc{IncPU}$$

The proof is similar as case IncU.

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \kappa(b) = \mathsf{DeclaredClass}(\beta) s' = merge(\rho, \varepsilon)}{\hat{\sigma}, \hat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa[a : \mathsf{DeclaredClass}(::b)], \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ \{s' \llcorner s\}}\ \textsc{IncPD}$$

The proof is similar as IncU.

10 Resource-like class declarations

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(c_{opt}, \rho, S)\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{class}\ \{a : H\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{class}\ \{a : H'\}}\ \textsc{CDecStep}$$

Omitted.

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(\bot, \rho, s)s' = merge(\rho, v_{\mathrm{H}})\beta\, \mathsf{baseof}_{\widehat{\kappa}}\, \alpha}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \kappa[a : \mathsf{DeclaredClass}(\beta)], \widehat{v_{\mathrm{C}}}, \mathtt{scope}\ (::a)\ s' \llcorner s}\ \textsc{CDecU}$$

By definition, we have $labels(\mathtt{class}\ \{a :\ v_{\mathrm{H}}\}) = labels(v_{\mathrm{H}})$, $labels(\mathsf{ClassDef}(\bot, \rho, s)) = labels(\rho) \cup labels(s)$ and $labels(\mathtt{scope}(::a)\ s' \llcorner s) = labels(s') \cup labels(s)$. We also know $labels(\mathsf{ClassDef}(\bot, \rho, s)) = labels(\rho) \cup labels(s) \subseteq labels(\kappa)$ and $labels(\kappa(a \to \mathsf{DeclaredClass}(\bot))) = labels(\hat{\kappa}) \setminus labels(\mathsf{ClassDef}(\bot, \rho, s))$ by definition. By the definition of function $merge$, we know $labels(s') \subseteq labels(\rho) \cup labels(v_{\mathrm{H}})$.

Suppose $\mu \models \hat{\sigma},\ \mu \models \hat{\kappa},\ \mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \mathtt{class}\ \{a :\ v_{\mathrm{H}}\}$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(\mathtt{class}\ \{a :\ v_{\mathrm{H}}\})\ \mu \models \beta \backsim v \to \mu \models \hat{\sigma} \wedge \forall v^{\beta} \in labels(\hat{\kappa})\ \mu \models \beta \backsim v \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(v_{\mathrm{H}})\ \mu \models \beta' \backsim v' \to \mu \models \hat{\sigma} \wedge \forall v^{\beta} \in labels(\hat{\kappa}(a \to \mathsf{DeclaredClass}(\bot))) \cup labels(\mathsf{ClassDef}(\bot, \rho, s))\mu \models \beta \backsim v \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(v_{\mathrm{H}})\ \mu \models \beta' \backsim v' \to \mu \models \hat{\sigma} \wedge \forall v^{\beta} \in labels(\hat{\kappa}(a \to \mathsf{DeclaredClass}(\bot))) \cup labels(\rho) \cup labels(s)\mu \models \beta \backsim v \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(v_{\mathrm{H}})\ \mu \models \beta' \backsim v' \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa}(a \to \mathsf{DeclaredClass}(\bot)) \wedge \forall v'^{\beta'} \in labels(s') \cup labels(s) \wedge \mu \models \hat{v_{\mathrm{C}}} \to \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa}(a \to \mathsf{DeclaredClass}(\bot)) \wedge \forall v'^{\beta'} \in labels(\mathtt{scope}\ (::a)\ s' \llcorner s) \wedge \mu \models \hat{v_{\mathrm{C}}}$. That is, if $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \mathtt{class}\ \{a : v_{\mathrm{H}}\}$ then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa}(a \to \mathsf{DeclaredClass}(\bot)) \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \mathtt{scope}\ (::a)\ s' \llcorner s$.

$$\frac{\widehat{\kappa(a)} = \mathsf{ClassDef}(b, \rho, s)\widehat{\kappa(b)} = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{include}\ b\ \mathtt{class}\ \{a : v_{\mathrm{H}}\}}\ \textsc{CDecPU}$$

We have $labels(\mathtt{class}\ \{a :\ v_{\mathrm{H}}\}) = labels(v_{\mathrm{H}})$ and $labels(\mathtt{include}\ b\ \mathtt{class}\ \{a :\ v_{\mathrm{H}}\}) = \emptyset \cup labels(v_{\mathrm{H}}) = labels(v_{\mathrm{H}})$ by definition. Suppose $\mu \models \hat{\sigma},\ \mu \models \hat{\kappa},\ \mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \mathtt{class}\ \{a :\ v_{\mathrm{H}}\}$. Then $\mu \models$

$\hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(\texttt{class}\ \{a :\ v_{\mathrm{H}}\})\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(v_{\mathrm{H}})\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(\texttt{include}\ b\ \texttt{class}\ \{a :\ v_{\mathrm{H}}\})\ \mu \models \beta \backsim v \rightarrow \forall v^{\beta} \in \mathit{labels}(\texttt{class}\ \{a :\ v_{\mathrm{H}}\})\ \mu \models \beta \backsim v \rightarrow$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{class}\ \{a : v_{\mathrm{H}}\} \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa}(a \rightarrow \mathsf{DeclaredClass}(\bot)) \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{include}\ b\ \texttt{class}\ \{a : v_{\mathrm{H}}\}$.

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \kappa(b) = \mathsf{DeclaredClass}(\beta) s' = \mathit{merge}(\rho, v_{\mathrm{H}})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \kappa[a : \mathsf{DeclaredClass}(::b)], \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ \{s' \llcorner s\}} \textsc{CDecPI}$$

The proof is similar as CDecU.

11 Scope

$$\frac{\alpha \in \{::, ::a, ::\mathtt{nd}\} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha}_{\mathrm{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ \alpha\ s \xrightarrow{\alpha'}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, \texttt{scope}\ \alpha\ s'} \textsc{ScopeStep}$$

By induction hypothesis, we know if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models s$ then $\mu \models \hat{\sigma'}$, $\mu \models \hat{\kappa'}$, $\mu \models \hat{v_{\mathrm{C}}}'$ and $\mu \models s'$. By definition we have $\mathit{labels}(\texttt{scope}\ \alpha\ s) = \mathit{labels}(s)$ and similarly $\mathit{labels}(\texttt{scope}\ \alpha\ s') = \mathit{labels}(s')$. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \texttt{scope}\ \alpha\ s$. Since $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(\texttt{scope}\ \alpha\ s)\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mathit{labels}(s) \rightarrow \mu \models \hat{\sigma'} \wedge \mu \models \hat{\kappa'} \wedge \mu \models \hat{v_{\mathrm{C}}}' \wedge \forall v'^{\beta'} \in \mathit{labels}(s')\ \mu \models \beta' \backsim v' \rightarrow \mu \models \hat{\sigma'} \wedge \mu \models \hat{\kappa'} \wedge \mu \models \hat{v_{\mathrm{C}}}' \wedge \forall v'^{\beta'} \in \mathit{labels}(\texttt{scope}\ \alpha\ s')\ \mu \models \beta' \backsim v'$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{scope}\ \alpha\ s \rightarrow \mu \models \hat{\sigma'} \wedge \mu \models \hat{\kappa'} \wedge \mu \models \hat{v_{\mathrm{C}}}' \wedge \mu \models \texttt{scope}\ \alpha\ s'$.

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha\ \mathtt{def}}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (\alpha\ \mathtt{def})\ s \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma'}, \texttt{scope}'\ (\alpha\ \mathtt{def})\ s'} \textsc{DefScopeStep}$$

The proof is similar as the rule ScopeStep.

$$\frac{\alpha \in \{::, ::a, ::\mathtt{nd}\}}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ \alpha\ \texttt{skip} \xrightarrow{\beta}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}} \textsc{ScopeDone}$$

We have $labels(\texttt{scope}\ \alpha\ \texttt{skip}) = labels(\texttt{skip}) = \emptyset$ by definition. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \texttt{scope}\ \alpha\ \texttt{skip}$, i.e. $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\forall v^{\beta} \in labels(\texttt{scope}\ \alpha\ \texttt{skip})\ \mu \models \beta \backsim v$. Then we have $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(\texttt{scope}\ \alpha\ \texttt{skip})\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(\texttt{skip})$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{scope}\ \alpha\ \texttt{skip} \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \mu \models \texttt{skip}$. That is if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \texttt{scope}\ \alpha\ \texttt{skip}$ then $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \texttt{skip}$.

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (\alpha\ \texttt{def})\ \texttt{skip} \xrightarrow{\alpha}_{\mathsf{s}} clear(\widehat{\sigma}, \alpha\ \texttt{def}), \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{DefScopeDone}$$

By definition, we have $labels(clear(\hat{\sigma}, \alpha\ \texttt{def})) = labels(\hat{\sigma}) \setminus labels(\{(\alpha'\ \texttt{def}\ x, v') \mid (\alpha'\ \texttt{def}\ x, v') \in \hat{\sigma}, \alpha = \alpha'\})$. We also have $labels(\texttt{scope}\ (\alpha\ \texttt{def})\ \texttt{skip}) = labels(\texttt{skip}) = \emptyset$. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \texttt{scope}\ (\alpha\ \texttt{def})\ \texttt{skip}$. Then $\forall v'^{\beta'} \in labels(\hat{\sigma})\ \mu \models \beta' \backsim v' \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge$
$\forall v^{\beta} \in labels(\texttt{scope}\ (\alpha\ \texttt{def})\ \texttt{skip})\ \mu \models \beta \backsim v \rightarrow \forall v'^{\beta'} \in labels(clear(\hat{\sigma}, \alpha\ \texttt{def})\ \mu \models \beta' \backsim v' \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(\texttt{skip})\ \mu \models \beta \backsim v$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{scope}\ (\alpha\ \texttt{def})\ \texttt{skip} \rightarrow \mu \models clear(\hat{\sigma}, \alpha\ \texttt{def}) \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{skip}$.

□

C.4 Proof of Invariant in Manifest Evaluation

Theorem C.4.1 (Invariant in Manifest Evaluation)**.** For any $\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, m \xrightarrow{N}_{m} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}} \vdash m'$, if $\mu \models \widehat{\sigma}$, $\mu \models \widehat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models m$ then $\mu \models \widehat{\sigma'}$, $\mu \models \widehat{\kappa'}$, $\mu \models \widehat{v'_{\mathrm{C}}}$ and $\mu \models m'$.

Proof. Prove by induction.

1 Top-Level Statements

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\vdots}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, , s'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, , s'}\ \textsc{TopScope}$$

By induction hypothesis, we have if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models s$ then $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models s'$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(s)\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(s')\ \mu \models \beta' \backsim v'$. Since $labels(s) = labels(s)$ and $labels(s') = labels(s')$, $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(s)\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(s')\ \mu \models \beta' \backsim v'$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge s \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge s'$ for $\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, s \xrightarrow{N}_{p} \hat{\sigma}', \kappa', v'_{\mathrm{C}}, s'$ as a program statement evaluation.

2 Sequential Composition

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, m_1 \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m'_1}{\widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m_1 \llcorner m_2 \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m'_1 \llcorner m_2}\ \textsc{MSeqStep}$$

By induction hypothesis, we have if $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models m_1$ then $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models m'_1$. We have $labels(m_1 m_2) = labels(m_1) \cup labels(m_2)$ and $labels(m'_1 m_2) = labels(m'_1) \cup labels(m_2)$ by definition. Suppose $\mu \models m_1 m_2$, i.e. $\forall v^{\beta} \in labels(m_1 m_2)\ \mu \models \beta \backsim v$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(m_1 m_2)\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(m_1) \cup labels(m_2)\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(m'_1) \cup labels(m_2)\ \mu \models \beta' \backsim v' \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v'^{\beta'} \in labels(m'_1 m_2)\ \mu \models \beta' \backsim v'$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models m_1 m_2 \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models m'_1 m_2$.

$$\frac{}{\widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, \mathtt{skip} \llcorner m \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m}\ \textsc{MSeqSkip}$$

We know $labels(\mathtt{skip}) = labels(\mathtt{skip}) \cup labels(m) = labels(m)$ by definition. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \mathtt{skip}\ m$, i.e. $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\forall v^{\beta} \in labels(\mathtt{skip}\ m)\ \mu \models \beta \backsim v$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in\in labels(\mathtt{skip}\ m)\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in\in labels(m)\ \mu \models \beta \backsim v$, i.e. $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \mathtt{skip}\ m \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models m$.

3 Node Definitions

$$\frac{\mathsf{nodeMatch}(N, Q)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{node}\ Q\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (\texttt{::nd})\ s}\ \textsc{NodeMatch}$$

By definition, we know $labels(\texttt{node}\ Q\ \{s\}) = labels(S)$ and $labels(\texttt{scope}\ (\texttt{::}\ \texttt{nd})\ s) = labels(s)$. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models \texttt{node}\ Q\ \{s\}$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \widehat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(\texttt{node}\ Q\ \{s\})\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \widehat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(s)\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \widehat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(\texttt{scope}\ (\texttt{::}\ \texttt{nd})\ s)\ \mu \models \beta \backsim v$. That is $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \widehat{v_{\mathrm{C}}} \wedge \mu \models \texttt{node}\ Q\ \{s\} \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \widehat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mu \models \texttt{scope}\ (\texttt{::}\ \texttt{nd})\ s$.

$$\frac{\neg\mathsf{nodeMatch}(N, Q)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{node}\ Q\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{NodeNoMatch}$$

By definition, we know $labels(\texttt{node}\ Q\ \{s\}) = labels(s)$ and $labels(\texttt{skip}) = \emptyset$. We know that $\forall v^{\beta} \in labels(\texttt{skip})\ \mu \models \beta \backsim v \rightarrow \forall v^{\beta} \in \emptyset\ \mu \models \beta \backsim v = \texttt{true}$. Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \widehat{v_{\mathrm{C}}}$ and $\mu \models \texttt{node}\ Q\ \{s\}$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \widehat{v_{\mathrm{C}}} \wedge \mu \models \texttt{node}\ Q\ \{s\} \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \widehat{v_{\mathrm{C}}} \wedge \texttt{true} \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \widehat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in labels(\texttt{skip})\ \mu \models \beta \backsim v$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \widehat{v_{\mathrm{C}}} \wedge \mu \models \texttt{node}\ Q\ \{s\} \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \widehat{v_{\mathrm{C}}} \wedge \forall v^{\beta} \in \mu \models \texttt{skip}$.

4 Defined resource types

$$\frac{u \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{define}\ u\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}[u : \mathsf{ResourceDef}(\rho, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{RDef}$$

The proof is similar as CDef.

5 Class Definitions

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ a\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \widehat{\kappa}[a : \mathsf{ClassDef}(\bot, \varepsilon, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{CDef}$$

We have $labels(\texttt{class}\ a\ \{s\}) = labels(s)$ and $labels(\texttt{skip}) = \emptyset$ by definition. We also know $labels(\kappa(a \rightarrow \mathsf{ClassDef}(\bot, \varepsilon, s))) = labels(\hat{\kappa}) \cup$

$labels(\mathsf{ClassDef}(\bot, \rho, s)) = labels(\hat{\kappa}) \cup labels(\rho) \cup labels(s)$ and
$labels(\kappa(a) = \bot) = \emptyset$ by definition.

Suppose $\mu \models \hat{\sigma}$, $\mu \models \hat{\kappa}$, $\mu \models \hat{v_{\mathrm{C}}}$ and $\mu \models \texttt{class}\ a\ \{s\}$. We also know $\forall v^{\beta} \in \texttt{skip}\ \mu \models \beta \backsim v \rightarrow \forall v^{\beta} \in \emptyset\ \mu \models \beta \backsim v = \texttt{true}$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{class}\ a\ \{s\} \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \forall v^{\beta} \in labels(\texttt{class}\ a\ \{s\})\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \forall v^{\beta} \in labels(\hat{\kappa})\ \mu \models \beta \backsim v \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \forall v^{\beta} \in labels(s)\ \mu \models \beta \backsim v \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \forall v^{\beta} \in labels(\hat{\kappa}) \cup labels(s)\ \mu \models \beta \backsim v \wedge \texttt{true} \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa}(a \rightarrow \mathsf{ClassDef}(\bot, \varepsilon, s)) \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{skip}$. Then $\mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa} \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{class}\ a\ \{s\} \rightarrow \mu \models \hat{\sigma} \wedge \mu \models \hat{\kappa}(a \rightarrow \mathsf{ClassDef}(\bot, \varepsilon, s)) \wedge \mu \models \hat{v_{\mathrm{C}}} \wedge \mu \models \texttt{skip}$

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ a\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \widehat{\kappa}[a : \mathsf{ClassDef}(b, \varepsilon, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{CDefI}$$

The proof is similar as CDef.

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ a\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \widehat{\kappa}[a : \mathsf{ClassDef}(\bot, \rho, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{CDefP}$$

The proof is similar as CDef.

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ a\ (\rho)\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}[a : \mathsf{ClassDef}(b, \rho, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{CDefPI}$$

The proof is similar as CDef.

□

Appendix D

Dependency Provenance

D.1 Propagation of Annotations in the Evaluation

D.1.1 Expressions

D.1.1.1 Boolean expressions

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e_1 \xrightarrow{\alpha} e_1'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e_1\,\mathtt{and}\,e_2 \xrightarrow{\alpha} e_1'\,\mathtt{and}\,e_2}\ \textsc{AndLeft}$$

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{false}^{\beta}\,\mathtt{and}\,e \xrightarrow{\alpha} \mathtt{false}^{\beta}}\ \textsc{AndRightI}$$

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{true}^{\beta}\,\mathtt{and}\,e \xrightarrow{\alpha} \mathtt{true}^{\beta}\,\mathtt{and}\,e'}\ \textsc{AndRightII}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},\mathtt{true}^{\beta_1}\,\mathtt{and}\,\mathtt{true}^{\beta_2} \xrightarrow{\alpha} \mathtt{true}^{\beta_1\cup\beta_2}}\ \textsc{AndValue}$$

$$\frac{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},e \xrightarrow{\alpha} e'}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},!\,e \xrightarrow{\alpha} !\,e'}\ \textsc{NotSTEP} \qquad \frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},!\mathtt{true}^{\beta} \xrightarrow{\alpha} \mathtt{false}^{\beta}}\ \textsc{NotValueI}$$

$$\frac{}{\widehat{\sigma},\widehat{\kappa},\widehat{v_{\mathrm{C}}},!\mathtt{false}^{\beta} \xrightarrow{\alpha} \mathtt{true}^{\beta}}\ \textsc{NotValueII}$$

D.1.2 Statements

D.1.2.1 Sequential composition

$$\frac{\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, s_1 \xrightarrow{\alpha}_s \hat{\sigma}', \hat{\kappa}', \hat{v'_{\mathrm{C}}}, s'_1}{\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, s_1\ s_2 \xrightarrow{\alpha}_s \hat{\sigma}', \hat{\kappa}', \hat{v'_{\mathrm{C}}}, s'_1\ s_2} \qquad \frac{}{\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, \texttt{skip}\ s \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, s}$$

D.1.2.2 Resource

$$\frac{\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, R \xrightarrow{\alpha} R'}{\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, R \xrightarrow{\alpha}_s \hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, R'} \qquad \frac{}{\hat{\sigma}, \hat{\kappa}, \hat{v_{\mathrm{C}}}, v_{\mathrm{R}} \xrightarrow{\alpha}_s \hat{\sigma}, \kappa, v_{\mathrm{R}}\ v_{\mathrm{C}}, \texttt{skip}}$$

D.1.2.3 Expression

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e'}\ \textsc{ExprStep} \qquad \frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, v^{\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{Expr}$$

D.1.2.4 Assignment

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x = e \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x = e'}\ \textsc{AssignStep}$$

$$\frac{x \notin dom(\sigma_\alpha)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \$x = v^{\beta} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}[(\alpha, x) : v^{\beta}], \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{Assign}$$

D.1.2.5 Unless

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e \xrightarrow{\alpha} e'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{unless}\ e\ \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{unless}\ e'\ \{s\}}\ \textsc{UnlessStep}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{unless}\ \texttt{true}^{\beta}\ \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{UnlessT}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{unless}\ \texttt{false}^{\beta}\ \{s\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s^{+\beta}}\ \textsc{UnlessF}$$

D.1.2.6 Include

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = \mathit{merge}(\rho, \varepsilon) \qquad \beta \,\mathsf{baseof}_\kappa\, \alpha}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}{}'_{\mathsf{s}} \widehat{\sigma}, \kappa[a : \mathsf{DeclaredClass}(\beta)], \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ s' \llcorner s} \textsc{IncU}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{DeclaredClass}(\beta)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}} \textsc{IncD}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \widehat{\kappa}(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ b\ \texttt{include}\ a} \textsc{IncPU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{DeclaredClass}(\beta) \qquad s' = \mathit{merge}(\rho, \varepsilon)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ a \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa[a : \mathsf{DeclaredClass}(::b)], \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ \{s' \llcorner s\}} \textsc{IncPD}$$

D.1.2.7 Defined resource type

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \{e : H\} \xrightarrow{\alpha}_{\mathsf{R}} \{e' : H'\}}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, u\ \{e : H\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, u\ \{e' : H'\}} \textsc{DefStep}$$

$$\frac{\widehat{\kappa}(u) = \mathsf{ResourceDef}(\rho, s) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, u\ \{w^\beta : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (\alpha\ \texttt{def})\ \{\$\mathit{title} = w^\beta \llcorner s' \llcorner s\}} \textsc{Def}$$

D.1.2.8 Resource declaration

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, e : H \xrightarrow{\alpha}_{\mathsf{R}} e' : H'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, t\ \{e : H\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, t\ \{e' : H'\}} \textsc{ResStep}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \widehat{v_{\mathrm{R}}} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}} \llcorner \widehat{v_{\mathrm{R}}}, \texttt{skip}} \textsc{ResDecl}$$

D.1.2.9 Resource-like-class-declaration

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(c_{opt}, \rho, S) \qquad \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ \{a : H\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ \{a : H'\}}\ \textsc{CDecStep}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = merge(\rho, v_{\mathrm{H}}) \qquad \beta\ \mathsf{baseof}_{\kappa}\ \alpha}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \kappa[a : \mathsf{DeclaredClass}(\beta)], \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ s' \llcorner s}\ \textsc{CDecU}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \widehat{\kappa}(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{include}\ b\ \texttt{class}\ \{a : v_{\mathrm{H}}\}}\ \textsc{CDecPU}$$

$$\frac{\widehat{\kappa}(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \widehat{\kappa}(b) = \mathsf{DeclaredClass}(\beta) \qquad s' = merge(\rho, v_{\mathrm{H}})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma}, \kappa[a : \mathsf{DeclaredClass}(::b)], \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (::a)\ \{s' \llcorner s\}}\ \textsc{CDecPD}$$

Scope

$$\frac{\alpha \in \{::, ::a, ::\texttt{nd}\} \qquad \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha}_{\mathrm{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ \alpha\ s \xrightarrow{\alpha'}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, \texttt{scope}\ \alpha\ s'}\ \textsc{ScopeStep}$$

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{\alpha\ \texttt{def}}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (\alpha\ \texttt{def})\ s \xrightarrow{\alpha}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, \texttt{scope}\ (\alpha\ \texttt{def})\ s'}\ \textsc{DefScopeStep}$$

$$\frac{\alpha \in \{::, ::a, ::\texttt{nd}\}}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ \alpha\ \texttt{skip}^{\beta} \xrightarrow{\beta}_{\mathsf{s}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}^{\beta}}\ \textsc{ScopeDone}$$

$$\frac{}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{scope}\ (\alpha\ \texttt{def})\ \texttt{skip}^{\beta} \xrightarrow{\alpha}_{\mathsf{s}} clear(\widehat{\sigma}, \alpha\ \texttt{def}), \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{skip}^{\beta}}\ \textsc{DefScopeDone}$$

D.1.3 Manifests

Statement

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{::}_{\mathsf{s}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, s \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, s'}\ \textsc{TopScope}$$

Sequential Composition

$$\frac{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, m_1 \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m'_1}{\widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m_1 \llcorner m_2 \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m'_1 \llcorner m_2} \text{ MSeqStep}$$

$$\frac{}{\widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, \mathtt{skip}^{\beta} \llcorner m \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma'}, \widehat{\kappa'}, \widehat{v'_{\mathrm{C}}}, m} \text{ MSeqSkip}$$

Node Definition

$$\frac{\mathsf{nodeMatch}(N, Q)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{node}\ Q\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{scope}\ (::\mathtt{nd})\ s} \text{ NodeMatch}$$

$$\frac{\neg\mathsf{nodeMatch}(N, Q)}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{node}\ Q\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{skip}} \text{ NodeNoMatch}$$

$$\begin{aligned}
\mathsf{nodeMatch}(N, N) &= \mathtt{true} \\
\mathsf{nodeMatch}(N, N') &= \mathtt{false} \quad (N \neq N') \\
\mathsf{nodeMatch}(N, (N', Ns)) &= \begin{cases} \mathtt{true} & \text{if } N = N' \\ \mathsf{nodeMatch}(N, Ns) & \text{if } N \neq N' \end{cases} \\
\mathsf{nodeMatch}(N, r) &= \begin{cases} \mathtt{true} & \text{if } N \in L(r) \\ \mathtt{false} & \text{if } N \notin L(r) \end{cases}
\end{aligned}$$

Defined resource type

$$\frac{u \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \mathtt{define}\ u\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}[u : \mathsf{ResourceDef}(\rho, s)], \widehat{v_{\mathrm{C}}}, \mathtt{skip}} \text{ RDef}$$

Class definition

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ a\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \widehat{\kappa}[a : \mathsf{ClassDef}(\bot, \varepsilon, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{CDef}$$

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ a\ \texttt{inherits}\ b^{\beta}\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \widehat{\kappa}[a : \mathsf{ClassDef}(b^{\beta}, \varepsilon, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{CDefI}$$

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ a\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \widehat{\kappa}[a : \mathsf{ClassDef}(\bot, \rho, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{CDefP}$$

$$\frac{a \notin dom(\widehat{\kappa})}{\widehat{\sigma}, \widehat{\kappa}, \widehat{v_{\mathrm{C}}}, \texttt{class}\ a\ (\rho)\ \texttt{inherits}\ b^{\beta}\ \{s\} \xrightarrow{N}_{\mathsf{m}} \widehat{\sigma}, \widehat{\kappa}[a : \mathsf{ClassDef}(b^{\beta}, \rho, s)], \widehat{v_{\mathrm{C}}}, \texttt{skip}}\ \textsc{CDefPI}$$