Interaction-Oriented Programming

by

Yu David Liu

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

September, 2007

Abstract

This dissertation describes the design and implementation of a general-purpose object-oriented (OO) programming language, *Classages*. The novel object model of *Classages* gives programmers refined control over modeling the interactions inside OO software, with inspirations drawn from human sociology. Key innovations include the language constructs of Mixers and Connectors as encapsulation-enforceable interfaces for specifying bi-directional interaction behaviors, and a novel type system called Pedigree Types to help programmers organize the object heap into a hierarchy reflecting the fundamental principle of hierarchical decomposition. Important properties of the language, including type soundness, hierarchy shape enforcement, and alias protection, are formally established and proved. A prototype compiler is implemented.

The object model of *Classages* sets a significant departure from the familiar one taken by Smalltalk, Java, C++, and C#. Its simple, expressive, and rigorously defined core enriches the theoretical foundations of OO languages. From the perspective of software engineering, the *Classages* language is particularly good at controlling software complexity, a crucial goal in modern software development.

Advisor: Prof. Scott Smith, The Johns Hopkins University

Readers: Prof. Michael Hicks, University of Maryland, College Park

Prof. Scott Smith, The Johns Hopkins University

Prof. Andreas Terzis, The Johns Hopkins University

Acknowledgements

I have been extremely lucky to study under Scott Smith. His unusual ability to think rigorously and intuitively at the same time, his determination to explore iconoclastic paths and seek fundamental solutions, his open-mindedness on selecting research directions, and his innate cheerfulness for tackling new problems, all constitute the best of a researcher and teacher I look up to and strive to learn from. The everyday Scott is accessible 24/7, remarkably modest, always willing to have meetings in cafes, and excited about lending me a book on Easter Island. As a language designer, I find it impossible to construct a language expressive enough to describe my gratitude and the good time I have had working with him, so let some traces of our interactions keep the good memory alive: “Occam’s Razor”,“Principles! Principle!”, “The Brave New World”, ”Eureka!”, “Be Here Now”.

I am indebted to my thesis committee members Michael Hicks and Andreas Terzis, both of whom have been very supportive throughout the years, from the oral exam in 2004 to the dissertation defense in 2007, with numerous helpful emails and meetings in between. I would like to thank Robert Rynasiewicz and Christian Scheideler for serving on my oral exam committee. I owe my debt of gratitude to Jens Palsberg and Matthew Flatt for their appreciation of my research, and their wise counsel and unstinting support for my career development. I am grateful to Jan Vitek for giving me the opportunity to attend the exciting “Trends in Concurrency” summer school in Bertinoro, Italy.

My thanks go to all the intrepid teammates in the programming language group. I would like to thank Paritosh Shroff for his help with the proof, and for joining me in many safaris, tracking down pizzas and burgers. I would like to thank Xiaoqi Lu for the pleasant collaboration on Coqa, and thank Ran Rinat for

patiently working with me on Cells. It was a great experience to have stimulating discussions with Fei Lu, Chris Skalka, Mark Thober, and Tiejun Wang, during our weekly PL seminars and beyond.

As the title of the dissertation suggests, I firmly believe in the values of interactions. The most convincing evidence of their merits is perhaps how much I myself have benefited from the interactions with my friends, each of whom I am thankful for. Among them, a number of people have offered more help than I could ever ask for, and have shaped me into who I am: Amitabha Bagchi, Susan Collins, Deborah D., Markus Dreyer, Xiaodong Fan, Haipeng Gong, Xiaoyan He, Jie Guo, James Long, Weikun Luo, Gideon Mann, Randy M., Michael Moon, Eric Perlman, David Rasch, Anshumal Sinha, Noah Smith, Michael V., Roxana Vicovanu, Binze Yang, Hao Zhou.

The majority of this dissertation was written in a number of Baltimore cafes that have been my secondary offices throughout the graduate school years: City Cafe in Mt. Vernon, Starbucks on St. Paul Street in Charles Village, One World Cafe on University Parkway, and the MSE Library's Cafe Q.

I dedicate this dissertation to my parents, Xinglin Liu and Suyun Teng. You named me after a journalist, but this dissertation is anything but journalism. I am deeply grateful for the freedom you have always given me to make my own decisions, explore my own path, and see the world in my own eyes. The freedom is a beautiful thing because it is full of your love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past two decades, modern software has experienced the explosive growth in size and complexity. When Microsoft's first operating system, MS-DOS, was released in the early 1980s, the software was written in just 4,000 lines of code [All01]. In contrast, the most recent operating system from the same company, Windows Vista, is believed to contain at least 50 millions lines of code [Dah07, Man07]. The emergence of open source software offers opportunities for collaborative software development, which further boosts the software growth: as of 2005, the code base of Debian GNU/Linux 3.1 is formed by 215 million lines of code [AGBRH05]. Recently, U.S. Department of Defense posed the following question to the software research community [NFG+06]: "Given the issues with today's software engineering, how can we build the systems of the future that are likely to have billions of lines of code?" Effective approaches to developing robust, well-structured, and maintainable software are highly sought after in real-world software development practices.

If you happen to be a sociologist, the situation above is likely to create a strong sense of *déjà vu*. Around 100 years ago, the world population increased dramatically and complex ways of social organization ensued. The search for effective approaches to studying human societies was also highly sought after. The outcome? Sociology as an independent discipline was born, and one of the founding fathers Georg Simmel provided an influential definition for studying society [Sim02]:

"Society is merely the name for a number of individuals, connected by interaction."

According to this definition, the complexity of society is not a result of the isolated behaviors of individuals, but more of *how they interact*. With this key insight, studying modern society with intimidating size and complexity became less intimidating than it had seemed. Could this time-honored sociological definition help our understanding and practice of software design? If we analogously view software as society, here is the Simmel's answer to the field of software design:

"Software is merely the name for a number of software building blocks, connected by interaction."

At the first glance, this Simmelian definition does not surprise us; it is standard knowledge that software is made of smaller building blocks pieced together. However, the implicit message sent out from this definition has a subtle shift of focus on software design: to help software developers capture the complexity of modern software, software engineering and programming language researchers should focus on providing better language constructs, methodologies, and tools for modeling the *interactions* of software building blocks. We broadly term this general design principle as *Interaction-Oriented Programming* (IOP).

We have explored how IOP can lead to novel language designs and software development models in various contexts [LS05, LS07, LS04, LRS03, LS06, LLS07, LS02, LST04], and this dissertation details a specific instance: how it can lead to the design and implementation of an object-oriented language, *Classages* [LS05, LS07]. The rest of this chapter gives a high-level account of the language, with a focus on the design goals. The important question "why a new language?" will be deferred to Chapter 3 after adequate background information is given in Chapter 2.

1.1 A Sneak Peek at *Classages*

For OO software, the building blocks are without doubt classes and objects. The IOP principle in this context is naturally reflected as

"Object-oriented software is merely the name for a number of classes and objects, connected by interaction."

On the highest level, the *Classages* language intends to answer the following question: how can we provide more refined support for interactions between classes and between objects? Indeed, in the standard object model used by Java/C++, interactions do happen: two classes interact via inheritance and two objects interact via message passing. However, do mechanisms such as inheritance and message passing provide the most natural language abstractions for what we intuitively know of interactions? For instance, consider how message passing will be modeling the following example, the most basic interaction in the human world:

A : Hello B!

B : Hello A! How are you?

A : I'm fine. Thank you, and you?

B : I'm fine too.

Most would agree that in this scenario, there is only *one* meaningful interaction happening, namely, greeting. When a program is written in Java/C++ where A and B are both objects, such a scenario will be implemented *four* messages passed back and forth. If message passing is viewed as Java's language abstraction for interactions, we would have four interactions for the previous scenario! Such a view is not only unnatural, but also can it lead to a wide range of problems when the program becomes more complex and an object interacts with many other objects. It is the task of this dissertation to illustrate these problems, and demonstrate how *Classages* can avoid them by providing more natural language abstractions for interactions. In *Classages*, "greeting" in the example above is represented by a first-class language construct.

Classages, as the name suggests, is a class-based OO language. Classes and objects are termed *classages* and *objectages* respectively in *Classages*. Following the standard definitions, a classage is a code template to describe what the objectages instantiated from it at run-time might look like. We now illustrate several high-level concepts and the design goals of *Classages* with a real-world programming example.

A Protein Folding Simulation Application Protein folding is the structure-forming process of proteins when the sequence of amino acids is given. The simplified design is illustrated by a Unified Modeling Language (UML) [Obj07] class diagram in Fig. 1.1. UML is a modeling standard for software design, independent of the programming language used at the implementation phase. UML helps software designers

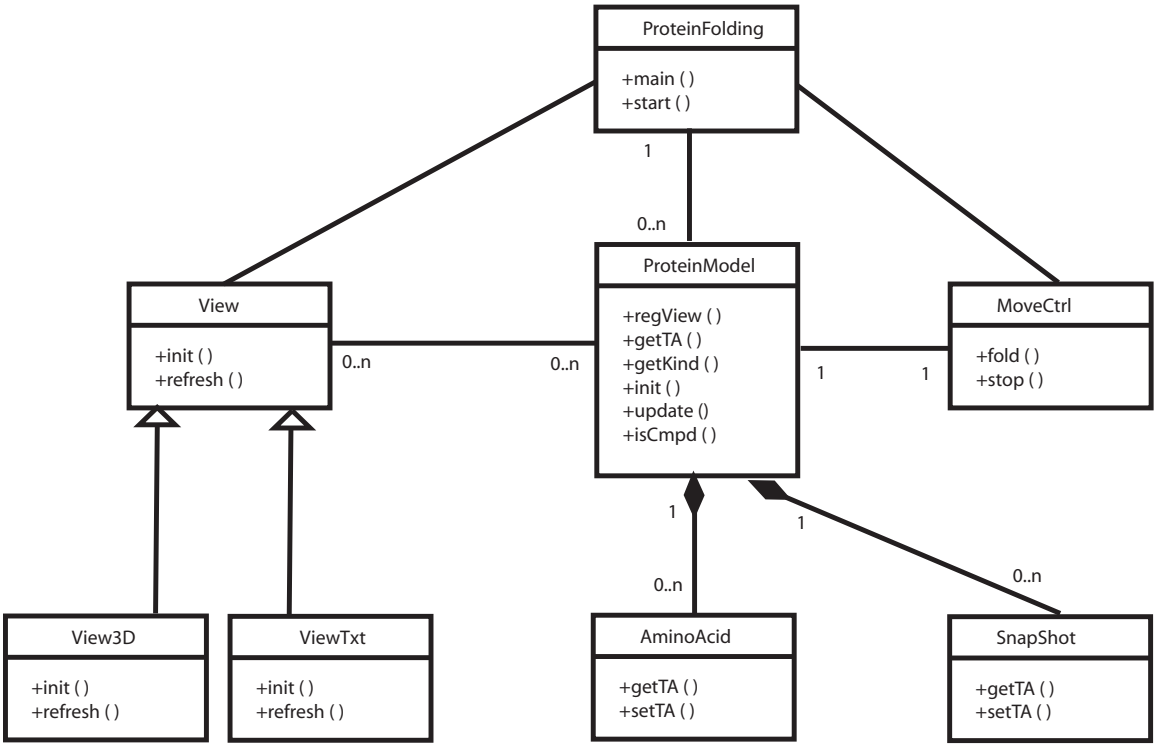


Figure 1.1: A Simplified Protein Folding Application in a UML Class Diagram (Method Signatures Omitted)

express their intentions on a relatively high level, at a relatively early phase. The application is constructed through a standard Model-View-Controller (MVC) pattern [KP88]:

- The "model" class – the `ProteinModel` class here – keeps track of the data. It holds a UML composition relationship with the `AminoAcid` class, denoting the fact that when this application is up and running, each `ProteinModel` instance is composed of multiple `AminoAcid` instances. The `ProteinModel` class also holds a UML composition relationship with the `Snapshot` class, which is used to periodically record the status of the `ProteinModel` object as it gradually folds (following the standard Momento pattern [GHJV95]). The 3D structure of the sequence is recorded by the `AminoAcid` instances, each of which records its *torsion angles* with respect to the previous `AminoAcid` instance in the sequence; the `getTA` and `setTA` methods are used for access to these values.

- The "view" class – `View` and its subclasses `View3D` and `ViewTxt` here – concerns how the data is displayed. For human interaction, the folding process is typically illustrated by 3D graphs (`View3D`) and/or text output (`ViewTxt`), following the Observer pattern [GHJV95].

- The "controller" class – `MovelCtrl` class here – manipulates the data. It includes the core algorithm whose major method `fold` repeatedly selects at random three consecutive amino acids on the sequence and *move*-s them in a way such that the resulting 3D structure has a lower energy.

After thousands of successful moves, the final structure of the protein is obtained by analyzing the statistical distribution of the `SnapShot`s.

How this diagram can be implemented in a Java-like language is clear. *Classages* implementations of the same diagram can be illustrated by Fig. 1.2, Fig. 1.3, and Fig. 1.4. Fig. 1.2 provides a "static", *i.e.* compile-time, view of the application composed of classages, while Fig. 1.3 and Fig. 1.4 are both snapshots of the running application composed of objectages, the "dynamic" view. Details of these figures will be elaborated further in later chapters of the dissertation. The focus for now is to "get a feel of " the language and understand its high-level goals.

Design Goals *Classages* achieve two main design goals in supporting interactions, the first one of which is relatively "local" as it concerns the interactions between a pair of classes or objects, while the second one is "global" in its focus on the entire interaction structure of the object heap. We now describe the two goals more concretely.

The first goal of *Classages* is to provide explicit language support for refined notions of binary interactions. The dissertation will elaborate on the following issues:

- What are the fundamental interactions to support? *Classages* explicitly support the inter-classage interaction between `View` and `Win3D` (as in Fig. 1.2, which eventually produces `View3D` in Fig. 1.3), and the inter-objectage interaction between `ProteinModel` and `MoveCtrl` (as in Fig. 1.3). Both forms of interactions carry the high-level notion of what one naturally knows of interactions, in the same spirit as the "greeting" interaction we gave at the beginning of the section. For instance, the inter-

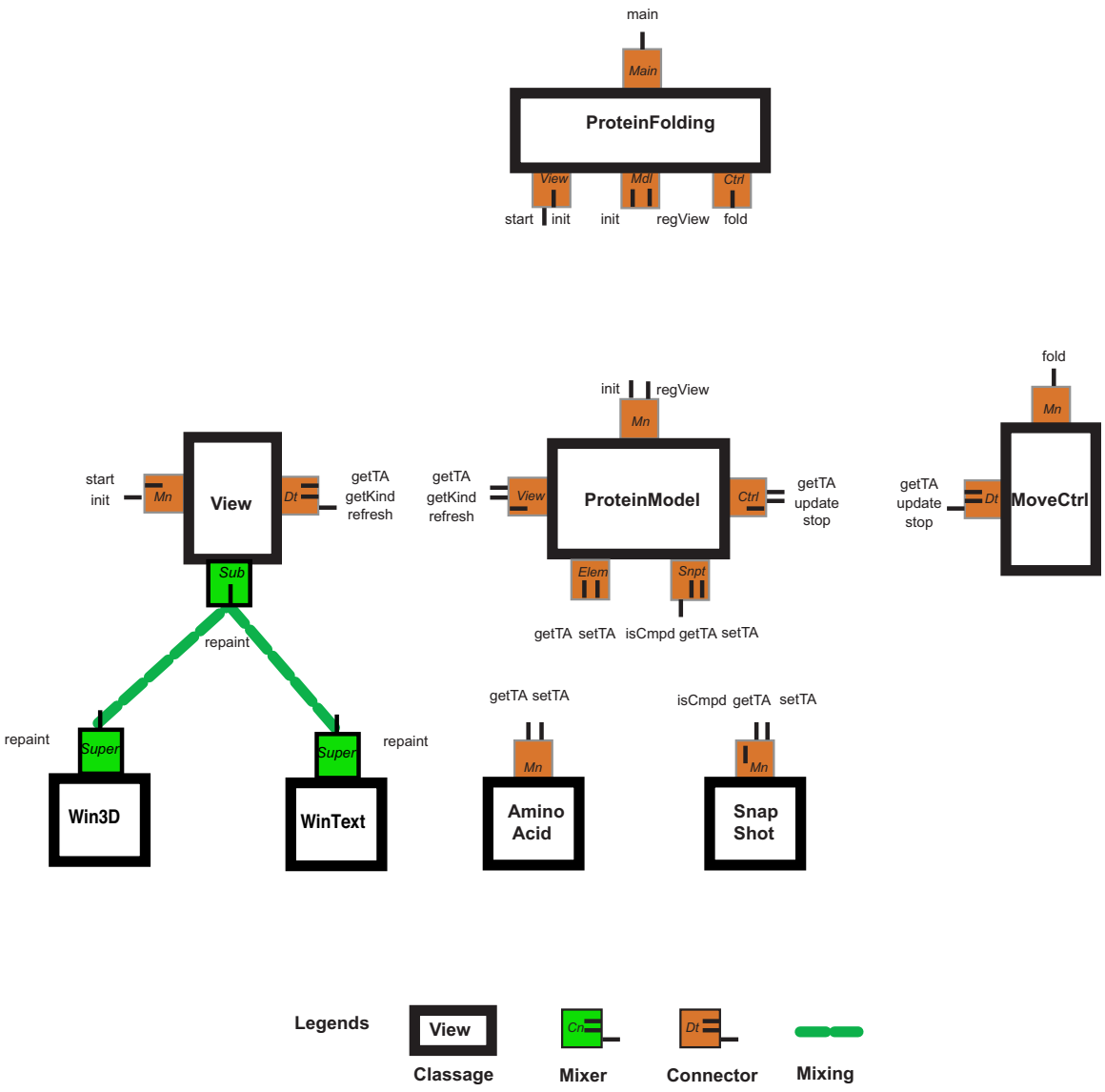


Figure 1.2: A Static View of Fig. 1.1 in *Classages*

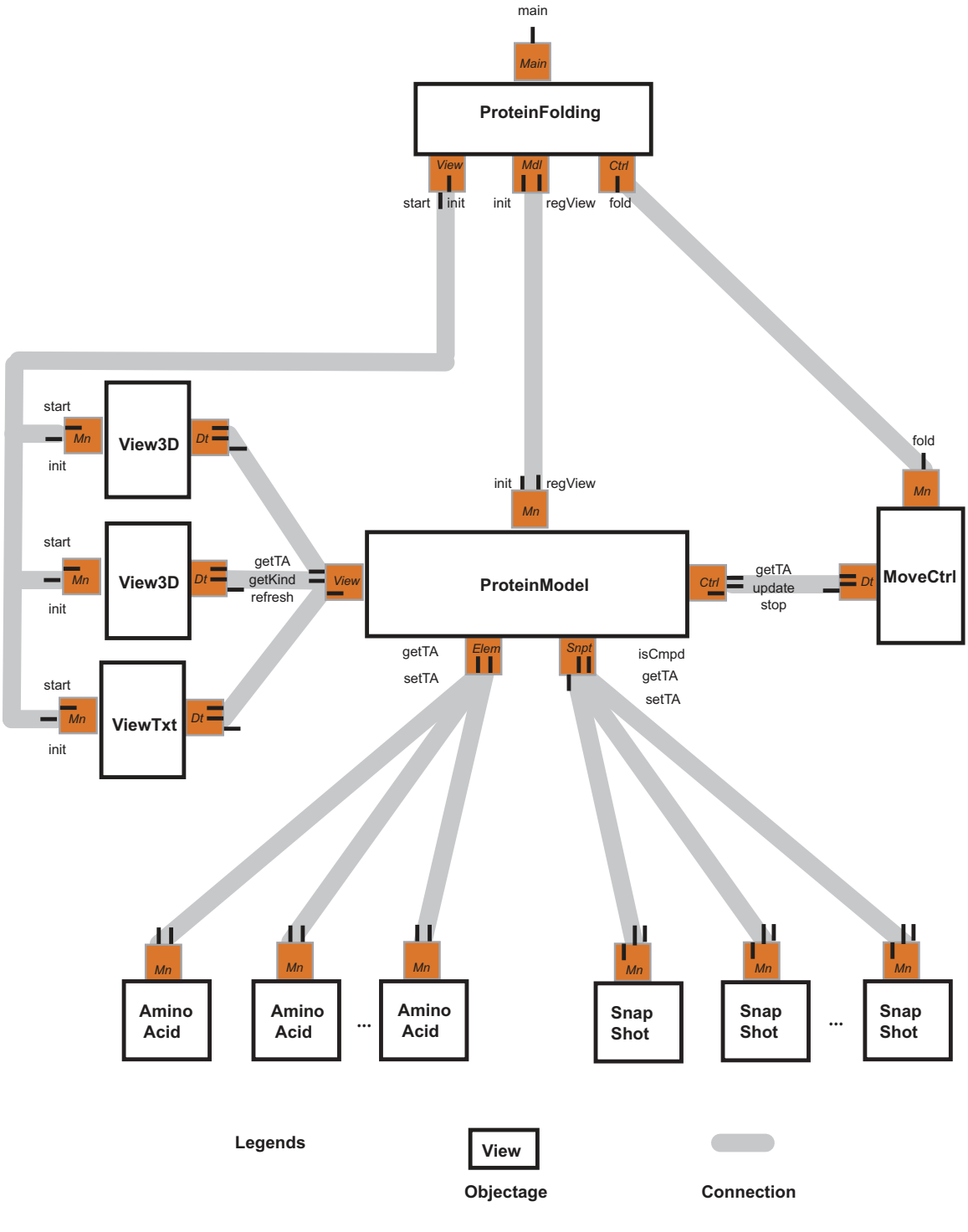


Figure 1.3: A Dynamic View of Fig. 1.1 in *Classages*

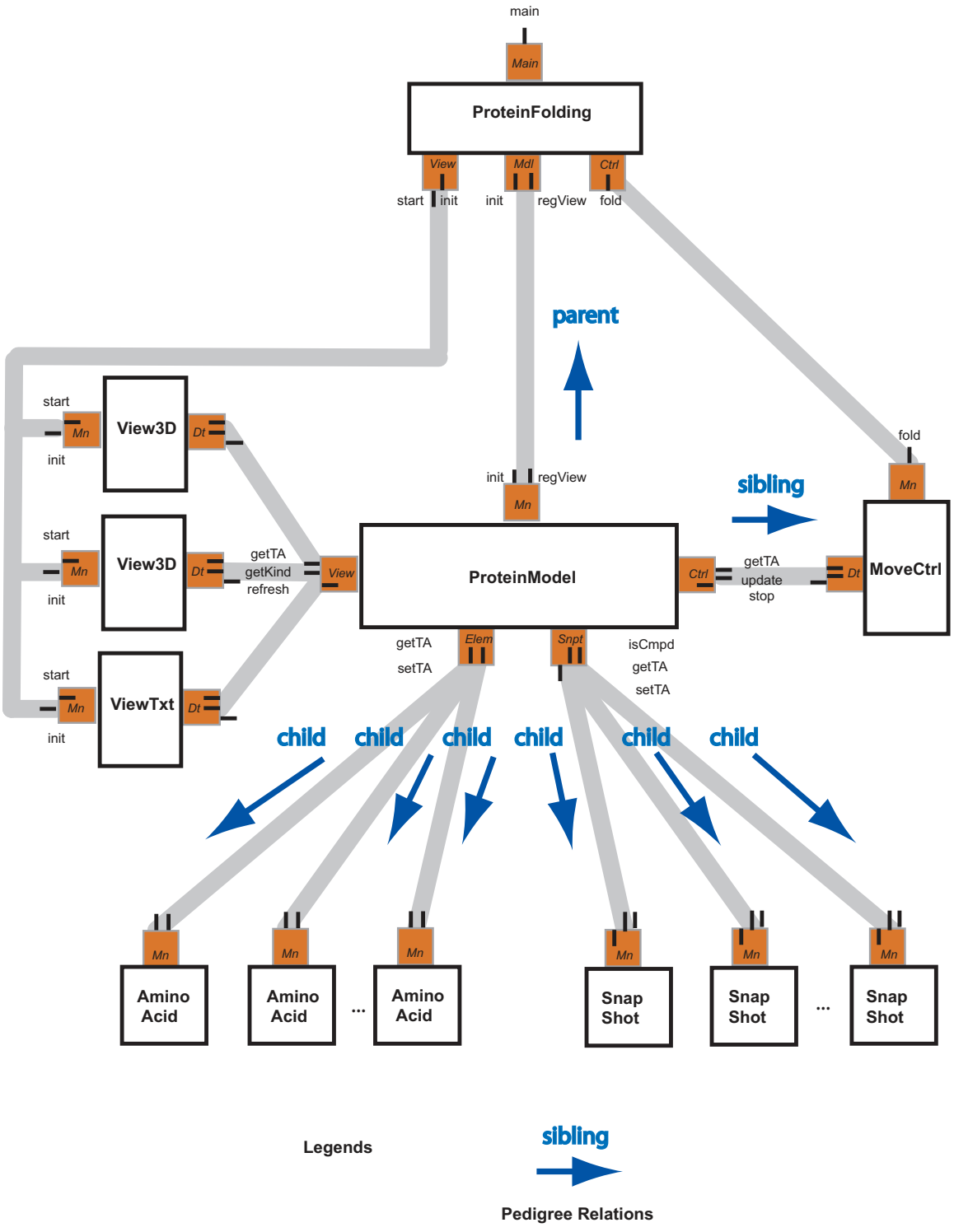


Figure 1.4: A Dynamic View of Fig. 1.1 in *Classages* with Pedigree Types

action between `View` and `Win3D` has the high-level meaning of "customization", while the interaction between `ProteinModel` and `MoveCtrl` has the high-level meaning of "data control".

- How to support the innate properties of interactions, such as their bi-directionality, multiplicity, lifespan, and statefulness? The bi-directionality property is reflected in the "data control" interaction between `ProteinModel` and `MoveCtrl`, where the two involved parties mutually depend on each other for that interaction, and send messages back and forth. The multiplicity property of interactions can be seen in the interaction between `ProteinModel` and `View3D`, as illustrated in Fig. 1.3, where a `ProteinModel` objectage can interact with multiple instances of `View3D` at the same time. Each interaction has its own lifespan, as there is always a point in time to establish and discontinue the interaction between a `ProteinModel` objectage and a `View3D` objectage. Interactions are also fundamentally stateful: for instance, in a many-to-many scenario for `ProteinModel` and `View3D` – *i.e.* each `ProteinModel` is displayed by multiple `View3D`'s while each `View3D` displays the results of multiple `ProteinModel`'s – the refreshing rate for displaying the protein should be a state belonging to the interaction of the two involved parties.

- Can we represent interactions as values? In *Classages*, the aforementioned objectage interactions between `ProteinModel` and `MoveCtrl`, or between `ProteinModel` and `View3D`, are represented as values that can be passed from one method to another, or stored in an object field.

- How to enforce the correctness of interactions without running the program? The type system of *Classages* can ensure all type-checked programs will never run into run-time errors as a result of "bad" interactions.

A fundamental design principle of *Classages* is that each binary interaction can only be established between a pair of *interaction interfaces*, each from one party being involved. Thus, each interaction interface becomes a specification point for each party on *its* view of a specific interaction, such as what it expects from the interaction, and what it can provide to the interaction. All together, interaction interfaces that a classage owns completely specify how this classage and its run-time instances can interact with the outside world. This is important for controlling software complexity: a clear specification in such a fashion gives strong

encapsulation of the building blocks (classes and objects), and makes explicit the dependencies between them.

The second goal of *Classages* is to enforce the "shape" of the object heap, *i.e.*, the global structure of interactions. One specific "shape" this dissertation focuses on is the hierarchy. Hierarchical decomposition is a fundamental principle for controlling software complexity. With many of today's systems evolving into Ultra-Large-Scale [NFG+06] ones, the gain of decomposing a system into a well-structured hierarchy and tackling complexity in a divide-and-conquer fashion cannot be overstated. Language support for static hierarchical decomposition is common knowledge for module systems. Dynamic hierarchical decomposition – *i.e.* shaping the object heap of a running application into a hierarchy of encapsulation – has the further benefits of promoting modular program understanding with per-instance precision and impacting more advanced language features such as security [VB99] and concurrency [BLR02, ACG+06]. As illustrated by Fig. 1.1, the interaction between the `ProteinModel` instance and the `AminoAcid` instance is not merely one between objects of "equal social status". The composition relationship between the two represents the programmer's intention of hierarchical decomposition and encapsulating `AminoAcid` instances inside the boundary of the `ProteinModel`. *Classages* explore how to use a novel type system called *Pedigree Types* to intuitively "shape" the dynamic object heap into a hierarchy – as illustrated in Fig. 1.4 – without running the program.

An imprecise yet intuitive way of understanding *Classages* is the language provides better programming language support for the *lines* in UML class diagrams, *i.e.* the relationships between classes and their instances. Make no mistake: our goal is not to explicitly create a UML-based programming language, but to create a language that does the best job of expressing class/object interactions; it just so happens that UML does a better job in this regard than is commonly found in OO language syntax, and so our language shares several features of UML. Some of the new features on the other hand do not have a direct UML analogue. Having a language that more directly expresses object interactions, such as those found in an initial UML design, will bring more explicit intuitions about the high-level design into the program, which will help make the programming phase a smooth continuation of the software design process.

Formalization and Implementation The *Classages* language core is rigorously formalized. Operational semantics is defined via small-step reductions. Static semantics is defined via a type system. Important properties such as type soundness, shape enforcement, and alias protection are proved. A prototype compiler is implemented and can be downloaded at `http://www.cs.jhu.edu/~yliu/Classages`.

1.2 Roadmap of the Dissertation

The rest of the dissertation is structured as follows. Chapter 2 provides an overview to the background knowledge related to this work. Chapter 3 gives a more technical account for the motivations behind *Classages*. Chapter 4 and Chapter 5 give an informal account for two technical parts at the heart of *Classages*, on modeling interactions and on pedigree types. These ideas are formalized rigorously in Chapter 6. A prototype implementation of the *Classages* is described in Chapter 7. Straightforward extensions and variations of the language are discussed in Chapter 8, before we conclude in Chapter 9. The appendix contains all the proofs for the formal properties we stated in the main text, together with auxiliary definitions needed by the proof.

Chapter 2

Background

This chapter provides some background information about important OO concepts and related object-oriented features and systems. Experienced readers can safely skip this chapter and move on directly to Chapter 3.

2.1 The Standard Object Model

We first summarize several fundamental concepts and recurring themes among OO languages. The term “standard” is without doubt a rather subjective one. Any attempt to summarize the kaleidoscopic landscape of object-oriented programming is certain to have numerous over-generalizations. The summary here follows the spirit of *A Theory of Objects* [AC96], a widely read book containing discussions on the essential features of OO languages.

The first OO programming language, Simula 67 [DMN70], was developed in the late 1960s. Compared with other programming paradigms – such as procedural languages and functional languages – OO languages are often considered [AC96] to have an intuitive correspondence between a software simulation of a physical system and the physical system itself. To develop a simulation program for protein folding, each protein – an ”object” in the physical world – is also naturally captured as a computation building block `protein` in the software system, simulating the status and the behavior of a protein. Such a `protein`

building block is thus figuratively named an *object* as well. The paradigm of constructing software systems with objects is referred to as OO Programming. Thanks to its intuitive alignment of the computer world with the physical world, OO programming flourished in the 1980s. Over the decades, many object-oriented languages have been designed. Widely known ones include Smalltalk [GR83], C++ [Str97], Self [US87], and more recently, Java and C#. Today, the vast majority of widely used languages – even if they were not designed to be object-oriented originally – also support object-oriented features as language extensions, such as Caml, Perl, and Python.

Classes, Objects, and their Internal Members The majority of OO languages in wide use are *class-based*, where a class is a code piece whose run-time instances are objects. The following Java code defines a class called `Protein` [1]:

```
class Protein {
    String name;
    ...
    void fold() {    ... code omitted ...    }
    void setName(String n) {    name = n;    }
    Protein(){... code omitted ...    }
}
```

This definition says, all `Protein` objects must have a `name` and know how to behave themselves if being asked to `fold` and `setName`. The internal members of the class, sometimes called its *implementation*, include *fields*, *methods*, and *constructors*. A field, such as `name` in the example, keeps track of the potentially mutable state information of objects. Different objects instantiated from the same class have the same number and type of fields, but what is contained in the fields can vary from one object to another. For instance, two objects instantiated from `Protein` can one have the name `''insulin''` in the `name` field while the other has `''hemogoblin''`. A method, such as `fold` and `setName` defines the behaviors of the objects instantiated from the class, on "what they can do". In class-based languages, all objects instantiated from the same class have the same methods and thus have the same behavior specification. Constructors are related to how objects are created. Class-based languages typically have an *instantiation expression* to create objects from classes, giving programmers the power to "give birth to" objects any time it is needed. For instance, in

[1] Visibility modifiers are ignored in this code fragment. It will be explained when encapsulation is introduced.

Java, the expression is **new**. When the **new** expression is evaluated, the constructor is invoked and the code inside is executed. For instance, two protein objects, `proteinx` and `proteiny`, can be created from the same `Protein`.

```
proteinx = new Protein();
proteiny = new Protein();
```

It is worth point out that some OO languages are *class-less*, sometimes called *object-based languages*. An example is the Self language. A key difference between a class-based language and a class-less language is how an object is "born". In a class-based language, an object is born through instantiation from a class, as we have just described, while in a class-less language, the program first describes a number of objects that will already be existing at bootstrapping time, and any object "born" after that is *cloned* from some previously existing object.

Classages is a class-based language. That said, many *Classages* features are orthogonal to the division of class-based languages and class-less languages, and can also be useful for designing class-less languages.

Messaging Inter-object interactions are modeled by *messaging*. The idea is that each method can be viewed as a message handler, and each object can thus be viewed as responding to the rest of the software system by responding to messages. For this reason, messaging is often called *method invocation*, a somewhat drab yet more revealing name. As an example, the following `Client` class defines a `start` method whose body involves sending a `fold` message to the `p` object:

```
class ProteinModel {
    public void start(Protein p) {    p.fold();    }
}
```

Following the semantics of most Java-like languages, sending a `fold` message to `p` amounts to the execution of the `fold` method on `p`, and then the return of the result to the sender. The process of object `p` finding the `fold` method is called a *message dispatch*, to be detailed soon. The message sender is often informally called a *client* of the receiver object.

Encapsulation Broadly, encapsulation refers to the ability to hide the internal complexities of a programming unit from the rest of the system. With encapsulation, a large program can be maintained, understood, and reasoned piece-by-piece, significantly increasing development productivity. Encapsulation is one of the central themes of programming language design, and its broad meaning aligns with modularity and information hiding, and has gone beyond OO programming. Influential language constructs in the history of programming languages – such as functions, modules, abstract data types, and software components – all reflect our need for this desirable property.

In the context of OO languages, encapsulation commonly refers to the ability to access fields and methods in a controlled manner. For instance, some OO languages disallow an object's fields to be accessed by another code other than the object's own methods. Languages like Java and C++ have a slightly more flexible notion. It is equipped with a number of *visibility modifiers*, such as **public** and **private**. This allows programmers to selectively hide/expose a number of fields and methods by declaring them to be **private**/**public**. For instance, the following Java code defines `Protein`'s `name` field can only be accessed by its own methods, while method `fold`, `setName` can be accessed by other objects via a messaging mechanism we will introduce later:

```
class Protein {
    private String name;
    ...
    public void fold() {    ... code omitted ...    }
    public void setName(String n) {    name = n;    }
}
```

Code Reuse Features: Inheritance and Mixins Earlier we have explained each class is a template for specifying a category of objects. Are classes related? For instance, it is known enzymes are a special subcategory of proteins. If our simulation program also intends to handle a number of enzymes, it will be natural to define a class called `Enzyme` as follows:

```
class Enzyme {
    String name;
    ...
    void fold() {    ... code omitted ...    }
    void setName(String n) {    name = n;    }
    void catalyze(Reaction r) {    ... code omitted ...    }
}
```

Compared with `Protein`, the `Enzyme` class has one more method `catalyze` to define some enzyme-unique behavior. Suppose `Enzyme`'s folding is no different from a general `Protein`, a downside of defining `Enzyme` from scratch as above is programmers would have to copy and paste the implementation of `fold` from `Protein` to `Enzyme`, an obvious bad smell for code maintenance. To promote code reuse, *i.e.* only keeping one copy of the `fold` implementation, most OO languages provide mechanisms to allow programmers to "piece code together".

Among these mechanisms the most widely used one is perhaps *inheritance*, so much so that many people believe it is one of the defining features of OO programming. The basic idea is that a class can "inherit" the fields and methods of another class, and define its own class on top of it rather than from scratch. For instance, Java is an inheritance-based language, and an `Enzyme` class can be defined as follows in Java:

```
class Enzyme extends Protein{
    void catalyze(Reaction r) {    code omitted    }
}
```

which defines a semantically equivalent class as our previous definition for `Enzyme` but avoids defining `folds` and `setName` methods and `name` field again. In such a situation, one commonly refers to `Enzyme` as the *subclass* and `Protein` as the *superclass*. The mechanism of inheritance is also called *subclassing*. One might ask what if the `Enzyme` class has its own distinct behaviors for `fold` and therefore needs to rewrite `fold`. This is allowed in most OO languages by the mechanism called *overriding*, so that the subclass can redefine the method. When that happens, the definition of the superclass is simply ignored [2]. For instance, Java programmers can define their own `fold` behavior by defining the `Enzyme` class this way:

[2] Strictly speaking, ignoring the superclass method is what most languages do, but not all. One exception is Beta [MMPN93].

```
class Enzyme extends Protein{
    void catalyze(Reaction r) {     code omitted     }
    void fold() {     new folding behavior here     }
}
```

If the subclass needs to invoke a method defined in the superclass, it can use what is known to be a *super call*. For instance, if an `Enzyme` object needs to invoke the `setName` method defined in the `Protein` class, it can use the expression **super**.`setName(''some name'')`.

Even though inheritance has been traditionally defined to be very close to other core OO mechanisms such as polymorphism, it is important to point out that, at least in essence, inheritance is nothing but a "code + code = bigger code" kind of code composition operation for the convenience of programmers. It has seen wide use, but has known problems such as multiple inheritance [Car88].

Inheritance is by no means the only way to achieve code composition either. One elegant alternative is mixins. A mixin is also a code template very similar to classes. The difference is how they are composed to form "bigger" code. For instance, one can define `Protein` and `Enzyme` mixins as follows:

```
mixin Protein {
    String name;
    ...
    void fold() {     ... code omitted ...     }
    void setName(String n) {    name = n;    }
}
class EnzymeAddOn {
    void catalyze(Reaction r) {     ... code omitted ...     }
}
mixin Enzyme = Protein + EnzymeAddOn
```

Here the `+` operator composes the code piece `Protein` and the code piece `EnzymeAddOn` to produce the code piece `Enzyme`. Several mixin systems exist. In some systems, `+` is defined to be slightly asymmetric, in the sense that if the two mixins have overlapping methods, the precedence of `+` will decide which one is the overriding one.

Object Types, Subtyping, and Subtype Polymorphism Programming languages are commonly equipped with a *type system*. A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute [Pie02]. A strong motivation

behind the design of a type system is to find bugs of a program without running the program itself, so that a bug in a rocket take-off control software can be found before the take-off really happens. In OO languages, objects – as "values" in the aforementioned definition – are commonly assigned with some *object types*. What is considered an object type varies from language to language, but it almost invariably includes the information on how it can be accessed by other objects. For instance, in Java-like languages, the following object type (in an informal fashion) can be assigned to object `proteinx` [3]:

```
object type Protein {
    responding to messages
        fold:  unit→ unit
        setName:  String → unit
}
```

When objects are assigned with types, a natural philosophical question is *subsumption*. Suppose we have another object type being `Enzyme`:

```
object type Enzyme {
    responding to messages
        fold:  unit→ unit
        catalyze:  Reaction → unit
        setName:  String → unit
}
```

Intuitively, any enzyme is also a protein. From a type-theoretic perspective, it is not surprising that type `Enzyme` is a "smaller" type subsumed by the "bigger" type `Protein`. In this case, we also say `Enzyme` is a subtype of `Protein`, or `Protein` is a supertype of `Enzyme`. OO languages typically have mechanisms to relate these two types, either giving programmers the explicit control to declare such a subtype-supertype relationship (*nominal subtyping*), or having the type system to infer based on their structures (*structural subtyping*).

In the OO context, an object with the subtype object type can always respond to messages supposed to be sent to an object with the supertype object type. If we have the `Client` class as defined earlier, the following code can be executed without any harm:

[3] The definition here is only an example for illustrating basic ideas, not an attempt to cover all possibilities. The representation of object types can be dramatically different for many reasons, such as choices between nominal types and structural types, or whether effects, pre-/post- conditions are expressed as type properties.

```
Client c = new Client();
Enzyme e;
c.start(e);
```

For class `Client`, even though its `start` method is expecting an object of object type `Protein`, it can also always take in objects with subtypes of `Protein`. The code `Client` thus takes on different "morphs" based on how they are used, an inherent feature in OO languages, sometimes referred to as *subtype polymorphism*.

Dynamic Dispatch and Static Dispatch Message dispatch by itself is a straightforward process: the object simply looks up on the class it is instantiated from, and invokes the method with the correct name [4]. The intricacy is in a typed language with subtyping, the method eventually being invoked may not be what the code directly suggests. Consider the example we demonstrated earlier. After `c.start(e)` is invoked, and the body of the `start` method of `client` is executed, method invocation `p.fold()` happens. If we simply look at the code, we might falsely believe it is the `fold` method of the `Protein` class is invoked. In a Java-like language however, when `p.fold()` is executed, the message dispatch mechanism will ask object `p` on *its* `fold` method, which may or may not be the same as the `fold` implementation in `Protein`, depending on which class the `Enzyme` instance is instantiated from. This mechanism is called *dynamic dispatch*. Almost all OO languages support dynamic dispatch for messaging.

In contrast, if message dispatch entirely depends on the static type information, the corresponding dispatching mechanism is called *static dispatch*. In the previous example, had the language been defined to support static dispatch, the `fold` method being looked up to will consistently be the one defined by the `Protein` class. Some languages support both dispatch mechanisms, such as C++, where different syntax is used for different ways of message dispatch.

Parametric Polymorphism Originally not considered a core feature of OO languages, parametric polymorphism grows into a mainstream OO mechanism, thanks to the popularity of C++ templates and Java generics. The idea behind parametric polymorphism is a class can be parameterized with type parameters, so that the class can be typed differently for different instantiations. For instance in Java generics, a one-space

[4] Here we disregard tangent features such as overloading, or advanced ways of message dispatch such as pattern matching-based, *etc*.

`Cell` class can be defined as follows.

```
class Cell<T>{
   T item;
   public T get() {return item; }
}
```

the `Cell` class above can thus hold data of different types depending on how their instances are instantiated. For instance, **new** `Cell(Integer)` will create an object of a `Cell` holding an `Integer` object, while **new** `Cell(Protein)` will create an object of a `Cell` holding an `Protein` object.

Parametric polymorphism is closely related to the evolution of a category of type systems built on top of OO languages: ownership type systems, which we will detail in Sec. 2.4.

Open Recursion Most OO languages allow one method body to invoke another method of the same object via a special variable called `this` (or `self` in some languages). For instance, `this.m(3)` means invoking method `m` of the same object. The special behavior of `this` is that it is *late-bound*, allowing a method defined in one class to invoke another method that is defined in some subclass. Late-bound invocations of this particular fashion are sometimes called *self-calls*.

Object-oriented languages provide open recursion, in which self-calls are dynamically dispatched, allowing subclasses to intercept self-calls from the superclass and thus depend on when it makes these calls. Open recursion is useful for many OO programming idioms such as the Template Method [GHJV95], which uses open recursion to invoke customized code provided by a subclass.

2.2 Traits

Traits [SDNB03] are a code composition mechanism for OO programs. They are in spirit similar to mixins, promoting code reuse without the need for inheritance. A trait is a parameterized set of methods; it serves as a stateless behavioral building block for classes and is the primitive unit of code reuse. With Traits, a subclass-equivalent of Java-like languages can be formed by gluing together the superclass-equivalent with a number of traits, the incremental difference in behavior with respect to their superclasses. Traits do not

employ inheritance as the composition operator. Instead, Trait composition is based on a set of composition operators that are complementary to single inheritance and result in better composition properties. The following formula can be intuitively used to describe the idea behind Traits:

Class = Superclass + State + Traits + Glue

A class formed by this formula conforms to the flattening property: the semantics of a class defined using traits is exactly the same as that of a class constructed directly from all of the non-overridden methods of the traits.

2.3 Encapsulation Policies

Encapsulation Policies [SDNW04, SBD04] aim at providing flexible encapsulation mechanisms for objects. An encapsulation policy expresses how a client can access the methods of a class. Unlike Java visibility modifiers where each modifier defines the same access for *all* clients, different encapsulation policies can be given to the same set of methods, depending on different uses from clients. Structurally, an encapsulation policy is a mapping from method signatures to potentially empty set of access attributes. An encapsulation policy P serves as a contract between the class and its client, meaning that a client accessing a class through the P may only access a method m with signature s according to the set of attributes $P(s)$. Access attributes defined in the Encapsulation Policies system include `c` (the method can be called), `r` (the method can be reimplemented) and `o` (the method can be overridden). The designer can associate an arbitrary number of encapsulation policies to a class. Each policy represents a set of encapsulation decisions that correspond to a certain usage scenario. The client can independently decide which encapsulation policy to apply and in which way the class with be used. A prototype has been implemented on top of Smalltalk.

2.4 Explicitly Parametric Ownership Type Systems

The seminal system of Flexible Alias Protection [NPV98] was built for protecting aggregate objects, so that the alias to the internal objects inside the aggregate do not leak out of the boundary of the

aggregate. Even though the goal of protecting aggregate objects had already been explored by earlier systems [Hog91, Alm97], it is the system of Flexible Alias Protection that first realized that not every object an aggregate refers to is part of its representation, an overly strong restriction imposed by earlier systems. As a result of this realization, a parametric type system was introduced to support fine-grained alias control. Since then, a large number of alias protection systems have been constructed using explicitly parameterized classes with polymorphic instantiations. We broadly categorize these approaches as Explicitly Parametric Ownership Type (EPOT) systems. The term "ownership" was perhaps first used to refer to object encapsulation in [CPN98] and the theoretical foundations of these systems were discussed in detail in [Cla01]. [PNC98] illustrated the relationship of object encapsulation as a hierarchy, the *ownership tree*. We now give a brief overview for these existing systems. The terms and syntax used in this description is most in sync with [Cla01].

EPOT systems give programmers fine-grained control over specifying ownership information for direct and indirect object types referred to by a class. For instance, a GUI dialog can be defined as follows in an EPOT system:

```
class Dialog <t, t1, t2> {
   Button<this, t2> button;
   Controller<t, t1> ctrl;
   Data<world> data;
   ...
}
```

In an EPOT system, an ownership type is formed by annotating the object type with a sequence of *context parameters*, where the first parameter denotes the owner of the object, and the rest are parameters that are propagated down the tree. Each object has a unique context, denoted by keyword `this`, which is the base case for context parameters. In the above example, field `button` has type `Button<this, t2>`, which would mean the owner (the first parameter of the list) is the current instance, *i.e.*, `Dialog`. The *owners-as-dominators* property of early EPOT systems says that objects outside of a particular owner can never access its owned objects. For instance, `Button button` here cannot be accessed by objects outside the `Dialog` instance on the ownership tree. The keyword `world` is introduced to denote the context represented by an imaginary object at the root of the ownership tree, which owns all "shared" objects. Intuitively, field `data`

with type `Data<world>` indicates that the `Data` is accessible by any object.

Researchers later realized the property of owner-as-dominators was sometimes too restrictive: for instance, the `Button button` might still need to listen to GUI events triggered by some `EventSource` outside the boundary of the `Dialog`. The common way to implement this in Java is to have the `Button` hold a `Listener` that can be called back from the `EventSource`. Programmers might want the `Button` to own the `Listener` to restrict access to it, but this would make even `EventSource` unable to access it. A few systems have been designed to address this problem [CD02, AC04, LP06], offering solutions that we refer to as *selective representation exposure*. A common thread in these papers is to make use of context parameters to specify access. For instance in [LP06], a solution to the aforementioned example problem is as follows:

```
class Button <t, e> { [e]Listener<this> l;  ...}
```

Here the `Listener` field is annotated with an *access modifier* `[e]`, meaning field `l`, although owned by the `Button`, can be accessed by anyone directly or indirectly owned by the object represented by context parameter `e`. If object `EventSource` happens to be one such `e`, it can access the `Listener`.

2.5 Universe Types

Universe Types [MPH01] (or sometimes called *Universes* in the literature) are an object encapsulation system following the "owner-as-modifier" principle: if an object o_1 is encapsulated by another object o_2, only o_2 can mutate o_1, but all other objects can access o_1 via read-only mode. Object references can be modified by three keywords as program invariants: `peer`, `rep` and `read-only` (or sometimes called `any`). If inside the scope of an object o_1, keyword `peer` is used to modify an object reference o_2, then o_2 must also be encapsulated by the object that encapsulates o_1. Keyword `rep` denotes encapsulation, where the modified object is encapsulated by the modifying object. All non-`peer` non-`rep` references are treated as `read-only`. Universe Types allow for the static consistency checking of some limited form of invariants declared by programmers. For instance, it is able to track the fact that if an object o_1 passes an `peer`

object reference o_2 to a `peer` object o_3, then the modification for the reference on o_3's side is also a `peer`. (Intuitively, it says, a `peer`'s `peer` is a `peer`). The modifier information is also recorded by the language run-time, so that a reference declared to be `read-only` can be dynamically cast to `peer` and `rep` if they happen to conform to the program invariants dynamically. Universe Types can be read as a language design effort, but the primary goal of Universe Types appears to be formal verification of object-oriented programs [DM05]. As a result, Universe Types have made design decisions that are reasonable from the formal verification point of view, but are restrictive and less expressive from the language design perspective, *e.g.* requiring all non-`peer`, non-`rep` accesses to be `read-only`. Universe Types are still an active project with recent developments; for instance, the language is extended with type genericity [DDM07].

2.6 Shape Invariant Enforcement Systems

Enforcing shape invariants for linked and recursive data structures has been explored by a number of type systems, such as Graph Types [KS93], Shape Types [FM97], and L_r [BRS99]. As an example, Graph Types are designed to enforce shape invariants on recursive data structures. For a data structure with a recursive field (say a `LinkedList` with a `next` field), Graph Types allow programmers to add a shape-enforcing modifier, called a *routing expression*, to the field, so that advanced properties of the `LinkedList` can be statically enforced, such as it being a doubly linked list, or being acyclic. As a dual to language design techniques, program analysis techniques for discovering shape invariants and checking data structure consistencies [SRW98, SRW99] have a similar goal but follow a different path: they are not designed for helping programmers express their intentions, but for inferring what programmers could have intended.

Chapter 3

Why a New Language?

This chapter aims at answering the "motivation" question from a technical perspective: why existing language models fall short, and where the *Classages* model excels.

3.1 Motivations for an Interaction-Centric Object Model

Since the dawn of OO languages, a class has been defined as a collection of fields and methods. Once this notion is fixed, the forms of interaction that can be expressed are narrowed: objects can interact by invoking each others' methods, and classes are combined together by a means such as inheritance or mixing. This is a less interaction-centric view than we pursue. In this section, we show how an interaction-centric object model can emerge from basic principles of language design. We believe the principles presented below are not sufficiently supported by mainstream OO languages.

3.1.1 On Separation of Statics and Dynamics

Principle 1. *Static and dynamic behaviors of an entity should be specified independently.*

In OO languages, a class has two roles to play: it both serves as a piece of code for reuse, and also as the "mold" to specify run-time behaviors of objects. Although a class's reuse behavior is orthogonal to its runtime behavior, existing OO models tend to treat a class's interface for reuse as closely related to its runtime

```
class Protein {
   private List sequence;
   public void add(AminoAcid o) {
      if (!sequence.contains(o)) sequence.add(o);
   }
   public void addAll(Collection c) {
      Iterator i = c.iterator();
      while (i.hasNext()) add(i.next());
   }
    ... other fields and methods ...
}
class SmallProtein extends Protein {
   private int count;
   private int threshold = 100;
   public void add(AminoAcid o) {
      count ++;
      if (count > threshold) { ... some error handling code ...}
         else super.add(o);
   }
   public void addAll(Collection c) {
      count += c.size();
      if (count > threshold) { ... some error handling code ...}
      super.addAll(c);
   }
}
```

Figure 3.1: An Instance of the Fragile Base Class Problem

interface. In mixin systems, the methods of a mixin compound that can take messages at runtime *happen to be* the methods that each participating mixin exports for static composition. Java is more advanced in this regard: its **protected** visibility modifier provides a way to distinguish between the two interfaces, because **protected** methods can only be used for static reuse, and not for runtime method invocation[1]. Overriding unfortunately muddies the waters here for Java. Overriding—a code-reuse operation—is tethered to dynamic dispatch, a runtime operation. The result of this is a mixed specification of static interactions and dynamic interactions, and a programmer's intent to reuse can conflict with their intent for runtime communication.

One particular problem arising from this conflict of intention is the Fragile Base Class problem [MS98], one of the most significant challenges faced by designers of object-oriented libraries. A common symptom of the problem is the correctness of the subclass depends on the *implementation details* of the base

[1] The modifier **protected** in Java also signifies package access, a property that is orthogonal to the discussion here.

class, violating the desirable principle of modular reasoning. One instance of the problem is illustrated by the example in Fig. 3.1. Here the `Protein` class is defined with two methods `add` and `addAll` for either adding one `AminoAcid` object or a collection of `AminoAcid` objects, respectively. A subclass of `Protein` called `SmallProtein` is defined. It is the same as `Protein` except that the amino acid sequence held by each `SmallProtein` object cannot exceed a pre-defined `threshold`. Both the `add` method and the `addAll` method is overridden to add checks to ensure the amino acid sequence does not exceed the `threshold`. If viewed separately, both `Protein` and `SmallProtein` are defined in an intuitive way.

Unfortunately, this piece of code would run incorrectly. Consider a `SmallProtein` object with a count of 94 `AminoAcid`'s in its `sequence` is sent with an `addAll` message with a `Collection` holding 5 `AminoAcid` objects. The `addAll` method of the `SmallProtein` object first correctly changed `count` to 99, and invoked **super**`.addAll(c)`. The `addAll` method in `Protein` is invoked, but note that definition depends on a self-call to `add`, which according to the semantics of open recursion, will be dispatched to the `add` method of the `SmallProtein` method. Thus, `count` will be updated again, and error is reported saying the threshold is over-passed.

How can we solve this problem? One solution would be to declare `add` in the `Protein` to be **final**, which according to the semantics of Java means the method cannot be overridden by any subclass. This indeed avoids the anomaly from happening, but in fact make programmers' life miserable: `SmallProtein` cannot be defined any longer! One may also argue that this may be a problem for Java, but for languages with support of static dispatch, programmers can easily change the invocation of `add` method in the `addAll` method of `Protein` to be statically dispatched. This solution however is also at the expense of programmers: the superclass – which in many cases are the library classes – are now making a decision for all possible subclasses. No open recursion is possible at all. What if for some subclasses of `Protein`, programmers do not intend to override `addAll`, but only intend to override the `add` method to expect the correct functioning of `addAll`? In general, "to have open recursion or not" is simply not a decision a superclass can make.

If we look deeper at the problem, the main weakness is that the subclass in Java-like languages are "powerless": when `SmallProtein` defines the `add` method, it is automatically considered *both* as a part of the run-time interface of the `SmallProtein` object to its clients, *and* as a part of the static interface to

its superclass, to make the superclass able to have open recursion semantics for self-calls. In fact, when the `SmallProtein` programmer defines the method, it is only intended to be put to the run-time interface, not the static one, but existing languages do not have a mechanism to express the intent.

3.1.2 On Interaction Bi-directionality

Principle 2. *Interaction is fundamentally bi-directional.*

The root *inter-* of the very word *interaction* already suggests a reciprocal relationship. It is important for OO programming because, even in existing languages, bi-directional interaction is also pervasive, though sometimes in a less explicit way:

- For class-class interaction via inheritance, a naive perspective would be that the superclass provides and the subclass consumes, but overriding reverses this: the subclass provides and the superclass consumes. Another example is the **abstract** modifier of Java.
- For object-object interaction, a naive perspective would be the message sender asks and the receiver answers, but callbacks, where the sender sends **this** as a parameter, exist because there is a need for the receiver to ask and the sender to answer. Other examples include the Observer design pattern, event systems, Web interactions, human-computer interaction applications (GUI, robotics), *etc*.

In the realm of relationships, bi-directional interactions suggest relationships with two-way navigability. The argument that bi-directional interaction can be modeled by two uni-directional interactions is not satisfactory to us, since the language cannot signify how the two uni-directional interactions are related, or how they depend on each other. It is about as intuitive as buying two cellphones, a "hear" cellphone from one company and a "talk" phone from another, and then using the two phones together every time you want to have a phone conversation.

3.1.3 On Interactions with Least Privilege

Principle 3. *Entities should declare the most precise interface, i.e. the interface with the least privilege, for each interaction they are involved with.*

```
interface ForAminoAcid {
    public int isCmpd();
}
class ProteinModel implements ForAminoAcid {
    public int isCmpd() { ...isCmpd code here... }
    public int init() { ...init code here... }
     ...other methods here...
}
class AminoAcid {
    public AminoAcid(ForAminoAcid o) { ... constructor code here }
     ...other methods here...
}
```

Figure 3.2: A Non-Solution to the Monolithic Encapsulation-Enforceable Interface Problem

Principle 4. *Each class should be able to have multiple interfaces for its static interactions. Each object should be able to have multiple interfaces for its dynamic interactions.*

The Principle of Least Privilege—a cornerstone of security research—also impacts encapsulation in OO languages: a class should only expose the components absolutely necessary for static interactions (such as inheritance), and an object should expose only the components absolutely necessary for dynamic interactions.

Mainstream OO languages however do not conform to this principle. Controlled by its various visibility modifiers, a class in a Java-like language can be viewed as only providing one monolithic encapsulation-enforceable interface for its subclasses, and one monolithic encapsulation-enforceable interface for its run-time clients. In Java, the first interface includes all methods and fields of the class that are declared either **public** or **protected**, while the second interface includes all methods and fields of the class that are declared **public**. Such a treatment provides *all subclasses* with the same access to a class, and *all objects* with the same access to an object. For instance in Fig. 1.1, a Java class has to declare `init` as a **public** method in the `ProteinModel` class, so that `ProteinFolding` object can access it. This decision however would not prevent an `AminoAcid` object from invoking the same method accidentally, and hence leading to design-level bugs (in this case, a violation of "an `AminoAcid` object should not initialize a `ProteinModel` object"). For this very simple example, the problem may seem to be easily detectable by having the program-

mer manually look through the code, but in large systems where a class potentially has several dozen **public** methods and its runtime instance may have hundreds of client objects, the question of which methods should be accessed by which objects is not as clear as the example here.

Readers familiar with Java might ask whether Java's **interface** mechanism solves the problem. For the previous example, one might be tempted to provide a "solution" as in Fig. 3.2. The attempt here is to define a Java **interface** named `ForAminoAcid` which only declares the methods needed by the `AminoAcid` object from the `ProteinModel` object. This **interface** is implemented by the `ProteinModel` class and does not include `init`. When class `AminoAcid` is defined, all `ProteinModel` references that are passed into the class are declared to have the `ForAminoAcid` **interface**, such as the constructor of `AminoAcid` defined above. It is the programmer's intention that the `AminoAcid` will only be able to invoke the `isCmpd` method and not `init`. This strategy however is flawed. Any implementation with an intention to invoke `init` can easily do so by using downcasting, in the form of `((ProteinModel)o).init()`. The fundamental reason behind this failure is that Java's **interface** mechanism is designed for subtyping, not for encapsulation. Historically, the mechanism excelled in freeing object subtyping from subclassing, and solving sticky problems such as multiple inheritance. With casting as a crucial feature for language with subtyping, code like `((ProteinModel)o).init()` is a necessary feature rather than a bug. Java's **interface** mechanism, even though termed "interface", is very different from what we mean by interaction interfaces which are encapsulation-enforceable.

3.1.4 On Relationship Lifespan

Principle 5. *A relationship's lifespan sits somewhere on the spectrum from permanent to long-lived to ephemeral.*

The design process of *Classages* is also a study of relationships. One important aspect of a relationship is its lifespan. We summarize support, both from the real world and from programming languages, in Table 3.1. In the real world, a *conceptual relationship* is a law, or a theorem: an invariant and unchanging relationship between concepts. For example, the concept of carbon monoxide is invariably a molecule

Support \ Lifespan	Permanent	Long-lived	Ephemeral
Real World	conceptual relationship	concrete relationship	concrete relationship
General OO	class relationship	object relationship	object relationship
UML	generalization	association	association
Java/C++	inheritance	object reference in fields	message passing

Table 3.1: The Spectrum of Relationship Lifespan

consisting of one carbon and one oxygen atom. The concrete molecule—such as a particular carbon-oxygen molecule is formed dynamically by a reaction, will be stable for some time, and will then decompose by another reaction. Other relationships will be very short, for example carbon monoxide from automobile exhaust that quickly oxidizes to form ozone. In this case the carbon-oxygen relationship was *ephemeral*.

Here we study how OO languages support relationships of different lifespans. In mainstream OO languages, conceptual relationships are classes permanently related by inheritance, and object message passing is an ephemeral relationship, starting when a method is invoked and ending when the return value is sent back. Two message sends to the same object are independent ephemeral events. Using message passing to model object-object relationships is arguably weaker than what is implied by a UML association, which is an explicit, potentially long-lived, object-object relationship. In the standard OO model, it is the *fields* that model long-term relationships, where one object can store a reference to another object in a field. This is especially true for the case of a UML aggregation or composition relationship [GAA04]. The standard model is less precisely defining a relationship, however: the communication is via messages which is a semi-public channel since many other objects – not just the objects being held in the fields – could also be sending messages to the same object. We clarify this perhaps subtle distinction with an example.

Suppose Fig. 1.1 was implemented in Java, and a model checking tool is interested in ensuring a very simple temporal constraint: every time the `ProteinModel` object refreshes the `View3D` object, the latter should call back `getTA`, to get the latest torsion angle data. An Observer design pattern will lead to a `View3D` object being implemented this way:

```
class View3D {
   private ProteinModel source;
   ...
   public refresh() {
      int x = source.getTA();
      ...
   }
}
```

Since there is no static way to ensure the `ProteinModel` object invoking `refresh` is indeed the *value* stored in `source`, the simple callback constraint we want to check will be very difficult to verify statically. The fundamental reason behind this is Java does not have a way to indicate the longstanding relationship between two objects, and thus how the `getTA` invocation here is always part of such a relationship between objects.

A lack of support for object relationship in the first place leads to a lack of support for its properties such as multiplicity and statefulness, which *Classages* will refine.

3.1.5 On Relationship Multiplicity

Principle 6. *A dynamic relationship between two sorts can be a one-to-one, one-to-many, or many-to-many relationship.*

This obvious fact of relationships does not need further justification; it is realized in UML class diagrams via multiplicity declarations.

In the example, a `ProteinModel` objectage might have multiple `View3D` view objectages for display, and a `View3D` objectage might at the same time display the shape of multiple `Sequence` objectages.

3.1.6 On Relationship Statefulness

Principle 7. *Relationships are frequently stateful in their own right.*

Long-lived relationships are rarely stateless. For instance in a data-view relationship such as that between `Sequence` and `View3D` objects, programmers might be interested in when each view was last

updated. A work-around in Java/C++ programming is to declare an object field either in `ProteinModel` or in `View3D` to store the information, but since the relationship between `ProteinModel` and `View3D` is many-to-many an array has to be used; maintaining an array in this case will require additional overhead. The fundamental problem here is that the last updated information is fundamentally part of the state of the data-view relationship itself.

3.2 Motivations for a Compositional Heap

Hierarchical decomposition is a fundamental principle for controlling software complexity. With ever larger-scale systems becoming ever more common, the importance of decomposing systems into well-structured hierarchies to manage complexity cannot be overstated. There has been a long history of programming language designs to conquer complexity in a hierarchical fashion, primarily the development of module systems. While module systems effectively form a codebase hierarchy, they are less effective at making hierarchical distinctions in the run-time heap structure. This hierarchical heap structure nonetheless pervasively exists: a `Main` object may intuitively be "composed" of a `View` object, a `Model` object and a `Controller` object via the MVC pattern, and the `View` object may be composed of several `Dialog` objects, and each `Dialog` object composed of several `Button`'s. Intuitively, this compositional structure can be viewed as a tree reference backbone "overlay" on the general heap reference graph. Giving programmers the power to explicitly define and reason about dynamic heap hierarchies helps promote program understanding: it already exists in their conception of the software, and bringing it out in the code itself will help refine and clarify the compositional structure of the heap.

Decomposition often indicates encapsulation: when a `Dialog` object contains a `Button` object, it suggests the `Button` should not be arbitrarily accessed by other `Dialog` objects. A key benefit of shaping the object heap into a hierarchy is that this notion of object encapsulation can be intuitively captured. This can be achieved by outlawing encapsulation-breaking pedigrees such as **nephew**: such a pedigree would allow a `Dialog` object to access the `Button` enclosed by a **sibling** `Dialog`, and so it is illegal. Static type systems for object encapsulation have been explored in detail in ownership type systems [NPV98, CPN98, Cla01].

The fundamental difference between our Pedigree Types and these systems is that we have a broader scope: we aim to overlay a decomposition hierarchy on the *entire runtime heap* of an application, while ownership type systems are targeted at enforcing localized hierarchies, such as protecting the internal representation of a `Sequence` object, say, a `LinkedList` object. Since ownership types excel at building hierarchies of depth 2 or 3, one might expect that deeper hierarchies could also be constructed. We will show in this dissertation how the construction of deeper hierarchies requires fundamentally different language design choices:

- *A decentralized control of hierarchy shaping*: each object needs to consider only what the hierarchy should be *from its perspective*, and the global hierarchy is built when all objects are collectively taking a consistent view.
- *A "don't care" default mode for object pedigree declarations*: when dealing with hundreds of kinds of objects, most of them should be placed just "somewhere on the hierarchy", and having this as the default mode then frees the programmer to only add declarations where they really matter.
- *An occasional need for selective exposure*: the hierarchy of decomposition is not always perfectly in line with the intention of encapsulation; when that happens, a mechanism must be present to allow access to an object from, say, its **uncle**.

Explicitly Parametric Ownership Type Systems (EPOT) systems excel at providing fine-grained internal representation protection in more localized hierarchies, such as one object protecting another object as its representation. Extending these systems to shape the entire heap, however, requires that a number of hurdles be crossed:

First, long forwarding chains of type parameters can arise in ownership type systems when the hierarchy becomes deep. Suppose we designed an application where a `Button` instance is 10 levels down on the ownership tree, and it intends to access some instance, say `GUI`, at depth 3. Thus, `GUI`'s `this` context parameter would have to be manually forwarded along the chain of levels 4-9 to finally reach the `Button`. If any class at level 4-9 has an incorrect forward, the program will not typecheck. This type of error is scattered along the entire chain and may be hard to detect.

Second, pre-knowledge of class dependencies and the ownership hierarchy is required before individual classes are written. Ownership type systems in this category build the hierarchy based on intricate forwarding of type parameters. How classes are dependent on each other affects the number of context parameters given to each class. The ordering restriction on context parameters [Cla01, CD02] further complicates the matter as programmers need to know the hierarchical structure to make sure the order of the context parameters reflecting the hierarchical enclosure relationship.

Third, long parameter lists often arise when designing library classes. Suppose ownership type annotations were to be added to the existing JDK library class `javax.swing.JButton`. This is one of the simpler Swing classes, but in Java 6, its implementation still references more than a dozen classes, including `JRootPane`, `AccessibleContext`, `ObjectOutputStream`, `Action`, *etc.*, with around 50 object type declarations. In this case, each of these object types would need to be associated with a context parameter list. As a library writer, one choice is to declare everything to be `world`, but this will severely confine the object structure ("why does an `ObjectOutStream` have to be a shared instance just because a programmer used a `JButton` somewhere?"), and so would not promote reuse. A better choice would be to annotate each object type with some context parameter list, and the context parameter list for `JButton` would be the concatenation of all these lists. Observe that classes such as `ObjectOutputStream` are all non-trivial classes themselves, and each such class may itself already have a long parameter list. If library designers thus want flexibility, *i.e.*, not to pre-confine where its direct or indirect references sit on the object ownership tree, the length of `JButton`'s context parameter list will be the sum of all context parameters of classes directly or indirectly reachable from `JButton`, a large number.

Fourth, there are problems of a brittle code interface. The context parameter list is part of the signature of a class. If a class's internal implementation is changed and now refers to more objects, new context parameters will likely be needed to specify ownership for the added object references. This change will impact not only the current class, but all classes referring to the class.

A number of object encapsulation systems have been designed without the use of explicit parametricity [Hog91, Alm97, MPH01, VB99, ACG+06]. They have the appeal of providing very simple programming interfaces to programmers, but the gain typically results from a loss of expressiveness. A compar-

ison with these systems will be given in Chapter 5.

Selective exposure has been discussed in the context of ownership type systems. In Joe [CD02], temporary exposure of an owned object can be granted to any object, provided that object does not hold the reference on the heap. This appears to be counter-intuitive in who should decide on exposure. In [BLS03], inner classes are used to access internal representations, but it requires pre-determined lexical scoping. Type-based exposure strategies appeared in [AC04] and [LP06], as we have reviewed in Sec. 2.4. We now identify a few complexities of these approaches:

- *Non-Local Access Decisions*: let us revisit the earlier example of Sec. 2.4 where the `Button` object exposes its `Listener` by declaring `[e]Listener<this>`. Observe that what the `Button` class programmer says is: "let whomever instantiates `e` decide the access policy for my listener". Essentially, to obtain partial internal representation exposure, the class becomes *powerless* with respect to all objects in `e`, which can now access its internals. If the `EventSource` object was sitting close to the root of ownership tree, while the `Button` was deep on the ownership tree, we are faced with another long forwarding chain. However, the consequence of careless forwarding is more serious here: if any object on the forward chain decided to pass `world` down rather than to forward the context parameter representing `EventSource`, all objects on the heap could obtain access to the `Listener` of `Button`. When decisions are made by parties that do not have a real stake in the process, careless mistakes can arise. Many objects on the long chain are these no-real-stake parties.

- *Nepotistic Access*: Another problem arises when `Button`'s context parameter `e` is instantiated with the owner of the `EventSource` object: all owned objects, direct or indirect, of `EventSource` now have access to the `Listener`. This style of access is inherent in designs where context parameters are used to declare access policies, because owned objects fundamentally can get the context parameters representing the owner, but there appears to be no philosophical justification for this choice of access control model and is perhaps unintuitive to programmers.

- *Powerless Owner in Presence of Reference Transferability*: Another inherent problem is once a reference to `Listener` is given out, the `Button` owner will lose control over how many times that

reference can be transferred and how many aliases will be created: the object represented by e and all its descendants can all hold an alias. This is particularly undesirable for object ownership, as the primary purpose of such a language feature is alias control.

Chapter 4

The Interaction-Oriented Object Model

In this chapter, we informally describe the details of the interaction-oriented object model of *Classages*. Discussion of Pedigree Types is deferred to Chapter 5. Formalization of the key ideas explained in this chapter will be given in Chapter 6.

4.1 The Protein Folding Example Revisited

Fig. 4.1 and Fig. 4.2 give the *Classages* code for selected classages illustrated in Fig. 1.2. Fig. 4.1 defines the `View` classage, together with its related ones, such as `Win3D`, `WinText`, `View3D`, `ViewText`. Fig. 4.2 defines the `ProteinModel` classage. The structures of these classage definitions mirror Fig. 1.2, with each interaction interface declared directly.

The Static View Every object-object interaction in Fig. 1.2 results in two interaction interfaces called *connectors* being defined, one on each classage. By comparing the `ProteinModel` class in the UML diagram and the classage in Fig. 1.2, notice that the public visible methods on `ProteinModel` in the UML diagram are split across five different connectors. This is important for improved object encapsulation: the `ProteinModel` classage can interact with different parties via different interfaces, always exposing the least and demanding the most. This aspect of our language is related to Encapsulation Policies

```
classage View {
    connector Mn {
        import void start()
        export void init() {...}
    }
    connector Dt {
        import double getTA()
        import int getKind()
        export void refresh() {...}
    }
    mixer Sub {
        import void repaint()
    }
}
classage Win3D {
    mixer Super {
        export void repaint() {...}
    }
}
classage WinText {
    mixer Super {
        export void repaint() {...}
    }
}
classage View3D = View + Win3D with Sub  >> Super
classage ViewText = View + WinText with Sub  >> Super
```

Figure 4.1: A *Classages* Code Snippet for `View` and Its Related Classages

```
classage ProteinModel {
   connector Mn {
      export void init() {...}
      export void regView(ViewI o) {...}
      export void unregView(ViewI o) {...}
   }
   connector Elem {
      import double getTA()
      import void setTA(double t)
   }
   connector Snpt {
      import double getTA(int i)
      import void setTA(double[] a)
      export boolean isCmpd() {...}
   }
   connector View {
      import void refresh()
      export double getTA(int i) = ::getTA
      export int getKind() {...}
      state Time lastUpdated
   }
   connector Ctrl {
      import void stop()
      export void update(double[] a) {
         forall (c::View) { c-> refresh();}
         Snapshot s = create Snapshot();
         Snpt p = connect s with Snpt >> Mn;
         p-> setTA(a);
         if (::allLocked()) { stop(); }
      }
      export double getTA(int i) = ::getTA
   }
   boolean allLocked() {...}
   double getTA(int i) {...}
   ...other local fields and methods ...
}
```

Figure 4.2: A *Classages* Code Snippet for `ProteinModel`

[SDNW04, SBD04].

At compile time, static composition can occur, as shown in Fig. 1.2. This diagram details how the ultimate `View3D` classage in Fig. 1.3 can have its responsibilities distributed over `View` and `Win3D`, and these can be pieced together via a *mixing* process similar to mixin composition [BC90, FKF98] to produce `View3D`. Here the `View` classage provides basic functionalities for forming the MVC pattern, and the `Win3D` classage provides the implementation details of `repaint`. The result of mixing is also a classage with mixers and connectors. Mixing happens between a pair of mixers; this is different from mixins, where there is no explicit interface of mixing, and as a result more name conflicts are likely. The mixing interaction illustrated in Fig. 1.2 is defined in *Classages* by the code snippet in Fig. 4.1. Mixers are bi-directional interfaces composed of both **import** and **export** declarations. The **import** declarations are "expectations" of methods by the other party participating in the mixing interaction, and the **export** declarations are "contributions" of the methods to the other party. Mixing is a method matching process where "contributions" meet "expectations".

Note that connectors are *late-binding interfaces*: they are only satisfied at runtime. At mixing time, the `View3D` classage takes on any connector defined by its components, as can be seen by comparing Figures 1.2 and 1.3.

The Dynamic View A connection is a form of long-lived and potentially stateful interaction between objectages, and whose legal actions, such as what methods can be invoked, are well-defined via interfaces. In Fig. 1.3, there is a live connection between the objectages `ProteinFolding` and `View3D`. At a high level, a connection represents a long-lived relationship between two objectages, and often embodies a UML association. A connection happens on a pair of connectors, another form of bi-directional interface with **import** and **export** declarations. To establish a connection between the `View` connector of the `Proteinfolding` objectage and the `Mn` connector of the `View3D` objectage, the `ProteinFolding` objectage at runtime can evaluate the following expression, where `v` is reference to objectage `View3D`:

`c` = **connect** `v` **with** `View >> Mn`;

This is again a process of "contributions" meeting "expectations", only it happens at runtime. The result of the **connect** expression, `c`, is called a *connection handle*, the first-class incarnation of the peer-to-peer communication relationship. During the connection, the `ProteinFolding` objectage can call `init` to initialize the 3D view of protein structures, via the *Classages* syntax `c->init()`. Connections can be terminated explicitly, via **disconnect** `c` expression, where `c` is a connection handle.

As another example, in Fig. 1.3 the objectage `ProteinModel` is holding multiple connections of `Snapshot` instances, indicating multiple snapshots of the protein structures have been taken and recorded. After connection, the `ProteinModel` objectage can call `getTA` to query the torsion angles of the just-created snapshot, expressed by `p -> getTA()` in *Classages*. Similarly, any `Snapshot` instance can check whether the data should be stored in compressed form by the code `isCmpd()`—the other party needs not be explicitly mentioned, it is implicit that the `ProteinModel` being interacted with is the one the `Snapshot` sits in.

A more complete example is shown in Fig. 4.2, where the `update` method in connector `Ctrl` is used to define the behavior when the `MoveCtrl` objectage finds a new "folding" that has a lower energy. The `Ctrl` connector of `ProteinModel` is going to be connected to the `MoveCtrl` objectage at runtime. When `update` is invoked, all the GUI views are updated (via the **forall** expression), and a new snapshot is stored (via the **connect** expression). By analyzing existing data, the `ProteinModel` objectage can make a decision as to whether the current "folding" is precise enough (via local method `allLocked`), and if the case, inform `MoveCtrl` to terminate (by invoking `stop`).

As in Java, each classage can also have multiple objectage instances at runtime; for example, there are two instances of `ViewText` in the Figure, indicating that the application currently has two different windows showing a textual display.

Mixing and Connecting As we have seen, two kinds of interactions play crucial roles in *Classages*: mixing and connection. The representations of these relationships in *Classages* are related to research in object relationship representation [Rum87, Kri94, DBFP95, BW05, GAA04]. At a high level, interaction interfaces

indicate a willingness to communicate, a role to play, and a relationship to participate in. In *Classages*, *all* interaction is via these interfaces; there is *no* syntax for a message send from one objectage to another independent of an established interface. This may seem like a radical departure from the norm, but we believe closing one door opens up many others.

4.2 From Principles to *Classages*

In Sec. 3.1, we described several principles for designing an interaction-centric object model. We now take a close look at how they are fulfilled by *Classages*. Along the way, we also introduce some language features left out of the previous discussions.

On Principle 1: Separation of Statics and Dynamics In *Classages*, how a classage is to be reused is completely orthogonal to how its runtime instance is to communicate with other objectages. The reuse (*i.e.* code composition) behaviors are all specified on *mixers*, while the runtime behaviors are all specified on *connectors*. Thus there is a completely natural separation of concerns. With this separation, overriding only happens on mixers, and message dispatch only on connectors. Flexibility is gained by this separation, as will be discussed in Sec. 4.5.1 below.

On Principle 2: Interaction Bi-Directionality In *Classages*, all interaction interfaces are by default defined as bi-directional, with both **export**s to indicate what it *provides* to the other party engaged in the interaction, and **import**s to indicate what it *expects* from the other to provide. Predictably, interactions (be it mixing or connecting) happen when two interacting parties match the interfaces with each other and get expectations (**import**s) satisfied by provisions (**export**s) of the other. Being "satisfied" is a type property enforced by the static typechecker. The two interaction interfaces do not need to be strict reflections of each other: for instance in Fig. 1.2, it is fine for the `Super` mixer to have extra exports other than `repaint`. In fact, for each matching **import-export** we only require that the **export** method be of a subtype of the **import** method.

With bi-directional interfaces, dependencies between classes and objects are made explicit, thus

reducing coupling between them and promoting understanding of classes in isolation. In the context of GUI design in Fig. 1.2, how a view objectage is controlled by other objectages, and how it can take commands from users and to control other objectages, are both clearly specified.

On Principle 3 and Principle 4: Least Privilege In *Classages*, programmers can define multiple mixers in a classage to adapt to different static composition needs, and multiple connectors in an objectage to support associations with different objectages. Fig. 1.2 gives one example: the `ProteinModel` classage has five connectors, `View`, `Ctrl`, `Mn`, `Elem` and `Snpt`. By exporting `init` in connector `Mn` but not others, *Classages* avoids the accidental use of this method by `MoveCtrl` and `View3D`.

Note that we are defining an encapsulation mechanism, and not a security mechanism, since connections are never refused and so access to data can always be gained by providing the proper interface. For example, in Fig. 1.3, if an objectage `BadGuy` intends to communicate with the `Protein` objectage via connector `Mn`, it is not possible to have access to method `getTA` of `ProteinModel`, and thus some accidental bugs may be avoided. On the other hand, if `BadGuy` had an interface for connecting with `Ctrl`, the `getTA` method would then be accessible, since the name `Ctrl` is public.

On Principle 5: Relationship Lifespan Mixing reflects the conceptual relationship between classages, since classages already mixed together cannot be severed at runtime, it represents a relationship with a permanent lifespan. Connection represents long-lived relationships. A connection is explicitly established via the **connect** expression, and it is alive until the connection is explicitly terminated via the **disconnect** expression. Since long-term relationships are more explicitly supported than in the standard object model, there is deeper knowledge about object relationships. We can show this by revisiting the Observer example we gave in Sec. 3.1, in *Classages*:

```
classage View3D {
    ...
    connector Dt {
        import int getTA()
        export refresh() {
            int x = getTA();
            ...
        }
    }
}
```

A connection between a `ProteinModel` objectage and a `View3D` objectage is established on the `Dt` connector on the `View3D` side. When `ProteinModel` invokes `refresh`, the `refresh()` of the connected `View3D` objectage is invoked, which leads to invocation of the `getTA` method, *always* of the original `ProteinModel`. This callback relationship is statically fixed, because there is an explicit object-object interaction defined between the `ProteinModel` and `View3D` objectages.

Classages' support for ephemeral relationships is to model them as brief versions of long-lived relationships: *Classages* reuses connectors for ephemeral relationships. To initiate an ephemeral interaction with method `m` defined in the connector `C` of objectage `o`, one can use syntactic sugar `o.m()` **at** `C`. This is de-sugared as a connection established with the `C` connector of `o`, method `m` being invoked, and then the connection terminated. The sugared syntax is intentionally similar to Java/C++ method invocation, because they both represent ephemeral relationships.

On Principle 6: Relationship Multiplicity Connectors in *Classages* are by default *generative*, *i.e.* multiple connections might be available to the same connector at the same time. For instance in Fig. 1.3, the `View` connector of the `ProteinModel` objectage is connected to three views, because the same data change might require multiple views to be notified. Likewise, the `Elem` connector is connected in with multiple `AminoAcid` instances and the `Snpt` connector with multiple `Snapshot` instances, showing the one-to-many multiplicity behind their relationships.

One important issue in this context is how an objectage can then distinguish different interactions on the same interface. Recall that each connection is given first-class status via a connection handle. Accord-

ing to *Classages* syntax, access to methods defined in connectors is via a handle to a live connection. For instance, for a `ProteinFolding` objectage to invoke the `init` method defined by the `View3D` objectage, the expression used is `c->init()`, where `c` is the connection handle representing the connection with the `View3D` objectage.

When a connector is known to interact with *only* one objectage at runtime, it is inconvenient for programmers to keep track of the connection handle explicitly. For convenience, *Classages* also allows programmers to specify a connector as a **singleton**, and then refer to an **import** or **export** method in it without using a handle, via syntax `I::m`, where `I` is the connector name and `m` is the method name.

On Principle 7: Relationship Statefulness *Classages* allows connections to hold their own state. It is achieved by allowing connectors to declare *connection-specific fields*. For example, see the `lastUpdated` **state** declarations in connector `View` of `ProteinModel` shown in Fig. 4.2. Every time a connection is established on the connector, fresh connector state is allocated. The state is private to the connection and cannot be directly accessed by other connections.

4.3 Abstract Syntax of *Classages*

Fig. 4.3 gives the abstract syntax for *Classages*. Notation $\overrightarrow{X}$ denotes a sequence of X. The syntax presented here is "middleweight" in the sense that it is "lighter" than the syntax accepted by the compiler we will describe in Chapter 7, but "heavier" than the core we will formalize in Chapter 6. The implemented compiler also accepts Java expressions as it is based on an extensible Java compiler framework [NCM03]. This choice is made intentionally as *Classages* defined as in Fig. 4.3 does not define useful features such as arrays, exceptions and rich primitive types. As *Classages* has its own object model, we might in the future choose to disable some Java features in the implementation, such as defining Java classes within a *Classages* program. For brevity, Fig. 4.3 also omits classage constructors. The formal system leaves out a number of expressions uninteresting from the perspective of formal reasoning, and also defines the syntax in a more condensed form.

program	::=	$\overrightarrow{atomic \mid compound \mid rename}$
atomic	::=	**classage** a $\{\overrightarrow{interface \mid local}\}$
compound	::=	**classage** a = a′+a″ **with** $\overrightarrow{\mathsf{k}' \texttt{ >> } \mathsf{k}''}$
	|	**classage** a = a′+a″ **with** $\overrightarrow{\mathsf{k}' \texttt{ >> } \mathsf{k}''}$ **as** k
rename	::=	**classage** a = a′ **rename** k >> k′
interface	::=	*mixer* | *connector*
local	::=	*field* | *method*
mixer	::=	**mixer** k $\{\overrightarrow{im \mid em}\}$
connector	::=	**connector** k $\{\overrightarrow{im \mid em \mid connf}\}$
	|	**singleton connector** k $\{\overrightarrow{im \mid em \mid connf}\}$
field	::=	*type* f = *exp*
method	::=	*type* m($fml_1, \ldots, fml_n$) {*exp*}
im	::=	**import** *methodSig*
em	::=	**export** *method*
connf	::=	**state** *field*
methodSig	::=	*type* m($formal_1, \ldots, formal_n$)
formal	::=	*type* x
exp	::=	() | x | *const* | **this** | **current**
	|	**create** a ($exp_1, \ldots, exp_n$)
	|	**connect** e **with** k >> k
	|	**disconnect** *exp*
	|	**forall**(x : k){*exp*}
	|	f | f :=*exp*
	|	e->f | e->f :=*exp*
	|	::m($exp_1, \ldots, exp_n$)
	|	n::m($exp_1, \ldots, exp_n$)
	|	*exp*->m($exp_1, \ldots, exp_n$)
	|	*exp*.m($exp_1, \ldots, exp_n$) **at** k
	|	*exp*; *exp*′
const		integer constant
a		classage name
k		interaction interface name
m		method name
f		field name
x		variable name
type	::=	a | k | **unit** | **int**

Figure 4.3: The "Middleweight" (Programmer-Friendly But Not Succinct) *Classages* Syntax

4.3.1 Top-Level Structures

A program in *Classages* is composed of classages. A classage has a unique name, and can be either defined from scratch, with constructors, local fields, local methods, and two kinds of interaction interfaces inside, or defined as the mixing of two classages. The resulting classage of the second case is called a *compound*. In addition to the basic grammar for mixing demonstrated in Sec. 4.1, **as** is used for combining the merged participating mixers to produce a new mixer, with a name specified by the **as** clause. The underlying process of producing the merged mixer is very similar to mixin composition. Renaming of interaction interfaces at the top level is also supported.

Within interaction interfaces, an **import** declaration specifies the type information for the method to be imported; an **export** declaration provides the method, with the standard syntax associating the method body directly with the **export** declaration. As an example, the `update` method has its body defined in the `Ctrl` connector of Fig. 4.2. For convenience, some sugared syntax is also provided. Also in Fig. 4.2

```
export double getTA(int i) = ::getTA
```

declares the exported method `getTA` has its method body bound to the local method `getTA` of the classage. It is equivalent to

```
export double getTA(int i) { return ::getTA(i); }
```

Likewise, **export** τ $\mathtt{m}(\tau_1\ \mathtt{x}_1, \ldots, \tau_k\ \mathtt{x}_k)$ $= \mathtt{n}::\mathtt{m}'$ declares the exported method `m` has its method body bound to a method `m'` declared in interaction interface named `n`, where `n` might be a mixer or a **singleton** connector. Syntactically, it is equivalent to **export** τ $\mathtt{m}(\tau_1\ \mathtt{x}_1, \ldots, \tau_k\ \mathtt{x}_k)$ $\{$**return** $\mathtt{n}::\mathtt{m}'(\mathtt{x}_1, \ldots, \mathtt{x}_n);\}$. When no ambiguity arises, programmers may use expression **export** τ $\mathtt{m}(\tau_1\ \mathtt{x}_1, \ldots, \tau_k\ \mathtt{x}_k)$ directly, meaning the `m` being exported has its body defined in either of the aforementioned cases, and there could only be exactly one definition matching the method signature.

4.3.2 Expressions

The *Classages* language has the usual forms of primitive expressions, including the unit value (), variables represented by meta-variable x, integer constants represented by meta-variable *const*, and self

Interaction \ Parameters	Primitive Data	Objectage References	Connection Handles
Mixing	OK	OK	OK
Connecting	OK	OK	No

Table 4.1: *Classages* Parameter Passing

reference **this**. Special keyword **current** refers to the handle to the connection the **current** expression is invoked on. It is only meaningful when it is defined inside the code of a connector. The expression makes sense because, to invoke any code (defined in connector export methods) there must already been a connection alive. There are several forms of method invocation, depending on where the method is defined:

- To invoke a method m defined in a mixer or a singleton connector, k::m($exp_1, \ldots, exp_n$) is used, where k is the name of the interface.
- To invoke a method m defined in a generative connector, e->m($exp_1, \ldots, exp_n$) is used, where e will eventually be evaluated to a connection handle.
- To invoke a method m defined locally, ::m($exp_1, \ldots, exp_n$) is used.
- To initiate an ephemeral interaction, expression e.m($exp_1, \ldots, exp_n$) **at** k is used. For this form of invocation, k must be a connector on objectage e that does not contain **import** and **state** declarations. This coincides with the fact that ephemeral interactions should always be stateless with no control dependencies.

Expressions f and ::f:=e are used for local field access, while expression e->f and e->f:=e are used for connection-specific field access. Objectage instantiation is achieved by an expression **create**, identical to Java's **new**. The return value of this expression is an objectage reference. We have explained other expressions in previous sections.

4.4 *Classages* Typing

Classages is a strongly typed language with explicit type declarations. There are two main type sorts: *objectage types* and *connection types*. They type the two important first-class values respectively:

objectage references and connection handles. A programmer declares a connection type by using a connector name, and the underlying connection type being used by the type system is the signature information for all invocable methods during the connection interaction. Predictably, the methods of concern include all those declared either as an **import** or an **export** on the connection interface where the connection happens, plus type information for connection-specific fields. We now focus on a few important aspects of *Classages*' type system.

4.4.1 Objectage Types and Polymorphism

As with other OO languages, *Classages* has a distinction between classage types and objectage types. Unlike classages, an objectage's runtime behavior, in view of other objectages, is solely dependent on its connectors. Thus, a programmer declares a type for an objectage reference by using a classage name, and the underlying objectage type used in the type system is the type information of all its connectors, each of which includes the connector's name and the method signatures of **import**s and **export**s declared inside. For convenience, we call the type information for each connector a *connector signature*. Thus, informally, an objectage type is a set of connector signatures.

This representation follows Java in the use of a class name to define an object type, but differs in other dimensions. Object subtyping in Java is intensionally defined by **extends** or **implements**, whereas *Classages* has a structural subtyping system at heart, and classage names themselves are not part of subtyping. The reason we take a different approach is to achieve maximum polymorphism: ontologically, we believe the essence of an objectage is how many ways it can communicate with other objectages, and what the ways are. Hence, as long as objectage $\mathcal{O}_1$ can support at least as many and compatible means of communications as $\mathcal{O}_2$, $\mathcal{O}_1$ should be allowed to appear anywhere $\mathcal{O}_2$ appears.

This intuition leads to the following design for objectage subtyping: $\mathcal{O}_1$ is a subtype of $\mathcal{O}_2$ iff $\mathcal{O}_1$ contains at least as many connector signatures as $\mathcal{O}_2$, and for any pair of connector signatures of the same name, the one contained by $\mathcal{O}_1$ (say $\mathcal{C}_1$) must be *more refined* than that of $\mathcal{O}_2$ (say $\mathcal{C}_2$). By "refined", we mean structural subtyping in support of both width subtyping and depth subtyping on connector signatures: (1) Width-wise, $\mathcal{C}_1$ may have more **export**s (covariant) and fewer **import**s (contravariant); (2) Depth-wise,

for each pair of methods **export**ed from both $\mathcal{C}_1$ and $\mathcal{C}_2$, the one in $\mathcal{C}_1$ must have a signature being the subtype of that in $\mathcal{C}_2$ (covariant); likewise, for each pair of methods **import**ed from both $\mathcal{C}_1$ and $\mathcal{C}_2$, the one in $\mathcal{C}_2$ must have a signature being a subtype of that in $\mathcal{C}_1$ (contravariant).

4.4.2 Typing Interactions

Whether a *Classages* interaction (connection, mixing) can be soundly established (even at runtime) is always checkable at compile time, by typechecking the interface matching. Intuitively, interactions between two parties are sound only when each party provides what the other party expects. In type terms, for any method declared as an **import** in one interface, the other interface must declare a method of the same name as an **export**, which at the same time has a signature being the subtype of that of the former. This can be thought of as a method specialization [AC96] happening at interaction time.

4.4.3 Typing Parameter Passing

Classages has two fundamentally different interactions, and to achieve appropriate encapsulation, different policies for parameter passing are needed for the different interactions. A table summarizing all the cases is presented in Table 4.1.

Specifically, a connection handle should not be passed across connections. For instance in Fig. 1.3 if the `ProteinModel` objectage were allowed to send connection handles to `MoveCtrl`, it might send over the one representing its connection with a `View3D` objectage. Thus, even though a `MoveCtrl` objectage does not have any connector to indicate its intention of communicating with an `View3D` objectage, a *de facto* interaction channel would be established. This would violate the principle that interactions must occur on explicit interfaces only. Note that this policy does not prevent an objectage reference from being passed over connections, and so the `ProteinModel` objectage can freely send an objectage reference of `View3D` to `MoveCtrl`: by analogy with in the real world, this is the standard "Mr. `MoveCtrl`, may I introduce Ms. `View3D`?" process, and is crucial for objectages to learn of each other. Once objectages are introduced, they need to connect with each other to have any meaningful interaction. By disallowing the passing of connection handles, we disallow masquerading: "Mr. `MoveCtrl`, can you pretend to be me and communicate with Ms.

`View3D`?”

4.5 Additional Language Features

In this section, we elucidate two technical aspects of *Classages*: overriding and message dispatch; and, interaction interfaces for compound classages.

4.5.1 Overriding and Message Dispatch

Overriding and message dispatch are closely related issues in Java-like languages. The key innovation of *Classages* in this regard is it decouples the two issues, so that programmers can separate the concern of how the code is composed (overriding) from how an objectage should behave at runtime (message dispatch), and thus achieve maximum flexibility. We now use a very simple example – a class `C` with a method `m` – to answer how a classage can be defined such that (1) `m` is overridable; (2) a message targeted to `m` is statically dispatched; (3) a message targeted to `m` is dynamically dispatched.

The various cases are shown in Fig. 4.4. The graphical notations (classages, mixers, connectors, imports, exports) we use here are the same as those used in Sec. 1.1, except those specifically given in the figure. Note that in a static composition-based language, there is no distinction between a superclass and a subclass. We here only use it in the informal sense, analogous to a similar situation like in Java. Also note that the “subclass” here is just the extension code, and needs to be composed with the “superclass” to get what is called a subclass in Java.

Overriding In *Classages*, the question of whether a method is overridable is solely a concern for mixers, *i.e.*, interfaces for code composition. Instead of introducing an “overridable” modifier for mixer methods, an overridable method in a mixer is defined by *declaring the same method both as an* **export** *and an* **import**. The **export** is the default, but overridable, implementation and the **import** indicates the interest to take in an overriding method which will take precedence over the default exported. An example is shown in Fig. 4.4 (a), where method `m` is defined as an overridable method in mixer `Inner` of classage `C`. Anywhere in the

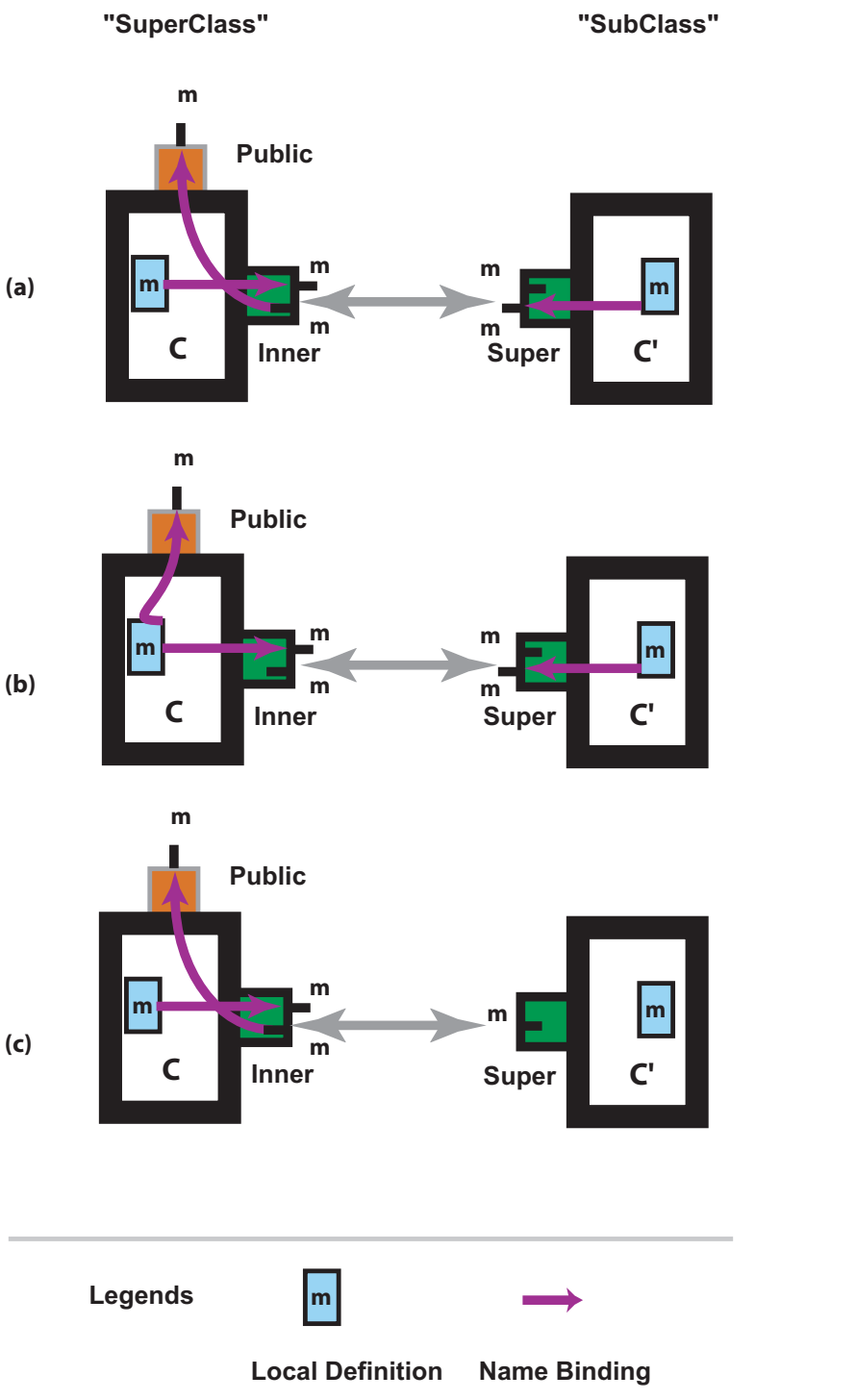


Figure 4.4: Overriding and Message Dispatch (a) static overriding and dynamic dispatch (b) static overriding and static dispatch (c) static overriding and dynamic dispatch, with no fragile base class problem

code of `C` that `m` of `Inner` is referred to, the body of `m` is (1) the **export** `m` of `Inner` if mixer `Inner` is not matched up via mixing; otherwise (2) the **export** provided by the other classage involved in the mixing interaction—in Fig. 4.4 (a), this means the **export** `m` provided by mixer `Super` of classage `C'`.

Dynamic & Static Dispatch On the other hand, dispatching messages from peer objectages is solely a concern for connectors, *i.e.*, runtime interfaces. Examples of dynamic dispatch and static dispatch for method `m` are shown in Fig. 4.4 (a) and (b) respectively. In (a), the **export** `m` in the connector `Public` is defined to be the `m` of `Inner` (referred to by expression `Inner::m`), *i.e.* the overridable method, while in (b) the **export** `m` in the connector is defined to be the local `m` (referred to by expression `::m`).

The Fragile Base Class Problem Revisited We have just mentioned two cases where *other* objectages might invoke `m`. Note that even inside the body of classage `C`, the "self-calls" follow the same pattern: programmers can still use the same pair of expressions – either `Inner::m`() or `::m`() to distinguish whether the dynamically bound method or the statically bound one should be called.

Interestingly, this obvious way of programming helps programmers avoid the Fragile Base Class Problem. Firstly, by equipping a language both with dynamic dispatch and static dispatch, the "superclass" author can have the choice of whether overriding should happen. This however is just part of the solution *Classages* provides. Indeed, any language that supports static dispatch can achieve similar effect explained earlier. A flip side of the Fragile Base Class problem is whether the "subclass" author can have a choice over whether overriding should happen. In *Classages*, even when the base class uses dynamic dispatch, the "subclass" can still use discretion to avoid accidental overriding, as is illustrated in Fig. 4.4 (c). Suppose when classage `C` is written, its author is unable to anticipate the overriding behaviors of its future "subclasses", and therefore defines the "self-calls" by using dynamic dispatch, so as not to lose expressiveness and generality. In *Classages*, the "subclass" author has a way of refusing overriding without making sacrifices over the runtime interface, by simply *not* **export**-*ing it on the* `Super` *interface*. Note that in C++, this is an impossible task.

Super and Inner Readers might wonder how a Java-style **super** call can be made. In both Fig. 4.4 (a) and (b), notice that classage `C'` on its `Super` mixer also declares method `m` as both **import** and **export**, *i.e.*,

overridable. The effect is one of *mutual overriding*, where C and C′ can use each other's m method. Hence when C′ is interested in calling method m defined in C, it can simply use Super::m.

Not only that, the essence of Beta-style [MMPN93] message dispatch is also supported. Fig. 4.4 (b) shows the case where message dispatch is always directed to C, and inside the body of the locally defined m, programmers can write Inner::m to call the one defined by C′.[1]

4.5.2 Interfaces for Compound Classages

In *Classages*, the difference between compound classages and atomic classages are only syntactical; semantically, a compound classage can always be "flattened out" and is equivalent to an atomic classage with mixers and connectors. In fact, this is also how the compiler currently implements it. With mixins, Traits, and many other existing code composition systems already designed, such a "flattening-out" process would have been fairly standard for *Classages* as well, except for one aspect: the need to handle connector composition at "flattening-out" time. Connectors are interfaces for runtime interactions and mixing interactions do not directly happen on them, but since they are associated with classages, mixing must consider how a compound classage should carry over the connectors of the classages participating in mixing.

Intuition We now consider a compound C from the mixing of A and B. Each declared connector signifies one means for its runtime instance to communicate with the outside world. Intuitively, objectages instantiated from the compound classage of A and B should be able to communicate with the outside world both by A's means and B's means. This suggests that the compound classage should have connectors both defined by A and by B.

One special case is A and B might declare connectors of the *same name*. Only two possibilities exist: accidental name clash or intentional name correspondence. For the first case programmers can just rename connectors so that name clash does not happen at mixing time. The second case is interesting because on a high level, a connector name can be viewed as the "keyword" of the "communication policies", and the overlapping of connector names signifies A and B intend to be involved in interactions of the same nature. An

[1] There is a weakness in modeling the use of **inner** inside a Beta-class at the bottom of the hierarchy from this view, because the meaning of the corresponding expression Inner::m could be wrong. But this case also needs special care in Beta.

example is when mixing happens between two classages both with a `Public` connector indicating interactions with the "general public". Such a name correspondence is usually not an accidental clash. The mixing process should allow the merging of such connectors, indicating their common interest in communicating with the "general public". Since a connector is composed of **import**s, **export**s and connection-specific fields, we need to separate our discussion into three parts to make clear what we mean by merging connectors.

First, declaring an **import** on a connector shows an expectation the classage has for the outside at runtime. Suppose now `A` has a connector `k` with **import** `m` defined, and yet in `B`'s connector `k`, `m` is not an **import**. If the resulting compound `C` were to include `m` as an **import**, it would raise `B`'s expectation for the outside world. One can imagine a **connect** `o` **with** `k >> k'` typechecked in `B` would fail in `C`, if objectage `o` in the expression happened not to provide `m`. On the other hand, if `C` were not to include `m` as an **import**, it would lower `A`'s expectation for the outside world. But remember when `A` is written, it has already assumed the expectations (**import**s) will be satisfied when connection is established. Thus it might contain code such as `h->m()` (`h` is a connection handle on connector `k`), which would fail to typecheck if `C` did not **import** `m`. This shows that `A` and `B` must always have the same **import** declaration in connector `k` to be mergeable in a type-safe way.

Second, declaring an **export** on a connector simply shows a classage's willingness to contribute to the outside world. Thus, as long as `A` and `B` do not compete to contribute the same method, the merged connector should have all **export** method from `A` and from `B`. By "compete" we mean the two methods are not distinguishable, which would confuse the party receiving the "contributions".

Third, declaring a connection-specific field simply shows an objectage holds some private data to the connection to be established on the holding connector. If `A` and `B` each hold some connection-specific field, they should both show up in the merged connector. In the case of name clash, they should be α-renamed, since such a name clash is purely accidental from the perspective of `A` and `B`.

The Classages Solution In summary, there are two conditions a mixing process must satisfy: (1) If both `A` and `B` define connectors by name `k`, the two connectors in concern must have identical **import** declarations. (2) If both `A` and `B` define a connector by name `k`, the two connectors in concern cannot **export** two methods

```
abstract class A {
    abstract public void a();
    protected void c() {...}
    private void l() {...}
}
interface Itr {
    public void b();
}
class B extends A implements Itr {
    final public void a() {...}
    public void b() {...};
    final protected void c() {...}
}
```

Figure 4.5: A Typical Java Snippet

which are not distinguishable from overloading sense.

The following describes how connectors of the two mixing-participating parties are carried over to the compound:

- If A or B (but not both) defines a connector by name k, the connector k also belongs to C.
- If both A and B defines a connector by name k and the conditions above are satisfied, C should also contain a connector k, with **import** declarations to be either of the two parties (they are the same anyway), with **export** merged, and connection-specific fields merged with free α-conversion.

How mixers are carried from the mixing-participating parties to the compound is standard: they are either matched and consumed (via **classage** C = A+B **with** $\overrightarrow{\mathsf{k} >> \mathsf{k}'}$) or matched and re-exported as a new mixer of the compound (via **classage** C = A+B **with** $\overrightarrow{\mathsf{k} >> \mathsf{k}' \textbf{ as } \mathsf{k}''}$), or unmatched and carried over to the compound.

4.6 Encoding Java in *Classages*

In this section we show how standard notions of class, object, inheritance, *etc* can be encoded in *Classages*. As an example, a Java snippet and its corresponding *Classages* implementation is shown in

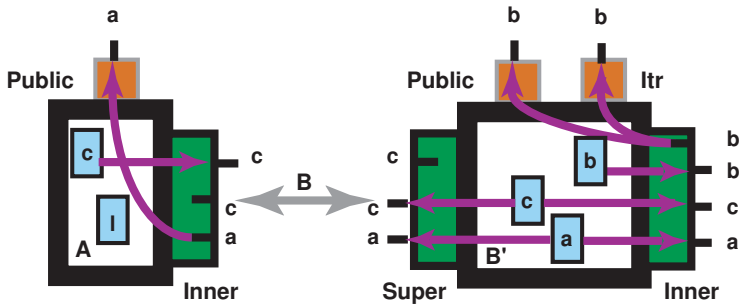


Figure 4.6: A *Classages* Illustration of Fig. 4.5

Fig. 4.5 and Fig. 4.6 respectively. For simplicity, we have left out the technicality that class `Object` is the default root class of all Java classes. The graphical notations (classages, mixers, connectors, imports, exports) we use here are the same as those used in Sec. 1.1.

The *Classages* object model can encode Java **class**, **interface**, methods with various modifiers (**public**, **protected**, **private**, **final** and **abstract**), **private** fields, overriding, dynamic dispatch, **super** invocation and subtyping. Non-**private** fields can always be modeled with a pair of getter/setter methods.

4.6.1 Intuition

Intuitively, a simple Java class without **interface** declarations can be thought of as defining three interactions with the outside world: (1) how it interacts with its superclass, (2) how it interacts with its subclass, and (3) how its instance communicates with peers at runtime. To support these three interactions, the encoding classage will use for (1) a mixer `Super`, for (2) a mixer `Inner`, and for(3) a connector `Public`. In Java, a **protected** method can be invoked inside the body of subclasses and cannot be invoked at runtime by peers; such a method in a *Classages* context will be exported to the `Inner` mixer, but not to the `Public` connector. Comparatively, a **public** method should be exported on both, and a **private** method should exported on neither. Modifier **abstract** means a method must be imported from a subclass, hence in *Classages* an **import** should be declared in `Inner` for the method. The essence of **final** is to avoid overriding, and this can be achieved by not including the method as an **import** on `Inner`. To support overriding, the actual

encoding is more complicated, in a way we explained in Sec. 4.5.1; see that section for the intuitions.

As explained in Sec. 3.1.3, Java's **interface** is not intended to strictly enforce encapsulation, but is meant to support subtype polymorphism. In that sense, as long as the subtyping relations of Java can be modeled in *Classages*, we do not need to encode this type-level language construct at all. The subtyping relation can be easily preserved in the encoding: *Classages* has a structural subtyping system, and objectage subtyping is only concerned with connectors. It can be easily seen in the encoding connector `Public` exports all **public** methods of a Java class, and it must contain all methods declared in a Java **interface** implemented by that Java class.

Java-style inheritance is encoded in *Classages* by mixing. In Java, a subclass `B` is defined based on the pre-knowledge of what superclass `A` it inherits from. In a composition-style language like ours, this pre-knowledge is not necessary, because the extension itself, *i.e.* what `B` has defined in its own body, could be a unit of reuse as well. Thus, encoding a Java-style subclass involves two steps: first define the extension using a classage, say `B′`, and then define `B` as mixing of `A` and `B′`. This is exactly what is done in Fig. 4.6.

4.6.2 Encoding

We now formally define the encoding of Java classes into classages. The encoding is defined on a linear inheritance hierarchy only, but the encoding has an unambiguous mapping onto a hierarchy tree. We start off by defining a few functions:

$\iota(\mathcal{C}, N)$	$::=$	**import**s in interface N of classage $\mathcal{C}$
$\epsilon(\mathcal{C}, N)$	$::=$	**export**s in interface N of classage $\mathcal{C}$
$\texttt{public}(C)$	$::=$	**public** methods in Java class C
$\texttt{protected}(C)$	$::=$	**protected** methods in Java class C
$\texttt{final}(C)$	$::=$	**final** methods in Java class C
$\texttt{abstract}(C)$	$::=$	**abstract** methods in Java class C
$\texttt{override}(C)$	$::=$	overriding, *i.e.* not newly defined methods in C

Given a linear Java inheritance hierarchy C_1, C_2, ..., C_n, where C_1 is the root class and C_i is a subclass of C_{i-1} for $2 \leq i \leq n$, the *Classages* encoding of C_i is a classage with a connector Public, and two mixers Inner and Super. For the root class C_1, its encoding $\mathcal{C}_1$ has the following structure, where $\overline{S}$ is the standard set negation:

$$
\begin{aligned}
\iota(\mathcal{C}_1, \mathsf{Public}) &= \emptyset \\
\epsilon(\mathcal{C}_1, \mathsf{Public}) &= \texttt{public}(C_1) \\
\iota(\mathcal{C}_1, \mathsf{Super}) &= \emptyset \\
\epsilon(\mathcal{C}_1, \mathsf{Super}) &= \emptyset \\
\iota(\mathcal{C}_1, \mathsf{Inner}) &= (\texttt{public}(C_1) \cup \texttt{protected}(C_1)) \cap \overline{\texttt{final}(C_1)} \\
\epsilon(\mathcal{C}_1, \mathsf{Inner}) &= (\texttt{public}(C_1) \cup \texttt{protected}(C_1)) \cap \overline{\texttt{abstract}(C_1)}
\end{aligned}
$$

This definition has largely been explained by the informal explanations earlier. Note that when both **import** and **export** of an interaction interface are empty, the interface does not exist: in our context, the root class does not have a Super mixer.

For any class C_i on the hierarchy where $2 \leq i \leq n$, we first compute a classage Δ_i, and then obtain $\mathcal{C}_i$ (the encoding classage for C_i) by mixing $\mathcal{C}_{i-1}$ and Δ_i, with the `Inner` mixer of the former matching the `Super` mixer of the latter. Δ_i is defined as follows:

$$
\begin{aligned}
\iota(\Delta_i, \mathsf{Public}) &= \emptyset \\
\epsilon(\Delta_i, \mathsf{Public}) &= \texttt{public}(C_i) \cap \overline{\texttt{override}(C_i)} \\
\iota(\Delta_i, \mathsf{Super}) &= \epsilon(\Delta_{i-1}, \mathsf{Inner}) \\
\epsilon(\Delta_i, \mathsf{Super}) &= \iota(\Delta_{i-1}, \mathsf{Inner}) \\
\iota(\Delta_i, \mathsf{Inner}) &= (\texttt{public}(C_i) \cup \texttt{protected}(C_i) \cup \epsilon(\Delta_i, \mathsf{Super})) \cap \overline{\texttt{final}(C_i)} \\
\epsilon(\Delta_i, \mathsf{Inner}) &= (\texttt{public}(C_1) \cup \texttt{protected}(C_1) \cup \iota(\Delta_i, \mathsf{Super})) \cap \overline{\texttt{abstract}(C_1)}
\end{aligned}
$$

This definition is inductive because Java's inheritance does not just allow a direct subclass to use or override a method defined by the superclass, it also allows use and overriding by any *indirect* subclass on the inheritance hierarchy. In a *Classages* context, this means a superclass method, no matter being used/overridden by the direct subclass or not, must be forwarded from the `Super` mixer to `Inner` mixer, since an indirect subclass might still use/override it. The presentation here is simply a mechanical encoding to demonstrate expressiveness; in reality, when one programs in *Classages*, one programs in a *Classages* way: Java's coding pattern is also optimized for inheritance. Programming in a language featuring code composition, programmers should not think of constructing deep hierarchies in the first place. For instance, by coding a labeled color point, a typical Java programmer will create an inheritance hierarchy: `Point`, `ColorPoint`, `LabeledColorPoint`, where as a *Classages* implementation would naturally be the composition of three classages: `Point`, `Color` and `Label`.

Java's method invocation is encoded by ephemeral invocations on default connector Public. Other expressions, such as new and field access can be encoded via the direct analogues in classages.

4.7 Programming in *Classages*

This section explores a few practical issues of *Classages* programming. The goal here is to illustrate how the language performs in not-so-obvious or seemingly-not-supported programming scenarios. It also aims to give readers a better understanding of the language expressiveness and potential limitations.

Programming Collection Classes We now explain how a Java programmer's need to define and use collection classes/interfaces such as `List` and `Set` is met in *Classages*. We first identify a few problems with Java's solution. The relationship between `ProteinModel` and `AminoAcid` in Fig. 1.1 will typically be implemented in Java by having a `ProteinModel` object holding a collection object, say `LinkedList`, which subsequently holds a number of `AminoAcid` objects. To refer to each `AminoAcid` object, some `Iterator` object also needs to be created. Such a solution demonstrates an impedance mismatch between design and implementation, in the sense that a simple one-to-many relationship between `ProteinModel` and `AminoAcid` demonstrated by UML suddenly becomes the interactions among four classes: `ProteinModel`, `AminoAcid`, `LinkedList` and `Iterator`. Such a discrepancy not only introduces problems in understanding, but also leads to subtle issues of alias protection [AC04].

Classages directly supports connectors with multiplicity. Thus, a one-to-many relationship between `ProteinModel` and `AminoAcid` in UML is faithfully implemented by only two classages. Functionalities represented by Java's `List` are directly supported: Java `List`'s `add` can be analogously thought of as the **connect** expression, Java's `remove` on lists as the **disconnect** expression, and iterations as the **forall** expression.

This solution covers the cases where programmers need relationship multiplicity but do not care what specific data structure is used to achieve it. However, there are situations where programmers need to pick a specific data structure, for instance, choosing a `HashMap` over a `LinkedList` to achieve efficiency.

Classages also support the definition and use of classages analogous to Java's `HashMap`: the analogous classage `HashMappage` would have some connector(s) to define how the hash map is maintained, including element addition, deletion and lookup. A second generative connector of `HashMappage` can support iteration over the map; there is no need to create a distinct `Iterator` object. A client objectage can communicate with a `HashMappage` objectage by establishing connections.

Self Mixing/Connecting Classages can be mixed with themselves, as long as they have matchable mixers. For instance, given a classage `A` with two mixers `In` and `Out`, where the **import** and **export** of these two mixers complement each other, it is possible to create a compound `AA` as follows:

classage `AA` $=$ `A` $+$ `A` **with** `In >> Out`

The resulting classage `AA` will still have two mixers, the `Out` mixer carried over from the first `A`, and the `In` mixer carried over from the second `A`.

Two objectages generated from the same classage can be obviously connected together, analogously with how two Java objects of the same class can communicate.

Reusable and Composable Connectors In a realistic language where library support is the norm, it is desirable to predefine a few commonly used connectors as reusable building blocks in the library, and users can compose their own connectors out of them. Although first-class connectors are not supported directly, they can easily be encoded in *Classages*. For reusability, consider the following scenario Java programmers often encounter: an object `MyButton` should implement a `MouseListener` interface to communicate with the event source, inside which callback methods such as `mousePressed` are defined. In *Classages*, `MouseListener` is naturally coded as a connector of the `MyButton` objectage. The question is then how a *Classages* library can define a reusable `MouseListener` connector so that the `MyButton` programmer does not have to define default behaviors of `MouseListener` from scratch. A reusable `MouseListener` can be encoded by defining a classage, say `MouseListenerage`, with one connector `MouseListener`, and one mixer, say, `MouseListenerMix`, where the mixer defines default behavior which can be overridden by mixing with other classages and the connector is defined as in Sec. 4.5.1. Classage `MyButton`

can thus be formed as the compound of `MouseListenerage` and a classage implementing the button functionalities. Such an encoding of reusable connectors also facilitates connector composibility.

4.8 Technical Related Work

Mixin systems [BC90, FKF98] address the issue of class composition as a replacement of inheritance. Classage mixing is in a similar vein, but *Classages* separates the interfaces for static behaviors from interfaces for dynamic behaviors, and each mixin only has one interface playing dual roles. In this regard, Traits are also designed with a separation of concern in mind: *traits* are only used for code composition, and are not instantiatable. Instantiation can only happen to a second language construct, *classes*, to glue *traits* together. *Classages* can use one language construct to model what both *traits* and *classes* are modeling, and still preserve separation of concerns, by declaring different kinds of interfaces on a single classage.

Encapsulation has always been a goal of OO languages. This can be traced back to the design of abstract data types. Encapsulation Policies (EP) allow a class to define multiple EPs, specifying how it can be reused or its instances can be communicated at runtime. *Classages* can also allow a class to define multiple policies for its encapsulation, by declaring multiple mixers and connectors. If the *Classages* principles in Sec. 3.1 are used to evaluate EP, it has a focus on the principle of least privilege (PRINCIPLE 3), and a separation between statics and dynamics (PRINCIPLE 1). EP does not single out the case that one object might encapsulate another object as its part. All objects are peers. Since the project is less focused on the interaction aspect of OO languages, encapsulation policies are not bi-directional.

JACQUES [CF05] is a model proposed to support so-called environmental acquisition. It aims to model UML's aggregation associations, loose whole-part relationships where the part might outlive the whole, and JACQUES focuses on how the part can import fields and methods from the whole, the environmental acquisition. Our connectors can be used to model all of the whole-part relationships of JACQUES. JACQUES's acquires declaration can be modeled by declaring **import**s in the part connector, and JACQUES's contains declaration can be analogously thought of as the connector on the whole. JACQUES demands that each class declare what classes are allowed to be a class's whole (using contained). This intensional approach is not

taken in *Classages*: any two objectages with matched connectors can form the relationship.

UML allows designers to declare *association classes* to represent the relationship between two classes. At programming language level, Rumbaugh [Rum87] first pointed out the importance of supporting first-class relationships, but mainstream OO languages' support for relationships is limited. Recently, as a reverse engineering effort, a tool [GAA04] was proposed to recover binary relationships from Java. A few projects, including complex associations [Kri94], first-class dependencies [DBFP95], and most recently, RelJ [BW05] have more explicit support for relationships than ours: relationships in these projects are not just first-class values as in *Classages*, they can also be independently defined by programmers as a top-level construct. Comparatively, our approach is more lightweight, but still covers the rich aspects of relationships as these projects cover.

Classages has syntax for directly support concepts embodied in several of the design patterns. Declaring explicit interfaces for classages is related to **Facade**. The design of interfaces with both imports and exports is a key component of **Observer**, allowing maximum de-coupling of interacting parties. By declaring **import**s on mixing interfaces, methods can be defined by other classages, which corresponds with **Template Method**.

For code reuse, *Classages* uses composition in place of inheritance. Although we have shown inheritance is encodable in *Classages* from a language perspective, it is indeed debatable whether composition can totally replace inheritance from the software engineering perspective. This perhaps can only be answered by the mass of programmers, but some experiments, *e.g.* refactoring Smalltalk's collection hierarchy using Traits [BSD03], indeed have shown promising signs for composition. Another question that needs practical *Classages* programming experience to decide is how much all the interfaces on classages get in the way compared to how much they help.

Chapter 5

Pedigree Types

In this chapter, we informally introduce the key innovations behind Pedigree Types. Pedigree Types are a general static type system which can be built on top of any programming language with arbitrary heap reference structures; in this dissertation we build pedigree types on top of our *Classages* object model. To show how the fundamental concepts also apply to more standard object-oriented languages, we start this section by explaining the key ideas behind pedigree types using Java-like syntax.

Fig. 5.1 illustrates a `Dialog` object with two `Button`'s. The `Dialog` logs important actions in two `Logger` objects, one of which (`publog`) is shared with the rest of the application, logging important actions such as when an `OK` button is pressed, while the other (`privlog`) is only used to log actions with limited impact during the lifecycle of the `Dialog`. A general `Controller` object controls the behaviors of the GUI objects. Fig. 5.2 is the UML object diagram for a possible snapshot of the program execution. Labels on object associations are pedigree relationships. Only object associations pertinent to the discussion are shown.

5.1 Hierarchy Shaping with Pedigree Types

The General Form A pedigree type is a regular object type prefixed with a pedigree qualifier, specifying where on the hierarchy an object should sit relative to the current instance. For example, in Fig. 5.1, pedigree

```
class Main {
   void main () {
      EventSource es = new sibling EventSource();
      Controller ctrl = new child Controller();
      Dialog d = new child Dialog(es, ctrl);
      d.init();
      ...
   }
}
class Dialog {
   EventSource es;
   Controller ctrl;
   sibling Logger publog;
   child Logger privlog;
   Dialog(EventSource es, Controller ctrl)
      { this.es = es; this.ctrl = ctrl; }
   void init() {
      Button ok = new child Button("OK", es, ctrl);
      Button cncl = new child Button("NO", es, ctrl);
      publog = new Logger("Shared Logger");
      privlog = new Logger("Private Logger");
      ok.init(publog, this);
      cncl.init(privlog, this);
   }
   void refresh() { ...}
}
class Button {
   Logger logger;
   EventSource es;
   String name;
   Controller ctrl;
   parent Dialog container;
   Button(String name, EventSource es, Controller ctrl)
      { this.name = name; this.es = es; this.ctrl = ctrl;}
   void init(Logger logger, Dialog container)
      {this.logger = logger; this.container = container;}
   void log() { ...}
   void refresh() { ...container.refresh(); }
   void oops() { // ctrl = new child Controller(); }
   void oops2() { // ctrl.oops3(logger); }
}
class Logger { ...}
class EventSource { ...}
class Controller {
   void oops3(Logger log) { ...}
   ...
}
```

Figure 5.1: An Example in Java-like Syntax

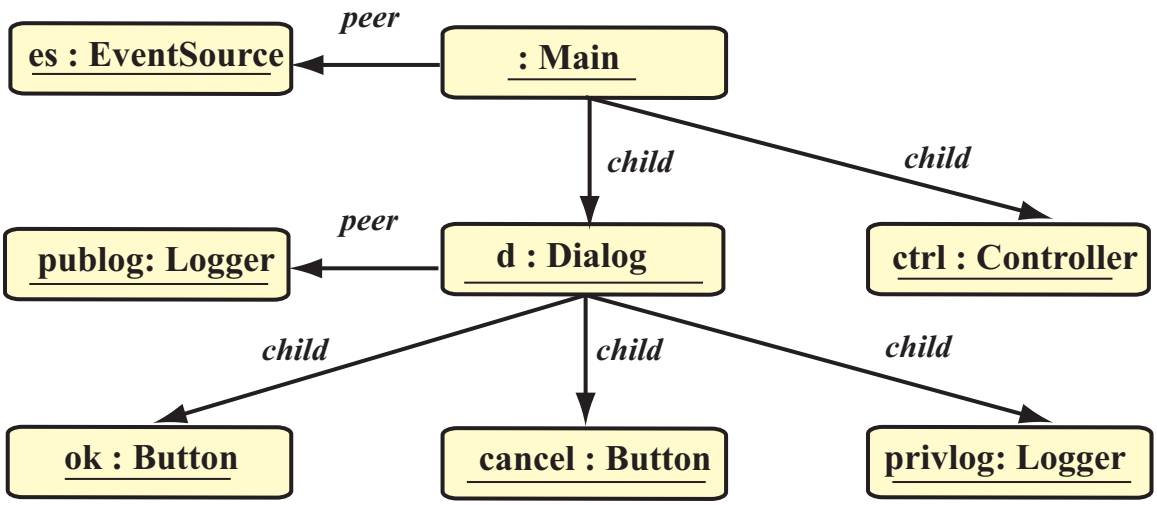


Figure 5.2: The UML Object Diagram for Fig. 5.1

type **sibling** `Logger` means any object stored in field `publog` must be of type `Logger`, and also be a **sibling** of the current `Dialog` instance on the hierarchy; similarly, pedigree type **child** `Logger` means any object stored in field `privlog` must be of object type `Logger` as well, but it must be a **child** of the current `Dialog` instance. When no confusion arises, we informally also say that field `publog` (or a variable) has a pedigree (or pedigree type) **sibling**.

In fact, qualifiers like **sibling** and **child** are just sugared syntax for special cases of a more general form:

$$\overbrace{(\textbf{parent})\ldots(\textbf{parent})}^{\mathtt{w}}\overbrace{(\textbf{child})\ldots(\textbf{child})}^{\mathtt{z}}$$

where $\mathtt{w} \in \{0, 1, 2, \ldots\}$ and $\mathtt{z} \in \{0, 1\}$. The invariant such a qualifier enforces is that, on the hierarchy the qualified object must be the $\overbrace{\textbf{parent}\text{'s }\textbf{parent}\text{'s}\ldots\textbf{parent}\text{'s}}^{\mathtt{w}}\overbrace{\textbf{child}\text{'s}\ldots\textbf{child}}^{\mathtt{z}}$ of the current object. We call $\mathtt{w}$ the *positive level* of the object being qualified and $\mathtt{z}$ the *negative level* of the object being qualified. We use the abbreviation $(\textbf{parent})^{\mathtt{w}}(\textbf{child})^{\mathtt{z}}$ for the above definition. When the negative level is zero, we can abbreviate the pedigree as simply $(\textbf{parent})^{\mathtt{w}}$.

Why This General Form? The general form elegantly covers the more familiar cases via the following sugar:

$$\textbf{child} \stackrel{\text{def}}{=} (\textbf{parent})^0(\textbf{child})^1$$

$$\textbf{self} \stackrel{\text{def}}{=} (\textbf{parent})^0(\textbf{child})^0$$

$$\textbf{sibling} \stackrel{\text{def}}{=} (\textbf{parent})^1(\textbf{child})^1$$

$$\textbf{parent} \stackrel{\text{def}}{=} (\textbf{parent})^1(\textbf{child})^0$$

$$\textbf{grandparent} \stackrel{\text{def}}{=} (\textbf{parent})^2(\textbf{child})^0$$

$$\textbf{aunt} = \textbf{uncle} \stackrel{\text{def}}{=} (\textbf{parent})^2(\textbf{child})^1$$

Declaring an object to have a **child** pedigree aligns with programmers' intention that the object is encapsulated. We will elaborate on this case in Sec. 5.2.

Pedigree **parent** provides a strong enforcement of nesting. For instance, the `container` field of the `Button` object is declared to have pedigree **parent**. Thus, if at any time the `Button` object's `refresh` method is invoked, which in turn needs to refresh its container `Dialog` object (a common practice in GUI programming), our type system can guarantee that the refreshed `Dialog` is indeed the one containing the `Button` object itself. Observe that a type with **parent** pedigree is a singleton type, since each object can only have one. No existing ownership type systems that we know of support singleton pedigrees. For instance, in a typical parametric ownership type system, the best a `Button` object can specify is to declare its `container` field with an ownership type whose first context parameter is the owner of the enclosing `Dialog`, which more or less is equivalent to our notion of **uncle**. All pedigree types with negative level 0 are singleton types, such as **grandparent**.

Pedigree **self** provides a precise way to type self references (Java's **this**). This qualifier is also a singleton type qualifier. The importance of typing **this** in a more precise way lies in the consequences of its propagation: when an object passes its **this** to others – a common programming idiom – the receiver side has the opportunity to precisely type the argument with singleton type qualifiers as well. For instance in the example, the `Button` object can have its `container` field hold a value of **parent** singleton pedigree: it is passed by `Dialog`'s method `init`, where **this** is passed. We now explain why the **parent** pedigree in the `Button` object matches the **self** pedigree in the `Dialog` object. Self-references are generally not typed so

precisely in object-oriented languages. Note especially how the **self** pedigree is fundamentally different from the `this` context parameter in EPOT systems: that parameter approximately captures the scenarios when programmers in our language would use the **child** pedigree to declare ownership.

Property of Pedigree Relativization A key property of pedigrees is that they can be relativized from the perspective of one object to that of another. Let us revisit the example we brought up to explain **self**. The `Button` object's `init` method expects an object of **parent** as its third argument `dialog`, while the object being passed in is **this**, which is of pedigree **self**. These qualifiers are not the same, but pedigree types are always declared relative to the current instance, and the `Button` object is a **child** of the `Dialog` object. From the `Dialog` object's perspective, it easily knows the **parent** from its **child**'s perspective is a **self** from its own perspective. The line `ok.init(publog,this)` thus typechecks.

Pedigree Subsumption Intuitively, oneself is a special case of his/her parent's children, *i.e.* a pedigree $(\textbf{parent})^0(\textbf{child})^0$ is a special case of $(\textbf{parent})^1(\textbf{child})^1$. Similarly, a parent is a special case of a grandparent's children, *i.e.* a pedigree $(\textbf{parent})^1(\textbf{child})^0$ is a special case of the $(\textbf{parent})^2(\textbf{child})^1$ pedigree.

(Strictly speaking, the keyword **sibling** in our language is not precise enough to cover what "a parent's child" means in the human world. The latter in fact means the combination of "being a peer" and "being self" in the human world. Likewise, "a parent's parent's child" in the human world in fact means the combination of "being an uncle" and "being a parent". This subtle distinction however does not cause too much confusion in programming, as attested by Universes, whose notation `peer` obviously also can be used to type self reference.)

In the general case, a pedigree $(\textbf{parent})^w(\textbf{child})^0$ represents a singleton set of objects which is a subset of those objects represented by the $(\textbf{parent})^{w+1}(\textbf{child})^1$ pedigree. This genealogical fact is captured by the subsumption relation on pedigrees.

Inter-procedural Pedigree Inference In our calculus, programmers only need to declare a pedigree qualifier on an object type when they care about the pedigree of that object. For all other occurrences with no qualifications, our type system can infer them. The inference algorithm is also able to track pedigree infor-

mation inter-procedurally. Consider the `oops` method at the end of the `Button` class, which if included would be a type error. If we only look at the code of `Button` itself, the code is perfectly legal. However, note that the field `ctrl` of the `Button` object is set in the constructor to be a `Controller` object held by the `Dialog` object, which is in turn held by `Main`. That `Controller` object is a **child** of the `Main` object, so its pedigree from the perspective of `Button` is definitely not **child** – the `Controller` is its **uncle**.

Our type system is constraint-based. The novel aspect is that constraints are on (positive and negative) levels, which range over (a subset of) the natural numbers. Thus, finding whether conflict exists in the constraint set is reduced to solving a system of linear diophantine equations over natural numbers, a well-studied problem [Sch98, CD94].

Polymorphic Pedigrees and Parametric Polymorphism Inferred pedigrees are treated polymorphically, so that the references in a class can have different pedigrees for different instances of the same class, aligning with the "I don't care" intention of programmers. For instance, the `ok Button` and the `cncl Button` can in fact have loggers of different pedigrees. We sometimes call object types without explicit pedigree qualification *polymorphic pedigree types*. The parametric polymorphism used here does not lie far from well-known type theoretic principles, where each class is viewed as being defined via a polymorphic `let` *a la* ML (Hindley-Milner) and polymorphic type variables are assigned for levels. We also support cyclic class definitions – the norm of object-oriented programming – which is not supported with pure `let`-polymorphism.

Different objects of the same class can obviously have different pedigrees: one `Logger` is instantiated as the **sibling** of the `Dialog`, and the other as the **child** of the `Dialog`. Objects can also be instantiated with an unspecified pedigree, as is the case for the instantiation of the two `Logger` instances, expressing "I don't care what pedigree it is instantiated with". In that case, it is the object usage which decides whether there is a satisfiable pedigree.

5.2 Alias Protection with Pedigree Types

The general form of pedigree types is consistent with the requirement of encapsulation. Intuitively, an object should only refer to its direct children ($\mathsf{w} = 0, \mathsf{z} = 1$), itself ($\mathsf{w} = 0, \mathsf{z} = 0$), direct and indirect ancestors ($\mathsf{w} > 0, \mathsf{z} = 0$) or direct children of its direct and indirect ancestors ($\mathsf{w} > 0, \mathsf{z} = 1$). For some concrete examples, it says **sibling**s can refer to each other freely; an object can always refer to its ancestors, captured by the notion of all cases where $\mathsf{w} > 1$, but not *vice versa*: **nephew** $= (\textbf{parent})^1(\textbf{child})^2$ violates the notion of encapsulation as it means to refer to the **child** of your **sibling** on the hierarchy of encapsulation, *i.e.* the internals of your **sibling**. What is referable by pedigree types is identical to what is commonly believed to be referable on the ownership tree [NPV98, CPN98].

Protecting Encapsulated Objects Against Indirect Leakage At first glance, protecting encapsulated objects is as simple as disallowing references with a **child** pedigree qualifier from being given out to any object with non-**child** pedigree. However, such a type system would not prevent *indirect leakage*. For instance, consider the pedigree relationship illustrated in Fig. 5.2. First of all, there is nothing wrong with the `Dialog` object passing its **child** object named `privlog` to its **child** object named `cncl`, as is found in the example when the constructor of the `cncl Button` is invoked. If our type system only checked whether **child** references were passed out, it would be happy to allow the `cncl Button` object to pass the reference to the `privlog Logger` object – not a **child** but a **sibling** from the perspective of the `cncl` object – to the `Controller` object (the one in the `ctrl` field). This however would violate encapsulation, as the `privlog` object is an internal representation of the `Dialog` object and should not be exposed to a **sibling** of the `Dialog` object.

Our type system is able to detect this indirect leakage. The key is that it always makes sure that both the sender and the receiver only handle references that can be associated with well-formed pedigree types. Had the passing of the private `Logger` object held by the `cncl Button` to the `Controller` object been allowed, relativization would imply the `Logger` object is the `Controller` object's **sibling**'s **child**, *i.e.* $(\textbf{parent})(\textbf{child})^2$. This is not a well-formed pedigree qualifier and would result in a type error.

```
classage Button {
   Logger logger;
   EventSource es;
   String name;
   Controller ctrl;
   parent Dialog container;
   Button(String name, EventSource es, Controller ctrl)
      { name:=name; es:=es; ctrl:=ctrl; }
   connector Lifecycle {
      export void init(Logger logger, Dialog container)
         { logger:=logger; container:=container; }
      export void log() { ...}
   }
   connector Listener {
      export void actionPerformed(Action a) { ...}
      // event handler code above
      import void startup();
   }
   void refresh() { ... }
   void startListen() {
      h1 = connect es with Listener >> MousePressed; // (*)
      h1-> startup();
      ...
   }
}
classage EventSource {
   connector MousePressed {
      import void actionPerformed(Action a);
      export void startup() { ...} //bootstrapping
   }
   void MousePressedNotify(Action a) {
      forall(h:MousePressed) { h-> actionPerformed(a);}
   }
   ...
}
```

Figure 5.3: `Button` and `EventSource` in *Classages* Syntax

5.3 Selective Exposure

The need for selective exposure can be explained via the `Button` and `EventSource` example earlier given in Fig. 5.1. The `Button` instance, say the `ok`, is currently encapsulated by the `Dialog`. To respond to any system event, it must allow the `EventSource` object to *call back* to its event handler. The standard Java approach to handle events of this flavor is to register the `Button` object reference in the `EventSource` object. However, this strategy would lead to alias protection violation when `Button` is an encapsulated object: the `EventSource` object is likely to be higher on the object hierarchy, and it should not directly access any "descendant" of the `Dialog` object, *e.g.* the `ok` `Button` object. Selective exposure is a significant threat to alias protection systems, since object hierarchies are often not perfectly in sync with the hierarchy for alias protection: even though the vast majority of `Button` references should be protected from access by random objects, there is one exceptional case for the `EventSource` object. A few systems today [CD02, AC04, LP06] offer a solution, but they have weaknesses that was reviewed in Sec. 3.2. The selective exposure problem can be elegantly solved in the *Classages* object model without any exposure-specific programmer annotation.

The `Button` class we presented in Fig. 5.1 can be partially rewritten as Fig. 5.3 in *Classages* code. All discussions of pedigree types in Sec. 5.1 also apply to the *Classages* model. For instance, the **parent** pedigree in Fig. 5.3 can be enforced as a nesting constraint in the same way as discussed previously.

Connection as the Exposure Mechanism A pleasant consequence of adopting the *Classages* model for hierarchical alias protection is that selective exposure is supported for free. This is illustrated in Fig. 5.4, where the `Button` initiates a connection with the `EventSource` objectage `es` via

connect `es` **with** `Listener >> MousePressed`

We argue that this is a good selective exposure mechanism because `Button` is *pro-active* in the process: the connection of concern can *only* be established by the `Button`, *not* the `EventSource`. Recall that to establish a connection in *Classages*, the initiating party must first have the objectage reference it prefers to interact with. Since `Button` is encapsulated by `Dialog`, there is no way for `EventSource` to receive

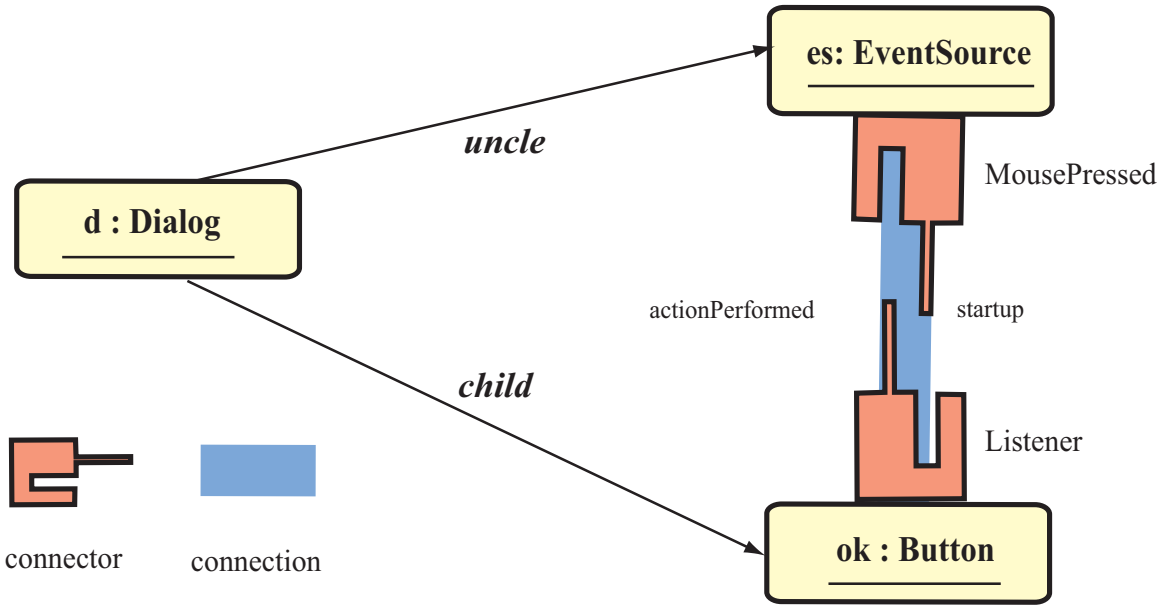


Figure 5.4: Selective Exposure of `Button` `ok` (Unrelated Connectors/Connections Omitted)

a reference to `Button`, let alone establish a connection with it. We believe this property demonstrates the essence of selective exposure: the "resource holder" should have control over the exposure, not the client who expects to receive it.

Also observe the connector definition makes the exposure precise up to the intention of the exposure initiator. The `Button` objectage explicitly declares the methods it intends to expose as **export**s in the `Listener` connector. A connector such as `Listener` provides an *exposure policy*. In the example above, the `EventSource` can only have access to its `actionPerformed` method, and nothing else. For different clients, `Button` might connect different connectors of its own to them, so that different exposure policies can apply. This aspect is related to the encapsulation policies of Traits, though the latter does not consider alias protection. In case an objectage feels uncomfortable with exposing anything at all, it always has the choice of declaring no **export**s. Selective exposure policies are not supported in existing selective exposure proposals [CD02, AC04, LP06], where the receiver of the exposure can access all methods of the exposed object.

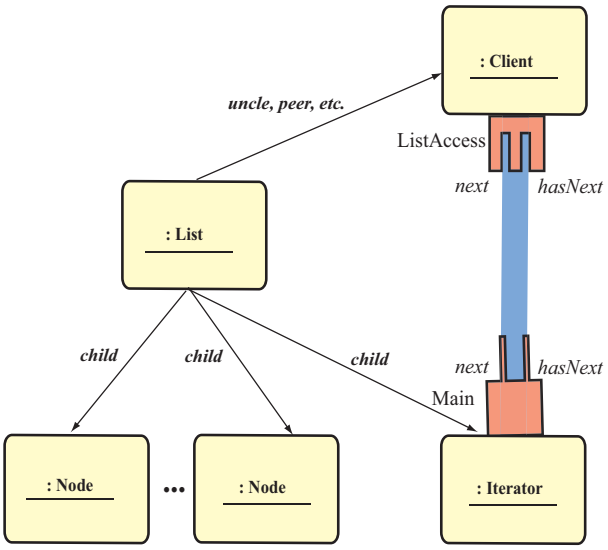
Another nice feature of this strategy is that no objectage aliases to the `Button` need be given out in the exposure process, and hence the exposure is point-wise precise to whom it exposes: `EventSource` will be able to access `actionPerformed`, and it is impossible for `EventSource` to secretly delegate its `Button` access to some third objectage. The fact that exposure happens without giving out objectage aliases is also the fundamental reason why a Java-like model, even with advanced features such as inner classes, would still fall short in imitating what we have here. A question might be raised on whether connection handles can leak, so that the good properties we just described are undermined. This is not possible in *Classages*: connection handles are local and cannot escape.

A Philosophical Reflection on Power Distribution We feel the key question of selective representation exposure is not how to do it, but *who should have the power to do it*.

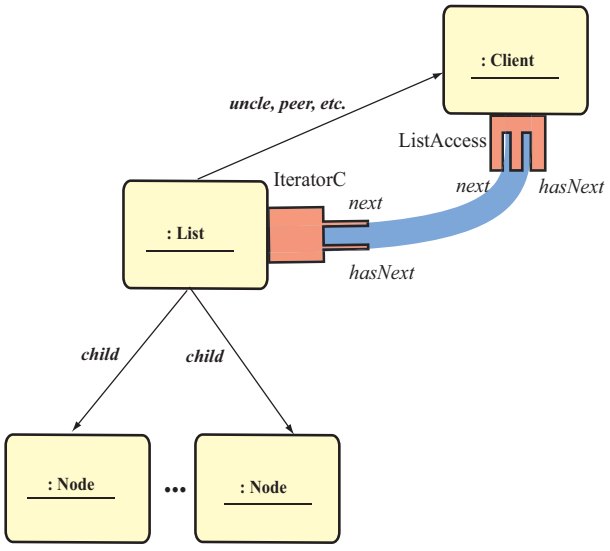
We believe in localized power distribution. When access decisions are scattered all over the ownership hierarchy, portions that do not have a big stake in the outcome of other portions can lead to errors by negligence. A human world analogy would be the case where a student receives his/her grade based on the decisions of the instructor and TAs who have a stake, not by the dean or the university president, who are more "powerful" but have less involvement. For software systems, non-local distribution of power leads to non-local bugs.

The Iterator Example The example here provides a solution to support callbacks in an object encapsulation context. Another classic example to demonstrate the problem of selective representation exposure is the Iterator example (see [AC04, LP06] for instance). The basic problem is a `List` object needs to protect its internal representation, say a number of `Node` objects. The `Iterator` object of that `List` would face a dilemma: it needs to access the `Node` so it cannot be outside the `List` on the ownership tree, while at the same time it needs to be accessed by clients of the `List`, in which case it cannot live inside the `List` on the ownership tree either.

In our calculus, iterators can be implemented as in Fig. 5.5. The idea is the `Iterator` objectage is owned by the `List`, but it will be exposed to client objectages, illustrated by Fig. 5.5(a). The `Iterator`



(a)



(b)

Figure 5.5: Two Ways to Solve the Iterator Problem

Existing Systems \ Pedigree Types	Pedigree Types inherit	Pedigree Types avoid
EPOT systems	expressiveness	problems in Sec. 3.2
Non-EPOT systems	simple programming interface	expressiveness loss
Universes in particular	pedigree declarations	problems in Sec. 3.2

Table 5.1: How Pedigree Types Relate to Existing Work

objectage can declare a connector `Main` to expose methods such as `hasNext` or `next`. The exposure can be achieved by either by the **connect** expression or the **share** expression. Fig. 5.5(a) demonstrates a general approach to solve the aforementioned "dilemma". For the specific example of iterators, *Classages* has a much more efficient way to program: declaring each iterator as a connector on the `List` objectage rather than having the iterator as an objectage. Clients interact with `List` directly by connecting to its `IteraterC` connector to iterate over all data contained in the `Node`'s, where the `InteraterC` can have per-connection states to keep track of the cursor of the iterator. This is illustrated in Fig. 5.5(b).

5.4 Technical Related Work

A category-wise comparison with object encapsulation systems can be roughly captured by in Table 5.1. The EPOT systems, *i.e.* the systems that explicitly make use of type parametricity, have been explained in detail in Chapter 3. Since our system does not use explicit type parametericity, a more related comparision is with exisiting non-EPOT systems. These systems generally excel at providing very simple programming interfaces, but usually at the cost of language expressiveness. Islands [Hog91] and Balloon [Alm97] severely restrict heap reference structures, by disallowing a "child" object to access any object outside its "parent". Universe Types [MPH01] disallow mutability for this form of access. Several systems in this category have a domain target and make simplifications based on need, such as Confined Types [VB99] for security, and Scoped Types [ACG+06] for real-time memory management. These systems achieve their domain goals, illustrating the value of object encapsulation in these domains.

The work most related to ours in this category is perhaps Universe Types. The similarity lies in the syntax: the two keywords for modifying object references, `peer` and `rep`, are intuitively the same as what

we call a **sibling** and a **child** on the pedigree tree. Under the hood, our type system is fundamentally different from Universe Types. In Universe Types, static pedigree information tracking across objects is limited. For instance, when object o_1 passes a reference declared to be a `peer` to an object o_2 who is the `rep` of o_1, the formal parameter on o_2 has to be `any`, the keyword indicating lack of pedigree information. Further passing such a reference to other objects would lose all pedigree information. Intuitively, the source of this problem is that o_2 should have declared the formal parameter with pedigree information such as `parent`'s `peer`, but keywords like this are simply not provided. Keyword `any` is not a parametrically polymorphic pedigree as we provide – it does not have the ability to remember the pedigree implicitly. To salvage the pedigree information loss, Universe Types allow programmers to dynamically use casting to cast an `any` reference back to a `peer` or a `child`. Putting aside for a moment the run-time overhead for such an operation, dynamic casting still does not solve the previous problem: the language does not have a keyword equivalent of `parent`'s `peer`, and o_2 in fact has no keyword to cast its `any` reference to so it is still stuck. Due to Universe Types' choice of keywords, for non-peer, non-child references the Universes system restricts them to be "read-only". This treatment allows Universe Types to address its own dilemma of how to deal with internal representation exposure, but it appears to have no philosophical foundation (why for example must access to a grandparent object must be read-only?), and is known to lead to programming difficulties: see [LP06]. In addition to the major gain of supporting global and static tracking pedigree information, our system also generalizes the pedigrees programmers can use, many of which are useful ones not supported by Universe Types, such as **self** and **parent**.

Solutions that allow omission of some hierarchy shaping qualifiers are limited. In Generic Ownership [PNCB06], any omission means `world`; In Universe Types, any omission means `peer`. These solutions restrict the shape of the hierarchy and do not align with programmers' "I don't care" intention. Limited inference algorithms have been designed, such SafeJava [Boy04] which only allows intra-procedural inference and AliasJava [AKC02] which is not decidable.

Despite a shared interest on specifying shape, Graph Types [KS93] are a data-structure-centric strategy and their focus is on the shape of the localized reference structure *inside the data structure*, not the shape of entire heap. In Pedigree Types, any two objects on the heap of any object types can explicitly

(given by the programmers) or implicitly (inferred by the type system) have a pedigree relation. Pedigree modifiers can be annotated anywhere an object type can appear, *e.g.* for a field, for method arguments, and for local values. This comparison also applies to other shape invariant enforcement systems. In general, these systems excel at expressing shapes other than hierarchies, while Pedigree Types take advantage of what one can do with an imposed hierarchy. Some properties such as acyclicity of recursive data structures interested in by shape enforcement type systems [KS93, FM97, BRS99] can also be enforced by Pedigree Types. For instance, the `next` field can be qualified with a **child** pedigree, and then the list is known to be acyclic: any instance of `LinkedList` can not be a **child** of itself.

Pedigree Types facilitate language support for several of the design patterns. In Fig. 1.4 Pedigree Types have been used to facilitate **Memento**, by protecting the internal state. Programs using the **Composite**, **Strategy**, **State** patterns gain more direct support in *Classages* for similar reasons.

Chapter 6

The Formal System

In this chapter, we formalize the ideas we have informally introduced previously, including the language abstractions of mixing and connection, together with Pedigree Types. We start off presenting a "formal core", which covers a subset of, but an essential part of, the language constructs and expressions included in the "middleweight" syntax in Chapter 4. Other expressions showing up in the "middleweight" syntax but not included in the "formal core" will be formally described in Sec. 6.5.

6.1 Abstract Syntax of the Formal Core

We first define some basic notation. $\overline{x_n}$ denotes a set $\{x_1, \ldots, x_n\}$. $\overrightarrow{x_n \mapsto y_n}$ is used to denote a mapping sequence (also called a mapping) $[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$. Given $M = \overrightarrow{x_n \mapsto y_n}$, $\mathrm{Dom}(M)$ denotes the domain of M, and it is defined as $\{x_1, \ldots x_n\}$. We also write $M(x_1) = y_1, \ldots M(x_n) = y_n$. A sequence, denoted $\overrightarrow{x_n}$, is defined as a degenerate form of mapping sequence where all elements in the domain map to **null**. When no confusion arises, we also drop the subscript n for sets and mapping sequences and simply use $\overline{x}$ and $\overrightarrow{x \mapsto y}$. Notation $\propto$ is used to denote containment, for instance $(x_2 \mapsto y_2) \propto M$. We write $M[x \mapsto y]$ as a mapping update: M and $M[x \mapsto y]$ are identical except that $M[x \mapsto y]$ maps x to y. Updatable mapping concatenation $\rhd$ is defined as $M_1 \rhd M \stackrel{\mathrm{def}}{=} M_1\{x_1 \mapsto y_1\} \ldots \{x_n \mapsto y_n\}$. $A - B$ is the standard set minus. $M \backslash S$ denotes the longest subsequence of M that maps any element x in $\mathrm{Dom}(M$ but not in S to

C	::=	$\overrightarrow{\mathsf{a} \mapsto \langle K; M; F \rangle}$	*classages*
K	::=	$\overrightarrow{\mathsf{k} \mapsto \langle I; E \rangle}$	*connectors*
I	::=	$\overrightarrow{\mathsf{m}}$	*imports*
E	::=	$\overrightarrow{\mathsf{m} \mapsto \lambda x.e}$	*exports*
M	::=	$\overrightarrow{\mathsf{m} \mapsto \lambda x.e}$	*local methods*
F	::=	$\overrightarrow{\mathsf{f}}$	*local fields*
Ct	::=	$\overrightarrow{\mathsf{a} \mapsto sig}$	*classage sigs*
sig	::=	$\langle Kt; Mt; Ft \rangle$	*classage sig*
Kt	::=	$\overrightarrow{\mathsf{k} \mapsto \langle It; Et \rangle}$	*connector sigs*
It	::=	$\overrightarrow{\mathsf{m} \mapsto (ct \to ct')}$	*import sigs*
Et	::=	$\overrightarrow{\mathsf{m} \mapsto (ct \to ct')}$	*export sigs*
Mt	::=	$\overrightarrow{\mathsf{m} \mapsto (st \to st')}$	*local method sigs*
Ft	::=	$\overrightarrow{\mathsf{f} \mapsto st}$	*field sigs*
st	::=	$ct \mid \mathsf{k}$	*types on sigs*
ct	::=	**unit** | **int** | *Ped* a	*types on connector sigs*
Ped	::=	$(\textbf{parent})^{\mathsf{w}}(\textbf{child})^{\mathsf{z}} \mid \epsilon$	*pedigree qualifier*
$e \in \mathbb{EXP}$	::=	$() \mid x \mid const \mid e; e' \mid$ **this**	*expressions*
	|	**create** *Ped* a	
	|	**connect** e **with** k >> k′	
	|	e->m(e')	
	|	**current**	
	|	**forall**$(x : \mathsf{k})\{e\}$	
	|	::m(e) | f | f:=e	
x			*variable*
const	$\in$	{..., -1, 0, 1, ...}	*integer*
w	$\in$	{0, 1, ...}	*positive level*
z	$\in$	{0, 1 }	*negative level*
a			*classage name*
k			*connector name*
m			*method name*
f			*field name*

Figure 6.1: Abstract Syntax for the *Classages* Formal Core

$M(x)$. Notation $M \mid {}^{elm}_{predicate}$ denotes the longest subsequence of M, say M', where $\forall elm \in M'.predicate$ always holds. We use $\emptyset$ to denote empty set and $[\,]$ to denote mapping sequences of length zero.

The abstract syntax of our calculus is defined in Fig. 6.1. We have explained most of the constructs in the previous section, in the sugared form. For the convenience of formalization, each classage is defined in two parts: the code (C) and the signature (Ct). A program in *Classages* is composed of classage code pieces (C), with one of them being the bootstrapping classage named `Main` with a special connector `Main`. A classage has a unique name (`a`), followed by a list of connectors (K), a list of local methods (M) and fields (F). We omit constructors for this presentation. Each connector also has a name (`k`), a list of **import** methods (I) and a list of **export** methods (E). Objectage instantiation is modeled by the expression **create** *Ped* `a`, which is identical to Java's **new** `a` expression, except that programmers can specify the pedigree for the newly created instance. The "I-don't-care" pedigree is represented as ϵ. Expression `::m`(e) invokes a locally defined method `m` with argument e. Expression `f` and `f:=`e are field reads and writes, respectively. Expression **forall**$(x$: `k`$)\{e\}$ iterates over all connection handles live on connector `k`. Each handle is represented as x and can be used in e. The rest of expressions have been explained in previous chapters. We do not model local variable declarations and assignments, as they can be encoded as local method invocations.

Expression **current** is a special handle that intuitively records "the current connection". It can only be used inside a connector. Expression **forall**$(x$: `k`$)\{e\}$ can be used to enumerate all connections currently alive on the connector `k`, where x is the connection handle iterating over all connections.

Classage signatures (Ct) are homomorphic with C, only with type information. User-declared types (st) include primitive types, pedigree types, and the type for connection handles, represented by the name of the connector (`k`). Connection handle types are not allowed on connector signatures. This is in line with our earlier discussion that connection handles must remain local (Sec. 5.3): the syntax restriction here disallows them from being passed through connectors, the only means for inter-objectage interaction in our language.

$$
\begin{array}{rcl}
\Psi(Ct) & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{a}_n \mapsto \mathit{psig}_n} \\
& \text{if} & Ct = \overrightarrow{\mathsf{a}_n \mapsto \mathit{sig}_n} \\
& & \forall\, i \in \{1, \ldots, n\},\ \mathit{psig}'_i = \Psi_{\mathtt{sig}}(\mathit{sig}_i, Ct, \emptyset) \\
& & \mathit{psig}_i = \forall\langle\alpha_1;\beta_1\rangle \ldots \forall\langle\alpha_u;\beta_u\rangle.\mathit{psig}'_i \\
& & FP(\mathit{psig}'_i) = \{\langle\alpha_1;\beta_1\rangle, \ldots, \langle\alpha_u;\beta_u\rangle\} \\
\Psi_{\mathtt{sig}}(\mathit{sig}, Ct, Z) & \stackrel{\text{def}}{=} & \langle \Psi_{\mathtt{cl}}(Kt, Ct, Z); \Psi_{\mathtt{ml}}(Mt, Ct, Z); \Psi_{\mathtt{fl}}(Ft, Ct, Z)\rangle \\
& \text{if} & \mathit{sig} = \langle Kt; Mt; Ft\rangle \\
\Psi_{\mathtt{cl}}(Kt, Ct, Z) & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{k} \mapsto \langle\ \Psi_{\mathtt{iel}}(It, Ct, Z);\ \Psi_{\mathtt{iel}}(Et, Ct, Z)\rangle} \\
& \text{if} & Kt = \overrightarrow{\mathsf{k} \mapsto \langle It; Et\rangle} \\
\Psi_{\mathtt{iel}}(\iota, Ct, Z) & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{m} \mapsto (\Psi_{\mathtt{ct}}(ct, Ct, Z) \to \Psi_{\mathtt{ct}}(ct', \mathcal{C}t, Z))} \\
& \text{if} & \iota = \overrightarrow{\mathsf{m} \mapsto (ct \to ct')}, \iota \text{ is } It \text{ or } Et \\
\Psi_{\mathtt{ml}}(Mt, Ct, Z) & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{m} \mapsto (\Psi_{\mathtt{st}}(st, Ct, Z) \to \Psi_{\mathtt{st}}(st', \mathcal{C}t, Z))} \\
& \text{if} & Mt = \overrightarrow{\mathsf{m} \mapsto (st \to st')} \\
\Psi_{\mathtt{fl}}(Ft, Ct, Z) & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{f} \mapsto \Psi_{\mathtt{st}}(st, Ct, Z)} \\
& \text{if} & Ft = \overrightarrow{\mathsf{f} \mapsto st} \\
\Psi_{\mathtt{ct}}(\textbf{int}, Ct, Z) & \stackrel{\text{def}}{=} & \textbf{int} \\
\Psi_{\mathtt{ct}}(\textbf{unit}, Ct, Z) & \stackrel{\text{def}}{=} & \textbf{unit} \\
\Psi_{\mathtt{ct}}(Ped\ \mathsf{a}, Ct, Z) & \stackrel{\text{def}}{=} & \begin{cases} \mu\mathsf{a}.\mathcal{K}\ @\rho & \text{if } \mathsf{a} \notin Z \\ \mathsf{a}@\rho & \text{if } \mathsf{a} \in Z \end{cases} \\
& \text{if} & \Psi_{\mathtt{sig}}(Ct(\mathsf{a}), Ct, Z \cup \{\mathsf{a}\}) = \langle \mathcal{K}; \mathcal{M}; \mathcal{F}\rangle \\
& & \rho = \begin{cases} (\textbf{parent})^{\alpha}(\textbf{child})^{\beta} & \text{if } Ped = \epsilon \\ & \quad \alpha, \beta \text{ fresh} \\ Ped & \text{otherwise} \end{cases} \\
\Psi_{\mathtt{st}}(st, Ct, Z) & \stackrel{\text{def}}{=} & \begin{cases} \mathsf{k} & st = \mathsf{k} \\ \Psi_{\mathtt{ct}}(st, Ct, Z) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 6.2: $\Psi(Ct) = \mathcal{C}$: Level Parameterization

$\mathcal{C}$	::=	$\overrightarrow{\mathsf{a} \mapsto \mathit{psig}}$	*parameterized sigs*
psig	::=	$\overrightarrow{\forall\langle\alpha;\beta\rangle}.\langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle$	*parameterized sig*
$\mathcal{K}$	::=	$\overrightarrow{\mathsf{k} \mapsto \langle\mathcal{I};\mathcal{E}\rangle}$	*connector sigs*
$\mathcal{I}, \mathcal{E}$	::=	$\overrightarrow{\mathsf{m} \mapsto (c\tau \to c\tau')}$	*import/export sigs*
$\mathcal{M}$	::=	$\overrightarrow{\mathsf{m} \mapsto (s\tau \to s\tau')}$	*local method sigs*
$\mathcal{F}$	::=	$\overrightarrow{\mathsf{f} \mapsto s\tau}$	*field sigs*
τ	::=	$s\tau \mid \bot$	*expression types*
$s\tau$	::=	$\mathsf{k} \mid c\tau$	*types on sigs*
$c\tau$	::=	$\textbf{unit} \mid \textbf{int} \mid \mu\mathsf{a}.\mathcal{K}\ @\rho \mid \mathsf{a}@\rho$	*types on connector sigs*
ρ	::=	$(\textbf{parent})^{\nu}(\textbf{child})^{\nu'}$	*pedigree expression*
ν	::=	$\nu + \nu' \mid \nu - \nu' \mid \alpha \mid \mathsf{w}$	*level expression*
α, β			*level type variable*

Figure 6.3: Signatures and Types in the Type System

(Sub-Bottom)
$\Omega \vdash \bot <: \tau \backslash \emptyset$

(Sub-Int)
$\Omega \vdash \textbf{int} <: \textbf{int} \backslash \emptyset$

(Sub-Unit)
$\Omega \vdash \textbf{unit} <: \textbf{unit} \backslash \emptyset$

(Sub-Recursive)
$$\frac{(\mathsf{a}_1 <: \mathsf{a}_2) \in \Omega}{\Omega \vdash \mathsf{a}_1 @ \rho_1 <: \mathsf{a}_2 @ \rho_2 \backslash subPed(\rho_1, \rho_2)}$$

(Sub-Non-Recursive)
$$\frac{\begin{array}{c}\mathrm{Dom}(\mathcal{K}_2) = \{\mathsf{k}_1, \ldots, \mathsf{k}_n\} \qquad \mathsf{a}'_1, \mathsf{a}'_2 \text{ fresh} \\ \mathcal{K}'_1 = \mathcal{K}_1\{\mathsf{a}'_1/\mathsf{a}_1\} \qquad \mathcal{K}'_2 = \mathcal{K}_2\{\mathsf{a}'_2/\mathsf{a}_2\} \qquad \forall i \in \{1, \ldots, n\}.\ \Omega \cup \{\mathsf{a}'_1 <: \mathsf{a}'_2\} \vdash_{\mathsf{c}} \mathcal{K}'_1(\mathsf{k}_i) <: \mathcal{K}'_2(\mathsf{k}_i) \backslash \Sigma_i\end{array}}{\Omega \vdash \mu \mathsf{a}_1.\mathcal{K}_1\ @\rho_1 <: \mu \mathsf{a}_2.\mathcal{K}_2\ @\rho_2 \backslash (\Sigma_1 \cup \ldots \Sigma_n \cup subPed(\rho_1, \rho_2))}$$

(Sub-Connector)
$$\frac{\begin{array}{c}\mathrm{Dom}(\mathcal{E}_2) = \{\mathsf{m}_1, \ldots, \mathsf{m}_p\} \\ \mathrm{Dom}(\mathcal{I}_1) = \{\mathsf{m}_{p+1}, \ldots, \mathsf{m}_q\} \\ \forall i \in \{1, \ldots, p\}.\ \Omega \vdash_{\mathtt{m}} \mathcal{E}_1(\mathsf{m}_i) <: \mathcal{E}_2(\mathsf{m}_i) \backslash \Sigma_i \qquad \forall j \in \{p+1, \ldots, q\}.\ \Omega \vdash_{\mathtt{m}} \mathcal{I}_2(\mathsf{m}_i) <: \mathcal{I}_1(\mathsf{m}_i) \backslash \Sigma_j\end{array}}{\Omega \vdash_{\mathsf{c}} \langle \mathcal{I}_1; \mathcal{E}_1 \rangle <: \langle \mathcal{I}_2; \mathcal{E}_2 \rangle \backslash (\Sigma_1 \cup \cdots \cup \Sigma_q)}$$

(Sub-Method)
$$\frac{\Omega \vdash c\tau'_1 <: c\tau'_2 \backslash \Sigma \qquad \Omega \vdash c\tau_2 <: c\tau_1 \backslash \Sigma'}{\Omega \vdash_{\mathtt{m}} c\tau_1 \mapsto c\tau'_1 <: c\tau_2 \mapsto c\tau'_2 \backslash (\Sigma \cup \Sigma')}$$

$$\begin{array}{c} subPed(\rho_1, \rho_2) \stackrel{\text{def}}{=} \{\nu_{11} - \nu_{12} =_{\mathtt{s}} \nu_{21} - \nu_{22}, 0 \leq_{\mathtt{s}} \nu_{12} \leq_{\mathtt{s}} \nu_{22} \leq_{\mathtt{s}} 1\} \\ \rho_1 = (\textbf{parent})^{\nu_{11}} (\textbf{child})^{\nu_{12}} \\ \rho_2 = (\textbf{parent})^{\nu_{21}} (\textbf{child})^{\nu_{22}} \end{array}$$

Figure 6.4: Subtyping Rules

$$\begin{array}{rcl}
FP_{\mathtt{sig}}(\langle \mathcal{K}; \mathcal{M}; \mathcal{F} \rangle) & \stackrel{\text{def}}{=} & FP_{\mathtt{cl}}(\mathcal{K}) \cup FP_{\mathtt{ml}}(\mathcal{M}) \cup FP_{\mathtt{fl}}(\mathcal{F}) \\
FP_{\mathtt{cl}}(\overrightarrow{\mathsf{k}_n \mapsto \langle \mathcal{I}_n; \mathcal{E}_n \rangle}) & \stackrel{\text{def}}{=} & FP_{\mathtt{ml}}(\mathcal{I}_1) \cup FP_{\mathtt{ml}}(\mathcal{E}_1) \cup \ldots FP_{\mathtt{ml}}(\mathcal{I}_n) \cup FP_{\mathtt{ml}}(\mathcal{E}_n) \\
FP_{\mathtt{ml}}(\overrightarrow{\mathsf{m}_n \mapsto (s\tau_n \to s\tau'_n)}) & \stackrel{\text{def}}{=} & FP_{\mathtt{st}}(s\tau_1) \cup FP_{\mathtt{st}}(s\tau'_1) \cup \ldots FP_{\mathtt{st}}(s\tau_n) \cup FP_{\mathtt{st}}(s\tau_n) \\
FP_{\mathtt{fl}}(\overrightarrow{\mathsf{f}_n \mapsto s\tau_n}) & \stackrel{\text{def}}{=} & FP_{\mathtt{st}}(s\tau_1) \cdots \cup FP_{\mathtt{st}}(s\tau_n)
\end{array}$$

$$\begin{array}{rcl}
FP_{\mathtt{st}}(\mathsf{k}) & \stackrel{\text{def}}{=} & \emptyset \\
FP_{\mathtt{st}}(\textbf{unit}) & \stackrel{\text{def}}{=} & \emptyset \\
FP_{\mathtt{st}}(\textbf{int}) & \stackrel{\text{def}}{=} & \emptyset \\
FP_{\mathtt{st}}(\mu \mathsf{a}.\mathcal{K}\ @(\textbf{parent})^{\alpha}(\textbf{child})^{\beta}) & \stackrel{\text{def}}{=} & \{\langle \alpha; \beta \rangle\} \cup FP_{\mathtt{cl}}(\mathcal{K}) \\
FP_{\mathtt{st}}(\mathsf{a}@(\textbf{parent})^{\alpha}(\textbf{child})^{\beta}) & \stackrel{\text{def}}{=} & \{\langle \alpha; \beta \rangle\}
\end{array}$$

Figure 6.5: Pedigree Type Variable Pair Occurrence in Signatures

6.2 The Type System

As we explained in Sec. 5.1, our type system supports parametric polymorphism, but in an implicit form – the programmer syntax in Fig. 6.1 does not provide explicit parameterization of classages. A straightforward function called Ψ is used to properly parameterize the classage signature into the more type-theory-friendly form where all type variables are bound at the beginning. We first introduce the Ψ function in Sec. 6.2.1, and then introduce the main parts of the type system in the rest of the section.

6.2.1 Level Parameterization

In the signature of `Button` in Fig. 5.1, the `logger` field is not qualified with a pedigree. Intuitively, the "I-don't-care-about-the-pedigree" intention translates as each instance of `Button` having a fresh pair of levels (positive level and negative level). The Ψ function is defined in Fig. 6.2 to help capture this notion. The definition is uninteresting for the most part: it simply goes through the structure of each signature (sig_i), and generates a pair of variables for each unqualified occurrence such as `logger` (*i.e. Ped* $= \epsilon$). These variables are called *level type variables*, with metavariable α ranging over positive pedigree levels, and β ranging over negative pedigree levels. The classage signatures produced by Ψ are bound with these variables, with helper function $FP()$ enumerating all pairs of level type variables occurring free in the signature. The Ψ function processes classage signatures transitively. This means a classage not only provides parametric polymorphism for the levels of direct objectage references in the classage, but also allows these parameters to be forwarded to provide polymorphism for indirect objectage references. Fig. 6.3 gives the format of the signature list ($\mathcal{C}$) used in most typing rules of the type system, and it is the result computed by $\Psi.\,\mathcal{C}$ is homomorphic to Ct in programmer syntax.

Helper function $FP()$ is rigorously defined in Fig. 6.5. $FP_{\texttt{sig}}(-)$ defines the set of pedigree type variables pairs occurring in a classage signature without $\forall$ bindings, $FP_{\texttt{cl}}(-)$ defines those in a connector signature list, $FP_{\texttt{ml}}(-)$ defines those in a method signature list (which can either be an import list, an export list, or a local method list), $FP_{\texttt{fl}}(-)$ defines those in a field signature list, $FP_{\texttt{st}}(-)$ defines those in a type enclosed in a signature.

In object-oriented languages, recursive types are the norm. In the Ψ function, we also unfold recursive types explicitly, using a variation of the standard μ-type techniques [AC93]. Set Z is used to keep track of recursion. In the resulting signature computed by Ψ, each occurrence of user-declared pedigree type is transformed into either the $\mu\mathsf{a}.\mathcal{K}\ @\rho$ form or the $\mathsf{a}@\rho$ form. $\mu\mathsf{a}.\mathcal{K}\ @\rho$ in principle still follows the general form of the pedigree type – a combination of pedigree (ρ) and the object type (the rest of it, with name a and structure information in $\mathcal{K}$). An object (objectage) type in *Classages* is represented as the list of connector signatures. This is intuitive: how an objectage is viewed by others is determined by its connectors. $\mathsf{a}@\rho$ is used for the objectage type already bound by μ in its enclosing classage signature. For convenience, we sometimes call the two forms of pedigree types *the μ form* and *the abbreviated form* respectively.

Note the different fonts for the metavariables as opposed to the syntax in Fig. 6.1. For instance, M represents a list of local methods, Mt represents a list of user-defined method signatures, and $\mathcal{M}$ represents a list of method signatures after the parameterization process.

6.2.2 Types and Subtyping

As defined in Fig. 6.3, expression types τ used by the type system are also similar to the programmer declared types, except that a $\perp$ type is added to typecheck uninitiated fields, and the pedigree type are now either in the μ form or in the abbreviated form. In the general case of expression typing, the positive levels and negative levels of a pedigree type might be linear expressions over level variables. The general form of ρ thus is a type expression, which we call a *pedigree expression*. The linear expressions in this case are called *level expressions*. In the abstract syntax, we omit parentheses for level expressions. They are implicitly added to preserve arithmetic precedence. For instance if $\nu_1 =_{\mathtt{s}} 1$ and $\nu_2 =_{\mathtt{s}} 1+3$, then $\nu_1 - \nu_2$ is $1-(1+3)$. When there is no need to distinguish the positive level and the negative level, we also use metavariable w to represent either level constant, and metavariable α to represent either level variable. To avoid confusion, equality/inequality symbols showing up in the constraint sets are denoted as $=_{\mathtt{s}}$, $\geq_{\mathtt{s}}$, $\leq_{\mathtt{s}}$ respectively.

Subtyping is defined in Fig. 6.4 with judgments of the form $\Omega \vdash \tau <: \tau' \backslash \Sigma$. This judgment reads "$\tau$ is a subtype of τ' under pedigree constraints Σ and assumptions Ω". The main rules are (Sub-Non-

(T-Global)

$$\frac{\Psi(Ct) = \mathcal{C} \qquad \mathrm{Dom}(\mathcal{C}) = \{\mathsf{a}_1, \ldots, \mathsf{a}_n\} \qquad \forall i \in \{1, \ldots, n\} \text{ such that } [\mathbf{me} \mapsto \mathsf{a}_i], \mathcal{C} \vdash_{\mathtt{cls}} C(\mathsf{a}_i) : \mathcal{C}(\mathsf{a}_i) \backslash \Sigma_i, \Pi_i \qquad \overrightarrow{\mathsf{a}_n \mapsto \langle \Sigma_n, \Pi_n \rangle}, [\,] \vdash_{\mathtt{cons}} \langle \mathtt{main}; [\,]; [\,] \rangle \backslash \Sigma \qquad \Sigma \text{ consistent}}{\vdash_{\mathtt{G}} C : Ct}$$

(T-Classage)

$$\frac{\mathrm{Dom}(F) = \mathrm{Dom}(\mathcal{F}) \qquad \Gamma,\, \mathcal{C} \vdash_{\mathtt{m}} M : \mathcal{M} \backslash \Sigma, \Pi \qquad \Gamma,\, \mathcal{C} \vdash_{\mathtt{c}} K : \mathcal{K} \backslash \Sigma', \Pi' \qquad \mathit{psig} = \forall \langle \alpha_1; \beta_1 \rangle, \ldots \forall \langle \alpha_n; \beta_n \rangle . \langle \mathcal{K}; \mathcal{M}; \mathcal{F} \rangle}{\Gamma, \mathcal{C} \vdash_{\mathtt{cls}} \langle K; M; F \rangle : \mathit{psig} \backslash \Sigma \cup \Sigma' \cup \bigcup_{i \in \{1,\ldots,n\}} \mathit{twoVals}(\beta_i), \Pi \cup \Pi'}$$

(T-Connectors)

$$\frac{\mathrm{Dom}(K) = \mathrm{Dom}(\mathcal{K}) = \{\mathsf{k}_1, \ldots, \mathsf{k}_n\} \qquad \forall i \in \{1, \ldots, n\} \qquad K(\mathsf{k}_i) = \langle I_i; E_i \rangle \qquad \mathcal{K}(\mathsf{k}_i) = \langle \mathcal{I}_i; \mathcal{E}_i \rangle \qquad \mathrm{Dom}(I_i) = \mathrm{Dom}(\mathcal{I}_i) \qquad \Gamma \rhd (\mathbf{current} \mapsto \mathsf{k}_i),\ \mathcal{C} \vdash_{\mathtt{m}} E_i : \mathcal{E}_i \backslash \Sigma_i, \Pi_i}{\Gamma,\, \mathcal{C} \vdash_{\mathtt{c}} K : \mathcal{K} \backslash \Sigma_1 \cup \ldots \Sigma_n, \Pi_1 \cup \ldots \Pi_n}$$

(T-Methods)

$$\frac{\mathrm{Dom}(M) = \mathrm{Dom}(\mathcal{M}) = \{\mathsf{m}_1, \ldots, \mathsf{m}_n\} \qquad \forall i \in \{1, \ldots, n\} \qquad M(\mathsf{m}_i) = \lambda x_i . e_i \qquad \mathcal{M}(\mathsf{m}_i) = \tau_i \mapsto \tau_i' \qquad \Gamma \rhd (x_i \mapsto \tau_i),\ \mathcal{C} \vdash\ e_i : \tau_i' \backslash \Sigma_i, \Pi_i}{\Gamma,\, \mathcal{C} \vdash_{\mathtt{m}} M : \mathcal{M} \backslash \Sigma_1 \cup \ldots \Sigma_n, \Pi_1 \cup \ldots \Pi_n}$$

Γ	::=	$\overrightarrow{x \mapsto \tau \mid \mathbf{me} \mapsto \mathsf{a} \mid \mathbf{current} \mapsto \mathsf{k}}$	*typing environment*
Σ	::=	$\overline{\nu =_{\mathtt{s}} 0}$	*constraint set*
Π	::=	$\overline{\langle \mathsf{a}; \rho; \sigma \rangle}$	*instantiation record set*

Figure 6.6: Top-Level Typing Rules and Definitions

Recursive) and (Sub-Recursive), which show the objectage type part of the pedigree type follows standard structural subtyping. Since types may be recursive, the standard rules for subtyping over recursive types – commonly known as the Amber rules – are used [AC93]. Data structure Ω, which is a partial order with elements of the form $\mathsf{a} <: \mathsf{a}'$, is standard for this purpose, as is the alpha conversion (*i.e.* $\mathcal{K}_1\{\mathsf{a}_1'/\mathsf{a}_1\}$ and $\mathcal{K}_2\{\mathsf{a}_2'/\mathsf{a}_2\}$ in (Sub-Non-Recursive)).

The interesting part of the subtyping rules is perhaps the subtyping of two pedigree expressions, captured by $subPed$. It captures the notion of pedigree subsumption (Sec. 5.1); the basic case of pedigree identity is also included implicitly by this definition. For instance, it can be easily observed that $subPed(\mathbf{self}, \mathbf{sibling})$ and $subPed(\mathbf{sibling}, \mathbf{sibling})$ are both consistent.

(T-Unit)
$\Gamma,\ \mathcal{C} \vdash () : \textbf{unit} \backslash \emptyset, \emptyset$

(T-Int)
$\Gamma,\ \mathcal{C} \vdash cst : \textbf{int} \backslash \emptyset, \emptyset$

(T-Var)
$\Gamma,\ \mathcal{C} \vdash x : \Gamma(x) \backslash \emptyset, \emptyset$

(T-Current)
$\Gamma,\ \mathcal{C} \vdash \textbf{current} : \Gamma(\textbf{current}) \backslash \emptyset, \emptyset$

(T-Create)

$$\frac{\begin{array}{c}\rho = \begin{cases} (\textbf{parent})^{\alpha}(\textbf{child}) & \text{if } \mathit{Ped} = \epsilon \text{ and } \alpha \text{ fresh} \\ \mathit{Ped} & \text{if } \mathit{Ped} = (\textbf{parent})^{\mathsf{w}}(\textbf{child}) \end{cases} \\ \mathcal{C}(\mathsf{a}) = \forall\langle\alpha_1;\beta_1\rangle,\ldots,\forall\langle\alpha_n;\beta_n\rangle.\langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle \qquad \alpha'_1,\ldots,\alpha'_n,\beta'_1,\ldots,\beta'_n \text{ fresh} \\ \sigma = [\langle\alpha_1;\beta_1\rangle \mapsto \langle\alpha'_1;\beta'_1\rangle,\ldots,\langle\alpha_n;\beta_n\rangle \mapsto \langle\alpha'_n\beta'_n\rangle] \qquad \Sigma = \bigcup_{i=\{1,\ldots,n\}} \mathit{twoVals}(\beta'_i)\end{array}}{\Gamma,\ \mathcal{C} \vdash \textbf{create } \mathit{Ped}\ \mathsf{a} : (\mu\mathsf{a}.\mathcal{K}\ @\rho)[\sigma] \backslash \Sigma, \{\langle \mathsf{a}; \rho; \sigma\rangle\}}$$

(T-Connect)

$$\frac{\begin{array}{c}\Gamma(\textbf{me}) = \mathsf{a}_1 \qquad \mathcal{C}(\mathsf{a}_1) = \overline{\forall\langle\alpha;\beta\rangle}.\ \langle\mathcal{K}_1;\mathcal{M}_1;\mathcal{F}_1\rangle \\ \Gamma,\ \mathcal{C} \vdash e : \mu\mathsf{a}_2.\mathcal{K}_2\ @\rho \backslash \Sigma, \Pi \qquad \mathcal{K}'_2 = \mathcal{K}_2\{\mathsf{a}_2 \circlearrowleft \mathcal{K}_2\} \qquad \rho \vdash \mathcal{K}_1(\mathsf{k}_1) \overset{\Sigma'}{\Longrightarrow} \mathcal{K}'_2(\mathsf{k}_2)\end{array}}{\Gamma,\ \mathcal{C} \vdash \textbf{connect } e \textbf{ with } \mathsf{k}_1\ \texttt{>>}\ \mathsf{k}_2 : \mathsf{k}_1 \backslash \Sigma \cup \Sigma', \Pi}$$

(T-HandleM)

$$\frac{\begin{array}{c}\mathcal{C}(\Gamma(\textbf{me})) = \overline{\forall\langle\alpha;\beta\rangle}.\ \langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle \qquad \Gamma,\ \mathcal{C} \vdash e : \mathsf{k} \backslash \Sigma, \Pi \\ \mathcal{K}(\mathsf{k}) = \langle\mathcal{I};\mathcal{E}\rangle \qquad (\mathcal{I} \rhd \mathcal{E})(\mathsf{m}) = \tau' \to \tau \qquad \mathrm{Dom}(\mathcal{I}) \cap \mathrm{Dom}(\mathcal{E}) = \emptyset \qquad \Gamma,\ \mathcal{C} \vdash e' : \tau' \backslash \Sigma', \Pi'\end{array}}{\Gamma,\ \mathcal{C} \vdash e\texttt{->}\mathsf{m}(e') : \tau \backslash \Sigma \cup \Sigma', \Pi \cup \Pi'}$$

Figure 6.7: Expression Typing Rules (Part 1)

(T-Forall)

$$\frac{\mathcal{C}(\Gamma(\textbf{me})) = \overline{\forall\langle\alpha;\beta\rangle}.\ \langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle \qquad \mathsf{k} \in \mathrm{Dom}(\mathcal{K}) \qquad \Gamma \rhd (x \mapsto \mathsf{k}),\ \mathcal{C} \vdash e : \tau\backslash\Sigma,\Pi}{\Gamma,\ \mathcal{C} \vdash \textbf{forall}(x : \mathsf{k})\{e\} : \tau\backslash\Sigma,\Pi}$$

(T-LocalM)

$$\frac{\mathcal{C}(\Gamma(\textbf{me})) = \overline{\forall\langle\alpha;\beta\rangle}.\ \langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle \qquad \mathcal{M}(\mathsf{m}) = \tau' \to \tau \qquad \Gamma,\ \mathcal{C} \vdash e : \tau'\backslash\Sigma,\Pi}{\Gamma,\ \mathcal{C} \vdash \mathtt{::}\mathsf{m}(e) : \tau\backslash\Sigma,\Pi}$$

(T-Self)

$$\frac{\Gamma(\textbf{me}) = \mathsf{a} \qquad \mathcal{C}(\mathsf{a}) = \overline{\forall\langle\alpha;\beta\rangle}.\ \langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle}{\Gamma,\ \mathcal{C} \vdash \textbf{this} : \mu\mathsf{a}.\mathcal{K}\ @\textbf{self}\backslash\emptyset,\emptyset}$$

(T-Sub)

$$\frac{\Gamma,\ \mathcal{C} \vdash e : \tau\backslash\Sigma,\Pi \qquad \emptyset \vdash \tau <: \tau'\backslash\Sigma'}{\Gamma,\ \mathcal{C} \vdash e : \tau'\backslash(\Sigma \cup \Sigma'),\Pi}$$

(T-Get)

$$\frac{\mathcal{C}(\Gamma(\textbf{me})) = \overline{\forall\langle\alpha;\beta\rangle}.\ \langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle \qquad \mathcal{F}(\mathsf{f}) = \tau}{\Gamma,\ \mathcal{C} \vdash \mathsf{f} : \tau\backslash\emptyset,\emptyset}$$

(T-Set)

$$\frac{\mathcal{C}(\Gamma(\textbf{me})) = \overline{\forall\langle\alpha;\beta\rangle}.\ \langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle \qquad \mathcal{F}(\mathsf{f}) = \tau \qquad \Gamma,\ \mathcal{C} \vdash e : \tau\backslash\Sigma,\Pi}{\Gamma,\ \mathcal{C} \vdash \mathsf{f}\mathtt{:=}e : \tau\backslash\Sigma,\Pi}$$

(T-Sequence)

$$\frac{\Gamma,\ \mathcal{C} \vdash e_1 : \tau_1\backslash\Sigma_1,\Pi_1 \qquad \Gamma,\mathcal{C} \vdash e_2 : \tau_2\backslash\Sigma_2,\Pi_2}{\Gamma,\ \mathcal{C} \vdash e_1;\ e_2 : \tau_2\backslash(\Sigma_1 \cup \Sigma_2),(\Pi_1 \cup \Pi_2)}$$

Figure 6.8: Expression Typing Rules (Part 2)

(Def-ConnectorMatch)

$$\frac{\mathrm{Dom}(\mathcal{I}_1) = \{\mathsf{m}_1, \ldots, \mathsf{m}_p\} \qquad \forall i \in \{1, \ldots, p\}.(\textbf{parent})^{\nu_2}(\textbf{child})^{\nu_1} \vdash \mathcal{E}_2(\mathsf{m}_i) \overset{\Sigma_i}{\underset{\mathtt{m}}{\Longrightarrow}} \mathcal{I}_1(\mathsf{m}_i)}{\mathrm{Dom}(\mathcal{I}_2) = \{\mathsf{m}_{p+1}, \ldots, \mathsf{m}_q\} \qquad \forall j \in \{p+1, \ldots, q\}.(\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2} \vdash \mathcal{E}_1(\mathsf{m}_j) \overset{\Sigma_j}{\underset{\mathtt{m}}{\Longrightarrow}} \mathcal{I}_2(\mathsf{m}_j)}$$

$$(\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2} \vdash \langle \mathcal{I}_1; \mathcal{E}_1 \rangle \overset{\Sigma_1 \cdots \cup \Sigma_q}{\Longrightarrow} \langle \mathcal{I}_2; \mathcal{E}_2 \rangle$$

(Def-MethodMatch)

$$\frac{match(c\tau'_{\mathtt{im}}, c\tau'_{\mathtt{em}}, (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}) = \Sigma_1 \qquad match(c\tau_{\mathtt{em}}, c\tau_{\mathtt{im}}, (\textbf{parent})^{\nu_b}(\textbf{child})^{\nu_a}) = \Sigma_2}{(\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b} \vdash c\tau'_{\mathtt{em}} \to c\tau_{\mathtt{em}} \overset{\Sigma_1 \cup \Sigma_2}{\underset{\mathtt{m}}{\Longrightarrow}} c\tau'_{\mathtt{im}} \to c\tau_{\mathtt{im}}}$$

$$\begin{array}{rcl}
match(c\tau, c\tau', \rho) & \stackrel{\text{def}}{=} & \Sigma \cup wfcons(c\tau, \rho) \\
& \text{if} & \emptyset \vdash convert(c\tau, \rho) <: c\tau' \backslash \Sigma \\
convert(\textbf{int}, \rho) & \stackrel{\text{def}}{=} & \textbf{int} \\
convert(\textbf{unit}, \rho) & \stackrel{\text{def}}{=} & \textbf{unit} \\
convert(\mu\mathsf{a}.\mathcal{K}\ @\rho_1, \rho_2) & \stackrel{\text{def}}{=} & \mu\mathsf{a}.\mathcal{K}\ @\rho_3 \\
& \text{if} & relativize(\rho_1, \rho_2) = \rho_3, \Sigma \\
\\
wfcons(\textbf{int}, \rho) & \stackrel{\text{def}}{=} & \emptyset \\
wfcons(\textbf{unit}, \rho) & \stackrel{\text{def}}{=} & \emptyset \\
wfcons(\mu\mathsf{a}.\mathcal{K}\ @\rho_1, \rho_2) & \stackrel{\text{def}}{=} & \Sigma \\
& \text{if} & relativize(\rho_1, \rho_2) = \rho_3, \Sigma \\
\\
relativize(\rho_1, \rho_2) & \stackrel{\text{def}}{=} & (\textbf{parent})^{\nu_{21}+\nu_{11}-\nu_{22}}(\textbf{child})^{\nu_{12}}, \{\nu_{21} \geq_{\mathtt{s}} \nu_{12}\} \\
& \text{if} & \rho_1 = (\textbf{parent})^{\nu_{11}}(\textbf{child})^{\nu_{12}} \\
& & \rho_2 = (\textbf{parent})^{\nu_{21}}(\textbf{child})^{\nu_{22}} \\
\\
relativizem(\rho, [\,]) & \stackrel{\text{def}}{=} & \rho, \emptyset \\
relativizem(\rho, [\rho_1, \ldots \rho_n]) & \stackrel{\text{def}}{=} & \rho'_n, \Sigma_1 \cup \ldots, \Sigma_n \\
& \text{if} & n \geq 1 \\
& & relativize(\rho, \rho_1) = \rho'_1, \Sigma_1 \\
& & relativize(\rho'_1, \rho_2) = \rho'_2, \Sigma_2 \\
& & \ldots \\
& & relativize(\rho'_{n-1}, \rho_n) = \rho'_n, \Sigma_n
\end{array}$$

Figure 6.9: Connector Matching and Pedigree Relativization

6.2.3 The Typechecking Process

Top-level typing rules are presented in Fig.6.6. Expression typing rules are given in Fig. 6.7 and Fig. 6.8.

The Rules The whole program is globally typechecked with (T-Global). This rule first typechecks each classage via $\vdash_{\texttt{cls}}$, and then merges the constraints via $\vdash_{\texttt{cons}}$. When there are no conflicting constraints in the final constraint set Σ, the program typechecks. This section focuses on per-classage typechecking. The discussion on constraint merging is deferred to Sec. 6.2.5. We also postpone discussion on typechecking the **create** expression and the data structure Π in the rules, as they are also closely related to constraint merging.

Rules (T-Classage), (T-Connectors), and (T-Methods) are straightforward for typing the respective constructs. Expressions are typed via the judgment $\Gamma, \mathcal{C} \vdash e : \tau \backslash \Sigma,\ \Pi$. Typing environment Γ is defined at the bottom of Fig. 6.6; it maps variables to types, the special keyword **me** to the name of the classage enclosing the expression, and the special keyword **current** to the name of the connector enclosing the expression. (T-HandleM) demonstrates the typing of invoking a connection handle. It guarantees that the method is indeed an **import** or an **export** defined in the connector the connection is established on. (T-Forall) checks the connector the expression enumerates over indeed exists ($\mathsf{k} \in \mathrm{Dom}(\mathcal{K})$) and then types the body e with the assumption that the enumerated handle x has connection handle type k. (T-LocalM) types local method invocation, and it checks the parameter type indeed matches the method signature in $\mathcal{M}$. (T-Self) types **this**, which is given a **self** pedigree. (T-Sub) bridges with the subtyping rules. (T-Connect) will be explained in Sec. 6.2.4.

Pedigree Constraints and Decidable Constraint Solving All constraints in Σ (defined at the bottom of Fig. 6.6) are constraining pedigree levels (positive and negative), which are natural numbers. The constraint solving task in our type system is to find nonnegative solutions to a system of linear diophantine equations. For positive level type variables, this is obvious: they range over $\{0, 1, \dots\}$, and finding a satisfiable positive level is equivalent to finding nonnegative solutions to linear equations. Negative level type variables slightly complicate the matter: they only range over $\{0, 1\}$ and not $\{0, 1, \dots\}$. Our solution is that when such a

$$
\begin{array}{rcl}
\mathcal{K}\{\mathsf{a} \circlearrowleft \mathcal{K}'\} & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{k} \mapsto \langle \mathcal{I}\{\mathsf{a} \circlearrowleft \mathcal{K}'\}; \mathcal{E}\{\mathsf{a} \circlearrowleft \mathcal{K}'\}\rangle} \\
 & \text{if} & \mathcal{K} = \overrightarrow{\mathsf{k} \mapsto \langle \mathcal{I}; \mathcal{E}\rangle} \\
\iota\{\mathsf{a} \circlearrowleft \mathcal{K}'\} & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{m} \mapsto (c\tau\{\mathsf{a} \circlearrowleft \mathcal{K}'\} \to c\tau'\{\mathsf{a} \circlearrowleft \mathcal{K}'\})} \\
 & \text{if} & \iota = \overrightarrow{\mathsf{m} \mapsto (c\tau \to c\tau')}, \iota \text{ is } \mathcal{I} \text{ or } \mathcal{E} \\
\textbf{int}\{\mathsf{a} \circlearrowleft \mathcal{K}'\} & \stackrel{\text{def}}{=} & \textbf{int} \\
\textbf{unit}\{\mathsf{a} \circlearrowleft \mathcal{K}'\} & \stackrel{\text{def}}{=} & \textbf{unit} \\
\mu\mathsf{a}.\mathcal{K}\ @\rho\{\mathsf{a}' \circlearrowleft \mathcal{K}'\} & \stackrel{\text{def}}{=} & \begin{cases} \mu\mathsf{a}.\mathcal{K}\{\mathsf{a}' \circlearrowleft \mathcal{K}'\}\ @\rho & \mathsf{a} \neq \mathsf{a}' \\ \mu\mathsf{a}.\mathcal{K}\ @\rho & \mathsf{a} = \mathsf{a}' \end{cases} \\
\mathsf{a}@\rho\{\mathsf{a}' \circlearrowleft \mathcal{K}'\} & \stackrel{\text{def}}{=} & \begin{cases} \mathsf{a}@\rho & \mathsf{a} \neq \mathsf{a}' \\ \mu\mathsf{a}.\mathcal{K}'\ @\rho & \mathsf{a} = \mathsf{a}' \end{cases}
\end{array}
$$

Figure 6.10: Recursive Type Unfolding

variable is used, one constraint is computed by the $twoVals$ function and merged to the main constraint set Σ, and $twoVals(\beta) \stackrel{\text{def}}{=} \{0 \leq_{\mathtt{s}} \beta \leq_{\mathtt{s}} 1\}$. Our type system sometimes also generates constraints in the form of $\nu \geq_{\mathtt{s}} 0$ and $\nu \leq_{\mathtt{s}} 1$. The former can be rewritten as $\nu =_{\mathtt{s}} \alpha$ for fresh α and the latter is equivalent to $1 - \nu \geq_{\mathtt{s}} 0$. Constraint $\nu_1 =_{\mathtt{s}} \nu_2$ is equivalent to $\nu_1 - \nu_2 =_{\mathtt{s}} 0$.

The algorithm for finding whether nonnegative solutions exist to a system of linear diophantine equations is decidable [Sch98]. As a result, the judgment $\vdash_{\mathtt{G}} C : Ct$ is also decidable since the type rules are deterministic modulo choice of fresh variables. So, the question of whether the program C is typeable with program signature Ct is decidable. Efficient ways of solving linear diophantine equations exist. For instance, the equations can be solved incrementally [CD94] to avoid solving a large set of equations all at once. Following [CD94], all constraints of the first loaded class can be reduced to just one – the one that represents the solution (which can be pre-computed at compile time), and every time a class is loaded, it only needs to solve the constraints associated with that class, together with the solution from the previous step (one constraint).

6.2.4 Connection and Pedigree Relativization

The **connect** expression is typechecked by (T-Connect). In Sec. 5.3, we mentioned such an expression establishes a connection if the pair of connectors "match up". Here "match up" is a type property with no

run-time overhead. It is defined in in Fig. 6.9 as (Def-ConnectorMatch). Judgment $\rho \vdash \langle \mathcal{I}_1; \mathcal{E}_1 \rangle \stackrel{\Sigma}{\Rightarrow} \langle \mathcal{I}_2; \mathcal{E}_2 \rangle$ says connector signature $\langle \mathcal{I}_1; \mathcal{E}_1 \rangle$ of one objectage matches connector signature $\langle \mathcal{I}_2; \mathcal{E}_2 \rangle$ of the other objectage under the constraints of Σ, if the second objectage is the ρ of the first one. The intuition behind matching is that the **import**s from one side must be satisfied by the **export**s on the other side, and the method signature must also match via standard covariant and contravariant subtyping, as shown in (Def-MethodMatch).

As pedigrees are always relative to the objectage they are declared in, pedigree relativization is needed for connector matching of two objectages. Relativization is captured by the $relativize$ function. $relativize(\rho_1, \rho_2) = \rho_3, \Sigma$ means a pedigree ρ_1 in one objectage is pedigree ρ_3 in the other objectage if the first objectage is the ρ_2 of the second one, with the constraints of Σ. We also define a function $relativizem(\rho, [\rho_1, \ldots \rho_n])$, which is a composition of the original $relativize$ function. It relativizes pedigree ρ to an objectage of pedigree ρ_1, and then from there relativizes the resulting pedigree to an objectage of its pedigree ρ_2, and so on. With function $relativize$ defined, the $wfcons$ function defines how a type is converted from one objectage to another, and the $wfcons$ function defines the constraints. Type conversion is pedigree relativization for pedigree types and isomorphic transformation otherwise. To facilitate the soundness proof, type conversion is always defined in the direction of value passing, *i.e.* we consistently convert a type being the contravariant of the **import** and covariant of the **export**. (Def-ConnectorMatch) and (Def-MethodMatch) implicitly use the intuitive fact that if one objectage is $(\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2}$ of the other, the second objectage is $(\textbf{parent})^{\nu_2}(\textbf{child})^{\nu_1}$ of the first.

Note that *convert* is not defined for connection handle types. This is important since the lack of a definition in effect prevents a connection handle from being passed from one objectage to another. The reason why this matters was explained in Sec. 5.3. When two connectors match against each other, one party has to *convert* the types declared in it (*i.e.* relativize the pedigrees) to the other, so that both parties are "on the same page" when comparing pedigrees.

Definition $\mathcal{K}\{\mathsf{a} \circlearrowleft \mathcal{K}'\}$ in (T-Connect) unfolds recursive types: all pedigree type occurrences of the abbreviated form $\mathsf{a}@\rho$ within $\mathcal{K}$ are unfolded to the μ form $\mu\mathsf{a}.\mathcal{K}' @\rho$. This definition is given in Fig. 6.10. Unfolding recursive types to equate the abbreviated form and the μ form is standard. In our calculus, it allows (Def-ConnectorMatch) to match a pair of pedigree types, even if one of them is in the abbreviated form.

$$
\begin{array}{rcll}
\langle \mathcal{K}; \mathcal{M}; \mathcal{F} \rangle[\sigma]_{\mathtt{sig}} & \stackrel{\text{def}}{=} & \langle \mathcal{K}[\sigma]_{\mathtt{cl}}; \mathcal{M}[\sigma]_{\mathtt{ml}}; \mathcal{F}[\sigma]_{\mathtt{fl}} \rangle & \\
\mathcal{K}[\sigma]_{\mathtt{cl}} & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{k} \mapsto \langle \mathcal{I}[\sigma]_{\mathtt{iel}}; \mathcal{E}[\sigma]_{\mathtt{iel}} \rangle} & \text{if } \mathcal{K} = \overrightarrow{\mathsf{k} \mapsto \langle \mathcal{I}; \mathcal{E} \rangle} \\
\mathcal{I}[\sigma]_{\mathtt{iel}} & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{m} \mapsto (\tau[\sigma]_{\mathtt{t}} \to \tau'[\sigma]_{\mathtt{t}})} & \text{if } \mathcal{I} = \overrightarrow{\mathsf{m} \mapsto (\tau \to \tau')} \\
\mathcal{E}[\sigma]_{\mathtt{iel}} & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{m} \mapsto (\tau[\sigma]_{\mathtt{t}} \to \tau'[\sigma]_{\mathtt{t}})} & \text{if } \mathcal{E} = \overrightarrow{\mathsf{m} \mapsto (\tau \to \tau')} \\
\mathcal{M}[\sigma]_{\mathtt{ml}} & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{m} \mapsto (\tau[\sigma]_{\mathtt{t}} \to \tau'[\sigma]_{\mathtt{t}})} & \text{if } \mathcal{M} = \overrightarrow{\mathsf{m} \mapsto (\tau \to \tau')} \\
\mathcal{F}[\sigma]_{\mathtt{fl}} & \stackrel{\text{def}}{=} & \overrightarrow{\mathsf{f} \mapsto \tau[\sigma]_{\mathtt{t}}} & \text{if } \mathcal{F} = \overrightarrow{\mathsf{f} \mapsto \tau} \\
 & & & \\
\bot[\sigma]_{\mathtt{t}} & \stackrel{\text{def}}{=} & \bot & \\
\mathsf{k}[\sigma]_{\mathtt{t}} & \stackrel{\text{def}}{=} & \mathsf{k} & \\
\textbf{unit}[\sigma]_{\mathtt{t}} & \stackrel{\text{def}}{=} & \textbf{unit} & \\
\textbf{int}[\sigma]_{\mathtt{t}} & \stackrel{\text{def}}{=} & \textbf{int} & \\
\mu\mathsf{a}.\mathcal{K}\ @\rho[\sigma]_{\mathtt{t}} & \stackrel{\text{def}}{=} & \mu\mathsf{a}.(\mathcal{K}[\sigma]_{\mathtt{cl}})\ @(\rho[\sigma]_{\mathtt{p}}) & \\
\mathsf{a}@\rho[\sigma]_{\mathtt{t}} & \stackrel{\text{def}}{=} & \mathsf{a}@(\rho[\sigma]_{\mathtt{p}}) & \\
 & & & \\
\rho[\sigma]_{\mathtt{p}} & \stackrel{\text{def}}{=} & (\textbf{parent})^{\nu_1[\sigma]_{\mathtt{l}}}(\textbf{child})^{\nu_2[\sigma]_{\mathtt{l}}} & \text{if } \rho = (\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2} \\
(\nu + \nu')[\sigma]_{\mathtt{l}} & \stackrel{\text{def}}{=} & \nu[\sigma]_{\mathtt{l}} + \nu'[\sigma]_{\mathtt{l}} & \\
\alpha[\sigma]_{\mathtt{l}} & \stackrel{\text{def}}{=} & \begin{cases} \alpha' & \sigma(\langle \alpha; \beta \rangle) = \langle \alpha'; \beta' \rangle \\ \alpha & \text{otherwise} \end{cases} & \\
\beta[\sigma]_{\mathtt{l}} & \stackrel{\text{def}}{=} & \begin{cases} \beta' & \sigma(\langle \alpha; \beta \rangle) = \langle \alpha'; \beta' \rangle \\ \beta & \text{otherwise} \end{cases} & \\
\mathsf{w}[\sigma]_{\mathtt{l}} & \stackrel{\text{def}}{=} & \mathsf{w} & \\
\mathsf{z}[\sigma]_{\mathtt{l}} & \stackrel{\text{def}}{=} & \mathsf{z} & \\
 & & & \\
\Sigma[\sigma]_{\mathtt{cons}} & \stackrel{\text{def}}{=} & \overline{\nu[\sigma]_{\mathtt{l}} =_{\mathtt{s}} 0} & \text{if } \Sigma = \overline{\nu =_{\mathtt{s}} 0}
\end{array}
$$

Figure 6.11: Pedigree Type Variable Substitution

Unfolding of $\mathcal{K}_1$ is unnecessary: it is easy to see the types on signatures in $\mathcal{C}$ are already unfolded, thanks to the Ψ definition.

6.2.5 Polymorphic Instantiation and Constraint Merging

(T-Create) says that if programmers do not give a pedigree to the instantiated objectage, it will freshly generate a positive level for it. Note that it is not allowed to have a newly instantiated objectage with a pedigree of negative level being 0. These pedigrees are singleton pedigrees, such as **parent**. The singleton property would not be preserved if programmers could freely instantiate objectages of such pedigrees.

We first give a definition for substitution. Since all type variables show up in pairs in our calculus,

a general form of substitution mapping is defined as follows:

$$\sigma \ ::= \ \overrightarrow{\langle\alpha;\beta\rangle \mapsto \langle\alpha';\beta'\rangle}$$

For a $\sigma = [\langle\alpha_1;\beta_1\rangle \mapsto \langle\alpha'_1;\beta'_1\rangle, \ldots, \langle\alpha_n;\beta_n\rangle \mapsto \langle\alpha'_n;\beta'_n\rangle]$. Substitution of α_1 with α'_1, $\ldots$, α_n with α'_n, β_1 with β'_1, $\ldots$, β_n with β'_n is then denoted $-[\sigma]$ for "$-$" being either a type, a classage signature, or a constraint set. Precisely, Fig. 6.11 defines the pedigree type variable substitution function for a number of type related data structures. $-[\sigma]_{\texttt{sig}}$ defines substitution over a classage signature (with $\forall$ bindings removed), $-[\sigma]_{\texttt{cl}}$ over a connector signature list, $-[\sigma]_{\texttt{iel}}$ over an import/export signature list, $-[\sigma]_{\texttt{ml}}$ over a local method signature list, $-[\sigma]_{\texttt{fl}}$ over a field signature list, $-[\sigma]_{\texttt{t}}$ over a type, $-[\sigma]_{\texttt{p}}$ over a pedigree expression, $-[\sigma]_{\texttt{l}}$ over a level expression, and $-[\sigma]_{\texttt{cons}}$ over a constraint set. When no confusion arises, we also use $-[\sigma]$ to represent substitution over $-$, where $-$ can be any of the aforementioned data structures.

The rest of the rule deals with type variable instantiation. The related technique – polymorphic type inference for object-oriented languages – is a well-studied area; relevant approaches include [EST95, Age96, WS01]. The technique being used here is closest to [EST95], where let-polymorphism is used in combination with type inference of object types. A perfect alignment with let-polymorphism would have been drawn, if it were not for the presence of mutually recursive classes. The latter greatly complicates the typing rules. To see why, let us start with the standard case of no recursive class definitions. In this case a program can be viewed as a nested let-expression:

let `ClassageA` = { ... } **in** `ClassageB` = { ... `x` = **create child** `ClassageA`; ... }

A constraint-based typechecker would proceed as follows. It first typechecks `ClassageA`, and suppose it has signature

$$\forall\langle\alpha_1;\beta_1\rangle, \ldots \forall\langle\alpha_n;\beta_n\rangle.\langle\mathcal{K}_A;\mathcal{M}_A;\mathcal{F}_A\rangle$$

$\vdash_{\texttt{cls}}$ computes constraints Σ_A. When `ClassageB` is typechecked, expression **create child** `ClassageA` is encountered, and will be assigned with type

$$\langle\mathcal{K}_A;\mathcal{M}_A;\mathcal{F}_A\rangle[\langle\alpha_1;\beta_1\rangle \mapsto \langle\alpha'_1;\beta'_1\rangle, \ldots, \langle\alpha_n;\beta_n\rangle \mapsto \langle\alpha'_n;\beta'_n\rangle]$$

where α'_1, $\ldots$, α'_n, β'_1, $\ldots$, β'_n are fresh. Constraints

(T-Merge)

$$\frac{P = [\langle \mathsf{a}_1; \rho_1; \sigma_1\rangle, \ldots, \langle \mathsf{a}_m; \rho_m; \sigma_m\rangle] \qquad \forall i \in \{1, \ldots, m\}.\ \mathsf{a}_i \neq \mathsf{a} \qquad G(\mathsf{a}) = \langle \Sigma; \Pi\rangle \quad \Pi = \{\langle \mathsf{a}'_1; \rho'_1; \sigma'_1\rangle, \ldots, \langle \mathsf{a}'_n; \rho'_n; \sigma'_n\rangle\} \qquad \forall j \in \{1, \ldots, n\}.\ G, P \rhd \langle \mathsf{a}; \rho; \sigma\rangle \vdash \langle \mathsf{a}'_i; \rho'_i; \sigma'_i\rangle \backslash \Sigma_i}{G, P \vdash_{\mathtt{cons}} \langle \mathsf{a}; \rho; \sigma\rangle \backslash \Sigma[\sigma] \cup \Sigma_1 \cup \cdots \cup \Sigma_n}$$

(T-Merge-Recursive)

$$\frac{\begin{array}{c} P = [\langle \mathsf{a}_1; \rho_1; \sigma_1\rangle, \ldots, \langle \mathsf{a}_m; \rho_m; \sigma_m\rangle] \\ \mathsf{a} = \mathsf{a}_p, p \in \{1, \ldots, m\} \qquad \sigma_p = [\langle \alpha_1; \beta_1\rangle \mapsto \langle \alpha'_1; \beta'_1\rangle, \ldots, \langle \alpha_u; \beta_u\rangle \mapsto \langle \alpha'_u; \beta'_u\rangle] \\ \sigma = [\langle \alpha_1; \beta_1\rangle \mapsto \langle \alpha''_1; \beta''_1\rangle, \ldots, \langle \alpha_u; \beta_u\rangle \mapsto \langle \alpha''_u; \beta''_u\rangle] \\ \forall i \in \{1, \ldots, u\} \\ \rho''_i = (\textbf{parent})^{\alpha''_i} (\textbf{child})^{\beta''_i} \\ \rho'''_i = (\textbf{parent})^{\alpha'''_i} (\textbf{child})^{\beta'''_i} \qquad relativizem(\rho''_i, [\rho_m, \rho_{m-1}, \ldots, \rho_{p+1}]) = \rho'''_i, \Sigma_i \end{array}}{G, P \vdash_{\mathtt{cons}} \langle \mathsf{a}; \rho; \sigma\rangle \backslash \bigcup_{i \in \{1\ldots,u\}} \{\alpha'_i =_{\mathtt{s}} \alpha'''_i\} \cup \{\beta'_i =_{\mathtt{s}} \beta'''_i\} \cup \Sigma_i}$$

G	::=	$\overrightarrow{\mathsf{a} \mapsto \langle \Sigma; \Pi\rangle}$	*classage summaries*
P	::=	$\overrightarrow{\langle \mathsf{a}; \rho; \sigma\rangle}$	*instantiation path*

Figure 6.12: Constraint Merging

$$\Sigma'_A = \Sigma_A[\langle \alpha_1; \beta_1\rangle \mapsto \langle \alpha'_1; \beta'_1\rangle, \ldots, \langle \alpha_n; \beta_n\rangle \mapsto \langle \alpha'_n; \beta'_n\rangle]$$

will be merged with the rest of the constraints collected from the body of `ClassageB`. `ClassageB` is typechecked to have its type and constraints.

This technique however does not work well for the general case of object-oriented languages, as recursive class references are sometimes needed:

`ClassageA` = { ... y = **create sibling** `ClassageB`; ... }

`ClassageB` = { ... x = **create child** `ClassageA`; ... }

The key issue here is when `ClassageA` is typechecked, `ClassageB` has not been typechecked yet, so the constraints associated with it, Σ_B, are unknown at that point.

Our polymorphic type inference strategy can be viewed as extending the idea let-polymorphism to recursive programs. Inference of polymorphic recursion has been extensively studied, see *e.g.* [Hen93]; we present one particular approach which works well in the context of mutually recursive classes. As we

explained in Sec. 6.2.3, typechecking occurs in two passes. We now explain the two passes with regard to polymorphic instantiations.

The first pass is the per-classage typechecking $\vdash_{\texttt{cls}}$. When expression **create sibling** `ClassageB` is encountered by (T-Create), all level type variables of `ClassageB` are chosen fresh as in the case of let-polymorphism, and the program is typechecked in the analogous way. However, the constraints Σ_B are not merged in since they are not yet known. Our system processes Σ_B lazily: the type rule (T-Create) adds an entry to the *instantiation record set* (Π in the type rules, defined in Fig. 6.6). In our example, an entry $\langle$`ClassageB`; **sibling**; $\sigma_B\rangle$ is added into Π, where σ_B maps the parameter list of the signature for `ClassageB` to the new fresh type variables. After the first pass, each classage is typechecked, with pedigree constraints collected in Σ, and Π indicating the constraints that need to be merged lazily. The information is represented by data structure called *summaries* (G in Fig. 6.12).

In the second pass, defined by $\vdash_{\texttt{cons}}$ in Fig. 6.12 and used by (T-Global), the typechecker starts from the bootstrapping classage `Main`, and checks its Π to lazily merge constraints, with substitution performed in the same flavor as the earlier example when Σ'_A was merged. Judgment $G, P \vdash_{\texttt{cons}} \langle \texttt{a}; \rho; \sigma\rangle \backslash \Sigma$ means Σ is the merged constraints of per-classage constraints of `a`, together with those that need to be merged lazily. This process is propagated through the dependency chain of Π and is defined by the (T-Merge) rule of Fig. 6.12. When there is a cycle in the Π dependency chain, the constraints must be merged via rule (T-Merge-Recursive) in the figure to avoid infinite looping. Data structure P in the rules is used to track the path on the dependency chain of Π from that of `Main` to the node representing the constraint set to be merged. For every step that the constraints of a particular classage are to be merged, the rules check whether that classage has already shown up in P. If so, a cyclic Π dependency chain has formed. Rather than further merging the constraints, (T-Merge-Recursive) simply makes sure the fresh type variables generated for typing the recurrent classage instantiation is related to those generated for typing the previous classage instantiation. This is achieved through pedigree relativization via $relativizem$.

6.2.6 The Modularity of our Type System

Our system typechecks each classage modularly via the $\vdash_{\texttt{cls}}$ definition, and collects constraints on pedigree variables. All type errors excepting inter-classage pedigree inconsistencies – such as the **import** on one side not being satisfied by the **export** on the other for connections, or locally assigning a **sibling** variable to a **child** – can be immediately reported. Inter-classage pedigree consistency is guaranteed by merging constraint sets obtained from each modular classage typechecking, via the $\vdash_{\texttt{cons}}$ definition. At first glance, this might appear to be a weakness of the system, since pedigree consistency cannot be checked modularly. In fact this is a feature of the language: note that we could have required all objectage types in connector method signatures to have explicit pedigrees specified, whereupon the system would trivially satisfy the modularity property of pedigree consistency checking. The reason we decided not to follow that approach is our concern for programmer productivity: the need for many annotations would greatly distract programmers. We believe there is a fundamental tradeoff in object encapsulation systems between the choice of modular systems requiring many programmer declarations which will never enter common programming practice, and inference systems not completely modular, as we propose, which stand a chance of entering common programming practice. Also observe that modularity in OO languages is never absolute, and whether a system is modular largely depends on how the term "modularity" is defined: typechecking one C++ class requires the presence of others as class names are used as object types; dynamic class loading in Java involves non-trivial loading constraint solving.

Though not a focus of this formalism, our system is able to work correctly in the context of dynamic class loading. What it implies is the constraint merging would happen incrementally at run time. This is nothing new, for example the JVM maintains a dynamic set of "loader constraints" on types [LB98]. The incremental algorithm for solving linear diophantine equations [CD94] also works well in this situation.

6.3 Operational Semantics

We define the form of the operational semantics here, with a focus on describing the data structures that will be used for stating formal properties. The reduction rules are defined in Fig. 6.14 and Fig. 6.15.

Δ		::=	$\langle H;\, W;\, R\rangle$	*configuration*
H		::=	$\overrightarrow{o \mapsto \langle \mathsf{a};\, \sigma;\, S\rangle}$	*heap*
W		::=	$\overrightarrow{c \mapsto itrw}$	*connection store*
$itrw$		::=	$\langle o_1;\, \mathsf{k}_1;\, o_2;\, \mathsf{k}_2\rangle$	*connection store entry*
R		::=	$\overrightarrow{\langle o_1;\, o_2\rangle \mapsto (\textbf{parent})^{\theta}(\textbf{child})}$	*pedigree relation store*
θ		::=	$\alpha \mid \mathsf{w}$	*positive pedigree in store*
S		::=	$\overrightarrow{\mathsf{f} \mapsto v}$	*field store*
v	$\in \mathbb{V}$	::=	$() \mid$ *const* $\mid o \mid c^{side} \mid$ **null**	*value*
exd	$\in \mathbb{EXD}$	::=	$e \mid v \mid$ **exception** $\mid$ $\mathbf{in}_{side}(o, exd)$	*extended expression*
o	$\in \mathbb{RID}$			*objectage reference*
c	$\in \mathbb{CID}$			*connection ID*
side		::=	$\mathtt{A} \mid \mathtt{P}$	*active/passive side*
$\mathbf{E}$	$\in \mathbb{EVC}$	::=	$\bullet$	
		$\mid$	**connect E with** k >> k'	
		$\mid$	f :=**E** $\mid$ ::$\mathsf{m}(\mathbf{E})$	
		$\mid$	**E**->$\mathsf{m}(e)$ $\mid$ v->$\mathsf{m}(\mathbf{E})$	
		$\mid$	k::$\mathsf{m}(\mathbf{E})$	
		$\mid$	$\mathbf{E}; e$	

Figure 6.13: Runtime Data Structure Definitions

(R-Context)

$$\frac{\Delta, exd \xrightarrow{C,\, o} \Delta', exd'}{\Delta, \mathbf{E}[\,exd\,] \xrightarrow{C,\, o} \Delta', \mathbf{E}[\,exd'\,]}$$

(R-Create)

$$\frac{o_2 \text{ fresh} \qquad \rho = \begin{cases} (\textbf{parent})^{\alpha}(\textbf{child}) & \text{if } Ped = \epsilon, \alpha \text{ fresh} \\ Ped & Ped = (\textbf{parent})^{\mathsf{w}}(\textbf{child}) \end{cases} \qquad \vdash_{\mathsf{G}} C : Ct \qquad \mathcal{C} = \Psi(Ct) \qquad \mathcal{C}(\mathsf{a}) = \forall\langle\alpha_1;\beta_1\rangle,\ldots,\forall\langle\alpha_n;\beta_n\rangle.\langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle \quad \alpha'_1,\ldots,\alpha'_n,\beta'_1,\ldots,\beta'_n \text{ fresh} \qquad \sigma = [\langle\alpha_1;\beta_1\rangle \mapsto \langle\alpha'_1;\beta'_1\rangle,\ldots,\langle\alpha_n;\beta_n\rangle \mapsto \langle\alpha'_n;\beta'_n\rangle] \qquad H' = H \uplus (o_2 \mapsto \langle \mathsf{a};\, \sigma;\, \biguplus_{\mathsf{f}\in \mathrm{Dom}(F)} (\mathsf{f} \mapsto \textbf{null})\rangle) \qquad R' = R \uplus (\langle o_2;\, o_1\rangle \mapsto \rho)}{\langle H;\, W;\, R\rangle, \textbf{create } Ped\ \mathsf{a} \xrightarrow{C,\, o_1} \langle H';\, W;\, R'\rangle, o_2}$$

(R-Connect)

$$\frac{c \text{ fresh}}{\langle H;\, W;\, R\rangle, \textbf{connect } o_2 \textbf{ with } \mathsf{k}_1 \text{ >> } \mathsf{k}_2 \xrightarrow{C,\, o_1} \langle H;\, W \uplus (c \mapsto \langle o_1;\mathsf{k}_1;\, o_2;\mathsf{k}_2\rangle);\, R\rangle, c^{\mathtt{A}}}$$

Figure 6.14: Reduction Rules: Part 1

(R-Get)

$$\frac{\Delta = \langle H; W; R\rangle \qquad H(o) = \langle \mathsf{a}; \sigma; S\rangle \qquad S(\mathsf{f}) \neq \textbf{null}}{\Delta, \mathsf{f} \xrightarrow{C,\ o} \Delta, S(\mathsf{f})}$$

(R-Set)

$$\frac{H(o) = \langle \mathsf{a}; \sigma; S\rangle \qquad S' = S\{\mathsf{f} \mapsto v\} \qquad \mathsf{f} \in \mathrm{Dom}(S)}{\langle H; W; R\rangle, \mathsf{f} := v \xrightarrow{C,\ o} \langle H\{o \mapsto \langle \mathsf{a}; \sigma; S'\rangle\}; W; R\rangle, v}$$

(R-LocalM)

$$\frac{H(o) = \langle \mathsf{a}; \sigma; S\rangle \qquad C(\mathsf{a}) = \langle K; M; F\rangle \qquad M(\mathsf{m}) = \lambda x.e}{\Delta, ::\mathsf{m}(v) \xrightarrow{C,\ o} \Delta, e\{v/x\}}$$

(R-HandleM)

$$\frac{\begin{array}{c}\Delta = \langle H; W; R\rangle \qquad W(c) = \begin{cases}\langle o_1; \mathsf{k}_1; o_2; \mathsf{k}_2\rangle & \text{if } \textit{side} = \mathtt{A} \\ \langle o_2; \mathsf{k}_2; o_1; \mathsf{k}_1\rangle & \text{if } \textit{side} = \mathtt{P}\end{cases} \\ H(o_i) = \langle \mathsf{a}_i; \sigma_i; S_i\rangle, \forall i \in \{1,2\} \qquad C(\mathsf{a}_i) = \langle K_i; M_i; F_i\rangle, \forall i \in \{1,2\} \\ K_i(\mathsf{k}_i) = \langle I_i; E_i\rangle, \forall i \in \{1,2\} \qquad E_i(\mathsf{m}) = \lambda x.e_i, \forall i \in \{1,2\} \qquad \textit{side'} = \begin{cases}\mathtt{A} & \text{if } \textit{side} = \mathtt{P} \\ \mathtt{P} & \text{if } \textit{side} = \mathtt{A}\end{cases}\end{array}}{\Delta, c^{\textit{side}}\text{->}\mathsf{m}(v) \xrightarrow{C,\ o} \begin{cases}\Delta, \textbf{in}_{\textit{side'}}(o_2, e_2\{v/x\}\{c^{\textit{side'}}/\textbf{current}\}) & \text{if } \mathsf{m} \in \mathrm{Dom}(I_1) \\ \Delta, e_1\{v/x\}\{c^{\textit{side}}/\textbf{current}\} & \text{if } \mathsf{m} \in \mathrm{Dom}(E_1)\end{cases}}$$

(R-Forall)

$$\frac{\Delta = \langle H; W; R\rangle \qquad [c_1 \mapsto \textit{itrw}_1, \ldots, c_m \mapsto \textit{itrw}_m] = W \mid {}^{c \mapsto \textit{itrw}}_{\textit{itrw} = \langle o; \mathsf{k}; o'; \mathsf{k}'\rangle} \qquad [c_{m+1} \mapsto \textit{itrw}_{m+1}, \ldots, c_n \mapsto \textit{itrw}_n] = W \mid {}^{c \mapsto \textit{itrw}}_{\textit{itrw} = \langle o'; \mathsf{k}'; o; \mathsf{k}\rangle}}{\Delta, \textbf{forall}(x : \mathsf{k})\{e\} \xrightarrow{C,\ o} \Delta, e\{c^{\mathtt{A}}{}_1/x\}; \ldots; e\{c^{\mathtt{A}}{}_m/x\}; e\{c^{\mathtt{P}}{}_{m+1}/x\}; \ldots; e\{c^{\mathtt{P}}{}_n/x\}}$$

(R-In-1)

$$\frac{\Delta, exd \xrightarrow{C,\ o'} \Delta', exd' \qquad exd' \neq \textbf{exception}}{\Delta, \textbf{in}_{\textit{side}}(o', exd) \xrightarrow{C,\ o} \Delta', \textbf{in}_{\textit{side}}(o', exd')}$$

(R-In-2)

$$\Delta, \textbf{in}_{\textit{side}}(o', v) \xrightarrow{C,\ o} \Delta, v$$

(R-Sequence)

$$\Delta, v; e \xrightarrow{C,\ o} \Delta, e$$

(R-Self)

$$\Delta, \textbf{this} \xrightarrow{C,\ o} \Delta, o$$

(R-Exception-Get)

$$\frac{\Delta = \langle H; W; R\rangle \qquad H(o) = \langle \mathsf{a}; \sigma; S\rangle \qquad S(\mathsf{f}) = \textbf{null}}{\Delta, \mathsf{f} \xrightarrow{C,\ o} \Delta, \textbf{exception}}$$

(R-Exception-In)

$$\frac{\Delta, exd \xrightarrow{C,\ o'} \Delta', \textbf{exception}}{\Delta, \textbf{in}_{\textit{side}}(o', exd) \xrightarrow{C,\ o} \Delta', \textbf{exception}}$$

Figure 6.15: Reduction Rules: Part 2

(R-Boot)

$$\frac{o \text{ fresh} \qquad \Delta = \langle H; [\,]; [\,]\rangle \qquad H = o \mapsto \langle \texttt{main}; [\,]; \biguplus_{\mathsf{f}\in \mathrm{Dom}(F)} (\mathsf{f} \mapsto \textbf{null})\rangle \qquad C(\texttt{main}) = \langle K; M; F\rangle \qquad K(\texttt{mainc}) = \langle \emptyset; E\rangle \qquad E(\texttt{mainm}) = \lambda x.e}{C \xrightarrow{\texttt{init}} \langle \Delta; o; e\rangle}$$

Figure 6.16: Bootstrapping

$$rel(R, o', o) \stackrel{\text{def}}{=} \rho \text{ if } relativizem((\textbf{parent})^0(\textbf{child})^0, [\rho_0, \rho_1, \ldots, \rho_{p-1}]) = \rho, \Sigma,\ [o_0, o_1, \ldots, o_p] \text{ is an } o' - o \text{ path on instantiation tree } R,\ relOne(R, o_i, o_{i+1}) = \rho_i, i \in \{0, \ldots, p-1\}$$

$$relOne(R, o', o) \stackrel{\text{def}}{=} \begin{cases} (\textbf{parent})^\theta(\textbf{child}) & R(\langle o'; o\rangle) = (\textbf{parent})^\theta(\textbf{child}) \\ (\textbf{parent})(\textbf{child})^\theta & R(\langle o; o'\rangle) = (\textbf{parent})^\theta(\textbf{child}) \end{cases}$$

Figure 6.17: Definition: the rel function

As shown in Fig. 6.13, values (of set $\mathbb{V}$) are either a connection handle in the form of c^{side} (denoting a connection with ID c. We will explain the *side* notation shortly), an objectage reference (with ID o), the standard Java-**null** like value **null**, or primitive data. Expressions are extended to include values, **exception**, and a closure expression only used by the reduction system to model method invocation: $\mathbf{in}_{side}(o, exd)$ means exd is to be evaluated with regard to objectage o; meta-variable *side* denotes on which side of the connection the expression exd is located in. It can either be on the connection initiating side (the "active" side, where $side = \mathtt{A}$), or on the side being connected (the "passive" side, where $side = \mathtt{P}$). Such a distinction does not affect proving soundness, but is helpful in constructing a decidable type-checking process for run-time expressions.

Runtime configuration Δ records the mutable state of the execution. It is composed of a standard heap (H), a store to record all connections (W), and a store to record how objectages are related by pedigree relations (R). An entry $o \mapsto \langle \mathsf{a}; \sigma; S \rangle$ in the heap says that an objectage o is instantiated from a classage named a, and has the mutable states recorded in S. Auxiliary structure σ is only used by the proof, and it keeps track of the type variable instantiation for each objectage. An entry $c \mapsto \langle o_1; \mathsf{k}_1; o_2; \mathsf{k}_2 \rangle$ in the connection store says a connection with ID c is connecting the connector k_1 of objectage o_1 and the connector k_2 of objectage o_2. R is another structure only used for proving the correctness of the type system and hence does not affect reduction. Every time a **create** expression is evaluated, the pedigree of the instantiated objectage relative to the instantiating objectage is recorded in R, and we are going to show the dynamic pedigree relations will indeed correspond to what pedigree qualifiers try to enforce statically. An entry $\langle o_1; o_2 \rangle \mapsto \rho$ in R says o_1 is ρ of o_2. Based on R, function $rel(R, o_1, o_2)$ produces the pedigree of o_1 relative to o_2 with constraints Σ. Intuitively, R forms a directed tree where the nodes are objectages (their IDs) and the edges signify how the objectages instantiate one another. Each edge can be imagined as being associated with a "weight", indicating the pedigree of the instantiated objectage relative to the instantiating objectage. The rel function thus simply computes the "weighted distance" between tree node o_1 and o_2. Readers can find this definition in Fig. 6.17. The definition makes use of several self-explicit definitions which we give below.

Definition 1 (Instantiation Graph)**.** *Let the set of labels LAB a subset of pedigree expressions each of which*

is in the form of (**parent**)$^{\theta}$(**child**). *Let the set of vertices* VER *a subset of* $\mathbb{RID}$. *Let the set of edges* EG *a subset of ordered pair of vertices over* $\mathbb{RID} \times \mathbb{RID}$. *Let the labelling function* lab *be a total function over* $EG \mapsto LAB$. *An instantiation graph is a graph defined as* $\langle VER; EG; lab \rangle$.

Definition 2 ($o - o'$ Up-Path)**.** *An* $o - o'$ *up-path in an instantiation graph* $\langle VER; EG; lab \rangle$ *is a finite sequence over* VER. *It has the form of* $[o_0, o_1, o_2, \ldots, o_p]$ *for some* $p \geq 0$, *where all elements are distinct,* $o_0 = o$, $o_p = o'$, *and* $\langle o_i; o_{i+1} \rangle \in EG$ *for any* $0 \leq i \leq p - 1$.

Definition 3 (Graph Representation of R)**.** *We define a helper function* $graphRep(R)$ *as*

$$graphRep(R) \stackrel{\text{def}}{=} \langle VER; \mathrm{Dom}(R); R \rangle$$

where $VER = \{o \mid \langle o_1; o_2 \rangle \in \mathrm{Dom}(R), o = o_1 \text{ or } o = o_2\}$.

Definition 4 (Instantiation Tree)**.** R *is an instantiation tree iff* $graphRep(R) = \langle VER; EG; lab \rangle$, *and*

- *the underlying undirected unlabeled graph induced by* $\langle VER; EG \rangle$ *is a connected, acyclic graph.*
- *a unique vertex* $o' \in VER$, *called the root, exists such that for every vertex* $o \in VER$ *there is an* $o - o'$ *up-path.*

Definition 5 ($o - o'$ Path)**.** *An* $o - o'$ *path in* R *is defined to be* $[o_0, o_1, \ldots, o_q, o'_{r-1}, o'_{r-2}, \ldots, o'_0]$, *if all elements are distinct, there exists some* o'' *s. t.* $[o_0, o_1, \ldots, o_q]$ *is an* $o - o''$ *up-path in* $graphRep(R)$ *and* $[o'_0, o'_1, \ldots, o'_r]$ *is an* $o' - o''$ *up-path in* $graphRep(R)$.

The reduction relation is $\Delta, exd \xrightarrow{C,\, o} \Delta', exd'$, meaning exd is one-step reduced to exd' in objectage o, the runtime configuration changes from Δ to Δ', and C, o remain unchanged over reductions. Multi-step reduction $\Delta, exd \xrightarrow{C,\, o}_* \Delta', exd'$ is the transitive closure of one-step reduction. $\langle \Delta; exd \rangle \Uparrow_{C,o}$ means the reduction diverges. The bootstrapping process is modeled by reduction $C \xrightarrow{\texttt{init}} \langle \Delta; o; exd \rangle$, which prepares the initial values for these data structures. The bootstrapping rule is defined in Fig. 6.16. A configuration $\langle \Delta; o; exd \rangle$ is said to be *attainable* from program C, denoted as $C \xrightarrow{\clubsuit} \langle \Delta; o; exd \rangle$ iff $C \xrightarrow{\texttt{init}} \langle \Delta_0; o; exd_0 \rangle$ and $\Delta_0, exd_0 \xrightarrow{C,\, o}_* \Delta, exd$.

$$
\begin{array}{rcl}
()\{exd/se\} & \stackrel{\text{def}}{=} & () \\
x\{exd/se\} & \stackrel{\text{def}}{=} & \begin{cases} exd & se = x \\ x & \text{otherwise} \end{cases} \\
\mathit{const}\{exd/se\} & \stackrel{\text{def}}{=} & \mathit{const} \\
e; e'\{exd/se\} & \stackrel{\text{def}}{=} & e\{exd/se\}; e'\{exd/se\} \\
\textbf{this}\{exd/se\} & \stackrel{\text{def}}{=} & \textbf{this} \\
\textbf{create } \mathit{Ped}\ \mathsf{a}\{exd/se\} & \stackrel{\text{def}}{=} & \textbf{create } \mathit{Ped}\ \mathsf{a} \\
\textbf{connect } e \textbf{ with } \mathsf{k}\ \texttt{>>}\ \mathsf{k}'\{exd/se\} & \stackrel{\text{def}}{=} & \textbf{connect } e\{exd/se\} \textbf{ with } \mathsf{k}\ \texttt{>>}\ \mathsf{k}' \\
e\texttt{->}\mathsf{m}(e')\{exd/se\} & \stackrel{\text{def}}{=} & e\{exd/se\}\texttt{->}\mathsf{m}(e'\{exd/se\}) \\
\textbf{current}\{exd/se\} & \stackrel{\text{def}}{=} & \begin{cases} exd & se = \textbf{current} \\ \textbf{current} & \text{otherwise} \end{cases} \\
\textbf{forall}(x : \mathsf{k})\{e\}\{exd/se\} & \stackrel{\text{def}}{=} & \begin{cases} \textbf{forall}(x : \mathsf{k})\{e\{exd/se\}\} & se \neq x \\ \textbf{forall}(x : \mathsf{k})\{e\} & \text{otherwise} \end{cases} \\
\texttt{::}\mathsf{m}(e)\{exd/se\} & \stackrel{\text{def}}{=} & \texttt{::}\mathsf{m}(e\{exd/se\}) \\
\mathsf{f}\{exd/se\} & \stackrel{\text{def}}{=} & \mathsf{f} \\
\mathsf{f}\texttt{:=}e\{exd/se\} & \stackrel{\text{def}}{=} & \mathsf{f}\texttt{:=}(e\{exd/se\})
\end{array}
$$

Figure 6.18: Expression Substitution

Fig. 6.18 defines the substitution over programmer-defined expressions. Function $e\{exd/se\}$ computes an expression in $\mathbb{EXD}$ which is identical to e except every occurrence of se is replaced by exd. In the definition, metavariable $e \in \mathbb{EXP}$, metavariable $exd \in \mathbb{EXD}$, and metavariable *se* is either x or **current**.

6.4 Theoretical Properties

We now establish the main properties of our calculus. Rigorous proofs are given in the Appdendix.

Theorem 1 (Type Soundness)**.** *If* $\vdash_{\mathtt{G}} C : Ct$ *and* $C \xrightarrow{\texttt{init}} \langle \Delta; o; exd \rangle$*, then either* $\langle \Delta; exd \rangle \Uparrow_{C,o}$*, or* $\Delta, exd \xrightarrow{C,\ o}_{*} \Delta', v$ *for some* $\Delta', v \in \mathbb{V}$*, or* $\Delta, exd \xrightarrow{C,\ o}_{*} \Delta',$ **exception**.

This theorem states that the execution of statically typed programs either diverges, computes to a value, or throws exceptions. In this calculus, exceptions are thrown when an uninitialized field is accessed. The theorem is established via lemmas of subject reduction, progress, and the bootstrapping process leads into a well-typed initial configuration.

We now state several important theorems about how pedigree types enforce pedigrees at runtime. Theorem 2 states that if a programmer puts a pedigree qualifier, say **sibling**, on a field of a classage, then at run time that field will always hold a reference which is an **sibling** of the current objectage. Theorem 3 says any objectage reference on the heap must refer to some objectage which can be named by a well-formed pedigree qualifier. No objectage can hold a reference to the internal representation of another objectage, since that relationship cannot be expressed by a well-formed pedigree type. Theorem 4 says any connection handle stored on the heap must be stored in the field of some objectage the connection is established on; no third party can obtain a connection handle.

Theorem 2 (Shape Enforcement)**.** *If* $\vdash_{\mathtt{G}} C : Ct$*, for all* $\mathtt{a} \in \mathrm{Dom}(Ct)$*,* $Ct(\mathtt{a}) = \langle Kt; Mt, Ft\rangle$*, for all* $(\mathtt{f} \mapsto t) \propto Ft$

$$t = (\textbf{parent})^{\mathtt{w}}(\textbf{child})^{\mathtt{z}}\ \mathtt{a}' \implies (rel(R, \mathcal{S}(\mathtt{f}), o) = \rho) \wedge (subPed(\rho, (\textbf{parent})^{\mathtt{w}}(\textbf{child})^{\mathtt{z}})\ \textit{consistent})$$

for all $o \in \mathrm{Dom}(H)$*,* $\Delta = \langle H;\ W; R\rangle$*,* $H(o) = \langle \mathtt{a}; \sigma; \mathcal{S}\rangle$*, and* $C \xrightarrow{\clubsuit} \langle \Delta; o'; exd\rangle$*,*

Theorem 3 (Objectage Alias Protection)**.** *If* $\vdash_{\mathtt{G}} C : Ct$ *and* $C \xrightarrow{\clubsuit} \langle \Delta; o'; exd\rangle$*, and* $\Delta = \langle H;\ W; R\rangle$*, then for all* $o \in \mathrm{Dom}(H)$*,* $H(o) = \langle \mathtt{a}; \sigma; S\rangle$*, and for all* $(\mathtt{f} \mapsto v) \propto S$

$$v \in \mathbb{RID} \implies \begin{cases} rel(R, v, o) = \rho \\ subPed(\rho, (\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2}) \cup \{\nu_1 \geq_{\mathtt{s}} 0\} \cup twoVals(\nu_2)\ \textit{consistent} \end{cases}$$

The previous property is analogous to the notion of deep ownership [Cla01] in ownership type systems.

Theorem 4 (Connection Handle Protection)**.** *If* $\vdash_{\mathtt{G}} C : Ct$ *and* $C \xrightarrow{\clubsuit} \langle \Delta; o'; exd\rangle$*, and* $\Delta = \langle H;\ W; R\rangle$*, then for all* $o \in \mathrm{Dom}(H)$*,* $H(o) = \langle \mathtt{a}; \sigma; S\rangle$*, and for all* $(\mathtt{f} \mapsto v) \propto S$

$$v = c^{side} \implies W(c) = \langle o_1; \mathtt{k}_1; o_2; \mathtt{k}_2\rangle \wedge o = \begin{cases} o_1 & \textit{if } side = \mathtt{A} \\ o_2 & \textit{if } side = \mathtt{P} \end{cases}.$$

6.5 Language Constructs Beyond the Formal Core

The "middleweight" syntax we gave in Chapter 4 (Fig. 4.3) contains a number of language constructs that have been left out from the formal core we have described so far. They are left out to avoid

distraction to the main features, and a potential of explosion of the proof. We now give a more rigorous formalization to these constructs.

6.5.1 Language Constructs Related to Code Interactions

A number of code fragment manipulation language constructs are left out from the core. They include top-level classage mixing and renaming, together with invocation of a mixer method. First, we provide a formalization-friendly syntax with these language constructs, and they are in sync with the "middleweight" top-level syntax. MX is added to the definition of classages, denoting a list of mixers which structurally is the same as the list of connectors.

CX	::=	$\overrightarrow{\mathsf{a} \mapsto ClsX}$	*classages with mixers*
$ClsX$	::=	$\langle MX; K; M; F \rangle$	*classage implementation with mixers*
	|	$\mathsf{a}'+\mathsf{a}''$ **with** $\overrightarrow{\mathsf{k}' \ >> \ \mathsf{k}''}$	
	|	$\mathsf{a}'+\mathsf{a}''$ **with** $\overrightarrow{\mathsf{k}' \ >> \ \mathsf{k}''}$ **as** k	
	|	a' **rename** $\mathsf{k} \ >> \ \mathsf{k}'$	
MX	::=	K	*mixers*

To connect with the formal core, we next show how such a syntax can be converted to the syntax of Fig. 6.1 via a pre-processing pass by the compiler. Observe that mixing is a compile-time interaction, and it makes sense that such interactions can be handled by the compiler statically. The pre-processing pass is defined by function $preprocess$ below:

As illustrated in Fig. 6.19, $preprocess()$ function takes a list of classages with mixers (represented by meta-variable CX) and produces a list of classages without mixers, the familiar form we used in defining the formal core (represented by meta-variable C). The function first "atomizes" the classage, and then remove the mixers from the "atomized" classage.

The process of atomization is captured by the $atomize$ function defined in Fig. 6.20. The definition might seem verbose, but the idea in fact is very straightforward. If the classage to be "atomized" is already an atomic classage ($ClsX = \langle MX; K; M; F \rangle$), the function is idempotent. If the classage in concern is the renaming of another classage ($ClsX = \mathsf{a}'$ **rename** $\mathsf{k} \ >> \ \mathsf{k}'$), the result is produced by first atomizing the other classage a', and then a simple substitution of all occurrences of k with k'. Notation $-\{\mathsf{k}'/\mathsf{k}\}$ is extended

$$
\begin{array}{rcl}
\mathit{preprocess}(CX) & \stackrel{\text{def}}{=} & \biguplus_{CX(\mathsf{a})=ClsX} \mathsf{a} \mapsto \mathit{demixer}(\mathit{atomize}(ClsX, CX)) \\
\mathit{demixer}(\langle MX; K; M; F\rangle) & \stackrel{\text{def}}{=} & \langle K; M \uplus \biguplus_{MX(\mathsf{k})=\langle I;E\rangle} \mathit{localize}(\mathsf{k}, \langle I, E\rangle); F\rangle \\
\mathit{localize}(\mathsf{k}, \langle I; E\rangle) & \stackrel{\text{def}}{=} & \biguplus_{E(\mathsf{m})=\lambda x.e} \mathit{localn}(\mathsf{k}, \mathsf{m}) \mapsto \lambda x.e \\
 & & \text{if } \mathrm{Dom}(I) \subseteq \mathrm{Dom}(E) \\
\mathit{refreshL}(\langle MX; K; M; F\rangle) & \stackrel{\text{def}}{=} & \langle MX; K; M; F\rangle\{\mathsf{m}'_1/\mathsf{m}_1\}\ldots\{\mathsf{m}'_u/\mathsf{m}_u\}\{\mathsf{f}'_1/\mathsf{f}_1\}\ldots\{\mathsf{f}'_v/\mathsf{f}_v\} \\
 & & \mathrm{Dom}(M) = \{\mathsf{m}_1, \ldots, \mathsf{m}_u\} \\
 & & \mathrm{Dom}(F) = \{\mathsf{f}_1, \ldots, \mathsf{f}_v\} \\
 & & \mathsf{m}'_1, \ldots, \mathsf{m}'_u, \mathsf{f}'_1, \ldots, \mathsf{f}'_v \text{ fresh} \\
\mathit{localn}(\mathsf{k}, \mathsf{m}) & \stackrel{\text{def}}{=} & \text{deterministically creates a local method name based on } \mathsf{k} \text{ and } \mathsf{m} \\
 & & \text{such as via hashing}
\end{array}
$$

Figure 6.19: Definition: the *preprocess* Function

to represent such a substitution. If the classage is a compound, there are two cases, depending on whether the mixers being mixed will be re-exported or not. If not ($ClsX = \mathsf{a}_1{+}\mathsf{a}_2$ **with** $\overrightarrow{\mathsf{k}_{(n)1} \texttt{ >> } \mathsf{k}_{(n)2}}$), the *atomize* function first atomizes the two classages forming the compound, *i.e.*, a_1 and a_2. Since these two classages might have accidentally identical local methods and fields, function *refreshL* is defined to α-convert these names so that no local name conflict is possible when the two classage implementations are merged ($M_1 \uplus M_2$ and $F_1 \uplus F_2$). The rest of the definition attempts to achieve two tasks: 1) How to handle mixed mixers 2) What the resulting connector list is. For 1), the idea is all methods in mixed mixers are converted to local methods, represented by the function *localize*, this function, defined in Fig. 6.19, first ensures all imports are satisfied by exports, and then comes up a unique name for each mixer export method via the deterministic function *localn*. The mixer list in the resulting atomized classage should have those mixed mixers removed (MX), and have the methods produced by *localize* added to the local method list. For 2), the connector list is merged according to the explanation in Sec. 4.5.2. Last, if the classage is a compound but the mixers being mixed will be re-exported ($ClsX = \mathsf{a}_1{+}\mathsf{a}_2$ **with** $\overrightarrow{\mathsf{k}_{(n)1} \texttt{ >> } \mathsf{k}_{(n)2} \textbf{ as } \mathsf{k}_n}$), the *atomize* function works almost identically as the previous case, except that rather than "localize" mixed mixers, they are added back to the mixer list with new names specified by **as** language construct. The process of removing the mixers from the "atomized" classage is merely a process of *localize*, as defined by the *demixer* function in Fig. 6.19.

$$
atomize(ClsX, CX) \stackrel{\text{def}}{=}
\begin{cases}
ClsX \\
\quad \text{if } ClsX = \langle MX; K; M; F \rangle \\
\\
atomize(CX(\mathsf{a}'), CX)\{\mathsf{k}'/\mathsf{k}\} \\
\quad \text{if } \; ClsX = \mathsf{a}' \textbf{ rename } \mathsf{k} \;>>\; \mathsf{k}' \\
\\
\langle MX; K; M_1 \uplus M_2 \uplus M; F_1 \uplus F_2 \rangle \\
\quad \text{if } \; ClsX = \mathsf{a}_1{+}\mathsf{a}_2 \textbf{ with } \overrightarrow{\mathsf{k}_{(n)1} \;>>\; \mathsf{k}_{(n)2}} \\
\qquad refreshL(atomize(CX(\mathsf{a}_1), CX)) = ClsX_1 \\
\qquad refreshL(atomize(CX(\mathsf{a}_2), CX)) = ClsX_2 \\
\qquad \mathsf{k}_1, \ldots, \mathsf{k}_n \text{ fresh} \\
\qquad ClsX_1\{\mathsf{k}_1/\mathsf{k}_{(1)1}\} \ldots \{\mathsf{k}_n/\mathsf{k}_{(n)1}\} = \langle MX_1; K_1; M_1; F_1 \rangle \\
\qquad ClsX_2\{\mathsf{k}_1/\mathsf{k}_{(1)2}\} \ldots \{\mathsf{k}_n/\mathsf{k}_{(n)2}\} = \langle MX_2; K_2; M_2; F_2 \rangle \\
\qquad MX = (MX_1 \backslash \{\mathsf{k}_1, \ldots \mathsf{k}_n\}) \uplus (MX_2 \backslash \{\mathsf{k}_1, \ldots \mathsf{k}_n\}) \\
\qquad M = \biguplus_{i=\{1,\ldots,n\}} localize(\mathsf{k}_i, \langle I_{(i)1} \uplus I_{(i)2}; E_{(i)1} \uplus E_{(i)2} \rangle) \\
\qquad \text{for } i \in \{1, \ldots, n\} \\
\qquad MX_1(\mathsf{k}_{(i)1}) = \langle I_{(i)1}; E_{(i)1} \rangle, MX_2(\mathsf{k}_{(i)2}) = \langle I_{(i)2}; E_{(i)2} \rangle \\
\qquad K(\mathsf{k}) = \begin{cases} \langle I_1 \rhd I_2; E_1 \uplus E_2 \rangle & \text{if } \mathsf{k} \in \mathrm{Dom}(K_1) \cap \mathrm{Dom}(K_2) \\ & \quad K_1(\mathsf{k}_1) = \langle I_1; E_1 \rangle \\ & \quad K_1(\mathsf{k}_2) = \langle I_2; E_2 \rangle \\ K_1(\mathsf{k}) & \text{otherwise if } \mathsf{k} \in \mathrm{Dom}(K_1) \\ K_2(\mathsf{k}) & \text{otherwise if } \mathsf{k} \in \mathrm{Dom}(K_2) \end{cases} \\
\\
\langle MX \uplus MX'; K; M_1 \uplus M_2; F_1 \uplus F_2 \rangle \\
\quad \text{if } \; ClsX = \mathsf{a}_1{+}\mathsf{a}_2 \textbf{ with } \overrightarrow{\mathsf{k}_{(n)1} \;>>\; \mathsf{k}_{(n)2} \textbf{ as } \mathsf{k}_n} \\
\qquad refreshL(atomize(CX(\mathsf{a}_1), CX)) = ClsX_1 \\
\qquad refreshL(atomize(CX(\mathsf{a}_2), CX)) = ClsX_2 \\
\qquad ClsX_1\{\mathsf{k}_1/\mathsf{k}_{(1)1}\} \ldots \{\mathsf{k}_n/\mathsf{k}_{(n)1}\} = \langle MX_1; K_1; M_1; F_1 \rangle \\
\qquad ClsX_2\{\mathsf{k}_1/\mathsf{k}_{(1)2}\} \ldots \{\mathsf{k}_n/\mathsf{k}_{(n)2}\} = \langle MX_2; K_2; M_2; F_2 \rangle \\
\qquad MX = (MX_1 \backslash \{\mathsf{k}_1, \ldots \mathsf{k}_n\}) \uplus (MX_2 \backslash \{\mathsf{k}_1, \ldots \mathsf{k}_n\}) \\
\qquad MX' = \biguplus_{i=\{1,\ldots,n\}} (\mathsf{k}_i \mapsto \langle I_{(i)1} \uplus I_{(i)2}; E_{(i)1} \uplus E_{(i)2} \rangle) \\
\qquad \text{for } i \in \{1, \ldots, n\} \\
\qquad MX_1(\mathsf{k}_{(i)1}) = \langle I_{(i)1}; E_{(i)1} \rangle, MX_2(\mathsf{k}_{(i)2}) = \langle I_{(i)2}; E_{(i)2} \rangle \\
\qquad K(\mathsf{k}) = \begin{cases} \langle I_1 \rhd I_2; E_1 \uplus E_2 \rangle & \text{if } \mathsf{k} \in \mathrm{Dom}(K_1) \cap \mathrm{Dom}(K_2) \\ & \quad K_1(\mathsf{k}_1) = \langle I_1; E_1 \rangle \\ & \quad K_1(\mathsf{k}_2) = \langle I_2; E_2 \rangle \\ K_1(\mathsf{k}) & \text{otherwise if } \mathsf{k} \in \mathrm{Dom}(K_1) \\ K_2(\mathsf{k}) & \text{otherwise if } \mathsf{k} \in \mathrm{Dom}(K_2) \end{cases}
\end{cases}
$$

Figure 6.20: Definition: the *atomize* Function

In general, what is defined here is a "code+code=code" operation that is well-explored by previous research on module systems, code fragment calculi, and mixin-like systems. One compact and rigorously proved calculus that is in spirit similar to the definition here is CMS [AZ02].

With the previous definitions defined, the semantics of the mixer method invocation can be trivially defined as a reduction to a local method invocation, as follows:

$$\text{(R-MixerM)}\quad \frac{\mathsf{m}' = \mathit{localn}(\mathsf{k}, \mathsf{m})}{\Delta, \mathsf{k}::\mathsf{m}(v) \xrightarrow{C,\, o} \Delta, ::\mathsf{m}'(v)}$$

6.5.2 Connection-Specific Fields

In Chapter 4, we discussed the language feature of connection-specific fields, as declared with the keyword **state** inside a connector. They specify mutable state generative for each connection, and only accessible for the objectage with that connection established on one end. We first extend the syntax with such fields represented in connectors, the expressions for read/write access for such fields, and the dynamic data structure of W, where each connection entry is associated with a pair of field stores, recording the connection-specific fields for both ends of the connection. Such syntactical extensions are defined as follows:

K	::=	$\overrightarrow{\mathsf{k} \mapsto \langle I; E; F\rangle}$	*connectors*
I, E, F		as defined in Fig. 6.1	*imports, exports, fields*
e	::=	$\cdots \mid e\texttt{->}\mathsf{f} \mid e\texttt{->}\mathsf{f}\texttt{:=}e'$	*expressions with connection-specific field access*
W	::=	$\overrightarrow{c \mapsto \langle o_1; \mathsf{k}_1; S_1; o_2; \mathsf{k}_2; S_2\rangle}$	*connection store*
S		as defined in Fig. 6.13	*field store*

Dynamic semantics for connection-specific field access is straightforward, as represented by (R-ConnGet) and (R-ConnSet). The **connect** expression is also slightly affected, since we need to allocate memory for these connection-specific field stores at connection time. They are represented by (R-ConnectWithConnField). Last, read access to an un-initialized field will lead to the standard **null** field

exception, illustrated by (R-Exception-ConnGet).

(R-ConnGet)

$$\frac{\Delta = \langle H;\, W;\, R\rangle \qquad W(c) = \begin{cases} \langle o;\mathsf{k};S;o';\mathsf{k}';S'\rangle & \text{if } \mathit{side} = \mathtt{A} \\ \langle o';\mathsf{k}';S';o;\mathsf{k};S\rangle & \text{if } \mathit{side} = \mathtt{P} \end{cases} \qquad S(\mathsf{f}) \neq \textbf{null}}{\Delta,\, c^{side}\texttt{->}\mathsf{f} \xrightarrow{C,\ o} \Delta, S(\mathsf{f})}$$

(R-ConnSet)

$$\frac{W_1(c) = \begin{cases} \langle o;\mathsf{k};S;o';\mathsf{k}';S'\rangle & \text{if } \mathit{side} = \mathtt{A} \\ \langle o';\mathsf{k}';S';o;\mathsf{k};S\rangle & \text{if } \mathit{side} = \mathtt{P} \end{cases} \qquad W_2 = \begin{cases} W_1\{c \mapsto \langle o;\mathsf{k};S\{\mathsf{f}\mapsto v\};o';\mathsf{k}';S'\rangle\} & \text{if } \mathit{side} = \mathtt{A} \\ W_1\{c \mapsto \langle o';\mathsf{k}';S';o;\mathsf{k};S\{\mathsf{f}\mapsto v\}\rangle\} & \text{if } \mathit{side} = \mathtt{P} \end{cases}}{\langle H;\, W_1;\, R\rangle, c\texttt{->}\mathsf{f}\texttt{:=}v \xrightarrow{C,\ o_0} \langle H;\, W_2;\, R\rangle, v}$$

(R-ConnectWithConnField)

$$\frac{H(o_i) = \langle \mathsf{a}_i;\sigma_i;S_i\rangle, \forall i \in \{1,2\} \qquad C(\mathsf{a}_i) = \langle K_i;M_i;F_i\rangle, \forall i \in \{1,2\} \qquad K_i(\mathsf{k}_i) = \langle I_i;E_i;F_i'\rangle, \forall i \in \{1,2\} \qquad S_i = \biguplus_{\mathsf{f}\in \mathrm{Dom}(F_i')} (\mathsf{f}\mapsto \textbf{null}) \qquad c \text{ fresh}}{\langle H;\, W;\, R\rangle, \textbf{connect } o_2 \textbf{ with } \mathsf{k}_1 \ \texttt{>>}\ \mathsf{k}_2 \xrightarrow{C,\ o_1} \langle H;\, W \uplus (c \mapsto \langle o_1;\mathsf{k}_1;S_1;o_2;\mathsf{k}_2;S_2\rangle);\, R\rangle, c^{\mathtt{A}}}$$

(R-Exception-ConnGet)

$$\frac{\Delta = \langle H;\, W;\, R\rangle \qquad W(c) = \begin{cases} \langle o;\mathsf{k};S;o';\mathsf{k}';S'\rangle & \text{if } \mathit{side} = \mathtt{A} \\ \langle o';\mathsf{k}';S';o;\mathsf{k};S\rangle & \text{if } \mathit{side} = \mathtt{P} \end{cases} \qquad S(\mathsf{f}) = \textbf{null}}{\Delta,\, c^{side}\texttt{->}\mathsf{f} \xrightarrow{C,\ o_0} \Delta, \textbf{exception}}$$

6.5.3 Disconnection

The "middleweight" syntax we defined in Fig. 4.3 allows programmers to "disconnect" a live connection by using expression **disconnect** e. We first extend the syntax with such an expression, and also the dynamic data structure of W, where each connection entry is associated with a *flag*, signifying whether the connection is live (denoted as **alive**) or not (denoted as **stale**). Such extensions are defined as follows:

e	::=	$\cdots \mid$ **disconnect** e	*expressions with disconnect*
W	::=	$\overrightarrow{c \mapsto \langle o_1;\mathsf{k}_1;o_2;\mathsf{k}_2;\mathit{flag}\rangle}$	*connection store*
flag	::=	**alive** | **stale**	*connection status*

Illustrated below, reducing the **disconnect** e expression is rigorously defined by (R-Disconnect). The argument must be a connection handle to be disconnected. The reduction simply changes the *flag* for

that connection entry to **stale**. Introducing such a feature will slightly change the reduction rules of several existing expressions. For (R-Connect), we should pre-set the new entry in the connection store to be **alive**, as represented by (R-ConnectDis) below. Rule (R-HandleM) will need to add a pre-condition to make sure the connection in concern is **alive**, and Rule (R-Forall) should only enumerate connection handles that are flagged as **alive**. We omit these two rules reflecting the change here. Finally, invoking a connection with a **stale** connection handle should throw an exception, as illustrated by (R-Exception-HandleM).

(R-Disconnect)

$$\frac{W(c) = \langle o_1; \mathsf{k}_1; o_2; \mathsf{k}_2; \mathit{flag} \rangle \qquad W' = W\{c \mapsto \langle o_1; \mathsf{k}_1; o_2; \mathsf{k}_2; \textbf{stale} \rangle\}}{\langle H;\, W;\, R \rangle, \textbf{disconnect}\ c^{side} \xrightarrow{C,\ o} \langle H;\, W';\, R \rangle, ()}$$

(R-ConnectDis)

$$\frac{c \text{ fresh}}{\langle H;\, W;\, R \rangle, \textbf{connect}\ o_2\ \textbf{with}\ \mathsf{k}_1\ \texttt{>>}\ \mathsf{k}_2 \xrightarrow{C,\ o_1} \langle H;\, W \uplus (c \mapsto \langle o_1; \mathsf{k}_1; o_2; \mathsf{k}_2; \textbf{alive} \rangle);\, R \rangle, c^{side}}$$

(R-Exception-HandleM)

$$\frac{\Delta = \langle H;\, W;\, R \rangle \qquad W(c) = \langle o_1; \mathsf{k}_1; o_2; \mathsf{k}_2; \textbf{stale} \rangle}{\Delta, c^{side}\texttt{->}\mathsf{m}(v) \xrightarrow{C,\ o} \Delta, \textbf{exception}}$$

6.5.4 Singleton Connectors

In Chapter 4, the "middleweight" syntax we defined (Fig. 4.3) allows programmers to declare connectors to be singletons, so that at most one connection can be alive on that connector at run time. Such an auxiliary construct is helpful for programming, because programmers can access a connector method via the connector name, rather than obtaining the connection handle first (given a connector name, the language guarantees there is at most one connection on that connector anyways). Supporting singleton connectors is very straightforward: all that is needed is for the language run-time to keep track of the number of connections alive on (singleton) connectors. To rigorously define this semantics, we first extend the run-time expression e with the $\mathsf{k}\texttt{::}\mathsf{m}(e')$ expression, representing invoking method m in singleton connector named k, with argument being e'. Note that this expression is by syntax identical to the mixer method access expression we defined in Sec. 6.5.1, but since the two can be easily differentiated by whether k is a mixer or a connector, no

ambiguity can arise. We also extend the connector syntax K to have a *kind* field to keep track of whether a connector is a singleton or not, as follows:

e	::=	$\cdots \mid \mathsf{k}::\mathsf{m}(e')$	*expressions with singleton connector method access*
K	::=	$\overrightarrow{\mathsf{k} \mapsto \langle I; E; \mathit{kind}\rangle}$	*connectors*
kind	::=	**singleton** | **generative**	*connector kind*

The reduction of the $\mathsf{k}::\mathsf{m}(e')$ expression is defined below by the rules of (R-SingletonM1) and (R-SingletonM2). The idea is to find out all connections currently alive to the connector k on objectage o. If their is only one alive, the expression is reduced to a connection handle invocation expression.

(R-SingletonM1)

$$\frac{\Delta = \langle H; W; R\rangle \qquad \{c\} = \{c_0 \mid W(c_0) = \langle o; \mathsf{k}; o'; \mathsf{k}'\rangle\}\emptyset = \{c_0 \mid W(c_0) = \langle o'; \mathsf{k}'; o; \mathsf{k}\rangle\}}{\Delta, \mathsf{k}::\mathsf{m}(v) \xrightarrow{C,\ o} \Delta, c^{\mathtt{A}}\text{->}\mathsf{m}(v)}$$

(R-SingletonM2)

$$\frac{\Delta = \langle H; W; R\rangle \qquad \emptyset = \{c_0 \mid W(c_0) = \langle o; \mathsf{k}; o'; \mathsf{k}'\rangle\}\{c\} = \{c_0 \mid W(c_0) = \langle o'; \mathsf{k}'; o; \mathsf{k}\rangle\}}{\Delta, \mathsf{k}::\mathsf{m}(v) \xrightarrow{C,\ o} \Delta, c^{\mathtt{P}}\text{->}\mathsf{m}(v)}$$

(R-ConnectSingle)

$$\frac{\begin{array}{c} H(o_i) = \langle \mathsf{a}_i; \sigma_i; S_i\rangle, \forall i \in \{1,2\} \\ C(\mathsf{a}_i) = \langle K_i; M_i; F_i\rangle, \forall i \in \{1,2\} \qquad K_i(\mathsf{k}_i) = \langle I_i; E_i; \mathit{kind}_i\rangle, \forall i \in \{1,2\} \\ c \text{ fresh} \qquad \mathit{kind}_1 = \textbf{singleton} \implies \{c_0 \mid W(c_0) = \langle o_1; \mathsf{k}_1; o; \mathsf{k}\rangle \text{ or } \langle o; \mathsf{k}; o_1; \mathsf{k}_1\rangle\} = \emptyset \\ \mathit{kind}_2 = \textbf{singleton} \implies \{c_0 \mid W(c_0) = \langle o_2; \mathsf{k}_2; o; \mathsf{k}\rangle \text{ or } \langle o; \mathsf{k}; o_2; \mathsf{k}_2\rangle\} = \emptyset \end{array}}{\langle H; W; R\rangle, \textbf{connect } o_2 \textbf{ with } \mathsf{k}_1 \ \text{>>}\ \mathsf{k}_2 \xrightarrow{C,\ o_1} \langle H; W \uplus (c \mapsto \langle o_1; \mathsf{k}_1; o_2; \mathsf{k}_2\rangle); R\rangle, c^{\mathtt{A}}}$$

(R-Exception-ConnectSingle)

$$\frac{\begin{array}{c} H(o_i) = \langle \mathsf{a}_i; \sigma_i; S_i\rangle, \forall i \in \{1,2\} \\ C(\mathsf{a}_i) = \langle K_i; M_i; F_i\rangle, \forall i \in \{1,2\} \qquad K_i(\mathsf{k}_i) = \langle I_i; E_i; \mathit{kind}_i\rangle, \forall i \in \{1,2\} \\ \mathit{kind}_1 = \textbf{singleton} \implies \{c_0 \mid W(c_0) = \langle o_1; \mathsf{k}_1; o; \mathsf{k}\rangle \text{ or } \langle o; \mathsf{k}; o_1; \mathsf{k}_1\rangle\} \neq \emptyset \\ \mathit{kind}_2 = \textbf{singleton} \implies \{c_0 \mid W(c_0) = \langle o_2; \mathsf{k}_2; o; \mathsf{k}\rangle \text{ or } \langle o; \mathsf{k}; o_2; \mathsf{k}_2\rangle\} \neq \emptyset \end{array}}{\langle H; W; R\rangle, \textbf{connect } o_2 \textbf{ with } \mathsf{k}_1 \ \text{>>}\ \mathsf{k}_2 \xrightarrow{C,\ o_1} \Delta, \textbf{exception}}$$

(R-Exception-SingletonM)

$$\frac{\{c \mid W(c) = \langle o; \mathsf{k}; o'; \mathsf{k}'\rangle \text{ or } \langle o'; \mathsf{k}'; o; \mathsf{k}\rangle\} = \emptyset}{\Delta, \mathsf{k}::\mathsf{m}(v) \xrightarrow{C,\ o} \Delta, \textbf{exception}}$$

Adding the feature of singleton connectors will affect how the **connect** expression is reduced. Specifically, trying to establish a connection to a singleton connector which already has a live connection

should throw an exception, and this is represented by (R-Exception-ConnectSingle). A pre-condition to make sure this exceptional case does not happen is added to the previous (R-Connect) rule, resulting in (R-ConnectSingle). The last rule, (R-Exception-SingletonM), defines the case where no connection is alive when the access happens. (Another abnormal case – multiple connections alive on the same connector – is not possible, since it is already precluded by (R-ConnectSingle)).

Chapter 7

Implementation

A prototype *Classages* compiler has been implemented using the Polyglot compiler framework [NCM03]. The extensibility of Polyglot allows programmers to reuse Java's existing features. The current implementation provides its own classage/objectage model and relies on Polyglot's base implementation of Java to provide useful language features such as arrays, exceptions and rich primitive types.

The implementation covers all language features introduced in the syntax (Sec. 4.3) except pedigree types, including translation to target Java code and the type system. All *Classages* code examples used in this paper can be successfully compiled. The current compiler is not equipped with libraries, and advanced features such as reflection are not included.

The compilation process consists of several standard passes including parsing, type-building, disambiguation, typechecking and Java code generation, performed by a series of visitor traversals on the AST. An important issue is *when* compound classages are "flattened-out" into atomic classages. Since mixing is a purely static interaction, the compiler could in theory perform flattening right after parsing. This approach however would lead to duplicate compilation: when a classage A is also used to form a compound B, a flattening of B early on would lead to the code inside A being typechecked twice. Our compiler hence does not flatten compound classages until the last pass, when the target code is produced. Compared with the specification of Sec. 4.5.2, the implementation is a bit more complicated due to the need for $\alpha-$conversion of local

fields and methods during flattening. The **classage** $cn = cn{+}cn$ **with** $\overrightarrow{n \texttt{ >> } n}$ syntax involves consumption of mixers; the current implementation handles it by giving the consumed mixers some internal names not overlapping with programmers' name space. The **classage** $cn = cn{+}cn$ **with** $\overrightarrow{n \texttt{ >> } n}$ **as** $\vec{n}$ syntax involves interface renaming.

7.1 The Translation

Each classage is translated into a top-level Java class, and its interaction interfaces are translated into Java inner classes, the use of which eases up implementation issues such as scoping. The Java top-level class contains factory methods for the creation of each inner class.

All top level classes contain a fields `cstore` to record live connections. Bookkeeping operations on these fields are defined in a common Java interface (named `SYS_CLASSAGE`) that all top-level classes implement. All inner classes contain one field named `other`: recording the other party communicating on the interaction interface. Recall connections happen between a *pair* of interaction interfaces. Bookkeeping operations for this field are defined in another common Java interface (named `SYS_I`) all inner classes implement. Inside each inner class, every **export** method is mapped to a **public** method, while every **import** method is implemented as a redirection to the method on `other` by the same name.

A **connect** `o` **with** `n1 >> n2` expression is translated to Java code instantiating the two inner classes representing `n1` and `n2` respectively, and updating the `other` fields of the two and the live connection stores. Without considering the subtyping intricacies discussed below in Sec. 7.2, the translation is:

```
x = this.factorymethod_n1();
y = o.factorymethod_n2();
x.other = y;
y.other = x;
this.cstore.add(y);
o.cstore.add(x);
```

7.2 Implementing Structural Subtyping in Java

One of the most challenging implementation issues is how *Classages*' structural subtyping system is to be implemented in Java, which uses nominal and not structural subtyping. Consider the following example, where B is a subtype of A, by structure:

```
classage A {
    connector Q {export void f(int x){...}}
}
classage B {
    connector Q {
        export void f(int x) {...}
        export void g(double x){...}
    }
    connector Z {...}
}
classage C {
    connector P {...}
    int m(A x) {
        P c = connect x with P >> Q;
        c->f(3);
    }
}
```

The tricky issue is the polymorphism supported by structural subtyping: local invocation `::m(b)` in C should typecheck, where b is an objectage of B. To make the same type-checkable *Classages* expression get through the Java typechecker, a naive implementation might just use casting. But Java's casting only allows casts up and down inheritance hierarchies, and casting one class to an unrelated class will fail. Some tricks can be attempted to build more nominal subtype relationships by introducing additional interfaces and implements relationships, but these approaches give incomplete results, especially considering Java does not support depth subtyping with its **interface** mechanism.

Our Solution We first state the guiding rules of our solution:

R1. Any objectage variable declaration is translated to a variable declaration with top type `SYS_CLASSAGE` in the target Java program. This includes variables in the form of method formal parameters, fields and

local variables.

R2. If a classage has an interaction interface named A, its implementing top-level Java class must implement a top-level one-method Java **interface** containing method **void** factorymethod_A().

R3. If an interaction interface contains an **export** method named m, its implementing Java inner class must implement a top-level one-method Java **interface** containing the signature of method m, where, as per **R1** above, all formal parameters of m with objectage types are replaced with the type SYS_CLASSAGE.

R1 is applied to ensure expressions involving the use of objectage subtyping should not be translated to target code rejected by Java's type checker. In the earlier example, if the formal parameter x of m is translated with a top type, one can imagine that invocation ::m(b) can always be translated as a Java-typecheckable expression, even when b is an objectage of type B.

Now consider the translation of **connect** x **with** P >> Q. In Sec. 7.1 we presented an incomplete translation, containing the Java statement y = o.factorymethod_n2(). Notice that now o according to **R1** has a top type, so the invocation does not typecheck in Java. One might be tempted to use downcasting, and fixing the statement in the form such as y = ((A)o).factorymethod_n2(). This approach does get around Java's static type checker, but at runtime, it will likely throw an exception due to invalid downcasting: although o is declared to have top type in the method of m, it can still be instantiated with an object with type, say B. As long as B is not nominally a subtype of A, the downcast will fail at runtime.

R2 is introduced to avoid runtime exceptions at downcasts. Using this rule, the same statement can be translated as y = ((I)o).factorymethod_n2(), where I is the one-method Java **interface** containing method factorymethod_n2. Since the name n2 is known locally, such a translation also does not depend on global information. Such a downcast always succeeds, because to ensure structural subtyping, any o must contain interaction interface n2, and hence must have implemented the same interface according to **R2**.

R3 is introduced in a similar vein as **R2**, only in this case on a different level: *Classages* objectage subtyping happens on two levels, with one level ensuring the subtype has at least as many as connectors as the supertype, and the other ensuring within the same connector, the methods also conform to the subtyping

constraint. **R3** is useful for the method level.

7.3 Toward a Production-Strength Language

The development of the *Classages* project follows a timeline common for a large number of language systems created from the research community: first the language semantics is rigorously defined and its properties proven (stage I); next a prototype is developed to allow programmers to write simple programs to test language features (stage II); last an efficient compiler with rich library support is developed (stage III). On this timeline, the *Classages* project is currently at the end of stage II. We now sketch several aspects of stage III development.

Efficient Implementation and Optimization As we have described earlier, the implementation here involves significant downcasting. As a result, the compiler will suffer from inefficiencies. This inefficiency results from the limitation of our choice of the target language (Java), as the latter does not have an object model in support of depth subtyping. A more efficient compiler can be developed by implementing the entire object model from scratch, *i.e.*, translating the *Classages* language constructs into non-OO target language constructs, and maintaining virtual tables manually. Efficient implementation and optimization along these lines are standard techniques for OO compiler construction.

Independent of the choice of target languages, the inclusion of some expressive language features in *Classages* inherently leads to some decisions being made dynamically. *Classages* structural subtyping will lead to dynamic lookup for method mn in expression $e\texttt{->}mn(e_1, \ldots, e_n)$, where a connection e is invoked. Program analysis techniques such as concrete class analysis [OPS92, PC94], CPA [Age96], and DCPA[WS01], can be used to optimize the program, so that a large number of dynamic lookup can be eliminated.

Library Support The current compiler does not have libraries for commonly reused code fragments. For language design projects that only extended existing languages with new language constructs, this does not appear to be a problem: for instance, for a language which is a Java extension, the compiler for the new

language can simply reuse the libraries developed for Java as a result of trivial backward compatibility. This however is not the case for our language. *Classages* has fundamentally changed the *object model* of that of Java's. If backward compatibility is to be supported in our case, the language will have to consider what objects are to be invoked by the dot syntax (Java syntax), what objects are to be connected first before invoking the connection handle (*Classages* syntax), and more importantly, if one *Classages* object is used as a Java object (and vice versa), how the language type system and run-time can function correctly. In the end, the solution is going to go beyond "merely an implementation" issue, but is instead similar in spirit to multi-language interoperability. If this track is to be followed, interesting sources to refer to include [GFF05, MF07].

A second approach is to develop *Classages*-specific libraries. One obvious way is to develop the libraries from scratch. For any mature compiler, this is the ultimate way to follow, but it could be time-consuming. An alternative solution is to port existing libraries such as Java libraries to *Classages*. How *Classages* can encode the main Java language constructs has been discussed and formally defined in Sec. 4.6.2. More care needs to be taken for transforming second-line Java constructs, such as inner classes and reflexion.

With the libraries developed, it will be interesting to conduct several medium/large-scale case studies on *Classages*.

Chapter 8

Extensions and Variations

In this chapter, we discuss several extensions and variations to make the *Classages* model even more expressive. They can be added to our core language system without difficulty.

8.1 Plugging and Pluggers

In addition to mixing and connections, a third form of interactions called *plugging* was described in [LS05]. Such an interaction is used for modeling whole-part relationships between objectages: An objectage can plug in a classage to its runtime, so that the run-time instance of the plugged-in classage becomes a "part" of the initiating objectage, the "whole". Plugging happens between a pair of interaction interfaces as well, the *plugger* of the initiating objectage and the mixer of the plugged-in classage. Structurally, the syntax of a plugger is identical to that of a connector or a mixer, only that the former can only be used for establishing plugging interactions.

Just like connection, plugging is another form of long-lived interaction. An an enhanced solution to Fig. 1.3 can be illustrated In Fig. 8.1, where the objectage `ProteinModel` is holding multiple pluggings of `Snapshot` instances, indicating multiple snapshots of the protein structures have been taken and recorded. Unlike connection interaction which happens between two objectages, a plugging interaction can be figuratively thought of as *a "part" being plugged into the object memory space of the objectage representing the*

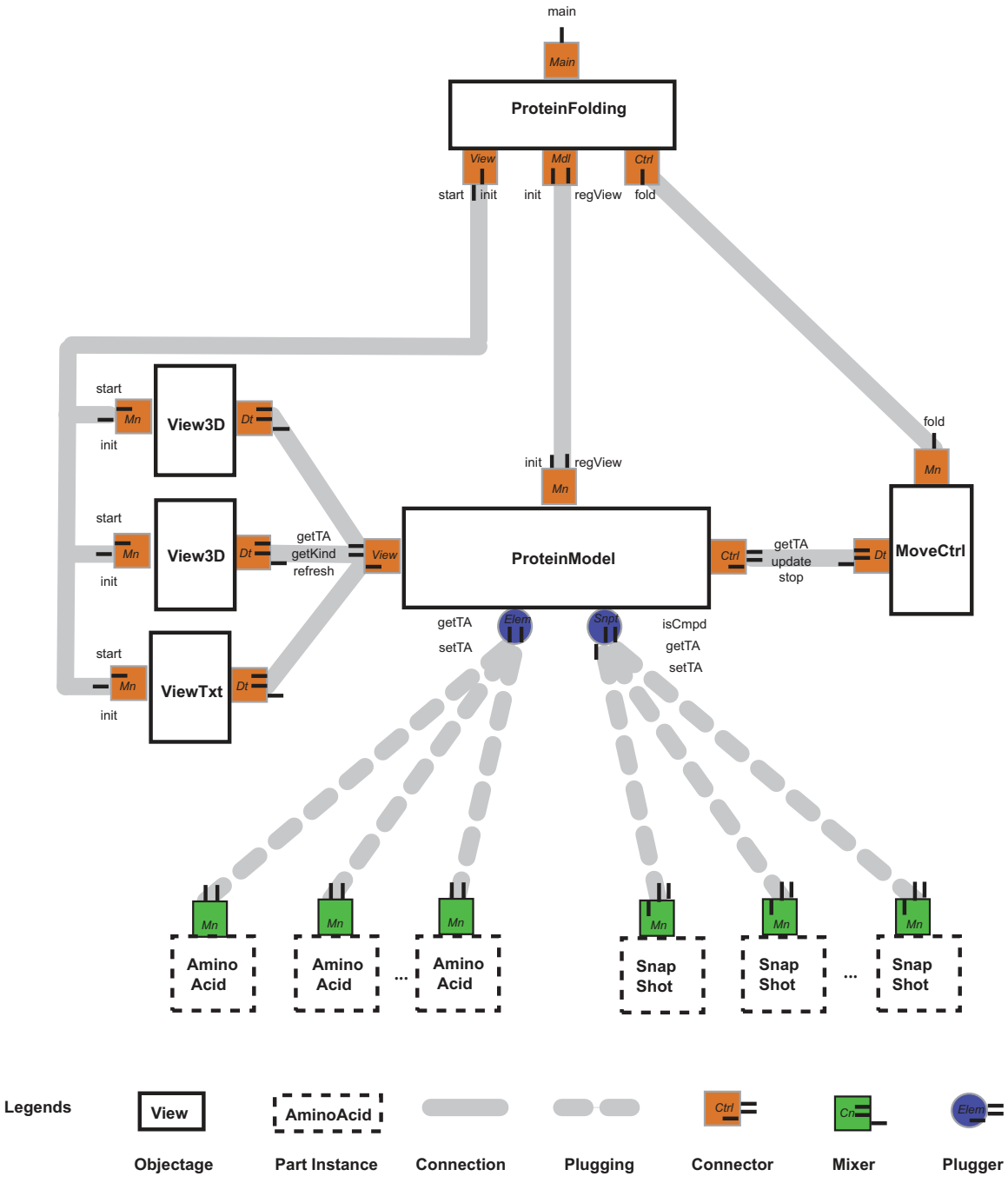


Figure 8.1: An Implementation of Fig. 1.1 in *Classages* with Plugging Interactions

"whole". From a memory management point of view, when the memory space for the whole is released, the part is also released, and from an access control point of view, outsiders of `ProteinModel` cannot have direct access to `Snapshot` instances. Pluggings give the programmer a strong tool for confining objects, and also can be looked on as a method for dynamically extending an already-created object. A plugging represents a more tightly-coupled relationship than a connection: it is analogous to a UML composition association, which constrains the part to not outlive the whole. Another implicit constraint in *Classages* is the part only comes into being as part of the whole—there is no such thing as an isolated part instance. Plugging happens on the *plugger* interface of the whole and the *mixer* interface of the part: the whole is a dynamic object, and at the time of plugging the part is but a code template to be instantiated. A plugging interaction between the the `Snpt` plugger of the `ProteinModel` objectage and the `Mn` mixer of the `Snapshot` classage is established at runtime by the following expression within `ProteinModel`:

```
p = plugin Snapshot with Snpt >> Mn;
```

The result of the **plugin** expression, `p`, is called a *plugging handle*, the first-class incarnation of a whole-part communication relationship. After plugging, the `ProteinModel` objectage can call `getTA` to query the torsion angles of the just-created snapshot, expressed by `p..getTA()` in *Classages*. Similarly, any `Snapshot` instance can check whether the data should be stored in compressed form by the code `isCmpd()`: the "whole" need not be explicitly mentioned, it is implicit that the `ProteinModel` being communicated is the one the `Snapshot` sits in. A plugee can itself have plugees, giving explicit language support for hierarchical containment relationships.

A large part of what is covered by plugging can be modeled by the introduction of pedigree types. For instance, an imprecise yet intuitive encoding of the previous **plugin** expression could be (if we reverse the kind of the `Snpt` and `Mn` interfaces to be connectors):

```
p = connect (create child Snapshot) with Snpt >> Mn;
```

The idea behind this encoding is that plugging can be roughly considered as instantiating an objectage being a **child** of the initiating objectage, and then establish an interaction between the two objectages. The property of confinement is the same: the objectage reference pointing to the `Snapshot` is still confined due to the

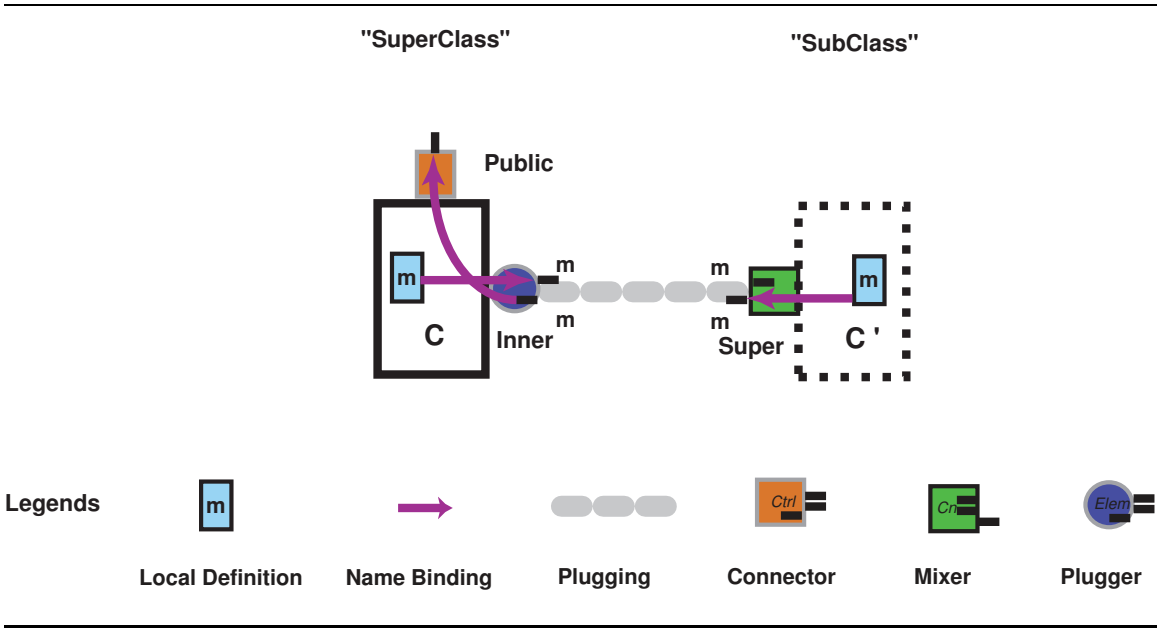


Figure 8.2: Dynamic Overriding with Plugging Interactions

pedigree type **child**, and the connection handle (in this case p) can not leak out the boundary of the initiating objectage either.

Does the introduction of pedigree types completely supersede the coverage of plugging? The answer is negative. There are at least two areas the design of plugging still has values of its own:

- Pluggers give a clear and declarative specification for how an objectage interacts with its "internal world". The separation of connectors and pluggers in fact reflects a topic at the center of philosophy for centuries: self and others. For a real-world analogy, connectors show how an objectage lives with its external world, *i.e.*, how it communicates with others. In the human world, it shows the "social" aspect of an individual. On the other hand, pluggers show how an objectage lives with its internal world, *i.e.*, how it is formed and how it evolves. In the human world, it shows the "introspective" aspect of an individual.

- Plugging makes dynamic overriding possible. So far we have only discussed static method overriding, the canonical form of overriding as found in Java and C++. With pluggers, it also elegantly supports a form of *dynamic overriding*, *i.e.* object methods being dynamically changed at runtime. Analogous

to what we have described in Fig. 4.4 for static overriding, we can achieve dynamic overriding as in Fig. 8.2. Note that here, *plugger* `Inner` of `C` here declares `m` as overridable, and `m` in connector `Public` is now defined as the method provided by this plugger. Similarly to the static overriding case, language semantics ensures anywhere in the code of `C` where `m` of `Inner` is referred to, the body of `m` is (1) the **export** `m` of `Inner` if plugger `Inner` is not plugged in with a plugee; and otherwise is (2) the **export** provided by the plugee. From the perspective of peer objectages, the instance of `C` has an ability to dynamically change the body of its public methods.

Since there is a significant overlapping with the design of plugging and the design of pedigree types, and yet they cannot strictly contain each other, one interesting question to ask is whether there is a possibility that they can live harmoniously in one language, with the expressiveness of both but no redundancy. We believe this is possible, and it in fact will lead to a design even one step further from what we are familiar with to be an object-oriented language: a language without an explicit instantiation expression such as the **create** expression of the current design. Note that the **plugin** expression is an implicit instantiation expression: for example, in the `Snapshot` example earlier, the instance of `Snapshot` is implicitly created before it is plugged into the run-time of `ProteinModel`.

8.2 Sharing

In Sec. 5.3, we discussed a solution of selective exposure via the **connect** expression. An optional mechanism – which we call sharing – can be designed to give the direct encapsulating party the power to share its encapsulated objectages to outsiders. The same scenario illustrated in Fig. 5.4 can be achieved by having the `Dialog d` evaluate

share `es >> b` **with** `MousePressed >> Listener`

Intuitively, a form of shared ownership is created here. The `Dialog` can be considered as the primary encapsulating party which performs delegation of sharing, while the `EventSource` is a secondary owner which can share the functionality of `Button`. In [Mit06], run-time analysis for real-world applications shows

more than 75% of ownership trees have shared ownership, and we feel there is thus a need to provide a direct programming abstraction for it.

The **share** expression is similar to the **connect** expression in that it gives `Dialog` the flexibility to freely choose the connector of its ownee to be exposed to the outsider. No reference to the ownee itself is given to the client.

Even though the language itself already supports selective exposure without **share** expression, including both the **connect** and **share** expressions in the same language has the benefit of supporting selective exposure with a relatively balanced power distribution between the owner and the ownee. The **share** expression is designed to give the owner the power to decide how the ownee should be exposed, while the **connect** expression is designed to allow the ownee itself to decide how it should expose itself. If the `Dialog` as an owner is not comfortable with having the `Button` talk to the `EventSource`, it should not pass an `EventSource` reference to `Button` to begin with.

The language abstract syntax with the **share** expression is defined as follows:

e	::=	$\cdots \mid$ **share** e `>>` e' **with** k `>>` k'	*expressions*

The typing rule can be defined as:

(T-Share)

$$\frac{\begin{array}{c}\Gamma,\ \mathcal{C} \vdash e : \mu\mathsf{a}_1.\mathcal{K}_1\ @\rho \backslash \Sigma, \Pi \qquad \Gamma,\ \mathcal{C} \vdash e' : \mu\mathsf{a}_2.\mathcal{K}_2\ @\textbf{child} \backslash \Sigma', \Pi' \\ \mathit{relativize}(\rho, \textbf{parent}) = \rho', \Sigma_{\mathtt{rel}} \qquad \mathcal{K}_1' = \mathcal{K}_1\{\mu\mathsf{a}_1.\mathcal{K}_1\ @\rho_x / \mathsf{a}_1 @\rho_x\} \\ \mathcal{K}_2' = \mathcal{K}_2\{\mu\mathsf{a}_2.\mathcal{K}_2\ @\rho_y / \mathsf{a}_2 @\rho_y\} \qquad \rho' \vdash \mathcal{K}_2'(\mathsf{k}_2) \overset{\Sigma''}{\Longrightarrow} \mathcal{K}_1'(\mathsf{k}_1) \qquad \Sigma_{\mathtt{share}} = \Sigma \cup \Sigma' \cup \Sigma'' \cup \Sigma_{\mathtt{rel}}\end{array}}{\Gamma,\ \mathcal{C} \vdash \textbf{share}\ e\ \texttt{>>}\ e'\ \textbf{with}\ \mathsf{k}_1\ \texttt{>>}\ \mathsf{k}_2 : \textbf{unit} \backslash \Sigma_{\mathtt{share}}, \Pi \cup \Pi'}$$

The reduction rule can be defined as:

(R-Share)

$$\frac{c \text{ fresh}}{\langle H;\, W;\, R\rangle,\ \textbf{share}\ o_1\ \texttt{>>}\ o_2\ \textbf{with}\ \mathsf{k}_1\ \texttt{>>}\ \mathsf{k}_2 \xrightarrow{C,\ o} \langle H;\, W \uplus (c \mapsto \langle o_2; \mathsf{k}_2; o_1; \mathsf{k}_1\rangle);\, R\rangle, ()}$$

The type system is expected to be sound with this extension.

8.3 User-Defined Level Constraints

Sometimes programmers may be interested in saying *e.g.*, "even though I don't care on what pedigrees these two object references have, the two pedigrees must be the same". It can be useful when an object is referring to objects known to be **sibling**'s among themselves, for instance forming an MVC pattern. Such constraints can be supported with almost no change to the formal system. Observe that the constraints our type system collects are exactly on pedigree levels. What is needed here is the same sort of constraints, only this time defined by programmers. Therefore, when typechecking a classage, user-defined level constraints (say $\Sigma_{\texttt{user}}$) can be merged with those generated by the type system. In (T-Classages), change $\Sigma \cup \Sigma'$ to $\Sigma \cup \Sigma' \cup \Sigma_{\texttt{user}}$.

8.4 Read-Only Exposure

Historically, solutions to address the problem of internal representation exposure – or alias protection in general – are very much related to read-only access [BNR01, MPH01]. The idea is that representation exposure is harmless if the outsider cannot mutate it. This assumption is indeed helpful in constructing sound type systems, but it makes some real-world programming idioms difficult to write. For instance in Sec. 5.3, the `Button`'s event-listening method `actionPerformed` should be able to mutate the state of the `Button` object.

In our system, whether read-only access should be granted or not is an orthogonal issue. If needed, read-only access in our system can be modeled by having the **export** methods of the connector on the representation objectage changed to be read-only. This modification can be added to (T-Connect) in a similar way. The typechecker should check whether the **connect** initiating party's connector has all **export**'s declared read-only. If not, a constraint is added to require the connected objectage to have a pedigree of **child** or **sibling**.

8.5 Un-Exposure

Observe that exposure via **connect** involves the establishment of connections between the representation and the party receiving the exposure. The flip side is to use **disconnect** expressions to remove the connections, so that exposure can be terminated dynamically. The trade-off is some connection handles might become *stale* if some removal of connection happens at run time, and a runtime check to avoid this is needed. The details of the **disconnect** expression have been discussed in Sec. 6.5.3.

Chapter 9

Conclusion

Only connect...

— epigraph of *Howards End* (1910), by E. M. Forster

Imagine all of today's computer applications – from your favorite XBox game to the Wall Street online trading system to the next Mars Rover controller – are developed in FORTRAN. That's likely to be how future citizens of our planet would react if all computer applications in 50 years were to be developed in C++, Java, or C#. Make no mistake: existing OO programming languages, just like FORTRAN, have made enormous contributions to modern technology. They themselves also evolve, going through the decades from Simula 67 to Smalltalk to C++ to Java and C#, with more language features included and better libraries developed. That being true, the underlying core object model seems to stay unchanged: objects interact merely through message passing. When the core object model was first designed, typical programs were not on the same magnitude in size and complexity as those today. Modern concepts of good software engineering were less pronounced. Software crisis due to software size and complexity was unheard-of. A simplistic interaction model for objects made sense. Decades later, at the same time as we marvel at the originality of the pioneers, we as language designers have to ask ourselves whether the seasoned object model is still our best choice for building software systems of the future that are likely to have billions of lines of code and complex structures of interaction. After all, the true spirit of our pioneers – those who designed Simula 67

for instance – is to be iconoclastic, provocative, and not too ready to settle for what we already have.

It is our hope that *Classages* is a step – though perhaps just a baby one – to fundamentally change the object model we have long taken for granted. The main take-away message for our readers is a philosophical one: to understand the complexity of a large system, we have to understand how different parts of the system interact, the *"matters in between"*. The key ideas of this dissertation – mixing, connection, and pedigree types – all reflect this philosophy: mixing helps to understand the "matters in between" code pieces; connection helps understand the "matters in between" run-time entities; pedigree types help understand the "matters in between" that shape the entire heap. One might disagree with some of the technical decisions we have made, or might challenge whether this is an OO language after all [1], but our ultimate hope is the philosophy of Interaction-Oriented Programming can be helpful in building next-generation programming languages in the scale-conscious era of software development.

9.1 Technical Contributions

This dissertation describes *Classages*, a type-safe, interaction-oriented, programming language with refined notions of inter-class and inter-object interactions and relationships. The language core is rigorously defined. Important formal properties are established and proved. A prototyped compiler is implemented.

First, *Classages* provides an interaction-oriented object model where classes and objects are fully encapsulated, with explicit interfaces for all interactions they might be involved in. *Classages* provide first-class support for the fundamental interactions within the object world: static composition of classes (mixing) and inter-runtime communication of objects (connection). Despite the fundamentally different nature of the two, they are unified under the same rationale: each interaction involves two parties; each party defines explicit interfaces on what it contributes to the interaction and what it expects from the interaction, and a successful interaction is established if the interfaces from both parties match each other. Despite the wide range of topics *Classages* touches upon, this unified view can both keep the formal language core small for

[1] We are often asked, "with so many changes to the guts of objects, are *Classages* an OO language after all?" We leave this categorization question to future historians (if any of them still remembers this language). The bottom line is, without languages like Smalltalk and Java, without previous work from the active research community of OO languages, there wouldn't have been the existence of *Classages*.

reasoning, and help programmers learn the language quickly. An encoding of Java's OO model in *Classages* is provided, showing how standard paradigms are supported. A sound type system with flexible notions of subtyping is designed to statically check the "correctness" of the interactions, despite that the interactions might likely happen at run time.

Second, Pedigree Types present an intuitive and powerful type system to shape the entire object heap into a hierarchy of decomposition and encapsulation. The intuitive nomenclature of human genealogy is applied to a type system to navigate hierarchies. Familiar genealogical pedigrees such as **parent**, **child**, **sibling** are unified into one compact general form. Inherent aspects of human genealogy trees are captured, including pedigree relativization and pedigree subsumption. Implicit parametric polymorphism is introduced to capture the intuitive notion of an "I don't care"-kind of polymorphic pedigree. The resulting system retains the expressiveness of ownership type systems, at the same time presenting a simple and intuitive programming interface to end users. A novel constraint-based type system is constructed to enforce the consistency of pedigree invariants, where constraints are linear systems of integer arithmetic equations, and constraint solving is finding a solution to these linear equations. Formal properties of pedigree invariant enforcement, and alias protection are established. A novel and simple solution to selective exposure is provided. Our mechanism does not give out aliases to the owned objects but selectively grants access. In conclusion, Pedigree Types put the power of an explicit formalism for defining and constraining the heap hierarchy into the hands of programmers, and give programmers a greater awareness of, and thus control of, the heap reference structure.

9.2 Future Work

The current implementation does not place efficiency and library support as the primary goal; improvement in this regard will become an immediate task, as we have described in Sec. 7.3.

It will be interesting to investigate how the interaction-oriented view can impact the design of advanced language features known to be useful for software development. For instance, interaction is fundamentally related to security: if there was no interaction, there would be no concern of security. In the

Classages model, all potential interactions an object can be involved in must go through explicitly defined interaction interfaces. Thus, these interfaces are the only possible "doors" to access the object and no "back-doors" exist. This is good news for security: if we can carefully restrict the access to these interaction interfaces, object-level security can be achieved. Furthermore, by studying the *interventions* of different interactions – using techniques such as information/control flow analysis – security properties of an application composed of many objects can also be reasoned. Another direction is to explore how an object model like *Classages* can impact concurrent programming. We would like to extend the current model, so that each objectage can conceptually serve as a concurrent unit, and the interactions between them can be the units for interleaving. Some inherent features of the *Classages* model, such as its first-class support for interactions, callbacks, will be conducive to designing a concurrent model with good properties of atomicity and determinism.

Pedigree Types open up several interesting possibilities we would like to explore in the future. First, when no pedigree qualifiers are declared by programmers, our type system becomes a *de facto* program analysis algorithm. The system in this paper only considers whether the constraints have solutions or not (*i.e.* the program typechecks or not), and it will be interesting to explore how the shape of solutions (vectors in linear equation solution space) is related to the shape of reference structures. Secondly, our system currently enforces object encapsulation, a crucial issue for object-oriented languages. In other contexts (*e.g.* high-level software architecture enforcement), references like **nephew** could be allowed. We speculate that only minor changes are needed to construct such a system purely for shape enforcement, such as removing the $twoVals$ constraints currently added to the constraint set.

9.3 Putting *Classages* in the Broader Context

We now put this dissertation in the broader context, giving a brief account on the research areas *not* exactly what *Classages* are targeted at (object-oriented language design), but with shared "spirit".

Modules and Components Module and component systems are well known as a means for giving clear interfaces for linking of code. Examples of module systems are ML modules [Mac84, Ler00], Modula-3 [Nel91], Units [FF98]. Research on software components [Szy02] is diverse. That includes a number of industrial component systems (such as COM+ [Mic], Javabeans [SUN], CORBA CCM [Obj02]) and component languages, such as Cells [RS02].

Classages can be viewed as cross-over thinking from module systems and components to the level of objects. Our design of interfaces might be structurally similar with systems in this category, but there is a fundamental semantic difference due to the fundamental difference between modules and objects. In this regard, *Classages* is closest to a component language we designed, *Assemblages* [LS04]. In *Assemblages*, fundamental interactions on the module/component level are directly supported in a programming language. *Assemblages* not only directly support static linking – which is standard for module/component systems – but also support 1) dynamic linking as a form of interaction between a running module and a piece of module code,with the module code being *plugged* into the runtime module; and 2) loosely coupled and potentially distributed communications as a form of interaction between two running modules.

Software Architecture Representation In the software engineering community, several architectural description languages [AG97, MRT99] also focus on modeling interactions between different software parts, but their focus is on modeling high-level software architectural constraints, not designing fine-grained language constructs such as classes and objects directly used by programmers. Many languages in this category do not have machine-executable semantics. ArchJava [ASCN03] is a programming language that can enforce architectural constraints using type systems. In ArchJava, the main question to answer is “what are best language constructs for expressing software architecture and promoting the reasoning over it?”. This is a very different from our main goal of designing an object model, and is bound to lead to very different design decisions, such as whether the language should support dynamic dispatch, subtype polymorphism, and alias protection.

Other Applications of Interaction-Oriented Programming We have explored interaction-oriented language design in contexts other than designing object models, including software deployment and version control [LS06], multi-core programming [LLS07], decentralized secure programming [LS02], and sensor-network programming [LST04]. These topics are beyond the scope of this dissertation.

Interactions Beyond Computer Science Interaction is by no means a novel concept in the vast realm of philosophy, and it has deep-rooted and well-established theoretical foundations in many branches of humanities. In this sense, this dissertation is a rather shallow and oversimplified interpretation of those ideas into the sub-domain of computer programming. Several tiny dots on this vast canvass – by no means comprehensive and complete – are summarized below. Readers should especially pay attention to how the parallel question of "what is it to be an object (in the programming language sense)?" is addressed in these other domains.

Interaction plays an important role in the development of philosophy, especially in the 20^{th} century. As an example, it can be reflected in Heidegger's notion of *Dasein* in his towering work of *Being and Time* [Hei62]. In that work, a central issue is to work out the question of the sense of being. Dasein, as a form of "being" or "existence", is by nature invested in social interaction, because one of the most fundamental ways of understand Being is through relationships. In Heidegger's system, Dasein is always a being engaged in the world. The fundamental mode of being is not that of a subject or of the objective but of the coherence of being-in-the-world, and all things stand in a relation to all other things. As another example, Wittgenstein's approach to "meaning" is inter-subjective and social in nature in its *Philosophical Investigations* [Wit01]. He is critical of the idea that individuals have privileged access to their own mental states; for him, mental phenomena grows out of our linguistic, social behavior. Language plays a central role in Wittgenstein's idea, but it only makes sense in the social context of practices. Even though individuals can certainly have private thoughts but these thoughts do not make any sense without a background of social interaction that they must have already participated in.

Symbolic Interactionism is a major sociological perspective with deep impact for microsociology and social psychology. With pioneers such as J. Dewey, Charles H. Cooley, and especially G. H. Mead, the theory of symbolic interactionism promotes the idea that mind and self are not part of the innate human

equipment but arise through social interaction, such as communication with others using symbols. The term itself is coined by Herbert Blumer [Blu69], and he set out three basic premises of the perspective:

- Human beings act toward things on the basis of the meanings that the things have for them.
- The meaning of such things is derived from, or arises out of, the social interaction that one has with one's fellows.
- These meanings are handled in, and modified through, an interpretive process used by the person in dealing with the things he/she encounters.

For symbolic interactionists, individuals interact with each other by interpreting or defining each other's actions instead of merely reacting to each other's actions. In this aspect, social life is understood with a focus on how reality is constructed by active and creative actors through their interactions with others.

In literary theory, interaction is related to the concept of *identity*. Contemporary literary theory commonly rejects the stable, central, and essential notion of identity to individuals: an individual exists as a nexus of social meanings and practices, with subject-positions within a shifting cultural, ideological, signifying field. Michel Foucault's notion of *discourse* [Fou84] rejects essentialist meaning, and according to him, the subject is a social construction of a discourse. The concept of *performativity* by Judith Butler [dGER01] further echos the idea, where identity is passed or brought to life through performative acts.

Appendix A

Auxiliary Functions and Typing Rules

A.1 Definition of Pedigree Type Variable Occurrence

$$
\begin{array}{rcl}
TV_{\mathtt{t}}(\tau) & \stackrel{\text{def}}{=} & \begin{cases} \emptyset & \tau = \bot \\ TV_{\mathtt{p}}(\rho) \cup TV_{\mathtt{cl}}(\mathcal{K}) & \tau = \mu \mathsf{a}.\mathcal{K}\ @\rho \\ TV_{\mathtt{pset}}(FP_{\mathtt{st}}(\tau)) & \text{otherwise} \end{cases} \\
TV_{\mathtt{p}}((\textbf{parent})^{\nu}(\textbf{child})^{\nu'}) & \stackrel{\text{def}}{=} & TV_{\mathtt{l}}(\nu) \cup TV_{\mathtt{l}}(\nu') \\
TV_{\mathtt{l}}(\nu + \nu') & \stackrel{\text{def}}{=} & TV_{\mathtt{l}}(\nu) \cup TV_{\mathtt{l}}(\nu') \\
TV_{\mathtt{l}}(\alpha) & \stackrel{\text{def}}{=} & \{\alpha\} \\
TV_{\mathtt{l}}(\beta) & \stackrel{\text{def}}{=} & \{\beta\} \\
TV_{\mathtt{l}}(\mathsf{w}) & \stackrel{\text{def}}{=} & \emptyset \\
TV_{\mathtt{l}}(\mathsf{z}) & \stackrel{\text{def}}{=} & \emptyset \\
\\
TV_{\mathtt{cons}}(\Sigma) & \stackrel{\text{def}}{=} & TV_{\mathtt{l}}(\nu_1) \cdots \cup TV_{\mathtt{l}}(\nu_n) \text{ if } \Sigma = \overline{\nu_n =_{\mathtt{s}} 0} \\
TV_{\mathtt{rel}}(R) & \stackrel{\text{def}}{=} & TV_{\mathtt{p}}(\rho_1) \cdots \cup TV_{\mathtt{p}}(\rho_n) \text{ if } R = \overrightarrow{\langle o_n; o'_n \rangle \mapsto \rho_n} \\
TV_{\mathtt{g}}(\mathcal{C}) & \stackrel{\text{def}}{=} & TV_{\mathtt{psig}}(\mathit{psig}_1) \cdots \cup TV_{\mathtt{psig}}(\mathit{psig}_n) \text{ if } \mathcal{C} = \overrightarrow{\mathsf{a}_n \mapsto \mathit{psig}_n} \\
TV_{\mathtt{psig}}(\overline{\forall \langle \alpha_n; \beta_n \rangle}.\langle \mathcal{K}; \mathcal{M}; \mathcal{F} \rangle) & \stackrel{\text{def}}{=} & \{\alpha_1, \beta_1, \ldots, \alpha_n, \beta_n\} \\
TV_{\mathtt{sig}}(\langle \mathcal{K}; \mathcal{M}; \mathcal{F} \rangle) & \stackrel{\text{def}}{=} & TV_{\mathtt{pset}}(FP_{\mathtt{sig}}(\langle \mathcal{K}; \mathcal{M}; \mathcal{F} \rangle)) \\
TV_{\mathtt{cl}}(\overrightarrow{\mathsf{k}_n \mapsto \langle \mathcal{I}_n; \mathcal{E}_n \rangle}) & \stackrel{\text{def}}{=} & TV_{\mathtt{ml}}(\mathcal{I}_1) \cup TV_{\mathtt{ml}}(\mathcal{E}_1) \cup \ldots TV_{\mathtt{ml}}(\mathcal{I}_n) \cup TV_{\mathtt{ml}}(\mathcal{E}_n) \\
TV_{\mathtt{ml}}(\overrightarrow{\mathsf{m}_n \mapsto (\tau_n \to \tau'_n)}) & \stackrel{\text{def}}{=} & TV_{\mathtt{t}}(\tau_1) \cup TV_{\mathtt{t}}(\tau'_1) \cup \ldots TV_{\mathtt{t}}(\tau_n) \cup TV_{\mathtt{t}}(\tau'_n) \\
\\
TV_{\mathtt{pset}}(\overline{\langle \alpha_n; \beta_n \rangle}) & \stackrel{\text{def}}{=} & \{\alpha_1, \beta_1, \ldots, \alpha_n, \beta_n\}
\end{array}
$$

Figure A.1: Pedigree Type Variable Occurrence

Fig. A.1 defines the set of pedigree type variables occurring in various data structures. $TV_{\mathtt{t}}(-)$ defines the set of pedigree type variables occurring in a type, $TV_{\mathtt{p}}(-)$ defines those in a pedigree expression, $TV_{\mathtt{l}}(-)$ defines those in a level expression, $TV_{\mathtt{cons}}(-)$ defines those in a constraint set, $TV_{\mathtt{rel}}(-)$ defines those in a pedigree relation store, $TV_{\mathtt{g}}(-)$ defines those in a global classage signature list, $TV_{\mathtt{psig}}(-)$ defines those in a parameterized classage signature, $TV_{\mathtt{sig}}(-)$ defines those in a classage signature without $\forall$ bindings, $TV_{\mathtt{cl}}(-)$ defines those in a connector signature list, $TV_{\mathtt{ml}}(-)$ defines those in a method signature list (which can be of either an import list, an export list, or a local method list), $TV_{\mathtt{pset}}(-)$ defines those in a set of pedigree type variable pairs.

Abbreviation When no confusion arises, we also use $TV(-)$ to represent the set of pedigree type variables occurring in $-$, where $-$ can be any of the aforementioned data structures.

A.2 Definition of Free Classage Name Occurrence

Fig. A.2 defines the set of classage names occurring free (*i.e.* not bound by μ) in various type-related data structures. Due to the recursive nature of these data structures, we use Z to keep track of those names that are already bound by μ. The rest of the definition is fairly standard.

Abbreviation When no confusion arises, we also use $FCN(-, Z)$ to represent the set of classage names occuring free in $-$ with bound names in set Z, where $-$ can be any of the aforementioned data structures.

A.3 Type Definitions for Dynamic Data Structures

See Fig. A.3. These definitions are standard.

A.4 Typing Environment Extension

We extend the previous definition of Γ to the the one in Fig. A.4 to reflect the case of dynamic typing. Several reserved variables are added: **iconf** represents the configuration at the time of typing, while

$$
\begin{array}{rcl}
FCN(\overrightarrow{\mathsf{a}_n \mapsto \mathit{psig}_n}) & \stackrel{\text{def}}{=} & FCN_{\mathtt{sig}}(\mathit{psig}_1, \emptyset) \cup \ldots FCN_{\mathtt{sig}}(\mathit{psig}_n, \emptyset) \\
FCN_{\mathtt{sig}}(\overline{\forall\langle\alpha;\beta\rangle.\langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle}, Z) & \stackrel{\text{def}}{=} & FCN_{\mathtt{cl}}(\mathcal{K}, Z) \cup FCN_{\mathtt{ml}}(\mathcal{M}, Z) \cup FCN_{\mathtt{fl}}(\mathcal{F}, Z) \\
FCN_{\mathtt{cl}}(\overrightarrow{\mathsf{k}_n \mapsto \langle\mathcal{I}_n;\mathcal{E}_n\rangle}, Z) & \stackrel{\text{def}}{=} & FCN_{\mathtt{iel}}(\mathcal{I}_1, Z) \cup FCN_{\mathtt{iel}}(\mathcal{E}_1, Z) \cup \ldots \\
 & & FCN_{\mathtt{iel}}(\mathcal{I}_n, Z) \cup FCN_{\mathtt{iel}}(\mathcal{E}_n, Z) \\
FCN_{\mathtt{iel}}(\overrightarrow{\mathsf{m}_n \mapsto (c\tau_n \to c\tau'_n)}, Z) & \stackrel{\text{def}}{=} & FCN_{\mathtt{ct}}(c\tau_1, Z) \cup FCN_{\mathtt{ct}}(c\tau'_1, Z) \cup \ldots \\
 & & FCN_{\mathtt{ct}}(c\tau_n, Z) \cup FCN_{\mathtt{ct}}(c\tau'_n, Z) \\
FCN_{\mathtt{ml}}(\overrightarrow{\mathsf{m}_n \mapsto (s\tau_n \to s\tau'_n)}, Z) & \stackrel{\text{def}}{=} & FCN_{\mathtt{st}}(s\tau_1, Z) \cup FCN_{\mathtt{st}}(s\tau'_1, Z) \cup \ldots \\
 & & FCN_{\mathtt{st}}(s\tau_n, Z) \cup FCN_{\mathtt{st}}(s\tau'_n, Z) \\
FCN_{\mathtt{fl}}(\overrightarrow{\mathsf{f}_n \mapsto s\tau_n}, Z) & \stackrel{\text{def}}{=} & FCN_{\mathtt{st}}(s\tau_1, Z) \cup \ldots \\
 & & FCN_{\mathtt{st}}(s\tau_n, Z) \\
FCN_{\mathtt{ct}}(\mathbf{int}, Z) & \stackrel{\text{def}}{=} & \emptyset \\
FCN_{\mathtt{ct}}(\mathbf{unit}, Z) & \stackrel{\text{def}}{=} & \emptyset \\
FCN_{\mathtt{ct}}(\mu\mathsf{a}.\mathcal{K}\ @\rho, Z) & \stackrel{\text{def}}{=} & FCN_{\mathtt{cl}}(\mathcal{K}, Z \cup \{\mathsf{a}\}) \\
FCN_{\mathtt{ct}}(\mathsf{a}@\rho, Z) & \stackrel{\text{def}}{=} & \begin{cases} \emptyset & \mathsf{a} \in Z \\ \{\mathsf{a}\} & \text{otherwise} \end{cases} \\
FCN_{\mathtt{st}}(s\tau, Z) & \stackrel{\text{def}}{=} & \begin{cases} \emptyset & s\tau = \mathsf{k} \\ FCN_{\mathtt{ct}}(s\tau, Z) & \text{otherwise} \end{cases}
\end{array}
$$

Figure A.2: Free Classage Names

$\mathbf{\Delta}$	::=	$\langle\mathcal{H}; W; R\rangle$	*configuration type*
$\mathcal{H}$	::=	$\overrightarrow{o \mapsto \langle\mathsf{a};\sigma;\mathcal{S}\rangle}$	*heap type*
$\mathcal{S}$	::=	$\overrightarrow{\mathsf{f} \mapsto \tau}$	*field store type*

Figure A.3: Runtime Data Structure Typing

currentRt represents the objectage where the typed expression is enclosed in.

Γ	::=	$\overrightarrow{x \mapsto \tau \mid \textbf{me} \mapsto \mathsf{a} \mid \textbf{current} \mapsto \mathsf{k} \mid \textbf{iconf} \mapsto \mathbf{\Delta} \mid \textbf{currentRt} \mapsto o}$	*typing environment*

Figure A.4: The (Re-)Definition of Typing Environment

A.5 Auxiliary Function $global$

Function $global(C, \mathcal{C}) = G$ is defined to compute the classage summaries G for a list of classages C with their corresponding signatures $\mathcal{C}$, defined as in Fig. A.5.

(Def-global)

$$\frac{\mathrm{Dom}(\mathcal{C}) = \{\mathsf{a}_1, \ldots, \mathsf{a}_n\} \qquad \forall i \in \{1, \ldots, n\} \qquad [\textbf{me} : \mathsf{a}_i], \mathcal{C} \vdash_{\mathtt{cls}} C(\mathsf{a}_i) : \mathcal{C}(\mathsf{a}_i) \backslash \Sigma_i, \Pi_i}{global(C, \mathcal{C}) \stackrel{\mathrm{def}}{=} \overrightarrow{\mathsf{a}_n \mapsto \langle \Sigma_n, \Pi_n \rangle}}$$

Figure A.5: Definition of the $global$ Function

A.6 Auxiliary Function $lazyCons$

Function $lazyCons(G, \Pi) = \Sigma$ is defined to compute all the constraints Σ if all constraints for all classages in Π (the instantiation record set) are eagerly merged. It is defined in Fig. A.6.

(Def-lazyCons1)

$$\frac{\Pi = \{\langle \mathsf{a}_1; \rho_1; \sigma_1 \rangle, \ldots \langle \mathsf{a}_n; \rho_n; \sigma_n \rangle\} \qquad \forall i \in \{1, \ldots n\} \qquad G, [\,] \vdash_{\mathtt{cons}} \langle \mathsf{a}_i; \rho_i; \sigma_i \rangle \backslash \Sigma_i}{lazyCons(G, \Pi) \stackrel{\mathrm{def}}{=} \Sigma_1 \cup \cdots \cup \Sigma_n}$$

(Def-lazyCons2)

$$lazyCons(G, \emptyset) \stackrel{\mathrm{def}}{=} \emptyset$$

Figure A.6: Definition of the $lazyCons$ Function

A.7 Well-Formed Configuration Types

See Fig. A.7.

(WFT-InstanceConfig)

$$\frac{\mathcal{C}, R \vdash_{\texttt{wftH}} \mathcal{H} \qquad \mathcal{C}, \mathcal{H}, R \vdash_{\texttt{wftItr}} W \backslash \Sigma \qquad \mathcal{C}, \mathcal{H} \vdash_{\texttt{wftRel}} R}{\mathcal{C} \vdash_{\texttt{wftI}} \langle \mathcal{H}; W; R \rangle \backslash \Sigma}$$

(WFT-Heap)

$$\frac{\begin{array}{c}\forall o \in \mathrm{Dom}(\mathcal{H}) \qquad \mathcal{H}(o) = \langle \mathsf{a}; \sigma; \mathcal{S} \rangle \qquad \mathcal{C}(\mathsf{a}) = \forall \langle \alpha_1; \beta_1 \rangle \ldots \forall \langle \alpha_n; \beta_n \rangle . \langle \mathcal{K}; \mathcal{M}; \mathcal{F} \rangle \\ TV(\mathrm{Ran}(\sigma)) \cap TV(R) = \emptyset \qquad TV(\mathrm{Ran}(\sigma)) \cap TV(\mathcal{C}) = \emptyset \\ \mathrm{Dom}(\sigma) = \{\langle \alpha_1; \beta_1 \rangle \ldots, \langle \alpha_n; \beta_n \rangle\} \qquad \forall \mathsf{f} \in \mathrm{Dom}(\mathcal{S}) \qquad \emptyset \vdash \mathcal{F}(\mathsf{f}) <: \mathcal{S}(\mathsf{f}) \backslash \emptyset \\ \vdash_{\texttt{wftSubst}} \sigma \qquad \forall o' \in \mathrm{Dom}(\mathcal{H}).\ o' \neq o, \mathcal{H}(o') = \langle \mathsf{a}'; \sigma'; \mathcal{S}' \rangle \implies TV(\mathrm{Ran}(\sigma)) \cap TV(\mathrm{Ran}(\sigma')) = \emptyset\end{array}}{\mathcal{C}, R \vdash_{\texttt{wftH}} \mathcal{H}}$$

(WFT-InstanceItr)

$$\frac{\begin{array}{c}\forall \langle o_1; \mathsf{k}_1; o_2; \mathsf{k}_2 \rangle \in \mathrm{Ran}(W) \qquad \mathcal{H}(o_i) = \langle \mathsf{a}_i; \sigma_i; \mathcal{S}_i \rangle, \forall i \in \{1, 2\} \\ \mathcal{C}(\mathsf{a}_i) = \overline{\forall \langle \alpha; \beta \rangle}.\langle \mathcal{K}_i; \mathcal{M}_i; \mathcal{F}_i \rangle, \forall i \in \{1, 2\} \qquad rel(R, o_2, o_1) \vdash (\mathcal{K}_1(\mathsf{k}_1))[\sigma_1] \overset{\Sigma}{\underset{\texttt{dc}}{\Rightarrow}} (\mathcal{K}_2(\mathsf{k}_2))[\sigma_2]\end{array}}{\mathcal{C}, \mathcal{H}, R \vdash_{\texttt{wftItr}} W \backslash \Sigma}$$

(WFT-InstanceRel)

$$\frac{R \text{ is an instantiation tree} \qquad graphRep(R) = \langle \mathrm{Dom}(\mathcal{H}); EDGE; lab \rangle \qquad TV(\mathcal{C}) \cap TV(R) = \emptyset}{\mathcal{C}, \mathcal{H} \vdash_{\texttt{wftRel}} R}$$

(WFT-Subst)

$$\frac{\begin{array}{c}\mathrm{Dom}(\sigma) = \{\langle \alpha_1; \beta_1 \rangle, \ldots, \langle \alpha_m; \beta_m \rangle\} \\ \mathrm{Ran}(\sigma) = \{\langle \alpha'_1; \beta'_1 \rangle, \ldots, \langle \alpha'_n; \beta'_n \rangle\} \qquad \alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_m, \alpha'_1, \ldots, \alpha'_n, \beta'_1, \ldots, \beta'_n \text{ all distinct}\end{array}}{\vdash_{\texttt{wftSubst}} \sigma}$$

Figure A.7: Well-Formedness Definitions

A.8 Auxiliary Definitions for Connector Matching at Dynamic Time

See Fig. A.8.

(Def-ConnectorMatchD)

$$\frac{\mathrm{Dom}(\mathcal{I}_1) = \{\mathsf{m}_1, \ldots, \mathsf{m}_p\} \qquad \forall i \in \{1, \ldots, p\}.(\textbf{parent})^{\nu_2}(\textbf{child})^{\nu_1} \vdash \mathcal{E}_2(\mathsf{m}_i) \overset{\Sigma_i}{\underset{\mathtt{dm}}{\Longrightarrow}} \mathcal{I}_1(\mathsf{m}_i)}{\mathrm{Dom}(\mathcal{I}_2) = \{\mathsf{m}_{p+1}, \ldots, \mathsf{m}_q\} \qquad \forall j \in \{p+1, \ldots, q\}.(\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2} \vdash \mathcal{E}_1(\mathsf{m}_j) \overset{\Sigma_j}{\underset{\mathtt{dm}}{\Longrightarrow}} \mathcal{I}_2(\mathsf{m}_j)}$$

$$(\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2} \vdash \langle \mathcal{I}_1; \mathcal{E}_1 \rangle \overset{\Sigma_1 \cdots \cup \Sigma_q}{\underset{\mathtt{dc}}{\Longrightarrow}} \langle \mathcal{I}_2; \mathcal{E}_2 \rangle$$

(Def-MethodMatchD)

$$\frac{matchd(c\tau'_{\mathtt{im}}, c\tau'_{\mathtt{em}}, (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}) = \Sigma_1 \qquad matchd(c\tau_{\mathtt{em}}, c\tau_{\mathtt{im}}, (\textbf{parent})^{\nu_b}(\textbf{child})^{\nu_a}) = \Sigma_2}{(\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b} \vdash c\tau'_{\mathtt{em}} \to c\tau_{\mathtt{em}} \overset{\Sigma_1 \cup \Sigma_2}{\underset{\mathtt{dm}}{\Longrightarrow}} c\tau'_{\mathtt{im}} \to c\tau_{\mathtt{im}}}$$

$$\begin{array}{rcl}
matchd(c\tau, c\tau', \rho) & \overset{\mathrm{def}}{=} & \Sigma \cup dwfcons(c\tau, c\tau', \rho) \\
 & \text{if} & \emptyset \vdash convert(c\tau, \rho) <: c\tau' \backslash \Sigma \\
dwfcons(\textbf{int}, \textbf{int}, \rho) & \overset{\mathrm{def}}{=} & \emptyset \\
dwfcons(\textbf{unit}, \textbf{unit}, \rho) & \overset{\mathrm{def}}{=} & \emptyset \\
dwfcons(\mu \mathsf{a}_1.\mathcal{K}_1 \ @\rho_1, \mu \mathsf{a}_2.\mathcal{K}_2 \ @\rho_2, \rho) & \overset{\mathrm{def}}{=} & \{\nu_b + \nu_{12} - \nu_{11} \leq_{\mathtt{s}} \nu_{22}\} \\
 & \text{if} & \rho_1 = (\textbf{parent})^{\nu_{11}}(\textbf{child})^{\nu_{12}} \\
 & & \rho_2 = (\textbf{parent})^{\nu_{21}}(\textbf{child})^{\nu_{22}} \\
 & & \rho = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}
\end{array}$$

Figure A.8: Connector Matching at Dynamic Time

A.9 Auxiliary Definitions to Bound Negative Levels

See Fig. A.9.

A.10 Configuration Typing

See Fig. A.10.

$$
\begin{array}{rcl}
wfSig(\mathcal{K}) & \stackrel{\text{def}}{=} & wfM(\mathcal{I}_1) \cup \ldots wfM(\mathcal{I}_n) \\
 & \cup & wfM(\mathcal{E}_1) \cup \ldots wfM(\mathcal{E}_n) \\
 & \text{if} & \mathcal{K} = \overrightarrow{\mathsf{k}_n \mapsto \langle \mathcal{I}_n ; \mathcal{E}_n \rangle} \\
wfM(\iota) & \stackrel{\text{def}}{=} & wfT(c\tau_1) \cup \ldots wfT(c\tau_n) \\
 & \cup & wfT(c\tau'_1) \cup \ldots wfT(c\tau'_n) \\
 & \text{if} & \iota = \overrightarrow{\mathsf{m}_n \mapsto (c\tau_n \rightarrow c\tau'_n \rangle)}, \iota \text{ is } \mathcal{I} \text{ or } \mathcal{E} \\
wfT(\textbf{int}) & \stackrel{\text{def}}{=} & \emptyset \\
wfT(\textbf{unit}) & \stackrel{\text{def}}{=} & \emptyset \\
wfT(\bot) & \stackrel{\text{def}}{=} & \emptyset \\
wfT(\mathsf{k}) & \stackrel{\text{def}}{=} & \emptyset \\
wfT(\mu \mathsf{a}.\mathcal{K} \mathbin{@} \rho) & \stackrel{\text{def}}{=} & wfP(\rho) \cup wfSig(\mathcal{K}) \\
wfT(\mathsf{a} @ \rho) & \stackrel{\text{def}}{=} & wfP(\rho) \\
wfP((\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}) & \stackrel{\text{def}}{=} & \{\nu_a \geq_{\mathsf{s}} 0\} \cup twoVals(\nu_b)
\end{array}
$$

Figure A.9: Negative Level Constraints

A.11 Auxilary Expression Typing

See Fig. A.11.

(T-Config)

$$\frac{\vdash_{\mathtt{G}} C : Ct \qquad \Psi(Ct) = \mathcal{C} \qquad G = global(C, \mathcal{C}) \qquad G, \mathcal{C} \vdash_{\mathtt{iconf}} \Delta : \boldsymbol{\Delta} \backslash \Sigma_{\mathtt{iconf}} \qquad G, \mathcal{C}, \boldsymbol{\Delta}, o \vdash_{\mathtt{exp}} exd : \tau \backslash \Sigma_{\mathtt{exp}}}{\vdash \langle C; \Delta; o; exd \rangle : \tau \backslash (\Sigma_{\mathtt{iconf}} \cup \Sigma_{\mathtt{exp}})}$$

(T-InstanceConfig)

$$\frac{\mathcal{C} \vdash_{\mathtt{wftI}} \langle \mathcal{H}; W; R \rangle \backslash \Sigma_{\mathtt{wfti}} \qquad G, \mathcal{C}, \langle \mathcal{H}; W; R \rangle \vdash_{\mathtt{h}} H : \mathcal{H} \backslash \Sigma_{\mathtt{h}}}{G, \mathcal{C} \vdash_{\mathtt{iconf}} \langle H; W; R \rangle : \langle \mathcal{H}; W; R \rangle \backslash \Sigma_{\mathtt{wfti}} \cup \Sigma_{\mathtt{h}}}$$

(T-Heap)

$$\frac{\mathrm{Dom}(H) = \mathrm{Dom}(\mathcal{H}) = \{o_1, \ldots, o_n\} \qquad \forall i \in \{1, \ldots n\} \qquad G, \mathcal{C}, \boldsymbol{\Delta}, o_i \vdash_{\mathtt{hc}} H(o_i) : \mathcal{H}(o_i) \backslash \Sigma_i}{G, \mathcal{C}, \boldsymbol{\Delta} \vdash_{\mathtt{h}} H : \mathcal{H} \backslash (\Sigma_1 \cup \ldots, \Sigma_n)}$$

(T-HeapCell)

$$\frac{G(\mathsf{a}) = \langle \Sigma; \Pi \rangle \qquad \mathcal{C}, \boldsymbol{\Delta}, o \vdash_{\mathtt{f}} S : \mathcal{S} \backslash \Sigma'}{G, \mathcal{C}, \boldsymbol{\Delta}, o \vdash_{\mathtt{hc}} \langle \mathsf{a}; \sigma; S \rangle : \langle \mathsf{a}; \sigma; \mathcal{S} \rangle \backslash (\Sigma[\sigma] \cup lazyCons(G, \Pi) \cup \Sigma'[\sigma])}$$

(T-FieldStore)

$$\frac{\mathrm{Dom}(S) = \mathrm{Dom}(\mathcal{S}) = \{\mathsf{f}_1, \ldots, \mathsf{f}_n\} \qquad \forall i \in \{1, \ldots n\} \qquad S(\mathsf{f}_i) \in \mathbb{V} \qquad [\mathbf{iconf} : \boldsymbol{\Delta}, \mathbf{currentRt} : o], \mathcal{C} \vdash S(\mathsf{f}_i) : \mathcal{S}(\mathsf{f}_i) \backslash \Sigma_i, \emptyset}{\mathcal{C}, \boldsymbol{\Delta}, o \vdash_{\mathtt{f}} S : \mathcal{S} \backslash (\Sigma_1 \cup \ldots \Sigma_n)}$$

(T-Exp)

$$\frac{\boldsymbol{\Delta} = \langle \mathcal{H}; W; R \rangle \qquad \mathcal{H}(o) = \langle \mathsf{a}; \sigma; \mathcal{S} \rangle \qquad [\mathbf{iconf} : \boldsymbol{\Delta}, \mathbf{me} : \mathsf{a}, \mathbf{currentRt} : o], \mathcal{C} \vdash exd : \tau \backslash \Sigma, \Pi}{G, \mathcal{C}, \boldsymbol{\Delta}, o \vdash_{\mathtt{exp}} exd : \tau[\sigma] \backslash (\Sigma[\sigma] \cup lazyCons(G, \Pi))}$$

Figure A.10: Configuration Typing

(T-Null)

$$\Gamma, \mathcal{C} \vdash \textbf{null} : \bot \backslash \emptyset, \emptyset$$

(T-RID)

$$\frac{\Gamma(\textbf{iconf}) = \langle \mathcal{H};\, W;\, R\rangle \qquad \mathcal{H}(o) = \langle \mathsf{a}; \sigma; \mathcal{S}\rangle \qquad \mathcal{C}(\mathsf{a}) = \overline{\forall\langle\alpha;\beta\rangle}.\langle \mathcal{K}; \mathcal{M}; \mathcal{F}\rangle \qquad rel(R, o, \Gamma(\textbf{currentRt})) = \rho \qquad \rho' = (\textbf{parent})^{\nu_a}(\textbf{parent})^{\nu_b}}{\Gamma, \mathcal{C} \vdash o : (\mu \mathsf{a}.\mathcal{K}[\sigma]\ @\rho') \backslash subPed(\rho, \rho') \cup \{\nu_a \geq_{\mathtt{s}} 0\}, \emptyset}$$

(T-CID)

$$\frac{\Gamma(\textbf{iconf}) = \langle \mathcal{H};\, W;\, R\rangle \qquad W(c) = \begin{cases} \langle o_1; \mathsf{k}_1; o_2; \mathsf{k}_2\rangle & \text{if } \mathit{side} = \mathtt{A} \\ \langle o_2; \mathsf{k}_2; o_1; \mathsf{k}_1\rangle & \text{if } \mathit{side} = \mathtt{P} \end{cases} \qquad \Gamma(\textbf{currentRt}) = o_1}{\Gamma, \mathcal{C} \vdash c^{side} : \mathsf{k}_1 \backslash \emptyset, \emptyset}$$

(T-In)

$$\frac{\begin{array}{c}\Gamma(\textbf{iconf}) = \boldsymbol{\Delta} = \langle \mathcal{H};\, W;\, R\rangle \qquad \Gamma(\textbf{currentRt}) = o \\ \mathcal{H}(o_1) = \langle \mathsf{a}_1; \sigma_1; \mathcal{S}_1\rangle \qquad \mathcal{H}(o) = \langle \mathsf{a}; \sigma; \mathcal{S}\rangle \qquad [\textbf{iconf} : \boldsymbol{\Delta}, \textbf{me} : \mathsf{a}_1, \textbf{currentRt} : o_1], \mathcal{C} \vdash \mathit{exd} : \tau_1 \backslash \Sigma_1, \Pi_1 \\ \rho = \begin{cases} (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b} & \text{if } \mathit{side} = \mathtt{P},\ rel(R_1, o_1, o) = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b} \\ (\textbf{parent})^{\nu_b}(\textbf{child})^{\nu_a} & \text{if } \mathit{side} = \mathtt{A},\ rel(R_1, o, o_1) = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b} \end{cases} \\ TV(\tau) \subseteq TV(\mathcal{C}(\mathsf{a})) \qquad TV(\tau_1) \subseteq TV(\mathcal{C}(\mathsf{a}_1)) \end{array}}{\Gamma, \mathcal{C} \vdash \textbf{in}_{\mathit{side}}(o_1, \mathit{exd}) : \tau \backslash wfT(\tau)[\sigma] \cup \Sigma_1[\sigma_1] \cup matchd(\tau_1[\sigma_1], \tau[\sigma], \rho) \cup lazyCons(G, \Pi), \emptyset}$$

Figure A.11: Auxiliary Typing Rules for Run-Time Expressions

Appendix B

Auxiliary Definitions and Properties of Constraints

Constraints in our calculus are linear ones. This chapter gives a rigorous specification on its syntactical forms and solutions.

B.1 Level Expressions

B.1.1 Concrete Syntax

In Chapter 6, we have defined level expressions as follows:

ν	::=	$\nu + \nu' \mid \nu - \nu' \mid lep$	*level expression (in abstract syntax)*
lep	::=	$\alpha \mid \mathsf{w}$	*level expression primitives*

Note that the syntax here is abstract, *i.e.*, the grammar itself is ambiguous. What is behind this syntax is a abstract syntax tree representation of level expressions. The concrete syntax adds parentheses to disambiguate the syntax:

ν	::=	$(\nu + \nu') \mid (\nu - \nu') \mid (lep)$	*level expression (in concrete syntax)*

One commonly used convention of linear expressions is that parentheses can be omitted if removing them still preserves left-associativity. Our calculus also uses this convention, the de-parenthesized syntax is defined as a syntactic sugar:

$$\nu_1\ op_1\ \nu_2\ op_2\ \nu_3 \ldots \nu_n \quad \stackrel{\text{def}}{=} \quad (((\nu_1\ op_1\ \nu_2)\ op_2\ \nu_3) \ldots \nu_n)$$
$$\text{where} \quad n \geq 1, \forall i \in \{1, \ldots, n\}, op \in \{\ +\ ,\ -\ \}$$

B.1.2 Semantic Definition of Level Expressions

Partial function $eval(SOL, \nu)$ is defined as follows, and it is undefined if $TV(\nu) \nsubseteq \mathrm{Dom}(SOL)$. SOL is the mapping from a type variable to non-negative integer constants. The definition is the standard linear expression evaluation in line with how linear expressions are evaluated. Note that $+$, $-$ are merely syntactical symbols. They themselves are not functions.

$$\frac{\begin{array}{c} eval(SOL, \nu_1) = const_1 \\ eval(SOL, \nu_2) = const_2 \\ const \text{ is the integer representing the algebraic addition of } const_1 \text{ and } const_2 \end{array}}{eval(SOL, (\nu_1 + \nu_2)) = const}$$

$$\frac{\begin{array}{c} eval(SOL, \nu_1) = const_1 \\ eval(SOL, \nu_2) = const_2 \\ const \text{ is the integer representing the algebraic subtraction of } const_1 \text{ and } const_2 \end{array}}{eval(SOL, (\nu_1 - \nu_2)) = const}$$

$$\frac{const \text{ is the integer representing } \mathsf{w}}{eval(SOL, (\mathsf{w})) = const} \qquad \frac{const \text{ is the integer representing } SOL(\alpha)}{eval(SOL, (\alpha)) = const}$$

SOL	::=	$\overrightarrow{\alpha \mapsto \mathsf{w}}$	*solution mapping*

B.1.3 Level Expressions Equivalence

Definition 6 (Syntactical Equivalence of Level Expressions)**.** *$\nu_1 = \nu_2$ denotes syntactical equivalence of level expressions. It is defined as syntactical isomorphism between ν_1 and ν_2.*

Definition 7 (Semantic Equivalence of Level Expressions)**.** $\nu_1 \equiv \nu_2$ *denotes semantic equivalence of level expressions. It is defined as* $eval(SOL, \nu_1) = const$ *iff* $eval(SOL, \nu_2) = const$.

Lemma 1 (Basic Properties of $\equiv$ for Level Expressions)**.** *The following properties always hold*

- (EL-reflexivity) $\nu_1 \equiv \nu_2$ *if* $\nu_1 = \nu_2$.
- (EL-symmetry) $\nu_1 \equiv \nu_2$ *if* $\nu_2 \equiv \nu_1$.
- (EL-transitivity) $\nu_1 \equiv \nu_3$ *if* $\nu_1 \equiv \nu_2$ *and* $\nu_2 \equiv \nu_3$.
- (EL-commutativity) $\nu_1 + \nu_2 \equiv \nu_2 + \nu_1$.
- (EL-associativity) $(\nu_1 + \nu_2) + \nu_3 \equiv \nu_1 + (\nu_2 + \nu_3)$.
- (EL-identity) $\nu + 0 \equiv \nu$ *and* $\nu - 0 \equiv \nu$.
- (EL-cancellation) $\nu - \nu \equiv 0$.
- (EL-addition) $\nu_1 + \nu_2 \equiv \nu_1' + \nu_2'$ *if* $\nu_1 \equiv \nu_1'$ *and* $\nu_2 \equiv \nu_2'$.
- (EL-subtraction) $\nu_1 - \nu_2 \equiv \nu_1' - \nu_2'$ *if* $\nu_1 \equiv \nu_1'$ *and* $\nu_2 \equiv \nu_2'$.
- (EL-minus) $\nu_1 - (\nu_2 + \nu_3) \equiv \nu_1 - \nu_2 - \nu_3$ *and* $\nu_1 - (\nu_2 - \nu_3) \equiv \nu_1 - \nu_2 + \nu_3$.

Proof. These are all basic properties of arithmetic expressions. The proofs can be found in numerous texts on elementary algebra.

□

With these basic properties, many other properties on arithmetic expressions can be derived. For instance, if $\nu_1 + \nu \equiv \nu_2 + \nu$, then $\nu_1 \equiv \nu_2$. This can be easily proved by applying (EL-subtraction), (EL-cancellation), and (EL-identity). In this thesis, we do not prove these derived properties one by one.

Corollary 1. *Both* $=$ *and* $\equiv$ *are equivalent relations over level expressions.*

Corollary 2 (Syntactical Equivalence Implies Semantic Equivalence)**.** *If* $\nu_1 = \nu_2$*, then* $\nu_1 \equiv \nu_2$.

Proof. See (EL-reflexivity). □

B.2 Constraints

B.2.1 Syntax

In Chapter 6, we have defined constraints as $\nu =_{\mathtt{s}} 0$. For convenience, the type system has used constraints with other syntactical forms which are merely mnemonic sugars defined as follows:

$$\begin{array}{rcl} \nu_1 =_{\mathtt{s}} \nu_2 & \stackrel{\text{def}}{=} & \nu_1 - \nu_2 =_{\mathtt{s}} 0 \\ \nu_1 \geq_{\mathtt{s}} \nu_2 & \stackrel{\text{def}}{=} & \nu_1 - \nu_2 - \alpha =_{\mathtt{s}} 0 \text{ if } \alpha \text{ fresh} \\ \nu_1 \leq_{\mathtt{s}} \nu_2 & \stackrel{\text{def}}{=} & \nu_2 - \nu_1 - \alpha =_{\mathtt{s}} 0 \text{ if } \alpha \text{ fresh} \end{array}$$

B.2.2 Semantic Definition of Constraints

We now give a semantic definition of constraints: their solutions.

Definition 8 (Constraint Solution)**.** *SOL is a solution for a constraint con, denoted as $SOL \models$ con, according to the definition below.*

$$\frac{eval(SOL, \nu) = const \qquad const \text{ is integer } 0}{SOL \models \nu =_{\mathtt{s}} 0}$$

B.3 Constraint Sets

B.3.1 Semantic Definition of Constraint Sets

The syntax of constraint sets are defined in Chapter 6. We now give a semantic definition of constraint sets: their solutions.

Definition 9 (Constraint Set Solution)**.** *SOL is a solution for a constraint set Σ, denoted as $SOL \models \Sigma$, iff $SOL \models con$ for all $con \in \Sigma$.*

Note that by definition any element in $\mathrm{Ran}(SOL)$ is a non-negative integer.

B.3.2 Constraint Set Equivalence and Implication

Definition 10 (Syntactical Equivalence of Constraint Sets)**.** $\Sigma_1 = \Sigma_2$ *denotes syntactical equivalence of constraint sets. It is defined as the standard set equality: all elements of* Σ_1 *and* Σ_2 *must be pair-wise syntactically isomorphic, but the order of the elements do not matter.*

Definition 11 (Semantic Equivalence of Constraint Sets)**.** $\Sigma_1 \equiv \Sigma_2$ *denotes semantic equivalence of constraint sets. It is defined iff* $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_2$ *and* $\Sigma_2 \xrightarrow{\texttt{algebra}} \Sigma_1$.

Definition 12 (Semantic Implication of Constraint Sets)**.** $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_2$ *is defined as follows: if* $SOL \models \Sigma_1$ *and* $TV(\Sigma_1) \cup TV(\Sigma_2) \in \mathrm{Dom}(SOL)$*, then* $SOL \models \Sigma_2$.

Lemma 2 (Basic Properties of $\xrightarrow{\texttt{algebra}}$ for Constraint Sets)**.** *The following properties always hold:*

- (ICS-reflexivity) $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_2$ *if* $\Sigma_1 = \Sigma_2$.
- (ICS-transitivity) $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_3$ *if* $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_2$ *and* $\Sigma_2 \xrightarrow{\texttt{algebra}} \Sigma_3$.
- (ICS-tautology)
  - $\emptyset \xrightarrow{\texttt{algebra}} \{\alpha \geq_{\mathsf{s}} 0\}$
  - $\emptyset \xrightarrow{\texttt{algebra}} \{\mathsf{w} \geq_{\mathsf{s}} 0\}$
  - $\emptyset \xrightarrow{\texttt{algebra}} \{0 \leq_{\mathsf{s}} 0 \leq_{\mathsf{s}} 1\}$
  - $\emptyset \xrightarrow{\texttt{algebra}} \{0 \leq_{\mathsf{s}} 1 \leq_{\mathsf{s}} 1\}$
  - $\emptyset \xrightarrow{\texttt{algebra}} \{\nu =_{\mathsf{s}} \nu'\}$ *if* $\nu \equiv \nu'$
- (ICS-relaxation) $\{\nu =_{\mathsf{s}} 0\} \xrightarrow{\texttt{algebra}} \emptyset$.
- (ICS-equal)
  - $\{\nu_1 =_{\mathsf{s}} \nu_2\} \xrightarrow{\texttt{algebra}} \{\nu_2 =_{\mathsf{s}} \nu_1\}$
  - $\{\nu_1 =_{\mathsf{s}} \nu_2, \nu_2 =_{\mathsf{s}} \nu_3\} \xrightarrow{\texttt{algebra}} \{\nu_1 =_{\mathsf{s}} \nu_3\}$
  - $\{\nu_1 =_{\mathsf{s}} \nu_2, \nu_3 =_{\mathsf{s}} \nu_4\} \xrightarrow{\texttt{algebra}} \{\nu_1 + \nu_3 =_{\mathsf{s}} \nu_2 + \nu_4\}$
  - $\{\nu_1 =_{\mathsf{s}} \nu_2, \nu_3 =_{\mathsf{s}} \nu_4\} \xrightarrow{\texttt{algebra}} \{\nu_1 - \nu_3 =_{\mathsf{s}} \nu_2 - \nu_4\}$

- (ICS-geq)

$$\{\nu_1 \geq_{\mathtt{s}} \nu_2, \nu_2 \geq_{\mathtt{s}} \nu_3\} \xrightarrow{\texttt{algebra}} \{\nu_1 \geq_{\mathtt{s}} \nu_3\}$$

$$\{\nu_1 \geq_{\mathtt{s}} \nu_2, \nu_3 \geq_{\mathtt{s}} \nu_4\} \xrightarrow{\texttt{algebra}} \{\nu_1 + \nu_3 \geq_{\mathtt{s}} \nu_2 + \nu_4\}$$

$$\{\nu_1 \geq_{\mathtt{s}} \nu_2, \nu_3 \leq_{\mathtt{s}} \nu_4\} \xrightarrow{\texttt{algebra}} \{\nu_1 - \nu_3 \geq_{\mathtt{s}} \nu_2 - \nu_4\}$$

- (ICS-leq)

$$\{\nu_1 \leq_{\mathtt{s}} \nu_2, \nu_2 \leq_{\mathtt{s}} \nu_3\} \xrightarrow{\texttt{algebra}} \{\nu_1 \leq_{\mathtt{s}} \nu_3\}$$

$$\{\nu_1 \leq_{\mathtt{s}} \nu_2, \nu_3 \leq_{\mathtt{s}} \nu_4\} \xrightarrow{\texttt{algebra}} \{\nu_1 + \nu_3 \leq_{\mathtt{s}} \nu_2 + \nu_4\}$$

$$\{\nu_1 \leq_{\mathtt{s}} \nu_2, \nu_3 \geq_{\mathtt{s}} \nu_4\} \xrightarrow{\texttt{algebra}} \{\nu_1 - \nu_3 \leq_{\mathtt{s}} \nu_2 - \nu_4\}$$

- (ICS-inter)

$$\{\nu_1 =_{\mathtt{s}} \nu_2\} \xrightarrow{\texttt{algebra}} \{\nu_1 \geq_{\mathtt{s}} \nu_2\}$$

$$\{\nu_1 =_{\mathtt{s}} \nu_2\} \xrightarrow{\texttt{algebra}} \{\nu_1 \leq_{\mathtt{s}} \nu_2\}$$

$$\{\nu_1 \geq_{\mathtt{s}} \nu_2\} \xrightarrow{\texttt{algebra}} \{\nu_2 \leq_{\mathtt{s}} \nu_1\}$$

$$\{\nu =_{\mathtt{s}} 0\} \xrightarrow{\texttt{algebra}} \{0 \leq_{\mathtt{s}} \nu \leq_{\mathtt{s}} 1\}$$

$$\{\nu =_{\mathtt{s}} 1\} \xrightarrow{\texttt{algebra}} \{0 \leq_{\mathtt{s}} \nu \leq_{\mathtt{s}} 1\}$$

- (ICS-substitution) $\{\nu =_{\mathtt{s}} 0, \nu_1 =_{\mathtt{s}} \nu_2\} \xrightarrow{\texttt{algebra}} \{\nu' =_{\mathtt{s}} 0\}$ *if* ν_1 *is a subtree on the syntax tree of* ν *and* ν' *is identical to* ν *except the subtree of* ν_1 *is replaced with* ν_2.

- (ICS-union) $\Sigma_1 \cup \Sigma_2 \xrightarrow{\texttt{algebra}} \Sigma_1' \cup \Sigma_2'$ *if* $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_1'$ *and* $\Sigma_2 \xrightarrow{\texttt{algebra}} \Sigma_2'$.

Proof. These are basic properties for systems of linear equations. Rigorous proof can be obtained via Def. 12.

□

Lemma 3 (Basic Properties of $\equiv$ for Constraint Sets)**.** *The following properties always hold:*

- (ECS-reflexivity) $\Sigma_1 \equiv \Sigma_2$ *if* $\Sigma_1 = \Sigma_2$.

- (ECS-symmetry) $\Sigma_1 \equiv \Sigma_2$ *if* $\Sigma_2 \equiv \Sigma_1$.

- (ECS-transitivity) $\Sigma_1 \equiv \Sigma_3$ *if* $\Sigma_1 \equiv \Sigma_2$ *and* $\Sigma_2 \equiv \Sigma_3$.

- (ECS-tautology)

$\emptyset \equiv \{\alpha \geq_{\mathtt{s}} 0\}$

$\emptyset \equiv \{\mathsf{w} \geq_{\mathtt{s}} 0\}$

$\emptyset \equiv \{0 \leq_{\mathtt{s}} 0 \leq_{\mathtt{s}} 1\}$

$\emptyset \equiv \{0 \leq_{\mathtt{s}} 1 \leq_{\mathtt{s}} 1\}$

$\emptyset \equiv \{\nu =_{\mathtt{s}} \nu'\}$ *if* $\nu \equiv \nu'$

- (ECS-equal) $\{\nu_1 =_{\mathtt{s}} \nu_2\} \equiv \{\nu_2 =_{\mathtt{s}} \nu_1\}$.
- (ECS-substitution) $\{\nu =_{\mathtt{s}} 0, \nu_1 =_{\mathtt{s}} \nu_2\} \xrightarrow{\texttt{algebra}} \{\nu' =_{\mathtt{s}} 0, \nu_1 =_{\mathtt{s}} \nu_2\}$ *if* ν_1 *is a subtree on the syntax tree of* ν *and* ν' *is identical to* ν *except the subtree of* ν_1 *is replaced with* ν_2.
- (ECS-union) $\Sigma_1 \cup \Sigma_2 \equiv \Sigma_1' \cup \Sigma_2'$ *if* $\Sigma_1 \equiv \Sigma_1'$ *and* $\Sigma_2 \equiv \Sigma_2'$.

Proof. These are basic properties for systems of linear equations. Rigorous proof can be obtained via Def. 12 and Def. 11. □

Corollary 3. $\xrightarrow{\texttt{algebra}}$ *is a pre-order.*

Proof. See (ICS-reflexivity) and (ICS-transitivity). □

Corollary 4. *Both* $=$ *and* $\equiv$ *are equivalent relations over constraint sets.*

Corollary 5 (Syntactical Equivalence Implies Semantic Equivalence)**.** *If* $\Sigma_1 = \Sigma_2$*, then* $\Sigma_1 \equiv \Sigma_2$.

Proof. See (ECS-reflexivity) and the definition of $\equiv$. □

Many secondary properties can be further proved from above. We do not list all of them. They are usually very intuitive and can be easily derived from the basic properties. Examples are as follows:

$$\{\nu \geq_{\mathtt{s}} 0\} \xrightarrow{\texttt{algebra}} \{\nu + \alpha \geq_{\mathtt{s}} 0\}$$
$$\{\nu \geq_{\mathtt{s}} 0\} \xrightarrow{\texttt{algebra}} \{\nu + \mathsf{w} \geq_{\mathtt{s}} 0\}$$
$$\{\nu_1 + \nu =_{\mathtt{s}} \nu_2 + \nu\} \equiv \{\nu_1 =_{\mathtt{s}} \nu_2\}$$
$$\{\nu_1 - \nu =_{\mathtt{s}} \nu_2 - \nu\} \equiv \{\nu_1 =_{\mathtt{s}} \nu_2\}$$
$$\{\nu_1 + \nu \geq_{\mathtt{s}} \nu_2 + \nu\} \equiv \{\nu_1 \geq_{\mathtt{s}} \nu_2\}$$
$$\{\nu_1 - \nu \geq_{\mathtt{s}} \nu_2 - \nu\} \equiv \{\nu_1 \geq_{\mathtt{s}} \nu_2\}$$
$$\{\nu_1 =_{\mathtt{s}} \nu_2\} \equiv \{\nu_1 - \nu_2 =_{\mathtt{s}} 0\}$$
$$\{\nu_1 \geq_{\mathtt{s}} \nu_2\} \equiv \{\nu_1 - \nu_2 \geq_{\mathtt{s}} 0\}$$

$$\{\nu_1 \leq_{\mathtt{s}} \nu_2\} \equiv \{\nu_1 - \nu_2 \leq_{\mathtt{s}} 0\}$$
$$\Sigma_1 \cup \Sigma_3 \equiv \Sigma_2 \cup \Sigma_4 \text{ if } \Sigma_1 \equiv \Sigma_2 \text{ and } \Sigma_3 \equiv \Sigma_4$$

Lemma 4 (Type Variable Substitution)**.** *If* $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_2$*, then* $\Sigma_1[\sigma] \xrightarrow{\texttt{algebra}} \Sigma_2[\sigma]$ *for any* σ*. If* $\Sigma_1 \equiv \Sigma_2$*, then* $\Sigma_1[\sigma] \equiv \Sigma_2[\sigma]$ *for any* σ*.*

Proof. This is merely the renaming of variables. Rigorously, case analysis on the $\xrightarrow{\texttt{algebra}}$ relation or the $\equiv$ relation. □

In the following proof, we only explicitly refer to the lemmas and corollaries in this chapter when the reasoning is not obvious. Otherwise they are implicitly used.

Appendix C

Proof

C.1 Properties of the Ψ Function

Lemma 5. *If $\Psi(Ct) = \mathcal{C}$, then $\mathcal{C}$ always conforms to the syntax defined in Fig. 6.3.*

Proof. Straightforward according to the definition of Ψ. □

The proof of the above lemma is simple; it however shows some fact about $\mathcal{C}$ that is useful for the rest of the proof. Specifically, if we know $\mathcal{C}$ is the result of $\Psi(Ct)$, then for any $\mathcal{C}(\mathsf{a}) = \overline{\forall\langle\alpha;\beta\rangle}\langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle$ and any $\mathcal{K}(\mathsf{k}) = \langle\mathcal{I};\mathcal{E}\rangle$ and any $\mathcal{I}(\mathsf{m}) = c\tau \to c\tau'$, then both $c\tau$ and $c\tau'$ can only be one of the four forms: **unit**, **int**, $\mu\mathsf{a}.\mathcal{K}'\ @\rho$ and $\mathsf{a}@\rho$. The similar result holds for any $\mathcal{E}(\mathsf{m}) = c\tau \to c\tau'$.

The lemma also says the result holds recursively. If $c\tau$ or $c\tau'$ above takes the form of $\mu\mathsf{a}.\mathcal{K}'\ @\rho$, then for any $\mathcal{K}'(\mathsf{k}') = \langle\mathcal{I}';\mathcal{E}'\rangle$ and for any $\mathcal{I}'(\mathsf{m}') = c\tau'' \to c\tau'''$, then both $c\tau''$ and $c\tau'''$ can only be one of the four forms. For any $\mathcal{E}'(\mathsf{m}') = c\tau'' \to c\tau'''$, then both $c\tau''$ and $c\tau'''$ can only be one of the four forms.

Lemma 6 (Top-Level Types in Signature $\Psi(Ct)$ Are Not Recursive Instance Types)**.** *If $\Psi(Ct) = \mathcal{C}$, then for any $\mathsf{a} \in \mathrm{Dom}(\mathcal{C})$, $FCN_{\mathtt{sig}}(\mathcal{C}(\mathsf{a}), \emptyset) = \emptyset$.*

Proof. Straightforward according to the definition of Ψ. Note that on the top level, set Z is set to be $\emptyset$, and hence no recursive instance types can appear on the top level of the parameterized signatures produced. □

Lemma 7. *If $\Psi(Ct) = \mathcal{C}$, then for any $\mathcal{C}(\mathsf{a}) = \overline{\forall\langle\alpha;\beta\rangle}.\langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle$, it holds that $\mathcal{K}\{\mathsf{a} \circlearrowleft \mathcal{K}\} = \mathcal{K}$.*

Proof. Strictly follows Lem. 6 and the definition of Ψ, and the definition for recursive type unfolding. □

Lemma 8 (Closed Quantification)**.** *If $\Psi(Ct) = \mathcal{C}$, then $\forall\ \mathcal{C}(\mathsf{a}) = \forall\langle\alpha_1;\beta_1\rangle \ldots \forall\langle\alpha_n;\beta_n\rangle.\ \langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle$, we have $FP_{\mathtt{sig}}(\langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle) = \{\langle\alpha_1;\beta_1\rangle \ldots \langle\alpha_n;\beta_n\rangle\}$ and $TV(\mathcal{C}(\mathsf{a})) = TV(\langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle)$.*

Proof. Trivial. □

Lemma 9 ($\Psi()$ Computes Well-Formed Signatures)**.** *If $\vdash_{\mathtt{G}} C : Ct$, $\Psi(Ct) = \mathcal{C}$, $G = global(C, \mathcal{C})$, then for any $G(\mathsf{a}) = \langle\Sigma;\Pi\rangle$, we know $\Sigma \xrightarrow{\texttt{algebra}} wfSig(\mathcal{K})$ for any $\mathcal{C}(\mathsf{a}) = \overline{\forall\langle\alpha_n;\beta_n\rangle}.\langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle$.*

Proof. According to the definition of $wfSig$, we know if we can prove $\Sigma \xrightarrow{\texttt{algebra}} wfT(c\tau) \cup wfT(c\tau')$ for any $\mathcal{K}(\mathsf{k}) = \langle\mathcal{I};\mathcal{E}\rangle$ and any $\mathcal{I}(\mathsf{m}) = c\tau' \to c\tau$ or $\mathcal{E}(\mathsf{m}) = c\tau' \to c\tau$, the conclusion holds by (ICS-union). We now only consider $c\tau$; the case for $c\tau'$ is identical. By the definition of Ψ function, we know $c\tau$ can only be 1) **int**. In this case $wfT(c\tau) = \emptyset$ by the definition of wfT, and the conclusion holds trivially. 2) **unit**. Identical to the previous case. 3) $\mu\mathsf{a}'.\mathcal{K}'\ @(\textbf{parent})^{\mathsf{w}}(\textbf{child})^{\mathsf{z}}$ for some a' and $\mathcal{K}'$. First of all, by (T-Global), (T-Classage), (T-Connectors), (T-Methods), we know $FP_{\mathtt{cl}}(\mathcal{K}') \subseteq FP_{\mathtt{sig}}(\langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle)$. By Lem. 8, we know $FP_{\mathtt{sig}}(\langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle) = \{\langle\alpha_1;\beta_1\rangle \ldots \langle\alpha_n;\beta_n\rangle\}$. Thus $FP_{\mathtt{cl}}(\mathcal{K}') \subseteq \{\langle\alpha_1;\beta_1\rangle \ldots \langle\alpha_n;\beta_n\rangle\}$. By (T-Classage), we know $twoVals(\beta_1) \cup \cdots \cup twoVals(\beta_n) \subseteq \Sigma$. Thus by the definition of $wfSig$, (ICS-relaxation), (ICS-tautology), we know $\Sigma \xrightarrow{\texttt{algebra}} twoVals(\beta_1) \cup \cdots \cup twoVals(\beta_n) \cup \{\alpha_1 \geq_{\mathtt{s}} 0\} \cup \cdots \cup \{\alpha_n \geq_{\mathtt{s}} 0\} = wfSig(\mathcal{K}')$. Again by the definition we know $wfT(c\tau) = \{\mathsf{w} \geq_{\mathtt{s}} 0\} \cup \{0 \leq_{\mathtt{s}} \mathsf{z} \leq_{\mathtt{s}} 1\} \cup wfSig(\mathcal{K}')$. Thus $\Sigma \xrightarrow{\texttt{algebra}} wfT(c\tau)$ by (ICS-tautology), (ICS-union). 4) $\mu\mathsf{a}'.\mathcal{K}'\ @(\textbf{parent})^{\alpha}(\textbf{child})^{\beta}$ for some a' and $\mathcal{K}'$. $wfT(c\tau) = \{\alpha \geq_{\mathtt{s}} 0\} \cup \{0 \leq_{\mathtt{s}} \beta \leq_{\mathtt{s}} 1\} \cup wfSig(\mathcal{K}')$. Following the same strategy as the previous case we can prove $\Sigma \xrightarrow{\texttt{algebra}} wfSig(\mathcal{K}')$. Also note that $\langle\alpha;\beta\rangle \in FP_{\mathtt{sig}}(\langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle)$. In the similar vein as the previous case, we know $\langle\alpha;\beta\rangle \in \{\langle\alpha_1;\beta_1\rangle \ldots \langle\alpha_n;\beta_n\rangle\}$. By (T-Classage), we know $twoVals(\beta) \subseteq \Sigma$. Thus by the definition of wfT and (ICS-tautology), we know $\Sigma \xrightarrow{\texttt{algebra}} \{\alpha \geq 0\} \cup twoVals(\beta)$. Thus $\Sigma \xrightarrow{\texttt{algebra}} wfT(c\tau)$ by (ICS-union).

□

C.2 Properties of Pedigree Type Variable Substitution

Lemma 10 (The Domain of σ Corresponds to Pedigree Type Variables of the Signature)**.** *If* $G, \mathcal{C} \vdash_{\texttt{wftI}} \langle \mathcal{H};\, W; R\rangle \backslash \Sigma_{\texttt{wfti}}$ *then for any* o *such that* $\mathcal{H}(o) = \langle \mathsf{a}; \sigma; \mathcal{S}\rangle$*, it always hold that* $TV(\mathcal{C}(\mathsf{a})) = TV(\mathrm{Dom}(\sigma))$.

Proof. Let $\mathcal{C}(\mathsf{a}) = \forall\langle\alpha_1;\beta_1\rangle \ldots \langle\alpha_n;\beta_n\rangle.\langle\mathcal{K};\mathcal{M};\mathcal{F}\rangle$. Obviously $TV(\mathcal{C}(\mathsf{a})) = \{\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n\}$ by the definition of TV. By assumption, (WFT-InstanceConfig), (WFT-Heap), we know $\mathrm{Dom}(\sigma) = \{\langle\alpha_1;\beta_1\rangle \ldots, \langle\alpha_n;\beta_n\rangle\}$. By the definition of TV, we know $TV(\mathrm{Dom}(\sigma)) = \{\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n\}$. Hence the conclusion holds. □

Lemma 11 (The Domain of σ is Disjoint with Pedigree Type Variables in R)**.** *If* $G, \mathcal{C} \vdash_{\texttt{wftI}} \langle \mathcal{H};\, W; R\rangle \backslash \Sigma_{\texttt{wfti}}$ *then for any* o *such that* $\mathcal{H}(o) = \langle \mathsf{a}; \sigma; \mathcal{S}\rangle$*, it always hold that* $TV(R) \cap TV(\mathrm{Dom}(\sigma)) = \emptyset$.

Proof. By (WFT-InstanceConfig), (WFT-InstanceRel), we know $TV(\mathcal{C}) \cap TV(R) = \emptyset$. By Lem. 10, we know $TV(\mathrm{Dom}(\sigma)) = TV(\mathcal{C}(\mathsf{a}))$. By the definition of TV, we know $TV(\mathcal{C}(\mathsf{a})) \subseteq TV(\mathcal{C})$. Thus $TV(\mathrm{Dom}(\sigma)) \cap TV(R) = \emptyset$. □

Lemma 12 (Pedigree Type Variable Substitution Over Implication)**.** $\Sigma \xrightarrow{\texttt{algebra}} \Sigma'$ *iff* $\Sigma[\sigma] \xrightarrow{\texttt{algebra}} \Sigma'[\sigma]$.

Proof. Trivial. □

C.3 Definition and Properties of the $lazyCons$ Function

Lemma 13 (Set Union with $lazyCons$)**.** $lazyCons(G, \Pi \cup \Pi') = lazyCons(G, \Pi) \cup lazyCons(G, \Pi')$

Proof. If either Π or Π' is $\emptyset$, the lemma follows trivially as $lazyCons(G, \emptyset) = \emptyset$ according to (Def-lazyCons1). Otherwise the lemma directly follows (Def-lazyCons2). □

Lemma 14 (Subsetting with $lazyCons$)**.** *If* $\Pi \subseteq \Pi'$*, then* $lazyCons(G, \Pi) \subseteq lazyCons(G, \Pi')$.

Proof. If Π' is $\emptyset$, then Π is $\emptyset$, the lemma follows trivially as $lazyCons(G, \emptyset) = \emptyset$ according to (Def-lazyCons1) and $\emptyset \subseteq \emptyset$. If Π is $\emptyset$, the lemma still holds as $\emptyset \subseteq lazyCons(G, \Pi')$ for any G and Π'. If neither Π nor Π' is $\emptyset$, the lemma directly follows (Def-lazyCons2).

□

C.4 Properties of Subtyping

Lemma 15. $wfP(\rho) \xrightarrow{\texttt{algebra}} subPed(\rho, \rho)$.

Proof. $subPed(\rho, \rho) = \{\nu_a - \nu_b =_{\mathtt{s}} \nu_a - \nu_b, 0 \leq_{\mathtt{s}} \nu_b \leq_{\mathtt{s}} \nu_b \leq_{\mathtt{s}} 1\}$ given $\rho = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}$. It can be easily seen that $\{\nu_a - \nu_b =_{\mathtt{s}} \nu_a - \nu_b, 0 \leq_{\mathtt{s}} \nu_b \leq_{\mathtt{s}} \nu_b \leq_{\mathtt{s}} 1\} \equiv \{0 \leq_{\mathtt{s}} \nu_b \leq_{\mathtt{s}} 1\}$ as $\nu_a - \nu_b \equiv \nu_a - \nu_b$ (proved via(EL-reflexivity), (EL-subtraction))and $\nu_b \equiv \nu_b$ (proved via (EL-reflexivity)), and (ICS-Tautology), (ICS-leq). At the same time $wfP(\rho) \xrightarrow{\texttt{algebra}} \{0 \leq_{\mathtt{s}} \nu_b \leq_{\mathtt{s}} 1\}$. The conclusion thus holds.

□

Lemma 16 (Subtyping Reflexivity)**.** *If* $\Omega \vdash \tau <: \tau \backslash \Sigma$*, then* $wfT(\tau) \xrightarrow{\texttt{algebra}} \Sigma$.

Proof. Induction on $\Omega \vdash \tau <: \tau \backslash \Sigma$, and case analysis on the last step of the derivation tree leading to the assumption. If it is an instance of (Sub-Bottom), (Sub-Int), (Sub-Unit), $\Sigma = \emptyset$. The conclusion holds trivially.

If the last step is an instance of (Sub-Recursive), let $\tau = \mathsf{a}@\rho$. Notice that $\Sigma = subPed(\rho, \rho)$ and $wfP(\rho) \xrightarrow{\texttt{algebra}} \Sigma$ according to Lem. 15. At the same time $wfT(\tau) = wfP(\rho)$. The conclusion thus holds.

If the last step is an instance of (Sub-Non-Recursive), let $\tau = \mu\mathsf{a}.\mathcal{K}\ @\rho$. $\Sigma = subPed(\rho, \rho) \cup \Sigma_{\mathtt{rest}}$ and $\Sigma_{\mathtt{rest}}$ is obtained by using an instance of (Sub-Connector) for all connector signatures in $\mathcal{K}$. By induction, we can easily know $wfSig(\mathcal{K}) \xrightarrow{\texttt{algebra}} \Sigma_{\mathtt{rest}}$. $wfP(\rho) \xrightarrow{\texttt{algebra}} subPed(\rho, \rho)$ according to Lem. 15. At the same time $wfT(\tau) = wfP(\rho) \cup wfSig(\mathcal{K})$. The conclusion thus holds.

□

Lemma 17 (Subsumption Transitivity)**.** *Given some* $g \geq 0$ *and*

$$\begin{aligned} \{\mathsf{a}_{11} <: \mathsf{a}_{12}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g2}\} &\vdash \tau_1 <: \tau_2 \backslash \Sigma \\ \{\mathsf{a}_{12} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g2} <: \mathsf{a}_{g3}\} &\vdash \tau_2 <: \tau_3 \backslash \Sigma' \\ \mathsf{a}_{11}, \mathsf{a}_{12}, \mathsf{a}_{13}, \ldots \mathsf{a}_{g1}, \mathsf{a}_{g2}, \mathsf{a}_{g3} &\ \textit{distinct} \end{aligned}$$

then $\{\mathsf{a}_{11} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g3}\} \vdash \tau_1 <: \tau_3 \backslash \Sigma''$ *and* $\Sigma \cup \Sigma' \xrightarrow{\texttt{algebra}} \Sigma''$.

Proof. Case analysis on τ.

$\boxed{\tau_1 = \bot}$: By (Sub-Bottom) we know $\{\mathsf{a}_{11} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g3}\} \vdash \bot <: \tau_3 \backslash \emptyset$ holds and it trivially holds

that $\Sigma \cup \Sigma' \xrightarrow{\texttt{algebra}} \emptyset$.

$\boxed{\tau_1 = \textbf{int}}$: By (Sub-Int) and $\{\mathsf{a}_{11} <: \mathsf{a}_{12}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g2}\} \vdash \tau_1 <: \tau_2 \backslash \Sigma$, we know $\Sigma = \emptyset$ and $\tau_2 = \textbf{int}$. Given $\{\mathsf{a}_{12} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g2} <: \mathsf{a}_{g3}\} \vdash \tau_2 <: \tau_3 \backslash \Sigma'$, by (Sub-Int), we know $\tau_3 = \textbf{int}$ and $\Sigma' = \emptyset$. The conclusion holds as $\{\mathsf{a}_{11} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g3}\} \vdash \textbf{int} <: \textbf{int} \backslash \emptyset \cup \emptyset$ according to (Sub-Int).

$\boxed{\tau_1 = \textbf{unit}}$: Analogous to the previous case.

$\boxed{\tau_1 = \mu \mathsf{a}_1.\mathcal{K}_1\ @(\textbf{parent})^{\nu_{11}}(\textbf{child})^{\nu_{12}}}$: By (Sub-Non-Recursive)

$$\begin{aligned}
\tau_2 &= \mu \mathsf{a}_2.\mathcal{K}_2\ @(\textbf{parent})^{\nu_{21}}(\textbf{child})^{\nu_{22}} \\
\mathrm{Dom}(\mathcal{K}_2) &= \{\mathsf{k}_1, \ldots, \mathsf{k}_n\} \\
\mathsf{a}'_1, \mathsf{a}'_2 &\text{ fresh} && \textbf{(17.1)} \\
\mathcal{K}'_1 &= \mathcal{K}_1\{\mathsf{a}'_1/\mathsf{a}_1\} && \textbf{(17.2)} \\
\mathcal{K}'_2 &= \mathcal{K}_2\{\mathsf{a}'_2/\mathsf{a}_2\} && \textbf{(17.3)} \\
\forall i \in \{1, \ldots, n\}.\ \{\mathsf{a}_{11} <: \mathsf{a}_{12}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g2}\} \cup \{\mathsf{a}'_1 <: \mathsf{a}'_2\} &\vdash_{\mathsf{c}} \mathcal{K}'_1(\mathsf{k}_i) <: \mathcal{K}'_2(\mathsf{k}_i) \backslash \Sigma_i && \textbf{(17.4)} \\
\Sigma &= \Sigma_1 \cup \ldots \Sigma_n \cup \{\nu_{11} - \nu_{12} =_{\mathsf{s}} \nu_{21} - \nu_{22}, 0 \leq_{\mathsf{s}} \nu_{12} \leq_{\mathsf{s}} \nu_{22} \leq_{\mathsf{s}} 1\}
\end{aligned}$$

Now that τ_2 is an instance type and $\Omega \vdash \tau_2 <: \tau_3 \backslash \Sigma'$ holds. By (Sub-Non-Recursive) we know we can let

$$\tau_3 = \mu \mathsf{a}_3.\mathcal{K}_3\ @(\textbf{parent})^{\nu_{31}}(\textbf{child})^{\nu_{32}}$$

and $\mathrm{Dom}(\mathcal{K}_3) \subseteq \mathrm{Dom}(\mathcal{K}_2)$ and for some $p \leq n$ we know

$$\begin{aligned}
\mathrm{Dom}(\mathcal{K}_3) &= \{\mathsf{k}_1, \ldots, \mathsf{k}_p\} \\
\mathsf{a}'_3 &\text{ fresh} && \textbf{(17.5)} \\
\mathcal{K}'_3 &= \mathcal{K}_3\{\mathsf{a}'_3/\mathsf{a}_3\} && \textbf{(17.6)} \\
\forall j \in \{1, \ldots, p\}.\ \{\mathsf{a}_{12} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g2} <: \mathsf{a}_{g3}\} \cup \{\mathsf{a}'_2 <: \mathsf{a}'_3\} &\vdash_{\mathsf{c}} \mathcal{K}'_2(\mathsf{k}_j) <: \mathcal{K}'_3(\mathsf{k}_j) \backslash \Sigma'_j && \textbf{(17.7)} \\
\Sigma' &= \Sigma'_1 \cup \ldots \Sigma'_p \cup \{\nu_{21} - \nu_{22} =_{\mathsf{s}} \nu_{31} - \nu_{32}, 0 \leq_{\mathsf{s}} \nu_{22} \leq_{\mathsf{s}} \nu_{32} \leq_{\mathsf{s}} 1\}
\end{aligned}$$

From **(17.4)**, we know $\mathrm{Dom}(\mathcal{K}'_2) \subseteq \mathrm{Dom}(\mathcal{K}'_1)$. Hence $\mathrm{Dom}(\mathcal{K}'_3) \subseteq \mathrm{Dom}(\mathcal{K}'_1)$. Now for any $j \in \{1, \ldots, p\}$, let $\mathcal{K}'_1(\mathsf{k}_j) = \langle \mathcal{I}_{(j)(1)}; \mathcal{E}_{(j)(1)} \rangle$, $\mathcal{K}'_2(\mathsf{k}_j) = \langle \mathcal{I}_{(j)(2)}; \mathcal{E}_{(j)(2)} \rangle$, $\mathcal{K}'_3(\mathsf{k}_j) = \langle \mathcal{I}_{(j)(3)}; \mathcal{E}_{(j)(3)} \rangle$.

Finding Relations between $\mathcal{E}_{(j)(1)}$ and $\mathcal{E}_{(j)(3)}$:

From **(17.7)** and (Sub-Connector), $\mathrm{Dom}(\mathcal{E}_{(j)(3)}) \subseteq \mathrm{Dom}(\mathcal{E}_{(j)(2)})$. From **(17.4)** and (Sub-Connector), $\mathrm{Dom}(\mathcal{E}_{(j)(2)}) \subseteq \mathrm{Dom}(\mathcal{E}_{(j)(1)})$. Hence $\mathrm{Dom}(\mathcal{E}_{(j)(3)}) \subseteq \mathrm{Dom}(\mathcal{E}_{(j)(1)})$. Let $\mathrm{Dom}(\mathcal{E}_{(j)(3)}) = \{\mathsf{m}_1, \ldots, \mathsf{m}_q\}$, for any $u \in \{1, \ldots, q\}$, let

$$\mathcal{E}_{(j)(1)}(\mathsf{m}_u) = \tau'_{(u)(j)(e1)} \rightarrow \tau_{(u)(j)(e1)}$$

$$\mathcal{E}_{(j)(2)}(\mathsf{m}_u) = \tau'_{(u)(j)(e2)} \to \tau_{(u)(j)(e2)}$$
$$\mathcal{E}_{(j)(3)}(\mathsf{m}_u) = \tau'_{(u)(j)(e3)} \to \tau_{(u)(j)(e3)}$$

From (**17.7**) and (Sub-Connector) and (Sub-Method) we know

$$\begin{array}{r}\{\mathsf{a}_{12} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g2} <: \mathsf{a}_{g3}\} \cup \{\mathsf{a}'_2 <: \mathsf{a}'_3\} \vdash \tau'_{(u)(j)(e3)} <: \tau'_{(u)(j)(e2)} \backslash \Sigma_{(u)(j)\texttt{econtra23}} \\ \{\mathsf{a}_{12} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g2} <: \mathsf{a}_{g3}\} \cup \{\mathsf{a}'_2 <: \mathsf{a}'_3\} \vdash \tau_{(u)(j)(e2)} <: \tau_{(u)(j)(e3)} \backslash \Sigma_{(u)(j)\texttt{eco23}} \\ \Sigma_{(1)(j)\texttt{eco23}} \cup \Sigma_{(1)(j)\texttt{econtra23}} \cdots \cup \Sigma_{(q)(j)\texttt{eco23}} \cup \Sigma_{(q)(j)\texttt{econtra23}} \subseteq \Sigma'_j\end{array} \quad \textbf{(17.8)}$$

Similarly, from (**17.4**) and (Sub-Connector) and (Sub-Method) we know

$$\begin{array}{r}\{\mathsf{a}_{11} <: \mathsf{a}_{12}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g2}\} \cup \{\mathsf{a}'_1 <: \mathsf{a}'_2\} \vdash \tau'_{(u)(j)(e2)} <: \tau'_{(u)(j)(e1)} \backslash \Sigma_{(u)(j)\texttt{econtra12}} \\ \{\mathsf{a}_{11} <: \mathsf{a}_{12}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g2}\} \cup \{\mathsf{a}'_1 <: \mathsf{a}'_2\} \vdash \tau_{(u)(j)(e1)} <: \tau_{(u)(j)(e2)} \backslash \Sigma_{(u)(j)\texttt{eco12}} \\ \Sigma_{(1)(j)\texttt{eco12}} \cup \Sigma_{(1)(j)\texttt{econtra12}} \cdots \cup \Sigma_{(q)(j)\texttt{eco12}} \cup \Sigma_{(q)(j)\texttt{econtra12}} \subseteq \Sigma_j\end{array} \quad \textbf{(17.9)}$$

By (**17.8**), (**17.9**), and the induction hypothesis, we know for $u \in \{1, \ldots, q\}$,

$$\begin{array}{r}\{\mathsf{a}_{11} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g3}, \mathsf{a}'_1 <: \mathsf{a}'_3\} \vdash \tau_{(u)(j)(e1)} <: \tau_{(u)(j)(e3)} \backslash \Sigma_{(u)(j)\texttt{eco13}} \\ \Sigma_{(u)(j)\texttt{eco12}} \cup \Sigma_{(u)(j)\texttt{eco23}} \xrightarrow{\texttt{algebra}} \Sigma_{(u)(j)\texttt{eco13}}\end{array}$$

Similarly, we know for $u \in \{1, \ldots, q\}$,

$$\begin{array}{r}\{\mathsf{a}_{11} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g3}, \mathsf{a}'_1 <: \mathsf{a}'_3\} \vdash \tau'_{(u)(j)(e3)} <: \tau'_{(u)(j)(e1)} \backslash \Sigma_{(u)(j)\texttt{econtra13}} \\ \Sigma_{(u)(j)\texttt{econtra12}} \cup \Sigma_{(u)(j)\texttt{econtra23}} \xrightarrow{\texttt{algebra}} \Sigma_{(u)(j)\texttt{econtra13}}\end{array}$$

Obviously,

$$\Sigma_j \cup \Sigma'_j \xrightarrow{\texttt{algebra}} \Sigma_{(1)(j)\texttt{eco13}} \cup \Sigma_{(1)(j)\texttt{econtra13}} \cup \ldots \Sigma_{(q)(j)\texttt{eco13}} \cup \Sigma_{(q)(j)\texttt{econtra13}}$$

Finding Relations between $\mathcal{I}_{(j)(1)}$ and $\mathcal{I}_{(j)(3)}$:

From (**17.7**), (**17.4**), and (Sub-Connector), Obvioulsy $\mathrm{Dom}(\mathcal{I}_{(j)(1)}) \subseteq \mathrm{Dom}(\mathcal{I}_{(j)(3)})$. Let $\mathrm{Dom}(\mathcal{I}_{(j)(1)}) = \{\mathsf{m}_1, \ldots, \mathsf{m}_r\}$, for any $v \in \{1, \ldots, r\}$, let

$$\mathcal{I}_{(j)(1)}(\mathsf{m}_v) = \tau'_{(v)(j)(i1)} \to \tau_{(v)(j)(i1)}$$
$$\mathcal{I}_{(j)(2)}(\mathsf{m}_v) = \tau'_{(v)(j)(i2)} \to \tau_{(v)(j)(i2)}$$
$$\mathcal{I}_{(j)(3)}(\mathsf{m}_v) = \tau'_{(v)(j)(i3)} \to \tau_{(v)(j)(i3)}$$

In the analogous way as we showed earlier for types on export signatures, we know for $v \in \{1, \ldots, r\}$

$$\{\mathsf{a}_{11} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g3}, \mathsf{a}'_1 <: \mathsf{a}'_3\} \vdash \tau_{(v)(j)(i3)} <: \tau_{(v)(j)(i1)} \backslash \Sigma_{(v)(j)\mathtt{ico13}}$$
$$\Sigma_{(v)(j)\mathtt{ico12}} \cup \Sigma_{(v)(j)\mathtt{ico23}} \xrightarrow{\mathtt{algebra}} \Sigma_{(v)(j)\mathtt{ico13}}$$

and $v \in \{1, \ldots, r\}$ we have

$$\{\mathsf{a}_{11} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g3}, \mathsf{a}'_1 <: \mathsf{a}'_3\} \vdash \tau'_{(v)(j)(i1)} <: \tau'_{(v)(j)(i3)} \backslash \Sigma_{(u)(j)\mathtt{econtra13}}$$
$$\Sigma_{(v)(j)\mathtt{icontra12}} \cup \Sigma_{(v)(j)\mathtt{icontra23}} \xrightarrow{\mathtt{algebra}} \Sigma_{(v)(j)\mathtt{icontra13}}$$

and

$$\Sigma_j \cup \Sigma'_j \xrightarrow{\mathtt{algebra}} \Sigma_{(1)(j)\mathtt{ico13}} \cup \Sigma_{(1)(j)\mathtt{icontra13}} \cup \ldots \Sigma_{(r)(j)\mathtt{ico13}} \cup \Sigma_{(r)(j)\mathtt{icontra13}}$$

Conclusion:

From the final results of the last two subgoals and (Sub-Connector), we know for $j \in \{1, \ldots, p\}$

$$\{\mathsf{a}_{11} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g3}, \mathsf{a}'_1 <: \mathsf{a}'_3\} \vdash_{\mathsf{c}} \mathcal{K}'_1(\mathsf{k}_j) <: \mathcal{K}'_3(\mathsf{k}_j) \backslash \Sigma''_j \quad \textbf{(17.10)}$$
$$\Sigma_j \cup \Sigma'_j \xrightarrow{\mathtt{algebra}} \Sigma''_j \quad \textbf{(17.11)}$$

Thus by (Sub-Non-Recursive), we know

$$\{\mathsf{a}_{11} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g3}\}\tau_1 <: \tau_3 \vdash \Sigma''$$
$$\Sigma'' = \Sigma''_1 \cup \ldots, \Sigma''_p \cup \{\nu_{11} - \nu_{12} =_{\mathsf{s}} \nu_{31} - \nu_{32}, 0 \leq_{\mathsf{s}} \nu_{12} \leq_{\mathsf{s}} \nu_{32} \leq_{\mathsf{s}} 1\}$$

Obviously

$$\left\{ \begin{array}{l} \nu_{11} - \nu_{12} =_{\mathsf{s}} \nu_{21} - \nu_{22} \\ 0 \leq_{\mathsf{s}} \nu_{12} \leq_{\mathsf{s}} \nu_{22} \leq_{\mathsf{s}} 1 \\ \nu_{21} - \nu_{22} =_{\mathsf{s}} \nu_{31} - \nu_{32} \\ 0 \leq_{\mathsf{s}} \nu_{22} \leq_{\mathsf{s}} \nu_{32} \leq_{\mathsf{s}} 1 \end{array} \right\} \xrightarrow{\mathtt{algebra}} \left\{ \begin{array}{l} \nu_{11} - \nu_{12} =_{\mathsf{s}} \nu_{31} - \nu_{32} \\ 0 \leq_{\mathsf{s}} \nu_{12} \leq_{\mathsf{s}} \nu_{32} \leq_{\mathsf{s}} 1 \end{array} \right\}$$

And the conclusion holds.

$\boxed{\tau_1 = \mathsf{a}_1@(\textbf{parent})^{\nu_{11}}(\textbf{child})^{\nu_{12}}}$ By (Sub-Recursive), τ_2 must also be a recursive instance type. Let $\tau_2 = \mathsf{a}_2@(\textbf{parent})^{\nu_{21}}(\textbf{child})^{\nu_{22}}$. We also know there exists some $w \in \{1, \ldots, g\}$ and $\mathsf{a}_1 = \mathsf{a}_{w1}$ and $\mathsf{a}_2 = \mathsf{a}_{w2}$. Similarly, we can let $\tau_3 = \mathsf{a}_3@(\textbf{parent})^{\nu_{31}}(\textbf{child})^{\nu_{32}}$. We also know there exists some $\mathsf{a}_3 = \mathsf{a}_{w3}$. Thus,

$$\{\mathsf{a}_{11} <: \mathsf{a}_{13}, \ldots \mathsf{a}_{w1} <: \mathsf{a}_{w3}, \ldots \mathsf{a}_{g1} <: \mathsf{a}_{g3}\} \vdash \mathsf{a}_{w1} <: \mathsf{a}_{w3} \backslash \Sigma''$$

and

$$\begin{aligned}
&\Sigma \cup \Sigma' \\
=&\{\nu_{11} - \nu_{12} =_{\mathtt{s}} \nu_{21} - \nu_{22}, 0 \leq_{\mathtt{s}} \nu_{12} \leq_{\mathtt{s}} \nu_{22} \leq_{\mathtt{s}} 1\} \\
&\cup \{\nu_{21} - \nu_{22} =_{\mathtt{s}} \nu_{31} - \nu_{32}, 0 \leq_{\mathtt{s}} \nu_{22} \leq_{\mathtt{s}} \nu_{32} \leq_{\mathtt{s}} 1\} \\
\xrightarrow{\texttt{algebra}} &\{\nu_{11} - \nu_{12} =_{\mathtt{s}} \nu_{31} - \nu_{32}, 0 \leq_{\mathtt{s}} \nu_{12} \leq_{\mathtt{s}} \nu_{32} \leq_{\mathtt{s}} 1\} \\
=&\Sigma''
\end{aligned}$$

□

Lemma 18 (Subsumption Transitivity Over Expression Types)**.** *Given*

$$\begin{aligned}
\emptyset \vdash \tau_1 <: \tau_2 \backslash \Sigma \\
\emptyset \vdash \tau_2 <: \tau_3 \backslash \Sigma'
\end{aligned}$$

then $\emptyset \vdash \tau_1 <: \tau_3 \backslash \Sigma''$ *and* $\Sigma \cup \Sigma' \xrightarrow{\texttt{algebra}} \Sigma''$.

Proof. A special case of Lem. 17. □

Lemma 19 (Finite Instances of the Subsumption Rule on Deduction Tree)**.** *If* $\Gamma, \mathcal{C} \vdash exd : \tau \backslash \Sigma, \Pi$ *and* $\tau \neq \mathsf{k}$*, then there exists a subdeduction of* $\Gamma, \mathcal{C} \vdash exd : \tau' \backslash \Sigma', \Pi$ *where* $\vdash \tau' <: \tau \backslash \Sigma_{\mathtt{sub}}$ *such that the last step in the subdeduction is an instance of exd's corresponding type rule, such as if* $exd = x$*, then the type rule is* (T-VAR)*; if* $exd = o$*, then the type rule is* (T-RID), etc*, and* $\Sigma \xrightarrow{\texttt{algebra}} \Sigma' \cup \Sigma_{\mathtt{sub}}$.

Proof. Case analysis on *exd*, we know each form of expression clearly must have at least one corresponding type rule. In addition, each form of expression clearly must have an instance of its corresponding type rule in the deduction. With a type system with subsumption, the most general form of the deduction tree for $\Gamma, \mathcal{C} \vdash exd : \tau \backslash \Sigma, \Pi$ has the shape illustrated below when $\tau \neq \mathsf{k}$ (if $\tau = \mathsf{k}$, the last step of deduction can not be an instance of (T-T-Sub) because there are not typing rules to subtype connection handle types):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \tau_1 <: \tau \backslash \Sigma_{\texttt{sub1}}} \quad \dfrac{\dfrac{\dfrac{\dfrac{\text{some predicates}}{\Gamma, \mathcal{C} \vdash \mathit{exd} : \tau' \backslash \Sigma', \Pi}}{\emptyset \vdash \tau' <: \tau_n \backslash \Sigma'_{\texttt{sub}} \quad \Gamma, \mathcal{C} \vdash \mathit{exd} : \tau' \backslash \Sigma', \Pi}\ \text{(T-Sub)}}{\Gamma, \mathcal{C} \vdash \mathit{exd} : \tau_n \backslash \Sigma' \cup \Sigma'_{\texttt{sub}}, \Pi}\ \text{(Some Rule)}}{\vdots}}{\emptyset \vdash \tau_2 <: \tau_1 \backslash \Sigma_{\texttt{sub2}} \quad \Gamma, \mathcal{C} \vdash \mathit{exd} : \tau_1 \backslash \Sigma' \cup \Sigma_{\texttt{sub2}} \cup \ldots \Sigma_{\texttt{subn}} \cup \Sigma'_{\texttt{sub}}, \Pi}\ \text{(T-Sub)}}{\Gamma, \mathcal{C} \vdash \mathit{exd} : \tau \backslash \Sigma' \cup \Sigma_{\texttt{sub1}} \cup \ldots \Sigma_{\texttt{subn}} \cup \Sigma'_{\texttt{sub}}, \Pi}\ \text{(T-Sub)}$$

The last n steps involve only the application of subsumption, and we know $\Sigma = \Sigma' \cup \Sigma_{\texttt{sub1}} \cup \ldots \Sigma_{\texttt{subn}} \cup \Sigma'_{\texttt{sub}}$. Given the judgments of $\emptyset \vdash \tau_1 <: \tau \backslash \Sigma_{\texttt{sub1}}, \emptyset \vdash \tau_2 <: \tau_1 \backslash \Sigma_{\texttt{sub2}}, \ldots, \emptyset \vdash \tau' <: \tau_n \backslash \Sigma'_{\texttt{sub}}$ and subtyping transitivity (Lem. 18), we know $\Sigma_{\texttt{sub1}} \cup \ldots \Sigma_{\texttt{subn}} \cup \Sigma'_{\texttt{sub}} \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{sub}}$. Thus $\Sigma \xrightarrow{\texttt{algebra}} \Sigma' \cup \Sigma_{\texttt{sub}}$ holds by (ICS-union).

□

Lemma 20 (Instantiation Preserves Subtyping)**.** *If* $TV(\tau) \subseteq TV(\mathrm{Dom}(\sigma))$*, then* $\emptyset \vdash \tau[\sigma] <: \tau \backslash \Sigma_{\texttt{inst}}$*, and* $wfT(\tau)[\sigma] \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{inst}}[\sigma]$ *for any* σ*.*

Proof. Case analysis on subtyping rules. □

Lemma 21 (Substitution Preserves Subtyping)**.** *If* $\emptyset \vdash \tau <: \tau' \backslash \Sigma$*, then* $\emptyset \vdash \tau[\sigma] <: \tau'[\sigma] \backslash \Sigma[\sigma]$*.*

Proof. Case analysis on subtyping rules. □

Lemma 22 (Unfolding Preserves Subtyping)**.** *If* $\emptyset \vdash \mu \mathsf{a}^{\mathbf{X}}.\mathcal{K}^{\mathbf{X}}\ @\rho^{\mathbf{X}} <: \mu \mathsf{a}^{\mathbf{Y}}.\mathcal{K}^{\mathbf{Y}}\ @\rho^{\mathbf{Y}} \backslash \Sigma$*, then*

$$\emptyset \vdash \mu \mathsf{a}^{\mathbf{X}}.\mathcal{K}^{\mathbf{X}}\{\mathsf{a}^{\mathbf{X}} \circlearrowleft \mathcal{K}^{\mathbf{X}}\}\ @\rho^{\mathbf{X}} <: \mu \mathsf{a}^{\mathbf{Y}}.\mathcal{K}^{\mathbf{Y}}\{\mathsf{a}^{\mathbf{Y}} \circlearrowleft \mathcal{K}^{\mathbf{Y}}\}\ @\rho^{\mathbf{Y}} \backslash \Sigma$$

Proof. Straightforward according to (Sub-Non-Recursive). Note that the notion of unfolding means that the derivation tree leading to $\emptyset \vdash \mu \mathsf{a}^{\mathbf{X}}.\mathcal{K}^{\mathbf{X}}\{\mathsf{a}^{\mathbf{X}} \circlearrowleft \mathcal{K}^{\mathbf{X}}\}\ @\rho^{\mathbf{X}} <: \mu \mathsf{a}^{\mathbf{Y}}.\mathcal{K}^{\mathbf{Y}}\{\mathsf{a}^{\mathbf{Y}} \circlearrowleft \mathcal{K}^{\mathbf{Y}}\}\ @\rho^{\mathbf{Y}} \backslash \Sigma$ can be obtained by replacing every instance of (Sub-Recursive) on the subtree of the derivation tree leading to $\emptyset \vdash \mu \mathsf{a}^{\mathbf{X}}.\mathcal{K}^{\mathbf{X}}\ @\rho^{\mathbf{X}} <: \mu \mathsf{a}^{\mathbf{Y}}.\mathcal{K}^{\mathbf{Y}}\ @\rho^{\mathbf{Y}} \backslash \Sigma$ with the entire tree leading to $\emptyset \vdash \mu \mathsf{a}^{\mathbf{X}}.\mathcal{K}^{\mathbf{X}}\ @\rho^{\mathbf{X}} <: \mu \mathsf{a}^{\mathbf{Y}}.\mathcal{K}^{\mathbf{Y}}\ @\rho^{\mathbf{Y}} \backslash \Sigma$. The constraints generated for every such subtree is simply a repeat of the original constraints. Hence the constraints stay put for the two derivations. □

C.5 Decidable Type Checking

Lemma 23 (Decidable Type Checking)**.**

$$\Gamma, \mathcal{C} \vdash \textit{exd} : \tau \backslash \Sigma^{\mathbf{X}}, \Pi^{\mathbf{X}} \quad \textbf{(23.1)}$$

$$\Gamma, \mathcal{C} \vdash \textit{exd} : \tau \backslash \Sigma^{\mathbf{Y}}, \Pi^{\mathbf{Y}} \quad \textbf{(23.2)}$$

the last steps of derivation leading to (**23.1**) *and* (**23.2**) *are not an instance of* (T-Sub) (**23.3**)

then $\Sigma^{\mathbf{X}} = \Sigma^{\mathbf{Y}}$ *and* $\Pi^{\mathbf{X}} = \Pi^{\mathbf{Y}}$.

Proof. Case analysis on (**23.2**). For each form of *exd*, there are only two type rules that can lead to the judgment, (T-Sub) or a unique expression type checking rule for that specific form $\textit{exd}_1$. The first one is not possible, and hence (**23.1**) and (**23.2**) are from the same derivation. The conclusion holds by induction. □

C.6 Properties of the $convert$ Function

We start off defining a weaker form of $subPed$, namely $subPedWeak$, since in the following lemmas, the stronger constraint computed by $subPed$ is not needed.

$$\begin{aligned} subPedWeak(\rho_1, \rho_2) \stackrel{\text{def}}{=} & \{\nu_{11} - \nu_{12} =_{\mathbf{s}} \nu_{21} - \nu_{22}, \nu_{12} \leq_{\mathbf{s}} \nu_{22}\} \\ & \rho_1 = (\textbf{parent})^{\nu_{11}} (\textbf{child})^{\nu_{12}} \\ & \rho_2 = (\textbf{parent})^{\nu_{21}} (\textbf{child})^{\nu_{22}} \end{aligned}$$

Lemma 24. $subPed(\rho_1, \rho_2) \xrightarrow{\texttt{algebra}} subPedWeak(\rho_1, \rho_2)$

Proof. Note that the only difference between the two functions is that $subPedWeak$ does not need the negative levels of ρ_1 and ρ_2 to range over $\{0, 1\}$. □

Lemma 25. *If* $\emptyset \vdash c\tau^{\mathbf{X}} <: c\tau^{\mathbf{Y}} \backslash \Sigma$*, then* $convert(c\tau^{\mathbf{X}}, \rho)$ *is defined iff* $convert(c\tau^{\mathbf{Y}}, \rho)$ *is defined.*

Proof. Case analysis on the assumption judgment. According to the definition of the meta-variable $c\tau$, we know $c\tau^{\mathbf{X}} \neq \bot$. By the definition of subtyping rules, there are four possibilities for the last step of deduction leading to it: an instance of (Sub-Int), an instance of (Sub-Unit), and an instance of (Sub-Non-Recursive)

and (Sub-Recursive). For the first three cases, $convert$ is defined for both. The last case is not possible since $\emptyset$ is used in the judgment. □

Lemma 26. *Given*

$$\begin{aligned} convert(c\tau, \rho^{\mathbf{X}}) &= c\tau^{\mathbf{X}} \\ \emptyset \vdash c\tau^{\mathbf{X}} &<: c\tau^{\mathbf{Z}} \backslash \Sigma^{\mathbf{X}} \end{aligned} \tag{26.1}$$

then

$$\begin{aligned} convert(c\tau, \rho^{\mathbf{Y}}) &= c\tau^{\mathbf{Y}} \\ \emptyset \vdash c\tau^{\mathbf{Y}} &<: c\tau^{\mathbf{Z}} \backslash \Sigma^{\mathbf{Y}} \\ \Sigma^{\mathbf{X}} \cup subPedWeak(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) &\xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}} \end{aligned}$$

Proof. According to the definition of $convert$, there are three possibilities for $c\tau$, be **int** or **unit**, or be $\mu \mathsf{a}.\mathcal{K}\ @\rho$. We now conduct case analysis on $c\tau$.

If $c\tau = \textbf{int}$, according to the assumption and the definition of $convert$, we know $c\tau^{\mathbf{X}} = \textbf{int}$. Thus we know $convert(c\tau, \rho^{\mathbf{Y}}) = convert(\textbf{int}, \rho^{\mathbf{Y}})$ is defined, hence $c\tau^{\mathbf{Y}} = \textbf{int}$. The last step of deduction of (**26.1**) must be an instance of (Sub-Int). Hence $c\tau^{\mathbf{Z}} = \textbf{int}$ and $\Sigma^{\mathbf{X}} = \emptyset$. Hence obviously we know $\emptyset \vdash c\tau^{\mathbf{Y}} <: c\tau^{\mathbf{Z}} \backslash \Sigma^{\mathbf{Y}}$ and $\Sigma^{\mathbf{Y}} = \emptyset$. The conclusion holds trivially.

If $c\tau = \textbf{unit}$, the conclusion holds in the same way as the case where $c\tau = \textbf{int}$.

If $c\tau = \mu \mathsf{a}.\mathcal{K}\ @\rho$, first let $\rho^{\mathbf{X}} = (\textbf{parent})^{\nu_1^{\mathbf{X}}}(\textbf{child})^{\nu_2^{\mathbf{X}}}$, and $\rho^{\mathbf{Y}} = (\textbf{parent})^{\nu_1^{\mathbf{Y}}}(\textbf{child})^{\nu_2^{\mathbf{Y}}}$, and $\rho = (\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2}$. By the definition of $convert$, $c\tau^{\mathbf{X}} = \mu \mathsf{a}.\mathcal{K}\ @(\textbf{parent})^{\nu_1 + \nu_1^{\mathbf{X}} - \nu_2^{\mathbf{X}}}(\textbf{child})^{\nu_2}$. $convert(c\tau, \rho^{\mathbf{Y}})$ obviously is defined, and $c\tau^{\mathbf{Y}} = \mu \mathsf{a}.\mathcal{K}\ @(\textbf{parent})^{\nu_1 + \nu_1^{\mathbf{Y}} - \nu_2^{\mathbf{Y}}}(\textbf{child})^{\nu_2}$ Note that the last step of deduction for (**26.1**) must be an instance of (Sub-Non-Recursive). Let $c\tau^{\mathbf{Z}} = \mu \mathsf{a}^{\mathbf{Z}}.\mathcal{K}^{\mathbf{Z}}\ @\rho^{\mathbf{Z}}$, and $\rho^{\mathbf{Z}} = (\textbf{parent})^{\nu_1^{\mathbf{Z}}}(\textbf{child})^{\nu_2^{\mathbf{Z}}}$. We know

$$\begin{aligned} \mathrm{Dom}(\mathcal{K}^{\mathbf{Z}}) &= \{\mathsf{k}_1, \ldots, \mathsf{k}_n\} \\ \forall i \in \{1, \ldots, n\}\ \{\mathsf{a} <: \mathsf{a}^{\mathbf{Z}}\} &\vdash_{\mathsf{c}} \mathcal{K}(\mathsf{k}_i) <: \mathcal{K}^{\mathbf{Z}}(\mathsf{k}_i) \backslash \Sigma_i \\ \Sigma^{\mathbf{X}} = \Sigma_1 \cup \ldots \Sigma_n \cup \{\nu_1 + \nu_1^{\mathbf{X}} - \nu_2^{\mathbf{X}} - \nu_2 =_{\mathsf{s}} \nu_1^{\mathbf{Z}} - \nu_2^{\mathbf{Z}}\} &\cup \{0 \leq_{\mathsf{s}} \nu_2 \leq_{\mathsf{s}} \nu_2^{\mathbf{Z}} \leq_{\mathsf{s}} 1\} \end{aligned}$$

According to (Sub-Non-Recursive) we also know

$$\begin{aligned} \emptyset \vdash c\tau^{\mathbf{Y}} &<: c\tau^{\mathbf{Z}} \backslash \Sigma^{\mathbf{Y}} \\ \Sigma^{\mathbf{Y}} = \Sigma_1 \cup \ldots \Sigma_n \cup \{\nu_1 + \nu_1^{\mathbf{Y}} - \nu_2^{\mathbf{Y}} - \nu_2 =_{\mathsf{s}} \nu_1^{\mathbf{Z}} - \nu_2^{\mathbf{Z}}\} &\cup \{\{0 \leq_{\mathsf{s}} \nu_2 \leq_{\mathsf{s}} \nu_2^{\mathbf{Z}} \leq_{\mathsf{s}} 1\} \end{aligned}$$

By the definition of $subPedWeak$, we know

$$subPedWeak(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) = \{\nu_1^{\mathbf{Y}} - \nu_2^{\mathbf{Y}} =_{\mathtt{s}} \nu_1^{\mathbf{X}} - \nu_2^{\mathbf{X}}\} \cup \{\nu_2^{\mathbf{Y}} \leq_{\mathtt{s}} \nu_2^{\mathbf{X}}\}$$

The conclusion obviously holds as

$$\left.\begin{array}{l} \nu_1 + \nu_1^{\mathbf{X}} - \nu_2^{\mathbf{X}} - \nu_2 =_{\mathtt{s}} \nu_1^{\mathbf{Z}} - \nu_2^{\mathbf{Z}} \\ 0 \leq_{\mathtt{s}} \nu_2 \leq_{\mathtt{s}} \nu_2^{\mathbf{Z}} \leq_{\mathtt{s}} 1 \\ \nu_1^{\mathbf{Y}} - \nu_2^{\mathbf{Y}} =_{\mathtt{s}} \nu_1^{\mathbf{X}} - \nu_2^{\mathbf{X}} \\ \nu_2^{\mathbf{Y}} \leq_{\mathtt{s}} \nu_2^{\mathbf{X}} \end{array}\right\} \xrightarrow{\texttt{algebra}} \left\{\begin{array}{l} \nu_1 + \nu_1^{\mathbf{Y}} - \nu_2^{\mathbf{Y}} - \nu_2 =_{\mathtt{s}} \nu_1^{\mathbf{Z}} - \nu_2^{\mathbf{Z}} \\ 0 \leq_{\mathtt{s}} \nu_2 \leq_{\mathtt{s}} \nu_2^{\mathbf{Z}} \leq_{\mathtt{s}} 1 \end{array}\right.$$

especially according to (ECS-equal) and (ICS-union).

□

Lemma 27. *Given* $\emptyset \vdash c\tau^{\mathbf{Z}} <: c\tau^{\mathbf{U}} \backslash \Sigma_{\mathtt{sub}}$, $wfcons(c\tau^{\mathbf{U}}, \rho^{\mathbf{X}}) = \Sigma^{\mathbf{UX}}$ *and* $\emptyset \vdash convert(c\tau^{\mathbf{U}}, \rho^{\mathbf{X}}) <: c\tau \backslash \Sigma$, *then*

$$dwfcons(c\tau^{\mathbf{Z}}, c\tau, \rho^{\mathbf{Y}}) = \Sigma^{\mathbf{ZY}}$$
$$\Sigma_{\mathtt{sub}} \cup \Sigma \cup \Sigma^{\mathbf{UX}} \cup subPedWeak(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{ZY}}$$

Proof. According to the definition of $wfcons$, there are three possibilities for $c\tau^{\mathbf{U}}$, be **int** or **unit**, or be $\mu\mathsf{a}.\mathcal{K}\ @\rho$. We now conduct case analysis on $c\tau^{\mathbf{U}}$.

If $c\tau^{\mathbf{U}} =$ **int**, by definition of $convert$, we know $convert(c\tau^{\mathbf{U}}, \rho^{\mathbf{Y}}) =$ **int**. By subtyping rules, we know $c\tau =$ **int**. Also by subtyping rules we know $c\tau^{\mathbf{Z}} =$ **int**. Therefore $dwfcons(c\tau^{\mathbf{Z}}, c\tau, \rho^{\mathbf{Y}})$ is defined and $\Sigma^{\mathbf{ZY}} = \emptyset$. The conclusion holds trivially.

If $c\tau^{\mathbf{U}} =$ **unit**, the conclusion holds in the same way as the case where $c\tau^{\mathbf{U}} =$ **int**.

If $c\tau^{\mathbf{U}} = \mu\mathsf{a}^{\mathbf{U}}.\mathcal{K}^{\mathbf{U}}\ @\rho^{\mathbf{U}}$, then by subtyping rules, we know $c\tau^{\mathbf{Z}}$ must also be an instance type in the μ form. $c\tau^{\mathbf{Z}} = \mu\mathsf{a}^{\mathbf{Z}}.\mathcal{K}^{\mathbf{Z}}\ @\rho^{\mathbf{Z}}$. Let $\rho^{\mathbf{U}} = (\textbf{parent})^{\nu_1^{\mathbf{U}}}(\textbf{child})^{\nu_2^{\mathbf{U}}}$ and $\rho^{\mathbf{Z}} = (\textbf{parent})^{\nu_1^{\mathbf{Z}}}(\textbf{child})^{\nu_2^{\mathbf{Z}}}$. According to (Sub-Non-Recursive), we know $\{\nu_1^{\mathbf{Z}} - \nu_2^{\mathbf{Z}} =_{\mathtt{s}} \nu_1^{\mathbf{U}} - \nu_2^{\mathbf{U}}\} \cup \{\nu_2^{\mathbf{Z}} \leq_{\mathtt{s}} \nu_2^{\mathbf{U}}\} \subseteq \Sigma_{\mathtt{sub}}$. Now let $c\tau = \mu\mathsf{a}'.\mathcal{K}'\ @\rho'$ and $\rho' = (\textbf{parent})^{\nu_1'}(\textbf{child})^{\nu_2'}$. Also let $\rho^{\mathbf{X}} = (\textbf{parent})^{\nu_1^{\mathbf{X}}}(\textbf{child})^{\nu_2^{\mathbf{X}}}$, and $\rho^{\mathbf{Y}} = (\textbf{parent})^{\nu_1^{\mathbf{Y}}}(\textbf{child})^{\nu_2^{\mathbf{Y}}}$. By the definition of $convert$, $\Sigma^{\mathbf{UX}} = \{\nu_2^{\mathbf{X}} \leq_{\mathtt{s}} \nu_1^{\mathbf{U}}\}$ and $\Sigma^{\mathbf{ZY}} = \{\nu_2^{\mathbf{Y}} + \nu_2^{\mathbf{Z}} - \nu_1^{\mathbf{Z}} \leq_{\mathtt{s}} \nu_2'\}$. By the definition of $subPedWeak$, we know

$$subPedWeak(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) = \{\nu_1^{\mathbf{Y}} - \nu_2^{\mathbf{Y}} =_{\mathtt{s}} \nu_1^{\mathbf{X}} - \nu_2^{\mathbf{X}}\} \cup \{\nu_2^{\mathbf{Y}} \leq_{\mathtt{s}} \nu_2^{\mathbf{X}}\}$$

By the assumption of $\emptyset \vdash convert(c\tau^{\mathbf{U}}, \rho^{\mathbf{X}}) <: c\tau \backslash \Sigma$ and (Sub-Non-Recursive), we know $\{\nu_1^{\mathbf{X}} - \nu_2^{\mathbf{X}} +$

$\nu_1^{\mathbf{U}} - \nu_2^{\mathbf{U}} =_{\mathtt{s}} \nu_1' - \nu_2'\} \cup \{\nu_2^{\mathbf{U}} \leq_{\mathtt{s}} \nu_2'\} \subseteq \Sigma$. The conclusion holds as

$$\left.\begin{array}{l} \nu_1^{\mathbf{Z}} - \nu_2^{\mathbf{Z}} =_{\mathtt{s}} \nu_1^{\mathbf{U}} - \nu_2^{\mathbf{U}} \\ \nu_2^{\mathbf{Z}} \leq_{\mathtt{s}} \nu_2^{\mathbf{U}} \\ \nu_1^{\mathbf{X}} - \nu_2^{\mathbf{X}} + \nu_1^{\mathbf{U}} - \nu_2^{\mathbf{U}} =_{\mathtt{s}} \nu_1' - \nu_2' \\ \nu_2^{\mathbf{U}} \leq_{\mathtt{s}} \nu_2' \\ \nu_2^{\mathbf{X}} \leq_{\mathtt{s}} \nu_1^{\mathbf{U}} \\ \nu_1^{\mathbf{Y}} - \nu_2^{\mathbf{Y}} =_{\mathtt{s}} \nu_1^{\mathbf{X}} - \nu_2^{\mathbf{X}} \\ \nu_2^{\mathbf{Y}} \leq_{\mathtt{s}} \nu_2^{\mathbf{X}} \end{array}\right\} \xrightarrow{\texttt{algebra}} \nu_2^{\mathbf{Y}} + \nu_2^{\mathbf{Z}} - \nu_1^{\mathbf{Z}} \leq_{\mathtt{s}} \nu_2'$$

This is because by (ICS-leq) and $\nu_2^{\mathbf{U}} \leq_{\mathtt{s}} \nu_2'$ and $\nu_2^{\mathbf{X}} \leq_{\mathtt{s}} \nu_1^{\mathbf{U}}$, we have $\nu_2^{\mathbf{U}} + \nu_2^{\mathbf{X}} \leq_{\mathtt{s}} \nu_2' + \nu_1^{\mathbf{U}}$. By $\nu_2^{\mathbf{Y}} \leq_{\mathtt{s}} \nu_2^{\mathbf{X}}$, we thus know $\nu_2^{\mathbf{U}} + \nu_2^{\mathbf{Y}} \leq_{\mathtt{s}} \nu_2' + \nu_1^{\mathbf{U}}$ which by (ICS-leq) implies $\nu_2^{\mathbf{Y}} + \nu_2^{\mathbf{U}} - \nu_1^{\mathbf{U}} \leq_{\mathtt{s}} \nu_2'$. Given the constraint $\nu_1^{\mathbf{Z}} - \nu_2^{\mathbf{Z}} =_{\mathtt{s}} \nu_1^{\mathbf{U}} - \nu_2^{\mathbf{U}}$, we know $\nu_2^{\mathbf{Y}} + \nu_2^{\mathbf{Z}} - \nu_1^{\mathbf{Z}} \leq_{\mathtt{s}} \nu_2'$.

□

Lemma 28 (Subtyping Preservation of Types in Signatures Over $convert$)**.** *If*

$$\begin{array}{r} convert(c\tau^{\mathbf{X}}, \rho) = c\tau^{\mathbf{X}}_{\mathtt{conv}} \\ convert(c\tau^{\mathbf{Y}}, \rho) = c\tau^{\mathbf{Y}}_{\mathtt{conv}} \\ \emptyset \vdash c\tau^{\mathbf{X}} <: c\tau^{\mathbf{Y}} \backslash \Sigma \end{array}$$

then $\emptyset \vdash c\tau^{\mathbf{X}}_{\mathtt{conv}} <: c\tau^{\mathbf{Y}}_{\mathtt{conv}} \backslash \Sigma'$ *and* $\Sigma \equiv \Sigma'$

Proof. Case analysis on the last step of the derivation leading to the judgment of $\emptyset \vdash c\tau^{\mathbf{X}} <: c\tau^{\mathbf{Y}} \backslash \Sigma$.

An instance of (Sub-Bottom): not possible since according to the definition of meta-varialbe $c\tau$, $c\tau^{\mathbf{X}}$ cannot be $\bot$.

An instance of (Sub-Int): then we know $c\tau^{\mathbf{X}} = c\tau^{\mathbf{Y}} =$ **int**, and $c\tau^{\mathbf{X}}_{\mathtt{conv}} = c\tau^{\mathbf{Y}}_{\mathtt{conv}} =$ **int**, and $\Sigma = \Sigma^{\mathbf{X}} = \Sigma^{\mathbf{Y}} = \Sigma' = \emptyset$ and hence the conclusion holds trivially.

An instance of (Sub-Unit): the same as (Sub-Int).

An instance of (Sub-Recursive): not possible since otherwise we know $c\tau^{\mathbf{X}}$ and $c\tau^{\mathbf{Y}}$ are both in the form of $\mathsf{a}_0@\rho_0$. $convert(c\tau^{\mathbf{X}}, \rho)$ and $convert(c\tau^{\mathbf{Y}}, \rho)$ would thus be undefined. Contradiction.

An instance of (Sub-Non-Recursive): first let $c\tau^{\mathbf{X}} = \mu\mathsf{a}^{\mathbf{X}}.\mathcal{K}^{\mathbf{X}}\ @(\textbf{parent})^{\nu_1^{\mathbf{X}}}(\textbf{child})^{\nu_2^{\mathbf{X}}}$, $c\tau^{\mathbf{Y}} = \mu\mathsf{a}^{\mathbf{Y}}.\mathcal{K}^{\mathbf{Y}}\ @(\textbf{parent})^{\nu_1^{\mathbf{Y}}}(\textbf{child})^{\nu_2^{\mathbf{Y}}}$, $\rho = (\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2}$. By (Sub-Non-Recursive), we know

$$\begin{array}{r} \mathrm{Dom}(\mathcal{K}^{\mathbf{Y}}) = \{\mathsf{k}_1, \ldots, \mathsf{k}_n\} \\ \forall i \in \{1, \ldots, n\}.\ \{\mathsf{a}^{\mathbf{X}} <: \mathsf{a}^{\mathbf{Y}}\} \vdash_{\mathtt{c}} \mathcal{K}^{\mathbf{X}}(\mathsf{k}_i) <: \mathcal{K}^{\mathbf{Y}}(\mathsf{k}_i) \backslash \Sigma_i \\ \Sigma = \Sigma_1 \cup \ldots \Sigma_n \cup \{\nu_1^{\mathbf{X}} - \nu_2^{\mathbf{X}} =_{\mathtt{s}} \nu_1^{\mathbf{Y}} - \nu_2^{\mathbf{Y}}, 0 \leq_{\mathtt{s}} \nu_2^{\mathbf{X}} \leq_{\mathtt{s}} \nu_2^{\mathbf{Y}} \leq_{\mathtt{s}} 1\} \end{array}$$

By the definition of $convert$, we know

$$c\tau^{\mathbf{X}}_{\mathtt{conv}} = \mathcal{K}^{\mathbf{X}}\ @(\mathbf{parent})^{\nu_1^{\mathbf{X}} + \nu_1 - \nu_2}(\mathbf{child})^{\nu_2^{\mathbf{X}}}$$
$$c\tau^{\mathbf{Y}}_{\mathtt{conv}} = \mathcal{K}^{\mathbf{Y}}\ @(\mathbf{parent})^{\nu_1^{\mathbf{Y}} + \nu_1 - \nu_2}(\mathbf{child})^{\nu_2^{\mathbf{Y}}}$$

By (Sub-Non-Recursive), we know

$$\Sigma' = \Sigma_1 \cup \ldots \Sigma_n \cup\ \{\nu_1^{\mathbf{X}} + \nu_1 - \nu_2 - \nu_2^{\mathbf{X}} =_{\mathtt{s}} \nu_1^{\mathbf{Y}} + \nu_1 - \nu_2 - \nu_2^{\mathbf{Y}}, 0 \leq_{\mathtt{s}} \nu_2^{\mathbf{X}} \leq_{\mathtt{s}} \nu_2^{\mathbf{Y}} \leq_{\mathtt{s}} 1\}$$

The conclusion holds by (ECS-equal).

□

C.7 Properties of Interface Matching

Lemma 29. *Given* $\emptyset \vdash c\tau^{\mathbf{Z}} <: c\tau^{\mathbf{U}} \backslash \Sigma_{\mathtt{sub}}$ *and* $\Sigma_1 = match(c\tau^{\mathbf{U}}, c\tau, \rho^{\mathbf{X}})$*, then it must hold that* $\Sigma_1 \cup \Sigma_{\mathtt{sub}} \cup subPedWeak(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) \xrightarrow{\mathtt{algebra}} matchd(c\tau^{\mathbf{Z}}, c\tau, \rho^{\mathbf{Y}})$.

Proof. According to the definition of $match$, we know $\emptyset \vdash convert(c\tau^{\mathbf{U}}, \rho^{\mathbf{X}}) <: c\tau \backslash \Sigma$ and $\Sigma_1 = \Sigma \cup wfcons(c\tau^{\mathbf{U}}, \rho^{\mathbf{X}})$. By Lem. 27, we know

$$\Sigma_{\mathtt{sub}} \cup \Sigma_1 \cup subPedWeak(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) \xrightarrow{\mathtt{algebra}} dwfcons(c\tau^{\mathbf{Z}}, c\tau, \rho^{\mathbf{Y}})$$

By $\emptyset \vdash convert(c\tau^{\mathbf{U}}, \rho^{\mathbf{X}}) <: c\tau \backslash \Sigma$ and Lem. 26, we know $\emptyset \vdash convert(c\tau^{\mathbf{U}}, \rho^{\mathbf{Y}}) <: c\tau \backslash \Sigma'$ and $\Sigma \cup subPedWeak(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) \xrightarrow{\mathtt{algebra}} \Sigma'$. Given $\emptyset \vdash c\tau^{\mathbf{Z}} <: c\tau^{\mathbf{U}} \backslash \Sigma_{\mathtt{sub}}$ and the fact that $convert(c\tau^{\mathbf{U}}, \rho^{\mathbf{Y}})$ is defined, by Lem. 25, we know $convert(c\tau^{\mathbf{Z}}, \rho^{\mathbf{Y}})$ is defined. By Lem. 28, we know $\emptyset \vdash convert(c\tau^{\mathbf{Z}}, \rho^{\mathbf{Y}}) <: convert(c\tau^{\mathbf{U}}, \rho^{\mathbf{Y}}) \backslash \Sigma_{\mathtt{sub}'}$ and $\Sigma_{\mathtt{sub}} \equiv \Sigma'_{\mathtt{sub}}$. By Lem. 18 (subtyping transitivity), we know

$$\emptyset \vdash convert(c\tau^{\mathbf{Z}}, \rho^{\mathbf{Y}}) <: c\tau \backslash \Sigma''$$
$$\Sigma' \cup \Sigma_{\mathtt{sub}'} \xrightarrow{\mathtt{algebra}} \Sigma''$$

Obviously

$$\Sigma_{\mathtt{sub}} \cup \Sigma_1 \cup subPedWeak(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) \xrightarrow{\mathtt{algebra}} \Sigma''$$

We know $matchd(c\tau^{\mathbf{Z}}, c\tau, \rho^{\mathbf{Y}}) = \Sigma'' \cup dwfcons(c\tau^{\mathbf{Z}}, c\tau, \rho^{\mathbf{Y}})$. The conclusion thus holds.

□

Lemma 30. *Given* $\emptyset \vdash c\tau^{\mathbf{U}} <: c\tau^{\mathbf{Z}} \backslash \Sigma_{\mathtt{sub}}$ *and* $\Sigma_1 = match(c\tau, c\tau^{\mathbf{U}}, \rho^{\mathbf{X}})$*, then it must hold that* $\Sigma_1 \cup \Sigma_{\mathtt{sub}} \cup subPedWeak(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) \xrightarrow{\mathtt{algebra}} matchd(c\tau, c\tau^{\mathbf{Z}}, \rho^{\mathbf{Y}})$.

Proof. Analogous to the proof for Lem. 29. □

Lemma 31. *If* $\rho^{\mathbf{U}} \vdash c\tau^{\mathbf{Z}'} \to c\tau^{\mathbf{Z}} \overset{\Sigma^{\mathbf{X}}}{\underset{\mathtt{m}}{\Longrightarrow}} c\tau^{\mathbf{X}'} \to c\tau^{\mathbf{X}}$ *and* $\emptyset \vdash_{\mathtt{m}} c\tau^{\mathbf{X}'} \to c\tau^{\mathbf{X}} <: c\tau^{\mathbf{Y}'} \to c\tau^{\mathbf{Y}} \backslash \Sigma$*, then* $\rho^{\mathbf{V}} \vdash c\tau^{\mathbf{Z}'} \to c\tau^{\mathbf{Z}} \overset{\Sigma^{\mathbf{Y}}}{\underset{\mathtt{dm}}{\Longrightarrow}} c\tau^{\mathbf{Y}'} \to c\tau^{\mathbf{Y}}$ *and* $\Sigma^{\mathbf{X}} \cup \Sigma \cup subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) \xrightarrow{\mathtt{algebra}} \Sigma^{\mathbf{Y}}$.

Proof. Let $\rho^{\mathbf{U}} = (\mathbf{parent})^{\nu_a^{\mathbf{U}}}(\mathbf{child})^{\nu_b^{\mathbf{U}}}$ and $\rho^{\mathbf{V}} = (\mathbf{parent})^{\nu_a^{\mathbf{V}}}(\mathbf{child})^{\nu_b^{\mathbf{V}}}$, $\rho^{\mathbf{U}}_{\mathtt{rev}} = (\mathbf{parent})^{\nu_b^{\mathbf{U}}}(\mathbf{child})^{\nu_a^{\mathbf{U}}}$, and $\rho^{\mathbf{V}}_{\mathtt{rev}} = (\mathbf{parent})^{\nu_b^{\mathbf{V}}}(\mathbf{child})^{\nu_a^{\mathbf{V}}}$. By assumption $\rho^{\mathbf{U}} \vdash c\tau^{\mathbf{Z}'} \to c\tau^{\mathbf{Z}} \overset{\Sigma^{\mathbf{X}}}{\underset{\mathtt{m}}{\Longrightarrow}} c\tau^{\mathbf{X}'} \to c\tau^{\mathbf{X}}$ and (Def-MethodMatch), we know

$$match(c\tau^{\mathbf{X}'}, c\tau^{\mathbf{Z}'}, \rho^{\mathbf{U}}) = \Sigma^{\mathbf{X}}_{\mathtt{c1}} \tag{31.1}$$
$$match(c\tau^{\mathbf{Z}}, c\tau^{\mathbf{X}}, \rho^{\mathbf{U}}_{\mathtt{rev}}) = \Sigma^{\mathbf{X}}_{\mathtt{c2}} \tag{31.2}$$
$$\Sigma^{\mathbf{X}} = \Sigma^{\mathbf{X}}_{\mathtt{c1}} \cup \Sigma^{\mathbf{X}}_{\mathtt{c2}} \tag{31.3}$$

By assumption $\emptyset \vdash_{\mathtt{m}} c\tau^{\mathbf{X}'} \to c\tau^{\mathbf{X}} <: c\tau^{\mathbf{Y}'} \to c\tau^{\mathbf{Y}} \backslash \Sigma$ and (Sub-Method),

$$\emptyset \vdash c\tau^{\mathbf{X}} <: c\tau^{\mathbf{Y}} \backslash \Sigma_1 \tag{31.4}$$
$$\emptyset \vdash c\tau^{\mathbf{Y}'} <: c\tau^{\mathbf{X}'} \backslash \Sigma_2 \tag{31.5}$$
$$\Sigma = \Sigma_1 \cup \Sigma_2 \tag{31.6}$$

With (**31.1**), (**31.5**), and Lem. 29, we have

$$\Sigma^{\mathbf{X}}_{\mathtt{c1}} \cup \Sigma_2 \cup subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) \xrightarrow{\mathtt{algebra}} matchd(c\tau^{\mathbf{Y}'}, c\tau^{\mathbf{Z}'}, \rho^{\mathbf{V}}) \tag{31.7}$$

With (**31.2**), (**31.4**), and Lem. 30, we have

$$\Sigma^{\mathbf{X}}_{\mathtt{c2}} \cup \Sigma_1 \cup subPedWeak(\rho^{\mathbf{V}}_{\mathtt{rev}}, \rho^{\mathbf{U}}_{\mathtt{rev}}) \xrightarrow{\mathtt{algebra}} matchd(c\tau^{\mathbf{Z}}, c\tau^{\mathbf{Y}}, \rho^{\mathbf{V}}_{\mathtt{rev}}) \tag{31.8}$$

Thus both $matchd(c\tau^{\mathbf{Y}'}, c\tau^{\mathbf{Z}'}, \rho^{\mathbf{V}})$ and $matchd(c\tau^{\mathbf{Z}}, c\tau^{\mathbf{Y}}, \rho^{\mathbf{V}}_{\mathtt{rev}})$ are defined. By (Def-MethodMatchD), we have $\rho^{\mathbf{V}} \vdash c\tau^{\mathbf{Z}'} \to c\tau^{\mathbf{Z}} \overset{\Sigma^{\mathbf{Y}}}{\underset{\mathtt{dm}}{\Longrightarrow}} c\tau^{\mathbf{Y}'} \to c\tau^{\mathbf{Y}}$, $\Sigma^{\mathbf{Y}} = matchd(c\tau^{\mathbf{Y}'}, c\tau^{\mathbf{Z}'}, \rho^{\mathbf{V}}) \cup matchd(c\tau^{\mathbf{Z}}, c\tau^{\mathbf{Y}}, \rho^{\mathbf{V}}_{\mathtt{rev}})$.

By the definition of $subPedWeak$

$$subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) = \{\nu_1^{\mathbf{V}} - \nu_2^{\mathbf{V}} =_{\mathtt{s}} \nu_1^{\mathbf{U}} - \nu_2^{\mathbf{U}}, \nu_2^{\mathbf{V}} \leq_{\mathtt{s}} \nu_2^{\mathbf{U}}\}$$
$$subPedWeak(\rho^{\mathbf{V}}_{\mathtt{rev}}, \rho^{\mathbf{U}}_{\mathtt{rev}}) = \{\nu_2^{\mathbf{V}} - \nu_1^{\mathbf{V}} =_{\mathtt{s}} \nu_2^{\mathbf{U}} - \nu_1^{\mathbf{U}}, \nu_1^{\mathbf{V}} \leq_{\mathtt{s}} \nu_1^{\mathbf{U}}\}$$

Thus by (ICS-leq) and the definition of $\equiv$, we know

$$subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) \equiv subPedWeak(\rho^{\mathbf{V}}_{\mathtt{rev}}, \rho^{\mathbf{U}}_{\mathtt{rev}}) \tag{31.9}$$

Obviously $\Sigma^{\mathbf{X}} \cup \Sigma \cup subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}}$ following (**31.3**), (**31.6**), (**31.7**), (**31.8**), (**31.9**).

□

Lemma 32. *If* $\rho^{\mathbf{U}} \vdash c\tau^{\mathbf{X}'} \to c\tau^{\mathbf{X}} \overset{\Sigma^{\mathbf{X}}}{\underset{\mathtt{m}}{\Longrightarrow}} c\tau^{\mathbf{Z}'} \to c\tau^{\mathbf{Z}}$ *and* $\emptyset \vdash_{\mathtt{m}} c\tau^{\mathbf{Y}'} \to c\tau^{\mathbf{Y}} <: c\tau^{\mathbf{X}'} \to c\tau^{\mathbf{X}} \backslash \Sigma$*, then* $\rho^{\mathbf{V}} \vdash c\tau^{\mathbf{Y}'} \to c\tau^{\mathbf{Y}} \overset{\Sigma^{\mathbf{Y}}}{\underset{\mathtt{dm}}{\Longrightarrow}} c\tau^{\mathbf{Z}'} \to c\tau^{\mathbf{Z}}$ *and* $\Sigma^{\mathbf{X}} \cup \Sigma \cup subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}}$.

Proof. Let $\rho^{\mathbf{U}} = (\mathbf{parent})^{\nu^{\mathbf{U}}_a}(\mathbf{child})^{\nu^{\mathbf{U}}_b}$ and $\rho^{\mathbf{V}} = (\mathbf{parent})^{\nu^{\mathbf{V}}_a}(\mathbf{child})^{\nu^{\mathbf{V}}_b}$, $\rho^{\mathbf{U}}_{\mathtt{rev}} = (\mathbf{parent})^{\nu^{\mathbf{U}}_b}(\mathbf{child})^{\nu^{\mathbf{U}}_a}$, and $\rho^{\mathbf{V}}_{\mathtt{rev}} = (\mathbf{parent})^{\nu^{\mathbf{V}}_b}(\mathbf{child})^{\nu^{\mathbf{V}}_a}$. By assumption $\rho^{\mathbf{U}} \vdash c\tau^{\mathbf{X}'} \to c\tau^{\mathbf{X}} \overset{\Sigma^{\mathbf{X}}}{\underset{\mathtt{m}}{\Longrightarrow}} c\tau^{\mathbf{Z}'} \to c\tau^{\mathbf{Z}}$ and (Def-MethodMatch), we know

$$match(c\tau^{\mathbf{Z}'}, c\tau^{\mathbf{X}'}, \rho^{\mathbf{U}}) = \Sigma^{\mathbf{X}}_{\mathtt{c1}} \tag{32.1}$$
$$match(c\tau^{\mathbf{X}}, c\tau^{\mathbf{Z}}, \rho^{\mathbf{U}}_{\mathtt{rev}}) = \Sigma^{\mathbf{X}}_{\mathtt{c2}} \tag{32.2}$$
$$\Sigma^{\mathbf{X}} = \Sigma^{\mathbf{X}}_{\mathtt{c1}} \cup \Sigma^{\mathbf{X}}_{\mathtt{c2}} \tag{32.3}$$

By assumption $\emptyset \vdash_{\mathtt{m}} c\tau^{\mathbf{Y}'} \to c\tau^{\mathbf{Y}} <: c\tau^{\mathbf{X}'} \to c\tau^{\mathbf{X}} \backslash \Sigma$ and (Sub-Method),

$$\emptyset \vdash c\tau^{\mathbf{Y}} <: c\tau^{\mathbf{X}} \backslash \Sigma_1 \tag{32.4}$$
$$\emptyset \vdash c\tau^{\mathbf{X}'} <: c\tau^{\mathbf{Y}'} \backslash \Sigma_2 \tag{32.5}$$
$$\Sigma = \Sigma_1 \cup \Sigma_2 \tag{32.6}$$

With (**32.1**), (**32.5**), and Lem. 30, we have

$$\Sigma^{\mathbf{X}}_{\mathtt{c1}} \cup \Sigma_2 \cup subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) \xrightarrow{\texttt{algebra}} matchd(c\tau^{\mathbf{Z}'}, c\tau^{\mathbf{Y}'}, \rho^{\mathbf{V}}) \tag{32.7}$$

With (**32.2**), (**32.4**), and Lem. 29, we have

$$\Sigma^{\mathbf{X}}_{\mathtt{c2}} \cup \Sigma_1 \cup subPedWeak(\rho^{\mathbf{V}}_{\mathtt{rev}}, \rho^{\mathbf{U}}_{\mathtt{rev}}) \xrightarrow{\texttt{algebra}} matchd(c\tau^{\mathbf{Y}}, c\tau^{\mathbf{Z}}, \rho^{\mathbf{V}}_{\mathtt{rev}}) \tag{32.8}$$

Thus both $matchd(c\tau^{\mathbf{Z}'}, c\tau^{\mathbf{Y}'}, \rho^{\mathbf{V}})$ and $matchd(c\tau^{\mathbf{Y}}, c\tau^{\mathbf{Z}}, \rho^{\mathbf{V}}_{\mathtt{rev}})$ are defined. By (Def-MethodMatchD), we have $\rho^{\mathbf{V}} \vdash c\tau^{\mathbf{Y}'} \to c\tau^{\mathbf{Y}} \overset{\Sigma^{\mathbf{Y}}}{\underset{\mathtt{dm}}{\Longrightarrow}} c\tau^{\mathbf{Z}'} \to c\tau^{\mathbf{Z}}$, $\Sigma^{\mathbf{Y}} = matchd(c\tau^{\mathbf{Z}'}, c\tau^{\mathbf{Y}'}, \rho^{\mathbf{V}}) \cup matchd(c\tau^{\mathbf{Y}}, c\tau^{\mathbf{Z}}, \rho^{\mathbf{V}}_{\mathtt{rev}})$. By the definition of $subPedWeak$

$$subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) = \{\nu^{\mathbf{V}}_1 - \nu^{\mathbf{V}}_2 =_{\mathtt{s}} \nu^{\mathbf{U}}_1 - \nu^{\mathbf{U}}_2, \nu^{\mathbf{V}}_2 \leq_{\mathtt{s}} \nu^{\mathbf{U}}_2\}$$
$$subPedWeak(\rho^{\mathbf{V}}_{\mathtt{rev}}, \rho^{\mathbf{U}}_{\mathtt{rev}}) = \{\nu^{\mathbf{V}}_2 - \nu^{\mathbf{V}}_1 =_{\mathtt{s}} \nu^{\mathbf{U}}_2 - \nu^{\mathbf{U}}_1, \nu^{\mathbf{V}}_1 \leq_{\mathtt{s}} \nu^{\mathbf{U}}_1\}$$

Thus by (ICS-leq) and the definition of $\equiv$, we know

$$subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) \equiv subPedWeak(\rho^{\mathbf{V}}_{\mathtt{rev}}, \rho^{\mathbf{U}}_{\mathtt{rev}}) \tag{32.9}$$

Obviously $\Sigma^{\mathbf{X}} \cup \Sigma \cup subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) \xrightarrow{\mathtt{algebra}} \Sigma^{\mathbf{Y}}$ following **(32.3)**, **(32.6)**, **(32.7)**, **(32.8)**, **(32.9)**.

□

Lemma 33 (Interface Matching Holds when the Connector on One Side is Refined)**.** *If*

$$\emptyset \vdash_{\mathtt{c}} \langle \mathcal{I}^{\mathbf{Y}}; \mathcal{E}^{\mathbf{Y}} \rangle <: \langle \mathcal{I}^{\mathbf{X}}; \mathcal{E}^{\mathbf{X}} \rangle \backslash \Sigma \tag{33.1}$$

$$\rho^{\mathbf{U}} \vdash \langle \mathcal{I}^{\mathbf{Z}}; \mathcal{E}^{\mathbf{Z}} \rangle \overset{\Sigma^{\mathbf{X}}}{\Longrightarrow} \langle \mathcal{I}^{\mathbf{X}}; \mathcal{E}^{\mathbf{X}} \rangle \tag{33.2}$$

then $\rho^{\mathbf{V}} \vdash \langle \mathcal{I}^{\mathbf{Z}}; \mathcal{E}^{\mathbf{Z}} \rangle \underset{\mathtt{dc}}{\overset{\Sigma^{\mathbf{Y}}}{\Longrightarrow}} \langle \mathcal{I}^{\mathbf{Y}}; \mathcal{E}^{\mathbf{Y}} \rangle$ *and* $\Sigma^{\mathbf{X}} \cup \Sigma \cup subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) \xrightarrow{\mathtt{algebra}} \Sigma^{\mathbf{Y}}$.

Proof. By **(33.1)** and (Sub-Connector), we know

$$\mathrm{Dom}(\mathcal{E}^{\mathbf{X}}) = \{\mathsf{m}_1, \ldots, \mathsf{m}_p\} \tag{33.3}$$

$$\mathrm{Dom}(\mathcal{I}^{\mathbf{Y}}) = \{\mathsf{m}_{p+1}, \ldots, \mathsf{m}_q\} \tag{33.4}$$

$$\forall i \in \{1, \ldots, p\}.\, \emptyset \vdash_{\mathtt{m}} \mathcal{E}^{\mathbf{Y}}(\mathsf{m}_i) <: \mathcal{E}^{\mathbf{X}}(\mathsf{m}_i) \backslash \Sigma_i \tag{33.5}$$

$$\forall j \in \{p+1, \ldots, q\}.\, \emptyset \vdash_{\mathtt{m}} \mathcal{I}^{\mathbf{X}}(\mathsf{m}_i) <: \mathcal{I}^{\mathbf{Y}}(\mathsf{m}_i) \backslash \Sigma_j \tag{33.6}$$

$$\Sigma = \Sigma_1 \cdots \cup \Sigma_q \tag{33.7}$$

Implicitly from **(33.4)** and **(33.6)**, we know $\mathrm{Dom}(\mathcal{I}^{\mathbf{Y}}) \subseteq \mathrm{Dom}(\mathcal{I}^{\mathbf{X}})$. Thus we know there exists some $s \geq q$ such that

$$\mathrm{Dom}(\mathcal{I}^{\mathbf{X}}) = \{\mathsf{m}_{p+1}, \ldots, \mathsf{m}_q, \ldots, \mathsf{m}_s\} \tag{33.8}$$

By **(33.2)** and (Def-ConnectorMatchD), we know implicitly $\mathrm{Dom}(\mathcal{I}^{\mathbf{Z}}) \subseteq \mathrm{Dom}(\mathcal{E}^{\mathbf{X}})$. Thus we know there exists some $r \leq p$ such that

$$\mathrm{Dom}(\mathcal{I}^{\mathbf{Z}}) = \{\mathsf{m}_1, \ldots, \mathsf{m}_r\} \tag{33.9}$$

First let $\rho^{\mathbf{U}} = (\textbf{parent})^{\nu_a^{\mathbf{U}}}(\textbf{child})^{\nu_b^{\mathbf{U}}}$ and $\rho^{\mathbf{V}} = (\textbf{parent})^{\nu_a^{\mathbf{V}}}(\textbf{child})^{\nu_b^{\mathbf{V}}}$. Subsequently we also let $\rho^{\mathbf{U}}_{\mathtt{rev}} = (\textbf{parent})^{\nu_b^{\mathbf{U}}}(\textbf{child})^{\nu_a^{\mathbf{U}}}$ and $\rho^{\mathbf{V}}_{\mathtt{rev}} = (\textbf{parent})^{\nu_b^{\mathbf{V}}}(\textbf{child})^{\nu_a^{\mathbf{V}}}$. By **(33.2)** and (Def-ConnectorMatchD), we know

$$\forall i \in \{1, \ldots, r\}. \rho^{\mathbf{U}}_{\mathtt{rev}} \vdash \mathcal{E}^{\mathbf{X}}(\mathsf{m}_i) \underset{\mathtt{m}}{\overset{\Sigma^{\mathbf{X}}_i}{\Longrightarrow}} \mathcal{I}^{\mathbf{Z}}(\mathsf{m}_i) \tag{33.10}$$

$$\forall j \in \{p+1, \ldots, s\}. \rho^{\mathbf{U}} \vdash \mathcal{E}^{\mathbf{Z}}(\mathsf{m}_j) \underset{\mathtt{m}}{\overset{\Sigma^{\mathbf{X}}_j}{\Longrightarrow}} \mathcal{I}^{\mathbf{X}}(\mathsf{m}_j) \tag{33.11}$$

$$\Sigma^{\mathbf{X}} = \Sigma^{\mathbf{X}}_1 \cdots \cup \Sigma^{\mathbf{X}}_r \cup \Sigma^{\mathbf{X}}_{p+1} \cdots \cup \Sigma^{\mathbf{X}}_s$$

By Lem. 31, **(33.4)**, **(33.6)**, **(33.11)** and the fact that $q \leq s$, we know

$$\forall j \in \{p+1, \ldots, q\}.\rho^{\mathbf{V}} \vdash \mathcal{E}^{\mathbf{Z}}(\mathsf{m}_j) \overset{\Sigma_j^{\mathbf{Y}}}{\underset{\mathtt{dm}}{\Longrightarrow}} \mathcal{I}^{\mathbf{Y}}(\mathsf{m}_j)$$
$$\forall j \in \{p+1, \ldots, q\}.\Sigma_j^{\mathbf{X}} \cup \Sigma_j \cup subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) \xrightarrow{\mathtt{algebra}} \Sigma_j^{\mathbf{Y}}$$

By Lem. 32, **(33.9)**, **(33.5)**, **(33.10)** and the fact that $r \leq p$, we know

$$\forall i \in \{1, \ldots, r\}.\rho_{\mathtt{rev}}^{\mathbf{V}} \vdash \mathcal{E}^{\mathbf{Y}}(\mathsf{m}_i) \overset{\Sigma_i^{\mathbf{Y}}}{\underset{\mathtt{dm}}{\Longrightarrow}} \mathcal{I}^{\mathbf{Z}}(\mathsf{m}_i)$$
$$\forall i \in \{1, \ldots, r\}.\Sigma_i^{\mathbf{X}} \cup \Sigma_i \cup subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) \xrightarrow{\mathtt{algebra}} \Sigma_i^{\mathbf{Y}}$$

By (Def-ConnectorMatchD), we know $\rho^{\mathbf{V}} \vdash \langle \mathcal{I}^{\mathbf{Z}}; \mathcal{E}^{\mathbf{Z}} \rangle \overset{\Sigma^{\mathbf{Y}}}{\underset{\mathtt{dc}}{\Longrightarrow}} \langle \mathcal{I}^{\mathbf{Y}}; \mathcal{E}^{\mathbf{Y}} \rangle$ holds where $\Sigma^{\mathbf{Y}} = \Sigma_1^{\mathbf{Y}} \cdots \cup \Sigma_r^{\mathbf{Y}} \cup \Sigma_{p+1}^{\mathbf{Y}} \cup \Sigma_q^{\mathbf{Y}}$. By (ICS-union), $\Sigma^{\mathbf{X}} \cup \Sigma \cup subPedWeak(\rho^{\mathbf{V}}, \rho^{\mathbf{U}}) \xrightarrow{\mathtt{algebra}} \Sigma^{\mathbf{Y}}$.

□

Lemma 34 (Interface Matching Preservation In the Presence of Subtyping)**.** *If*

$$\emptyset \vdash \mu\mathsf{a}^{\mathbf{Y}}.\mathcal{K}^{\mathbf{Y}} @\rho^{\mathbf{Y}} <: \mu\mathsf{a}^{\mathbf{X}}.\mathcal{K}^{\mathbf{X}} @\rho^{\mathbf{X}} \backslash \Sigma_{\mathtt{sub}} \tag{34.1}$$
$$\rho^{\mathbf{X}} \vdash \mathcal{K}^{\mathbf{Z}}(\mathsf{k}) \overset{\Sigma^{\mathbf{X}}}{\Longrightarrow} \mathcal{K}^{\mathbf{X}}(\mathsf{k}') \tag{34.2}$$

then $\rho^{\mathbf{Y}} \vdash \mathcal{K}^{\mathbf{Z}}(\mathsf{k}) \overset{\Sigma^{\mathbf{Y}}}{\underset{\mathtt{dc}}{\Longrightarrow}} \mathcal{K}^{\mathbf{Y}}(\mathsf{k}')$ *and* $\Sigma^{\mathbf{X}} \cup \Sigma_{\mathtt{sub}} \xrightarrow{\mathtt{algebra}} \Sigma^{\mathbf{Y}}$.

Proof. By (**34.1**) and (Sub-Non-Recursive), we know

$$\mathrm{Dom}(\mathcal{K}^{\mathbf{X}}) = \{\mathsf{k}_1, \ldots, \mathsf{k}_n\} \tag{34.3}$$
$$\mathsf{a}_{\mathtt{new}}^{\mathbf{Y}}, \mathsf{a}_{\mathtt{new}}^{\mathbf{X}} \text{ fresh}$$
$$\mathcal{K}_{\mathtt{new}}^{\mathbf{Y}} = \mathcal{K}^{\mathbf{Y}}\{\mathsf{a}_{\mathtt{new}}^{\mathbf{Y}}/\mathsf{a}^{\mathbf{Y}}\}$$
$$\mathcal{K}_{\mathtt{new}}^{\mathbf{X}} = \mathcal{K}^{\mathbf{X}}\{\mathsf{a}_{\mathtt{new}}^{\mathbf{X}}/\mathsf{a}^{\mathbf{X}}\}$$
$$\forall i \in \{1, \ldots, n\}.\ \{\mathsf{a}_{\mathtt{new}}^{\mathbf{Y}} <: \mathsf{a}_{\mathtt{new}}^{\mathbf{X}}\} \vdash_{\mathsf{c}} \mathcal{K}_{\mathtt{new}}^{\mathbf{Y}}(\mathsf{k}_i) <: \mathcal{K}_{\mathtt{new}}^{\mathbf{X}}(\mathsf{k}_i) \backslash \Sigma_i \tag{34.4}$$
$$\Sigma_{\mathtt{sub}} = \Sigma_1 \cup \ldots \Sigma_n \cup subPed(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) \tag{34.5}$$

By (**34.2**), we know $\mathsf{k}' \in \mathrm{Dom}(\mathcal{K}^{\mathbf{X}})$. Thus by (**34.3**), we know $\mathsf{k}' = \mathsf{k}_u$ for some $u \in \{1, \ldots, n\}$. Thus, we know $\mathsf{k}' \in \mathrm{Dom}(\mathcal{K}^{\mathbf{Y}})$, $\mathsf{k}' \in \mathrm{Dom}(\mathcal{K}_{\mathtt{new}}^{\mathbf{Y}})$, $\mathsf{k}' \in \mathrm{Dom}(\mathcal{K}_{\mathtt{new}}^{\mathbf{X}})$ because of (**34.4**) and the fact that substitution of classage names do not change the domain of connector signature lists. Let $\mathcal{K}_{\mathtt{new}}^{\mathbf{Y}}(\mathsf{k}') = \langle \mathcal{I}^{\mathbf{Y}}; \mathcal{E}^{\mathbf{Y}} \rangle$ and $\mathcal{K}_{\mathtt{new}}^{\mathbf{X}}(\mathsf{k}') = \langle \mathcal{I}^{\mathbf{X}}; \mathcal{E}^{\mathbf{X}} \rangle$. By (**34.4**), we know $\emptyset \vdash_{\mathsf{c}} \langle \mathcal{I}^{\mathbf{Y}}; \mathcal{E}^{\mathbf{Y}} \rangle <: \langle \mathcal{I}^{\mathbf{X}}; \mathcal{E}^{\mathbf{X}} \rangle \backslash \Sigma_u$. Together with (**34.2**) and Lem. 33, we know

$$\rho^{\mathbf{Y}} \vdash \mathcal{K}^{\mathbf{Z}}(\mathsf{k}) \overset{\Sigma^{\mathbf{Y}}}{\underset{\mathtt{dc}}{\Longrightarrow}} \mathcal{K}^{\mathbf{Y}}(\mathsf{k}')$$

$$\Sigma^{\mathbf{X}} \cup \Sigma_u \cup subPedWeak(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}}$$

By Lem. 24, we know $subPed(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) \xrightarrow{\texttt{algebra}} subPedWeak(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}})$. Hence we know $\Sigma^{\mathbf{X}} \cup \Sigma_u \cup subPed(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}}$ by (ICS-transitivity), (ICS-union). The rest of the proof is to show $\Sigma^{\mathbf{X}} \cup \Sigma_{\texttt{sub}} \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}}$. By adding $\Sigma_1 \cup \ldots \Sigma_n$ on both sides, and (ICS-union), we know

$$\Sigma^{\mathbf{X}} \cup \Sigma_1 \cup \ldots \Sigma_n \cup subPed(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}} \cup \Sigma_1 \cup \ldots \Sigma_n \cup subPed(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}})$$

By (ICS-relaxation), we know

$$\Sigma^{\mathbf{X}} \cup \Sigma_1 \cup \ldots \Sigma_n \cup subPed(\rho^{\mathbf{Y}}, \rho^{\mathbf{X}}) \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}}$$

which exactly is $\Sigma^{\mathbf{X}} \cup \Sigma_{\texttt{sub}} \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}}$, the conclusion. □

Lemma 35 (Interface Matching Preservation In the Presence of Pedigree Refinement)**.** *If* $\rho^{\mathbf{X}} \vdash \mathcal{K}(\mathsf{k}) \overset{\Sigma^{\mathbf{X}}}{\underset{\texttt{dc}}{\Longrightarrow}} \mathcal{K}'(\mathsf{k}')$*, then* $\rho^{\mathbf{Y}} \vdash \mathcal{K}(\mathsf{k}) \overset{\Sigma^{\mathbf{Y}}}{\underset{\texttt{dc}}{\Longrightarrow}} \mathcal{K}(\mathsf{k}')$ *and* $\Sigma^{\mathbf{X}} \cup \Sigma_{\texttt{sub}} \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}}$.

Proof. The proof for this lemma is analogous to Lem. 34, only it is simpler since structural subtyping does not need to be considered. □

Lemma 36 (Consistent Matching in Presence of Substitution)**.** *Given* $\rho \vdash \langle \mathcal{I}_1; \mathcal{E}_1 \rangle \overset{\Sigma}{\underset{\texttt{dc}}{\Longrightarrow}} \langle \mathcal{I}_2; \mathcal{E}_2 \rangle$ *and some* σ *such that* $\vdash_{\texttt{wftSubst}} \sigma$ *and* $TV(\mathrm{Ran}(\sigma)) \cap TV(\Sigma) = \emptyset$*,* $TV(\mathrm{Dom}(\sigma)) \cap TV(\langle \mathcal{I}_2; \mathcal{E}_2 \rangle) = \emptyset$*, then* $\rho \vdash \langle \mathcal{I}_1; \mathcal{E}_1 \rangle[\sigma] \overset{\Sigma'}{\underset{\texttt{dc}}{\Longrightarrow}} \langle \mathcal{I}_2; \mathcal{E}_2 \rangle$ *holds and* $\Sigma' = \Sigma[\sigma]$.

Proof. Straightforward following the definition of (Def-ConnectorMatchD). Note that $\langle \mathcal{I}_1; \mathcal{E}_1 \rangle[\sigma]$ results in a connector signature where all occurences of pedigree type variables in $\mathcal{I}_1$ and $\mathcal{E}_1$ are substituted as long as they are in $TV(\mathrm{Dom}(\sigma))$. All the constraints collected by (Def-ConnectorMatchD) are related to pedigree type variables, which are either from $\langle \mathcal{I}_1; \mathcal{E}_1 \rangle$ or from $\langle \mathcal{I}_2; \mathcal{E}_2 \rangle$. There are only four possibilities for these type variables:

- those in $TV(\langle \mathcal{I}_1; \mathcal{E}_1 \rangle)$ and in $TV(\mathrm{Dom}(\sigma))$.
- those in $TV(\langle \mathcal{I}_1; \mathcal{E}_1 \rangle)$ but not in $TV(\mathrm{Dom}(\sigma))$.
- those in $TV(\langle \mathcal{I}_2; \mathcal{E}_2 \rangle)$ but not in $TV(\mathrm{Dom}(\sigma))$.

- those in $TV(\langle \mathcal{I}_2; \mathcal{E}_2 \rangle)$ and in $TV(\mathrm{Dom}(\sigma))$.

Note that one condition of the lemma is $TV(\mathrm{Dom}(\sigma)) \cap TV(\langle \mathcal{I}_2; \mathcal{E}_2 \rangle) = \emptyset$. Therefore the last case is not possible. The constraints collected with the presence of substitution (*i.e.* Σ') will be on these pedigree type variables with possible substitution following σ. If the type variables are of the second or the third case, they will not be substituted for. The occurrences of them in Σ' will be identical to those corresponding occurrences in Σ. In the first case, they will, and the substitution is merely a renaming of variables. All in all, $\Sigma' = \Sigma[\sigma]$.

□

C.8 Properties of the Instantiation Tree and the rel Function

Lemma 37 (Unique Path in Instantiation Tree)**.** *If R is an instantiation tree and an $o - o'$ path exists in $graphRep(R)$, then the path is unique.*

Proof. Prove by contradiction. Suppose two distinct $o - o'$ paths exist and let them be $[o_0^{\mathbf{X}}, o_1^{\mathbf{X}}, \ldots, o_p^{\mathbf{X}}]$ for some $p \geq 0$ and $[o_0^{\mathbf{Y}}, o_1^{\mathbf{Y}}, \ldots, o_q^{\mathbf{Y}}]$ for some $q \geq 0$. Not to lose generality, let us assume $p \leq q$. By the definition of Def. 5, we know $o_0^{\mathbf{X}} = o_0^{\mathbf{Y}} = o$ and $o_p^{\mathbf{X}} = o_q^{\mathbf{Y}} = o'$. We now perform case analysis.

If for any $0 \leq i \leq p$, we have $o_i^{\mathbf{X}} = o_i^{\mathbf{Y}}$, then we know $o_p^{\mathbf{X}} = o_p^{\mathbf{Y}}$. According to the definition of Def. 5, we know $o_0^{\mathbf{Y}}$, $o_1^{\mathbf{Y}}$, …, $o_q^{\mathbf{Y}}$ are distinct. Earlier we have already known $o_p^{\mathbf{X}} = o_q^{\mathbf{Y}}$. As a result, the only possible to have $o_p^{\mathbf{X}} = o_p^{\mathbf{Y}}$ is to have $p = q$, but this would make the two paths being identical. Contradiction.

Otherwise, if there exists some r such that $0 \leq r < p$, and for any $0 \leq i \leq r$, we have $o_i^{\mathbf{X}} = o_i^{\mathbf{Y}}$, but $o_{r+1}^{\mathbf{X}} \neq o_{r+1}^{\mathbf{Y}}$. Let j and k to be the lowest numbers such that $r + 1 \leq j \leq p$, $r + 1 \leq k \leq q$ and $o_j^{\mathbf{X}} = o_k^{\mathbf{Y}}$. Such j and k must exist since $j = p$ and $k = q$ satisfy all the conditions (but might not be the lowest numbers). Let $graphRep(R) = \langle VER; EDGE; lab \rangle$. According to the definition of R being an instanatiation tree, we know the underlying undirected unlabeled graph induced by $\langle VER; EDGE \rangle$ is an acyclic graph. However, if we traverse the graph following the order of $o_r^{\mathbf{X}}$, $o_{r+1}^{\mathbf{X}}$, …, $o_j^{\mathbf{X}}$, $o_{k-1}^{\mathbf{Y}}$, $o_{k-2}^{\mathbf{Y}}$, …, $o_r^{\mathbf{Y}}$, a cycle is formed. This is indeed a cycle because according to Def. 5, we know $o_r^{\mathbf{X}}$, $o_{r+1}^{\mathbf{X}}$, …, $o_j^{\mathbf{X}}$ are distinct; $o_{k-1}^{\mathbf{Y}}$, $o_{k-2}^{\mathbf{Y}}$, …, $o_{r+1}^{\mathbf{Y}}$ are distinct; the two sequences are also pairwise distinct as a result of how j and k are selected; $o_j^{\mathbf{X}} = o_k^{\mathbf{Y}}$; and $o_r^{\mathbf{X}} = o_r^{\mathbf{Y}}$.

□

Corollary 6 (Unique $o - o'$ Up-Path in Instantiation Tree)**.** *If R is an instantiation tree and an $o - o'$ up-path exists in $graphRep(R)$, then the up-path is unique.*

Proof. If a $o - o'$ up-path exists in $graphRep(R)$, it must be in the form of $[o_0, o_1, \ldots, o_p]$ for some $p \geq 0$ according to the definition of up-path (Def. 2). First, we know o_0, o_1, ..., o_p are distinct. Second, we know there exists o_p, such that $[o_0, o_1, \ldots, o_p]$ is a $o_0 - o_p$ up-path (by assumption), and $[o_p]$ is a $o_p - o_p$ up-path (by the trivial use of definition of up-path). Thus, $[o_0, o_1, \ldots, o_p]$ is a path according to Def. 5. By Lem. 37, we know such a path is unique.

□

Lemma 38 (Ubiquitous Path in Instantiation Tree)**.** *If R is an instantiation tree and $graphRep(R) = \langle VER; EDGE; lab \rangle$, then given any $o \in VER$ and any $o' \in VER$, the $o - o'$ path is defined in R.*

Proof. According to the definition of instantiation tree, we know there is a root $o_{\texttt{root}} \in VER$ and there is an $o - o_{\texttt{root}}$ up-path and an $o' - o_{\texttt{root}}$ up-path. Let the $o - o_{\texttt{root}}$ up-path be $[o_0, o_1, \ldots, o_s]$ and obviously according to the definition of up-path, we know $o = o_0$ and $o_{\texttt{root}} = o_s$. Let q be the smallest number such that $0 \leq q \leq s$ and an $o' - o_q$ up-path exists. q must exist since q can always be s if no smaller one exists. Now let the $o' - o_q$ up-path be $[o'_0, o'_1, \ldots, o'_r]$. Obviously $[o_0, o_1, \ldots, o_q]$ is an $o - o_q$ up-path according to the definition of up-path. Let $[o_0, o_1, \ldots, o_q, o'_{r-1}, o'_{r-2}, \ldots, o'_0]$ be the resulting path. It is easy to see all elements on this path are indeed distinct: 1) $o_0, o_1, \ldots, o_q$ are distinct by the definition of $o - o_{\texttt{root}}$ up-path; 2) $o'_{r-1}, o'_{r-2}, \ldots, o'_0$ are distinct by the definition of $o' - o_q$ up-path; 3) If there exists some $0 \leq qq < q$ and some $0 \leq rr \leq r - 1$ where $o_{qq} = o'_{rr}$, then obviously both $[o_0, o_1, \ldots, o_{qq}]$ and $[o'_0, o'_1, \ldots, o'_{rr}]$ are also up-paths, and qq thus is the smallest number satisfying the previous conditions rather than q. Contradiction. Thus, the path indeed exists, and it is $[o_0, o_1, \ldots, o_q, o'_{r-1}, o'_{r-2}, \ldots, o'_0]$.

□

Lemma 39 (Path Reversal)**.** *If $[o_0^{\mathbf{X}}, o_1^{\mathbf{X}}, \ldots o_p^{\mathbf{X}}]$ is an $o_0^{\mathbf{X}} - o_p^{\mathbf{X}}$ path in R, then the $o_p^{\mathbf{X}} - o_0^{\mathbf{X}}$ path must be $[o_p^{\mathbf{X}}, o_{p-1}^{\mathbf{X}}, \ldots o_0^{\mathbf{X}}]$.*

Proof. By definition of path. □

Lemma 40 (Subpath)**.** *Given an $o_0 - o_p$ path in R being $[o_0, o_1, \ldots o_p]$ for some $p \geq 0$, then any subsequence $[o_r, o_{r+1}, \ldots o_s]$ for $r \geq 0$ and $s \leq p$ is an $o_r - o_s$ path.*

Proof. By definition of path. □

Lemma 41. *Given any R, if the $o - o'$ up-path is $[o_0, \ldots, o_p]$ for some $p \geq 0$, then for any o'', if the $o - o''$ up-path exists, it is either $[o_0, \ldots, o_q]$ for some $q \leq p$ or $[o_0, \ldots, o_p, o_{p+1}, \ldots, o_q]$ for some $q > p$.*

Proof. According to the definition of instantiation tree, we know there exists a root $o_{\texttt{root}}$ in R, and there is an $o' - o_{\texttt{root}}$ up-path. By Cor. 6, we know up-paths are unique if they exist. Not to lose generality, let us suppose it is $[o_p, o_{p+1}, \ldots, o_r]$, where $r \geq p$, $o_r = o_{\texttt{root}}$. By the definition of up-path, the concatenation of the $o - o'$ up-path and the $o' - o_{\texttt{root}}$ up-path (with o' only showing up once) must also be an $o - o_{\texttt{root}}$ up-path. Similarly, we know there is an $o'' - o_{\texttt{root}}$ up-path and the concatenation of the $o - o''$ up-path and the $o'' - o_{\texttt{root}}$ up-path (with o'' only showing up once) must also be an $o - o_{\texttt{root}}$ up-path. Since the $o - o_{\texttt{root}}$ up-path is unique, we know $o'' = o_i$ for some $i \in \{1, \ldots, r\}$.

□

Lemma 42 (Path and Up-Path Composability)**.** *Given two sequences $[o_0^{\mathbf{X}}, o_1^{\mathbf{X}}, \ldots o_p^{\mathbf{X}}]$ and $[o_0^{\mathbf{Y}}, o_1^{\mathbf{Y}}, \ldots o_q^{\mathbf{Y}}]$ where $p \geq 0$ and $q \geq 0$, if $o_p^{\mathbf{X}} = o_0^{\mathbf{Y}}$ and $o_0^{\mathbf{X}}, o_1^{\mathbf{X}}, \ldots o_{p-1}^{\mathbf{X}}, o_0^{\mathbf{Y}}, o_1^{\mathbf{Y}}, \ldots o_q^{\mathbf{Y}}$ are all distinct, then for some R we know $[o_0^{\mathbf{X}}, o_1^{\mathbf{X}}, \ldots o_{p-1}^{\mathbf{X}}, o_0^{\mathbf{Y}}, o_1^{\mathbf{Y}}, \ldots o_q^{\mathbf{Y}}]$*

- *is an up-path in R, if the two sequences are up-paths in R.*
- *is the reverse of an up-path in R, if the two sequences are both reverses of up-paths in R.*
- *is a path in R, if the first sequence is an up-path in R, and the second sequence is a path in R.*
- *is a path in R, if the first sequence is a path in R, and the second sequence is the reverse of an up-path in R.*
- *is a path in R, if the two sequences are paths in R.*

Proof. All except the last one strictly follow the definition of Def. 5 and Def. 2. We now only prove the last one. Case analysis on the $[o_0^{\mathbf{X}}, o_1^{\mathbf{X}}, \ldots, o_p^{\mathbf{X}}]$ path. If it is an up-path, then the conclusion holds because composing an up-path and a path is still a path by the definition of path (see the third bullet in the lemma). If $[o_0^{\mathbf{X}}, o_1^{\mathbf{X}}, \ldots, o_{p-1}^{\mathbf{X}}]$ is not an up-path, then there must exist some $0 \leq r \leq p$ such that $[o_0^{\mathbf{X}}, o_1^{\mathbf{X}}, \ldots, o_{r-1}^{\mathbf{X}}]$ is an up-path, and $[o_p^{\mathbf{X}}, o_{p-1}^{\mathbf{X}}, o_r^{\mathbf{X}}]$ is an up-path. Under this circumstance,we perform a case analysis for the path $[o_0^{\mathbf{Y}}, o_1^{\mathbf{Y}}, \ldots, o_q^{\mathbf{Y}}]$. If this path is formed by starting off having an up-path of longer than 1, *i.e.*, there exists some $0 < s \leq q$, such that $[o_0^{\mathbf{Y}}, \ldots, o_s^{\mathbf{Y}}]$ is an up-path, then since $o_0^{\mathbf{Y}} = o_p^{\mathbf{X}}$ according to the definition, Lem. 41 says that $[o_p^{\mathbf{X}}, o_{p-1}^{\mathbf{X}}, o_r^{\mathbf{X}}]$ and $[o_0^{\mathbf{Y}}, \ldots, o_s^{\mathbf{Y}}]$ must having one contain another. Since both up-paths here have a length of more than 1, then at least one element (other than the starting point) will be the same, say $o_{p-1}^{\mathbf{X}} = o_1^{\mathbf{Y}}$. This would violate the assumption of the subcase, where all elements on $[o_0^{\mathbf{X}}, o_1^{\mathbf{X}}, \ldots, o_{p-1}^{\mathbf{X}}, o_0^{\mathbf{Y}}, o_1^{\mathbf{Y}}, \ldots, o_q^{\mathbf{Y}}]$ are distinct. Contradiction and hence this case is not possible. Therefore, path $[o_0^{\mathbf{Y}}, o_1^{\mathbf{Y}}, \ldots, o_q^{\mathbf{Y}}]$ in this case must strictly be the reverse of an up-path (this also trivially includes the case where $p = 0$), and by the definition of path, $[o_0^{\mathbf{X}}, o_1^{\mathbf{X}}, \ldots o_{p-1}^{\mathbf{X}}, o_0^{\mathbf{Y}}, o_1^{\mathbf{Y}}, \ldots o_q^{\mathbf{Y}}]$ must be a path (see the fourth bullet in the lemma).

□

Lemma 43 (Deterministic rel Definition)**.** *If R is an instantiation tree, then if $rel(R, o', o)$ is defined, it is deterministic.*

Proof. We first need to show the two cases in the rel definition is disjoint. This is obvious since if $o' = o$ we know the $o' - o$ path on R by definition is a sequence of one element. The second case thus can not be satisfied since it demands the sequence to be at least a length of 2. If $o' \neq o$, then the first case can not be satisfied.

Next we show when $o' \neq o$, the second case of the rel definition computes the result deterministically. First note that if an $o' - o$ path exists, according to Lem. 37, we know such a path is unique. The $o' - o$ path computed by the rel function, *i.e.*, $[o_0, o_1, \ldots, o_p]$ for some $p \geq 1$, is thus determined deterministically. Next we need to make sure $relOne$ computes the result deterministically, *i.e.*, $relOne(R, o_i, o_{i+1})$ is deterministic for any $i \in \{0, \ldots, p-1\}$. First note that according to the definition of $o' - o$ path, we know o_0, o_1, ...,

o_p are all distinct. The only way to introduce non-determinism to $relOne$ is for some i, $\langle o_i; o_{i+1}\rangle \in \mathrm{Dom}(R)$ and $\langle o_{i+1}; o_i\rangle \in \mathrm{Dom}(R)$ at the same time. Let $graphRep(R) = \langle VER; EDGE; lab\rangle$. According to the definition of R being an instanatiation tree, we know the underlying undirected unlabeled graph induced by $\langle VER; EDGE\rangle$ is an acyclic graph. But what is implied by the previous case is between o_i and o_{i+1} there are two edges in the undirected unlabeled graph, and a cycle is thus formed by traversing from o_i to o_{i+1} through one edge, and then from o_{i+1} to o_i through the other. Contradiction. Hence $relOne(R, o_i, o_{i+1})$ is deterministic for any $i \in \{0, \ldots, p-1\}$. The last part of the proof is to show $relativizem$ is deterministic. This is obvious according to its definition.

□

Lemma 44 (Domain of rel)**.** *If R is an instantiation tree and $graphRep(R) = \langle VER; EDGE; lab\rangle$, then*

- *given any $o \in VER$ and any $o' \in VER$, $rel(R, o', o)$ is defined.*
- *if $rel(R, o', o)$ is defined, then $o \in VER$ and $o' \in VER$.*

Proof. We first prove the first part. If $o' = o$, then we know the function is defined to be $(\textbf{parent})^0(\textbf{child})^0$. Otherwise, note that according to Lem. 38, we know the $o' - o$ path computed by the rel function, *i.e.*, $[o_0, o_1, \ldots, o_p]$ for some $p \geq 1$, must exist. According to the definition of $o' - o$ path (Def. 5), we know it is the concatenation of an up-path and the reverse of another up-path. According to the definition of up-path, we know $relOne(R, o_i, o_{i+1})$ must be defined for any $i \in \{0, \ldots, p-1\}$. The last part of the proof is to show $relativizem$ is defined, which holds trivially since the function always computes.

For the second part of the proof, note that according to the definition of rel, we know there exists a $o' - o$ path. According to the definition of path and up-path, we know $o \in VER$ and $o' \in VER$.

□

Lemma 45. *If $rel(R, o', o) = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}$ then $\nu_b = 0$.*

Proof. If $o' = o$, then by definition of rel, $relativizem((\textbf{parent})^0(\textbf{child})^0, [\,]) = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}$, hence $\nu_b = 0$ by $relativizem$ definition. Otherwise, $relativizem((\textbf{parent})^0(\textbf{child})^0, [\rho_0, \rho_1, \ldots, \rho_p]) =$

$(\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}, \Sigma$ for some $p \geq 0$. Regardless of ρ_0, ρ_1, ..., ρ_p, we know $\nu_b = 0$ according to the definition of $relativizem$ and $relativize$.

□

Lemma 46 (Free Type Variables in the Result of rel)**.** *If $rel(R, o, o') = \rho$, then $TV(\rho) \subseteq TV(R)$.*

Proof. Strictly follows the definition of rel. □

Lemma 47. *If $rel(R, o^{\mathbf{Y}}, o^{\mathbf{X}}) = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}$, then $rel(R, o^{\mathbf{X}}, o^{\mathbf{Y}}) = (\textbf{parent})^{\nu'_a}(\textbf{child})^{\nu'_b}$ and $\nu_a + \nu'_a \equiv 0$.*

Proof. By Lem. 44 and the assumption that $rel(R, o^{\mathbf{Y}}, o^{\mathbf{X}})$ is defined, we know $rel(R, o^{\mathbf{X}}, o^{\mathbf{Y}})$ is defined as well. If $o^{\mathbf{Y}} = o^{\mathbf{X}}$, then by the definition of the function rel, we know $\nu_a = 0$, $\nu_b = 0$, $\nu'_a = 0$, $\nu'_b = 0$. Then the conclusion holds trivially according to (ICS-tautology). If $o^{\mathbf{Y}} \neq o^{\mathbf{X}}$, by the assumption of $rel(R, o^{\mathbf{Y}}, o^{\mathbf{X}}) = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}, \Sigma$ we know there exists an $o^{\mathbf{Y}} - o^{\mathbf{X}}$ path on the instantiation tree R, *i.e.*, there exists some $p \geq 1$ and distinct $o_0, o_1, o_2, \ldots, o_p$ such that $o_0 = o^{\mathbf{Y}}$, $o_p = o^{\mathbf{X}}$ and

$$\forall i \in \{0, \ldots, p-1\}\ relOne(R, o_i, o_{i+1}) = \rho_i = (\textbf{parent})^{\nu_{(i)a}}(\textbf{parent})^{\nu_{(i)b}}$$

By the definition of rel, we know (the following equations use $\equiv$ rather than $=$ because we have freely moved the order of the linear terms, via (EL-commutativity)):

$$\begin{aligned}\nu_a \equiv & \nu_{(0)a} - \nu_{(0)b} \\ & + \nu_{(1)a} - \nu_{(1)b} \\ & + \nu_{(2)a} - \nu_{(2)b} \\ & \ldots \\ & + \nu_{(p-1)a} - \nu_{(p-1)b}\end{aligned}$$

By Lem. 39, we know $o_p, o_{p-1}, \ldots, o_0$ is an $o^{\mathbf{X}} - o^{\mathbf{Y}}$ path on R. By the definition of rel, we know

$$\begin{aligned}\nu'_a \equiv & \nu_{(p-1)b} - \nu_{(p-1)a} \\ & + \nu_{(p-2)b} - \nu_{(p-2)a} \\ & \ldots \\ & + \nu_{(0)b} - \nu_{(0)a}\end{aligned}$$

Obviously the conclusion holds. □

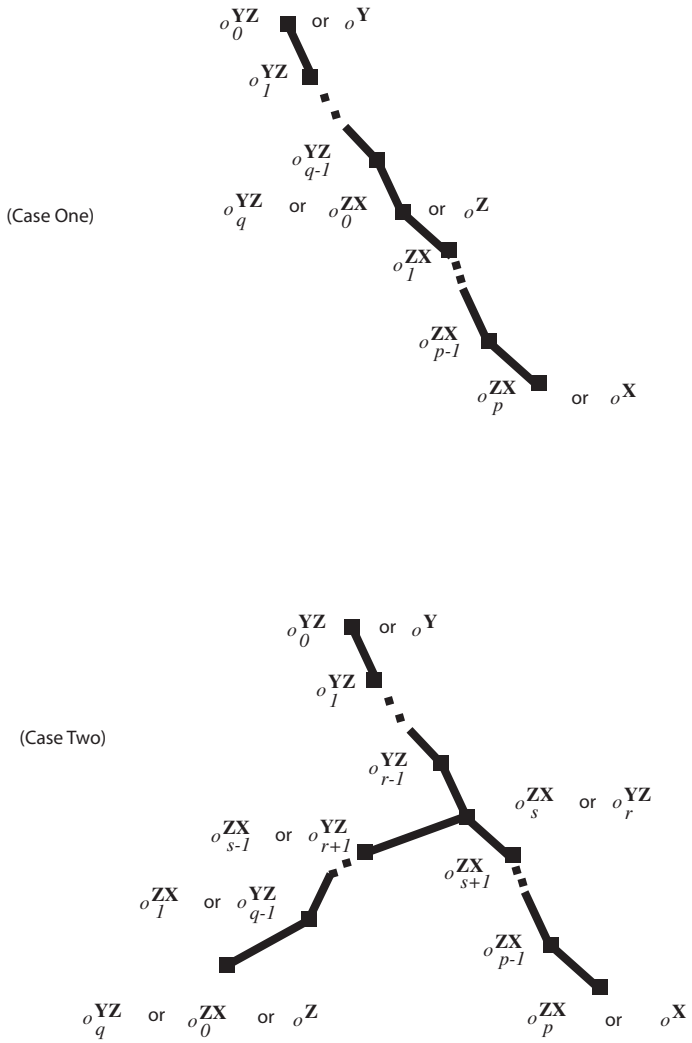


Figure C.1: Two Cases for Lem. 48

Lemma 48 (Function rel is Transitive)**.** *If*

$$\begin{aligned} rel(R, o^{\mathbf{Z}}, o^{\mathbf{X}}) &= \rho^{\mathbf{ZX}} \\ rel(R, o^{\mathbf{Y}}, o^{\mathbf{Z}}) &= \rho^{\mathbf{YZ}} \\ relativize(\rho^{\mathbf{YZ}}, \rho^{\mathbf{ZX}}) &= (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}, \Sigma \end{aligned}$$

then

$$\begin{aligned} rel(R, o^{\mathbf{Y}}, o^{\mathbf{X}}) &= (\textbf{parent})^{\nu'_a}(\textbf{child})^{\nu'_b} \\ \nu_a &\equiv \nu'_a \\ \nu_b &\equiv \nu'_b \end{aligned}$$

Proof. Let $\rho^{\mathbf{ZX}} = (\textbf{parent})^{\nu_a^{\mathbf{ZX}}}(\textbf{child})^{\nu_b^{\mathbf{ZX}}}$ and $\rho^{\mathbf{YZ}} = (\textbf{parent})^{\nu_a^{\mathbf{YZ}}}(\textbf{child})^{\nu_b^{\mathbf{YZ}}}$.

Case $o^{\mathbf{Y}} = o^{\mathbf{Z}}$

By the assumption of $rel(R, o^{\mathbf{Y}}, o^{\mathbf{Z}}) = \rho^{\mathbf{ZX}}$ and the definition of rel, we know $\nu_a^{\mathbf{YZ}} = 0$ and $\nu_b^{\mathbf{YZ}} = 0$. By the assumption of $relativize(\rho^{\mathbf{YZ}}, \rho^{\mathbf{ZX}}) = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}, \Sigma$ and the definition of $relativize$, we know $\nu_a = \nu_a^{\mathbf{ZX}} - \nu_b^{\mathbf{ZX}} + 0$ and $\nu_b = 0$. By Lem. 45 and the assumption of $rel(R, o^{\mathbf{Z}}, o^{\mathbf{X}}) = \rho^{\mathbf{ZX}}$, we know $\nu_b^{\mathbf{ZX}} = 0$. Hence $\nu_a \equiv \nu^{\mathbf{ZX}_a}$ by (EL-cancellation), (EL-identity). We also know $rel(R, o^{\mathbf{Y}}, o^{\mathbf{X}}) = rel(R, o^{\mathbf{Z}}, o^{\mathbf{X}}) = \rho^{\mathbf{ZX}}$. Thus $\nu'_a = \nu_a^{\mathbf{ZX}}$ and $\nu'_b = \nu_b^{\mathbf{ZX}}$. Thus $\nu_a \equiv \nu'_a$ and $\nu_b \equiv \nu'_b$ trivially hold.

Case $o^{\mathbf{Z}} = o^{\mathbf{X}}$

By the assumption of $rel(R, o^{\mathbf{Z}}, o^{\mathbf{X}}) = \rho^{\mathbf{ZX}}$ and the definition of rel, we know $\nu_a^{\mathbf{ZX}} = 0$ and $\nu_b^{\mathbf{ZX}} = 0$. By the assumption of $relativize(\rho^{\mathbf{YZ}}, \rho^{\mathbf{ZX}}) = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}, \Sigma$ and the definition of $relativize$, we know $\nu_a = 0 - 0 + \nu_a^{\mathbf{YZ}}$ and $\nu_b = \nu_b^{\mathbf{YZ}}$. Hence $\nu_a \equiv \nu^{\mathbf{YZ}_a}$ by (EL-cancellation), (EL-identity). We also know $rel(R, o^{\mathbf{Y}}, o^{\mathbf{X}}) = rel(R, o^{\mathbf{Y}}, o^{\mathbf{Z}}) = \rho^{\mathbf{YZ}}$. Thus $\nu'_a = \nu_a^{\mathbf{YZ}}$ and $\nu'_b = \nu_b^{\mathbf{YZ}}$. Thus $\nu_a \equiv \nu'_a$ and $\nu_b \equiv \nu'_b$ by (EL-cancellation).

Case $o^{\mathbf{Y}} \neq o^{\mathbf{Z}}$ and $o^{\mathbf{Z}} \neq o^{\mathbf{X}}$

By assumption $rel(R, o^{\mathbf{Z}}, o^{\mathbf{X}}) = \rho^{\mathbf{ZX}}$ and $o^{\mathbf{Z}} \neq o^{\mathbf{X}}$, there exists an $o^{\mathbf{Z}} - o^{\mathbf{X}}$ path $[o_0^{\mathbf{ZX}}, o_1^{\mathbf{ZX}}, o_2^{\mathbf{ZX}}, \ldots, o_p^{\mathbf{ZX}}]$ such that $p \geq 1$, $o_0^{\mathbf{ZX}} = o^{\mathbf{Z}}$, $o_p^{\mathbf{ZX}} = o^{\mathbf{X}}$ and

$$\begin{aligned} \forall i \in \{0, \ldots, p-1\}\ relOne(R, o_i^{\mathbf{ZX}}, o_{i+1}^{\mathbf{ZX}}) &= \rho_i^{\mathbf{ZX}} = (\textbf{parent})^{\nu_{(i)a}^{\mathbf{ZX}}}(\textbf{parent})^{\nu_{(i)b}^{\mathbf{ZX}}} \\ relativizem((\textbf{parent})^0(\textbf{child})^0, [\rho_0^{\mathbf{ZX}}, \rho_1^{\mathbf{ZX}}, \rho_2^{\mathbf{ZX}}, \ldots \rho_{p-1}^{\mathbf{ZX}}) &= \rho^{\mathbf{ZX}}, \Sigma^{\mathbf{ZX}} \end{aligned}$$

By the definition of $relativizem$, we know (the following equations use $\equiv$ rather than $=$ because we have freely moved the order of the linear terms, via (EL-commutativity)):

$$\begin{aligned}\nu_a^{\mathbf{ZX}} \equiv & \nu_{(0)a}^{\mathbf{ZX}} - \nu_{(0)b}^{\mathbf{ZX}} \\ & + \nu_{(1)a}^{\mathbf{ZX}} - \nu_{(1)b}^{\mathbf{ZX}} \\ & + \nu_{(2)a}^{\mathbf{ZX}} - \nu_{(2)b}^{\mathbf{ZX}} \\ & \cdots \\ & + \nu_{(p-1)a}^{\mathbf{ZX}} - \nu_{(p-1)b}^{\mathbf{ZX}} \\ \nu_b^{\mathbf{ZX}} = & 0\end{aligned}$$

Similarly, by assumption $rel(R, o^{\mathbf{Y}}, o^{\mathbf{Z}}) = \rho^{\mathbf{YZ}}$ and $o^{\mathbf{Y}} \neq o^{\mathbf{Z}}$, there exists an $o^{\mathbf{Y}} - o^{\mathbf{Z}}$ path in the form of $[o_0^{\mathbf{YZ}}, o_1^{\mathbf{YZ}}, o_2^{\mathbf{YZ}}, \ldots, o_q^{\mathbf{YZ}}]$ such that $q \geq 1$, $o_0^{\mathbf{YZ}} = o^{\mathbf{Y}}$, $o_q^{\mathbf{YZ}} = o^{\mathbf{Z}}$ and

$$\forall i \in \{0, \ldots, q-1\}\ relOne(R, o_i^{\mathbf{YZ}}, o_{i+1}^{\mathbf{YZ}}) = \rho_i^{\mathbf{YZ}} = (\mathbf{parent})^{\nu_{(i)a}^{\mathbf{YZ}}} (\mathbf{parent})^{\nu_{(i)b}^{\mathbf{YZ}}}$$
$$relativizem((\mathbf{parent})^0(\mathbf{child})^0, [\rho_0^{\mathbf{YZ}}, \rho_1^{\mathbf{YZ}}, \rho_2^{\mathbf{YZ}}, \ldots \rho_{q-1}^{\mathbf{YZ}}) = \rho^{\mathbf{YZ}}, \Sigma^{\mathbf{YZ}}$$

By the definition of $relativizem$, we know:

$$\begin{aligned}\nu_a^{\mathbf{YZ}} \equiv & \nu_{(0)a}^{\mathbf{YZ}} - \nu_{(0)b}^{\mathbf{YZ}} \\ & + \nu_{(1)a}^{\mathbf{YZ}} - \nu_{(1)b}^{\mathbf{YZ}} \\ & + \nu_{(2)a}^{\mathbf{YZ}} - \nu_{(2)b}^{\mathbf{YZ}} \\ & \cdots \\ & + \nu_{(q-1)a}^{\mathbf{YZ}} - \nu_{(q-1)b}^{\mathbf{YZ}} \\ \nu_b^{\mathbf{YZ}} = & 0\end{aligned}$$

By the assumption of $relativize(\rho^{\mathbf{YZ}}, \rho^{\mathbf{ZX}}) = (\mathbf{parent})^{\nu_a}(\mathbf{child})^{\nu_b}, \Sigma$ and the definition of $relativize$, we know

$$\begin{aligned}\nu_a \equiv & \nu_{(0)a}^{\mathbf{ZX}} - \nu_{(0)b}^{\mathbf{ZX}} \\ & + \nu_{(1)a}^{\mathbf{ZX}} - \nu_{(1)b}^{\mathbf{ZX}} \\ & + \nu_{(2)a}^{\mathbf{ZX}} - \nu_{(2)b}^{\mathbf{ZX}} \\ & \cdots \\ & + \nu_{(p-1)a}^{\mathbf{ZX}} - \nu_{(p-1)b}^{\mathbf{ZX}} \\ & + \nu_{(0)a}^{\mathbf{YZ}} - \nu_{(0)b}^{\mathbf{YZ}} \\ & + \nu_{(1)a}^{\mathbf{YZ}} - \nu_{(1)b}^{\mathbf{YZ}} \\ & + \nu_{(2)a}^{\mathbf{YZ}} - \nu_{(2)b}^{\mathbf{YZ}} \\ & \cdots\end{aligned} \tag{48.1}$$

$$
\begin{aligned}
&+\nu^{\mathbf{YZ}}_{(q-1)a} - \nu^{\mathbf{YZ}}_{(q-1)b} \\
&-0 \\
\nu_b \equiv &0
\end{aligned}
\tag{48.2}
$$

To reach the conclusion, we have two subcases, illustrated in Fig. C.1. They are differentiated by how the path $[o^{\mathbf{YZ}}_0, o^{\mathbf{YZ}}_1, \dots o^{\mathbf{YZ}}_q]$ overlaps with path $[o^{\mathbf{ZX}}_0, o^{\mathbf{ZX}}_1, \dots o^{\mathbf{ZX}}_p]$:

$$
\forall i \in \{0, \dots, q-1\}. o^{\mathbf{YZ}}_i \notin \{o^{\mathbf{ZX}}_0, \dots, o^{\mathbf{ZX}}_p\} \tag{**Case 48.A**}
$$

$$
\begin{cases}
r \in \{0, \dots, q-1\} \\
s \in \{0, \dots, p\} \\
o^{\mathbf{YZ}}_r = o^{\mathbf{ZX}}_s \\
\forall j \in \{0, \dots, r-1\}. o^{\mathbf{YZ}}_j \notin \{o^{\mathbf{ZX}}_0, \dots, o^{\mathbf{ZX}}_p\}
\end{cases}
\tag{**Case 48.B**}
$$

Obviously, the two cases are disjoint and cover all the possibilities. This is because (**Case 48.A**) says the two paths do not overlap at all, while (**Case 48.B**) says they do overlap, and since they overlap, let the first node on the path of $[o^{\mathbf{YZ}}_0, o^{\mathbf{YZ}}_1, \dots o^{\mathbf{YZ}}_q]$ to overlap the other path be $o^{\mathbf{YZ}}_r$. These two cases are illustrated by Fig. C.1 (a) and Fig. C.1 (b) respectively, and we now divide the proof into two.

Subcase (**Case 48.A**)

According to Lem. 42, we know $[o^{\mathbf{YZ}}_0, o^{\mathbf{YZ}}_1, \dots, o^{\mathbf{YZ}}_{q-1}, o^{\mathbf{ZX}}_0, o^{\mathbf{ZX}}_1, \dots, o^{\mathbf{ZX}}_p]$ is an $o^{\mathbf{Y}} - o^{\mathbf{X}}$ path. Thus according to the definition of rel we know

$$
\begin{aligned}
\nu'_a \equiv &\nu^{\mathbf{YZ}}_{(0)a} - \nu^{\mathbf{YZ}}_{(0)b} \\
&+\nu^{\mathbf{YZ}}_{(1)a} - \nu^{\mathbf{YZ}}_{(1)b} \\
&+\nu^{\mathbf{YZ}}_{(2)a} - \nu^{\mathbf{YZ}}_{(2)b} \\
&\dots \\
&+\nu^{\mathbf{YZ}}_{(q-1)a} - \nu^{\mathbf{YZ}}_{(q-1)b} \\
&+\nu^{\mathbf{ZX}}_{(0)a} - \nu^{\mathbf{ZX}}_{(0)b} \\
&+\nu^{\mathbf{ZX}}_{(1)a} - \nu^{\mathbf{ZX}}_{(1)b} \\
&+\nu^{\mathbf{ZX}}_{(2)a} - \nu^{\mathbf{ZX}}_{(2)b} \\
&\dots \\
&+\nu^{\mathbf{ZX}}_{(p-1)a} - \nu^{\mathbf{ZX}}_{(p-1)b} \\
\nu'_b = &0
\end{aligned}
$$

Note that the right hand side of the two equations above for ν'_a and ν'_b are identical for the two equations for ν_a and ν_b in (**48.1**). Thus $\nu_a \equiv \nu'_a$ and $\nu_b \equiv \nu'_b$ hold according to (EL-transitivity).

Subcase (**Case 48.B**)

By Lem. 40 (sub-path), we know $[o_s^{\mathbf{ZX}}, \ldots, o_p^{\mathbf{ZX}}]$ as a subsequence is also a path. Similarly, $[o_0^{\mathbf{YZ}}, \ldots, o_r^{\mathbf{YZ}}]$ is a path. According to the condition of (**Case 48.B**) such that $o_r^{\mathbf{YZ}} = o_s^{\mathbf{ZX}}$ and $\forall j \in \{0, \ldots, r-1\}.o_j^{\mathbf{YZ}} \notin \{o_0^{\mathbf{ZX}}, \ldots, o_p^{\mathbf{ZX}}\}$, we know the two paths above do not have overlapping nodes (as illustrated in the figure as well). Thus, according to Lem. 42 (path composability), we know $[o_0^{\mathbf{YZ}}, o_1^{\mathbf{YZ}}, \ldots, o_r^{\mathbf{YZ}}, o_{s+1}^{\mathbf{ZX}}, \ldots, o_p^{\mathbf{ZX}}]$ is a path. According to the definition of rel, we have

$$\begin{aligned}
\nu'_a \equiv & \nu^{\mathbf{YZ}}_{(0)a} - \nu^{\mathbf{YZ}}_{(0)b} \\
& + \nu^{\mathbf{YZ}}_{(1)a} - \nu^{\mathbf{YZ}}_{(1)b} \\
& + \nu^{\mathbf{YZ}}_{(2)a} - \nu^{\mathbf{YZ}}_{(2)b} \\
& \ldots \\
& + \nu^{\mathbf{YZ}}_{(r-1)a} - \nu^{\mathbf{YZ}}_{(r-1)b} \\
& + \nu^{\mathbf{ZX}}_{(s)a} - \nu^{\mathbf{ZX}}_{(s)b} \\
& \ldots \\
& + \nu^{\mathbf{ZX}}_{(p-1)a} - \nu^{\mathbf{ZX}}_{(p-1)b} \\
\nu'_b = & 0
\end{aligned} \tag{48.3}$$

By Lem. 40 (sub-path), we know $[o_0^{\mathbf{ZX}}, \ldots, o_s^{\mathbf{ZX}}]$ as a subsequence of path $[o_0^{\mathbf{ZX}}, \ldots, o_p^{\mathbf{ZX}}]$ is a $o_0^{\mathbf{ZX}} - o_s^{\mathbf{ZX}}$ path. Similarly we know $[o_r^{\mathbf{YZ}}, \ldots, o_q^{\mathbf{YZ}}]$ as a subsequence of $[o_0^{\mathbf{YZ}}, \ldots, o_q^{\mathbf{YZ}}]$ is a $o_r^{\mathbf{YZ}} - o_q^{\mathbf{YZ}}$ path. Earlier we know $o_0^{\mathbf{ZX}} = o_q^{\mathbf{YZ}} = o^{\mathbf{Z}}$ and $o_s = o_r$. By Lem. 39, we know $[o_0^{\mathbf{ZX}}, \ldots, o_s^{\mathbf{ZX}}]$ and $[o_r^{\mathbf{YZ}}, \ldots, o_q^{\mathbf{YZ}}]$ are simply reversals of each other. Hence

$$\begin{aligned}
o_s^{\mathbf{ZX}} &= o_r^{\mathbf{YZ}} \\
o_{s-1}^{\mathbf{ZX}} &= o_{r+1}^{\mathbf{YZ}} \\
&\ldots \\
o_1^{\mathbf{ZX}} &= o_{q-1}^{\mathbf{YZ}} \\
o_0^{\mathbf{ZX}} &= o_q^{\mathbf{YZ}}
\end{aligned}$$

Hence by the definition of $relOne$, we know

$$\begin{aligned}
\nu^{\mathbf{ZX}}_{(s-1)a} &= \nu^{\mathbf{YZ}}_{(r)b} \\
\nu^{\mathbf{ZX}}_{(s-1)b} &= \nu^{\mathbf{YZ}}_{(r)a} \\
\nu^{\mathbf{ZX}}_{(s-2)a} &= \nu^{\mathbf{YZ}}_{(r+1)b} \\
\nu^{\mathbf{ZX}}_{(s-2)b} &= \nu^{\mathbf{YZ}}_{(r+1)a} \\
&\ldots \\
\nu^{\mathbf{ZX}}_{(0)a} &= \nu^{\mathbf{YZ}}_{(q-1)b} \\
\nu^{\mathbf{ZX}}_{(0)b} &= \nu^{\mathbf{YZ}}_{(q-1)a}
\end{aligned} \tag{48.4}$$

Earlier we have computed ν_a and ν_b as (**48.1**). Substituting $\nu^{\mathbf{ZX}}_{(s-1)a}$, $\nu^{\mathbf{ZX}}_{(s-1)b}$, $\ldots$, $\nu^{\mathbf{ZX}}_{(0)a}$, $\nu^{\mathbf{ZX}}_{(0)b}$ with its counterparts according to the equations above, together with the basic definition of $\equiv$ such as $\nu_1 + \nu_2 - \nu_2 \equiv \nu_1$ ((EL-cancellation)) and $\nu_1 + \nu_2 \equiv \nu_2 + \nu_1$ ((EL-commutivity)), we can rewrite (**48.1**) as

$$
\begin{array}{rcll}
\nu_a \equiv & \nu^{\mathbf{YZ}}_{(q-1)b} - \nu^{\mathbf{YZ}}_{(q-1)a} & \equiv & \nu^{\mathbf{ZX}}_{(s)a} - \nu^{\mathbf{ZX}}_{(s)b} \qquad \equiv \nu'_a \\
& +\nu^{\mathbf{YZ}}_{(q-2)b} - \nu^{\mathbf{YZ}}_{(q-2)a} & & +\nu^{\mathbf{ZX}}_{(s+1)a} - \nu^{\mathbf{ZX}}_{(s+1)b} \\
& \ldots & & \ldots \\
& +\nu^{\mathbf{YZ}}_{(r)b} - \nu^{\mathbf{YZ}}_{(r)a} & & +\nu^{\mathbf{ZX}}_{(p-1)a} - \nu^{\mathbf{ZX}}_{(p-1)b} \\
& +\nu^{\mathbf{ZX}}_{(s)a} - \nu^{\mathbf{ZX}}_{(s)b} & & +\nu^{\mathbf{YZ}}_{(0)a} - \nu^{\mathbf{YZ}}_{(0)b} \\
& +\nu^{\mathbf{ZX}}_{(s+1)a} - \nu^{\mathbf{ZX}}_{(s+1)b} & & +\nu^{\mathbf{YZ}}_{(1)a} - \nu^{\mathbf{YZ}}_{(1)b} \\
& \ldots & & +\nu^{\mathbf{YZ}}_{(2)a} - \nu^{\mathbf{YZ}}_{(2)b} \\
& +\nu^{\mathbf{ZX}}_{(p-1)a} - \nu^{\mathbf{ZX}}_{(p-1)b} & & \ldots \\
& +\nu^{\mathbf{YZ}}_{(0)a} - \nu^{\mathbf{YZ}}_{(0)b} & & +\nu^{\mathbf{YZ}}_{(r-1)a} - \nu^{\mathbf{YZ}}_{(r-1)b} \\
& +\nu^{\mathbf{YZ}}_{(1)a} - \nu^{\mathbf{YZ}}_{(1)b} & & \\
& +\nu^{\mathbf{YZ}}_{(2)a} - \nu^{\mathbf{YZ}}_{(2)b} & & \\
& \ldots & & \\
& +\nu^{\mathbf{YZ}}_{(r-1)a} - \nu^{\mathbf{YZ}}_{(r-1)b} & & \\
& +\nu^{\mathbf{YZ}}_{(r)a} - \nu^{\mathbf{YZ}}_{(r)b} & & \\
& +\nu^{\mathbf{YZ}}_{(r+1)a} - \nu^{\mathbf{YZ}}_{(r+1)b} & & \\
& \ldots & & \\
& +\nu^{\mathbf{YZ}}_{(q-1)a} - \nu^{\mathbf{YZ}}_{(q-1)b} & & \\
\nu_b = & 0 = \nu'_b & &
\end{array}
$$

□

Lemma 49 (Function rel is Differential)**.** *If*

$$rel(R, o^{\mathbf{X}}, o^{\mathbf{Z}}) = \rho^{\mathbf{XZ}} \tag{49.1}$$

$$rel(R, o^{\mathbf{Y}}, o^{\mathbf{Z}}) = \rho^{\mathbf{YZ}} \tag{49.2}$$

$$\rho^{\mathbf{XZ}} = (\textbf{parent})^{\nu^{\mathbf{XZ}}_a}(\textbf{child})^{\nu^{\mathbf{XZ}}_b}$$

$$\rho^{\mathbf{ZX}} = (\textbf{parent})^{\nu^{\mathbf{XZ}}_b}(\textbf{child})^{\nu^{\mathbf{XZ}}_a}$$

$$relativize(\rho^{\mathbf{YZ}}, \rho^{\mathbf{ZX}}) = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b}, \Sigma \tag{49.3}$$

then

$$rel(R, o^{\mathbf{Y}}, o^{\mathbf{X}}) = (\textbf{parent})^{\nu'_a}(\textbf{child})^{\nu'_b}$$

$$\nu_a =_{\mathbf{s}} \nu'_a$$

$$\nu_b =_{\mathbf{s}} \nu'_b$$

Proof. Let $\rho^{\mathbf{YZ}} = (\textbf{parent})^{\nu^{\mathbf{YZ}}_a}(\textbf{child})^{\nu^{\mathbf{YZ}}_b}$. By the assumption of $rel(R, o^{\mathbf{X}}, o^{\mathbf{Z}}) = \rho^{\mathbf{XZ}}$, and Lem. 47, we know

$$rel(R, o^{\mathbf{Z}}, o^{\mathbf{X}}) = (\textbf{parent})^{\nu_a^{\mathbf{ZX}}}(\textbf{child})^{\nu_b^{\mathbf{ZX}}} \tag{49.4}$$

$$\nu^{\mathbf{ZX}} + \nu^{\mathbf{XZ}} \equiv 0 \tag{49.5}$$

By the definition of $relativize$, we know for some Σ':

$$relativize(\rho^{\mathbf{YZ}}, (\textbf{parent})^{\nu_a^{\mathbf{ZX}}}(\textbf{child})^{\nu_b^{\mathbf{ZX}}}) = (\textbf{parent})^{\nu_a^{\mathbf{ZX}} - \nu_b^{\mathbf{ZX}} + \nu_a^{\mathbf{YZ}}}(\textbf{child})^{\nu_b^{\mathbf{YZ}}}, \Sigma'$$

With this, (**49.2**), (**49.4**), and Lem. 48, we know

$$\nu'_a \equiv \nu_a^{\mathbf{ZX}} - \nu_b^{\mathbf{ZX}} + \nu_a^{\mathbf{YZ}}$$
$$\nu'_b \equiv \nu_b^{\mathbf{YZ}}$$

and we know

$$\begin{aligned}
\nu_a^{\mathbf{ZX}} - \nu_b^{\mathbf{ZX}} + \nu_a^{\mathbf{YZ}} =& \text{By (\textbf{49.4}), Lem. 45} \\
& \nu_a^{\mathbf{ZX}} + \nu_a^{\mathbf{YZ}} \\
=& \text{By (\textbf{49.5})} \\
& - \nu_a^{\mathbf{XZ}} + \nu_a^{\mathbf{YZ}} \\
=& \text{By (\textbf{49.1}), Lem. 45} \\
& \nu_b^{\mathbf{XZ}} - \nu_a^{\mathbf{XZ}} + \nu_a^{\mathbf{YZ}} \\
=& \text{By (\textbf{49.3}), the definition of } relativize \\
& \nu_a
\end{aligned}$$

$$\begin{aligned}
\nu_b^{\mathbf{YZ}} =& \text{By (\textbf{49.3}), the definition of } relativize \\
& \nu_b
\end{aligned}$$

Hence the conclusion holds.

□

C.9 Properties of Configuration Strengthening

Lemma 50 (Type Preservation for Values Over Configuration Strengthening)**.** *Given the following conditions*

$$v \in \mathbb{V}$$
$$\Gamma_1, \mathcal{C} \vdash v : \tau \backslash \Sigma, \Pi$$
$$\Gamma_1(\textbf{iconf}) \preceq_{\texttt{iconf}} \Gamma_2(\textbf{iconf})$$
$$\Gamma_1(\textbf{currentRt}) = \Gamma_2(\textbf{currentRt})$$

$$
\begin{array}{lcll}
\boldsymbol{\Delta}_1 \preceq_{\texttt{iconf}} \boldsymbol{\Delta}_2 & \stackrel{\text{def}}{=} & (\forall o \in \mathrm{Dom}(\mathcal{H}_1).\mathcal{H}_1(o) = \mathcal{H}_2(o)) \\
& \wedge & (\forall c \in \mathrm{Dom}(W_1).W_1(c) = W_2(c)) \\
& \wedge & (\forall \langle o; o' \rangle \in \mathrm{Dom}(R_1).R_1(\langle o; o' \rangle) = R_2(\langle o; o' \rangle)) \\
& \text{where} & \boldsymbol{\Delta}_i = \langle \mathcal{H}_i; W_i; R_i \rangle, \forall i = \{1, 2\}
\end{array}
$$

Figure C.2: Definition: Strength of Configuration Types

then $\Gamma_2, \mathcal{C} \vdash v : \tau \backslash \Sigma, \Pi$

Proof. Induction on $\Gamma_1, \mathcal{C} \vdash v : \tau \backslash \Sigma, \Pi$ and case analysis on the last step of the derivation leading to this judgment. If the last step is an instance of (T-Sub), we thus know $\Gamma_1, \mathcal{C} \vdash v : \tau' \backslash \Sigma', \Pi$, $\emptyset \vdash \tau' <: \tau \backslash \Sigma_{\texttt{sub}}$ and $\Sigma = \Sigma' \cup \Sigma_{\texttt{sub}}$. By induction, we know $\Gamma_2, \mathcal{C} \vdash v : \tau' \backslash \Sigma', \Pi$. Thus by (T-Sub), we know $\Gamma_1, \mathcal{C} \vdash v : \tau \backslash \Sigma' \cup \Sigma_{\texttt{sub}}, \Pi$, which is the conclusion. If the last step is an instance of (T-Unit), (T-Int), (T-Null), we know the conclusion holds trivially. All other typing rules are not possible due to the syntax of v, except (T-CID) and (T-RID), which we discuss now. First let $\Gamma_1(\textbf{iconf}) = \langle \mathcal{H}_1; W_1; R_1 \rangle$ and $\Gamma_2(\textbf{iconf}) = \langle \mathcal{H}_2; W_2; R_2 \rangle$ and $\Gamma_1(\textbf{currentRt}) = \Gamma_2(\textbf{currentRt}) = o_1$.

If the last step is an instance of (T-CID), we know $v = c^{side}$ for some c^{side}, $\Sigma = \emptyset$, $\Pi = \emptyset$, $\tau = \mathsf{k}_1$ for some k_1 and $W_1(c) = \langle o_1; \mathsf{k}_1; o_2; \mathsf{k}_2 \rangle$ or else $\langle o_2; \mathsf{k}_2; o_1; \mathsf{k}_1 \rangle$. According to the definition of $\preceq_{\texttt{iconf}}$, we know $W_2(c) = W_1(c)$. By (T-CID), the conclusion holds.

If the last step is an instance of (T-RID), we know $v = o$ for some o and

$$
\begin{array}{r}
\mathcal{H}_1(o) = \langle \mathsf{a}; \sigma; \mathcal{S} \rangle \\
\mathcal{C}(\mathsf{a}) = \overline{\forall \langle \alpha; \beta \rangle}.\langle \mathcal{K}; \mathcal{M}; \mathcal{F} \rangle \\
rel(R_1, o, o_1) = \rho, \Sigma_{\texttt{rel}}
\end{array}
$$

$\Sigma = \Sigma_{\texttt{rel}} \cup subPed(\rho', \rho)$, $\Pi = \emptyset$. $\tau = (\mu \mathsf{a}.\mathcal{K}[\sigma]\ @\rho')$. By the definition of $\preceq_{\texttt{iconf}}$, we know $\mathcal{H}_2(o) = \mathcal{H}_1(o)$. Also we know $(\forall \langle o; o' \rangle \in \mathrm{Dom}(R_1).R_1(\langle o; o' \rangle) = R_2(\langle o; o' \rangle))$. The rel function is defined on top of it; and it can by easily seen that $rel(R_2, o, o_1) = rel(R_1, o, o_1)$. The conclusion thus holds by (T-RID).

□

Lemma 51 (Heap Type Preservation over Configuration Strengthening)**.** *Given* $\boldsymbol{\Delta}_1 \preceq_{\texttt{iconf}} \boldsymbol{\Delta}_2, G, \mathcal{C}, \boldsymbol{\Delta}_1 \vdash_{\texttt{h}}$

$H : \mathcal{H}\backslash\Sigma$, *then* $G, \mathcal{C}, \mathbf{\Delta}_2 \vdash_{\mathtt{h}} H : \mathcal{H}\backslash\Sigma$

Proof. The key to this is whether the field store types are preserved or not. It is indeed true because of the previous lemma since all field stores have values. □

Lemma 52 (Configuration Strengthening Over Reductions)**.** *Given*

$$\Delta_1, exd_1 \xrightarrow{C,\ o} \Delta_2, exd_2$$
$$G, \mathcal{C} \vdash_{\mathtt{iconf}} \Delta_i : \mathbf{\Delta}_i\backslash\Sigma_i, \forall i = \{1, 2\}$$

then $\mathbf{\Delta}_1 \preceq_{\mathtt{iconf}} \mathbf{\Delta}_2$.

Proof. Straightforward by inductions on $\Delta_1, exd_1 \xrightarrow{C,\ o} \Delta_2, exd_2$, and case analysis for exd_1 on the last induction step. Case analysis on exd_1.

Case: $exd_1 = \mathbf{E}_a[\, exd_{\mathtt{redexa1}}\,]$ where $\mathbf{E}_a \neq \bullet$

By (R-Context), we immediately have:

$$\Delta_1, exd_{\mathtt{redexa1}} \xrightarrow{C,\ o} \Delta_2, exd_{\mathtt{redexa2}}$$

The conclusion thus holds by induction.

Case: $exd_1 = \mathbf{in}_{\rightarrow}(o^{\mathbf{Y}}, exd_1^{\mathbf{Y}}), exd_1^{\mathbf{Y}} \notin \mathbb{V}$

There are two possibilities exd_1 can be reduced, via (R-In-1) or (R-Exception-In). If the reduction follows (R-Exception-In), we immediately know $\mathbf{\Delta}_2 = \mathbf{\Delta}_1$, and thus the conclusion holds trivially. If the reduction follows (R-In-1), we immediately know:

$$\Delta_1, exd_1^{\mathbf{Y}} \xrightarrow{C,\ o^{\mathbf{Y}}} \Delta_2, exd_2^{\mathbf{Y}}$$
$$exd_2^{\mathbf{Y}} \neq \textbf{exception}$$

The conclusion thus holds by induction.

Other Cases

In all other cases, such a lemma is only concerned with the structural difference between $\mathbf{\Delta}_1$ and $\mathbf{\Delta}_2$, and therefore there is no surprise for this proof. The specific structures of $\mathbf{\Delta}_1$, $\mathbf{\Delta}_2$ that conform to the well-formedness constraints can be easily proved. □

C.10 Substitution Lemma

Lemma 53 (Substitution)**.** *If metavariable* se *represents either* x *or* **current***, and*

$$\Gamma \rhd (se : \tau'), \mathcal{C} \vdash e : \tau \backslash \Sigma, \Pi \tag{53.1}$$
$$\Gamma, \mathcal{C} \vdash exd : \tau' \backslash \Sigma', \Pi' \tag{53.2}$$
$$e \in \mathbb{EXP}$$

then $\Gamma, \mathcal{C} \vdash e\{exd/se\} : \tau \backslash \Sigma_{\mathtt{s}}, \Pi_{\mathtt{s}}$ *and* $\Sigma_{\mathtt{s}} \subseteq \Sigma \cup \Sigma'$*, and* $\Pi_{\mathtt{s}} \subseteq \Pi \cup \Pi'$.

Proof. Induction on (**53.1**) and case analysis on e. Note that $e \in \mathbb{EXP}$ instead of $\mathbb{EXD}$. Thus, expressions that can only show up at run time are not possible cases for this lemma.

Case: (T-Sub) concludes (**53.1**)

Thus by (T-Sub), we know

$$\Gamma \rhd (se : \tau'), \mathcal{C} \vdash e : \tau \backslash \Sigma, \Pi \tag{53.3}$$
$$\emptyset \vdash \tau <: \tau \backslash \Sigma_{\mathtt{expsub}} \tag{53.4}$$
$$\Sigma = \Sigma \cup \Sigma_{\mathtt{expsub}} \tag{53.5}$$

By induction hypothesis, (**53.3**), (**53.2**), we immediately know $\Gamma, \mathcal{C} \vdash e\{exd/se\} : \tau \backslash \Sigma_{\mathtt{s}}, \Pi_{\mathtt{s}}$ and $\Sigma_{\mathtt{s}} \subseteq \Sigma \cup \Sigma'$, and $\Pi_{\mathtt{s}} \subseteq \Pi \cup \Pi'$. By (T-Sub) and (**53.4**), we know $\Gamma, \mathcal{C} \vdash e\{exd/se\} : \tau \backslash \Sigma_{\mathtt{s}} \cup \Sigma_{\mathtt{expsub}}, \Pi_{\mathtt{s}}$ By (**53.5**) and the previous fact that $\Sigma_{\mathtt{s}} \subseteq \Sigma \cup \Sigma'$, we know $\Sigma_{\mathtt{s}} \cup \Sigma_{\mathtt{expsub}} \subseteq \Sigma \cup \Sigma' \cup \Sigma_{\mathtt{expsub}} = \Sigma \cup \Sigma'$. The conclusion holds.

Case: (T-Sub) does not conclude (**53.1**), $e =$ **create** *Ped* a

In this case the last step must be an instance of (T-Create). Note that the prequents of (T-Create) does not depend on Γ. Thus if (**53.1**) holds, we know $\Gamma, \mathcal{C} \vdash e : \tau \backslash \Sigma, \Pi$. Since se does not occur in **create** *Ped* a, we know $e\{exd/se\} = e$. Thus, we know $\Gamma, \mathcal{C} \vdash e\{exd/se\} : \tau \backslash \Sigma, \Pi$. The conclusion holds since $\Sigma \subseteq \Sigma \cup \Sigma'$ and $\Pi \subseteq \Pi \cup \Pi'$.

Cases similar to the previous case where $e =$ **create** *Ped* a

The following cases can be proved in an almost identical way as $e =$ **create** *Ped* a. The key common things among them are se always occurs free in these expressions, and their corresponding typing rules do not depend on $\Gamma(se)$ in the prequents.

$$e = ()$$
$$e = \mathit{const}$$
$$e = \textbf{this}$$
$$e = \mathsf{f}$$

Case: (T-Sub) does not conclude (**53.1**), $e = x$

In this case the last step must be an instance of (T-VAR). By (T-VAR) we know $\Gamma(x) = \tau$, $\Sigma = \emptyset$, $\Pi = \emptyset$. There are two cases: $se = x$ or $se \neq x$. We now perform case analysis.

$\underline{se = x}$: in this case $e\{exd/se\} = x\{exd/x\} = exd$. According to (**53.2**) we know $\Gamma, \mathcal{C} \vdash e\{exd/se\} : \tau'\backslash\Sigma', \Pi'$. When $se = x$, we also know $\Gamma(x) = \Gamma(se) = \tau'$. Earlier we have just proved $\Gamma(x) = \tau$. Hence $\tau' = \tau$. Rewriting the previous judgment we know $\Gamma, \mathcal{C} \vdash e\{exd/se\} : \tau\backslash\Sigma', \Pi'$. The conclusion holds since $\Sigma' \cup \Sigma \subseteq \Sigma' \cup \Sigma$ and $\Pi' \subseteq \Pi \cup \Pi'$.

$\underline{se \neq x}$: in this case by (T-Var) we know from (**53.1**) the $(se : \tau')$ entry does not affect the typing of x. Thus $\Gamma, \mathcal{C} \vdash x : \tau\backslash\Sigma, \Pi$ We also know $e\{exd/se\} = x$. Thus $\Gamma, \mathcal{C} \vdash e\{exd/se\} : \tau\backslash\Sigma, \Pi$. The conclusion holds since $\Sigma \cup \Sigma \subseteq \Sigma' \cup \Sigma$ and $\Pi \subseteq \Pi \cup \Pi'$.

Case: (T-Sub) does not conclude (**53.1**), $e =$ **current**

This case is almost identical to the case of $e = x$. The two subcases to be discussed are $se =$ **current** and $se \neq$ **current**.

Case: (T-Sub) does not conclude (**53.1**), $e = e^{\mathbf{X}}; e^{\mathbf{Y}}$

In this case the last step must be an instance of (T-Sequence). By (T-Sequence) we know

$$\Gamma \rhd (se : \tau'), \mathcal{C} \vdash e^{\mathbf{X}} : \tau^{\mathbf{X}}\backslash\Sigma^{\mathbf{X}}, \Pi^{\mathbf{X}} \quad \textbf{(53.6)}$$
$$\Gamma \rhd (se : \tau'), \mathcal{C} \vdash e^{\mathbf{Y}} : \tau^{\mathbf{Y}}\backslash\Sigma^{\mathbf{Y}}, \Pi^{\mathbf{Y}} \quad \textbf{(53.7)}$$
$$\Sigma = \Sigma^{\mathbf{X}} \cup \Sigma^{\mathbf{Y}} \quad \textbf{(53.8)}$$
$$\Pi = \Pi^{\mathbf{X}} \cup \Pi^{\mathbf{Y}} \quad \textbf{(53.9)}$$
$$\tau = \tau^{\mathbf{Y}} \quad \textbf{(53.10)}$$

With (**53.6**), (**53.2**), and induction hypothesis, we know

$$\Gamma, \mathcal{C} \vdash e^{\mathbf{X}}\{exd/se\} : \tau^{\mathbf{X}} \backslash \Sigma_0^{\mathbf{X}}, \Pi_0^{\mathbf{X}} \quad \textbf{(53.11)}$$
$$\Sigma_0^{\mathbf{X}} \subseteq \Sigma^{\mathbf{X}} \cup \Sigma' \quad \textbf{(53.12)}$$
$$\Pi_0^{\mathbf{X}} \subseteq \Pi^{\mathbf{X}} \cup \Pi' \quad \textbf{(53.13)}$$

Similarly, with (**53.7**), (**53.2**), and induction hypothesis, we know

$$\Gamma, \mathcal{C} \vdash e^{\mathbf{Y}}\{exd/se\} : \tau^{\mathbf{Y}} \backslash \Sigma_0^{\mathbf{Y}}, \Pi_0^{\mathbf{Y}} \quad \textbf{(53.14)}$$
$$\Sigma_0^{\mathbf{Y}} \subseteq \Sigma^{\mathbf{Y}} \cup \Sigma' \quad \textbf{(53.15)}$$
$$\Pi_0^{\mathbf{Y}} \subseteq \Pi^{\mathbf{Y}} \cup \Pi' \quad \textbf{(53.16)}$$

With (**53.11**), (**53.14**), and (T-Sequence), we know $\Gamma, \mathcal{C} \vdash e^{\mathbf{X}}\{exd/se\}; e^{\mathbf{Y}}\{exd/se\} : \tau^{\mathbf{Y}} \backslash \Sigma_0^{\mathbf{X}} \cup \Sigma_0^{\mathbf{Y}}, \Pi_0^{\mathbf{X}} \cup \Pi_0^{\mathbf{Y}}$. With (**53.10**) and the fact of $e\{exd/se\} = e^{\mathbf{X}}\{exd/se\}; e^{\mathbf{Y}}\{exd/se\}$, we know

$$\Gamma, \mathcal{C} \vdash e\{exd/se\} : \tau \backslash \Sigma_0^{\mathbf{X}} \cup \Sigma_0^{\mathbf{Y}}, \Pi_0^{\mathbf{X}} \cup \Pi_0^{\mathbf{Y}}$$

With (**53.8**), we know $\Sigma = \Sigma^{\mathbf{X}} \cup \Sigma^{\mathbf{Y}}$. By (**53.12**) and (**53.15**), it is obvious that $\Sigma_0^{\mathbf{X}} \cup \Sigma_0^{\mathbf{Y}} \subseteq \Sigma^{\mathbf{X}} \cup \Sigma^{\mathbf{Y}} \cup \Sigma' = \Sigma \cup \Sigma'$. With (**53.9**), we know $\Pi = \Pi^{\mathbf{X}} \cup \Pi^{\mathbf{Y}}$. By (**53.13**) and (**53.16**), it is obvious that $\Pi_0^{\mathbf{X}} \cup \Pi_0^{\mathbf{Y}} \subseteq \Pi^{\mathbf{X}} \cup \Pi^{\mathbf{Y}} \cup \Pi' = \Pi \cup \Pi'$.

Cases similar to the previous case where $e = e^{\mathbf{X}}; e^{\mathbf{Y}}$

The following cases can be proved in an almost identical way as $e = e^{\mathbf{X}}; e^{\mathbf{Y}}$. The key common things with them are inductions are always used, and their corresponding typing rules strictly accumulate Σ and Π from the pre-quents to the sequents

$$e = \textbf{connect}\ e^{\mathbf{Y}}\ \textbf{with}\ \mathsf{k}^{\mathbf{X}}\ \texttt{>>}\ \mathsf{k}^{\mathbf{Y}}$$
$$e = e^{\mathbf{X}}\texttt{->}\mathsf{m}(e^{\mathbf{Y}})$$
$$e = \texttt{::}\mathsf{m}(e^{\mathbf{Y}})$$
$$e = \mathsf{f}\texttt{:=}e^{\mathbf{Y}}$$

Case: (T-Sub) does not conclude (**53.1**), $e = \textbf{forall}(x : \mathsf{k})\{e^{\mathbf{Y}}\}$

In this case the last step must be an instance of (T-Forall). Hence by (T-Forall) we know

$$(\Gamma \rhd (se : \tau'))(\textbf{me}) = \mathsf{a}^{\mathbf{X}} \quad \textbf{(53.17)}$$
$$\mathcal{C}(\mathsf{a}^{\mathbf{X}}) = \overline{\forall \langle \alpha^{\mathbf{X}}; \beta^{\mathbf{X}} \rangle}.\ \langle \mathcal{K}^{\mathbf{X}}; \mathcal{M}^{\mathbf{X}}; \mathcal{F}^{\mathbf{X}} \rangle \quad \textbf{(53.18)}$$

$$\mathsf{k} \in \mathrm{Dom}(\mathcal{K}^{\mathbf{X}}) \tag{53.19}$$
$$\Gamma \rhd (se : \tau') \rhd (x : \mathsf{k}),\ \mathcal{C} \vdash e^{\mathbf{Y}} : \tau \backslash \Sigma^{\mathbf{Y}}, \Pi^{\mathbf{Y}} \tag{53.20}$$
$$\Sigma = \Sigma^{\mathbf{Y}} \tag{53.21}$$
$$\Pi = \Pi^{\mathbf{Y}} \tag{53.22}$$

With (**53.17**) it is obvious that

$$\Gamma(\mathbf{me}) = \mathsf{a}^{\mathbf{X}} \tag{53.23}$$

There are two cases: $se = x$ or $se \neq x$. We now perform case analysis.

$\underline{se \neq x}$: in this case we know $\Gamma \rhd (se : \tau') \rhd (x : \mathsf{k}) = \Gamma \rhd (x : \mathsf{k}) \rhd (se : \tau')$. With this, (**53.20**), (**53.2**), and induction hypothesis, we know

$$\Gamma \rhd (x : \mathsf{k}), \mathcal{C} \vdash e^{\mathbf{Y}}\{exd/se\} : \tau \backslash \Sigma_0^{\mathbf{Y}}, \Pi_0^{\mathbf{Y}} \tag{53.24}$$
$$\Sigma_0^{\mathbf{Y}} \subseteq \Sigma^{\mathbf{Y}} \cup \Sigma' \tag{53.25}$$
$$\Pi_0^{\mathbf{Y}} \subseteq \Pi^{\mathbf{Y}} \cup \Pi' \tag{53.26}$$

With (**53.23**), (**53.18**), (**53.24**), (**53.19**), and (T-Forall) we know $\Gamma, \mathcal{C} \vdash$ **forall**$(x : \mathsf{k})\{e^{\mathbf{Y}}\{exd/se\}\} : \tau \backslash \Sigma_0^{\mathbf{Y}}, \Pi_0^{\mathbf{Y}}$. With the fact that according to the definition of substitution we know the following holds when $se \neq x$: $e\{exd/se\} =$ **forall**$(x : \mathsf{k})\{e^{\mathbf{Y}}\{exd/se\}\}$. We thus know $\Gamma, \mathcal{C} \vdash e\{exd/se\} : \tau \backslash \Sigma_0^{\mathbf{Y}}, \Pi_0^{\mathbf{Y}}$. With (**53.22**) we know $\Sigma = \Sigma^{\mathbf{Y}}$. By (**53.25**) thus $\Sigma_0^{\mathbf{Y}} \subseteq \Sigma^{\mathbf{Y}} \cup \Sigma' = \Sigma \cup \Sigma'$. With (**53.22**) we know $\Pi = \Pi^{\mathbf{Y}}$. By (**53.26**) it is obvious that $\Pi_0^{\mathbf{Y}} \subseteq \Pi^{\mathbf{Y}} \cup \Pi' = \Pi \cup \Pi'$.

$\underline{se = x}$: in this case by the definition of $\rhd$ we know $\Gamma \rhd (se : \tau') \rhd (x : \mathsf{k}) = \Gamma \rhd (x : \mathsf{k})$. Thus (**53.20**) can be rewritten as $\Gamma \rhd (x : \mathsf{k}),\ \mathcal{C} \vdash e^{\mathbf{Y}} : \tau \backslash \Sigma^{\mathbf{Y}}, \Pi^{\mathbf{Y}}$. With this, and (**53.23**), (**53.18**), (**53.19**), and (T-Forall) we know $\Gamma, \mathcal{C} \vdash$ **forall**$(x : \mathsf{k})\{e^{\mathbf{Y}}\} : \tau \backslash \Sigma^{\mathbf{Y}}, \Pi^{\mathbf{Y}}$. According to the definition of substitution we know the following holds when $se = x$: $e\{exd/se\} =$ **forall**$(x : \mathsf{k})\{e^{\mathbf{Y}}\}$. Thus $\Gamma, \mathcal{C} \vdash e\{exd/se\} : \tau \backslash \Sigma^{\mathbf{Y}}, \Pi^{\mathbf{Y}}$. With (**53.21**) we know $\Sigma = \Sigma^{\mathbf{Y}}$. With (**53.22**) we know $\Pi = \Pi^{\mathbf{Y}}$. The conclusion holds because $\Sigma \subseteq \Sigma \cup \Sigma'$ and $\Pi \subseteq \Pi \cup \Pi'$ hold trivially. □

C.11 Properties of Values

Lemma 54. *If* $\Gamma, \mathcal{C} \vdash v : \tau \backslash \Sigma, \Pi$, *then* $\Pi = \emptyset$.

Proof. Trivial case analysis on v and induction on the last step of derivation leading to the judgment. All typing rule related to values have Π as an emptyset. See (T-Int), (T-Unit), (T-RID), (T-CID), (T-Null). (T-Sub) does not alter Π as well. □

Lemma 55 (Value Passing In the Direction that rel is Computed)**.** *Given*

$$\boldsymbol{\Delta} = \langle \mathcal{H};\, W;\, R \rangle$$

$$G, \mathcal{C} \vdash_{\texttt{wftI}} \boldsymbol{\Delta} \backslash \Sigma_{\texttt{wfti}} \tag{55.1}$$

$$\mathcal{H}(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; \mathcal{S}^{\mathbf{X}} \rangle$$

$$\mathcal{H}(o^{\mathbf{Y}}) = \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; \mathcal{S}^{\mathbf{Y}} \rangle$$

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash v : \tau^{\mathbf{X}} \backslash \Sigma^{\mathbf{X}}, \Pi^{\mathbf{X}} \tag{55.2}$$

$$rel(R, o^{\mathbf{Y}}, o^{\mathbf{X}}) = \rho^{\mathbf{YX}} = (\textbf{parent})^{\nu_a^{\mathbf{YX}}} (\textbf{child})^{\nu_b^{\mathbf{YX}}} \tag{55.3}$$

$$\rho^{\mathbf{XY}} = (\textbf{parent})^{\nu_b^{\mathbf{YX}}} (\textbf{child})^{\nu_a^{\mathbf{YX}}}$$

$$matchd(\tau^{\mathbf{X}}[\sigma^{\mathbf{X}}], \tau^{\mathbf{Y}}[\sigma^{\mathbf{Y}}], \rho^{\mathbf{XY}}) = \Sigma_{\texttt{match}} \tag{55.4}$$

$$TV(\tau^{\mathbf{Y}}) \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{Y}})) \tag{55.5}$$

$$TV(\tau^{\mathbf{X}}) \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{X}})) \tag{55.6}$$

then there exist some $\Sigma^{\mathbf{Y}}$ *and* $\Pi^{\mathbf{Y}}$ *such that*

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}} \end{array}\right], \mathcal{C} \vdash v : \tau^{\mathbf{Y}} \backslash \Sigma^{\mathbf{Y}}, \Pi^{\mathbf{Y}} \tag{55.GoalA}$$

$$\Sigma^{\mathbf{X}}[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{match}} \cup wfT(\tau^{\mathbf{Y}})[\sigma^{\mathbf{Y}}] \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \tag{55.GoalB}$$

$$\Pi^{\mathbf{Y}} = \emptyset \tag{55.GoalC}$$

Proof. Case analysis on v.

$\boxed{v = () \text{ or } const}$

We only prove the case where $v = ()$. The other case can be proved analogously. If $\tau^{\mathbf{X}} = \mathsf{k}$ for some k, then since there is no subtyping rules exist for connection handle types, we know the last step of deriviation leading to (**55.2**) must not be an instance of (T-Sub). Given the form of $v = ()$, it must be an instance of (T-Unit). According to this rule $\tau^{\mathbf{X}} = \textbf{unit}$. Contradiction. Thus $\tau^{\mathbf{X}} \neq \mathsf{k}$ for any k. By (**55.2**) and Lem. 19, we know there exists

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash () : \tau_0^{\mathbf{X}} \backslash \Sigma_0^{\mathbf{X}}, \Pi^{\mathbf{X}} \quad \textbf{(55.7)}$$

$$\emptyset \vdash \tau_0^{\mathbf{X}} <: \tau^{\mathbf{X}} \backslash \Sigma_{\mathtt{subv}} \quad \textbf{(55.8)}$$

and the last step of derivation leading to (**55.7**) is an instance of (T-Unit). By that rule we know $\tau_0^{\mathbf{X}} = \textbf{unit}$, $\Sigma_0^{\mathbf{X}} = \emptyset$, and $\Pi^{\mathbf{X}} = \emptyset$. The only subtyping rule that can be used to lead to (**55.8**) is (Sub-Unit), and therefore $\tau^{\mathbf{X}} = \textbf{unit}$, and $\Sigma_{\mathtt{subv}} = \emptyset$. By the definition of substitution, $\tau^{\mathbf{X}}[\sigma^{\mathbf{X}}] = \textbf{unit}[\sigma^{\mathbf{X}}] = \textbf{unit}$. Thus according to (**55.4**), we know $\tau^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] = \textbf{unit}$. By the definition of substitution, we know $\tau^{\mathbf{Y}} = \textbf{unit}$. (**55.GoalA**) holds trivially according to (T-Unit) and we know $\Sigma^{\mathbf{Y}} = \emptyset$ and $\Pi^{\mathbf{Y}} = \emptyset$. (**55.GoalB**) trivially holds because $\Sigma^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] = \emptyset$. (**55.GoalC**) also trivially holds.

$\boxed{v = \textbf{null}}$

By (T-Null), we know

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}}\end{array}\right], \mathcal{C} \vdash v : \bot \backslash \emptyset, \emptyset$$

By (Sub-Bottom) and (T-Sub), we know

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}}\end{array}\right], \mathcal{C} \vdash v : \tau^{\mathbf{Y}} \backslash \emptyset, \emptyset$$

It is obvious that (**55.GoalA**) and (**55.GoalB**) and (**55.GoalC**) hold.

$\boxed{v = c^{side}}$

This is not possible. Proof by contradiction. If a derivation tree leading to (**55.2**) existed, then either 1) $\tau^{\mathbf{X}} = \mathsf{k}$ for some k and the last step of the derviation is an instance of (T-CID); 2) $\tau^{\mathbf{X}} \neq \mathsf{k}$ for any k and according to (**55.2**) and Lem. 19, we know there would exist

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash c^{side} : \tau_0^{\mathbf{X}} \backslash \Sigma_0^{\mathbf{X}}, \Pi^{\mathbf{X}} \quad \textbf{(55.9)}$$

$$\emptyset \vdash \tau_0^{\mathbf{X}} <: \tau^{\mathbf{X}} \backslash \Sigma_{\mathtt{subv}} \quad \textbf{(55.10)}$$

and the last step of derivation leading to (**55.9**) is (T-CID). This is not possible because according to that rule

$\tau_0^{\mathbf{X}}$ would be a connection handle type, and there does not exist a subtyping rule for connection handle type so that (**55.10**) cannot be obtained. If 1) is the case, then observe that $matchd(\tau^{\mathbf{X}}[\sigma^{\mathbf{X}}], \tau^{\mathbf{Y}}[\sigma^{\mathbf{Y}}], \rho^{\mathbf{XY}})$ is not defined by the definition of $matchd$. Contradiction with (**55.4**).

$\boxed{v = o^{\mathbf{Z}}}$

Let $\tau^{\mathbf{X}} = \mu \mathsf{a}^{\mathbf{X}}.\mathcal{K}^{\mathbf{X}}\ @\rho^{\mathbf{X}}$ where $\rho^{\mathbf{X}} = (\textbf{parent})^{\nu_a^{\mathbf{X}}}(\textbf{child})^{\nu_b^{\mathbf{X}}}$. Also let $\tau^{\mathbf{Y}} = \mu \mathsf{a}^{\mathbf{Y}}.\mathcal{K}^{\mathbf{Y}}\ @\rho^{\mathbf{Y}}$ where $\rho^{\mathbf{Y}} = (\textbf{parent})^{\nu_a^{\mathbf{Y}}}(\textbf{child})^{\nu_b^{\mathbf{Y}}}$. In this case if $\tau^{\mathbf{X}} = \mathsf{k}$ for some k, then since there does not exist a subtyping rule for connection handle types, we know the last step of deriviation leading to (**55.2**) must not be an instance of (T-Sub). Given the form of $v = o^{\mathbf{Z}}$, it must be an instance of (T-RID). According to this rule $\tau^{\mathbf{X}}$ is an objectage type. Contradiction. Thus $\tau^{\mathbf{X}} \neq \mathsf{k}$ for any k. By (**55.2**) and Lem. 19, we know there exists

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash o^{\mathbf{Z}} : \tau_0^{\mathbf{X}} \backslash \Sigma_0^{\mathbf{X}}, \Pi^{\mathbf{X}} \tag{55.11}$$

$$\emptyset \vdash \tau_0^{\mathbf{X}} <: \tau^{\mathbf{X}} \backslash \Sigma_{\mathtt{subv}} \tag{55.12}$$

$$\Sigma^{\mathbf{X}} \xrightarrow{\mathtt{algebra}} \Sigma_0^{\mathbf{X}} \cup \Sigma_{\mathtt{subv}} \tag{55.13}$$

and the last step of derivation leading to (**55.11**) is an instance of (T-RID). By that rule we know

$$\mathcal{H}(o^{\mathbf{Z}}) = \langle \mathsf{a}^{\mathbf{Z}}; \sigma^{\mathbf{Z}}; \mathcal{S}^{\mathbf{Z}} \rangle \tag{55.14}$$

$$\mathcal{C}(\mathsf{a}^{\mathbf{Z}}) = \overline{\forall \langle \alpha; \beta \rangle} \langle \mathcal{K}^{\mathbf{Z}}; \mathcal{M}^{\mathbf{Z}}; \mathcal{F}^{\mathbf{Z}} \rangle \tag{55.15}$$

$$rel(R, o^{\mathbf{Z}}, o^{\mathbf{X}}) = \rho^{\mathbf{ZX}} \tag{55.16}$$

$$\rho^{\mathbf{ZX}} = (\textbf{parent})^{\nu_a^{\mathbf{ZX}}}(\textbf{child})^{\nu_b^{\mathbf{ZX}}}$$

$$\rho_0^{\mathbf{X}} = (\textbf{parent})^{\nu_{a0}^{\mathbf{X}}}(\textbf{child})^{\nu_{b0}^{\mathbf{X}}}$$

$$\tau_0^{\mathbf{X}} = \mu \mathsf{a}^{\mathbf{Z}}.\mathcal{K}^{\mathbf{Z}}[\sigma^{\mathbf{Z}}]\ @\rho_0^{\mathbf{X}} \tag{55.17}$$

$$\Sigma_0^{\mathbf{X}} = subPed(\rho^{\mathbf{ZX}}, \rho_0^{\mathbf{X}}) \cup \{\nu_{a0}^{\mathbf{X}} \geq_{\mathtt{s}} 0\} \tag{55.18}$$

$$\Pi^{\mathbf{X}} = \emptyset \tag{55.19}$$

According to (**55.1**), (WFT-InstatnceConfig), (WFT-Heap), we know $TV(\mathcal{C}(\mathsf{a}^{\mathbf{X}})) = TV(\mathrm{Dom}(\sigma^{\mathbf{X}}))$. By assumption (**55.6**), we know $TV(\tau^{\mathbf{X}}) \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{X}}))$. Hence

$$TV(\tau^{\mathbf{X}}) \subseteq TV(\mathrm{Dom}(\sigma^{\mathbf{X}})) \tag{55.20}$$

and similarly, by (**55.5**), we know

$$TV(\tau^{\mathbf{Y}}) \subseteq TV(\mathrm{Dom}(\sigma^{\mathbf{Y}})) \tag{55.21}$$

From (**55.1**), (WFT-InstatnceConfig), (WFT-Heap), we know $TV(\mathcal{C}(\mathsf{a}^{\mathbf{Z}})) = TV(\mathrm{Dom}(\sigma^{\mathbf{Z}}))$. By definition of $\mathcal{K}^{\mathbf{Z}}$, we know $TV(\mathcal{K}^{\mathbf{Z}}) \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{Z}}))$. Thus $TV(\mathcal{K}^{\mathbf{Z}}) \subseteq TV(\mathrm{Dom}(\sigma^{\mathbf{Z}}))$. Thus by the definition of substitution, $TV(\mathcal{K}^{\mathbf{Z}}[\sigma^{\mathbf{Z}}]) \subseteq TV(\mathrm{Ran}(\sigma^{\mathbf{Z}}))$. By (**55.1**), (WFT-Config), (WFT-InstanceConfig), (WFT-Heap), we know $TV(\mathrm{Ran}(\sigma^{\mathbf{Z}})) \cap TV(\mathcal{C}) = \emptyset$. Yet by (**55.20**), we also know $TV(\mathrm{Dom}(\sigma^{\mathbf{X}})) = TV(\mathcal{C}(\mathsf{a}^{\mathbf{X}})) \subseteq TV(\mathcal{C})$. Therefore $TV(\mathcal{K}^{\mathbf{Z}}[\sigma^{\mathbf{Z}}]) \cap TV(\mathrm{Dom}(\sigma^{\mathbf{X}})) = \emptyset$. Hence

$$\mathcal{K}^{\mathbf{Z}}[\sigma^{\mathbf{Z}}][\sigma^{\mathbf{X}}] = \mathcal{K}^{\mathbf{Z}}[\sigma^{\mathbf{Z}}] \tag{55.22}$$

With (**55.12**) and Lem. 21, we know $\emptyset \vdash \tau_0^{\mathbf{X}}[\sigma^{\mathbf{X}}] <: \tau^{\mathbf{X}}[\sigma^{\mathbf{X}}] \backslash \Sigma_{\mathtt{subv}}[\sigma^{\mathbf{X}}]$. By Lem. 28, we know

$$\emptyset \vdash \mathit{convert}(\tau_0^{\mathbf{X}}[\sigma^{\mathbf{X}}], \rho^{\mathbf{XY}}) <: \mathit{convert}(\tau^{\mathbf{X}}[\sigma^{\mathbf{X}}], \rho^{\mathbf{XY}}) \backslash \Sigma_{\mathtt{subv0}} \tag{55.23}$$

$$\Sigma_{\mathtt{subv0}} \equiv \Sigma_{\mathtt{subv}}[\sigma^{\mathbf{X}}] \tag{55.24}$$

From (**55.4**) and the definition of matchd, we know

$$\emptyset \vdash \mathit{convert}(\tau^{\mathbf{X}}[\sigma^{\mathbf{X}}], \rho^{\mathbf{XY}}) <: \tau^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \backslash \Sigma_{\mathtt{sub}} \tag{55.25}$$

and $\Sigma_{\mathtt{sub}} \subseteq \Sigma_{\mathtt{match}}$ and

$$\{\nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] + \nu_b^{\mathbf{YX}} - \nu_a^{\mathbf{YX}} - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] =_{\mathtt{s}} \nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] - \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]\} \cup \{0 \leq_{\mathtt{s}} \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \leq_{\mathtt{s}} \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \leq_{\mathtt{s}} 1\} \subseteq \Sigma_{\mathtt{sub}} \tag{55.26}$$

By (**55.23**), (**55.25**), and Lem. 18 (subtyping transitivity), we know

$$\emptyset \vdash \mathit{convert}(\tau_0^{\mathbf{X}}[\sigma^{\mathbf{X}}], \rho^{\mathbf{XY}}) <: \tau^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \backslash \Sigma_{\mathtt{sub0}} \tag{55.27}$$

$$\Sigma_{\mathtt{sub}} \cup \Sigma_{\mathtt{subv0}} \xrightarrow{\mathtt{algebra}} \Sigma_{\mathtt{sub0}} \tag{55.28}$$

By the definition of $\mathit{convert}$, we know it must hold that $\mathit{convert}(\tau_0^{\mathbf{X}}[\sigma^{\mathbf{X}}], \rho^{\mathbf{XY}}) = \mu\mathsf{a}^{\mathbf{Z}}.\mathcal{K}^{\mathbf{Z}}[\sigma^{\mathbf{Z}}][\sigma^{\mathbf{X}}]\ @\rho^{\mathbf{X}'}$ for some $\rho^{\mathbf{X}'}$ we do not care here. Earlier we know (**55.22**), thus the previous equation can be rewritten as $\mathit{convert}(\tau_0^{\mathbf{X}}[\sigma^{\mathbf{X}}], \rho^{\mathbf{XY}}) = \mu\mathsf{a}^{\mathbf{Z}}.\mathcal{K}^{\mathbf{Z}}[\sigma^{\mathbf{Z}}]\ @\rho^{\mathbf{X}'}$. By (**55.27**) and (Sub-Non-Recursive),

$$\begin{aligned}
\mathrm{Dom}(\mathcal{K}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]) &= \{\mathsf{k}_1, \ldots, \mathsf{k}_n\} \\
\mathsf{a}_{\mathtt{new}}^{\mathbf{Z}}, \mathsf{a}_{\mathtt{new}}^{\mathbf{Y}} &\text{ fresh} \\
\mathcal{K}_{\mathtt{new}}^{\mathbf{Z}} &= \mathcal{K}^{\mathbf{Z}}[\sigma^{\mathbf{Z}}]\{\mathsf{a}_{\mathtt{new}}^{\mathbf{Z}}/\mathsf{a}^{\mathbf{Z}}\} \\
\mathcal{K}_{\mathtt{new}}^{\mathbf{Y}} &= \mathcal{K}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]\{\mathsf{a}_{\mathtt{new}}^{\mathbf{Y}}/\mathsf{a}^{\mathbf{Y}}\} \\
\forall i \in \{1, \ldots, n\}.\{\mathsf{a}_{\mathtt{new}}^{\mathbf{Z}} <: \mathsf{a}_{\mathtt{new}}^{\mathbf{Y}}\} &\vdash_{\mathtt{c}} \mathcal{K}_{\mathtt{new}}^{\mathbf{Z}}(\mathsf{k}_i) <: \mathcal{K}_{\mathtt{new}}^{\mathbf{Y}}(\mathsf{k}_i) \backslash \Sigma_i \\
\Sigma_{\mathtt{structure}} = \Sigma_1 \cup \ldots \Sigma_n &\subseteq \Sigma_{\mathtt{sub0}}
\end{aligned} \tag{55.29}$$

Next Task: Reaching (**55.GoalA**) and (**55.GoalC**)

We first show $rel(R, o^{\mathbf{Z}}, o^{\mathbf{Y}})$ must be defined. This is indeed true because: $relativize$ is a total function, and hence $relativize(\rho^{\mathbf{ZX}}, \rho^{\mathbf{XY}})$ is defined. According to Lem. 49, **(55.3)**, **(55.16)**, we know $rel(R, o^{\mathbf{Z}}, o^{\mathbf{Y}})$ is defined. As a result, let

$$rel(R, o^{\mathbf{Z}}, o^{\mathbf{Y}}) = \rho^{\mathbf{ZY}} = (\textbf{parent})^{\nu_a^{\mathbf{ZY}}} (\textbf{child})^{\nu_b^{\mathbf{ZY}}} \tag{55.30}$$

By (T-RID),and (**55.14**), (**55.15**), (**55.30**), we know

$$\left[\begin{array}{l} \textbf{iconf} : \mathbf{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}} \end{array}\right], \mathcal{C} \vdash o^{\mathbf{Z}} : \mu \mathsf{a}^{\mathbf{Z}}.\mathcal{K}^{\mathbf{Z}}[\sigma^{\mathbf{Z}}] \ @\rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \backslash subPed(\rho^{\mathbf{ZY}}, \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]) \cup \{\nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \geq_{\mathtt{s}} 0\}, \emptyset$$

By (T-Sub) and (**55.29**),

$$\left[\begin{array}{l} \textbf{iconf} : \mathbf{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}} \end{array}\right], \mathcal{C} \vdash o^{\mathbf{Z}} : \mu \mathsf{a}^{\mathbf{Y}}.\mathcal{K}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \ @\rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \backslash \Sigma_1^{\mathbf{ZY}}, \emptyset$$

$$\Sigma_1^{\mathbf{ZY}} = subPed(\rho^{\mathbf{ZY}}, \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]) \cup \{\nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \geq_{\mathtt{s}} 0\} \cup \Sigma_{\mathtt{structure}} \cup subPed(\rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}], \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}])$$

which is precisely

$$\left[\begin{array}{l} \textbf{iconf} : \mathbf{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}} \end{array}\right], \mathcal{C} \vdash o^{\mathbf{Z}} : \tau^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \backslash \Sigma_1^{\mathbf{ZY}}, \emptyset$$

From (**55.1**) and Lem. 10, we have $TV(\mathcal{C}(\mathsf{a}^{\mathbf{Y}})) = TV(\mathrm{Dom}(\sigma^{\mathbf{Y}}))$. By assumption (**55.5**), we know $TV(\tau^{\mathbf{Y}}) \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{Y}}))$. Hence $TV(\tau^{\mathbf{Y}}) \subseteq TV(\mathrm{Dom}(\sigma^{\mathbf{Y}}))$. Thus the pre-condition of Lem. 20 holds. Therefore we know

$$\begin{array}{c} \emptyset \vdash \tau^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] <: \tau^{\mathbf{Y}} \backslash \Sigma_{\mathtt{inst}} \\ wfT(\tau^{\mathbf{Y}})[\sigma^{\mathbf{Y}}] \xrightarrow{\mathtt{algebra}} \Sigma_{\mathtt{inst}}[\sigma^{\mathbf{Y}}] \end{array} \tag{55.31}$$

By (T-Sub)

$$\left[\begin{array}{l} \textbf{iconf} : \mathbf{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}} \end{array}\right], \mathcal{C} \vdash o^{\mathbf{Z}} : \tau^{\mathbf{Y}} \backslash \Sigma_1^{\mathbf{ZY}} \cup \Sigma_{\mathtt{inst}}, \emptyset$$

This is precisely (**55.GoalA**), and we have

$$\Sigma^{\mathbf{Y}} = subPed(\rho^{\mathbf{ZY}}, \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]) \cup \{\nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \geq_{\mathtt{s}} 0\} \cup \Sigma_{\mathtt{structure}} \cup subPed(\rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}], \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]) \cup \Sigma_{\mathtt{inst}}$$
$$\Pi^{\mathbf{Y}} = \emptyset$$

The second equation also trivially leads to (**55.GoalC**).

Next Task: Reaching (**55.GoalB**)

We first prove several facts that will be used for later proof.

Fast Fact 55.A: how is $\tau^{\mathbf{X}}[\sigma^{\mathbf{X}}]$ against substitution $[\sigma^{\mathbf{Y}}]$? From (**55.20**) and by the definition of substitution, we know $TV(\tau^{\mathbf{X}}[\sigma^{\mathbf{X}}]) \subseteq TV(\mathrm{Ran}(\sigma^{\mathbf{X}}))$. By (**55.1**), (WFT-Config), (WFT-InstanceConfig), (WFT-Heap), we know $TV(\mathrm{Ran}(\sigma^{\mathbf{X}})) \cap TV(\mathcal{C}) = \emptyset$. Yet by (**55.21**), we know $TV(\mathrm{Dom}(\sigma^{\mathbf{Y}})) = TV(\mathcal{C}(\mathsf{a}^{\mathbf{Y}})) \subseteq TV(\mathcal{C})$. Therefore $TV(\tau^{\mathbf{X}}[\sigma^{\mathbf{X}}]) \cap TV(\mathrm{Dom}(\sigma^{\mathbf{Y}})) = \emptyset$. As a result $\tau^{\mathbf{X}}[\sigma^{\mathbf{X}}][\sigma^{\mathbf{Y}}] = \tau^{\mathbf{X}}[\sigma^{\mathbf{X}}]$.

Fast Fact 55.B: how are pedigree expressions computed by the rel function against substitution $[\sigma^{\mathbf{X}}]$ and $[\sigma^{\mathbf{Y}}]$? By (**55.1**), and Lem. 11, we know it must hold that $TV(\mathrm{Dom}(\sigma^{\mathbf{X}})) \cap TV(R) = \emptyset$. By Lem. 46, we know $TV(\nu_a^{\mathbf{ZX}}) \cup TV(\nu_b^{\mathbf{ZX}})) \subseteq TV(R)$. Hence $\nu_a^{\mathbf{ZX}}[\sigma^{\mathbf{X}}] = \nu_a^{\mathbf{ZX}}$, $\nu_b^{\mathbf{ZX}}[\sigma^{\mathbf{X}}] = \nu_b^{\mathbf{ZX}}$, and $\rho^{\mathbf{ZX}}[\sigma^{\mathbf{X}}] = \rho^{\mathbf{ZX}}$. For similar reasons, we can also easily conclude $\nu_a^{\mathbf{ZY}}[\sigma^{\mathbf{Y}}] = \nu_a^{\mathbf{ZY}}$, $\nu_b^{\mathbf{ZY}}[\sigma^{\mathbf{Y}}] = \nu_b^{\mathbf{ZY}}$, and $\rho^{\mathbf{ZY}}[\sigma^{\mathbf{Y}}] = \rho^{\mathbf{ZY}}$. Again for similar reasons, we can conclude $\nu_a^{\mathbf{YX}}[\sigma^{\mathbf{Y}}] = \nu_a^{\mathbf{YX}}$, $\nu_b^{\mathbf{YX}}[\sigma^{\mathbf{Y}}] = \nu_b^{\mathbf{YX}}$.

Fast Fact 55.C: how is $\Sigma_{\mathtt{structure}}$ against substitution $[\sigma^{\mathbf{Y}}]$? By definition (**55.29**) and the subtyping rules we know $TV(\Sigma_{\mathtt{structure}}) = TV(\mathcal{K}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]) \cup TV(\mathcal{K}^{\mathbf{Z}}[\sigma^{\mathbf{Z}}])$. From (**55.1**), (WFT-InstatnceConfig), (WFT-Heap), we know $TV(\mathcal{C}(\mathsf{a}^{\mathbf{Y}})) = TV(\mathrm{Dom}(\sigma^{\mathbf{Y}}))$. By definition of $\mathcal{K}^{\mathbf{Y}}$, we know $TV(\mathcal{K}^{\mathbf{Y}}) \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{Y}}))$. Thus $TV(\mathcal{K}^{\mathbf{Y}}) \subseteq TV(\mathrm{Dom}(\sigma^{\mathbf{Y}}))$. By the definition of substitution, $TV(\mathcal{K}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]) \subseteq TV(\mathrm{Ran}(\sigma^{\mathbf{Y}}))$. By (**55.1**), (WFT-Config), (WFT-InstanceConfig), (WFT-Heap), we know $TV(\mathrm{Ran}(\sigma^{\mathbf{Y}})) \cap TV(\mathcal{C}) = \emptyset$. Yet by (**55.21**), we also know $TV(\mathrm{Dom}(\sigma^{\mathbf{Y}})) = TV(\mathcal{C}(\mathsf{a}^{\mathbf{Y}})) \subseteq TV(\mathcal{C})$. Therefore $TV(\mathcal{K}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]) \cap TV(\mathrm{Dom}(\sigma^{\mathbf{Y}})) = \emptyset$. Similarly, we know $TV(\mathcal{K}^{\mathbf{Z}}[\sigma^{\mathbf{Z}}]) \cap TV(\mathrm{Dom}(\sigma^{\mathbf{Y}})) = \emptyset$. All together, we know $\Sigma_{\mathtt{structure}}[\sigma^{\mathbf{Y}}] = \Sigma_{\mathtt{structure}}$ by the definition of substitution.

Fast Fact 55.D: how are $\rho^{\mathbf{X}}$ and $\rho^{\mathbf{ZX}}$ related? We now prove a subytyping relationship between the two:

$\Sigma^{\mathbf{X}}$

By (**55.13**), (ICS-relaxation)

$$\xrightarrow{\texttt{algebra}} \Sigma_{\texttt{subv}} \cup \Sigma_0^{\mathbf{X}}$$

By (**55.18**), (**55.12**), (ICS-relaxation)

$$\xrightarrow{\texttt{algebra}} subPed(\rho^{\mathbf{ZX}}, \rho_0^{\mathbf{X}}) \cup subPed(\rho_0^{\mathbf{X}}, \rho^{\mathbf{X}})$$

By definition of $subPed$

$$\xrightarrow{\texttt{algebra}} subPed(\rho^{\mathbf{ZX}}, \rho^{\mathbf{X}})$$

With these facts, we now turn to the proof (**55.GoalB**). By definition

$$relativize(\rho^{\mathbf{ZX}}, \rho^{\mathbf{XY}}) = (\textbf{parent})^{\nu_a^{\mathbf{ZX}} + \nu_b^{\mathbf{YX}} - \nu_a^{\mathbf{YX}}}(\textbf{child})^{\nu_b^{\mathbf{ZX}}}, \Sigma_{\texttt{relative}}$$

With this, by Lem. 49, (**55.3**), (**55.16**), we know

$$\nu_a^{\mathbf{ZY}} \equiv \nu_a^{\mathbf{ZX}} + \nu_b^{\mathbf{YX}} - \nu_a^{\mathbf{YX}} \tag{55.32}$$

$$\nu_b^{\mathbf{ZY}} \equiv \nu_b^{\mathbf{ZX}} \tag{55.33}$$

(**55.GoalB**) holds because of three factors. Firstly,

$$\Sigma^{\mathbf{X}}[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{match}}$$

By definition of substitution, **Fast Fact 55.D**

$$\xrightarrow{\texttt{algebra}} subPed(\rho^{\mathbf{ZX}}, \rho^{\mathbf{X}})[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{match}}$$

By (**55.26**), (ICS-relaxation)

$$\xrightarrow{\texttt{algebra}} subPed(\rho^{\mathbf{ZX}}, \rho^{\mathbf{X}})[\sigma^{\mathbf{X}}] \cup \left\{ \begin{array}{l} \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] + \nu_b^{\mathbf{YX}} - \nu_a^{\mathbf{YX}} - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] =_{\mathtt{s}} \nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] - \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \\ 0 \leq_{\mathtt{s}} \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \leq_{\mathtt{s}} \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \leq_{\mathtt{s}} 1 \end{array} \right\}$$

By definition of substitution, definition of $subPed$, **Fast Fact 55.B**

$$= \left\{ \begin{array}{l} \nu_a^{\mathbf{ZX}} - \nu_b^{\mathbf{ZX}} =_{\mathtt{s}} \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \\ 0 \leq_{\mathtt{s}} \nu_b^{\mathbf{ZX}} \leq_{\mathtt{s}} \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \leq_{\mathtt{s}} 1 \\ \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] + \nu_b^{\mathbf{YX}} - \nu_a^{\mathbf{YX}} - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] =_{\mathtt{s}} \nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] - \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \\ 0 \leq_{\mathtt{s}} \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \leq_{\mathtt{s}} \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \leq_{\mathtt{s}} 1 \end{array} \right\}$$

By (ICS-leq)

$$\xrightarrow{\texttt{algebra}} \left\{ \begin{array}{l} \nu_a^{\mathbf{ZX}} - \nu_b^{\mathbf{ZX}} =_{\mathtt{s}} \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \\ \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] + \nu_b^{\mathbf{YX}} - \nu_a^{\mathbf{YX}} - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] =_{\mathtt{s}} \nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] - \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \\ 0 \leq_{\mathtt{s}} \nu_b^{\mathbf{ZX}} \leq_{\mathtt{s}} \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \leq_{\mathtt{s}} 1 \end{array} \right\}$$

By (**55.32**), (**55.33**), (ECS-tautology)

$$\equiv \left\{ \begin{array}{l} \nu_a^{\mathbf{ZX}} - \nu_b^{\mathbf{ZX}} =_{\mathtt{s}} \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \\ \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] + \nu_b^{\mathbf{YX}} - \nu_a^{\mathbf{YX}} - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] =_{\mathtt{s}} \nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] - \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \\ 0 \leq_{\mathtt{s}} \nu_b^{\mathbf{ZX}} \leq_{\mathtt{s}} \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \leq_{\mathtt{s}} 1 \\ \nu_a^{\mathbf{ZY}} =_{\mathtt{s}} \nu_a^{\mathbf{ZX}} + \nu_b^{\mathbf{YX}} - \nu_a^{\mathbf{YX}} \\ \nu_b^{\mathbf{ZY}} =_{\mathtt{s}} \nu_b^{\mathbf{ZX}} \end{array} \right\}$$

By (ICS-equiv), (ICS-leq)

$$\xrightarrow{\texttt{algebra}} \{\nu_a^{\mathbf{ZY}} - \nu_b^{\mathbf{ZY}} =_{\mathtt{s}} \nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] - \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}], 0 \leq_{\mathtt{s}} \nu_b^{\mathbf{ZY}} \leq_{\mathtt{s}} \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \leq_{\mathtt{s}} 1\}$$

By definition of $subPed$

$$\equiv subPed(\rho^{\mathbf{ZY}}, \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}])$$

By **Fast Fact 55.B**, and the trivial fact that $\rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}][\sigma^{\mathbf{Y}}] = \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]$

$$= subPed(\rho^{\mathbf{ZY}}, \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}])[\sigma^{\mathbf{Y}}]$$

Secondly,

$$\begin{aligned}
&\Sigma^{\mathbf{X}}[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{match}} \\
&\quad \text{By } \textbf{(55.13)} \\
&\xrightarrow{\texttt{algebra}} \Sigma_{\texttt{subv}}[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{match}} \\
&\quad \text{By } \textbf{(55.29)}, \textbf{(55.24)}, \textbf{(55.28)}, \text{(ICS-relaxation)} \\
&\xrightarrow{\texttt{algebra}} \Sigma_{\texttt{structure}} \cup \Sigma_{\texttt{match}} \\
&\quad \text{By } \textbf{(55.26)} \\
&\xrightarrow{\texttt{algebra}} \Sigma_{\texttt{structure}} \cup \{0 \leq_{\texttt{s}} \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \leq_{\texttt{s}} 1\} \\
&\quad \text{By definition of } subPed, \text{ Lem. 15} \\
&\equiv \Sigma_{\texttt{structure}} \cup subPed(\rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}], \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]) \\
&\quad \text{By } \textbf{Fast Fact 55.C}, \text{ and the trivial fact that } \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}][\sigma^{\mathbf{Y}}] = \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \\
&= (\Sigma_{\texttt{structure}} \cup subPed(\rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}], \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]))[\sigma^{\mathbf{Y}}]
\end{aligned}$$

Thirdly,

$$\begin{aligned}
&wfT(\tau^{\mathbf{Y}})[\sigma^{\mathbf{Y}}] \\
&\quad \text{By definition of } wftT \\
&\xrightarrow{\texttt{algebra}} wfT(\tau^{\mathbf{Y}})[\sigma^{\mathbf{Y}}] \cup \{\nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \geq_{\texttt{s}} 0\} \\
&\quad \text{By } \textbf{(55.31)} \\
&\xrightarrow{\texttt{algebra}} \Sigma_{\texttt{inst}}[\sigma^{\mathbf{Y}}] \cup \{\nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \geq_{\texttt{s}} 0\}
\end{aligned}$$

□

Lemma 56 (Value Passing Opposite to the Direction that rel is Computed)**.** *Given*

$$\boldsymbol{\Delta} = \langle \mathcal{H}; W; R \rangle$$

$$G, \mathcal{C} \vdash_{\texttt{wftI}} \boldsymbol{\Delta} \backslash \Sigma_{\texttt{wfti}} \quad \textbf{(56.1)}$$

$$\mathcal{H}(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; \mathcal{S}^{\mathbf{X}} \rangle$$

$$\mathcal{H}(o^{\mathbf{Y}}) = \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; \mathcal{S}^{\mathbf{Y}} \rangle$$

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash v : \tau^{\mathbf{X}} \backslash \Sigma^{\mathbf{X}}, \Pi^{\mathbf{X}} \quad \textbf{(56.2)}$$

$$rel(R, o^{\mathbf{X}}, o^{\mathbf{Y}}) = \rho^{\mathbf{XY}} = (\textbf{parent})^{\nu_a^{\mathbf{XY}}} (\textbf{child})^{\nu_b^{\mathbf{XY}}} \quad \textbf{(56.3)}$$

$$matchd(\tau^{\mathbf{X}}[\sigma^{\mathbf{X}}], \tau^{\mathbf{Y}}[\sigma^{\mathbf{Y}}], \rho^{\mathbf{XY}}) = \Sigma_{\texttt{match}} \quad \textbf{(56.4)}$$

$$TV(\tau^{\mathbf{Y}}) \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{Y}})) \quad \textbf{(56.5)}$$

$$TV(\tau^{\mathbf{X}}) \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{X}})) \quad \textbf{(56.6)}$$

then there exist some $\Sigma^{\mathbf{Y}}$ *and* $\Pi^{\mathbf{Y}}$ *such that*

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}}\end{array}\right], \mathcal{C} \vdash v : \tau^{\mathbf{Y}} \backslash \Sigma^{\mathbf{Y}}, \Pi^{\mathbf{Y}} \tag{56.GoalA}$$

$$\Sigma^{\mathbf{X}}[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{match}} \cup wfT(\tau^{\mathbf{Y}})[\sigma^{\mathbf{Y}}] \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \tag{56.GoalB}$$

$$\Pi^{\mathbf{Y}} = \emptyset \tag{56.GoalC}$$

Proof. Case analysis on v.

$\boxed{v = () \text{ or } const \text{ or } \textbf{null} \text{ or } c^{side}}$

Identical to the respective cases in the proof of Lem. 55.

$\boxed{v = o^{\mathbf{Z}}}$

This case is almost identical to that of Lem. 55 as well. Specifically, **Fast Fact 56.A**, **Fast Fact 56.C**, **Fast Fact 56.D** are identical to the counterparts in the proof of Lem. 55. For **Fast Fact 56.B**, it is also identical to its coutepart in Lem. 55 except we use the facts of $\nu_a^{\mathbf{XY}}[\sigma^{\mathbf{Y}}] = \nu_a^{\mathbf{XY}}$ and $\nu_b^{\mathbf{XY}}[\sigma^{\mathbf{Y}}] = \nu_b^{\mathbf{XY}}$. The real difference is how $\rho^{\mathbf{XY}}$ is defined. It affects $\Sigma_{\texttt{match}}$ and how $\rho^{\mathbf{ZY}}$ is computed. First, we know by (**56.4**)

$$\{\nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] + \nu_a^{\mathbf{XY}} - \nu_b^{\mathbf{XY}} - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] =_{\mathtt{s}} \nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] - \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]\} \cup \{0 \leq_{\mathtt{s}} \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \leq_{\mathtt{s}} \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \leq_{\mathtt{s}} 1\} \subseteq \Sigma_{\texttt{match}} \tag{56.7}$$

By definition

$$relativize(\rho^{\mathbf{ZX}}, \rho^{\mathbf{XY}}) = (\textbf{parent})^{\nu_a^{\mathbf{ZX}} + \nu_a^{\mathbf{XY}} - \nu_b^{\mathbf{XY}}} (\textbf{child})^{\nu_b^{\mathbf{ZX}}}, \Sigma_{\texttt{relative}}$$

With this, by Lem. 48, (**56.3**), (**55.16**), we know

$$\nu_a^{\mathbf{ZY}} \equiv \nu_a^{\mathbf{ZX}} + \nu_a^{\mathbf{XY}} - \nu_b^{\mathbf{XY}} \tag{56.8}$$

$$\nu_b^{\mathbf{ZY}} \equiv \nu_b^{\mathbf{ZX}} \tag{56.9}$$

and compared with Lem. 55, the only difference it makes is how $subPed(\rho^{\mathbf{ZY}}, \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}])[\sigma^{\mathbf{Y}}]$ is implied:

$\Sigma^{\mathbf{X}}[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{match}}$

By definition of substitution, **Fast Fact 56.D**

$\xrightarrow{\texttt{algebra}} subPed(\rho^{\mathbf{ZX}}, \rho^{\mathbf{X}})[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{match}}$

By (**56.7**), (ICS-relaxation)

$$\xrightarrow{\texttt{algebra}} subPed(\rho^{\mathbf{ZX}}, \rho^{\mathbf{X}})[\sigma^{\mathbf{X}}] \cup \left\{ \begin{array}{l} \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] + \nu_a^{\mathbf{XY}} - \nu_b^{\mathbf{XY}} - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] =_{\mathtt{s}} \nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] - \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \\ 0 \leq_{\mathtt{s}} \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \leq_{\mathtt{s}} \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \leq_{\mathtt{s}} 1 \end{array} \right\}$$

By definition of substitution, definition of $subPed$, **Fast Fact 56.B**

$$= \left\{ \begin{array}{l} \nu_a^{\mathbf{ZX}} - \nu_b^{\mathbf{ZX}} =_{\mathsf{s}} \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \\ 0 \leq_{\mathsf{s}} \nu_b^{\mathbf{ZX}} \leq_{\mathsf{s}} \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \leq_{\mathsf{s}} 1 \\ \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] + \nu_a^{\mathbf{XY}} - \nu_b^{\mathbf{XY}} - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] =_{\mathsf{s}} \nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] - \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \\ 0 \leq_{\mathsf{s}} \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \leq_{\mathsf{s}} \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \leq_{\mathsf{s}} 1 \end{array} \right\}$$

By (ICS-leq)

$$\xrightarrow{\texttt{algebra}} \left\{ \begin{array}{l} \nu_a^{\mathbf{ZX}} - \nu_b^{\mathbf{ZX}} =_{\mathsf{s}} \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \\ \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] + \nu_a^{\mathbf{XY}} - \nu_b^{\mathbf{XY}} - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] =_{\mathsf{s}} \nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] - \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \\ 0 \leq_{\mathsf{s}} \nu_b^{\mathbf{ZX}} \leq_{\mathsf{s}} \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \leq_{\mathsf{s}} 1 \end{array} \right\}$$

By (**56.8**), (**56.9**), (ECS-tautology)

$$\equiv \left\{ \begin{array}{l} \nu_a^{\mathbf{ZX}} - \nu_b^{\mathbf{ZX}} =_{\mathsf{s}} \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] \\ \nu_a^{\mathbf{X}}[\sigma^{\mathbf{X}}] + \nu_a^{\mathbf{XY}} - \nu_b^{\mathbf{XY}} - \nu_b^{\mathbf{X}}[\sigma^{\mathbf{X}}] =_{\mathsf{s}} \nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] - \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \\ 0 \leq_{\mathsf{s}} \nu_b^{\mathbf{ZX}} \leq_{\mathsf{s}} \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \leq_{\mathsf{s}} 1 \\ \nu_a^{\mathbf{ZY}} =_{\mathsf{s}} \nu_a^{\mathbf{ZX}} + \nu_a^{\mathbf{XY}} - \nu_b^{\mathbf{XY}} \\ \nu_b^{\mathbf{ZY}} =_{\mathsf{s}} \nu_b^{\mathbf{ZX}} \end{array} \right\}$$

By (ICS-equiv), (ICS-leq)

$$\xrightarrow{\texttt{algebra}} \{\nu_a^{\mathbf{ZY}} - \nu_b^{\mathbf{ZY}} =_{\mathsf{s}} \nu_a^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] - \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}], 0 \leq_{\mathsf{s}} \nu_b^{\mathbf{ZY}} \leq_{\mathsf{s}} \nu_b^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \leq_{\mathsf{s}} 1\}$$

By definition of $subPed$

$$\equiv subPed(\rho^{\mathbf{ZY}}, \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}])$$

By **Fast Fact 56.B**, and the trivial fact that $\rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}][\sigma^{\mathbf{Y}}] = \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]$

$$= subPed(\rho^{\mathbf{ZY}}, \rho^{\mathbf{Y}}[\sigma^{\mathbf{Y}}])[\sigma^{\mathbf{Y}}]$$

□

Lemma 57 (Type Preservation for Values Over Γ Change)**.** *The following judgments can lead to one another:*

- $\Gamma, \mathcal{C} \vdash v : \tau \backslash \Sigma, \Pi$.
- $\Gamma \rhd (\textbf{me} : \mathsf{a}), \mathcal{C} \vdash v : \tau \backslash \Sigma, \Pi$ *for any* a.
- $\Gamma \rhd (\textbf{current} : \mathsf{k}), \mathcal{C} \vdash v : \tau \backslash \Sigma, \Pi$ *for any* k.

Proof. Case analysis on typing rules for values. For all of them, *i.e.*, (T-Unit), (T-Int), (T-Null), (T-RID), (T-CID), none is dependent on $\Gamma(\textbf{me})$ or $\Gamma(\textbf{current})$. □

C.12 Properties of the Evaluation Context

Lemma 58 (Type Preservation for Evaluation Contexts Over Replacement and Configuration Strengthening)**.**

For some G, Γ, $\mathcal{C}$, and $\mathbf{\Delta}$, given

$$\Gamma, \mathcal{C} \vdash \mathbf{E}[\, \mathit{exd}^{\mathbf{X}} \,] : \tau \backslash \Sigma^{\mathbf{X}}, \Pi^{\mathbf{X}} \quad \textbf{(58.Cond1)}$$

$$\Gamma, \mathcal{C} \vdash \mathit{exd}^{\mathbf{X}} : \tau' \backslash \Sigma^{\mathbf{X}}_{\mathtt{exp}}, \Pi^{\mathbf{X}}_{\mathtt{exp}} \quad \textbf{(58.Cond2)}$$

$$\Gamma \rhd [\textbf{iconf} : \boldsymbol{\Delta}], \mathcal{C} \vdash \mathit{exd}^{\mathbf{Y}} : \tau' \backslash \Sigma^{\mathbf{Y}}_{\mathtt{exp}}, \Pi^{\mathbf{Y}}_{\mathtt{exp}} \quad \textbf{(58.Cond3)}$$

$$\Sigma^{\mathbf{X}}_{\mathtt{iconf}} \cup \Sigma^{\mathbf{X}}_{\mathtt{exp}}[\sigma] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{X}}_{\mathtt{exp}}) \xrightarrow{\mathtt{algebra}} \Sigma^{\mathbf{Y}}_{\mathtt{iconf}} \cup \Sigma^{\mathbf{Y}}_{\mathtt{exp}}[\sigma] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{Y}}_{\mathtt{exp}}) \quad \textbf{(58.Cond4)}$$

$$\Gamma[\textbf{iconf}] \preceq_{\mathtt{iconf}} \boldsymbol{\Delta} \quad \textbf{(58.Cond5)}$$

the last step of derivation leading to (**58.Cond2**) *is not an instance of* (T-Sub) **(58.Cond6)**

then

$$\Gamma \rhd [\textbf{iconf} : \boldsymbol{\Delta}], \mathcal{C} \vdash \mathbf{E}[\, \mathit{exd}^{\mathbf{Y}} \,] : \tau \backslash \Sigma^{\mathbf{Y}}, \Pi^{\mathbf{Y}} \textit{ for some } \Sigma^{\mathbf{Y}} \textit{ and } \Pi^{\mathbf{Y}}$$

$$\Sigma^{\mathbf{X}}_{\mathtt{iconf}} \cup \Sigma^{\mathbf{X}}[\sigma] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{X}}) \xrightarrow{\mathtt{algebra}} \Sigma^{\mathbf{Y}}_{\mathtt{iconf}} \cup \Sigma^{\mathbf{Y}}[\sigma] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{Y}})$$

Proof. Induction on the height of the derivation tree of (**58.Cond1**), and case analysis on the rule being applied on the last step and the form of $\mathbf{E}$.

Case: the last step is an instance of (T-Sub), regardless of the form of $\mathbf{E}$

By (T-Sub),

$$\Gamma, \mathcal{C} \vdash \mathbf{E}[\, \mathit{exd}^{\mathbf{X}} \,] : \tau_{\mathtt{sub}} \backslash \Sigma^{\mathbf{X}}_{\mathtt{sub}}, \Pi^{\mathbf{X}}$$
$$\emptyset \vdash \tau_{\mathtt{sub}} <: \tau \backslash \Sigma_{\mathtt{sub}}$$
$$\Sigma^{\mathbf{X}} = \Sigma^{\mathbf{X}}_{\mathtt{sub}} \cup \Sigma_{\mathtt{sub}}$$

By induction hypothesis, we know

$$\Gamma \rhd [\textbf{iconf} : \boldsymbol{\Delta}], \mathcal{C} \vdash \mathbf{E}[\, \mathit{exd}^{\mathbf{Y}} \,] : \tau_{\mathtt{sub}} \backslash \Sigma^{\mathbf{Y}}_{\mathtt{sub}}, \Pi^{\mathbf{Y}}_{\mathtt{sub}} \quad \textbf{(58.7)}$$

$$\Sigma^{\mathbf{X}}_{\mathtt{iconf}} \cup \Sigma^{\mathbf{X}}_{\mathtt{sub}}[\sigma] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{X}}) \xrightarrow{\mathtt{algebra}} \Sigma^{\mathbf{Y}}_{\mathtt{iconf}} \cup \Sigma^{\mathbf{Y}}_{\mathtt{sub}}[\sigma] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{Y}}_{\mathtt{sub}}) \quad \textbf{(58.8)}$$

By (T-Sub) and (**58.7**), we have

$$\Gamma \rhd [\textbf{iconf} : \boldsymbol{\Delta}], \mathcal{C} \vdash \mathbf{E}[\, \mathit{exd}^{\mathbf{Y}} \,] : \tau \backslash \Sigma^{\mathbf{Y}}_{\mathtt{sub}} \cup \Sigma_{\mathtt{sub}}, \Pi^{\mathbf{Y}}_{\mathtt{sub}}$$

Union $\Sigma_{\mathtt{sub}}[\sigma]$ on both side of (**58.8**), by (ICS-union), and the definition of substitution, we know

$$\Sigma^{\mathbf{X}}_{\mathtt{iconf}} \cup (\Sigma^{\mathbf{X}}_{\mathtt{sub}} \cup \Sigma_{\mathtt{sub}})[\sigma] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{X}}) \xrightarrow{\mathtt{algebra}} \Sigma^{\mathbf{Y}}_{\mathtt{iconf}} \cup (\Sigma^{\mathbf{Y}}_{\mathtt{sub}} \cup \Sigma_{\mathtt{sub}})[\sigma] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{Y}}_{\mathtt{sub}}) \quad \textbf{(58.9)}$$

Let $\Sigma^{\mathbf{Y}} = \Sigma^{\mathbf{Y}}_{\mathtt{sub}} \cup \Sigma_{\mathtt{sub}}$, $\Pi^{\mathbf{Y}} = \Pi^{\mathbf{Y}}_{\mathtt{sub}}$, and given the previous fact that $\Sigma^{\mathbf{X}} = \Sigma^{\mathbf{X}}_{\mathtt{sub}} \cup \Sigma_{\mathtt{sub}}$, we know

$$\Gamma \rhd [\textbf{iconf} : \boldsymbol{\Delta}], \mathcal{C} \vdash \mathbf{E}[\, \mathit{exd}^{\mathbf{Y}} \,] : \tau \backslash \Sigma^{\mathbf{Y}}, \Pi^{\mathbf{Y}}$$

$$\Sigma^{\mathbf{X}}_{\texttt{iconf}} \cup \Sigma^{\mathbf{X}}[\sigma] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{X}}) \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}}_{\texttt{iconf}} \cup \Sigma^{\mathbf{Y}}[\sigma] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{Y}})$$

Case: the last step is not an instance of (T-Sub), $\mathbf{E} = \bullet$

In this case, (**58.Cond1**) is

$$\Gamma, \mathcal{C} \vdash \mathit{exd}^{\mathbf{X}} : \tau \backslash \Sigma^{\mathbf{X}}, \Pi^{\mathbf{X}}$$

Since neither the last step of this derivation nor that of (**58.Cond2**) is an instance of (T-Sub), by Lem. 23, we know $\Sigma^{\mathbf{X}} = \Sigma^{\mathbf{X}}_{\texttt{exp}}$ and $\Pi^{\mathbf{X}} = \Pi^{\mathbf{X}}_{\texttt{exp}}$. The conclusion thus holds by (**58.Cond3**) and (**58.Cond4**).

Case: the last step is not an instance of (T-Sub), $\mathbf{E} = \mathbf{E}_0\,\texttt{->m}(e)$

According to the definition of evaluation context hole substitution, we know

$$\mathbf{E}[\mathit{exd}^{\mathbf{X}}] = \mathbf{E}_0[\mathit{exd}^{\mathbf{X}}]\,\texttt{->m}(e)$$

Thus the last non-subsumption step of the derivation tree leading to (**58.Cond1**) must be an instance of (T-HandleM), in which case we have

$$\Gamma, \mathcal{C} \vdash \mathbf{E}_0[\mathit{exd}^{\mathbf{X}}] : \tau_1 \backslash \Sigma^{\mathbf{X}}_1, \Pi^{\mathbf{X}}_1 \quad \textbf{(58.10)}$$
$$\Gamma, \mathcal{C} \vdash e : \tau_2 \backslash \Sigma_2, \Pi_2 \quad \textbf{(58.11)}$$
$$\Sigma_3 = \mathit{some\_constraints}(\Gamma(\textbf{me}), \Gamma(\textbf{current}), \tau_1, \tau_2, \tau, \mathcal{C}) \quad \textbf{(58.12)}$$
$$\Sigma^{\mathbf{X}} = \Sigma^{\mathbf{X}}_1 \cup \Sigma_2 \cup \Sigma_3$$
$$\Pi^{\mathbf{X}} = \Pi^{\mathbf{X}}_1 \cup \Pi_2$$

where $\mathit{some\_constraints}$ include all other constraints on that rule, which show the invariants (possibly) among $\Gamma(\textbf{me})$, $\Gamma(\textbf{current})$, τ_1, τ_2, τ, $\mathcal{C}$. We here do not give their specific forms because the following proof does not depend on it. This gives the proof some generality that can be similarly applied to other cases. Now let $\Gamma' = \Gamma \rhd [\textbf{iconf} : \mathbf{\Delta}]$. Given (**58.10**), (**58.Cond2**), (**58.Cond3**), (**58.Cond4**), (**58.Cond5**), by induction hypothesis, we know

$$\Gamma', \mathcal{C} \vdash \mathbf{E}_0[\mathit{exd}^{\mathbf{Y}}] : \tau_1 \backslash \Sigma^{\mathbf{Y}}_1, \Pi^{\mathbf{Y}}_1 \quad \textbf{(58.13)}$$
$$\Sigma^{\mathbf{X}}_{\texttt{iconf}} \cup \Sigma^{\mathbf{X}}_1[\sigma] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{X}}_1) \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{Y}}_{\texttt{iconf}} \cup \Sigma^{\mathbf{Y}}_1[\sigma] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{Y}}_1) \quad \textbf{(58.14)}$$

Since $e \in \mathbb{EXP}$, by Lem. 61, we know its type judgment still holds even if the configuration is changed. Thus

with (**58.11**) we know

$$\Gamma', \mathcal{C} \vdash e : \tau_2 \backslash \Sigma_2, \Pi_2 \tag{58.15}$$

Obviously $\Gamma(\textbf{me}) = \Gamma'(\textbf{me})$ and $\Gamma(\textbf{current}) = \Gamma'(\textbf{current})$. Thus it follows that the following holds:

$$\begin{aligned}\Sigma_3 =& some\_constraints(\Gamma'(\textbf{me}), \Gamma'(\textbf{current}), \tau_1, \tau_2, \tau, \mathcal{C}) \\ =& some\_constraints(\Gamma(\textbf{me}), \Gamma(\textbf{current}), \tau_1, \tau_2, \tau, \mathcal{C})\end{aligned} \tag{58.16}$$

With (**58.13**), (**58.15**), (**58.16**) and by (T-HandleM) we can immediately prove

$$\begin{aligned}\Gamma', \mathcal{C} \vdash \mathbf{E}_0[exd^{\mathbf{Y}}]\texttt{->m}(e) : \tau \backslash \Sigma^{\mathbf{Y}}, \Pi^{\mathbf{Y}} \\ \Sigma^{\mathbf{Y}} = \Sigma_1^{\mathbf{Y}} \cup \Sigma_2 \cup \Sigma_3 \\ \Pi^{\mathbf{Y}} = \Pi_1^{\mathbf{Y}} \cup \Pi_2\end{aligned}$$

Thus,

$$\begin{aligned}& \Sigma_{\texttt{iconf}}^{\mathbf{X}} \cup \Sigma^{\mathbf{X}}[\sigma] \cup lazyCons(G, \Pi^{\mathbf{X}}) \\ =& \Sigma_{\texttt{iconf}}^{\mathbf{X}} \cup (\Sigma_1^{\mathbf{X}} \cup \Sigma_2 \cup \Sigma_3)[\sigma] \cup lazyCons(G, \Pi_1^{\mathbf{X}} \cup \Pi_2) \\ & \text{By definition of substitution and } lazyCons \text{ (Lem. 13)} \\ =& \Sigma_{\texttt{iconf}}^{\mathbf{X}} \cup \Sigma_1^{\mathbf{X}}[\sigma] \cup \Sigma_2[\sigma] \cup \Sigma_3[\sigma] \cup lazyCons(G, \Pi_1^{\mathbf{X}}) \cup lazyCons(G, \Pi_2) \\ & \text{By (58.14), (ICS-union)} \\ \xrightarrow{\texttt{algebra}} & \Sigma_{\texttt{iconf}}^{\mathbf{Y}} \cup \Sigma_1^{\mathbf{Y}}[\sigma] \cup \Sigma_2[\sigma] \cup \Sigma_3[\sigma] \cup lazyCons(G, \Pi_1^{\mathbf{Y}}) \cup lazyCons(G, \Pi_2) \\ & \text{By definition of substitution and } lazyCons \text{ (Lem. 13)} \\ =& \Sigma_{\texttt{iconf}}^{\mathbf{Y}} \cup (\Sigma_1^{\mathbf{Y}} \cup \Sigma_2 \cup \Sigma_3)[\sigma] \cup lazyCons(G, \Pi_1^{\mathbf{Y}} \cup \Pi_2) \\ =& \Sigma_{\texttt{iconf}}^{\mathbf{Y}} \cup \Sigma^{\mathbf{Y}}[\sigma] \cup lazyCons(G, \Pi^{\mathbf{Y}})\end{aligned}$$

Case: the last step is not an instance of (T-Sub), $\mathbf{E} = v\texttt{->m}(\mathbf{E}_0)$

According to the definition of evaluation context hole substitution, we know $\mathbf{E}[exd^{\mathbf{X}}] = v\texttt{->}\mathbf{E}_0[exd^{\mathbf{X}}]$. Thus the last non-subsumption step of the derivation tree leading to (**58.Cond1**) must be an instance of (T-HandleM), in which case we have

$$\begin{aligned}\Gamma, \mathcal{C} \vdash v : \tau_1 \backslash \Sigma_1, \Pi_1 \\ \Gamma, \mathcal{C} \vdash \mathbf{E}_0[exd^{\mathbf{X}}] : \tau_2 \backslash \Sigma_2^{\mathbf{X}}, \Pi_2^{\mathbf{X}} \\ \Sigma_3 = some\_constraints(\Gamma(\textbf{me}), \Gamma(\textbf{current}), \tau_1, \tau_2, \tau, \mathcal{C}) \\ \Sigma^{\mathbf{X}} = \Sigma_1 \cup \Sigma_2^{\mathbf{X}} \cup \Sigma_3 \\ \Pi^{\mathbf{X}} = \Pi_1 \cup \Pi_2^{\mathbf{X}}\end{aligned}$$

Since $v \in \mathbb{V}$, by performing case analysis on v we immediately know $\Pi_1 = \emptyset$. Given the fact that (**58.Cond5**) holds, and $\Gamma(\textbf{currentRt}) = (\Gamma \rhd [\textbf{iconf} : \boldsymbol{\Delta}])(\textbf{currentRt})$. We know $\Gamma \rhd [\textbf{iconf} : \boldsymbol{\Delta}], \mathcal{C} \vdash v : \tau_1 \backslash \Sigma_1, \Pi_1$ according to Lem. 50 on type preservation of values with the presence of a stronger configuration. The rest of the proof is the same as the previous case.

Case: the last step is not an instance of (T-Sub), $\mathbf{E}$ is some other form

According to the definition of evaluation context, we know $\mathbf{E}$ can also be one of the following forms.

$$\begin{aligned} &\textbf{connect } \mathbf{E}_0 \textbf{ with } \mathsf{k} \ \texttt{>>} \ \mathsf{k}' \\ &\mathsf{f} := \mathbf{E}_0 \\ &:: \mathsf{m}(\mathbf{E}_0) \\ &\mathsf{k} :: \mathsf{m}(\mathbf{E}_0) \\ &\mathbf{E}_0; e \end{aligned}$$

All the cases can be easily proved following the spirit of the previous cases.

□

Lemma 59 (Typing Subexpressions in the Context Hole)**.** *If there exists a derivation tree $\mathcal{DT}$ concluding $\Gamma, \mathcal{C} \vdash \mathbf{E}[\, exd \,] : \tau \backslash \Sigma, \Pi$, then there exists τ' and Σ' and Π' and a derivation tree $\mathcal{DT}'$ that occurs in $\mathcal{DT}$ in the position corresponding to $\bullet$, such that*

$$\Gamma, \mathcal{C} \vdash exd : \tau' \backslash \Sigma', \Pi' \tag{59.1}$$
the last step of derivation concluding (**59.1**) *is not* (T-Sub)
$$\Sigma' \subseteq \Sigma$$
$$\Pi' \subseteq \Pi$$

Proof. Induction on the height of the deduction tree of $\Gamma, \mathcal{C} \vdash \mathbf{E}[\, exd \,] : \tau \backslash \Sigma, \Pi$, and case analysis on the last step for $\mathbf{E}$. Straightforward. □

C.13 Other Properties

Lemma 60 (Configuration Constraints Are Not Changed Over Instantiation Substitution)**.** *If $G, \mathcal{C} \vdash_{\texttt{iconf}} \Delta : \langle \mathcal{H}; W; R \rangle \backslash \Sigma$, and for any o such that $\mathcal{H}(o) = \langle \mathsf{a}; \sigma; \mathcal{S} \rangle$, we know $\Sigma[\sigma] = \Sigma$.*

Proof. Directly follows (WFT-InstanceConfig). □

Lemma 61 (Type Preservation for User-Defined Expressions)**.** *If* $e \in \mathbb{EXP}$ *and* $\Gamma, \mathcal{C} \vdash e : \tau \backslash \Sigma, \Pi$*, then* $\Gamma \rhd [\textbf{iconf} : \boldsymbol{\Delta}, \textbf{currentRt} : o], \mathcal{C} \vdash e : \tau \backslash \Sigma, \Pi$ *for any* $\boldsymbol{\Delta}$*,* o*.*

Proof. Straightforward induction on $\Gamma, \mathcal{C} \vdash e : \tau \backslash \Sigma, \Pi$ and case analysis on the last non-subsumption step. Note that since e ranges over $\mathbb{EXP}$ (note: it is not the set for extended expressions, $\mathbb{EXD}$), which only needs to be typechecked by rules defined for user-definable expressions. None of the rules depends on **iconf** or **currentRt** in Γ. □

C.14 Initial Configuration Proof

Main Lemma M1 (Initial Configuration)**.** *If* $\vdash_{\mathtt{G}} C : Ct$ *and* $C \xrightarrow{\mathtt{init}} \langle \Delta; o; exd \rangle$*, then* $\vdash \langle C; \Delta; o; exd \rangle : \tau \backslash \Sigma$*.*

Proof. Straightforward on the intial state. □

C.15 Subject Reduction for Expression c^{side} ->m(v)

This is the most complex case for subject reduction. We have two sub-cases, depending on whether the computation is continued in the same objectage or in a different one. They are covered by Lem. 62 and Lem. 63, respectively.

Lemma 62 (Subject Reduction (R-HandleM) Without Overriding)**.** *Given*

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash c^{side}\texttt{->}\mathsf{m}(v) : \tau \backslash \Sigma_1, \Pi_1 \tag{62.1}$$

$$\vdash_{\mathtt{G}} C : Ct,\ \mathcal{C} = \Psi(Ct),\ G = global(C, \mathcal{C}) \tag{62.2}$$

$$G, \mathcal{C} \vdash_{\mathtt{iconf}} \Delta : \boldsymbol{\Delta} \backslash \Sigma_{\mathtt{iconf}} \tag{62.3}$$

$$\Delta = \langle H;\ W;\ R \rangle$$

$$W(c) = \left\{ \begin{array}{ll} \langle o^{\mathbf{X}}; \mathsf{k}^{\mathbf{X}}; o^{\mathbf{Y}}; \mathsf{k}^{\mathbf{Y}} \rangle & \textit{if } side = \mathtt{A} \\ \langle o^{\mathbf{Y}}; \mathsf{k}^{\mathbf{Y}}; o^{\mathbf{X}}; \mathsf{k}^{\mathbf{X}} \rangle & \textit{if } side = \mathtt{P} \end{array} \right. \tag{62.4}$$

$$H(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; S^{\mathbf{X}} \rangle,\ H(o^{\mathbf{Y}}) = \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; S^{\mathbf{Y}} \rangle$$

$$C(\mathsf{a}^{\mathbf{X}}) = \langle K^{\mathbf{X}}; M^{\mathbf{X}}; F^{\mathbf{X}} \rangle,\ C(\mathsf{a}^{\mathbf{Y}}) = \langle K^{\mathbf{Y}}; M^{\mathbf{Y}}; F^{\mathbf{Y}} \rangle$$

$$K^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}}) = \langle I^{\mathbf{X}}; E^{\mathbf{X}} \rangle,\ K^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}}) = \langle I^{\mathbf{Y}}; E^{\mathbf{Y}} \rangle$$

$$E^{\mathbf{X}}(\mathsf{m}) = \lambda x.e^{\mathbf{X}} \tag{62.5}$$

and the last step on the deduction tree of (**62.1**) *is* (T-HandleM)*, then*

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}\\ \textbf{me} : \mathsf{a}^{\mathbf{X}}\\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash e^{\mathbf{X}}\{v/x\}\{c^{side}/\textbf{current}\} : \tau \backslash \Sigma_2, \Pi_2$$

and

$$\Sigma_1[\sigma^{\mathbf{X}}] \cup \Sigma_{\mathtt{iconf}} \xrightarrow{\mathtt{algebra}} \Sigma_2[\sigma^{\mathbf{X}}] \cup \mathit{lazyCons}(G, \Pi_2)$$

Proof. With (**62.2**), (T-Global), (T-Classage), (T-Connectors), (T-Methods), and the definition of global, we know

$$\begin{aligned}
\mathcal{C}(\mathsf{a}^{\mathbf{X}}) &= \overline{\forall \langle \alpha^{\mathbf{X}}; \beta^{\mathbf{X}} \rangle}.\ \langle \mathcal{K}^{\mathbf{X}}; \mathcal{M}^{\mathbf{X}}; \mathcal{F}^{\mathbf{X}} \rangle \\
\mathcal{K}^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}}) &= \langle \mathcal{I}^{\mathbf{X}}; \mathcal{E}^{\mathbf{X}} \rangle \\
\mathrm{Dom}(I^{\mathbf{X}}) &= \mathrm{Dom}(\mathcal{I}^{\mathbf{X}}) \\
\mathrm{Dom}(E^{\mathbf{X}}) &= \mathrm{Dom}(\mathcal{E}^{\mathbf{X}}) \\
\mathcal{C}(\mathsf{a}^{\mathbf{Y}}) &= \overline{\forall \langle \alpha^{\mathbf{Y}}; \beta^{\mathbf{Y}} \rangle}.\ \langle \mathcal{K}^{\mathbf{Y}}; \mathcal{M}^{\mathbf{Y}}; \mathcal{F}^{\mathbf{Y}} \rangle \\
\mathcal{K}^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}}) &= \langle \mathcal{I}^{\mathbf{Y}}; \mathcal{E}^{\mathbf{Y}} \rangle \\
\mathrm{Dom}(I^{\mathbf{Y}}) &= \mathrm{Dom}(\mathcal{I}^{\mathbf{Y}}) \\
\mathrm{Dom}(E^{\mathbf{Y}}) &= \mathrm{Dom}(\mathcal{E}^{\mathbf{Y}}) \\
\mathcal{E}^{\mathbf{X}}(\mathsf{m}) &= c\tau^{\mathbf{X}}_{\mathtt{contra}} \to c\tau^{\mathbf{X}}_{\mathtt{co}} \\
[\textbf{me} : \mathsf{a}^{\mathbf{X}}, \textbf{current} : \mathsf{k}^{\mathbf{X}}, x : c\tau^{\mathbf{X}}_{\mathtt{contra}}], \mathcal{C} \vdash e^{\mathbf{X}} &: c\tau^{\mathbf{X}}_{\mathtt{co}} \backslash \Sigma^{\mathbf{X}}_{\mathtt{m}}, \Pi^{\mathbf{X}}_{\mathtt{m}} \qquad (\mathbf{62.6}) \\
G(\mathsf{a}^{\mathbf{X}}) &= \langle \Sigma^{\mathbf{X}}_{\mathtt{cls}}; \Pi^{\mathbf{X}}_{\mathtt{cls}} \rangle \\
\Sigma^{\mathbf{X}}_{\mathtt{m}} \subseteq \Sigma^{\mathbf{X}}_{\mathtt{cls}}, \Pi^{\mathbf{X}}_{\mathtt{m}} &\subseteq \Pi^{\mathbf{X}}_{\mathtt{cls}}
\end{aligned}$$

With (**62.3**) and (T-InstanceConfig), (T-Heap), (T-HeapCell), we know

$$\begin{aligned}
\boldsymbol{\Delta} &= \langle \mathcal{H};\ W; R \rangle \\
\mathcal{H}(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; \mathcal{S}^{\mathbf{X}} \rangle,\ \mathcal{H}(o^{\mathbf{Y}}) &= \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; \mathcal{S}^{\mathbf{Y}} \rangle \\
\Sigma^{\mathbf{X}}_{\mathtt{cls}}[\sigma^{\mathbf{X}}] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{X}}_{\mathtt{cls}}) &\subseteq \Sigma_{\mathtt{iconf}} \qquad (\mathbf{62.7})
\end{aligned}$$

By Lem. 14 and earlier fact that $\Pi^{\mathbf{X}}_{\mathtt{m}} \subseteq \Pi^{\mathbf{X}}_{\mathtt{cls}}$, we know $\mathit{lazyCons}(G, \Pi^{\mathbf{X}}_{\mathtt{m}}) \subseteq \mathit{lazyCons}(G, \Pi^{\mathbf{X}}_{\mathtt{cls}})$. By $\Sigma^{\mathbf{X}}_{\mathtt{m}} \subseteq \Sigma^{\mathbf{X}}_{\mathtt{cls}}$, and Lem. 12, we know $\Sigma^{\mathbf{X}}_{\mathtt{m}}[\sigma^{\mathbf{X}}] \subseteq \Sigma^{\mathbf{X}}_{\mathtt{cls}}[\sigma^{\mathbf{X}}]$. And hence $\Sigma^{\mathbf{X}}_{\mathtt{m}}[\sigma^{\mathbf{X}}] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{X}}_{\mathtt{m}}) \subseteq \Sigma^{\mathbf{X}}_{\mathtt{cls}}[\sigma^{\mathbf{X}}] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{X}}_{\mathtt{cls}})$. With this, and (**62.7**), we know

$$\Sigma^{\mathbf{X}}_{\mathtt{m}}[\sigma^{\mathbf{X}}] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{X}}_{\mathtt{m}}) \subseteq \Sigma_{\mathtt{iconf}} \qquad (\mathbf{62.8})$$

By (**62.1**) and (T-HandleM), we have

$$\exists \mathsf{k}_0 \text{ s. t. } \left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash c^{side} : \mathsf{k}_0 \backslash \Sigma_{\mathtt{cid}}, \Sigma_{\mathtt{cid}}$$

$$\mathcal{K}^{\mathbf{X}}(\mathsf{k}_0) = \langle \mathcal{I}_0; \mathcal{E}_0 \rangle$$

$$(\mathcal{I}_0 \rhd \mathcal{E}_0)(\mathsf{m}) = \tau' \to \tau$$

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash v : \tau' \backslash \Sigma_{\mathtt{v}}, \Pi_{\mathtt{v}} \tag{62.9}$$

$$\Sigma_1 = \Sigma_{\mathtt{cid}} \cup \Sigma_{\mathtt{v}}$$

$$\Pi_1 = \Pi_{\mathtt{cid}} \cup \Pi_{\mathtt{v}}$$

$$\mathrm{Dom}(\mathcal{I}_0) \cap \mathrm{Dom}(\mathcal{E}_0) = \emptyset$$

By (**62.4**), with (T-CID), we know it must be true that $\mathsf{k}_0 = \mathsf{k}^{\mathbf{X}}$, $\Sigma_{\mathtt{cid}} = \emptyset$, and $\Pi_{\mathtt{cid}} = \emptyset$. (No subtyping rule exists for connection handles, the previous judgment for c^{side} thus must have its last deduction step not being (T-Sub).) The previous equation of $\mathcal{K}^{\mathbf{X}}(\mathsf{k}_0) = \langle \mathcal{I}_0; \mathcal{E}_0 \rangle$ thus in fact is $\mathcal{K}^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}}) = \langle \mathcal{I}^{\mathbf{X}}; \mathcal{E}^{\mathbf{X}} \rangle$ as we have already known. Rewrite the previous equations, we have

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash c^{side} : \mathsf{k}^{\mathbf{X}} \backslash \emptyset, \emptyset \tag{62.10}$$

$$(\mathcal{I}^{\mathbf{X}} \rhd \mathcal{E}^{\mathbf{X}})(\mathsf{m}) = \tau' \to \tau \tag{62.11}$$

$$\Sigma_1 = \Sigma_{\mathtt{v}} \tag{62.12}$$

$$\Pi_1 = \Pi_{\mathtt{v}} \tag{62.13}$$

$$\mathrm{Dom}(\mathcal{I}^{\mathbf{X}}) \cap \mathrm{Dom}(\mathcal{E}^{\mathbf{X}}) = \emptyset$$

With (**62.11**), the fact that $\mathrm{Dom}(\mathcal{I}^{\mathbf{X}}) \cap \mathrm{Dom}(\mathcal{E}^{\mathbf{X}}) = \emptyset$, together with the definition of $\rhd$,we know $\mathsf{m} \in \mathrm{Dom}(\mathcal{E}^{\mathbf{X}}) \implies \mathcal{E}^{\mathbf{X}}(\mathsf{m}) = \tau' \to \tau$. Earlier we have shown $\mathrm{Dom}(E^{\mathbf{X}}) = \mathrm{Dom}(\mathcal{E}^{\mathbf{X}})$. By (**62.5**), we indeed know $\mathsf{m} \in \mathrm{Dom}(E^{\mathbf{X}})$, and thus we know $\mathcal{E}^{\mathbf{X}}(\mathsf{m}) = \tau' \to \tau$. Note that earlier we know $\mathcal{E}^{\mathbf{X}}(\mathsf{m}) = c\tau^{\mathbf{X}}_{\mathtt{contra}} \to c\tau^{\mathbf{X}}_{\mathtt{co}}$. Hence $\tau' = c\tau^{\mathbf{X}}_{\mathtt{contra}}$ and $\tau = c\tau^{\mathbf{X}}_{\mathtt{co}}$. Rewrite (**62.6**) and we have: $[\textbf{me} : \mathsf{a}^{\mathbf{X}}, \textbf{current} : \mathsf{k}^{\mathbf{X}}, x : \tau'], \mathcal{C} \vdash e^{\mathbf{X}} : \tau \backslash \Sigma^{\mathbf{X}}_{\mathtt{m}}, \Pi^{\mathbf{X}}_{\mathtt{m}}$. Now obviously $e^{\mathbf{X}} \in \mathbb{EXP}$ (since it is the non-reduced body of a method), and by Lem. 61, we know the type of a user-defined expression is always preserved if we change **iconf** and **currentRt**, and hence we have

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{current} : \mathsf{k}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \\ x : \tau' \end{array}\right], \mathcal{C} \vdash e^{\mathbf{X}} : \tau \backslash \Sigma^{\mathbf{X}}_{\mathtt{m}}, \Pi^{\mathbf{X}}_{\mathtt{m}} \tag{62.14}$$

Earlier we have shown how to type v in (**62.9**). Since $v \in \mathbb{V}$, by Lem. 57 we know adding **current** to the typing environment does not change the typing, and hence

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{current} : \mathsf{k}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash v : \tau \backslash \Sigma_{\mathtt{v}}, \Pi_{\mathtt{v}} \tag{62.15}$$

By (**62.14**), (**62.15**) and the substitution lemma (Lem. 53), we know

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{current} : \mathsf{k}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash e^{\mathbf{X}}\{v/x\} : \tau \backslash \Sigma_{\mathtt{m}}^{\mathbf{X}} \cup \Sigma_{\mathtt{v}}, \Pi_{\mathtt{m}}^{\mathbf{X}} \cup \Pi_{\mathtt{v}}$$

Earlier we have also shown how to type c^{side} in (**62.10**), by (T-CID), we have

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{current} : \mathsf{k}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash c^{side} : \mathsf{k}^{\mathbf{X}} \backslash \emptyset, \emptyset$$

By the substitution lemma (Lem. 53) again, we know

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash e^{\mathbf{X}}\{v/x\}\{c^{side}/\textbf{current}\} : \tau \backslash \Sigma_{\mathtt{m}}^{\mathbf{X}} \cup \Sigma_{\mathtt{v}}, \Pi_{\mathtt{m}}^{\mathbf{X}} \cup \Pi_{\mathtt{v}}$$

By (**62.9**) and Lem. 54 we know $\Pi_{\mathtt{v}} = \emptyset$. Let $\Sigma_2 = \Sigma_{\mathtt{m}}^{\mathbf{X}} \cup \Sigma_{\mathtt{v}}$ and $\Pi_2 = \Pi_{\mathtt{m}}^{\mathbf{X}}$. We know the above judgment is precisely the conclusion iff we can prove $\Sigma_1[\sigma^{\mathbf{X}}] \cup \Sigma_{\mathtt{iconf}} \xrightarrow{\texttt{algebra}} \Sigma_2[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_2)$. This indeed holds because

$$\begin{array}{rl} & \Sigma_1[\sigma^{\mathbf{X}}] \cup \Sigma_{\mathtt{iconf}} \\ & \text{By (\textbf{62.12})} \\ = & \Sigma_{\mathtt{v}}[\sigma^{\mathbf{X}}] \cup \Sigma_{\mathtt{iconf}} \\ & \text{By (\textbf{62.8}), (ICS-relaxation)} \\ \xrightarrow{\texttt{algebra}} & \Sigma_{\mathtt{m}}^{\mathbf{X}}[\sigma^{\mathbf{X}}] \cup \Sigma_{\mathtt{v}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\mathtt{m}}^{\mathbf{X}}) \\ & \text{By the definition of } \Sigma_2 \text{ and } \Pi_2 \text{ above} \\ = & \Sigma_2[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_2) \end{array}$$

□

Lemma 63 (Subject Reduction over (R-HandleM) with Overriding)**.** *Given*

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash c^{side}\texttt{->}\mathsf{m}(v) : \tau \backslash \Sigma_1, \Pi \tag{63.1}$$

$$\vdash_{\mathtt{G}} C : Ct,\ \mathcal{C} = \Psi(Ct),\ G = global(C, \mathcal{C}) \tag{63.2}$$

$$G, \mathcal{C} \vdash_{\texttt{iconf}} \Delta : \boldsymbol{\Delta} \backslash \Sigma_{\texttt{iconf}} \tag{63.3}$$
$$\Delta = \langle H;\, W;\, R \rangle$$
$$W(c) = \begin{cases} \langle o^{\mathbf{X}}; \mathsf{k}^{\mathbf{X}}; o^{\mathbf{Y}}; \mathsf{k}^{\mathbf{Y}} \rangle & \textit{if } side = \texttt{A} \\ \langle o^{\mathbf{Y}}; \mathsf{k}^{\mathbf{Y}}; o^{\mathbf{X}}; \mathsf{k}^{\mathbf{X}} \rangle & \textit{if } side = \texttt{P} \end{cases} \tag{63.4}$$
$$H(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; S^{\mathbf{X}} \rangle,\ H(o^{\mathbf{Y}}) = \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; S^{\mathbf{Y}} \rangle$$
$$C(\mathsf{a}^{\mathbf{X}}) = \langle K^{\mathbf{X}}; M^{\mathbf{X}}; F^{\mathbf{X}} \rangle,\ C(\mathsf{a}^{\mathbf{Y}}) = \langle K^{\mathbf{Y}}; M^{\mathbf{Y}}; F^{\mathbf{Y}} \rangle$$
$$K^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}}) = \langle I^{\mathbf{X}}; E^{\mathbf{X}} \rangle,\ K^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}}) = \langle I^{\mathbf{Y}}; E^{\mathbf{Y}} \rangle$$
$$E^{\mathbf{Y}}(\mathsf{m}) = \lambda x.e^{\mathbf{Y}}$$
$$\mathsf{m} \in \mathrm{Dom}(I^{\mathbf{X}}) \tag{63.5}$$
$$side' = \begin{cases} \texttt{P} & \textit{if } side = \texttt{A} \\ \texttt{A} & \textit{if } side = \texttt{P} \end{cases}$$

and the last step on the deduction tree of (**63.1**) *is* (T-HandleM)*, then*

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash \textbf{in}_{side'}(o^{\mathbf{Y}}, e^{\mathbf{Y}}\{v/x\}\{c^{side'}/\textbf{current}\}) : \tau \backslash \Sigma_2, \Pi$$

and

$$\Sigma_1[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{iconf}} \xrightarrow{\texttt{algebra}} \Sigma_2$$

Proof. With (**63.2**), and (T-Global), (T-Classage), (T-Connectors), (T-Methods), and the definition of *global*, we know

$$\mathcal{C}(\mathsf{a}^{\mathbf{X}}) = \overline{\forall \langle \alpha_u^{\mathbf{X}}; \beta_u^{\mathbf{X}} \rangle}.\ \langle \mathcal{K}^{\mathbf{X}}; \mathcal{M}^{\mathbf{X}}; \mathcal{F}^{\mathbf{X}} \rangle$$
$$\mathcal{K}^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}}) = \langle \mathcal{I}^{\mathbf{X}}; \mathcal{E}^{\mathbf{X}} \rangle$$
$$\mathrm{Dom}(I^{\mathbf{X}}) = \mathrm{Dom}(\mathcal{I}^{\mathbf{X}})$$
$$\mathrm{Dom}(E^{\mathbf{X}}) = \mathrm{Dom}(\mathcal{E}^{\mathbf{X}})$$
$$\mathcal{C}(\mathsf{a}^{\mathbf{Y}}) = \overline{\forall \langle \alpha_v^{\mathbf{Y}}; \beta_v^{\mathbf{Y}} \rangle}.\ \langle \mathcal{K}^{\mathbf{Y}}; \mathcal{M}^{\mathbf{Y}}; \mathcal{F}^{\mathbf{Y}} \rangle$$
$$\mathcal{K}^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}}) = \langle \mathcal{I}^{\mathbf{Y}}; \mathcal{E}^{\mathbf{Y}} \rangle$$
$$\mathrm{Dom}(I^{\mathbf{Y}}) = \mathrm{Dom}(\mathcal{I}^{\mathbf{Y}})$$
$$\mathrm{Dom}(E^{\mathbf{Y}}) = \mathrm{Dom}(\mathcal{E}^{\mathbf{Y}})$$
$$\mathcal{E}^{\mathbf{Y}}(\mathsf{m}) = c\tau^{\mathbf{Y}}_{\texttt{contra}} \to c\tau^{\mathbf{Y}}_{\texttt{co}}$$
$$[\textbf{me} : \mathsf{a}^{\mathbf{Y}}, \textbf{current} : \mathsf{k}^{\mathbf{Y}}, x : c\tau^{\mathbf{Y}}_{\texttt{contra}}], \mathcal{C} \vdash e^{\mathbf{Y}} : c\tau^{\mathbf{Y}}_{\texttt{co}} \backslash \Sigma^{\mathbf{Y}}_{\mathsf{m}}, \Pi^{\mathbf{Y}}_{\mathsf{m}} \tag{63.6}$$
$$G(\mathsf{a}^{\mathbf{Y}}) = \langle \Sigma^{\mathbf{Y}}_{\texttt{cls}}; \Pi^{\mathbf{Y}}_{\texttt{cls}} \rangle$$
$$\Sigma^{\mathbf{Y}}_{\mathsf{m}} \subseteq \Sigma^{\mathbf{Y}}_{\texttt{cls}}, \Pi^{\mathbf{Y}}_{\mathsf{m}} \subseteq \Pi^{\mathbf{Y}}_{\texttt{cls}}$$

By Lem. 9 and its pre-condition (**63.2**), we know $\Sigma^{\mathbf{Y}}_{\texttt{cls}} \xrightarrow{\texttt{algebra}} wfT(c\tau^{\mathbf{Y}}_{\texttt{co}}) \cup wfT(c\tau^{\mathbf{Y}}_{\texttt{contra}})$. With (**63.3**), (T-InstanceConfig), (T-Heap), (T-HeapCell), we know

$$\boldsymbol{\Delta} = \langle \mathcal{H};\, W;\, R \rangle$$

$$\mathcal{H}(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; \mathcal{S}^{\mathbf{X}} \rangle,\ \mathcal{H}(o^{\mathbf{Y}}) = \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; \mathcal{S}^{\mathbf{Y}} \rangle$$
$$\Sigma^{\mathbf{Y}}_{\mathtt{cls}}[\sigma^{\mathbf{Y}}] \cup lazyCons(G, \Pi^{\mathbf{Y}}_{\mathtt{cls}}) \subseteq \Sigma_{\mathtt{iconf}} \tag{63.7}$$

By Lem. 14 and earlier fact that $\Pi^{\mathbf{Y}}_{\mathtt{m}} \subseteq \Pi^{\mathbf{Y}}_{\mathtt{cls}}$, we know $lazyCons(G, \Pi^{\mathbf{Y}}_{\mathtt{m}}) \subseteq lazyCons(G, \Pi^{\mathbf{Y}}_{\mathtt{cls}})$. Thus $lazyCons(G, \Pi^{\mathbf{Y}}_{\mathtt{cls}}) \xrightarrow{\mathtt{algebra}} lazyCons(G, \Pi^{\mathbf{Y}}_{\mathtt{m}})$ by (ICS-relaxation). By $\Sigma^{\mathbf{Y}}_{\mathtt{m}} \subseteq \Sigma^{\mathbf{Y}}_{\mathtt{cls}}$ and $\Sigma^{\mathbf{Y}}_{\mathtt{cls}} \xrightarrow{\mathtt{algebra}} wfT(c\tau^{\mathbf{Y}}_{\mathtt{contra}})$, and Lem. 12, we know $\Sigma^{\mathbf{Y}}_{\mathtt{cls}}[\sigma^{\mathbf{Y}}] \xrightarrow{\mathtt{algebra}} wfT(c\tau^{\mathbf{Y}}_{\mathtt{contra}})[\sigma^{\mathbf{Y}}] \cup \Sigma^{\mathbf{Y}}_{\mathtt{m}}[\sigma^{\mathbf{Y}}]$. And hence we know

$$\Sigma^{\mathbf{Y}}_{\mathtt{cls}}[\sigma^{\mathbf{Y}}] \cup lazyCons(G, \Pi^{\mathbf{Y}}_{\mathtt{cls}}) \xrightarrow{\mathtt{algebra}} wfT(c\tau^{\mathbf{Y}}_{\mathtt{contra}})[\sigma^{\mathbf{Y}}] \cup \Sigma^{\mathbf{Y}}_{\mathtt{m}}[\sigma^{\mathbf{Y}}] \cup lazyCons(G, \Pi^{\mathbf{Y}}_{\mathtt{m}})$$

With this, and (**63.7**), we know

$$\Sigma_{\mathtt{iconf}} \xrightarrow{\mathtt{algebra}} wfT(c\tau^{\mathbf{Y}}_{\mathtt{contra}})[\sigma^{\mathbf{Y}}] \cup \Sigma^{\mathbf{Y}}_{\mathtt{m}}[\sigma^{\mathbf{Y}}] \cup lazyCons(G, \Pi^{\mathbf{Y}}_{\mathtt{m}}) \tag{63.8}$$

Next Task: Typing $c^{side}, v, \mathcal{I}^{\mathbf{X}}(\mathsf{m})$

By (**63.1**) and (T-HandleM) we know

$$\begin{aligned}
\exists \mathsf{k}_0 \text{ s. t. } \begin{bmatrix} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{bmatrix}, \mathcal{C} \vdash c^{side} : \mathsf{k}_0 \backslash \Sigma_{\mathtt{cid}}, \Sigma_{\mathtt{cid}} \\
\mathcal{K}^{\mathbf{X}}(\mathsf{k}_0) = \langle \mathcal{I}_0; \mathcal{E}_0 \rangle \\
(\mathcal{I}_0 \rhd \mathcal{E}_0)(\mathsf{m}) = \tau' \to \tau \\
\begin{bmatrix} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{bmatrix}, \mathcal{C} \vdash v : \tau' \backslash \Sigma_{\mathtt{v}}, \Pi_{\mathtt{v}} \\
\Sigma_1 = \Sigma_{\mathtt{cid}} \cup \Sigma_{\mathtt{v}} \\
\Pi = \Pi_{\mathtt{cid}} \cup \Pi_{\mathtt{v}} \\
\mathrm{Dom}(\mathcal{I}_0) \cap \mathrm{Dom}(\mathcal{E}_0) = \emptyset
\end{aligned} \tag{63.9}$$

By (**63.4**), with (T-CID), we know it must be true that $\mathsf{k}_0 = \mathsf{k}^{\mathbf{X}}$, $\Sigma_{\mathtt{cid}} = \emptyset$, and $\Pi_{\mathtt{cid}} = \emptyset$. (No subtyping rule exists for connection handles, the previous judgment for c^{side} thus must have its last deduction step not being (T-Sub).) The previous equation of $\mathcal{K}^{\mathbf{X}}(\mathsf{k}_0) = \langle \mathcal{I}_0; \mathcal{E}_0 \rangle$ thus in fact is $\mathcal{K}^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}}) = \langle \mathcal{I}^{\mathbf{X}}; \mathcal{E}^{\mathbf{X}} \rangle$ as we have already known. Rewrite the previous equations, we have

$$\begin{bmatrix} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{bmatrix}, \mathcal{C} \vdash c^{side} : \mathsf{k}^{\mathbf{X}} \backslash \emptyset, \emptyset \tag{63.10}$$
$$(\mathcal{I}^{\mathbf{X}} \rhd \mathcal{E}^{\mathbf{X}})(\mathsf{m}) = \tau' \to \tau \tag{63.11}$$
$$\Sigma_1 = \Sigma_{\mathtt{v}} \tag{63.12}$$

$$\Pi = \Pi_{\mathtt{v}} \tag{63.13}$$
$$\mathrm{Dom}(\mathcal{I}^{\mathbf{X}}) \cap \mathrm{Dom}(\mathcal{E}^{\mathbf{X}}) = \emptyset$$

With (**63.11**), the fact that $\mathrm{Dom}(\mathcal{I}^{\mathbf{X}}) \cap \mathrm{Dom}(\mathcal{E}^{\mathbf{X}}) = \emptyset$, together with the definition of $\triangleright$,we know $\mathsf{m} \in \mathrm{Dom}(\mathcal{I}^{\mathbf{X}}) \implies \mathcal{I}^{\mathbf{X}}(\mathsf{m}) = \tau' \to \tau$. Earlier we have shown $\mathrm{Dom}(I^{\mathbf{X}}) = \mathrm{Dom}(\mathcal{I}^{\mathbf{X}})$. By (**63.5**), we indeed know $\mathsf{m} \in \mathrm{Dom}(I^{\mathbf{X}})$, and thus we know $\mathcal{I}^{\mathbf{X}}(\mathsf{m}) = \tau' \to \tau$. Entirely identical to our proof for $\Sigma_{\mathtt{iconf}} \xrightarrow{\mathtt{algebra}} wfT(c\tau^{\mathbf{Y}}_{\mathtt{contra}})$ (See (**63.8**)), we are able to show

$$\Sigma_{\mathtt{iconf}} \xrightarrow{\mathtt{algebra}} wfT(\tau)[\sigma^{\mathbf{X}}] \tag{63.14}$$

Next Task: the Type Variables in τ, τ', $c\tau^{\mathbf{Y}}_{\mathtt{contra}}$, $c\tau^{\mathbf{Y}}_{\mathtt{co}}$

By Lem. 8 and the definition of $TV_{\mathtt{sig}}$, we know $TV(\mathcal{C}(\mathsf{a}^{\mathbf{X}})) = TV(\mathcal{K}^{\mathbf{X}}) \cup TV(\mathcal{M}^{\mathbf{X}}) \cup TV(\mathcal{F}^{\mathbf{X}})$. Obviously, $TV(\mathcal{I}^{\mathbf{X}}) \subseteq TV(\mathcal{K}^{\mathbf{X}})$. Since $\mathcal{I}^{\mathbf{X}}(\mathsf{m}) = \tau' \to \tau$. We thus know

$$TV(\tau) \cup TV(\tau') \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{X}})) \tag{63.15}$$

Similarly, since $\mathcal{E}^{\mathbf{Y}}(\mathsf{m}) = c\tau^{\mathbf{Y}}_{\mathtt{contra}} \to c\tau^{\mathbf{Y}}_{\mathtt{co}}$. We thus know

$$TV(c\tau^{\mathbf{Y}}_{\mathtt{contra}}) \cup TV(c\tau^{\mathbf{Y}}_{\mathtt{co}}) \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{Y}})) \tag{63.16}$$

Next Task: Finding Inter-Runtime Type Relations

There are two cases depending on the definition of $W(c)$.

Case $side = \mathtt{A}$. In this case we know $W(c) = \langle o^{\mathbf{X}}; \mathsf{k}^{\mathbf{X}}; o^{\mathbf{Y}}; \mathsf{k}^{\mathbf{Y}} \rangle$. According to (**63.3**), (T-InstanceConfig), (WFT-InstanceConfig) and (WFT-InstanceItr), we know

$$rel(R, o^{\mathbf{Y}}, o^{\mathbf{X}}) = \rho^{\mathbf{YX}} \tag{63.17}$$
$$\rho^{\mathbf{YX}} = (\textbf{parent})^{\nu_{\mathtt{a}}^{\mathbf{YX}}}(\textbf{child})^{\nu_{\mathtt{b}}^{\mathbf{YX}}}$$
$$\rho^{\mathbf{YX}} \vdash \mathcal{K}^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}})[\sigma^{\mathbf{X}}] \underset{\mathtt{dc}}{\overset{\Sigma^{\mathbf{YX}}_{\mathtt{match}}}{\Longrightarrow}} \mathcal{K}^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}})[\sigma^{\mathbf{Y}}]$$
$$\Sigma^{\mathbf{YX}}_{\mathtt{match}} \subseteq \Sigma_{\mathtt{iconf}} \tag{63.18}$$

With the $\underset{\mathtt{dc}}{\Rightarrow}$ judgment above, by (Def-ConnectorMatchD) and (Def-MethodMatchD) and (Sub-Method),

we know

$$matchd(c\tau^{\mathbf{Y}}_{\mathtt{co}}[\sigma^{\mathbf{Y}}], \tau[\sigma^{\mathbf{X}}], \rho^{\mathbf{YX}}) = \Sigma^{\mathbf{YX}}_{\mathtt{sub}} \quad \textbf{(63.19)}$$

$$matchd(\tau'[\sigma^{\mathbf{X}}], c\tau^{\mathbf{Y}}_{\mathtt{contra}}[\sigma^{\mathbf{Y}}], (\textbf{parent})^{\nu^{\mathbf{YX}}_{\mathtt{b}}}(\textbf{child})^{\nu^{\mathbf{YX}}_{\mathtt{a}}}) = \Sigma^{\mathbf{YX}'}_{\mathtt{sub}} \quad \textbf{(63.20)}$$

$$(\Sigma^{\mathbf{YX}}_{\mathtt{sub}} \cup \Sigma^{\mathbf{YX}'}_{\mathtt{sub}}) \subseteq \Sigma^{\mathbf{YX}}_{\mathtt{match}} \subseteq \Sigma_{\mathtt{iconf}} \quad \textbf{(63.21)}$$

Case $side = \mathtt{P}$. In this case we know $W(c) = \langle o^{\mathbf{Y}}; \mathsf{k}^{\mathbf{Y}}; o^{\mathbf{X}}; \mathsf{k}^{\mathbf{X}} \rangle$. According to (**63.3**), (T-InstanceConfig), (WFT-InstanceConfig) and (WFT-InstanceItr), we know

$$rel(R, o^{\mathbf{X}}, o^{\mathbf{Y}}) = \rho^{\mathbf{XY}} \quad \textbf{(63.22)}$$

$$\rho^{\mathbf{XY}} = (\textbf{parent})^{\nu^{\mathbf{XY}}_{\mathtt{a}}}(\textbf{child})^{\nu^{\mathbf{XY}}_{\mathtt{b}}}$$

$$\rho^{\mathbf{XY}} \vdash \mathcal{K}^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}})[\sigma^{\mathbf{Y}}] \overset{\Sigma^{\mathbf{XY}}_{\mathtt{match}}}{\underset{\mathtt{dc}}{\Longrightarrow}} \mathcal{K}^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}})[\sigma^{\mathbf{X}}]$$

$$\Sigma^{\mathbf{XY}}_{\mathtt{match}} \subseteq \Sigma_{\mathtt{iconf}} \quad \textbf{(63.23)}$$

With the $\underset{\mathtt{dc}}{\Longrightarrow}$ judgment above, by (Def-ConnectorMatchD) and (Def-MethodMatchD) and (Sub-Method), we know

$$matchd(c\tau^{\mathbf{Y}}_{\mathtt{co}}[\sigma^{\mathbf{Y}}], \tau[\sigma^{\mathbf{X}}], (\textbf{parent})^{\nu^{\mathbf{XY}}_{\mathtt{b}}}(\textbf{child})^{\nu^{\mathbf{XY}}_{\mathtt{a}}}) = \Sigma^{\mathbf{XY}}_{\mathtt{sub}} \quad \textbf{(63.24)}$$

$$matchd(\tau'[\sigma^{\mathbf{X}}], c\tau^{\mathbf{Y}}_{\mathtt{contra}}[\sigma^{\mathbf{Y}}], \rho^{\mathbf{XY}}) = \Sigma^{\mathbf{XY}'}_{\mathtt{sub}} \quad \textbf{(63.25)}$$

$$(\Sigma^{\mathbf{XY}}_{\mathtt{sub}} \cup \Sigma^{\mathbf{XY}'}_{\mathtt{sub}}) \subseteq \Sigma^{\mathbf{XY}}_{\mathtt{match}} \subseteq \Sigma_{\mathtt{iconf}} \quad \textbf{(63.26)}$$

Next Task: Typing v in $o^{\mathbf{Y}}$

Since there are two cases for inter-runtime type relations, there are two corresponding cases for typing v.

Case $side = \mathtt{A}$. By the important Lem. 55 on value passing, together with pre-conditions of (**63.9**), (**63.20**), (**63.17**), (**63.15**), (**63.16**), we know

$$\begin{bmatrix} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}} \end{bmatrix}, \mathcal{C} \vdash v : c\tau^{\mathbf{Y}}_{\mathtt{contra}} \backslash \Sigma^{\mathbf{YX}'}_{\mathtt{v}}, \emptyset$$

$$\Sigma_{\mathtt{v}}[\sigma^{\mathbf{X}}] \cup wfT(c\tau^{\mathbf{Y}}_{\mathtt{contra}})[\sigma^{\mathbf{Y}}] \cup \Sigma^{\mathbf{YX}'}_{\mathtt{sub}} \xrightarrow{\mathtt{algebra}} \Sigma^{\mathbf{YX}'}_{\mathtt{v}}[\sigma^{\mathbf{Y}}]$$

First quickly from (**63.12**), we know

$$\Sigma_1[\sigma^{\mathbf{X}}] \cup wfT(c\tau^{\mathbf{Y}}_{\mathtt{contra}})[\sigma^{\mathbf{Y}}] \cup \Sigma^{\mathbf{YX}'}_{\mathtt{sub}} \xrightarrow{\mathtt{algebra}} \Sigma^{\mathbf{YX}'}_{\mathtt{v}}[\sigma^{\mathbf{Y}}] \quad \textbf{(63.27)}$$

Since $v \in \mathbb{V}$, by Lem. 57 (value typing is independent of **current**). Thus we know

$$\begin{bmatrix} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{current} : \mathsf{k}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}} \end{bmatrix}, \mathcal{C} \vdash v : c\tau^{\mathbf{Y}}_{\texttt{contra}} \backslash \Sigma^{\mathbf{YX}'}_{\mathtt{v}}, \emptyset \tag{63.28}$$

Case $side = \texttt{P}$. By the important Lem. 56 on value passing, together with pre-conditions of (**63.9**), (**63.25**), (**63.22**), (**63.15**), (**63.16**), we know

$$\begin{bmatrix} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}} \end{bmatrix}, \mathcal{C} \vdash v : c\tau^{\mathbf{Y}}_{\texttt{contra}} \backslash \Sigma^{\mathbf{XY}'}_{\mathtt{v}}, \emptyset$$

$$\Sigma_{\mathtt{v}}[\sigma^{\mathbf{X}}] \cup wfT(c\tau^{\mathbf{Y}}_{\texttt{contra}})[\sigma^{\mathbf{Y}}] \cup \Sigma^{\mathbf{XY}'}_{\texttt{sub}} \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{XY}'}_{\mathtt{v}}[\sigma^{\mathbf{Y}}]$$

Like the previous case, we know

$$\Sigma_{\mathtt{v}}[\sigma^{\mathbf{X}}] \cup wfT(c\tau^{\mathbf{Y}}_{\texttt{contra}})[\sigma^{\mathbf{Y}}] \cup \Sigma^{\mathbf{XY}'}_{\texttt{sub}} \xrightarrow{\texttt{algebra}} \Sigma^{\mathbf{XY}'}_{\mathtt{v}}[\sigma^{\mathbf{Y}}] \tag{63.29}$$

$$\begin{bmatrix} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{current} : \mathsf{k}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}} \end{bmatrix}, \mathcal{C} \vdash v : c\tau^{\mathbf{Y}}_{\texttt{contra}} \backslash \Sigma^{\mathbf{XY}'}_{\mathtt{v}}, \emptyset \tag{63.30}$$

Next Task: Typing Substitutions and **in** Expression

By (**63.6**), and $e^{\mathbf{Y}} \in \mathbb{EXP}$ (since it is the non-reduced body of a method), and by Lem. 61, we know the type of a user-defined expression is always preserved if we change **iconf** and **currentRt**, and hence rewrite the previous judgment we have

$$\begin{bmatrix} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{current} : \mathsf{k}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}} \\ x : c\tau^{\mathbf{Y}}_{\texttt{contra}} \end{bmatrix}, \mathcal{C} \vdash e^{\mathbf{Y}} : c\tau^{\mathbf{Y}}_{\texttt{co}} \backslash \Sigma^{\mathbf{Y}}_{\mathtt{m}}, \Pi^{\mathbf{Y}}_{\mathtt{m}} \tag{63.31}$$

We now perform the familiar case analysis.

Case $side = \texttt{A}$. With (**63.28**) and the substitution lemma (Lem. 53), we know

$$\begin{bmatrix} \textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{current} : \mathsf{k}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}} \end{bmatrix}, \mathcal{C} \vdash e^{\mathbf{Y}}\{v/x\} : c\tau^{\mathbf{Y}}_{\texttt{co}} \backslash \Sigma^{\mathbf{Y}}_{\mathtt{m}} \cup \Sigma^{\mathbf{YX}'}_{\mathtt{v}}, \Pi^{\mathbf{Y}}_{\mathtt{m}}$$

By (**63.10**), (T-CID), we know $[\textbf{iconf} : \boldsymbol{\Delta}, \textbf{me} : \mathsf{a}^{\mathbf{Y}}, \textbf{currentRt} : o^{\mathbf{Y}}], \mathcal{C} \vdash c^{side'} : \mathsf{k}^{\mathbf{Y}} \backslash \emptyset, \emptyset$. With this, by the substitution lemma (Lem. 53) again, we know

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}}\end{array}\right], \mathcal{C} \vdash e^{\mathbf{Y}}\{v/x\}\{c^{side'}/\textbf{current}\} : c\tau^{\mathbf{Y}}_{\mathtt{co}} \backslash \Sigma^{\mathbf{Y}}_{\mathtt{m}} \cup \Sigma^{\mathbf{YX'}}_{\mathtt{v}}, \Pi^{\mathbf{Y}}_{\mathtt{m}}$$

With this, by (T-In), together with (**63.19**), (**63.17**), (**63.15**), (**63.16**), we know

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash \textbf{in}_{\mathtt{P}}(o^{\mathbf{Y}}, e^{\mathbf{Y}}\{v/x\}\{c^{side'}/\textbf{current}\}) : \tau \backslash \Sigma_2, \emptyset \tag{63.32}$$

where

$$\Sigma_2 = wfT(\tau)[\sigma^{\mathbf{X}}] \cup (\Sigma^{\mathbf{Y}}_{\mathtt{m}} \cup \Sigma^{\mathbf{YX'}}_{\mathtt{v}})[\sigma^{\mathbf{Y}}] \cup \Sigma^{\mathbf{YX}}_{\mathtt{sub}} \cup lazyCons(G, \Pi^{\mathbf{Y}}_{\mathtt{m}})$$

Earlier in (**63.13**) , we have shown that $\Pi = \Pi_{\mathtt{v}}$. By Lem. 54 and (**63.9**), we know $\Pi_{\mathtt{v}} = \emptyset$. Thus $\Pi = \emptyset$. Thus the previous judgment is precisely the conclusion as long as we can prove $\Sigma_1[\sigma^{\mathbf{X}}] \cup \Sigma_{\mathtt{iconf}} \xrightarrow{\texttt{algebra}} \Sigma_2$. It indeed holds because

$$\begin{aligned}
&\Sigma_1[\sigma^{\mathbf{X}}] \cup \Sigma_{\mathtt{iconf}} \\
&\quad \text{By (63.8), (63.21), (63.18), (ICS-relaxation), (ICS-union)} \\
\xrightarrow{\texttt{algebra}}\ &\Sigma_1[\sigma^{\mathbf{X}}] \cup wfT(\tau^{\mathbf{Y}}_{\mathtt{contra}})[\sigma^{\mathbf{Y}}] \cup \Sigma^{\mathbf{YX'}}_{\mathtt{sub}} \cup \Sigma_{\mathtt{iconf}} \\
&\quad \text{By (63.27)} \\
\xrightarrow{\texttt{algebra}}\ &\Sigma^{\mathbf{YX'}}_{\mathtt{v}}[\sigma^{\mathbf{Y}}] \cup \Sigma_{\mathtt{iconf}} \\
&\quad \text{By (63.21), (63.18), (ICS-relaxation)} \\
\xrightarrow{\texttt{algebra}}\ &\Sigma^{\mathbf{YX'}}_{\mathtt{v}}[\sigma^{\mathbf{Y}}] \cup \Sigma^{\mathbf{YX}}_{\mathtt{sub}} \cup \Sigma_{\mathtt{iconf}} \\
&\quad \text{By (63.8), (63.14), (ICS-union)} \\
\xrightarrow{\texttt{algebra}}\ &\Sigma^{\mathbf{YX'}}_{\mathtt{v}}[\sigma^{\mathbf{Y}}] \cup \Sigma^{\mathbf{YX}}_{\mathtt{sub}} \cup wfT(\tau)[\sigma^{\mathbf{X}}] \cup \Sigma^{\mathbf{Y}}_{\mathtt{m}}[\sigma^{\mathbf{Y}}] \cup lazyCons(G, \Pi^{\mathbf{Y}}_{\mathtt{m}}) \\
&\quad \text{By definition of substitution} \\
=\ &wfT(\tau)[\sigma^{\mathbf{X}}] \cup (\Sigma^{\mathbf{Y}}_{\mathtt{m}} \cup \Sigma^{\mathbf{YX'}}_{\mathtt{v}})[\sigma^{\mathbf{Y}}] \cup \Sigma^{\mathbf{YX}}_{\mathtt{sub}} \cup lazyCons(G, \Pi^{\mathbf{Y}}_{\mathtt{m}}) \\
=\ &\Sigma_2
\end{aligned}$$

Case $side = \mathtt{P}$. Following the identical way as the previous case, we will have the similar judgment as (**63.32**):

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta} \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash \textbf{in}_{\mathtt{A}}(o^{\mathbf{Y}}, e^{\mathbf{Y}}\{v/x\}\{c^{side'}/\textbf{current}\}) : \tau \backslash \Sigma_2, \emptyset$$

$$\Sigma_2 = wfT(\tau)[\sigma^{\mathbf{X}}] \cup (\Sigma^{\mathbf{Y}}_{\mathtt{m}} \cup \Sigma^{\mathbf{XY'}}_{\mathtt{v}})[\sigma^{\mathbf{Y}}] \cup \Sigma^{\mathbf{XY}}_{\mathtt{sub}} \cup lazyCons(G, \Pi^{\mathbf{Y}}_{\mathtt{m}})$$

Following the identical way as the previous case, we can show $\Pi = \emptyset$. Thus the previous judgment is

precisely the conclusion as long as we can prove $\Sigma_1[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{iconf}} \xrightarrow{\texttt{algebra}} \Sigma_2$. It indeed holds following the identical reasoning as the previous case, except that every use of (**63.21**), (**63.18**), (**63.27**) in the previous case is now replaced with a use of (**63.26**), (**63.23**), (**63.29**).

□

C.16 Subject Reduction for the connect Expression

Lemma 64 (Subject Reduction over (R-Connect))**.** *Given*

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash \textbf{connect } o^{\mathbf{Y}} \textbf{ with } \mathsf{k}^{\mathbf{X}} \ >> \ \mathsf{k}^{\mathbf{Y}} : \tau \backslash \Sigma_1, \Pi \tag{64.1}$$

$$\vdash_{\mathtt{G}} C : Ct, \mathcal{C} = \Psi(Ct), G = \mathit{global}(C, \mathcal{C}) \tag{64.2}$$

$$\Delta_1 = \langle H;\, W_1;\, R\rangle$$

$$\Delta_2 = \langle H;\, W_2;\, R\rangle$$

$$W_2 = W_1 \uplus (c \mapsto \langle o^{\mathbf{X}}; \mathsf{k}^{\mathbf{X}}; o^{\mathbf{Y}}; \mathsf{k}^{\mathbf{Y}}\rangle)$$

$$G, \mathcal{C} \vdash_{\texttt{iconf}} \Delta_1 : \boldsymbol{\Delta}_1 \backslash \Sigma_{\texttt{iconf1}} \tag{64.3}$$

$$G, \mathcal{C} \vdash_{\texttt{iconf}} \Delta_2 : \boldsymbol{\Delta}_2 \backslash \Sigma_{\texttt{iconf2}} \tag{64.4}$$

$$c \textit{ fresh}$$

and the last step on the deduction tree of (**64.1**) *is* (T-Connect)*, then* $[\textbf{iconf} : \boldsymbol{\Delta}_2, \textbf{me} : \mathsf{a}^{\mathbf{X}}, \textbf{currentRt} : o^{\mathbf{X}}], \mathcal{C} \vdash c^{\mathtt{A}} : \tau \backslash \Sigma_2, \Pi$ *and* $\Sigma_1[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{iconf1}} \xrightarrow{\texttt{algebra}} \Sigma_2[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{iconf2}}$.

Proof. By (**64.3**), (T-InstanceConfig), (T-Heap), (T-HeapCell), we know the correspondence between Δ_1 and $\boldsymbol{\Delta}_1$. Hence we have $\boldsymbol{\Delta}_1 = \langle \mathcal{H};\, W_1;\, R\rangle$ for some $\mathcal{H}$. Similarly, by By (**64.34**), (T-InstanceConfig), (T-Heap), (T-HeapCell), we have $\boldsymbol{\Delta}_2 = \langle \mathcal{H};\, W_2;\, R\rangle$. We also let $\mathcal{H}(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; \mathcal{S}^{\mathbf{X}}\rangle$. By (**64.1**), (T-Connect) we know

$$\mathcal{C}(\mathsf{a}^{\mathbf{X}}) = \overline{\forall \langle \alpha^{\mathbf{X}}; \beta^{\mathbf{X}}\rangle} \langle \mathcal{K}^{\mathbf{X}}; \mathcal{M}^{\mathbf{X}}; \mathcal{F}^{\mathbf{X}}\rangle$$

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash o^{\mathbf{Y}} : (\mu \mathsf{a}_{\mathsf{e}}^{\mathbf{Y}}.\mathcal{K}_{\mathsf{e}}^{\mathbf{Y}}\ @\rho_{\mathsf{e}}^{\mathbf{Y}}) \backslash \Sigma^{\mathbf{Y}}, \Pi \tag{64.5}$$

$$\mathcal{K}_{\texttt{new}}^{\mathbf{Y}} = \mathcal{K}_{\mathsf{e}}^{\mathbf{Y}}\{\mathsf{a}_{\mathsf{e}}^{\mathbf{Y}} \circlearrowleft \mathcal{K}_{\mathsf{e}}^{\mathbf{Y}}\} \tag{64.6}$$

$$\rho_{\mathsf{e}}^{\mathbf{Y}} \vdash \mathcal{K}^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}}) \xRightarrow{\Sigma_{\texttt{matchs}}} \mathcal{K}_{\texttt{new}}^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}}) \tag{64.7}$$

$$\Sigma_1 = \Sigma^{\mathbf{Y}} \cup \Sigma_{\texttt{matchs}} \tag{64.8}$$

$$\tau = \mathsf{k}^{\mathbf{X}} \tag{64.9}$$

By Lem. 19, we know there must exist

$$\emptyset \vdash \mu \mathsf{a}^{\mathbf{Y}}_{\mathtt{rf}}.\mathcal{K}^{\mathbf{Y}}_{\mathtt{rf}}\ @\rho^{\mathbf{Y}}_{\mathtt{rf}} <: \mu \mathsf{a}^{\mathbf{Y}}_{\mathsf{e}}.\mathcal{K}^{\mathbf{Y}}_{\mathsf{e}}\ @\rho^{\mathbf{Y}}_{\mathsf{e}} \backslash \Sigma_{\mathtt{sub}} \tag{64.10}$$

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash o^{\mathbf{Y}} : \mu \mathsf{a}^{\mathbf{Y}}_{\mathtt{rf}}.\mathcal{K}^{\mathbf{Y}}_{\mathtt{rf}}\ @\rho^{\mathbf{Y}}_{\mathtt{rf}} \backslash \Sigma^{\mathbf{Y}}_{\mathtt{rf}}, \Pi \tag{64.11}$$

$$\Sigma^{\mathbf{Y}} \xrightarrow{\ \mathtt{algebra}\ } \Sigma^{\mathbf{Y}}_{\mathtt{rf}} \cup \Sigma_{\mathtt{sub}} \tag{64.12}$$

and the last step of the subdeduction for (**64.11**) is an instance of (T-RID), and hence by (T-RID), we know

$$rel(R, o^{\mathbf{Y}}, o^{\mathbf{X}}) = \rho = (\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2} \tag{64.13}$$

$$\Sigma^{\mathbf{Y}}_{\mathtt{rf}} = subPed(\rho, \rho^{\mathbf{Y}}_{\mathtt{rf}}) \cup \{\nu_1 \geq_{\mathtt{s}} 0\} \tag{64.14}$$

$$\Pi = \emptyset \tag{64.15}$$

$$\mathcal{H}(o^{\mathbf{Y}}) = \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; \mathcal{S}^{\mathbf{Y}} \rangle$$

$$\mathcal{C}(\mathsf{a}^{\mathbf{Y}}) = \overline{\forall \langle \alpha^{\mathbf{Y}}; \beta^{\mathbf{Y}} \rangle} \langle \mathcal{K}^{\mathbf{Y}}; \mathcal{M}^{\mathbf{Y}}; \mathcal{F}^{\mathbf{Y}} \rangle$$

$$\mathcal{K}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] = \mathcal{K}^{\mathbf{Y}}_{\mathtt{rf}} \tag{64.16}$$

$$\mathsf{a}^{\mathbf{Y}} = \mathsf{a}^{\mathbf{Y}}_{\mathtt{rf}} \tag{64.17}$$

By Lem. 22 and (**64.10**), we know

$$\emptyset \vdash \mu \mathsf{a}^{\mathbf{Y}}_{\mathtt{rf}}.\mathcal{K}^{\mathbf{Y}}_{\mathtt{rf}}\{\mathsf{a}^{\mathbf{Y}}_{\mathtt{rf}} \circlearrowleft \mathcal{K}^{\mathbf{Y}}_{\mathtt{rf}}\}\ @\rho^{\mathbf{Y}}_{\mathtt{rf}} <: \mu \mathsf{a}^{\mathbf{Y}}_{\mathsf{e}}.\mathcal{K}^{\mathbf{Y}}_{\mathsf{e}}\{\mathsf{a}^{\mathbf{Y}}_{\mathsf{e}} \circlearrowleft \mathcal{K}^{\mathbf{Y}}_{\mathsf{e}}\}\ @\rho^{\mathbf{Y}}_{\mathsf{e}} \backslash \Sigma_{\mathtt{sub}} \tag{64.18}$$

According to (**64.2**), we know $\mathcal{C} = \Psi(Ct)$. By Lem. 7, we know $\mathcal{K}^{\mathbf{Y}}\{\mathsf{a}^{\mathbf{Y}} \circlearrowleft \mathcal{K}^{\mathbf{Y}}\} = \mathcal{K}^{\mathbf{Y}}$. Given the previous fact that $\mathcal{K}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] = \mathcal{K}^{\mathbf{Y}}_{\mathtt{rf}}$ and $\mathsf{a}^{\mathbf{Y}} = \mathsf{a}^{\mathbf{Y}}_{\mathtt{rf}}$ and the fact that pedigree type variable substitution has no effect on recursive type unfolding, we can easily see $\mathcal{K}^{\mathbf{Y}}_{\mathtt{rf}}\{\mathsf{a}^{\mathbf{Y}}_{\mathtt{rf}} \circlearrowleft \mathcal{K}^{\mathbf{Y}}_{\mathtt{rf}}\} = \mathcal{K}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]\{\mathsf{a}^{\mathbf{Y}} \circlearrowleft \mathcal{K}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]\} = (\mathcal{K}^{\mathbf{Y}}\{\mathsf{a}^{\mathbf{Y}} \circlearrowleft \mathcal{K}^{\mathbf{Y}}\})[\sigma^{\mathbf{Y}}] = \mathcal{K}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] = \mathcal{K}^{\mathbf{Y}}_{\mathtt{rf}}$. Combining this and (**64.6**), judgment (**64.18**) can be rewritten as

$$\emptyset \vdash \mu \mathsf{a}^{\mathbf{Y}}_{\mathtt{rf}}.\mathcal{K}^{\mathbf{Y}}_{\mathtt{rf}}\ @\rho^{\mathbf{Y}}_{\mathtt{rf}} <: \mu \mathsf{a}^{\mathbf{Y}}_{\mathsf{e}}.\mathcal{K}^{\mathbf{Y}}_{\mathtt{new}}\ @\rho^{\mathbf{Y}}_{\mathsf{e}} \backslash \Sigma_{\mathtt{sub}} \tag{64.19}$$

By Lem. 34 and (**64.7**) and (**64.19**), we know

$$\rho^{\mathbf{Y}}_{\mathtt{rf}} \vdash \mathcal{K}^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}}) \underset{\mathtt{dc}}{\overset{\Sigma_{\mathtt{matchd}}}{\Longrightarrow}} \mathcal{K}^{\mathbf{Y}}_{\mathtt{rf}}(\mathsf{k}^{\mathbf{Y}})$$

$$\Sigma_{\mathtt{matchs}} \cup \Sigma_{\mathtt{sub}} \xrightarrow{\ \mathtt{algebra}\ } \Sigma_{\mathtt{matchd}} \tag{64.20}$$

By (**64.16**), and the trivial fact that $\mathcal{K}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}](\mathsf{k}^{\mathbf{Y}}) = \mathcal{K}^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}})[\sigma^{\mathbf{Y}}]$, we know:

$$\rho^{\mathbf{Y}}_{\mathtt{rf}} \vdash \mathcal{K}^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}}) \underset{\mathtt{dc}}{\overset{\Sigma_{\mathtt{matchd}}}{\Longrightarrow}} \mathcal{K}^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}})[\sigma^{\mathbf{Y}}]$$

By Lem. 35, we know

$$\rho \vdash \mathcal{K}^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}}) \xRightarrow[\mathtt{dc}]{\Sigma_{\mathtt{matchd}} \cup subPed(\rho, \rho^{\mathbf{Y}}_{\mathtt{rf}})} \mathcal{K}^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}})[\sigma^{\mathbf{Y}}] \tag{64.21}$$

According to (**64.3**) and (WFT-Heap), we know $TV(\mathrm{Ran}(\sigma^{\mathbf{X}})) \cap TV(\langle \mathcal{K}^{\mathbf{X}}; \mathcal{M}^{\mathbf{X}}; \mathcal{F}^{\mathbf{X}} \rangle) = \emptyset$. Thus it is obvious that

$$TV(\mathrm{Ran}(\sigma^{\mathbf{X}})) \cap TV(\mathcal{K}^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}})) = \emptyset \tag{64.22}$$

Again by (**64.3**) and (WFT-Heap), we $TV(\mathrm{Ran}(\sigma^{\mathbf{X}})) \cap TV(\mathrm{Ran}(\sigma^{\mathbf{Y}})) = \emptyset$. By (WFT-Heap) and Lem. 8 (closed quantification), we know $TV(\mathrm{Dom}(\sigma^{\mathbf{Y}})) = TV(\langle \mathcal{K}^{\mathbf{Y}}; \mathcal{M}^{\mathbf{Y}}; \mathcal{F}^{\mathbf{Y}} \rangle)$. By definition of substitution, we know

$$TV(\mathrm{Ran}(\sigma^{\mathbf{Y}})) = TV(\langle \mathcal{K}^{\mathbf{Y}}; \mathcal{M}^{\mathbf{Y}}; \mathcal{F}^{\mathbf{Y}} \rangle [\sigma^{\mathbf{Y}}]) \tag{64.23}$$

Thus we know $TV(\mathrm{Ran}(\sigma^{\mathbf{X}})) \cap TV(\langle \mathcal{K}^{\mathbf{Y}}; \mathcal{M}^{\mathbf{Y}}; \mathcal{F}^{\mathbf{Y}} \rangle [\sigma^{\mathbf{Y}}]) = \emptyset$. Therefore it is obvious that

$$TV(\mathrm{Ran}(\sigma^{\mathbf{X}})) \cap TV(\mathcal{K}^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}})[\sigma^{\mathbf{Y}}]) = \emptyset \tag{64.24}$$

By (Def-ConnectorMatchD), we know $TV(\Sigma_{\mathtt{matchd}}) = TV(\mathcal{K}^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}})) \cup TV(\mathcal{K}^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}})[\sigma^{\mathbf{Y}}])$. By (**64.22**) and (**64.24**), it thus follows that

$$TV(\mathrm{Ran}(\sigma^{\mathbf{X}})) \cap TV(\Sigma_{\mathtt{matchd}}) = \emptyset \tag{64.25}$$

By (**64.3**) and (WFT-Heap), we know $TV(\mathrm{Dom}(\sigma^{\mathbf{X}})) \cap TV(\mathrm{Ran}(\sigma^{\mathbf{Y}})) = \emptyset$. By (**64.23**), we hence know

$$TV(\mathrm{Dom}(\sigma^{\mathbf{X}})) \cap TV(\mathcal{K}^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}})[\sigma^{\mathbf{Y}}]) = \emptyset \tag{64.26}$$

By (**64.3**) and (WFT-Heap), we know $\vdash_{\mathtt{wftSubst}} \sigma^{\mathbf{X}}$. By (**64.21**), (**64.25**), (**64.26**), we can use Lem. 36 (consistent matching in presence of substitution), so that:

$$\rho \vdash \mathcal{K}^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}})[\sigma^{\mathbf{X}}] \xRightarrow[\mathtt{dc}]{\Sigma_{\mathtt{match}}} \mathcal{K}^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}})[\sigma^{\mathbf{Y}}] \tag{64.27}$$

$$\Sigma_{\mathtt{match}} = \Sigma_{\mathtt{matchd}}[\sigma^{\mathbf{X}}] \cup subPed(\rho, \rho^{\mathbf{Y}}_{\mathtt{rf}})[\sigma^{\mathbf{X}}] \tag{64.28}$$

Subgoal: proving $G, \mathcal{C} \vdash_{\mathtt{iconf}} \Delta_2 : \mathbf{\Delta}_2 \backslash \Sigma_{\mathtt{iconf2}}$: First by (**64.3**) and (T-InstanceConfig), we know

$$\mathcal{C} \vdash_{\mathtt{wftI}} \langle \mathcal{H}; W_1; R \rangle \backslash \Sigma_{\mathtt{wfti1}} \tag{64.29}$$

$$G, \mathcal{C}, \langle \mathcal{H}; W_1; R\rangle \vdash_{\mathtt{h}} H : \mathcal{H} \backslash \Sigma_{\mathtt{h}} \tag{64.30}$$
$$\Sigma_{\mathtt{iconf1}} = \Sigma_{\mathtt{wfti1}} \cup \Sigma_{\mathtt{h}} \tag{64.31}$$

By (**64.29**) and (WFT-InstanceConfig), we know $\mathcal{C}, \mathcal{H}, R \vdash_{\mathtt{wftItr}} W_1 \backslash \Sigma_{\mathtt{wfti1}}$. By (**64.13**), (**64.27**), and (WFT-InstanceItr), we know $\mathcal{C}, \mathcal{H}, R \vdash_{\mathtt{wftItr}} c \mapsto \langle o^{\mathbf{X}}; \mathsf{k}^{\mathbf{X}}; o^{\mathbf{Y}}; \mathsf{k}^{\mathbf{Y}}\rangle \backslash \Sigma_{\mathtt{match}}$. Observe c is fresh, and hence the following $\uplus$ is indeed defined according to (WFT-InstanceItr):

$$\mathcal{C}, \mathcal{H}, R \vdash_{\mathtt{wftItr}} W_1 \uplus (c \mapsto \langle o^{\mathbf{X}}; \mathsf{k}^{\mathbf{X}}; o^{\mathbf{Y}}; \mathsf{k}^{\mathbf{Y}}\rangle) \backslash \Sigma_{\mathtt{match}} \cup \Sigma_{\mathtt{wfti1}}$$

By (WFT-InstanceConfig) we know

$$\mathcal{C} \vdash_{\mathtt{wftI}} \mathbf{\Delta}_2 \backslash \Sigma_{\mathtt{wfti2}} \tag{64.32}$$
$$\Sigma_{\mathtt{wfti2}} = \Sigma_{\mathtt{match}} \cup \Sigma_{\mathtt{wfti1}} \tag{64.33}$$

With the definition of $\preceq_{\mathtt{iconf}}$ we know $\mathbf{\Delta}_1 = \langle \mathcal{H}; W_1; R\rangle \preceq_{\mathtt{iconf}} \langle \mathcal{H}; W_2; R\rangle = \mathbf{\Delta}_2$. By Lem. 51 and (**64.30**), we know $G, \mathcal{C}, \mathbf{\Delta}_2 \vdash_{\mathtt{h}} H : \mathcal{H} \backslash \Sigma_{\mathtt{h}}$. With this, together with (**64.31**), (**64.32**), (**64.33**), and by the rule of (T-InstanceConfig), we know $G, \mathcal{C} \vdash_{\mathtt{iconf}} \Delta_2 : \mathbf{\Delta}_2 \backslash \Sigma_{\mathtt{iconf2}}$ where

$$\Sigma_{\mathtt{iconf2}} = \Sigma_{\mathtt{iconf1}} \cup \Sigma_{\mathtt{match}} \tag{64.34}$$

Subgoal: expression typing: By (T-CID), we know $[\mathbf{iconf} : \mathbf{\Delta}_2, \mathbf{me} : \mathsf{a}^{\mathbf{X}}, \mathbf{currentRt} : o^{\mathbf{X}}], \mathcal{C} \vdash c^{\mathtt{A}} : \mathsf{k}^{\mathbf{X}} \backslash \emptyset, \emptyset$. By (**64.9**) we know $\tau = \mathsf{k}^{\mathbf{X}}$. Earlier we also know $\Pi = \emptyset$. The above judgment is therefore the conclusion, and we know $\Sigma_2 = \emptyset$. The rest of the proof is merely to prove $\Sigma_{\mathtt{iconf1}} \cup \Sigma_1[\sigma^{\mathbf{X}}] \xrightarrow{\mathtt{algebra}} \Sigma_{\mathtt{iconf2}} \cup \Sigma_2[\sigma^{\mathbf{X}}]$. It holds because

$$\begin{aligned}
&\Sigma_{\mathtt{iconf1}} \cup \Sigma_1[\sigma^{\mathbf{X}}] \\
&\quad \text{by Lem. 60, the definition of substitution} \\
=&(\Sigma_{\mathtt{iconf1}} \cup \Sigma_1)[\sigma^{\mathbf{X}}] \\
&\quad \text{by (64.8)} \\
=&(\Sigma_{\mathtt{iconf1}} \cup \Sigma_{\mathtt{matchs}} \cup \Sigma^{\mathbf{Y}})[\sigma^{\mathbf{X}}] \\
&\quad \text{by (64.12) , (64.14), (ICS-union)} \\
\xrightarrow{\mathtt{algebra}}&(\Sigma_{\mathtt{iconf1}} \cup \Sigma_{\mathtt{matchs}} \cup \Sigma_{\mathtt{sub}} \cup subPed(\rho, \rho^{\mathbf{Y}}_{\mathtt{rf}}))[\sigma^{\mathbf{X}}] \\
&\quad \text{by (64.20), (ICS-union)} \\
\xrightarrow{\mathtt{algebra}}&(\Sigma_{\mathtt{iconf1}} \cup \Sigma_{\mathtt{matchd}} \cup subPed(\rho, \rho^{\mathbf{Y}}_{\mathtt{rf}}))[\sigma^{\mathbf{X}}] \\
&\quad \text{by definition of substitution, Lem. 60}
\end{aligned}$$

$$\xrightarrow{\texttt{algebra}} \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{matchd}}[\sigma^{\mathbf{X}}] \cup subPed(\rho, \rho_{\texttt{rf}}^{\mathbf{Y}})[\sigma^{\mathbf{X}}]$$
by (**64.28**)
$$= \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{match}}$$
by (**64.34**), $\Sigma_2 = \emptyset$
$$= \Sigma_{\texttt{iconf2}} \cup \Sigma_2[\sigma^{\mathbf{X}}]$$

□

C.17 Main Subject Reduction Proof

Main Lemma M2 (Subject Reduction)**.** *Given*

$$\vdash \langle C; \Delta_1; o^{\mathbf{X}}; exd_1 \rangle : \tau \backslash \Sigma_1 \qquad \textbf{(M2.General1)}$$
$$\Delta_1, exd_1 \xrightarrow{C,\, o^{\mathbf{X}}} \Delta_2, exd_2 \qquad \textbf{(M2.General2)}$$
$$exd_2 \neq \mathbf{E}[\,\textbf{exception}\,] \textit{ for any } \mathbf{E} \qquad \textbf{(M2.General3)}$$

then $\vdash \langle C; \Delta_2; o^{\mathbf{X}}; exd_2 \rangle : \tau \backslash \Sigma_2$ *and* $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_2$.

Proof. By (**M2.General1**) and (T-Config), (T-Exp), we know there exists $\mathcal{C}$ such that

$$\vdash_{\texttt{G}} C : Ct$$
$$\mathcal{C} = \Psi(Ct)$$
$$G = global(C, \mathcal{C})$$
$$G, \mathcal{C} \vdash_{\texttt{iconf}} \Delta_1 : \mathbf{\Delta}_1 \backslash \Sigma_{\texttt{iconf1}} \qquad \textbf{(M2.General4)}$$
$$\mathbf{\Delta}_1 = \langle \mathcal{H}_1; W_1; R_1 \rangle$$
$$\mathcal{H}_1(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; \mathcal{S}^{\mathbf{X}} \rangle$$
$$\begin{bmatrix} \textbf{iconf} : \mathbf{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{bmatrix}, \mathcal{C} \vdash exd_1 : \tau_{\texttt{exp}} \backslash \Sigma_{\texttt{exp1}}, \Pi_{\texttt{exp1}} \qquad \textbf{(M2.General5)}$$
$$\Sigma_1 = \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp1}}) \qquad \textbf{(M2.General6)}$$
$$\tau = \tau_{\texttt{exp}}[\sigma^{\mathbf{X}}] \qquad \textbf{(M2.General7)}$$

We now perform induction on the derivation leading to (**M2.General5**), and case analysis on the last step of the derivation and the syntactical form of exd_1.

Case: (T-Sub) concludes (**M2.General5**), regardless of the form of exd_1

By (T-Sub) and (**M2.General5**),

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_1\\ \textbf{me} : \mathsf{a}^{\mathbf{X}}\\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash exd_1 : \tau'_{\mathtt{exp}} \backslash \Sigma'_{\mathtt{exp1}}, \Pi_{\mathtt{exp1}}$$

$$\emptyset \vdash \tau'_{\mathtt{exp}} <: \tau_{\mathtt{exp}} \backslash \Sigma_{\mathtt{sub}} \qquad \textbf{(M2.Sub1)}$$

$$\Sigma_{\mathtt{exp1}} = \Sigma'_{\mathtt{exp1}} \cup \Sigma_{\mathtt{sub}} \qquad \textbf{(M2.Sub2)}$$

By (T-Config), (T-Exp), (**M2.General4**), we have

$$\vdash \langle C; \Delta_1; o^{\mathbf{X}}; exd_1 \rangle : \tau'_{\mathtt{exp}}[\sigma^{\mathbf{X}}] \backslash \Sigma_{\mathtt{iconf1}} \cup \Sigma'_{\mathtt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\mathtt{exp1}})$$

By (**M2.General2**), (**M2.General3**), and induction hypothesis, we know

$$\vdash \langle C; \Delta_2; o^{\mathbf{X}}; exd_2 \rangle : \tau'_{\mathtt{exp}}[\sigma^{\mathbf{X}}] \backslash \Sigma'_2$$
$$\Sigma_{\mathtt{iconf1}} \cup \Sigma'_{\mathtt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\mathtt{exp1}}) \xrightarrow{\mathtt{algebra}} \Sigma'_2$$

According to (T-Config), we know there exists some $\boldsymbol{\Delta}_2$ such that $G, \mathcal{C} \vdash_{\mathtt{iconf}} \Delta_2 : \boldsymbol{\Delta}_2 \backslash \Sigma_{\mathtt{iconf2}}$. According to Lem. 52, given (**M2.General2**), (**M2.General4**), we immediately know $\boldsymbol{\Delta}_1 \preceq_{\mathtt{iconf}} \boldsymbol{\Delta}_2$. By the definition of $\preceq_{\mathtt{iconf}}$,we know

$$\boldsymbol{\Delta}_2 = \langle \mathcal{H}_2; W_2; R_2 \rangle$$
$$\mathcal{H}_2(o^{\mathbf{X}}) = \mathcal{H}_1(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; \mathcal{S}^{\mathbf{X}} \rangle$$

By (T-Exp), (T-Config), we know

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_2\\ \textbf{me} : \mathsf{a}^{\mathbf{X}}\\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash exd_2 : \tau'_{\mathtt{exp}} \backslash \Sigma'_{\mathtt{exp2}}, \Pi_{\mathtt{exp2}} \qquad \textbf{(M2.Sub3)}$$
$$\Sigma'_2 = \Sigma_{\mathtt{iconf2}} \cup \Sigma'_{\mathtt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\mathtt{exp2}})$$

hence

$$\Sigma_{\mathtt{iconf1}} \cup \Sigma'_{\mathtt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\mathtt{exp1}}) \xrightarrow{\mathtt{algebra}} \Sigma_{\mathtt{iconf2}} \cup \Sigma'_{\mathtt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\mathtt{exp2}}) \qquad \textbf{(M2.Sub4)}$$

By (T-Sub), (**M2.Sub3**), and (**M2.Sub1**), we have

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_2\\ \textbf{me} : \mathsf{a}^{\mathbf{X}}\\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash exd_2 : \tau_{\mathtt{exp}} \backslash \Sigma'_{\mathtt{exp2}} \cup \Sigma_{\mathtt{sub}}, \Pi_{\mathtt{exp2}}$$

By (T-Exp), (T-Config), (**M2.General7**), we have

$$\vdash \langle C; \Delta_2; o^{\mathbf{X}}; exd_2 \rangle : \tau \backslash \Sigma_2$$

$$\Sigma_2 = \Sigma_{\texttt{iconf2}} \cup (\Sigma'_{\texttt{exp2}} \cup \Sigma_{\texttt{sub}})[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp2}}) \quad \textbf{(M2.Sub5)}$$

Union both sides of (**M2.Sub4**) with $\Sigma_{\texttt{sub}}[\sigma^{\mathbf{X}}]$ and by (ICS-union), we have

$$\Sigma_{\texttt{iconf1}} \cup (\Sigma'_{\texttt{exp1}} \cup \Sigma_{\texttt{sub}})[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp1}}) \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{iconf2}} \cup (\Sigma'_{\texttt{exp2}} \cup \Sigma_{\texttt{sub}})[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp2}})$$

With (**M2.General6**), (**M2.Sub2**), we know the left side is precisely Σ_1. With (**M2.Sub5**), we know the right hand side is precisely Σ_2. Hence $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_2$.

Case: (T-Sub) does not conclude (**M2.General5**), $exd_1 = \mathbf{E}_a[\, exd_{\texttt{redex1}}\,]$ where $\mathbf{E}_a \neq \bullet$

By grammar, the expression can only be reduced via (R-Context). In this case (**M2.General5**) is

$$\begin{bmatrix} \textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{bmatrix}, \mathcal{C} \vdash \mathbf{E}_a[\, exd_{\texttt{redex1}}\,] : \tau_{\texttt{exp}} \backslash \Sigma_{\texttt{exp1}}, \Pi_{\texttt{exp1}} \quad \textbf{(M2.Context1)}$$

By Lem. 59 on typing the expression in the context hole, we immediately know

$$\begin{bmatrix} \textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{bmatrix}, \mathcal{C} \vdash exd_{\texttt{redex1}} : \tau_{\texttt{redex}} \backslash \Sigma_{\texttt{redex1}}, \Pi_{\texttt{redex1}} \quad \textbf{(M2.Context2)}$$

$$\Sigma_{\texttt{redex1}} \subseteq \Sigma_{\texttt{exp1}} \quad \textbf{(M2.Context3)}$$

$$\Pi_{\texttt{redex1}} \subseteq \Pi_{\texttt{exp1}} \quad \textbf{(M2.Context4)}$$

the last step of derivation leading to (**M2.Context2**) is not an instance of (T-Sub) **(M2.Context5)**

By (T-Exp), (T-Config), (**M2.General4**),

$$\vdash \langle C; \Delta_1; o^{\mathbf{X}}; exd_{\texttt{redex1}} \rangle : \tau_{\texttt{redex}}[\sigma^{\mathbf{X}}] \backslash (\Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{redex1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{redex1}})) \quad \textbf{(M2.Context6)}$$

Applying (R-Context) to (**M2.General2**), we know

$$\Delta_1, exd_{\texttt{redex1}} \xrightarrow{C,\, o^{\mathbf{X}}} \Delta_2, exd_{\texttt{redex2}} \quad \textbf{(M2.Context7)}$$
$$exd_2 = \mathbf{E}_a[\, exd_{\texttt{redex2}}\,]$$

By (**M2.General3**), $\mathbf{E}_a[\, exd_{\texttt{redex2}}\,] \neq \mathbf{E}[\,\textbf{exception}\,]$ for any $\mathbf{E}$. Assume $exd_{\texttt{redex2}} = \mathbf{E}_b[\,\textbf{exception}\,]$ for some $\mathbf{E}_2$. Were this true, we would have $\mathbf{E}_a[\,\mathbf{E}_b[\,\textbf{exception}\,]\,] \neq \mathbf{E}[\,\textbf{exception}\,]$ for any evaluation context $\mathbf{E}$. This obviously is not true, since we can construct $\mathbf{E}$ by replacing $\bullet$ in $\mathbf{E}_1$ with $\mathbf{E}_2$, and the result is an evaluation context by its recursive definition. Contradiction. Hence we know:

$$exd_{\texttt{redex2}} \neq \mathbf{E}[\,\textbf{exception}\,] \text{ for any } \mathbf{E} \quad \textbf{(M2.Context8)}$$

By (**M2.Context6**), (**M2.Context7**), (**M2.Context8**), and the induction hypothesis,

$$\vdash \langle C; \Delta_2; o^{\mathbf{X}}; exd_{\texttt{redex2}} \rangle : \tau_{\texttt{redex}}[\sigma^{\mathbf{X}}] \backslash \Sigma_{\texttt{redexconfig2}}$$
$$(\Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{redex1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{redex1}})) \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{redexconfig2}} \quad \textbf{(M2.Context9)}$$

According to (T-Config), we know there exists some $\mathbf{\Delta}_2$ such that

$$G, \mathcal{C} \vdash_{\texttt{iconf}} \Delta_2 : \mathbf{\Delta}_2 \backslash \Sigma_{\texttt{iconf2}} \quad \textbf{(M2.Context10)}$$

According to Lem. 52, given (**M2.General2**), (**M2.General4**), we immediately know $\mathbf{\Delta}_1 \preceq_{\texttt{iconf}} \mathbf{\Delta}_2$. By the definition of $\preceq_{\texttt{iconf}}$,we know

$$\mathbf{\Delta}_2 = \langle \mathcal{H}_2; W_2; R_2 \rangle$$
$$\mathcal{H}_2(o^{\mathbf{X}}) = \mathcal{H}_1(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; \mathcal{S}^{\mathbf{X}} \rangle$$

By (T-Exp), (T-Config), we know

$$\left[\begin{array}{l} \textbf{iconf} : \mathbf{\Delta}_2 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array} \right], \mathcal{C} \vdash exd_{\texttt{redex2}} : \tau_{\texttt{redex}} \backslash \Sigma_{\texttt{redex2}}, \Pi_{\texttt{redex2}} \quad \textbf{(M2.Context11)}$$
$$\Sigma_{\texttt{redexconfig2}} = \Sigma_{\texttt{iconf2}} \cup \Sigma_{\texttt{redex2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{redex2}})$$

With this and (**M2.Context9**), we have

$$\Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{redex1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{redex1}}) \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{iconf2}} \cup \Sigma_{\texttt{redex2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{redex2}}) \quad \textbf{(M2.Context12)}$$

Note that an important lemma Lem. 58 says that type is preserved for evaluation contexts over replacement (with the presence of configuration strengthening). Its pre-conditions are satisfied by (**M2.Context1**), (**M2.Context2**), (**M2.Context5**), (**M2.Context11**), (**M2.Context12**), and $\mathbf{\Delta}_1 \preceq_{\texttt{iconf}} \mathbf{\Delta}_2$. Thus,

$$\left[\begin{array}{l} \textbf{iconf} : \mathbf{\Delta}_2 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array} \right], \mathcal{C} \vdash \mathbf{E}_a[\, exd_{\texttt{redex2}} \,] : \tau_{\texttt{exp}} \backslash \Sigma_{\texttt{exp2}}, \Pi_{\texttt{exp2}}$$
$$\Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp1}}) \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{iconf2}} \cup \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp2}})$$

By (T-Exp), (T-Config), (**M2.General7**),

$$\vdash \langle C; \Delta_2; o^{\mathbf{X}}; \mathbf{E}_a[\, exd_{\texttt{redex2}} \,] \rangle : \tau \backslash (\Sigma_{\texttt{iconf2}} \cup \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp2}})) \quad \textbf{(M2.Context13)}$$

Thus, we know $\Sigma_2 = \Sigma_{\texttt{iconf2}} \cup \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp2}})$ and $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_2$.

Case: (T-Sub) does not conclude (**M2.General5**), $exd_1 = \textbf{create}\ Ped\ \mathsf{a}^{\mathbf{Y}}$

By (**M2.General4**), (T-InstanceConfig), (T-Heap), (T-HeapCell), (T-FieldStore), we know the correspondence between Δ_1 and $\mathbf{\Delta}_1$. Hence we have $\Delta_1 = \langle H_1; W_1; R_1\rangle$ for some H_1. The only possible way that exd_1 can be reduced is via (R-Create). By that rule,

$$\begin{aligned}
\mathcal{C}(\mathsf{a}^{\mathbf{Y}}) &= \forall\langle\alpha_1;\beta_1\rangle,\ldots,\forall\langle\alpha_n;\beta_n\rangle.\langle\mathcal{K}^{\mathbf{Y}};\mathcal{M}^{\mathbf{Y}};\mathcal{F}^{\mathbf{Y}}\rangle\\
&\alpha'_1,\ldots,\alpha'_n,\beta'_1,\ldots,\beta'_n \text{ fresh}\\
\sigma^{\mathbf{Y}} &= [\langle\alpha_1;\beta_1\rangle\mapsto\langle\alpha'_1;\beta'_1\rangle,\ldots,\langle\alpha_n;\beta_n\rangle\mapsto\langle\alpha'_n;\beta'_n\rangle]\\
\Delta_2 &= \langle H_2; W_1; R_2\rangle\\
o^{\mathbf{Y}} &\text{ fresh}\\
C(\mathsf{a}^{\mathbf{Y}}) &= \langle K^{\mathbf{Y}};M^{\mathbf{Y}};F^{\mathbf{Y}}\rangle\\
H_2 &= H_1 \uplus (o^{\mathbf{Y}}\mapsto\langle\mathsf{a}^{\mathbf{Y}};\sigma^{\mathbf{Y}};S^{\mathbf{Y}}\rangle\\
S^{\mathbf{Y}} &= \biguplus_{\mathsf{f}\in\mathrm{Dom}(F^{\mathbf{Y}})}(\mathsf{f}\mapsto\textbf{null})\\
R_2 &= R_1\uplus(\langle o^{\mathbf{Y}};o^{\mathbf{X}}\rangle\mapsto\rho)\\
\rho &= \begin{cases}(\textbf{parent})^{\alpha}(\textbf{child}) & \text{if } Ped=[\,],\alpha \text{ fresh}\\ Ped & \text{if } Ped=(\textbf{parent})^{\mathsf{w}}(\textbf{child})\end{cases}
\end{aligned}$$

By (**M2.General5**) and (T-Create), we know

$$\begin{aligned}
\tau_{\mathtt{exp}} &= \mu\mathsf{a}^{\mathbf{Y}}.\mathcal{K}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}]\ @\rho \qquad\qquad \textbf{(M2.Create1)}\\
\Pi_{\mathtt{exp1}} &= \{\langle\mathsf{a}^{\mathbf{Y}};\rho;\sigma^{\mathbf{Y}}\rangle\}
\end{aligned}$$

By the definition of $lazyCons$, we know $G,[\,]\vdash_{\mathtt{cons}}\langle\mathsf{a}^{\mathbf{Y}};\rho;\sigma^{\mathbf{Y}}\rangle\backslash lazyCons(G,\Pi_{\mathtt{exp1}})$. By (T-Merge) we know

$$\begin{aligned}
G(\mathsf{a}^{\mathbf{Y}}) &= \langle\Sigma^{\mathbf{Y}};\Pi^{\mathbf{Y}}\rangle\\
\Sigma^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \subseteq lazyCons(G,\Pi_{\mathtt{exp1}}) &\subseteq \Sigma_1 \qquad\qquad \textbf{(M2.Create2)}
\end{aligned}$$

Let $\mathbf{\Delta}_2 = \langle\mathcal{H}_2; W_1; R_2\rangle$, where $\mathcal{H}_2 = \mathcal{H}_1\uplus(o^{\mathbf{Y}}\mapsto\langle\mathsf{a}^{\mathbf{Y}};\sigma^{\mathbf{Y}};\mathcal{S}^{\mathbf{Y}}\rangle$ and $\mathcal{S}^{\mathbf{Y}} = \biguplus_{\mathsf{f}\in\mathrm{Dom}(\mathcal{F}^{\mathbf{Y}})}(\mathsf{f}\mapsto\bot)$.

Subgoal: Proving $\mathcal{C}\vdash_{\mathtt{wftI}}\mathbf{\Delta}_2\backslash\Sigma_{\mathtt{wfti2}}$ and $\Sigma_{\mathtt{wfti2}}\subseteq\Sigma_1$

By (WFT-InstanceConfig), we know this proof can be divided into three parts. First, we need to show $\mathcal{C},R_2\vdash_{\mathtt{wftH}}\mathcal{H}_2$. Earlier we know (**M2.General4**) holds, and hence $\mathcal{C},R_1\vdash_{\mathtt{wftH}}\mathcal{H}_1$ holds. The only difference between $\mathcal{H}_2$ and $\mathcal{H}_1$ is the former has one more cell indexed by $o^{\mathbf{Y}}$. By (WFT-Heap), we know it suffices if we can prove the following

- $TV(\mathrm{Ran}(\sigma^{\mathbf{Y}}))\cap TV(R_2)=\emptyset$; This holds since $\alpha'_1,\ldots,\alpha'_n,\beta'_1,\ldots,\beta'_n$ are all fresh.

- $TV(\mathrm{Ran}(\sigma^{\mathbf{Y}})) \cap TV(\mathcal{C}) = \emptyset$. Same as above.
- $\mathrm{Dom}(\sigma^{\mathbf{Y}}) = \{\langle \alpha_1; \beta_1 \rangle, \ldots \langle \alpha_n; \beta_n \rangle\}$. By definition.
- $\forall \mathsf{f} \in \mathrm{Dom}(\mathcal{S}^{\mathbf{Y}})$ so that $\emptyset \vdash \bot <: \mathcal{S}^{\mathbf{Y}}(\mathsf{f}) \backslash \emptyset$. Trivially holds according to (Sub-Bottom).
- $\vdash_{\texttt{wftSubst}} \sigma^{\mathbf{Y}}$. Trivially holds according to (WFT-Subst).
- For any two distinct o_1 and o_2, $\mathcal{H}_2(o_1) = \langle \mathsf{a}_1; \sigma_1; \mathcal{S}_1 \rangle$ and $\mathcal{H}_2(o_2) = \langle \mathsf{a}_2; \sigma_2; \mathcal{S}_2 \rangle$, and $TV(\mathrm{Ran}(\sigma_1)) \cap TV(\mathrm{Ran}(\sigma_2)) = \emptyset$. If either o_1 or o_2 is $o^{\mathbf{Y}}$, then we know it must hold because $\alpha'_1, \ldots, \alpha'_n, \beta'_1, \ldots, \beta'_n$ are all fresh. If neither of them is $o^{\mathbf{Y}}$, it means both of them are in $\mathrm{Dom}(\mathcal{H}_1)$. The condition holds because $\mathcal{C}, R_1 \vdash_{\texttt{wftH}} \mathcal{H}_1$ holds.

Second, we need to show $\mathcal{C}, \mathcal{H}_2, R_2 \vdash_{\texttt{wftItr}} W_1 \backslash \Sigma_{\texttt{wfti2}}$ and $\Sigma_{\texttt{wfti2}} \subseteq \Sigma_1$. By (WFT-InstanceItr), we know this can be proved trivially since $\mathcal{C}, \mathcal{H}_1, R_1 \vdash_{\texttt{wftItr}} W_1 \backslash \Sigma_{\texttt{wfti1}}$ according to (**M2.General4**). The underlying reason is that the newly added entry in $\mathcal{H}_2$ and R_2 does not affect the typing of all previously existing interactions. In fact, we also know $\Sigma_{\texttt{wfti1}} = \Sigma_{\texttt{wfti2}}$. According to (**M2.General4**), we know $\Sigma_{\texttt{wfti1}} \subseteq \Sigma_{\texttt{iconf1}} \subseteq \Sigma_1$. Hence $\Sigma_{\texttt{wfti2}} \subseteq \Sigma_1$.

Third, we need to show $\mathcal{C}, \mathcal{H}_2 \vdash_{\texttt{wftRel}} R_2$. By (**M2.General4**), we know $\mathcal{C}, \mathcal{H}_1 \vdash_{\texttt{wftRel}} R_1$ holds. Thus we know R_1 is an instantiation tree, $graphRep(R_1) = \langle VER_1; EDGE_1; lab_1 \rangle$, $VER_1 = \mathrm{Dom}(\mathcal{H}_1)$. The change from R_1 to R_2 is simply to add a new node $o^{\mathbf{Y}}$ and a new edge from $o^{\mathbf{Y}}$ to $o^{\mathbf{X}}$ where $o^{\mathbf{X}} \in \mathrm{Dom}(\mathcal{H}_1)$. Since $o^{\mathbf{Y}}$ is fresh and the edge from $o^{\mathbf{Y}}$ to $o^{\mathbf{X}}$ connects the new node $o^{\mathbf{Y}}$ with the instantiation graph represented by R_1, the graph represented by R_2 is still a connected, acyclic, instantiation graph. Since R_1 is an instantiation tree, every node in VER_1 has an up-path to its root. Especially, $o^{\mathbf{X}}$ has an up-path to the root. Thus, there also exists an up-path from $o^{\mathbf{Y}}$ to the root, simply by preceding the up-path from $o^{\mathbf{X}}$ to the root with $o^{\mathbf{Y}}$. All together we are able to show R_2 is an instantiation tree as well. In addition, it trivially holds that given $graphRep(R_2) = \langle VER_2; EDGE_2; lab_2 \rangle$, $VER_2 = VER_1 \cup \{o^{\mathbf{Y}}\} = \mathrm{Dom}(\mathcal{H}_1) \cup \{o^{\mathbf{Y}}\} = \mathrm{Dom}(\mathcal{H}_2)$. The last pre-condition for proving $\mathcal{C}, \mathcal{H}_2 \vdash_{\texttt{wftRel}} R_2$ is $TV(\mathcal{C}) \cap TV(R_2) = \emptyset$. According to $\mathcal{C}, \mathcal{H}_1 \vdash_{\texttt{wftRel}} R_1$, we already know $TV(\mathcal{C}) \cap TV(R_1) = \emptyset$. We know

$$TV(R_2) = TV(R_1) \cup \begin{cases} \alpha & \text{if } Ped = [\,] \\ \emptyset & \text{otherwise} \end{cases}$$

In the first case note that α is fresh. Thus it is obvious that $TV(\mathcal{C}) \cap TV(R_2) = \emptyset$.

Subgoal: Proving $G, \mathcal{C}, \mathbf{\Delta}_2 \vdash_{\mathtt{h}} H_2 : \mathcal{H}_2 \backslash \Sigma_{\mathtt{h2}}$ and $\Sigma_{\mathtt{h2}} \subseteq \Sigma_1$

By (T-Heap), we first need to prove $\mathrm{Dom}(H_2) = \mathrm{Dom}(\mathcal{H}_2)$ which is obvious. Let $\mathrm{Dom}(H_2) = \mathrm{Dom}(\mathcal{H}_2) = \{o_1, \ldots, o_n\}$. The rest of the proof is to prove $\forall i \in \{1, \ldots, n\}$, $G, \mathcal{C}, \mathbf{\Delta}_2, o_i \vdash_{\mathtt{hc}} H_2(o_i) : \mathcal{H}_2(o_i) \backslash \Sigma_{(i)}$ and $\Sigma_{(i)} \subseteq \Sigma_1$. According to (**M2.General4**), (T-InstanceConfig), we already know $G, \mathcal{C}, \mathbf{\Delta}_1 \vdash_{\mathtt{h}} H_1 : \mathcal{H}_1 \backslash \Sigma_{\mathtt{h1}}$ and $\Sigma_{\mathtt{h1}} \subseteq \Sigma_1$. By the definition of $\preceq_{\mathtt{iconf}}$, we immediately know $\mathbf{\Delta}_1 \preceq_{\mathtt{iconf}} \mathbf{\Delta}_2$. By Lem. 51, we know $G, \mathcal{C}, \mathbf{\Delta}_2 \vdash_{\mathtt{h}} H_1 : \mathcal{H}_1 \backslash \Sigma_{\mathtt{h1}}$ and $\Sigma_{\mathtt{h1}} \subseteq \Sigma_1$. By (T-Heap), we know $\forall o \in \mathrm{Dom}(\mathcal{H}_1)$ $G, \mathcal{C}, \mathbf{\Delta}_2, o \vdash_{\mathtt{hc}} H_1(o) : \mathcal{H}_1(o) \backslash \Sigma$ and $\Sigma \subseteq \Sigma_1$. With this, by (T-Heap), we know the rest of the proof is to show $G, \mathcal{C}, \mathbf{\Delta}_2, o^{\mathbf{Y}} \vdash_{\mathtt{hc}} \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; S^{\mathbf{Y}} \rangle : \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; \mathcal{S}^{\mathbf{Y}} \rangle \backslash \Sigma^{\mathbf{Y}}_{\mathtt{hc}}$ and $\Sigma^{\mathbf{Y}}_{\mathtt{hc}} \subseteq \Sigma_1$. According to (T-HeapCell), the only non-trivial facts to prove are

$$\mathcal{C}, \mathbf{\Delta}_2, o^{\mathbf{Y}} \vdash_{\mathtt{f}} S^{\mathbf{Y}} : \mathcal{S}^{\mathbf{Y}} \backslash \Sigma_{\mathtt{f}} \qquad \textbf{(M2.Create3)}$$

$$\Sigma^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \cup \mathit{lazyCons}(G, \Pi^{\mathbf{Y}}) \cup \Sigma_{\mathtt{f}} \subseteq \Sigma_1 \qquad \textbf{(M2.Create4)}$$

We now prove (**M2.Create3**). According to (T-FieldStore), we first need to show $\mathrm{Dom}(S^{\mathbf{Y}}) = \mathrm{Dom}(\mathcal{S}^{\mathbf{Y}})$. This can be demonstrated by $\vdash_{\mathtt{G}} C : Ct$ and (T-Global) and (T-Classages), which lead to $\mathrm{Dom}(F^{\mathbf{Y}}) = \mathrm{Dom}(\mathcal{F}^{\mathbf{Y}})$. The judgment then holds since **null** $\in \mathbb{V}$ and $[\textbf{iconf} : \mathbf{\Delta}_2, \textbf{currentRt} : o \vdash \textbf{null} : \bot \backslash \emptyset, \emptyset$ for any o according to (T-Null). With this proof, we also know that $\Sigma_{\mathtt{f}} = \emptyset$. To prove (**M2.Create4**), note that according to (**M2.Create2**), we already know $\Sigma^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \subseteq \Sigma_1$. Together with the fact that $\Sigma_{\mathtt{f}} = \emptyset$, the equation holds if we can show $\mathit{lazyCons}(G, \Pi^{\mathbf{Y}}) \subseteq \mathit{lazyCons}(G, \Pi_{\mathtt{exp1}})$. This is indeed true according to the definition of $\mathit{lazyCons}$ and (T-Merge).

Reaching the Conclusion: With the previous two subgoals reached, by (T-InstanceConfig), we know

$$G, \mathcal{C} \vdash_{\mathtt{iconf}} \Delta_2 : \mathbf{\Delta}_2 \backslash \Sigma_{\mathtt{iconf2}} \text{ and } \Sigma_{\mathtt{iconf2}} \subseteq \Sigma_1 \qquad \textbf{(M2.Create5)}$$

For the rest of the proof, by definition, we know $\mathit{relOne}(R_2, o^{\mathbf{Y}}, o^{\mathbf{X}}) = \rho$, and hence $\mathit{rel}(R_2, o^{\mathbf{Y}}, o^{\mathbf{X}}) = \rho, \emptyset$. By assumption, we know $\rho = (\textbf{parent})^{\alpha}(\textbf{child})$ or $\rho = (\textbf{parent})^{\mathsf{w}}(\textbf{child})$. In the first case let $\Sigma_{\mathtt{exp2}} = \{\alpha \geq_{\mathtt{s}} 0\} \cup \{wf(1)\}$. In the second case, let $\Sigma_{\mathtt{exp2}} = \{\mathsf{w} \geq_{\mathtt{s}} 0\} \cup \{wf(1)\}$. By (T-RID), and the previous result on $\tau_{\mathtt{exp}}$ in (**M2.Create1**), we know

$$\left[\begin{array}{l}\textbf{iconf} : \mathbf{\Delta}_2 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash o^{\mathbf{Y}} : \tau_{\texttt{exp}} \backslash \Sigma_{\texttt{exp2}}, \emptyset$$

According to Lem. 52, given (**M2.General4**), (**M2.Create5**), we immediately know $\mathbf{\Delta}_1 \preceq_{\texttt{iconf}} \mathbf{\Delta}_2$. By the definition of $\preceq_{\texttt{iconf}}$,we know $\mathcal{H}_2(o^{\mathbf{X}}) = \mathcal{H}_1(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; \mathcal{S}^{\mathbf{X}} \rangle$. By (T-Config), (T-Exp), (T-InstanceConfig),

$$\vdash \langle C; \Delta_2; o^{\mathbf{X}}; o^{\mathbf{Y}} \rangle : \tau \backslash \Sigma_2$$
$$\Sigma_2 = \Sigma_{\texttt{iconf2}} \cup \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \emptyset)$$

By (ECS-Tautology), we know $\Sigma_{\texttt{exp2}} \equiv \emptyset$. By the definition of $lazyCons$, we know $lazyCons(G, \emptyset) = \emptyset$. By (**M2.Create5**), we know $\Sigma_{\texttt{iconf2}} \subseteq \Sigma_1$. Hence $\Sigma_2 \subseteq \Sigma_1$. Thus $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_2$ by (ICS-relaxation).

Case: (T-Sub) does not conclude (**M2.General5**), $exd_1 =$ **connect** $o^{\mathbf{Y}}$ **with** $\mathsf{k}^{\mathbf{X}}$ >> $\mathsf{k}^{\mathbf{Y}}$

By grammar, there is only one possible way that exd_1 can be reduced. The case is proved in Lem. 64.

Case: (T-Sub) does not conclude (**M2.General5**), $exd_1 = c^{side}$ ->$\mathsf{m}(v)$

By grammar, the expression can only be reduced via (R-HandleM), and the last step of the derivation is an instance of (T-HandleM). In this case, we know $\Delta_2 = \Delta_1$. The rule of (R-HandleM) is defined with two subcases, which we now elaborate. By (**M2.General4**), (T-InstanceConfig), (T-Heap), (T-HeapCell), (T-FieldStore), we know the correspondence between Δ_1 and $\mathbf{\Delta}_1$. Hence we have $\Delta_1 = \langle H_1; W_1; R_1 \rangle$ for some H_1, and $H_1(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; S^{\mathbf{X}} \rangle$ for some $S^{\mathbf{X}}$. Given

$$W_1(c) = \begin{cases} \langle o^{\mathbf{X}}; \mathsf{k}^{\mathbf{X}}; o^{\mathbf{Y}}; \mathsf{k}^{\mathbf{Y}} \rangle & \text{if } side = \mathtt{A} \\ \langle o^{\mathbf{Y}}; \mathsf{k}^{\mathbf{Y}}; o^{\mathbf{X}}; \mathsf{k}^{\mathbf{X}} \rangle & \text{if } side = \mathtt{P} \end{cases}$$
$$H_1(o^{\mathbf{Y}}) = \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; S^{\mathbf{Y}} \rangle$$
$$C(\mathsf{a}^{\mathbf{X}}) = \langle K^{\mathbf{X}}; M^{\mathbf{X}}; F^{\mathbf{X}} \rangle$$
$$C(\mathsf{a}^{\mathbf{Y}}) = \langle K^{\mathbf{Y}}; M^{\mathbf{Y}}; F^{\mathbf{Y}} \rangle$$
$$K^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}}) = \langle I^{\mathbf{X}}; E^{\mathbf{X}} \rangle$$
$$K^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}}) = \langle I^{\mathbf{Y}}; E^{\mathbf{Y}} \rangle$$

the two cases are $\mathsf{m} \in \mathrm{Dom}(E^{\mathbf{X}})$ and $\mathsf{m} \in \mathrm{Dom}(I^{\mathbf{X}})$. For the first case, by Lem. 62, we know

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash exd_2 : \tau_{\texttt{exp}} \backslash \Sigma_{\texttt{exp2}}, \Pi_{\texttt{exp2}}$$

$$\Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{iconf1}} \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp2}} \quad \textbf{(M2.HandleM1)}$$

By (T-Config), (T-Exp), (**M2.General7**), we know

$$\vdash \langle C; \Delta_1; o^{\mathbf{X}}; exd_2 \rangle : \tau \backslash \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp2}})$$

and

$$\begin{array}{l} \Sigma_1 \\ \text{By (\textbf{M2.General6})} \\ = \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp1}}) \\ \text{By (ICS-relaxation)} \\ \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}] \\ \text{By (\textbf{M2.HandleM1})} \\ \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp2}}) \end{array}$$

For the second and the third case, by Lem. 63, we know

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash exd_2 : \tau_{\texttt{exp}} \backslash \Sigma_{\texttt{exp2}}, \Pi_{\texttt{exp1}}$$

$$\Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{iconf1}} \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{exp2}} \quad \textbf{(M2.HandleM2)}$$

By (T-Config), (T-Exp), (**M2.General7**), we know

$$\vdash \langle C; \Delta_1; o^{\mathbf{X}}; exd_2 \rangle : \tau \backslash \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp1}})$$

and

$$\begin{array}{l} \Sigma_1 \\ \text{by (\textbf{M2.General6})} \\ = \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp1}}) \\ \text{by Lem. 60, the definition of substitution} \\ = \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{iconf1}}[\sigma^{\mathbf{X}}] \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}][\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp1}}) \\ \text{by (\textbf{M2.HandleM2})} \\ \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp1}}) \end{array}$$

Case: (T-Sub) does not conclude (**M2.General5**), $exd_1 = \mathsf{f}$

The expression can be reduced in two ways, via (R-Get) or (R-Exception-Get). If the second is the case, we know $exd_2 =$ **exception**, which contradicts assumption (**M2.General3**). We now consider the first case. By (**M2.General4**), (T-InstanceConfig), (T-Heap), (T-HeapCell), (T-FieldStore), we know the correspondence between Δ_1 and $\mathbf{\Delta}_1$. Hence we have $\Delta_1 = \langle H_1; W_1; R_1\rangle$ for some H_1, and $H_1(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; S^{\mathbf{X}}\rangle$ for some $S^{\mathbf{X}}$, and

$$\left[\begin{array}{l}\textbf{iconf} : \mathbf{\Delta}_1 \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash S^{\mathbf{X}}(\mathsf{f}) : \mathcal{S}^{\mathbf{X}}(\mathsf{f})\backslash\Sigma_{\mathtt{f}}, \emptyset \qquad \textbf{(M2.Get1)}$$

$$S^{\mathbf{X}}(\mathsf{f}) \in \mathbb{V}$$

$$\Sigma_{\mathtt{f}}[\sigma^{\mathbf{X}}] \subseteq \Sigma_{\mathtt{iconf1}} \qquad \textbf{(M2.Get2)}$$

By (R-Get), we have $exd_2 = S^{\mathbf{X}}(\mathsf{f})$ and $\Delta_2 = \Delta_1$. By (**M2.General5**) and (T-Get), we know

$$\mathcal{C}(\mathsf{a}^{\mathbf{X}}) = \overline{\forall\langle\alpha;\beta\rangle}\langle\mathcal{K}^{\mathbf{X}}; \mathcal{M}^{\mathbf{X}}; \mathcal{F}^{\mathbf{X}}\rangle$$

$$\mathcal{F}^{\mathbf{X}}(\mathsf{f}) = \tau_{\mathtt{exp}} \qquad \textbf{(M2.Get3)}$$

By (**M2.General4**), (T-InstanceConfig), (WFT-InstanceConfig) and (WFT-Heap), we know

$$\emptyset \vdash \mathcal{F}^{\mathbf{X}}(\mathsf{f}) <: \mathcal{S}^{\mathbf{X}}(\mathsf{f})\backslash\emptyset \qquad \textbf{(M2.Get4)}$$

By (T-Sub), together with (**M2.Get1**), (**M2.Get3**), (**M2.Get4**), we know

$$[\textbf{iconf} : \mathbf{\Delta}_1, \textbf{currentRt} : o^{\mathbf{X}}], \mathcal{C} \vdash S^{\mathbf{X}}(\mathsf{f}) : \tau_{\mathtt{exp}}\backslash\Sigma_{\mathtt{f}}, \emptyset$$

Earlier we have shown $S^{\mathbf{X}}(\mathsf{f}) \in \mathbb{V}$. By Lem. 57, we know value typing is independent of the **me** context. Together with the earlier fact that $exd_2 = S(\mathsf{f})$, we know

$$\left[\begin{array}{l}\textbf{iconf} : \mathbf{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash exd_2 : \tau_{\mathtt{exp}}\backslash\Sigma_{\mathtt{f}}, \emptyset$$

In this case we know

$$\vdash \langle C; \Delta_1; o^{\mathbf{X}}; exd_2\rangle : \tau\backslash\Sigma_2$$

$$\Sigma_2 = \Sigma_{\mathtt{iconf1}} \cup \Sigma_{\mathtt{f}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \emptyset)$$

By (**M2.General6**), we know $\Sigma_1 = \Sigma_{\mathtt{iconf1}} \cup \Sigma_{\mathtt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\mathtt{exp1}})$. With (**M2.Get2**), we know $\Sigma_2 \subseteq \Sigma_1$. And hence $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_2$ by (ICS-relaxation).

Case: (T-Sub) does not conclude (**M2.General5**), $exd_1 = \mathsf{f} := v$

By (**M2.General4**), (T-InstanceConfig), (T-Heap), (T-HeapCell), (T-FieldStore), we know the correspondence between Δ_1 and $\mathbf{\Delta}_1$. Hence we have $\Delta_1 = \langle H_1; W_1; R_1\rangle$ for some H_1, and $H_1(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; S^{\mathbf{X}}\rangle$ for some $S^{\mathbf{X}}$. The expression of exd_1 can only be reduced via (R-Set). By that rule, $exd_2 = v$, and $\Delta_2 = \langle H_1\{o^{\mathbf{X}} \mapsto \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; S^{\mathbf{X}}_{\mathtt{new}}\}; W_1; R_1\rangle$ and $S^{\mathbf{X}}_{\mathtt{new}} = S^{\mathbf{X}}\{\mathsf{f} \mapsto v\}$. By (**M2.General5**) and (T-Set), we know

$$\mathcal{C}(\mathsf{a}^{\mathbf{X}}) = \overline{\forall\langle\alpha;\beta\rangle}\langle\mathcal{K}^{\mathbf{X}}; \mathcal{M}^{\mathbf{X}}; \mathcal{F}^{\mathbf{X}}\rangle$$

$$\mathcal{F}^{\mathbf{X}}(\mathsf{f}) = \tau_{\mathtt{exp}} \qquad \textbf{(M2.Set1)}$$

$$\left[\begin{array}{l}\textbf{iconf} : \mathbf{\Delta}_1\\ \textbf{me} : \mathsf{a}^{\mathbf{X}}\\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash exd_2 : \tau_{\mathtt{exp}}\backslash\Sigma_{\mathtt{exp1}}, \Pi_{\mathtt{exp1}} \qquad \textbf{(M2.Set2)}$$

By (**M2.General4**), (T-InstanceConfig), (WFT-InstanceConfig), (WFT-Heap), we know $\emptyset \vdash \mathcal{F}^{\mathbf{X}}(\mathsf{f}) <: \mathcal{S}^{\mathbf{X}}(\mathsf{f})\backslash\emptyset$. Earlier we know $exd_2 = v$, and $\tau_{\mathtt{exp}} = \mathcal{F}^{\mathbf{X}}(\mathsf{f})$. By Lem. 54, we know $\Pi_{\mathtt{exp1}} = \emptyset$. Thus by (T-Sub) and (**M2.Set2**), we know $[\textbf{iconf} : \mathbf{\Delta}_1, \textbf{me} : \mathsf{a}^{\mathbf{X}}, \textbf{currentRt} : o^{\mathbf{X}}], \mathcal{C} \vdash v : \mathcal{S}^{\mathbf{X}}(\mathsf{f})\backslash\Sigma_{\mathtt{exp1}}, \emptyset$. By Lem. 57, we know

$$[\textbf{iconf} : \mathbf{\Delta}_1, \textbf{currentRt} : o^{\mathbf{X}}], \mathcal{C} \vdash v : \mathcal{S}^{\mathbf{X}}(\mathsf{f})\backslash\Sigma_{\mathtt{exp1}}, \emptyset \qquad \textbf{(M2.Set3)}$$

We next prove how the old heap and the new heap correspond. First let us study the heap before reduction. Observe by (**M2.General4**), and following (T-InstanceConfig), (T-Heap), (T-HeapCell), (T-FieldStore), we know

$$G, \mathcal{C}, \mathbf{\Delta}_1, o^{\mathbf{X}} \vdash_{\mathtt{hc}} \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; S^{\mathbf{X}}\rangle : \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; \mathcal{S}^{\mathbf{X}}\rangle\backslash\Sigma_{\mathtt{hc}}$$
$$\Sigma_{\mathtt{hc}} = \Sigma_{\mathsf{g}}[\sigma^{\mathbf{X}}] \cup \mathit{lazyCons}(G, \Pi_{\mathsf{g}}) \cup (\Sigma_{(1)} \cup \cdots \cup \Sigma_{(n)})[\sigma^{\mathbf{X}}] \subseteq \Sigma_{\mathtt{iconf1}}$$
$$\mathrm{Dom}(S^{\mathbf{X}}) = \mathrm{Dom}(\mathcal{S}^{\mathbf{X}}) = \{\mathsf{f}_1, \ldots, \mathsf{f}_n\}$$
$$\forall i \in \{1, \ldots, n\}$$
$$S^{\mathbf{X}}(\mathsf{f}_i) \in \mathbb{V}$$
$$[\textbf{iconf} : \mathbf{\Delta}_1, \textbf{currentRt} : o^{\mathbf{X}}], \mathcal{C} \vdash S^{\mathbf{X}}(\mathsf{f}_i) : \mathcal{S}^{\mathbf{X}}(\mathsf{f}_i)\backslash\Sigma_i, \emptyset$$
$$\mathsf{f} = \mathsf{f}_p, \text{ for some } 1 \leq p \leq n$$

Now let us study the heap after reduction. First recall (**M2.Set3**). By (**M2.General4**), and following (T-InstanceConfig), (T-Heap), (T-HeapCell), (T-FieldStore), we know

$$G, \mathcal{C}, \mathbf{\Delta}_1, o^{\mathbf{X}} \vdash_{\mathtt{hc}} \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; S^{\mathbf{X}}_{\mathtt{new}}\rangle : \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; \mathcal{S}^{\mathbf{X}}\rangle\backslash(\Sigma_{\mathtt{hc}} - \Sigma_p[\sigma^{\mathbf{X}}]) \cup \Sigma_{\mathtt{exp1}}[\sigma^{\mathbf{X}}]$$

By the basic property of set, $(\Sigma_{\texttt{hc}} - \Sigma_p[\sigma^{\mathbf{X}}]) \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}] \subseteq \Sigma_{\texttt{hc}} \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}]$. Thus we know $G, \mathcal{C} \vdash_{\texttt{iconf}} \Delta_2 : \boldsymbol{\Delta}_1 \backslash \Sigma_{\texttt{iconf2}}$ for some $\Sigma_{\texttt{iconf2}}$ and $\Sigma_{\texttt{iconf2}} \subseteq \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}]$. By this judgment, (**M2.Set2**), and (T-Config), we know

$$\vdash \langle C; \Delta_2; o^{\mathbf{X}}; exd_2 \rangle : \tau \backslash \Sigma_2$$
$$\Sigma_2 = \Sigma_{\texttt{iconf2}} \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp1}})$$

By (**M2.General6**), we know $\Sigma_1 = \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp1}})$. Hence we know $\Sigma_2 \subseteq \Sigma_1$. And hence $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_2$ by (ICS-relaxation).

Case: (T-Sub) does not conclude (**M2.General5**), $exd_1 = \texttt{::m}(v)$

This expression can only be reduced via (R-LocalM). By that rule, we know $C(\mathsf{a}^{\mathbf{X}}) = \langle K^{\mathbf{X}}; M^{\mathbf{X}}; F^{\mathbf{X}} \rangle$, $M^{\mathbf{X}}(\mathsf{m}) = \lambda x.e^{\mathbf{X}}$, $exd_2 = e^{\mathbf{X}}\{v/x\}$, and $\Delta_2 = \Delta_1$. By $\vdash_{\texttt{G}} C : Ct$, (T-Global), (T-Classage), (T-Methods), we know

$$\mathcal{C}(\mathsf{a}^{\mathbf{X}}) = \overline{\forall.\langle\alpha;\beta\rangle}\langle \mathcal{K}^{\mathbf{X}}; \mathcal{M}^{\mathbf{X}}; \mathcal{F}^{\mathbf{X}} \rangle$$
$$\mathcal{M}^{\mathbf{X}}(\mathsf{m}) = \tau_0^{\mathbf{X}} \to \tau^{\mathbf{X}}$$
$$[\mathbf{me} : \mathsf{a}^{\mathbf{X}}, x : \tau_0^{\mathbf{X}}], \mathcal{C} \vdash e^{\mathbf{X}} : \tau^{\mathbf{X}} \backslash \Sigma_{\mathsf{e}}, \Pi_{\mathsf{e}}$$
$$G(\mathsf{a}) = \langle \Sigma_{\mathsf{g}}; \Pi_{\mathsf{g}} \rangle$$
$$\Sigma_{\mathsf{e}} \subseteq \Sigma_{\mathsf{g}} \qquad \textbf{(M2.LocalM1)}$$
$$\Pi_{\mathsf{e}} \subseteq \Pi_{\mathsf{g}} \qquad \textbf{(M2.LocalM2)}$$

By (**M2.General5**) and (T-LocalM), we have

$$\tau_{\texttt{exp}} = \tau^{\mathbf{X}} \qquad \textbf{(M2.LocalM3)}$$
$$\begin{bmatrix} \mathbf{iconf} : \boldsymbol{\Delta}_1 \\ \mathbf{me} : \mathsf{a}^{\mathbf{X}} \\ \mathbf{currentRt} : o^{\mathbf{X}} \end{bmatrix}, \mathcal{C} \vdash v : \tau_0^{\mathbf{X}} \backslash \Sigma_{\texttt{exp1}}, \Pi_{\texttt{exp1}} \qquad \textbf{(M2.LocalM4)}$$

Given the typing rules for values, we know $\Pi_{\texttt{exp1}} = \emptyset$. Now obviously $e^{\mathbf{X}} \in \mathbb{EXP}$, and by Lem. 61, we know the type of a user-defined expression is always preserved if we change **iconf**, and **currentRt**, and hence we have

$$\begin{bmatrix} \mathbf{iconf} : \boldsymbol{\Delta}_1 \\ \mathbf{me} : \mathsf{a}^{\mathbf{X}} \\ \mathbf{currentRt} : o^{\mathbf{X}} \\ x : \tau_0^{\mathbf{X}} \end{bmatrix}, \mathcal{C} \vdash e^{\mathbf{X}} : \tau^{\mathbf{X}} \backslash \Sigma_{\mathsf{e}}, \Pi_{\mathsf{e}} \qquad \textbf{(M2.LocalM5)}$$

By (**M2.LocalM4**), and (**M2.LocalM5**), and the substitution lemma (Lem. 53), we know

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash e^{\mathbf{X}}\{v/x\} : \tau^{\mathbf{X}} \backslash \Sigma_{\mathtt{e}} \cup \Sigma_{\mathtt{exp1}}, \Pi_{\mathtt{e}} \cup \Pi_{\mathtt{exp1}}$$

We also know

$$\begin{aligned}\Sigma_1 &= \Sigma_{\mathtt{iconf1}} \cup \Sigma_{\mathtt{exp1}}[\sigma^{\mathbf{X}}] \\ \Sigma_2 = \Sigma_{\mathtt{iconf1}} \cup \Sigma_{\mathtt{e}}[\sigma^{\mathbf{X}}] \cup \Sigma_{\mathtt{exp1}}[\sigma^{\mathbf{X}}] \cup \mathit{lazyCons}(G, \Pi_{\mathtt{e}}) \\ &\vdash \langle C; \Delta_1; o^{\mathbf{X}}; e^{\mathbf{X}}\{v/x\}\rangle : \tau \backslash \Sigma_2\end{aligned}$$

By (**M2.LocalM2**) and Lem. 14, we know $\mathit{lazyCons}(G, \Pi_{\mathtt{e}}) \subseteq \mathit{lazyCons}(G, \Pi_{\mathtt{g}})$. By (**M2.LocalM1**), and the definition of substitution, we know $\Sigma_{\mathtt{e}}[\sigma^{\mathbf{X}}] \subseteq \Sigma_{\mathtt{g}}[\sigma^{\mathbf{X}}]$. With (T-InstanceConfig), (T-Heap), (T-HeapCell), we know $\Sigma_{\mathtt{g}}[\sigma^{\mathbf{X}}] \subseteq \Sigma_{\mathtt{iconf1}}$, and $\mathit{lazyCons}(G, \Pi_{\mathtt{g}}) \subseteq \Sigma_{\mathtt{iconf1}}$. All together, we have

$$\begin{aligned}\Sigma_{\mathtt{e}}[\sigma^{\mathbf{X}}] &\subseteq \Sigma_{\mathtt{iconf1}} \\ \mathit{lazyCons}(G, \Pi_{\mathtt{e}}) &\subseteq \Sigma_{\mathtt{iconf1}}\end{aligned}$$

This, together with (**M2.LocalM3**) is enough to prove $\Sigma_2 \subseteq \Sigma_1$. And hence $\Sigma_1 \xrightarrow{\mathtt{algebra}} \Sigma_2$ by (ICS-relaxation).

Case: (T-Sub) does not conclude (**M2.General5**), $exd_1 = \textbf{forall}(x : \mathsf{k})\{e\}$

This expression can only be reduced via (R-Forall). By that rule, we know

$$\begin{aligned}[c_1 \mapsto \mathit{itrw}_1, \ldots, c_m \mapsto \mathit{itrw}_m] &= W \mid {}^{c \mapsto \mathit{itrw}}_{\mathit{itrw} = \langle o; \mathsf{k}; o'; \mathsf{k}'\rangle} \\ [c_{m+1} \mapsto \mathit{itrw}_{m+1}, \ldots, c_n \mapsto \mathit{itrw}_n] &= W \mid {}^{c \mapsto \mathit{itrw}}_{\mathit{itrw} = \langle o'; \mathsf{k}'; o; \mathsf{k}\rangle}\end{aligned}$$

and $\Delta_2 = \Delta_1$. Thus by (T-CID), we know for any c_i for $1 \leq i \leq m$, we have [**iconf** : $\boldsymbol{\Delta}_1$, **me** : $\mathsf{a}^{\mathbf{X}}$, **currentRt** : $o^{\mathbf{X}}$], $\mathcal{C} \vdash c^{\mathtt{A}}{}_i : \mathsf{k} \backslash \emptyset, \emptyset$. By (**M2.General5**) and (T-Forall), we know [**iconf** : $\boldsymbol{\Delta}_1$, **me** : $\mathsf{a}^{\mathbf{X}}$, **currentRt** : $o^{\mathbf{X}}$, $x : \mathsf{k}$], $\mathcal{C} \vdash e : \tau_{\mathtt{exp}} \backslash \Sigma_{\mathtt{exp1}}, \Pi_{\mathtt{exp1}}$. Thus by Lem. 53 (substitution lemma) we know

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash e\{c^{\mathtt{A}}{}_1/x\} : \tau_{\mathtt{exp}} \backslash \Sigma_{(1)}, \Pi_{(1)}$$

where $\Sigma_{(1)} \subseteq \Sigma_{\mathtt{exp1}}, \Pi_{(1)} \subseteq \Pi_{\mathtt{exp1}}$

$\ldots$

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash e\{c^{\mathtt{A}}{}_m/x\} : \tau_{\mathtt{exp}} \backslash \Sigma_{(m)}, \Pi_{(m)}$$

where $\Sigma_{(m)} \subseteq \Sigma_{\mathtt{exp1}}, \Pi_{(m)} \subseteq \Pi_{\mathtt{exp1}}$

Similarly, we know

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash e\{\mathcal{c}^{\mathtt{P}}{}_{m+1}/x\} : \tau_{\mathtt{exp}} \backslash \Sigma_{(m+1)}, \Pi_{(m+1)}$$

where $\Sigma_{(m+1)} \subseteq \Sigma_{\mathtt{exp1}}, \Pi_{(m+1)} \subseteq \Pi_{\mathtt{exp1}}$

$\ldots$

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash e\{\mathcal{c}^{\mathtt{P}}{}_{n}/x\} : \tau_{\mathtt{exp}} \backslash \Sigma_{(n)}, \Pi_{(n)}$$

where $\Sigma_{(n)} \subseteq \Sigma_{\mathtt{exp1}}, \Pi_{(n)} \subseteq \Pi_{\mathtt{exp1}}$

Hence by (T-Sequence), we know

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash e\{\mathcal{c}^{\mathtt{A}}{}_{1}/x\}; \ldots; e\{\mathcal{c}^{\mathtt{A}}{}_{m}/x\} e\{\mathcal{c}^{\mathtt{P}}{}_{m+1}/x\}; \ldots; e\{\mathcal{c}^{\mathtt{P}}{}_{n}/x\} : \tau_{\mathtt{exp}} \backslash$$

$$\Sigma_{(1)} \cup \cdots \cup \Sigma_{(n)}, \Pi_{(1)} \cup \cdots \cup \Pi_{(n)}$$

By (T-Config)

$$\vdash \langle C; \Delta_2; o^{\mathbf{X}}; e\{\mathcal{c}^{\mathtt{A}}{}_{1}/x\}; \ldots; e\{\mathcal{c}^{\mathtt{A}}{}_{m}/x\} e\{\mathcal{c}^{\mathtt{P}}{}_{m+1}/x\}; \ldots; e\{\mathcal{c}^{\mathtt{P}}{}_{n}/x\} \rangle : \tau \backslash \Sigma_2$$
$$\Sigma_2 = \Sigma_{\mathtt{iconf1}} \cup (\Sigma_{(1)} \cup \cdots \cup \Sigma_{(n)})[\sigma^{\mathbf{X}}] \cup \mathit{lazyCons}(G, (\Pi_{(1)} \cup \cdots \cup \Pi_{(n)}))$$

By (**M2.General6**), we know $\Sigma_1 = \Sigma_{\mathtt{iconf1}} \cup \Sigma_{\mathtt{exp1}}[\sigma^{\mathbf{X}}] \cup \mathit{lazyCons}(G, \Pi_{\mathtt{exp1}})$. By the definition of substitution, we know $(\Sigma_{(1)} \cup \cdots \cup \Sigma_{(n)})[\sigma^{\mathbf{X}}] \subseteq \Sigma_{\mathtt{exp1}}[\sigma^{\mathbf{X}}]$. By the basic properties of $\mathit{lazyCons}$ (Lem. 13), we know $\mathit{lazyCons}(G, (\Pi_{(1)} \cup \cdots \cup \Pi_{(n)})) \subseteq \mathit{lazyCons}(G, \Pi_{\mathtt{exp1}})$. Hence we know $\Sigma_2 \subseteq \Sigma_1$. And hence $\Sigma_1 \xrightarrow{\mathtt{algebra}} \Sigma_2$ by (ICS-relaxation).

Case: (T-Sub) does not conclude (**M2.General5**), $exd_1 = \mathbf{in}_{\mathit{side}}(o^{\mathbf{Y}}, exd_1^{\mathbf{Y}}), exd_1^{\mathbf{Y}} \notin \mathbb{V}$

By (**M2.General5**) and (T-In), we know

$$\mathcal{H}_1(o^{\mathbf{Y}}) = \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; \mathcal{S}^{\mathbf{Y}} \rangle \qquad \textbf{(M2.InE1)}$$

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}}\end{array}\right], \mathcal{C} \vdash exd_1^{\mathbf{Y}} : \tau^{\mathbf{Y}} \backslash \Sigma_1^{\mathbf{Y}}, \Pi_1^{\mathbf{Y}} \qquad \textbf{(M2.InE2)}$$

$$\rho = \begin{cases} (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b} & \text{if } \mathit{side} = \mathtt{P}, \mathit{rel}(R_1, o^{\mathbf{Y}}, o^{\mathbf{X}}) = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b} \\ (\textbf{parent})^{\nu_b}(\textbf{child})^{\nu_a} & \text{if } \mathit{side} = \mathtt{A}, \mathit{rel}(R_1, o^{\mathbf{X}}, o^{\mathbf{Y}}) = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b} \end{cases} \qquad \textbf{(M2.InE3)}$$

$$\mathit{matchd}(\tau^{\mathbf{Y}}[\sigma^{\mathbf{Y}}], \tau^{\mathbf{X}}[\sigma^{\mathbf{X}}], \rho) = \Sigma_{\mathtt{sub}} \qquad \textbf{(M2.InE4)}$$

$$TV(\tau^{\mathbf{X}}) \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{X}})) \qquad \textbf{(M2.InE5)}$$

$$TV(\tau^{\mathbf{Y}}) \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{Y}})) \qquad \textbf{(M2.InE6)}$$

$$\Sigma_{\mathtt{exp1}} = \mathit{wfT}(\tau^{\mathbf{X}})[\sigma^{\mathbf{X}}] \cup \Sigma_1^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \cup \Sigma_{\mathtt{sub}} \cup \mathit{lazyCons}(G, \Pi_1^{\mathbf{Y}})$$

$$\Pi_{\texttt{exp1}} = \emptyset$$
$$\tau_{\texttt{exp}} = \tau^{\mathbf{X}}$$

Let

$$\Sigma_1' = \Sigma_{\texttt{iconf1}} \cup \Sigma_1^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \cup lazyCons(G, \Pi_1^{\mathbf{Y}}) \quad \textbf{(M2.InE7)}$$

With (**M2.InE1**) and (**M2.InE2**), and (T-Exp), and (T-Config), we know

$$\vdash \langle C; \Delta_1; o^{\mathbf{Y}}; exd_1^{\mathbf{Y}} \rangle : \tau^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \backslash \Sigma_1' \quad \textbf{(M2.InE8)}$$

By (**M2.General2**) and the fact that $exd_1^{\mathbf{Y}} \notin \mathbb{V}$, we know there are two ways such an expression can be reduced, via (R-In-1) or (R-Exception-In). If the reduction follows (R-Exception-In), we immediately know $exd_2 =$ **exception**. This contradicts (**M2.General3**), and hence (R-In-1) is the only way of reduction that satisfies the assumption of this lemma. By (R-In-1), we immediately know $\Delta_1, exd_1^{\mathbf{Y}} \xrightarrow{C,\, o^{\mathbf{Y}}} \Delta_2, exd_2^{\mathbf{Y}}$ where $exd_2^{\mathbf{Y}} \neq$ **exception**. By induction hypothesis, we have:

$$\vdash \langle C; \Delta_2; o^{\mathbf{Y}}; exd_2^{\mathbf{Y}} \rangle : \tau^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \backslash \Sigma_2' \quad \textbf{(M2.InE9)}$$
$$\Sigma_1' \xrightarrow{\texttt{algebra}} \Sigma_2' \quad \textbf{(M2.InE10)}$$

By (**M2.InE9**) and (T-Config), we know

$$G, \mathcal{C} \vdash_{\texttt{iconf}} \Delta_2 : \mathbf{\Delta}_2 \backslash \Sigma_{\texttt{iconf2}} \quad \textbf{(M2.InE11)}$$
$$\mathbf{\Delta}_2 = \langle \mathcal{H}_2; W_2; R_2 \rangle$$

By Lem. 52, we know $\mathbf{\Delta}_1 \preceq_{\texttt{iconf}} \mathbf{\Delta}_2$, and then by the definition of $\preceq_{\texttt{iconf}}$, we immediately know

$$\mathcal{H}_2(o^{\mathbf{Y}}) = \mathcal{H}_1(o^{\mathbf{Y}}) = \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; \mathcal{S}^{\mathbf{Y}} \rangle \quad \textbf{(M2.InE12)}$$

Again by (**M2.InE9**) and (T-Config), (T-Exp), we also know

$$\begin{bmatrix} \textbf{iconf} : \mathbf{\Delta}_2 \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}} \end{bmatrix}, \mathcal{C} \vdash exd_2^{\mathbf{Y}} : \tau^{\mathbf{Y}} \backslash \Sigma_2^{\mathbf{Y}}, \Pi_2^{\mathbf{Y}} \quad \textbf{(M2.InE13)}$$
$$\Sigma_2' = \Sigma_{\texttt{iconf2}} \cup \Sigma_2^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \cup lazyCons(G, \Pi_2^{\mathbf{Y}}) \quad \textbf{(M2.InE14)}$$

By (T-In), (**M2.InE12**), (**M2.InE13**), (**M2.InE3**), (**M2.InE4**), (**M2.InE5**), (**M2.InE6**), we know

$$\Sigma_{\texttt{exp2}} = wfT(\tau^{\mathbf{X}})[\sigma^{\mathbf{X}}] \cup \Sigma_2^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \cup \Sigma_{\texttt{sub}} \cup lazyCons(G, \Pi_2^{\mathbf{Y}})$$

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_2 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}}\end{array}\right], \mathcal{C} \vdash \textbf{in}_{side}(o^{\mathbf{Y}}, exd_2^{\mathbf{Y}}) : \tau^{\mathbf{X}} \backslash \Sigma_{\texttt{exp2}}, \emptyset$$

Let $\Sigma_2 = \Sigma_{\texttt{iconf2}} \cup \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \emptyset)$. By (**M2.General7**) we know $\tau^{\mathbf{X}}[\sigma^{\mathbf{X}}] = \tau_{\texttt{exp}}[\sigma^{\mathbf{X}}] = \tau$. And by (T-Exp), and (T-Config), we know $\vdash \langle C; \Delta_2; o^{\mathbf{X}}; \textbf{in}_{side}(o^{\mathbf{Y}}, exd_2^{\mathbf{Y}})\rangle : \tau \backslash \Sigma_2$. The rest of the proof is to show $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_2$. It holds because

Σ_1

By assumption

$= \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp1}})$

By (ICS-relaxation)

$\xrightarrow{\texttt{algebra}} \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}]$

By Lem. 60

$= (\Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp1}})[\sigma^{\mathbf{X}}]$

By assumption

$= (\Sigma_{\texttt{iconf1}} \cup wfT(\tau^{\mathbf{X}})[\sigma^{\mathbf{X}}] \cup \Sigma_1^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \cup \Sigma_{\texttt{sub}} \cup lazyCons(G, \Pi_1^{\mathbf{Y}}))[\sigma^{\mathbf{X}}]$

By (**M2.InE10**), Lem. 12, (ICS-union)

$\xrightarrow{\texttt{algebra}} (\Sigma_{\texttt{iconf2}} \cup wfT(\tau^{\mathbf{X}})[\sigma^{\mathbf{X}}] \cup \Sigma_2^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \cup \Sigma_{\texttt{sub}} \cup lazyCons(G, \Pi_2^{\mathbf{Y}}))[\sigma^{\mathbf{X}}]$

By assumption

$= (\Sigma_{\texttt{iconf2}} \cup \Sigma_{\texttt{exp2}})[\sigma^{\mathbf{X}}]$

By Lem. 60

$= \Sigma_{\texttt{iconf2}} \cup \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}]$

By definition of $lazyCons$

$= \Sigma_{\texttt{iconf2}} \cup \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \emptyset)$

$= \Sigma_2$

Case: (T-Sub) does not conclude (**M2.General5**), $exd_1 = \textbf{in}_{side}(o^{\mathbf{Y}}, v)$

This expression can only be reduced via (R-In-2). By that rule, we know $exd_2 = v$, $\Delta_2 = \Delta_1$. By (**M2.General5**) and (T-In), we immediately know

$$\mathcal{H}_1(o^{\mathbf{Y}}) = \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; \mathcal{S}^{\mathbf{Y}}\rangle$$

$$\left[\begin{array}{l}\textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{Y}} \\ \textbf{currentRt} : o^{\mathbf{Y}}\end{array}\right], \mathcal{C} \vdash v : \tau^{\mathbf{Y}} \backslash \Sigma^{\mathbf{Y}}, \Pi^{\mathbf{Y}}$$

$$\rho = \begin{cases} (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b} & \text{if } side = \texttt{P}, rel(R_1, o^{\mathbf{Y}}, o^{\mathbf{X}}) = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b} \\ (\textbf{parent})^{\nu_b}(\textbf{child})^{\nu_a} & \text{if } side = \texttt{A}, rel(R_1, o^{\mathbf{X}}, o^{\mathbf{Y}}) = (\textbf{parent})^{\nu_a}(\textbf{child})^{\nu_b} \end{cases}$$

$$matchd(\tau^{\mathbf{Y}}[\sigma^{\mathbf{Y}}], \tau^{\mathbf{X}}[\sigma^{\mathbf{X}}], \rho) = \Sigma_{\texttt{sub}}$$

$$TV(\tau^{\mathbf{X}}) \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{X}}))$$

$$TV(\tau^{\mathbf{Y}}) \subseteq TV(\mathcal{C}(\mathsf{a}^{\mathbf{Y}}))$$
$$\Sigma_{\texttt{exp1}} = wfT(\tau^{\mathbf{X}})[\sigma^{\mathbf{X}}] \cup \Sigma^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \cup \Sigma_{\texttt{sub}} \cup lazyCons(G, \Pi^{\mathbf{Y}})$$
$$\Pi_{\texttt{exp1}} = \emptyset$$
$$\tau_{\texttt{exp}} = \tau^{\mathbf{X}}$$

If *side* $=$ `A`, we know the above conditions are all the pre-conditions needed for Lem. 56 for value passing. If *side* $=$ `P`, we know the above conditions are all the pre-conditions needed for Lem. 55 for value passing. In both lemmas, the following conclusions hold:

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash v : \tau^{\mathbf{X}} \backslash \Sigma_{\texttt{exp2}}, \Pi_{\texttt{exp2}}$$
$$wfT(\tau^{\mathbf{X}})[\sigma^{\mathbf{X}}] \cup \Sigma^{\mathbf{Y}}[\sigma^{\mathbf{Y}}] \cup \Sigma_{\texttt{sub}} \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}]$$
$$\Pi_{\texttt{exp2}} = \emptyset$$

By (T-Config), and the fact that $\tau^{\mathbf{X}}[\sigma^{\mathbf{X}}] = \tau_{\texttt{exp}}[\sigma^{\mathbf{X}}] = \tau$, we know $\vdash \langle C; \Delta_2; o^{\mathbf{X}}; v\rangle : \tau \backslash \Sigma_2$ where $\Sigma_2 = \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp2}})$. The rest of the proof is to show $\Sigma_1 \xrightarrow{\texttt{algebra}} \Sigma_2$. Recall that in (**M2.General6**), we know $\Sigma_1 = \Sigma_{\texttt{iconf1}} \cup \Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\texttt{exp1}})$. The implication indeed holds because the following facts and (ICS-union):

- Given $\Pi_{\texttt{exp2}} = \Pi_{\texttt{exp1}} = \emptyset$, together with the definition of $lazyCons$, we know it must hold that $lazyCons(G, \Pi_{\texttt{exp1}}) = lazyCons(G, \Pi_{\texttt{exp2}}) = \emptyset$.
- Obviously $\Sigma_{\texttt{exp1}} \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}]$ by (ICS-relaxation), (ICS-transitivity). Applying substitution on both sides and Lem. 12, $\Sigma_{\texttt{exp1}}[\sigma^{\mathbf{X}}] \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{exp2}}[\sigma^{\mathbf{X}}]$.
- $\Sigma_{\texttt{iconf1}} \xrightarrow{\texttt{algebra}} \Sigma_{\texttt{iconf1}}$ holds by (ICS-reflexivity).

Case: (T-Sub) does not conclude (**M2.General5**), $exd_1 = v; e$

This expression can only be reduced via (R-Sequence). By that rule, we know $exd_2 = e$, and $\Delta_2 = \Delta_1$. By (**M2.General5**) and (T-Sequence), we know

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash e : \tau_{\texttt{exp}} \backslash \Sigma_{\texttt{e}}, \Pi_{\texttt{e}}$$
$$\Sigma_{\texttt{e}} \subseteq \Sigma_{\texttt{exp1}}$$

$$\Pi_{\mathtt{e}} \subseteq \Pi_{\mathtt{exp1}}$$

By (T-Config)

$$\vdash \langle C; \Delta_2; o^{\mathbf{X}}; e\rangle : \tau \backslash \Sigma_2$$
$$\Sigma_2 = \Sigma_{\mathtt{iconf1}} \cup \Sigma_{\mathtt{e}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\mathtt{e}})$$

By (**M2.General6**), we know $\Sigma_1 = \Sigma_{\mathtt{iconf1}} \cup \Sigma_{\mathtt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\mathtt{exp1}})$. By the definition of substitution and the basic properties of $lazyCons$ (Lem. 13), we know $\Sigma_2 \subseteq \Sigma_1$. And hence $\Sigma_1 \xrightarrow{\mathtt{algebra}} \Sigma_2$ by (ICS-relaxation).

Case: (T-Sub) does not conclude (**M2.General5**), $exd_1 =$ **this**

This expression can only be reduced via (R-Self). By that rule, we know $exd_2 = o^{\mathbf{X}}$, and $\Delta_2 = \Delta_1$. By (**M2.General5**) and (T-Self), we know $\mathcal{C}(\mathsf{a}^{\mathbf{X}}) = \overline{\forall\langle\alpha;\beta\rangle}\langle\mathcal{K}^{\mathbf{X}}; \mathcal{M}^{\mathbf{X}}; \mathcal{F}^{\mathbf{X}}\rangle$, we know $\tau_{\mathtt{exp}} = \mu\mathsf{a}^{\mathbf{X}}.\mathcal{K}^{\mathbf{X}}$ @**self**, and $\Sigma_{\mathtt{exp1}} = \emptyset$, and $\Pi_{\mathtt{exp1}} = \emptyset$. By (T-RID), we know

$$\left[\begin{array}{l} \textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{array}\right], \mathcal{C} \vdash o^{\mathbf{X}} : \mu\mathsf{a}^{\mathbf{X}}.\mathcal{K}^{\mathbf{X}}[\sigma^{\mathbf{X}}] \text{ @}\textbf{self}\backslash subPed(\textbf{self}, \textbf{sibling}), \emptyset$$

By (T-Config), (T-Exp), we have $\vdash \langle C; \Delta_1; o^{\mathbf{X}}; o^{\mathbf{X}}\rangle : \mu\mathsf{a}^{\mathbf{X}}.\mathcal{K}^{\mathbf{X}}[\sigma^{\mathbf{X}}]$ @**self**$[\sigma^{\mathbf{X}}]\backslash\Sigma_2$, where $\Sigma_2 = \Sigma_{\mathtt{iconf1}} \cup subPed(\textbf{self}, \textbf{sibling})[\sigma^{\mathbf{X}}] \cup lazyCons(G, \emptyset)$. Observe that

$$(\mu\mathsf{a}^{\mathbf{X}}.\mathcal{K}^{\mathbf{X}}[\sigma^{\mathbf{X}}] \text{ @}\textbf{self})[\sigma^{\mathbf{X}}] = (\mu\mathsf{a}^{\mathbf{X}}.\mathcal{K}^{\mathbf{X}} \text{ @}\textbf{self})[\sigma^{\mathbf{X}}] = \tau_{\mathtt{exp}}[\sigma^{\mathbf{X}}] = \tau$$

The previous judgment thus is exactly the conclusion if we can prove $\Sigma_1 \xrightarrow{\mathtt{algebra}} \Sigma_2$, this is indeed true according to the following:

Σ_1
By (**M2.General6**)
$=\Sigma_{\mathtt{iconf1}} \cup \Sigma_{\mathtt{exp1}}[\sigma^{\mathbf{X}}] \cup lazyCons(G, \Pi_{\mathtt{exp1}})$
By the previous facts $\Sigma_{\mathtt{exp1}} = \emptyset$, $\Pi_{\mathtt{exp1}} = \emptyset$, the definition of $lazyCons$
$=\Sigma_{\mathtt{iconf1}}$
$subPed(\textbf{self}, \textbf{sibling}) = \{0 - 0 =_{\mathtt{s}} 1 - 1, 0 \leq_{\mathtt{s}} 0 \leq_{\mathtt{s}} 1 \leq_{\mathtt{s}} 1\} \equiv \emptyset$, (ECS-tautology)
$\equiv\Sigma_{\mathtt{iconf1}} \cup subPed(\textbf{self}, \textbf{sibling})[\sigma^{\mathbf{X}}]$
By the definition of $lazyCons$
$=\Sigma_{\mathtt{iconf1}} \cup subPed(\textbf{self}, \textbf{sibling})[\sigma^{\mathbf{X}}] \cup lazyCons(G, \emptyset)$
$=\Sigma_2$

□

C.18 Progress Proof

Main Lemma M3 (Progress). *If* $\vdash \langle C; \Delta_1; o^{\mathbf{X}}; exd_1 \rangle : \tau \backslash \Sigma$*, then either* $exd_1 \in \mathbb{V}$*, or their exist* Δ_2*,* exd_2 *such that* $\Delta_1, exd_1 \xrightarrow{C,\, o^{\mathbf{X}}} \Delta_2, exd_2$.

Proof. By assumption and (T-Config), (T-Exp), we know there exists $\mathcal{C}$ such that

$$\vdash_{\mathtt{G}} C : Ct \qquad \textbf{(M3.General1)}$$
$$\mathcal{C} = \Psi(Ct) \qquad \textbf{(M3.General2)}$$
$$G = global(C, \mathcal{C})$$
$$G, \mathcal{C} \vdash_{\mathtt{iconf}} \Delta_1 : \boldsymbol{\Delta}_1 \backslash \Sigma_{\mathtt{iconf1}} \qquad \textbf{(M3.General3)}$$
$$\boldsymbol{\Delta}_1 = \langle \mathcal{H}_1; W_1; R_1 \rangle$$
$$\mathcal{H}_1(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; \mathcal{S}^{\mathbf{X}} \rangle \qquad \textbf{(M3.General4)}$$
$$\begin{bmatrix} \textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{bmatrix}, \mathcal{C} \vdash exd_1 : \tau_{\mathtt{exp}} \backslash \Sigma_{\mathtt{exp1}}, \Pi_{\mathtt{exp1}} \qquad \textbf{(M3.General5)}$$

We now perform induction on the derivation leading to (**M3.General5**), and case analysis on the last step of the derivation and the syntactical form of exd_1.

Case: (T-Sub) concludes (**M3.General5**), regardless of the form of exd_1

By (T-Sub) and (**M3.General5**),

$$\begin{bmatrix} \textbf{iconf} : \boldsymbol{\Delta}_1 \\ \textbf{me} : \mathsf{a}^{\mathbf{X}} \\ \textbf{currentRt} : o^{\mathbf{X}} \end{bmatrix}, \mathcal{C} \vdash exd_1 : \tau'_{\mathtt{exp}} \backslash \Sigma'_{\mathtt{exp1}}, \Pi_{\mathtt{exp1}}$$

By (T-Config), (T-Exp), (**M3.General3**), we have $\vdash \langle C; \Delta_1; o^{\mathbf{X}}; exd_1 \rangle : \tau'_{\mathtt{exp1global}} \backslash \Sigma'_{\mathtt{exp1global}}$ for some $\tau_{\mathtt{exp1global1}}$ and $\Sigma'_{\mathtt{exp1global}}$. By induction hypothesis, the conclusion holds.

Case: (T-Sub) does not conclude (**M3.General5**), $exd_1 = \mathbf{E}_a[\, exd_{\mathtt{redexa1}}\,]$ where $\mathbf{E}_a \neq \bullet$

By Lem. 59 (Subexpression Typing) and (**M3.General5**), we know there exist some $\Sigma_{\mathtt{redexa1}}$ and $\Pi_{\mathtt{redexa1}}$, such that $[\textbf{iconf} : \boldsymbol{\Delta}_1, \textbf{me} : \mathsf{a}^{\mathbf{X}}, \textbf{currentRt} : o^{\mathbf{X}}], \mathcal{C} \vdash exd_{\mathtt{redexa1}} : \tau_{\mathtt{redexa1}} \backslash \Sigma_{\mathtt{redexa1}}, \Pi_{\mathtt{redexa1}}$. There are two possibilities, either $exd_{\mathtt{redexa1}} \in \mathbb{V}$ or $exd_{\mathtt{redexa1}} \notin \mathbb{V}$.

If $exd_{\mathtt{redexa1}} \notin \mathbb{V}$, by (T-Config), $\vdash \langle C; \Delta_1; o^{\mathbf{X}}; exd_{\mathtt{redexa1}} \rangle : \tau_{\mathtt{redexa1global}} \backslash \Sigma_{\mathtt{redexa1global}}$ for some $\tau_{\mathtt{redexa1global}}$ and $\Sigma_{\mathtt{redexa1global}}$. By induction hypothesis and the fact that $exd_{\mathtt{redexa1}} \notin \mathbb{V}$, we know there must exist some Δ_2, $exd_{\mathtt{redexa2}}$ such that $\Delta_1, exd_{\mathtt{redexa1}} \xrightarrow{C,\, o^{\mathbf{X}}} \Delta_2, exd_{\mathtt{redexa2}}$. By (R-Context), it immediately follows that $\Delta_1, exd_1 \xrightarrow{C,\, o^{\mathbf{X}}} \Delta_2, exd_2$. The conclusion holds.

If $exd_{\texttt{redexa1}} \in \mathbb{V}$, we need to perform a case analysis on the inner most part of $\mathbf{E}_a$ containing $exd_{\texttt{redexa1}}$. By assumption we know $\mathbf{E}_a \neq \bullet$. By the definition of evaluation context, we know there exists another evaluation context $\mathbf{E}_b$ such that $exd_1 = \mathbf{E}_b[\, exd_{\texttt{redexb1}}\,]$ and $exd_{\texttt{redexb1}}$ being one of the following forms: $exd_{\texttt{redexa1}}\texttt{->m}(e_1)$, or $v\texttt{->m}(exd_{\texttt{redexa1}})$, or **connect** $exd_{\texttt{redexa1}}$ **with** $\texttt{k} \texttt{ >> } \texttt{k}'$, or $\texttt{f := }exd_{\texttt{redexa1}}$, or $\texttt{::m}(exd_{\texttt{redexa1}})$, or $exd_{\texttt{redexa1}}; e$. We now elaborate on the first case, and all the other cases can be proved in the identical way. Note that by Lem. 59 (Subexpression Typing) and (**M3.General5**), we immediately know there exist some $\Sigma_{\texttt{redexb1}}$ and $\Pi_{\texttt{redexb1}}$, such that $[\textbf{iconf} : \mathbf{\Delta}_1, \textbf{me} : \texttt{a}^{\mathbf{X}}, \textbf{currentRt} : o^{\mathbf{X}}], \mathcal{C} \vdash exd_{\texttt{redexb1}} : \tau_{\texttt{redexb1}} \backslash \Sigma_{\texttt{redexb1}}, \Pi_{\texttt{redexb1}}$. By (T-Config), (T-Exp), $\vdash \langle C; \Delta_1; o^{\mathbf{X}}; exd_{\texttt{redexb1}} \rangle : \tau_{\texttt{redexb1global}} \backslash \Sigma_{\texttt{redexb1global}}$ for some $\tau_{\texttt{redexb1global}}$ and $\Sigma_{\texttt{redexb1global}}$. By induction hypothesis and the fact that $exd_{\texttt{redexb1}} \notin \mathbb{V}$, we know there must exist some Δ_2, $exd_{\texttt{redexb2}}$ such that $\Delta_1, exd_{\texttt{redexb1}} \xrightarrow{C,\, o^{\mathbf{X}}} \Delta_2, exd_{\texttt{redexb2}}$. By (R-Context), it immediately follows that $\Delta_1, exd_1 \xrightarrow{C,\, o^{\mathbf{X}}} \Delta_2, exd_2$. The conclusion holds.

Case: (T-Sub) does not conclude (**M3.General5**), $exd_1 = \textbf{create}\ Ped\ \texttt{a}^{\mathbf{Y}}$

By grammar, the expression can only be reduced via (R-Create). By (**M3.General1**), (**M3.General2**), we know $\vdash_{\texttt{G}} C : Ct$ and $\mathcal{C} = \Psi(Ct)$. From (**M3.General5**) and (T-Create), we know $\texttt{a}^{\mathbf{Y}} \in \mathrm{Dom}(\mathcal{C})$. The last non-trivial pre-condition for (R-Create) is when $Ped \neq \epsilon$, we need to guarantee its negative level is constant 1. This is indeed guaranteed by (T-Create).

Case: (T-Sub) does not conclude (**M3.General5**), $exd_1 = \textbf{connect}\ o^{\mathbf{Y}}\ \textbf{with}\ \texttt{k}^{\mathbf{X}} \texttt{ >> } \texttt{k}^{\mathbf{Y}}$

By grammar, the expression can only be reduced via (R-Connect), which always reduces.

Case: (T-Sub) does not conclude (**M3.General5**), $exd_1 = c^{side}\texttt{->m}(v)$

By grammar, the expression can only be reduced via (R-HandleM). We now prove its pre-conditions can be satisfied. By (**M3.General5**), (T-HandleM), we know $\mathcal{C}(\texttt{a}^{\mathbf{X}}) = \overline{\forall \langle \alpha; \beta \rangle} \langle \mathcal{K}^{\mathbf{X}}; \mathcal{M}^{\mathbf{X}}; \mathcal{F}^{\mathbf{X}} \rangle$, there exists $\texttt{k}^{\mathbf{X}}$ such that $\mathcal{K}^{\mathbf{X}}(\texttt{k}^{\mathbf{X}}) = \langle \mathcal{I}^{\mathbf{X}}; \mathcal{E}^{\mathbf{X}} \rangle$, $\texttt{m} \in \mathrm{Dom}(\mathcal{I}^{\mathbf{X}} \rhd \mathcal{E}^{\mathbf{X}})$, $\mathrm{Dom}(\mathcal{I}^{\mathbf{X}}) \cap \mathrm{Dom}(\mathcal{E}^{\mathbf{X}}) = \emptyset$, and $[\textbf{iconf} : \mathbf{\Delta}_1, \textbf{me} : \texttt{a}^{\mathbf{X}}, \textbf{currentRt} : o^{\mathbf{X}}], \mathcal{C} \vdash c^{side} : \texttt{k}^{\mathbf{X}} \backslash \Sigma_{\texttt{cid}}, \Pi_{\texttt{cid}}$. The only typing rule that can be applied to lead to such a judgment is (T-CID). (Note that no subtyping rules exist for connection handle types) By that rule, we know

$$W_1(c) = \begin{cases} \langle o^{\mathbf{X}}; \texttt{k}^{\mathbf{X}}; o^{\mathbf{Y}}; \texttt{k}^{\mathbf{Y}} \rangle & \text{if } side = \texttt{A} \\ \langle o^{\mathbf{Y}}; \texttt{k}^{\mathbf{Y}}; o^{\mathbf{X}}; \texttt{k}^{\mathbf{X}} \rangle & \text{if } side = \texttt{P} \end{cases}$$

By (**M3.General3**), (WFT-InstanceConfig), (WFT-InstanceItr), we know it must hold that $\mathcal{H}_1(o^{\mathbf{Y}}) = \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; \mathcal{S}^{\mathbf{Y}} \rangle$, $\mathcal{C}(\mathsf{a}^{\mathbf{Y}}) = \overline{\forall \langle \alpha; \beta \rangle}.\langle \mathcal{K}^{\mathbf{Y}}; \mathcal{M}^{\mathbf{Y}}; \mathcal{F}^{\mathbf{Y}} \rangle$, $\mathcal{K}^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}}) = \langle \mathcal{I}^{\mathbf{Y}}; \mathcal{E}^{\mathbf{Y}} \rangle$, and (Def-ConnectorMatchD), we know no matter $W_1(c)$ is in the form of $\langle o^{\mathbf{X}}; \mathsf{k}^{\mathbf{X}}; o^{\mathbf{Y}}; \mathsf{k}^{\mathbf{Y}} \rangle$ or $\langle o^{\mathbf{Y}}; \mathsf{k}^{\mathbf{Y}}; o^{\mathbf{X}}; \mathsf{k}^{\mathbf{X}} \rangle$, it always holds that $\mathrm{Dom}(\mathcal{I}^{\mathbf{X}}[\sigma^{\mathbf{X}}]) \subseteq \mathrm{Dom}(\mathcal{E}^{\mathbf{Y}}[\sigma^{\mathbf{Y}}])$. By the definition of substitution, we know it does not alter the domain of the mapping for import signatures and export signatures, and hence $\mathrm{Dom}(\mathcal{I}^{\mathbf{X}}) \subseteq \mathrm{Dom}(\mathcal{E}^{\mathbf{Y}})$. At the beginning of the proof, we have shown $\mathsf{m} \in \mathrm{Dom}(\mathcal{I}^{\mathbf{X}} \rhd \mathcal{E}^{\mathbf{X}})$. From the definition of operator $\rhd$, we know $\mathsf{m} \in \mathrm{Dom}(\mathcal{E}^{\mathbf{Y}}) \cup \mathrm{Dom}(\mathcal{E}^{\mathbf{X}})$. According to (**M3.General3**) and (T-InstanceConfig), we know $\Delta_1 = \langle H_1; W_1; R_1 \rangle$ for some H_1. By (**M3.General1**), (**M3.General2**), (T-Global), (T-Classage), (T-Connectors), (T-Methods), we know

$$
\begin{aligned}
C(\mathsf{a}^{\mathbf{X}}) &= \langle K^{\mathbf{X}}; M^{\mathbf{X}}; F^{\mathbf{X}} \rangle \\
C(\mathsf{a}^{\mathbf{Y}}) &= \langle K^{\mathbf{Y}}; M^{\mathbf{Y}}; F^{\mathbf{Y}} \rangle \\
K^{\mathbf{X}}(\mathsf{k}^{\mathbf{X}}) &= \langle I^{\mathbf{X}}; E^{\mathbf{X}} \rangle \\
K^{\mathbf{Y}}(\mathsf{k}^{\mathbf{Y}}) &= \langle I^{\mathbf{Y}}; E^{\mathbf{Y}} \rangle \\
\mathsf{m} &\in \mathrm{Dom}(E^{\mathbf{X}}) \cup \mathrm{Dom}(E^{\mathbf{Y}})
\end{aligned}
$$

By (T-Methods), we also know $E^{\mathbf{X}}(\mathsf{m})$ and $E^{\mathbf{Y}}(\mathsf{m})$ can be written in the form of $\lambda x.e$. All pre-conditions of (R-HandleM) are satisfied. The reduction thus can happen.

Case: (T-Sub) does not conclude (**M3.General5**), $exd_1 = \mathsf{f}$

By grammar, the expression can only be reduced via (R-Get) or (R-Exception-Get). We now prove either of the two rules can have their pre-conditions satisfied. According to (**M3.General3**) and (T-InstanceConfig), we know $\Delta_1 = \langle H_1; W_1; R_1 \rangle$ for some H_1. By (**M3.General4**), (T-Heap), and (T-HeapCell), we also know $H_1(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; S^{\mathbf{X}} \rangle$ is defined for some $S^{\mathbf{X}}$. Thus, either for the case of (R-Get) or for the case of (R-Exception-Get), the key is to prove $\mathsf{f} \in \mathrm{Dom}(S^{\mathbf{X}})$. It can be proved with the following facts:

- By (T-Get), we know $\mathcal{C}(\mathsf{a}^{\mathbf{X}}) = \overline{\forall \langle \alpha; \beta \rangle} \langle \mathcal{K}^{\mathbf{X}}; \mathcal{M}^{\mathbf{X}}; \mathcal{F}^{\mathbf{X}} \rangle$ and $\mathsf{f} \in \mathrm{Dom}(\mathcal{F}^{\mathbf{X}})$.
- By (**M3.General3**), (T-InstanceConfig), and (WFT-Heap), we know $\mathrm{Dom}(\mathcal{S}^{\mathbf{X}}) \subseteq \mathrm{Dom}(\mathcal{F}^{\mathbf{X}})$.
- By (**M3.General3**), (T-InstanceConfig), (T-Heap), (T-HeapCell), and (T-FieldStore), we know $\mathrm{Dom}(S^{\mathbf{X}}) = \mathrm{Dom}(\mathcal{S}^{\mathbf{X}})$.

Case: (T-Sub) does not conclude (**M3.General5**), $exd_1 = \mathsf{f} := v$

Entirely analogous to the previous case. The only difference is the expression can only be reduced via (R-Set), and the typing rule to be applied is (T-Set).

Case: (T-Sub) does not conclude (**M3.General5**), $exd_1 = :: \mathsf{m}(v)$

By grammar, the expression can only be reduced via (R-LocalM). We now prove all pre-conditions of this rule can be satisfied. According to (**M3.General3**) and (T-InstanceConfig), we know $\Delta_1 = \langle H_1; W_1; R_1 \rangle$ for some H_1. By (**M3.General4**), (T-Heap), and (T-HeapCell), we also know $H_1(o^{\mathbf{X}}) = \langle \mathsf{a}^{\mathbf{X}}; \sigma^{\mathbf{X}}; S^{\mathbf{X}} \rangle$ is defined for some $S^{\mathbf{X}}$. By (**M3.General5**), (T-LocalM), we know $\mathcal{C}(\mathsf{a}^{\mathbf{X}}) = \overline{\forall \langle \alpha; \beta; \rangle} \langle \mathcal{K}^{\mathbf{X}}; \mathcal{M}^{\mathbf{X}}; \mathcal{F}^{\mathbf{X}} \rangle$ and $\mathsf{m} \in \mathrm{Dom}(\mathcal{M}^{\mathbf{X}})$. By (**M3.General1**), (**M3.General2**), (T-Global), (T-Classage), we know $C(\mathsf{a}^{\mathbf{X}}) = \langle K^{\mathbf{X}}; M^{\mathbf{X}}; F^{\mathbf{X}} \rangle$ and $\mathrm{Dom}(\mathcal{M}^{\mathbf{X}}) = \mathrm{Dom}(M^{\mathbf{X}})$. Thus we know $\mathsf{m} \in \mathrm{Dom}(M^{\mathbf{X}})$. By (T-Methods), we also know $M^{\mathbf{X}}(\mathsf{m})$ can be written in the form of $\lambda x.e$. All pre-conditions of (R-LocalM) are thus satisfied.

Case: (T-Sub) does not conclude (**M3.General5**), $exd_1 = \textbf{forall}(x : \mathsf{k})\{e\}$

By grammar, the expression can only be reduced via (R-Forall), which always reduces.

Case: (T-Sub) does not conclude (**M3.General5**), $exd_1 = \textbf{in}_{side}(o^{\mathbf{Y}}, exd_1^{\mathbf{Y}}), exd_1^{\mathbf{Y}} \notin \mathbb{V}$

By grammar, the expression can only be reduced via (R-In-1) and (R-Exception-In). We now prove either of the two rules can have the pre-conditions satisfied. By (**M3.General5**), (T-In), we know

$$\mathcal{H}_1(o^{\mathbf{Y}}) = \langle \mathsf{a}^{\mathbf{Y}}; \sigma^{\mathbf{Y}}; \mathcal{S}^{\mathbf{Y}} \rangle$$
$$\left[\ \textbf{iconf} : \boldsymbol{\Delta}_1, \textbf{me} : \mathsf{a}^{\mathbf{Y}}, \textbf{currentRt} : o^{\mathbf{Y}} \ \right], \mathcal{C} \vdash exd_1^{\mathbf{Y}} : \tau_{\mathtt{in}} \backslash \Sigma_{\mathtt{in}}, \Pi_{\mathtt{in}}$$

for some $\tau_{\mathtt{in}}$, $\Sigma_{\mathtt{in}}$, $\Pi_{\mathtt{in}}$. Since in this case $exd_1^{\mathbf{Y}} \notin \mathbb{V}$, by induction hypothesis, we know there must exist $\boldsymbol{\Delta}_2$, $exd_2^{\mathbf{Y}}$, such that $\Delta_1, exd_1^{\mathbf{Y}} \xrightarrow{C,\, o^{\mathbf{Y}}} \Delta_2, exd_1^{\mathbf{Y}}$. Thus, if $exd_2^{\mathbf{Y}} = \textbf{exception}$, the expression reduces following (R-In-Exception). Otherwise the expression reduces following (R-In-1).

Case: (T-Sub) does not conclude (**M3.General5**), $exd_1 = \textbf{in}_{side}(o^{\mathbf{Y}}, v)$

By grammar, the expression can be reduced via (R-In-2) or (R-Exception-In). The second case in fact is not possible since its pre-condition requires that v can be reduced to **exception** which is not possible according to all reduction rules producing **exception** ((R-Exception-In) and (R-Exception-Get)). Thus, the reduction

always follows (R-In-2), which always reduces.

Case: (T-Sub) does not conclude (**M3.General5**), $exd_1 = v; e$

By grammar, the expression can only be reduced via (R-Sequence), which always reduces.

Case: (T-Sub) does not conclude (**M3.General5**), $exd_1 = \textbf{this}$

By grammar, the expression can only be reduced via (R-Self), which always reduces.

□

C.19 Proof for Theorem 1

Proof. Standard proof following subject reduction (**Main Lemma M2**) and progress (**Main Lemma M3**), together with a lemma to show the initial configuration is well-typed (**Main Lemma M1**). □

C.20 Proof for Theorem 2

We first state a corollary.

Corollary 7 (Well-Typed Attainable Configuration)**.** *If* $\vdash_{\mathtt{G}} C : Ct$ *and* $C \xrightarrow{\clubsuit} \langle \Delta; o; exd \rangle$*, then there exists* τ *and* Σ *such that* $\vdash \langle C; \Delta; o; exd \rangle : \tau \backslash \Sigma$.

Proof. This is in fact weaker than Theorem 1, following subject reduction (**Main Lemma M2**) and the fact that initial configuration is well-typed (**Main Lemma M1**). □

Followed is the proof for Theorem 2.

Proof. Let $\mathcal{C} = \Psi(Ct)$. By the definition of Ψ and the correspondence between $\mathcal{C}$ and Ct, we know for any $\mathsf{a} \in \mathrm{Dom}(Ct)$, $Ct(\mathsf{a}) = \langle Kt; Mt, Ft \rangle$, any for any $(\mathsf{f} \mapsto (\textbf{parent})^{\mathsf{w}}(\textbf{child})^{\mathsf{z}}\ \mathsf{a}') \propto Ft$, the corresponding $\mathcal{C}$ must have $\mathcal{C}(\mathsf{a}) = \overline{\langle \alpha; \beta \rangle} \langle \mathcal{K}; \mathcal{M}, \mathcal{F} \rangle$ and $\mathcal{F}(\mathsf{f}) = \mu \mathsf{a}'.\mathcal{K}'\ @(\textbf{parent})^{\mathsf{w}}(\textbf{child})^{\mathsf{z}}$ for some $\mathcal{K}'$. By Corollary 7 we know $\vdash \langle C; \Delta; o'; exd \rangle : \tau \backslash \Sigma_{\mathtt{whole}}$. Also we know $\mathcal{C} \vdash_{\mathtt{iconf}} \Delta : \boldsymbol{\Delta} \backslash \Sigma_{\mathtt{iconf}}$ for some $\boldsymbol{\Delta}$ and $\Sigma_{\mathtt{iconf}} \subseteq \Sigma_{\mathtt{whole}}$. By (T-InstanceConfig), (T-Heap), (T-HeapCell), we know $\boldsymbol{\Delta} = \langle \mathcal{H}; W; R \rangle$ for some $\mathcal{H}$ and $\mathcal{S}$ such that $\mathrm{Dom}(H) = \mathrm{Dom}(\mathcal{H})$ and $\mathcal{H}(o) = \langle \mathsf{a}; \sigma; \mathcal{S} \rangle$. In addition, all fields on the heap are known to be

well-typed. Hence, $[\textbf{iconf} : \boldsymbol{\Delta}, \textbf{currentRt} : o], \mathcal{C} \vdash S(\mathsf{f}) : \mathcal{S}(\mathsf{f})\backslash\Sigma_{\mathtt{f}}, \emptyset$ and $\Sigma_{\mathtt{f}} \subseteq \Sigma_{\mathtt{iconf}}$. By the same rule (T-InstanceConfig), we know $\mathcal{C} \vdash_{\mathtt{wftI}} \langle \mathcal{H}; W; R\rangle\backslash\Sigma_{\mathtt{wfti}}$. By (WFT-Heap), we know $\vdash \mathcal{S}(\mathsf{f}) <: \mathcal{F}(\mathsf{f})\backslash\emptyset$. Hence $[\textbf{iconf} : \boldsymbol{\Delta}, \textbf{currentRt} : o], \mathcal{C} \vdash S(\mathsf{f}) : \mathcal{F}(\mathsf{f})\backslash\Sigma_{\mathtt{f}}, \emptyset$. To lead to this judgment, by Lem. 19, we know

$$\begin{aligned}[\textbf{iconf} : \boldsymbol{\Delta}, \textbf{currentRt} : o], \mathcal{C} \vdash S(\mathsf{f}) : \tau_{\mathtt{exp}}\backslash\Sigma'_{\mathtt{f}}, \emptyset \\ \emptyset \vdash \tau_{\mathtt{exp}} <: \mathcal{F}(\mathsf{f})\backslash\Sigma_{\mathtt{sub}} \\ \Sigma_{\mathtt{f}} = \Sigma'_{\mathtt{f}} \cup \Sigma_{\mathtt{sub}}\end{aligned}$$

and the last step of the derivation leading to the judgment above is an instance of (T-RID). Let $\tau_{\mathtt{exp}} = \mu\mathsf{a}''.\mathcal{K}''\ @\rho''$. By (T-RID), we know $rel(R, S(\mathsf{f}), o) = \rho$ and $subPed(\rho, \rho'') \subseteq \Sigma'_{\mathtt{f}}$.Earlier we have shown $\mathcal{F}(\mathsf{f}) = \mu\mathsf{a}'.\mathcal{K}'\ @(\textbf{parent})^{\mathsf{w}}(\textbf{child})^{\mathsf{z}}$. Thus By (Sub-Non-Recursive), we know

$$subPed(\rho'', (\textbf{parent})^{\mathsf{w}}(\textbf{child})^{\mathsf{z}}) \subseteq \Sigma_{\mathtt{sub}}$$

By the definition of $subPed$, we know

$$subPed(\rho, \rho'') \cup subPed(\rho'', (\textbf{parent})^{\mathsf{w}}(\textbf{child})^{\mathsf{z}}) \xrightarrow{\mathtt{algebra}} subPed(\rho, (\textbf{parent})^{\mathsf{w}}(\textbf{child})^{\mathsf{z}})$$

Earlier we have shown $\Sigma_{\mathtt{f}} \subseteq \Sigma_{\mathtt{iconf}} \subseteq \Sigma_{\mathtt{whole}}$. By Main Lemma M1, we know the initial constraint set is consistent. By Main Lemma M2, we know all constraints associated with subsequent reduction steps are subset of the initial one, and therefore they are also consistent, *i.e.* $\Sigma_{\mathtt{whole}}$ is consistent. As a result, $subPed(\rho, (\textbf{parent})^{\mathsf{w}}(\textbf{child})^{\mathsf{z}}) \cup \Sigma$ is consistent.

□

C.21 Proof for Theorem 3

Proof. By Corollary 7 we know $\vdash \langle C; \Delta; o'; exd\rangle : \tau\backslash\Sigma_{\mathtt{whole}}$. Also we know $\mathcal{C} \vdash_{\mathtt{iconf}} \Delta : \boldsymbol{\Delta}\backslash\Sigma_{\mathtt{iconf}}$ for some $\boldsymbol{\Delta}$ and $\Sigma_{\mathtt{iconf}} \subseteq \Sigma_{\mathtt{whole}}$. By (T-InstanceConfig), (T-Heap), (T-HeapCell), we know $\boldsymbol{\Delta} = \langle \mathcal{H}; W; R\rangle$ for some $\mathcal{H}$ and $\mathcal{S}$ such that $\mathrm{Dom}(H) = \mathrm{Dom}(\mathcal{H})$ and $\mathcal{H}(o) = \langle \mathsf{a}; \sigma; \mathcal{S}\rangle$. In addition, all fields on the heap are known to be well-typed. Hence, $[\textbf{iconf} : \boldsymbol{\Delta}, \textbf{currentRt} : o], \mathcal{C} \vdash S(\mathsf{f}) : \mathcal{S}(\mathsf{f})\backslash\Sigma_{\mathtt{f}}, \emptyset$ and $\Sigma_{\mathtt{f}} \subseteq \Sigma_{\mathtt{iconf}}$. By the same rule (T-InstanceConfig), we know $\mathcal{C} \vdash_{\mathtt{wftI}} \langle \mathcal{H}; W; R\rangle\backslash\Sigma_{\mathtt{wfti}}$. By (WFT-Heap), we know $\vdash \mathcal{S}(\mathsf{f}) <: \mathcal{F}(\mathsf{f})\backslash\emptyset$. Hence $[\textbf{iconf} : \boldsymbol{\Delta}, \textbf{currentRt} : o], \mathcal{C} \vdash S(\mathsf{f}) : \mathcal{F}(\mathsf{f})\backslash\Sigma_{\mathtt{f}}, \emptyset$. To lead to this judgment, by Lem. 19, we know

$$[\textbf{iconf} : \boldsymbol{\Delta}, \textbf{currentRt} : o], \mathcal{C} \vdash S(\mathsf{f}) : \tau_{\mathtt{exp}} \backslash \Sigma'_{\mathtt{f}}, \emptyset$$
$$\emptyset \vdash \tau_{\mathtt{exp}} <: \mathcal{F}(\mathsf{f}) \backslash \Sigma_{\mathtt{sub}}$$
$$\Sigma_{\mathtt{f}} = \Sigma'_{\mathtt{f}} \cup \Sigma_{\mathtt{sub}}$$

and the last step of the derivation lead to the judgment above is an instance of (T-RID). First let us assume $\tau_{\mathtt{exp}} = \mu \mathsf{a}.\mathcal{K}\ @(\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2}$. By (T-RID), we know $rel(R, S(\mathsf{f}), o) = \rho$ and

$$\Sigma'_{\mathtt{f}} = subPed(\rho, (\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2}) \cup \{\nu_1 \geq_{\mathtt{s}} 0\}$$

Earlier we have shown $\Sigma_{\mathtt{f}} \subseteq \Sigma_{\mathtt{iconf}} \subseteq \Sigma_{\mathtt{whole}}$. By Main Lemma M1, we know the initial constraint set is consistent. By Main Lemma M2, we know all constraints associated with subsequent reduction steps are subset of the initial one, and therefore they are also consistent, *i.e.* $\Sigma_{\mathtt{whole}}$ is consistent. Thus $\Sigma'_{\mathtt{f}}$ is consistent. Note that by the definition of $subPed$, we trivially know $subPed(\rho, (\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2}) \xrightarrow{\mathtt{algebra}} twoVals(\nu_2)$. Hence $subPed(\rho, (\textbf{parent})^{\nu_1}(\textbf{child})^{\nu_2}) \cup \{\nu_1 \geq_{\mathtt{s}} 0\} \cup twoVals(\nu_2)$ is consistent.

□

C.22 Proof for Theorem 4

Proof. This proof resembles the one for Theorem 2. Let $\mathcal{C} = \Psi(Ct)$. By the definition of Ψ and the correspondence between $\mathcal{C}$ and Ct, we know for any $\mathsf{a} \in \mathrm{Dom}(Ct)$, $Ct(\mathsf{a}) = \langle Kt; Mt, Ft\rangle$, for any $(\mathsf{f} \mapsto \mathsf{k}) \propto Ft$, the corresponding $\mathcal{C}$ must have $\mathcal{C}(\mathsf{a}) = \overline{\langle\alpha;\beta\rangle}.\langle\mathcal{K}; \mathcal{M}, \mathcal{F}\rangle$, $\mathcal{F}(\mathsf{f}) = \mathsf{k}$ for some k. By Corollary 7 we know $\vdash \langle C; \Delta; o'; exd\rangle : \tau \backslash \Sigma_{\mathtt{whole}}$. By (T-Config), we know $\mathcal{C} \vdash_{\mathtt{iconf}} \Delta : \boldsymbol{\Delta} \backslash \Sigma_{\mathtt{iconf}}$ for some $\boldsymbol{\Delta}$. By (T-InstanceConfig), (T-Heap), (T-HeapCell), we know $\boldsymbol{\Delta} = \langle \mathcal{H}; W; R\rangle$ for some $\mathcal{H}$ and $\mathcal{S}$ such that $\mathrm{Dom}(H) = \mathrm{Dom}(\mathcal{H})$ and $\mathcal{H}(o) = \langle \mathsf{a}; \sigma; \mathcal{S}\rangle$. In addition, all fields on the heap are known to be well-typed. Hence, $[\textbf{iconf} : \boldsymbol{\Delta}, \textbf{currentRt} : o], \mathcal{C} \vdash S(\mathsf{f}) : \mathcal{S}(\mathsf{f}) \backslash \Sigma_{\mathtt{f}}, \emptyset$ and $\Sigma_{\mathtt{f}} \subseteq \Sigma_{\mathtt{iconf}}$. By the same rule (T-InstanceConfig), we know $\mathcal{C} \vdash_{\mathtt{wftI}} \langle \mathcal{H}; W; R\rangle \backslash \Sigma_{\mathtt{wfti}}$. By (WFT-Heap), we know $\vdash \mathcal{S}(\mathsf{f}) <: \mathcal{F}(\mathsf{f}) \backslash \emptyset$. Hence $[\textbf{iconf} : \boldsymbol{\Delta}, \textbf{currentRt} : o], \mathcal{C} \vdash S(\mathsf{f}) : \mathsf{k} \backslash \Sigma_{\mathtt{f}}, \emptyset$. This is the only possibility for $S(\mathsf{f}) = v = c^{side}$. To lead to this judgment, we know the last step in the derivation must be an instance of (T-CID). (Note that there is no subtyping rule available for connection handle types.) By that rule, we know the conclusion holds.

□

Bibliography

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, 1993.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[AC04] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *Proceedings of the 18th European Conference of Object-Oriented Programming (ECOOP)*, pages 1–25, 2004.

[ACG+06] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped types and aspects for real-time java. In *Proceedings of the 20th European Conference of Object-Oriented Programming (ECOOP)*, pages 124–147, 2006.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[AGBRH05] J. J. Amor, J. M. Gonzalez-Barahona, G. Robles, and I. Herraiz. Measuring libre software using debian 3.1 (sarge) as a case study: preliminary results. *Upgrade Magazine*, August 2005.

[Age96] Ole Agesen. *Concrete type inference: delivering object-oriented applications*. PhD thesis, Stanford University, Stanford, CA, USA, 1996.

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 311–330, 2002.

[All01] Roy A. Allan. *A History of the Personal Computer: The People and the Technology*. Allan Publishing, 2001.

[Alm97] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings of the 11th European Conference of Object-Oriented Programming (ECOOP)*, pages 32–59, 1997.

[ASCN03] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language support for connector abstractions. In *Proceedings of the 17th European Conference of Object-Oriented Programming (ECOOP)*, pages 74–102, 2003.

[AZ02] D. Ancona and E. Zucca. A calculus of module systems. *Journal of functional programming*, 11:91–132, 2002.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the Joint ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) and the 4th European Conference of Object-Oriented Programming (ECOOP)*, pages 303–311, 1990.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230, New York, NY, USA, 2002. ACM Press.

[BLS03] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 213–223, 2003.

[Blu69] Herbert Blumer. *Symbolic Interactionism: Perspective and Method*. Berkeley: University of California Press, 1969.

[BNR01] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of

uniqueness and read-only. In *Proceedings of the 15th European Conference of Object-Oriented Programming (ECOOP)*, pages 2–27, London, UK, 2001. Springer-Verlag.

[Boy04] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.

[BRS99] Michael Benedikt, Thomas W. Reps, and Shmuel Sagiv. A decidable logic for describing linked data structures. In *European Symposium on Programming*, pages 2–19, 1999.

[BSD03] Andrew P. Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the smalltalk collection classes. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 47–64, 2003.

[BW05] Gavin Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *FOOL'05: Proceedings of the 12th International Workshop on Foundations of Object-oriented Languages*, 2005.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988.

[CD94] Evelyne Contejean and Herve Devie. An efficient incremental algorithm for solving systems of linear diophantine equations. *Information and Computation*, 113(1):143–172, 1994.

[CD02] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310, 2002.

[CF05] Richard Cobbe and Matthias Felleisen. Environmental acquisition revisited. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 14–25, 2005.

[Cla01] Dave Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, July 2001.

[CPN98] Dave Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 48–64, 1998.

[Dah07] Howard Dahdah. Tanenbaum outlines his vision for a grandma-proof os, online at `http://www.computerworld.com.au/index.php/id;1942598204;pp;1`. January 2007.

[DBFP95] Stéphane Ducasse, Mireille Blay-Fornarino, and Anne-Marie Pinna. A reflective model for first class dependencies. In *Proceedings of the 1995 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 265–280, 1995.

[DDM07] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *Proceedings of the 21st European Conference of Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.

[dGER01] Paul du Gay, Jessica Evans, and Peter Redman, editors. *Identity: A Reader*. Sage Publications Ltd., 2001.

[DM05] W. Dietl and P. Muller. Universes: Lightweight ownership for jml. *Journal of Object Technology*, 4(8):5–32, 2005.

[DMN70] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. SIMULA - Common Base Language. Technical Report S-22, Norwegian computing center, Oslo, October 1970.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proceedings of the 1995 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–184, 1995.

[FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, pages 236–248, 1998.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 171–183, 1998.

[FM97] Pascal Fradet and Daniel Le Métayer. Shape types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 27–39, New York, NY, USA, 1997. ACM Press.

[Fou84] Michel Foucault. *The Foucault Reader (edited by Paul Rabinow)*. Pantheon, 1984.

[GAA04] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the uml cake. In *Proceedings of the 2004 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 301–314, 2004.

[GFF05] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. *SIGPLAN Not.*, 40(10):231–245, 2005.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[Hei62] Martin Heidegger. *Being and Time (translated by John Macquarrie and Edward Robinson)*. HarperOne, 1962.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, 1993.

[Hog91] John Hogg. Islands: aliasing protection in object-oriented languages. In *Proceedings of the 1991 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 271–285, New York, NY, USA, 1991. ACM Press.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[Kri94] B. Kristensen. Complex Associations: Abstractions in Object-Oriented Modeling. In *Proceedings of the 1994 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 272–283, 1994.

[KS93] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 196–205, Charleston, South Carolina, 1993.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 36–44, 1998.

[Ler00] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[LLS07] Yu David Liu, Xiaoqli Lu, and Scott F. Smith. Coqa: Concurrent objects with quantized atomicity. Technical report, The Johns Hopkins University, Baltimore, Maryland, March 2007.

[LP06] Yi Lu and John Potter. On ownership and accessibility. In *Proceedings of the 20th European Conference of Object-Oriented Programming (ECOOP)*, pages 99–123, 2006.

[LRS03] Yu David Liu, Ran Rinat, and Scott F. Smith. Component Assemblies and Component Runtimes. Technical report, The Johns Hopkins University, Baltimore, Maryland, March 2003.

[LS02] Yu David Liu and Scott F. Smith. A Component Security Infrastructure. In *Foundation of Computer Security Workshop*, 2002.

[LS04] Yu David Liu and Scott F. Smith. Modules With Interfaces for Dynamic Linking and Communication. In *Proceedings of the 18th European Conference of Object-Oriented Programming (ECOOP)*, pages 414–439, 2004.

[LS05] Yu David Liu and Scott F. Smith. Interaction-based Programming with Classages. In *Proceedings of the 2005 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 191–209, 2005.

[LS06] Yu David Liu and Scott F. Smith. A Formal Framework for Component Deployment. In *Proceedings of the 2006 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 325–344, 2006.

[LS07] Yu David Liu and Scott F. Smith. Pedigree Types. Technical report, The Johns Hopkins University, Baltimore, Maryland, July 2007.

[LST04] Yu David Liu, Scott F. Smith, and Andreas Terzis. A High-Level Language for Sensor Networks. Technical report, The Johns Hopkins University, Baltimore, Maryland, June 2004.

[Mac84] D. MacQueen. Modules for Standard ML. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 409–423, 1984.

[Man07] Stephen Manes. Dim vista, online at `http://www.forbes.com/`. Feburary 2007.

[MF07] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 3–10, 2007.

[Mic] Microsoft. Component Object Model Technologies, `http://www.microsoft.com/com/`.

[Mit06] Nick Mitchell. The runtime structure of object ownership. In *Proceedings of the 20th European Conference of Object-Oriented Programming (ECOOP)*, pages 74–98, 2006.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co., 1993.

[MPH01] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[MRT99] Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 44–53, 1999.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *Proceedings of the 12th European Conference of Object-Oriented Programming (ECOOP)*, pages 355–382, 1998.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152, 2003.

[Nel91] Greg Nelson, editor. *Systems programming with Modula-3*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[NFG+06] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau. Ultra-Large-Scale Systems - The Software Challenge of the Future. Technical report, Software Engineering Institute, Carnegie Mellon, June 2006.

[NPV98] James Noble, John Potter, and Jan Vitek. Flexible alias protection. In *Proceedings of the 12th European Conference of Object-Oriented Programming (ECOOP)*, Brussels, Belgium, July 1998.

[Obj02] Object Management Group. CORBA Components, June 2002.

[Obj07] Object Management Group. *The Unified Modeling Language (version 2.1.1), online at* `http://www.omg.org`, 2007.

[OPS92] Nicholas Oxhoj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 329–349, London, UK, 1992. Springer-Verlag.

[PC94] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 324–340, New York, NY, USA, 1994. ACM Press.

[Pie02] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[PNC98] J. Potter, J. Noble, and D. Clarke. The ins and outs of objects. In *Proceedings of the Australian Software Engineering Conference*, Adelaide, Australia, 1998.

[PNCB06] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 311–324, 2006.

[RS02] Ran Rinat and Scott F. Smith. Modular internet programming with cells. In *Proceedings of the 16th European Conference of Object-Oriented Programming (ECOOP)*, pages 257–280, 2002.

[Rum87] James Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Proceedings of the 1987 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 466–481, 1987.

[SBD04] Nathanael Schärli, Andrew P. Black, and Stéphane Ducasse. Object-oriented encapsulation for dynamically typed languages. In *Proceedings of the 2004 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 130–149, 2004.

[Sch98] Alexander Schrijver. *Theory of linear and integer programming*. Wiley, 1998.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *Proceedings of the 17th European Conference of Object-Oriented Programming (ECOOP)*, pages 248–274, 2003.

[SDNW04] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Composable encapsulation policies. In *Proceedings of the 18th European Conference of Object-Oriented Programming (ECOOP)*, pages 26–50, 2004.

[Sim02] Georg Simmel. *The Sociology of Georg Simmel (translated and edited by Kurt H. Wolff, book published in 1950)*. The Free Press, 1902.

[SRW98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[SRW99] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[Str97] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[SUN] SUN. The JavaBeans Specification, `http://java.sun.com/products/javabeans/docs/spec.html`.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of the 1987 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 227–242, New York, NY, USA, 1987. ACM Press.

[VB99] Jan Vitek and Boris Bokowski. Confined types. In *Proceedings of the 1999 ACM SIGPLAN*

Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 82–96, 1999.

[Wit01] Ludwig Wittgenstein. *Philosophical Investigations (translated by G.E.M. Anscombe)*. Blackwell Publishing Limited, third edition, 2001.

[WS01] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for java. In *Proceedings of the 15th European Conference of Object-Oriented Programming (ECOOP)*, pages 99–117, 2001.

Vita

Yu Liu, nicknamed David, was born on May 30^{th}, 1978 in Sanmenxia, China. He joined the Special Class for Gifted Students at South China University of Technology in 1994. After three happy years of playing with ring theory, Laplace transforms, and RISC instructions, he was awarded a Bachelor's Degree in Computer Science in 1997 with the highest honors. He subsequently conducted research on object-oriented software engineering in the same University, where he earned a Master's Degree in 2000. Since then, David has been pursing his Ph.D. study at Johns Hopkins University, having fun with programming language design and implementation, object-oriented languages, component and module systems, language-based security, multi-core programming, and sensor networks.