END-TO-END INFORMATION FLOW SECURITY FOR JAVA

by

Mark Andrew Thober

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

Baltimore, Maryland

October, 2007

© Mark Andrew Thober 2007

All rights reserved

Abstract

The increasing digitalization of individual, business, and government information leads to more sensitive information being used in computer systems. This results in the requirement for modern systems to ensure that sensitive information is not leaked. Information flow control is a programming language-based mechanism that focuses on securing the dissemination of information through programs. Information flow type systems aim to statically guarantee that programs do not permit leaks of sensitive information to unauthorized locations.

This dissertation focuses on improving the usability of information flow type systems, and on developing a new technique for proving a static information flow system is correct. We present a static information flow type inference system for Middleweight Java (MJ) that automatically infers information flow labels, thus avoiding the need for a multitude of program annotations. Additionally, policies need only be specified on IO channels, the critical flow boundary. Our type system includes a high degree of parametric polymorphism, necessary to allow classes to be used in multiple security contexts, and to properly distinguish the security policies of different IO channels. To further facilitate the writing

of practical programs, we add downgrading constructs to the language that permit minor leaks of information in explicitly allowable instances. We also describe how users can define top-level security policies for programs, allowing the policy to be easily viewed in the API.

We prove a noninterference property for programs that interactively input and output data: changes to high security input data are not visible at low security outputs. We use a new proof technique to show noninterference using a small-step operational semantics that is augmented with a syntactic representation of the type derivation. This shows the strong correlation between the static type system that is guaranteeing program security and the run-time manipulation of data.

Advisor:	Scott Smith, The Johns Hopkins University
Readers:	Jason Eisner, The Johns Hopkins University
	Jeffrey Foster, University of Maryland, College Park
	Scott Smith, The Johns Hopkins University

for Sarah

Acknowledgements

I must firstly thank Scott for his support and guidance; he helped me become a researcher. Thanks to Jason Eisner and Jeff Foster for reading this dissertation and providing many helpful comments and suggestions for improving it. Thanks to the members of the Hopkins Programming Languages Lab: Yu Liu, Xiaoqi Lu, Paritosh Shroff, and Chris Skalka. Their insightful discussions were helpful in understanding my own research and that of related works. Christian Scheideler aided my pursuits in theoretical computer science.

I greatly appreciate the many other computer science graduate students at Johns Hopkins that have helped me in my work and made life enjoyable: Ankur Bhargava, Amitabha Bagchi, Amitabh Chaudhary, Chris Riley, Kishore Kothapalli, Jatin Chhugani; Scott Doerrie and Swaroop Sridhar took time to break for coffee and tea. System administrators Steve Rifkin and Steve DeBlasio worked diligently to make sure our computer systems were running.

Thanks to all those who have offered prayers and spiritual support and encouragement, including the attendees of the Johns Hopkins Graduate Christian Fellowship and the

members of Second Presbyterian Church in Baltimore (particularly the choir). Thanks to all who have helped remind me that God is sovereign over all things, even my research.

Many thanks to the following for their friendship: Brian Macdonald, Dwight and Maria Schwartz, Michael and Catherine Paraskewich, Tim Wetzel, Tim Leary, Brian Cannon, Jamie and Laurel Magruder, Giuseppe Tinaglia and Hua Xu.

Thanks to all those at Nebraska Wesleyan University who helped me learn and made my time there enjoyable, especially O. William McClung and Tony Epp.

To my family, especially my parents and brothers, to whom I owe much of who I am; I will always be thankful for your love.

My final debt of gratitude goes to my wife Sarah. Without her unending support, patience, and love, this dissertation would not exist. To her I will be forever grateful.

Soli Deo Gloria

Contents

List of Figures

Chapter 1

Introduction

Sensitive information is ubiquitous in modern computer systems. Information is becoming increasingly digitalized and more accessible in every domain, from the individual to business and to government. This includes such data as national security secrets, financial information, business trade secrets, medical records, Social Security and credit card numbers, even an individual's monthly budget. Unfortunately, one need not look far for instances of sensitive data being exposed: from universities revealing confidential student information [Smi07], to customer information being stolen from businesses [Ber07], to breaches in national security [Yam07]. These information leaks (and countless others) emphasize the need for today's computer systems to ensure that the information they manipulate is secured.

Much effort has been expended to provide network and systems-level security. Indeed, the use of *firewalls* is now common among computer systems trying to avoid having information stolen. Though this type of security is certainly necessary and useful, it is insufficient, as characterized in the following quotation.

> "Focusing only on firewall protection is like building a fortress and then failing to take into account all the doors and windows that were inserted in the walls to enable data to flow to and from the castle domain. The data has to move in and out of the fortress, and it needs to be protected as it does, as required by the nature of a specific piece of information and the uses to which it will be put. Broadly speaking, protection in this context means that restrictions need to be in place so that exchange of data does not violate applicable privacy laws and so that confidential and proprietary business data does not lose its proprietary status and enter the public domain."
>
> – William A. Tanenbaum in the *New York Law Journal* [Tan07]

Even a firewall must allow applications to pass data through it, otherwise the system would be completely isolated. Therefore, an additional measure of security is required to establish the security of the applications that work with sensitive information.

The most common technique for addressing this problem is to employ an *access control* system that regulates which information a particular process may obtain and manipulate. Though other forms of systems-level access control may be employed, *language-based security* mechanisms operate at the level of a programming language to define and enforce security properties of programs. A good example of these features is the Java Security Architecture [GMPS97], which incorporates the sandbox model and stack inspection, though there are others [PSS01, SS00b, HO03].

Although access control is important for controlling data in applications, it is again insufficient, since it does not control the *propagation* of information through an application. Many programs must handle multiple types of sensitive information, and an access control system has no way of ensuring that each piece of information flows only to the correct places. Access control only determines what a program may access, but says nothing about what it may actually do with the information. A few examples of information leaks follow.

- Log files are commonly used in many applications; they may be used for debugging purposes, tracking system errors, usage monitoring, etc. Hence, log files are generally not meant to

contain sensitive information, and may be put in more accessible locations, such as where programmers or system administrators may view them. Since a program has access rights to read and write such log files, an access control system cannot keep sensitive data from being written to these files. This problem is relatively common, where applications write usernames and passwords to log files [U.S04, Sun07, Sec05].

- Encryption is a powerful mechanism for maintaining data security in storage and transport. However, applications that use this data must first decrypt it to perform necessary computation. This opens the door for potential mismanagement of the data if it escapes the program without being encrypted beforehand. Examples of information that was leaked when it should have been encrypted are frequent [Bis07, Wos03]. Again, access control has no way forcing a program to encrypt data before output.

- Accidental and intensional exposure of sensitive information via email is a significant security concern, illustrated by the following examples.

  In 2001, Eli Lilly and Company inadvertently exposed the email addresses of 669 subscribers to its Prozac email reminder service, meant to remind subscribers to take or refill their medication [U.S02]. A program written to automatically send these emails included the addresses of all subscribers in the "To:" field of the email, thus exposing the entire subscriber list. The program certainly had access to use the "To:" field, but should have used "Bcc:" instead.

  In 2007, an email containing top-secret nuclear weapons information of the United States was sent over insecure networks by board members of Los Alamos National Security, LLC (LANS) [Zag07]. Although this may partially be attributed to human error—since the originator of the email apparently did not know that the information was highly classified—stricter

system controls of data can aid in preventing such leaks. For example, a secure email system could prevent data that is marked as classified from being sent over insecure channels.

These examples all illustrate the point that more stringent control over the dissemination of sensitive information is necessary.

1.1 Information Flow Control

While access control provides security only for defining what is accessible to a particular process, *information flow control* regulates the actual dissemination of information. Once information is released to a process, the access control system cannot control how this information is dispersed. Hence programmer error or malice may cause information to flow to unauthorized locations.

Information flow control can be described by two types of security properties. *Secrecy* properties maintain that secret, or confidential, information should not be exposed to unauthorized users or processes; for example, a user's credit card number should not be viewable by those who do not have proper authority. *Integrity* properties provide that highly trusted data is not polluted by untrusted data; for example, a company may not want data coming from unregulated sources, such as the Internet, being written to their financial records database. Integrity is commonly viewed as a *dual* to secrecy [Bib77]: secrecy enforcement does not permit data to be leaked out to untrusted destinations, while integrity enforcement disallows data coming in from untrusted sources. Since these properties are duals, we focus on secrecy, and our formal system only tracks secrecy types (we discuss how integrity may be tracked in Section 3.2.12). However, we include integrity in this chapter and Chapter 2 to illustrate its importance.

Difficulties arise in securing information flows due to *covert channels*, which may cause improper information leaks. Examples include *indirect flows*, which arise due to the control structure of a program; *termination channels* may expose information based on the termination or non-termination of a program; *timing channels* can leak information by observing how long a program runs; even *resource usage channels* can leak information by observing how much of a particular resource a process uses in a given execution.

Given the difficulty of tracking all of these kinds of covert channels, the majority of information flow control systems guard against leaks only from *direct* and *indirect* flows, and we adopt this model (see Chapter 5 for a description of related systems). Direct flows occur when data is directly passed through a program, by assignments and other data-centric operations. The following piece of code is a direct flow from `h` into `l`, assuming `h` contains some sensitive data.

```
l = h + 1;
```

Indirect flows occur due to the control structure of a program. Again, in the following example, assume `h` contains sensitive data.

```
l = 0;
if (h < 0)
 {l = 1;}
 else {}
```

In this example, though `h` does not directly flow into `l`, by observing `l` after this code is executed, one can still learn information about `h`; namely, `h` is non-negative if and only if `l` is `0`. These indirect flows make information flow control extremely difficult to track at run-time, due to the possibility that information leaks may occur when no assignments are made under a high branch

condition. In our above example, when `h` $\geq$ `0` the `else` branch is executed, so `l` is not assigned under `h`; yet a leak still occurs in the value of `l` showing that `h` is non-negative.

Denning and Denning [DD77] first showed that a static program analysis can be used to capture these flows, with the added benefit of not incurring a run-time overhead. Since then, a great deal of research in information flow control has been dedicated to defining type systems that statically guarantee that high security data will not affect low security data [SM03, BN02, HR98, PS02, Mye99a, VSI96]. A *noninterference* [GM82] property is usually shown for well-typed programs: low security outputs are not affected by any high security inputs.

While the foundations of static information flow systems are solid, their usability needs improvement. The overhead for adding information flow security to programs is potentially large, since existing systems require many security annotations to be added to the code; programmers are required to specify the security level of variables, function arguments, return values, *etc.* With large numbers of annotations, the likelihood of having incorrect annotations also increases: a mistake can get lost in the noise of so many annotations. Input/output is another important practical concern which has also not been fully integrated into static information flow systems.

1.2 Information Flow Inference in Java

The primary goal of the work presented in this dissertation is to define a static system for practical data secrecy and integrity protection to aid programmers in securing programs they write, and to formally prove its correctness. We describe a static information flow type inference system for a core subset of Java (namely, Middleweight Java [BPP03]) that is formally proven correct using a new technique to show noninterference. The type inference system automatically infers informa-

tion flow labels, thus avoiding the need for a large number of program annotations [ST07]. Policies need only be specified on IO channels, which we will argue to be the only real flow boundary that must be considered. The type system includes a high degree of parametric polymorphism, necessary to allow classes to be used in multiple security contexts, and to distinguish policies of different IO channels. Noninterference is shown using a small-step operational semantics that is augmented with a *syntactic* representation of the type derivation.

Our work places the focus on input and output points as *the* important boundaries for securing data. Thus, we are only indirectly concerned about internal flows, in how they ultimately will relate to the inputs and outputs. In general, we should speak of securing the *component interface* [Szy98], since a runtime system may be composed of multiple independent components with distinct security policies; here we focus on just the IO boundary for simplicity.

As is common practice in information flow type systems, we associate a flow label with each program value. Labels are explicitly placed on input data and checking policies explicitly declared at output points; for points in between, the type system automatically infers the labels and so programmers do not need to add declarations. Input statements are of the form $\texttt{read}_{(\texttt{L}_\texttt{s},\texttt{L}_\texttt{i})}\texttt{(fd)}$, where $\texttt{L}_\texttt{s}$ and $\texttt{L}_\texttt{i}$ are the declared security level policy for secrecy and integrity of the channel, respectively, and `fd` is the file descriptor that names the channel. Similarly, output statements are of the form $\texttt{write}_{(\texttt{L}_\texttt{s},\texttt{L}_\texttt{i})}(\texttt{e},\texttt{fd})$, which outputs the result of expression `e` to the channel `fd`.

For practicality, we also support the ability to downgrade (*declassify*) secrecy labels, and upgrade (*endorse*) integrity labels when deemed safe to do so. In many cases, disallowing all information flows is too restrictive, and some small, controlled information leaks may be desired. For example, a program that authenticates a user's password must permit the user to know whether

or not the input password is the correct one; this response reveals something about the password on the system, but is nevertheless a necessary leak. As discussed in Section 4.7, declassification is difficult to formalize; yet, we believe our top-level policies, described in Section 2.8, will help programmers to determine and verify which declassifications are "safe" and expose those that are dangerous.

The type inference system provides an expressive form of parametric polymorphism. Polymorphism is crucial for modeling information flows with fine enough granularity. Different objects of the same class (e.g. two completely different `HashSet` objects) may be used in different security contexts, which must be differentiated in the analysis. Otherwise, secure programs may be rejected by a type system that unnecessarily merges flows. In our system, security policies on IO channels are defined at the level of Java `Stream` classes. This allows a `LowOutputStream` class to have a different security requirement than a `HighOutputStream` class. As described in Chapter 2, our fine-grained polymorphic type inference algorithm is essential for providing a fine enough distinction on IO channels.

To demonstrate the correctness of our system, we prove a type soundness result, and we also show a noninterference property, extended to account for interactive inputs and outputs. The majority of previous works assume only a batch model of IO, although O'Neill *et. al.* recently described a technique for enforcing information flow security for interactive IO, using a simple imperative language and basic type system [OCC06]. We show that well-typed programs (lacking declassifications—see Section 4.1) conform to a noninterference property: that two runs with identical low inputs will have identical low outputs, even if high inputs differ. As mentioned above, we have developed a new technique for proving noninterference, using a small-step operational seman-

tics that is augmented with the type derivation from our type inference algorithm. Since our type inference system contains a high degree of polymorphism, we use the entire type derivation tree to precisely align the low-behavior of two runs. This allows us to prove that two runs with equivalent low inputs are identical in their low execution behavior, resulting in equivalent low output streams.

One weakness of Java and other programming languages is how the IO points can get buried in the code through subclassing, method calls, *etc*. This in turn makes it difficult to observe the policies on the use of IO channels without digging through the whole program. This lack of a clear top-level IO interface means anyone who wants to understand the information flow properties of a whole program must have knowledge of the code details in order to understand what information flows occur through IO. We describe how users may define concise top-level policies for IO points in the program. Declassification and endorsement operations may also be defined via top-level policies at the level of class methods, so their validity can be more easily observed. This reduces the burden on both the programmer as well as the policy validator – the security policy for the whole program is now defined in one place.

The result of our strong type inference system and IO policy declarations is a system for a core subset of the Java language, where programmers need only specify the security policy of IO channels, and the type system ensures the program does not violate the policy.

1.3 Outline

The remainder of this dissertation is structured as follows. In Chapter 2, we present an overview of our information flow type inference system, including example programs that illustrate the usefulness and power of our system. Chapter 3 formally describes the language and type system.

In Chapter 4, we prove our formal results, describing our new semantic model and proof technique for proving soundness and noninterference of programs with an information flow typing. Chapter 5 examines work that is related to this dissertation. Chapter 6 concludes and offers some avenues for further research.

Chapter 2

System Overview

In this chapter, we informally describe the information flow type inference system and some of the challenges that must be addressed. We also provide some examples to illustrate the usefulness and expressive power of the system.

The syntax of our language is based on Middleweight Java (MJ) [BPP03], extended with labeled input and output operations, declassification and endorsement syntax, as well as other minor additions. There are three major reasons why MJ is a good language in which to define our information flow system. Firstly, we use a smaller core calculus of Java, as the full language is quite large and difficult to prove formal properties about; the added complexity and number of cases would make the proofs overwhelming, as the proof size and complexity is already quite substantial (see Chapter 4). Further, these additional features add little to showing the correctness or our approach. Secondly, with the addition of IO and operators, MJ contains the core language features necessary for reasoning about information flow, including conditionals, methods, and mutable state; MJ contains the core *imperative* features of Java, in addition to all of the *functional* features of the smaller

Featherweight Java [IPW99]. Finally, MJ is a valid subset of Java, in that any MJ program is also an executable program; this makes it more feasible to expand our system to the full language.

We define a static constraint-based type inference system, with a form of automatic label polymorphism inference that is related to CPA-style concrete class analyses [Age95, SW00, WS01]. The need for label polymorphism inference will become evident when we study the example programs of Sections 2.5 and 2.6.

In the remainder of this chapter, we first discuss adding input and output to the language in Section 2.1. We follow with a discussion of integrity in Section 2.2, then describe declassification and endorsement in Section 2.3. In Section 2.4, we discuss how constants are treated in our type system, and how default policies are defined. Section 2.5 presents an example program illustrating how programs may be written in our language and describes how the information flow types are inferred for this example. Section 2.6 presents some example programs motivating the need for the substantial polymorphism employed by the type system. Section 2.7 discusses usability aspects of our system, including a comparison to other Java-based information flow systems. Section 2.8 describes a mechanism for writing top-level policies for controlling information flows.

2.1 Input and Output

Input and output statements are $\texttt{read}_{(\texttt{L},\texttt{L}')}(\texttt{fd})$ and $\texttt{write}_{(\texttt{L},\texttt{L}')}(\texttt{e},\texttt{fd})$, where `fd` is the file descriptor of the IO channel, `e` is what is written to the output channel, and `L` and $\texttt{L}'$ are sets of labels specifying the secrecy and integrity levels of the channel, respectively. Although we include integrity in this chapter, our formal language omits integrity tracking since it is a dual property to secrecy (see Section 2.2 below). For convenience, we use labels sets and the usual set relations as our

security lattice [Den76]. Hence, security levels are ordered according to the subset relation $\subseteq$, with least upper bound operator $\cup$ (set union) and greatest lower bound operator $\cap$ (set intersection). The lowest security level is the empty set $\emptyset$ and the highest security level the universal set. We could use other security lattices by simply using a different partial ordering with different least upper bound and greatest lower bound operators. However, this would also alter how declassification is defined, since our definition is a removal of security labels from the label set (see Section 2.3).

At the point of a read operation, the returned value is tagged with the security labels of the channel. Further, checks are performed to ensure it is safe to read in the current security context. For example, a low read must not occur under a high guard. Otherwise, an attacker would notice that the amount of data read from a low stream would differ if the high guard differed. In the following example, if `h` contains high data, an execution that takes the `then` branch will read from a low stream once, while another execution taking the `else` branch will read from the stream twice, indirectly leaking information.

```
if (h < 0)
 {read_(∅,∅)(fd);}
 else {read_(∅,∅)(fd);read_(∅,∅)(fd);}
```

At each write, the labels on the value to be put to the channel are checked against the channel policy, to ensure that high secrecy data is not output to a low secrecy channel (and dually, that low integrity data does not flow into a high integrity channel). Similar checks to the above read case are employed, disallowing writes to low streams under high guards.

2.2 Integrity

Integrity is an important dimension of information flow security that may be treated as simply a dual to secrecy [Bib77]. This allows us to simplify our formal system in Chapter 3 to only track secrecy types, since integrity tracking is nearly identical; we describe how the type system can be extended to integrity in Section 3.2.12. However, in this chapter, we describe aspects of integrity and include integrity in our examples, since it is an important dimension of information assurance. For example, SQL injection attacks are often a source of information security breaches, where an attacker uses malformed queries to gain unauthorized access (e.g. [Kei07]). An integrity policy may disallow untrusted data from the Internet from flowing into an SQL query.

There are some subtle issues in the integrity domain, which we do not fully model [LMZ03, LZ05b]; we treat all code as trusted, since untrusted code may damage the integrity of the data it operates on (if a method purported to add two numbers together does something else); we guarantee only that data labeled *trusted* is not infected by *untrusted* data – more complex integrity policies that specify the validity of the data itself is beyond the scope of the present work.

2.3 Declassification and Endorsement

Our language includes a `Declassify(e,L)` statement, which returns `e` with the secrecy labels `L` removed (using the set difference operator). This serves to declassify data in infrequent, explicitly allowable instances [ML97, ZM01]. For example, the encrypted version of some high security data may be declassified, so it can be sent over an open link. The quality of the encryption algorithm then justifies this declassification.

With declassification defined in this manner, it is easy to see that the security level of the value will only decrease in the security lattice as specified by the labels being removed. This contrasts with other notions of declassification that either make each declassified value public, or simply change the security level of the result—which should really be called a *reclassification*, since it doesn't enforce the security level to be lower in the security lattice.

The integrity dual, `Endorse(e,L)`, increases the integrity label of the argument, specifying increased confidence in the data. For example, as in Perl's taint mode [COW00], tainted input data may be sanitized by pattern-matching, which concludes that the data is properly formatted. Programmers must be very careful when using declassify and endorse operations, because they may reveal too much information, or may not fully realize the integrity of the information, and thereby compromise security. In Section 2.8.2, we show how policies can be written in our system for better controlling declassify and endorse operations.

2.4 Program Constants and Default Policies

The use of security-critical constants directly in the program text can create security holes: hard-coded secret data may be mislabeled and leak out of a program through output operations, or by an unauthorized agent reading the source code itself. Similarly, program constants may adversely affect data integrity, *e.g.* if a rogue string constant is inadvertently written as a user's password. Remarkably, programmers continue to make such mistakes, even in recent commercially available programs and devices [KSRW04, Rob04, Fre05, Sym05], where hard-coded passwords and cryptographic keys resulted in security problems.

We take the approach that hard-coding of secret data or low-integrity data simply should not happen (*i.e.* should be avoided by the programmer). The only reasonable way to view program constants are as low secrecy but high integrity data, and this is how our type system treats all constants.

Establishing default policies for input and output channels is a closely related problem. This is important for establishing security for programs where not all IO channels have been given a security policy, and in describing policies for the standard input and output streams (`System.in`, `System.out` and `System.err` in Java). The default policy for an input channel is established as low secrecy and low integrity. This means the data is considered public and unreliable, which is a natural default for an unknown channel. The default policy for an output channel is also low secrecy and low integrity. This means the channel is considered observable to public users, and does not require any degree of confidence in the integrity of the data being output.

2.5 An Example Java Program

In this section we elaborate on how information flow is controlled at IO points in our system, by studying an example program. In the following section we then give an overview of our parametric polymorphism and label inference system.

Input and output channels in Java are created through subclassing, creating classes such as `FileInputStream`, `DataOutputStream`, `SocketInputStream`, etc. We build on this approach by defining different information flow policies via subclassing the core IO classes. In particular, a different subclass is created for each distinct security category of IO. This 1-1 relationship between

class definitions and security policies makes for an *object-oriented* approach to information flow policies, harmonizing with the existing language structures.

Note that the manner in which Java actually carries out IO using streams is more complicated than we describe, using native method calls; in our system, these low-level calls are modeled by our read and write operations. Furthermore, observable IO can occur in other ways in Java, outside the `Stream` classes, such as through a method for file `rename`. We do not presently model these other forms of IO in our system, so we are simplifying a bit. Additional forms of IO can be modeled in a similar fashion to the read and write streams that we employ, by specifying the necessary security policies for these other types of channels.

We now focus on an example program for changing passwords, where data security is important in both secrecy and integrity dimensions. This example is somewhat oversimplified but is short enough to illustrate the key concepts. Firstly, we want to provide secrecy for the user name and password information contained on the system, making sure this information is not leaked to a public channel, *i.e.* the screen. Secondly, we want to ensure the integrity of the system password file by not allowing it to be tainted by improper data, thereby altering user names and passwords on the system. These are two well-defined goals for a programmer of a password changing application.

We take some liberties with syntax that is not described in our formal calculus, such as the use of `super()` and a `while` loop. We make some abbreviations to shorten the presentation, `IS` for `InputStream`, `OS` for `OutputStream`, `PS` for `PrintStream`. Other obvious abbreviations have been made, and some code is omitted that is uninteresting for our purposes. Finally, assume that `FileIS` and `FileOS` classes each include a field `int fd` in their definition, representing the name of the file descriptor for the stream.

```
1: class PwdFileIS extends FileIS {
2:  int read() { return read_({high,sys},{trusted})(fd); }
3: }
4: class UserBufferedIS extends BufferedIS {
5:  int read() { return Endorse(super.read(),{trusted});}
6: }
7: class PwdFileOS extends FileOS {
8:  void write(int v) { write_({high,sys},{trusted})(v, fd); }
9: }
```

Figure 2.1: Example Program: `Stream` classes

We split the program into three figures to improve readability. Figure 2.1 contains the IO stream definitions; Figure 2.2 defines the `PwdFile` class, which operates on password files; Figure 2.3 contains `main`.

The modifications needed to support information flow analysis here are minor. The most significant requirement is to define distinct subclasses of `InputStream` and `OutputStream` for each distinct IO policy, shown in Figure 2.1. In this case we are defining three new IO policies, in the classes `PwdFileIS` and `UserBufferedIS` (for input), and `PwdFileOS` (for output). For `PwdFileIS`, the `read` method labels input values with `high` and `sys` for secrecy and `trusted` for integrity; this defines the security policy for data that is read from the system. The `write` method of `PwdFileOS` allows secrecy labels `high` and `sys`, and requires the integrity label `trusted`, thereby enforcing the policy that anything with a subset of secrecy labels {`high`,`sys`} may be written to the channel, but that only `trusted` data in the integrity domain may be written. The `UserBufferedIS` class is defined with an `Endorse` operation, expressing confidence in the integrity of the data on the channel. This is defined through the `BufferedIS` class that serves as a wrapper for other classes. As we shall see in the `main` portion of the program, it is used for reading from the keyboard

```
1:  class PwdFile extends Object {
2:   String fileName; String tempName;
3:   PwdFile(String fName, String tName) {
4:    this.fileName = fName;
5:    this.tempName = tName;
6:   }
7:   boolean isUser(String line, uname, oldpwd) {
8:    // parse line and return true if uname and oldpwd both match
9:   }
10:  Reader getPwdReader() {
11:   PwdFileIS fin = new PwdFileIS(fileName);
12:   return new BufferedReader(new ISReader(fin));
13:  }
14:  Writer getWriter() {
15:   PwdFileOS fout = new PwdFileOS(tempName);
16:   return new PrintWriter(fout);
17:  }
18:  boolean ChangePwd(String uname,oldpwd,newpwd){
19:   boolean success = false;
20:   String line;
21:   BufferedReader passIn = getPwdReader();
22:   PrintWriter tempOut = getWriter();
23:   while((line = passIn.readLine()) != null) {
24:    if (isUser(line,uname,oldpwd)) {
25:     tempOut.println(uname + ":" + newpwd);
26:     success = true;
27:    } else { tempOut.println(line) }
28:   }
29:   // rename tempFile to fileName
30:   return Declassify(success,{high,sys});
31:  }
32: } // end class PwdFile
```

Figure 2.2: Example Program: `PwdFile` class

input. We define the endorsement policy in this manner, as opposed to defining a new security channel, since the secrecy level of the channel being read is inconsequential. Here we only express confidence in the integrity of the channel. However, this endorsement emphasizes the point that

```
1:  void main(){
2:   String fileName = "/etc/passwd";
3:   String tempName = "/tmp/tmppasswd";
4:   PwdFile pf = new PwdFile(fileName,tempName);
5:   UserBufferedIS userin = new UserBufferedIS(System.in);
6:   BufferedReader br = new BufferedReader(new ISReader(userin));
7:
8:   System.out.println("Enter username:");
9:   String uname = br.readLine();
10:  System.out.println("Enter current password:");
11:  String oldpasswd = br.readLine();
12:  System.out.println("Enter new password:");
13:  String newpasswd = br.readLine();
14:
15:  boolean success = pf.ChangePwd(uname,oldpwd,newpwd);
16:  if (success) { System.out.println("Success"); }
17:  else { System.out.println("Failure"); }
18: }
```

Figure 2.3: Example Program: `main`

security downgrades must be done with great care, as reads from the keyboard should not always be assumed to be trustworthy.

The class `PwdFile` in Figure 2.2 is used for operating over password files. The method `isUser` parses a line from the Unix password file and returns true if the username and password are correct. The `getPwdReader` method returns a `BufferedReader` for reading from the password file using a nicer interface. Note that the underlying input stream is a `PwdFileIS`, so reads from the reader returned by this method will have the security policy defined in `PwdFileIS`. Similarly, the `getWriter` method returns a new `PrintWriter` with the underlying policy of a `PwdFileOS`. The `ChangePwd` method changes the password of a user on the system. The return value of the

`ChangePwd` method is declassified, which is necessary to allow the success or failure of the program to be output to the screen.

In Figure 2.3 the `main` method of the program is defined for changing a user's password. A `PwdFile` object is created for accessing the password file. A `UserBufferedIS` object is created for reading user input, and is wrapped by a `BufferedReader` object. Using a `UserBufferedIS` asserts the stream is trusted, since this class includes an endorsement of the data. After reading the user input, the `ChangePwd` method of the `PwdFile` object is invoked, and the program prints either `Success` or `Failure` back to the screen, depending on whether or not the password change was successful.

This program shows how code is written in the language; no explicit parametric type declarations are needed, and no label type declarations need to be placed on variables – type parametricity and variable information flow labels are both inferred automatically. So, the underlying Java program only needs to be changed to declare the appropriate IO channels and policies, and to add any needed downgrades and upgrades. The underlying program structure remains largely unchanged, *e.g.* a `PwdFileIS` object `fin` is still accessed via `fin.read()`, with no need for annotation.

Proper typing of this example imposes some requirements on the type system: the type of the `read` and `write` methods simply *cannot* be the same across all subclasses, otherwise all of the work we made to separate the policies in separate classes would be for nothing since the type system would merge the information flows. So, a form of parametric polymorphism is needed to distinguish between subclasses. It is even more subtle because a variable declared to be an `InputStream` can at runtime be any of its subclasses such as `PwdFileIS` or `UserBufferedIS`, and so it may look very difficult to type these methods distinctly. Our solution is to use a polymorphic form of concrete class

analysis [Age95]: we use a constraint-based type system that specializes the type of an object at each method call site for each different type of object that it could be. This technique leads to a very accurate typing [Age95, WS01], and allows the methodology of placing different security policies in different subclasses to be sound yet expressive. The most obvious forms of polymorphic type inference, based on treating each class or interface as polymorphic and not each method body and method invocation, are too weak to properly treat examples such as the `InputStream` mentioned above.

2.6 Polymorphism

To better illustrate the expressiveness of our polymorphic type system we show an alternate implementation of the `ChangePwd` method, one that takes an `InputStream` and `OutputStream` as arguments for reading from and writing to the password file, respectively.

```
boolean ChangePwd(IS in,OS out,String uname,oldpwd,newpwd){
 boolean success = false; String line;
 BR passIn = new BR(new ISReader(in));
 PrintWriter tempOut = new PrintWriter(out);
 // ... same code as above
}
```

The following code uses this new implementation. `SysFileOS` is subclassed from `FileOS`, and the `write` method of the new class checks the output data for only the secrecy label `sys`. In the `main` portion, two different calls are made to `ChangePwd`, one with a `PwdFileOS`, as before, and one to a `SysFileOS`.

```
class SysFileOS extends FileOS {
 void write(int v) { write({sys},{trusted})(v,fd);}
 }
```

```
void main() {
 // ... same code as above
 PwdFileIS in = new PwdFileIS("/etc/passwd");
 PwdFileOS pout = new PwdFileOS(tempName);
 SysFileOS sout = new SysFileOS("/etc/sysfile");
 pf.ChangePwd(in,pout,uname,oldpwd,newpwd);
 pf.ChangePwd(in,sout,uname,oldpwd,newpwd);
}
```

It is critical that the different calls to `ChangePwd` are typed differently, since their arguments have different security policies. Our polymorphic type system is expressive enough to directly support this new `ChangePwd` method. Additionally, since we are statically inferring the concrete classes of objects, we can create different security policies for overriding methods, and the type system will know the correct policy to use. In this example, the first call to `ChangePwd` will type properly, but the second call will cause a type error, since the data passed to the `write` method of the `SysFileOS` is labeled with `high`, which is not permitted, since the policy of `SysFileOS` only permits data labeled with at most the `sys` label.

In addition to the need for polymorphism for discriminating IO streams, we also need polymorphism for code re-use. Code should be reusable in multiple contexts, which may have different information flow policies. This means concretely that library classes and methods must be allowed to be instantiated at multiple security contexts, and the type system must not merge all of the flows. We illustrate this with the example in Figure 2.4 of different `HashSet` objects: one holding high data, and the other holding low data.

In this example, we define two input stream classes, one for reading in high data, and one for low data, and an output stream class for writing low data. The program reads from both high

```
1:  class HighFileIS extends FileIS {
2:   int read() { return read_({high},{trusted})(fd); }
3:  }
4:  class LowFileIS extends FileIS {
5:   int read() { return read_(∅,∅)(fd); }
6:  }
7:  class LowFileOS extends FileOS {
8:   void write(int v) { write_(∅,∅)(v,fd); }
9:  }
10: void main() {
11:  HashSet highSet = new HashSet();
12:  FileIS hin = new HighFileIS("high_infile");
13:  int i; int j;
14:  while(i = hin.read())
15:   { highSet.add(i); }
16:  HashSet lowSet = new HashSet();
17:  FileIS lin = new LowFileIS("low_infile");
18:  while(j = lin.read())
19:   { lowSet.add(i); }
20:  Iterator lowIt = lowSet.iterator();
21:  FileOS lowout = new LowFileOS("low_outfile");
22:  lowout.write(lowIt.next());
23:  }
```

Figure 2.4: Example Program: `HashSet`

and low streams into separate `HashSet` objects. A value is then taken from the `HashSet` containing low data, and written to the low output channel.

This clearly shows the need for polymorphism over security levels. If the types for these two `HashSet` objects were merged, the program would be rejected, because high data would appear to flow out a low channel. Our system views `HashSet` as polymorphic and the `highSet` and `lowSet` are typed distinctly, so the program typechecks.

2.7 Usability

A major goal of this work is to improve the practical use of static information flow control systems. Therefore, before describing the formal system, let us first turn our attention to aspects of *usability* [Nie93, JIMK03].

Unfortunately, usability is difficult to measure, and so proper justification that one system is more usable than other will require empirical studies. This is problematic, since very few information flow systems have been implemented, with Jif [Mye99a] and FlowCaml [PS02] as notable exceptions. Hence, these implemented systems are inherently more usable than those that are not implemented (including ours). Measuring usability is further complicated by these systems having very few users outside the system developers; although a few applications have been written in Jif, allowing for some study of its usability [HAM06, AS05].

Since empirical evidence is unavailable, we discuss usability in this section by comparing the languages and type systems of our system with the most related Java-based systems: Jif [Mye99a, MZZ+01] and the type system by Banerjee *et. al.* [SBN04]. Further description of related systems may be found in Chapter 5.

Our approach emphasizes making programs be as close as possible to Java programs. Indeed, as shown in our examples in Sections 2.5 and 2.6, the only difference between programs written in our language and Java programs are in how low-level read and write operations are defined, and the addition of `Declassify` and `Endorse` syntax. We believe this is a vast improvement over current systems that require many security annotations.

Jif requires much of the code to be annotated, in order for information flows to be controlled. Examples of this include variables: `int{public} x;`, function arguments and returns

`int{secret} foo (int{secret} x)`. Our system avoids this, as all security labels are completely inferred from the IO policies. While Jif provides some class polymorphism, parameters must be *explicitly* declared. For example, a vector implementation is defined in Jif as follows, where `L` are label parameters.

```
public class Vector[label L] extends AbstractList[L] {
 private int{L} length;
 ...
```

Our classes and methods are *implicitly* polymorphic, and require no such annotations. This means existing Java code can be used as is (provided IO policies are declared), where libraries must be re-worked in Jif, which requires all code to be annotated.

The Java-based system by Banerjee *et. al.* employs parameterized polymorphism similar to that of Jif, where fields, method arguments, and method return types are all annotated with either label variables or constant security levels. For example, a class `TAX` may be written as follows, where α_i represent label variables.

```
class TAX ⟨ α1 ⟩ extends Object {
 (int,α1) income;
 (int,α4) tax((int,α4) salary) {
 ...
```

This has the same drawback we discussed above: libraries must be re-worked, as all code must be annotated. Again, we have no such requirement since our polymorphism is all implicit.

As stated above, one of the goals of our system is to provide information flow security with minimal changes to the Java syntax, with the only difference between our language and Java being the description of security policies at reads and writes (and uses of declassification and endorsement, when necessary). In fact, any MJ program is valid in our syntax, when assuming that IO channels

are given a default security policy. This facilitates a more *learnable* (and *memorable*) system, since programmers will not need to learn a new programming syntax. In addition, programmers can now think about information security in a more natural manner, based on information policies of IO channels, instead of reasoning about all of the internal flows and intricacies of the type system.

In Section 2.8, we discuss how a top-level policy description may be used to bring the policy further out of the internals of the code making it viewable in the API. This is important in two regards. Firstly, it makes it simpler to add information flow controls to a program, a key aspect both of *learnability* and *efficiency*. Secondly, it makes the policy more clear. This is an important part of reducing *errors*, since the type system can only check that a program is secure with respect to a given policy. So, it is up to the programmer to ensure that the policy given is the correct one. By making the policy viewable in the API, our top-level policies make the inspection of the policy simpler.

In comparison to these systems (and also other information flow type systems), only our type system emphasizes the importance of IO in controlling information flows: in our system policies are defined only at the boundaries of programs. This is a substantial improvement, as the goal of information flow control is to provide end-to-end security. Further, only our system models interactive IO, which underscores our approach to achieving a more realistic setting for information flow control in programs.

In the following chapter, we provide the formal description of the type system. The goal of increased usability is a strong motivator for how our type inference system functions. Indeed, the type inference system is complex by design: the system does all the hard work so the programmer doesn't have to.

2.8 Policies

In this section, we describe how class-based policies may be declared at the top level of a program, meaning the policy will not be buried in the code, but can be seen in the API. This also provides a simpler means of adding information flow controls to programs, since the underlying programs will not need to include any explicit flow annotations and so there is no need to define a new language syntax for an information flow extension.

Policies are of two different kinds. IO policies specify the security levels of IO channels. Declassification/endorsement policies specify the downgrading/upgrading of security levels. We shall discuss these policies separately in the subsequent sections. First, let us look at a policy for the program for changing passwords in Section 2.5.

```
class PwdFileIS: ({high,sys},{trusted})
class UserBufferedIS
 read():  Endorse({trusted})
class PwdFileOS: ({high,sys},{trusted})
class PwdFile
 ChangePwd(String uname,oldpwd,newpwd):  Declassify({high,sys})
```

The IO policy for the class `PwdFileIS` is used for input channels that are $\{\texttt{high}, \texttt{sys}\}$ secrecy and $\{\texttt{trusted}\}$ integrity, and `PwdFileOS` has the same policy for output channels. Declassification and endorsement policies are specified for the `ChangePwd` method of `PwdFile` and `read` method of `UserBufferedIS`, respectively. This example shows how the policy can be easily viewed in the API, since it is now clear what the security policy on each IO class is, and where declassifications occur.

In addition, this type of policy shows how easy it is to add information flow controls to a program. Given a valid program and a policy, we can easily translate the program into a new

program with security levels on `read` and `write` expressions of `InputStream` and `OutputStream` subclasses, and `Declassify` (and `Endorse`) policies on method return values, when downgrading (or upgrading) is warranted. Even though programs may contain no explicit information flow policy information, it still may be necessary to rewrite parts of a program for purposes of adding a fine-grained information flow policy: a unique subclass needs to be defined for each different IO security policy. This can be viewed as a good step, because it leads to a more object-oriented information flow policy.

2.8.1 Read and Write Policies

IO policies are used for specifying the security level of input and output channels. `read` policies declare the sets of security labels for an input channel using the Java representation of an `InputStream` subclass. Hence, the `read` method of the subclass is re-written to perform a low-level read operation with the security labels given by the policy. In a similar manner, `write` policies declare the sets of security labels for an output channel using the Java representation of an `OutputStream` subclass. The `write` method is re-written to perform a low-level write operation with the security labels given by the policy.

Any sub-classes of `InputStream` and `OutputStream` that do not have a defined policy receive the default policy, described in Section 2.4. Hence all unspecified input streams are low secrecy and low integrity; the default policy for an output stream is also low secrecy and low integrity.

Expanding the system to full Java requires additional alterations to the classes. This means that all native methods within an IO stream class need to be re-defined to attach the policy. For example, the Java `FileInputStream` class has a native method `readBytes` that reads a number of bytes into an array.

2.8.2 Declassify and Endorse Policies

Since declassification and endorsements are intentional information leaks, they must be applied with great care. There has been a great deal of research focused on controlling these mechanisms so that they do not inadvertently break a program's security [SS05] (see Section 5.1 for further description). We take an approach similar to Hicks *et. al.*'s declassifiers [HKMH06] and use `Declassify` policies to specify what labels will be declassified from a method's return value. Note that although this provides the ability to specify declassification policies at the top-level, declassification of data requires knowledge of the underlying code to be sure the data is truly diluted enough to warrant declassification, so it must be used with care. `Declassify` policies can only be applied to methods with non-`void` return types, since it is the value that is returned from the method that is declassified. `Endorse` policies are defined analogously for integrity upgrading.

Although performing declassification generally requires some confidence in how the code manages sensitive data, with these policies the API will show which methods permit declassification, which can be checked for accuracy. This provides a high-level view of declassification to aid in inspecting for spurious policies; e.g. it makes implicit sense to declassify the result of an encryption method, whereas a declassification of a simple get method is usually not warranted. Restricting declassification to method returns also makes the use of declassification more apparent, so spurious declassifications are less likely to be missed.

2.8.3 Policies at Code Deployment Time

Since top-level policies may be used to declare the relevant security properties of the code, they permit policies to be defined at deployment time. This allows the users deploying an application

to tailor it to fit their own security requirements. However, declassification (and endorsement) is a delicate issue that usually requires some inspection of the code, so automatic insertion of declassify is often a bad idea. Security requirements should remain part of the software development process, from design through deployment. These software engineering challenges are out of the scope of this dissertation, but are an interesting avenue to explore in future work.

Chapter 3

Types for Data Tracking and Checking

We now present the formal type inference system. In order to simplify the reasoning and presentation of the system, we define a *label type inference system* solely for typing information flows, and use the existing MJ type system for normal MJ typechecking not related to information flow. Our label type system is strong enough to handle any valid MJ program, including those with mutually recursive class definitions, and method recursion. A program type checks if and only if it type checks in both the MJ type system and the label type system. Unless otherwise noted, when discussing types, we mean the label type system.

3.1 The Language

Our language is an extension of Middleweight Java (MJ) [BPP03]. MJ contains the basic object constructs of Java, including state; it omits some of the more complex features of Java, such as exceptions, for and while loops, arrays, statics, and access levels (private/public). This makes it much easier to establish formal properties of the system. Although MJ includes them, we eliminate

P	::=	$\overline{\texttt{CL}};\overline{\texttt{s}}$	*program*
CL	::=	$\texttt{class C extends C}\ \{\overline{\texttt{C f}};\ \texttt{K}\ \overline{\texttt{M}}\}$	*class*
K	::=	$\texttt{C}(\overline{\texttt{C x}})\{\texttt{super}(\overline{\texttt{e}});\overline{\texttt{s}}\}$	*constructor*
M	::=	$\texttt{RT m}(\overline{\texttt{C x}})\ \{\overline{\texttt{s}}\}$	*method*
RT	::=	$\texttt{C} \mid \texttt{void} \mid \texttt{int} \mid \texttt{boolean}$	*return type*
L	::=	$\{\overline{\texttt{l}}\}$, where l are unique labels.	*label*
CO	::=	$\texttt{c} \mid \texttt{b} \mid \texttt{null} \mid \texttt{fd}$	*constant*
e	::=	$\texttt{x} \mid \texttt{this} \mid \texttt{CO} \mid \texttt{e.f} \mid (\texttt{C})\ \texttt{e} \mid$ $\texttt{e} \oplus \texttt{e} \mid \texttt{pe} \mid \texttt{Declassify}(\texttt{e},\texttt{L}) \mid$ $\texttt{read}_{\texttt{L}}(\texttt{fd})$	*expression*
pe	::=	$\texttt{e.m}(\overline{\texttt{e}}) \mid \texttt{new C}(\overline{\texttt{e}}) \mid$	*promotable expression*
s	::=	$\texttt{pe;} \mid \texttt{if}\ (\texttt{e})\ \{\overline{\texttt{s}}\}\ \texttt{else}\ \{\overline{\texttt{s}}\} \mid$ $\texttt{;} \mid \{\overline{\texttt{s}}\} \mid \texttt{e.f} = \texttt{e;} \mid \texttt{return e;} \mid$ $\texttt{write}_{\texttt{L}}(\texttt{e},\texttt{fd});$	*statement*

Figure 3.1: Grammar

local variables from the formal calculus, which complexify the operational semantics and proofs. In Section 3.2.4, we show how local variable typings are a straight forward extension of object field typing. We add *constants* (of type `int` and `boolean`), *operators* (`+`, `-` etc.), in order to better reason about information flows in real programs. We also add low-level read and write operations to the language, of the form $\texttt{read}_{\texttt{L}}(\texttt{fd})$ and $\texttt{write}_{\texttt{L}}(\texttt{e},\texttt{fd})$, where `fd` is the file descriptor of the IO channel, `e` is what is written to the output channel, and `L` is a set of labels specifying the secrecy level of the channel. We also add a `Declassify(e,L)` construct, which removes the secrecy labels in `L` from those on `e`. Since secrecy and integrity are dual properties, we only present the secrecy label types in the formal system. In Section 3.2.12, we discuss how integrity label types and endorsement syntax can be added to the system.

The grammar for our Extended MJ (EMJ) language is given in Figure 3.1. We make some brief remarks regarding this syntax that was not already described; since the remaining definitions are obvious, any further description may be found in the MJ document [BPP03]. As in MJ, we

assume there is a distinguished class `Object`. We write $\overline{\texttt{s}}$ to indicate a sequence of zero or more statements, (with similar meaning for $\overline{\texttt{e}}$, $\overline{\texttt{CL}}$, etc.) Programs are defined as a set of classes and a sequence of statements representing `main`. A class consists of a number of fields, a constructor, and some number of methods. We use the following notation for metavariables: `C` ranges over class names, `f` over field names, `m` over method names, and `x` over variables. Constants, `CO`, are one of three varieties; integers are represented by `c` and `fd` (we use `fd` as an easy way to denote low-level file descriptors); booleans (either `true` or `false`) are represented by `b`; null pointers are represented by `null`. Binary operations are represented by $\oplus$. Java allows certain expressions to be promoted to statements by affixing a semicolon. Hence `pe` is defined to permit method invocations and object creations to be promotable expressions.

We assume some familiarity with Java, and since MJ is a valid subset, we do not reproduce its typing or semantic definitions, which can be found elsewhere [BPP03]. However, we now discuss some aspects of the normal typing (i.e., not label typing) of programs in EMJ. EMJ follows MJ and types expressions with respect to a global class table, CT, that contains the types of all classes. At the top level a sequence of statements $\overline{\texttt{s}}$ corresponding to the `main` method is typechecked with respect to this table. In addition to the standard type rules for MJ, we add the type rules corresponding to the EMJ extensions, which are straightforward: $\texttt{read}_{\texttt{L}}(\texttt{fd})$ is typed to return an integer and $\texttt{write}_{\texttt{L}}(\texttt{e}, \texttt{fd})$ outputs an integer (`e` has an integer type); `fd` is also of integer type, since this is how low-level file descriptors are formulated in Java (the class `FileDescriptor` is an object wrapper for low-level file descriptors). For $\texttt{Declassify}(\texttt{e}, \texttt{L})$, the resulting type of the expression is the same type as `e`, since the label tracking is only handled in the label typing rules.

τ	::=	$\langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle \mid t$	*types*
$\mathcal{S}$	::=	$\emptyset \mid \{\overline{\mathtt{l}}\} \mid s^\sigma \mid \mathcal{S} \cup \mathcal{S} \mid \mathcal{S} - \mathtt{L} \mid \mathcal{F}.\mathtt{f}.\mathsf{S} \mid$ **set** $\mathcal{S}$	*secrecy types*
$\mathcal{F}$	::=	$\emptyset \mid \{\overline{\mathtt{f}} \mapsto \overline{\tau}\} \mid f^\sigma \mid \mathcal{F}.\mathtt{f}.\mathsf{F} \mid$ **set** $\mathcal{F}$	*field types*
$\mathcal{A}$	::=	$\mathtt{C} \mid \mathtt{void} \mid \mathtt{int} \mid \mathtt{boolean} \mid \alpha^\sigma \mid \mathcal{F}.\mathtt{f}.\mathsf{A} \mid$ **set** $\mathcal{A}$	*alpha types*
σ	::=	$\mathtt{C}, \mathtt{m}, \overline{\mathcal{A}}, \mathcal{A}_t, \mathcal{A}_r \mid \epsilon$	*contours*
$s^\sigma, f^\sigma, \alpha^\sigma, s_p$			*label variables*
t	::=	$\langle s^\sigma, f^\sigma, \alpha^\sigma \rangle$	*type variables*
κ	::=	$\forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}$	*method types*
c	::=	$\mathcal{S} <: \mathcal{S} \mid \mathcal{F} <: \mathcal{F} \mid \mathcal{A} <: \mathcal{A}$	*constraints*
		$\mid \mathcal{A}.\mathtt{m}(\overline{\tau}, \tau_t \xrightarrow{pc} \tau_r) \mid SC(\mathtt{L}, \mathcal{S})$	
$\mathcal{C}$	::=	$\{c\} \mid \mathcal{C} \cup \mathcal{C} \mid \emptyset$	*constraint sets*
pc	::=	$\mathcal{S}$	*program counter type*
s	::=	$s^\sigma \mid \mathit{f}.\mathtt{f}.\mathsf{S}$	*transitive types*
f	::=	$f^\sigma \mid \mathit{f}.\overline{\mathtt{f}.\mathsf{F}}$	*transitive types*
a	::=	$\alpha^\sigma \mid \mathit{f}.\mathtt{f}.\mathsf{A}$	*transitive types*

$\tau <: \tau'$ is short for $\mathcal{S} <: \mathcal{S}', \mathcal{F} <: \mathcal{F}', \mathcal{A} <: \mathcal{A}'$
where $\tau = \langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle$ and $\tau' = \langle \mathcal{S}', \mathcal{F}', \mathcal{A}' \rangle$.

Figure 3.2: Notation

3.2 Label Types

EMJ values are either objects or primitive constants. Objects may be labeled, as may the internal fields of an object. Thus, label types, τ, are three-tuples $\langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle$; $\mathcal{S}$ is a set of secrecy labels for the current object, $\mathcal{F}$ is a record containing sets of labels, representing the internal fields of the object, and $\mathcal{A}$ is an α-type, a type representing the concrete class of the object, explained below. The type definitions are summarized in Figure 3.2.

Each field of an object has its own labels, represented by the field type $\mathcal{F}$, which is a mapping of field names to types, $\{\mathtt{f}_1 \mapsto \tau_1, \ldots, \mathtt{f}_n \mapsto \tau_n\}$. The individual labels may be accessed by a dot notation: $\mathcal{F}.\mathtt{f}.\mathsf{S}$ is the secrecy type on the `f` field of the object, $\mathcal{F}.\mathtt{f}.\mathsf{F}$ is the field type of the `f` field, and $\mathcal{F}.\mathtt{f}.\mathsf{A}$ is the α-type of the `f` field. Primitive constants are labeled as objects with no fields.

The α-types are used to express a form of parametric polymorphism over the inheritance hierarchy, allowing the superclass and subclass to differ in their labeling. The usual Java type declaration is insufficient for determining the class of an object, as it may be an object of a subclass, which contains a different policy, or returns different labels. As discussed in Section 2, we need a more expressive form of polymorphism. Our analysis is closely related to Data-polymorphic CPA [SW00, WS01], a variant of CPA [Age95]. This ensures creation of distinct *contours* (polyinstantiations) when needed to give the type expressivity required for our system (Sections 3.2.6 and 3.2.7), while on the other hand merging enough contours to make sure the analysis terminates (Section 3.2.9).

σ defines the contours necessary for polymorphic method typing, and type variables are extended to allow a contour superscript, (e.g. s^{σ}) and ϵ represents no superscript. The definition of contours allow them to be distinct to each receiver type `C`, method name `m`, argument type $\overline{\mathcal{A}}$ (where $\mathcal{A}_t$ is the type of `this` and is treated as an argument), and return type $\mathcal{A}_r$. Initially, all α-types have no superscript. They may receive a superscript during a polyinstantiation in the closure. Further description may be found in Sections 3.2.6 and 3.2.7. For convenience, we generally omit the superscript on variables when it is unimportant.

We use `l` to represent a concrete label, and s for label variables in the secrecy domain. Notation `L` refers to a set of concrete labels $\{\overline{\texttt{l}}\}$, and label sets $\mathcal{S}$ may contain both concrete label sets and label variables, the latter used when the concrete label is not yet known. For example, when typing methods, the argument labels are variables since the actual labels are not instantiated until the method is invoked. Hence, $\mathcal{S}$ is either the empty set, a set of concrete labels, a label variable, a union of secrecy types, a set difference of secrecy types (for when declassify is used), a field access,

or a set type. f is a field variable referring to abstract fields of an object, and $\mathcal{F}$ is either an abstract field, a concrete field mapping, a field access, or a set type. α is a variable referring to an unknown class, and $\mathcal{A}$ is either an abstract class α, a concrete class $\mathtt{C}$, a field access, or a set type. As discussed above, field accesses $\mathcal{F}.\mathtt{f}.\mathsf{S}$, $\mathcal{F}.\mathtt{f}.\mathsf{F}$, and $\mathcal{F}.\mathtt{f}.\mathsf{A}$ are used for the types of object fields. Set types, **set** $\mathcal{S}$, **set** $\mathcal{F}$, and **set** $\mathcal{A}$ are used for typing assignment statements, as described in Section 3.2.5. t denotes a full three-tuple of label types, and is simply short-hand; throughout this dissertation, we will often write t_i and refer to its elements as s_i, etc.

We implicitly work over a simple equational theory of sets in typing and constraint closure, when secrecy types are concrete sets. Concrete unions, $\mathcal{S} \cup \mathcal{S}'$, where $\mathcal{S} = \{\overline{\mathtt{l}}\}$ and $\mathcal{S}' = \{\overline{\mathtt{l}'}\}$ are considered equivalent to the unioned set, $\mathcal{S} \cup \mathcal{S}' = \{\overline{\mathtt{l}}, \overline{\mathtt{l}'}\}$ (without repeats). $\mathcal{S} - \mathtt{L}$ is also equivalent to the obvious set difference when $\mathcal{S}$ is a concrete label set. Further, $\cup$ is implicitly symmetric: $\mathcal{S}_1 \cup \mathcal{S}_2$ and $\mathcal{S}_2 \cup \mathcal{S}_1$ are interchangeable. For field access, $\{\overline{\mathtt{f}} \mapsto \overline{\tau}\}.\mathtt{f}_i.\mathsf{S}$ is equivalent to $\mathcal{S}_i$, where $\mathtt{f}_i \mapsto \langle\, \mathcal{S}_i, \mathcal{F}_i, \mathcal{A}_i \,\rangle$ is in the mapping $\{\overline{\mathtt{f}} \mapsto \overline{\tau}\}$. A similar equivalence analogously holds for any $\{\overline{\mathtt{f}} \mapsto \overline{\tau}\}.\mathtt{f}_i.\mathsf{F}$, or $\{\overline{\mathtt{f}} \mapsto \overline{\tau}\}.\mathtt{f}_i.\mathsf{A}$.

We use a label table, LT, to keep track of the label types of all classes when typing expressions. This is analogous to the class table CT of the MJ type system that keeps track of all class types. However, since we are inferring label types here, we must build up the label table while typing the classes, as discussed in Section 3.2.3.

Label type rules are of the form $\Gamma, pc \vdash \mathtt{e} : \tau \backslash \mathcal{C}$ and $\Gamma, pc \vdash \mathtt{s} : \tau \backslash \mathcal{C}$, meaning in type environment Γ, with program counter type pc, expression $\mathtt{e}$ or statement $\mathtt{s}$ has label type τ with constraint set $\mathcal{C}$. Γ binds variables to label type variables, $\Gamma(\mathtt{x}) = t$. The program counter type is used to track implicit flows through programs by indicating the current security context; this is

a standard feature of information flow type systems. We generally refer to this type as simply the program counter.

The constraint set, $\mathcal{C}$, contains normal subtyping constraints $<:$ for secrecy, field, and α-types ($<:$ indicates the direction of information flow, so if $\mathcal{S} <: \mathcal{S}'$, we say $\mathcal{S}$ *flows into* $\mathcal{S}'$). In addition, secrecy check constraints of the form $SC(\mathtt{L}, \mathcal{S})$ are placed in $\mathcal{C}$ and the closure process will need to verify their correctness (Section 3.2.7). Method constraints $\mathcal{A}.\mathtt{m}(\overline{\tau}, \tau_t \xrightarrow{pc} \tau_r)$ represent the type of a call site (invocation), and contain the necessary information for polymorphic instantiation of method types. $\mathcal{A}$ is the α-type representing the class of the object being invoked with the method `m`; $\overline{\tau}$ are the types of the explicit arguments, τ_t is the type of `this`, τ_r is the return type, and pc is the type of the current program counter. Method types in the label table are universally quantified, $\forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}$, so they may vary parametrically; $\overline{t_l}$ represent all local type variables in the method body typing, $\overline{t_x}$ represent the method arguments, t_t represents `this`, t_r represents the return type, and s_p represents program counter. This allows distinct contours to be formed for each combination of argument type and call site. We present a simplified version of our polymorphic types in Section 3.2.6, and detail this analysis when discussing the constraint closure in Section 3.2.7. Transitive types, s, f, and a are defined to allow creation of the proper constraints in the closure through transitivity rules (see Figure 3.9). They ensure transitivity only occurs through type variables.

Now that we have defined our notation, we proceed by discussing specific elements of the type inference system separately.

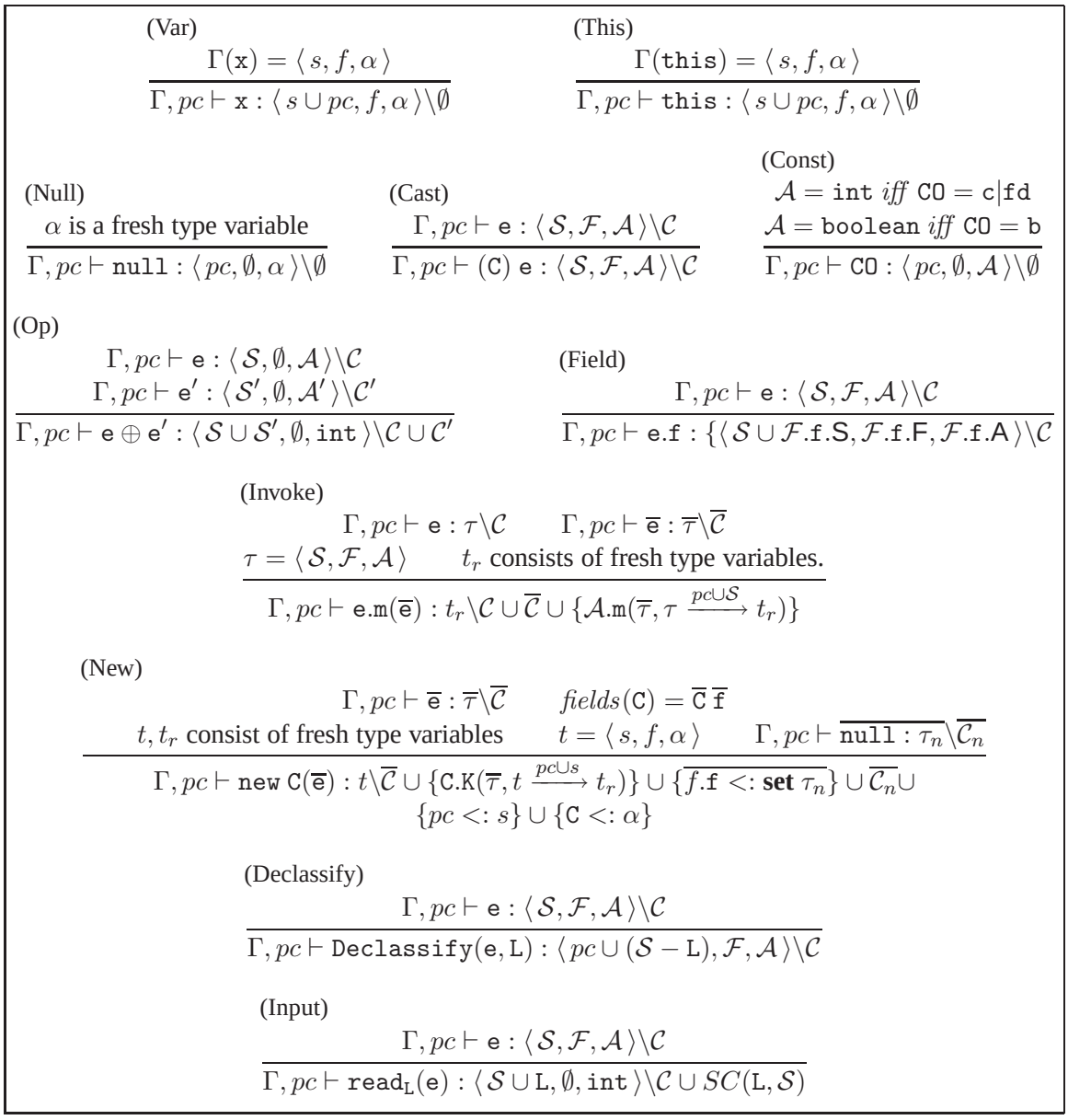


Figure 3.3: Label Type Rules for Expressions

3.2.1 Expression Typing

The label type inference rules for expressions are given in Figure 3.3. Using (Var), variable types are looked up in the type environment Γ, and the program counter is added to the type (`this` is a special variable, treated in the same way). (Const) types constants as label types containing only

pc for secrecy, reflecting our view that constants should by default have no secrecy as discussed in section 2.4. We use the same rule to type integers and booleans for simplicity. Using (Null), `null` values are given a similar type to constants, only with a fresh α-type indicating that `null` can take the type of any class.

In (Cast), the label types remain the same, since casting operations do not change the data in the object, only the actual (non-label) type of the object. (Op) assigns the secrecy label type of a binary operation the union of the labels of the arguments; for simplicity, we assume operations return integers. The field portions of the types of `e` and `e'` are always empty, since operations may only be performed on constants, which have no fields. In (Field), the secrecy type includes the labels on the field within the object, along with the labels the object itself carries; the field and α-types are taken from the field type of the object.

In (Invoke), the method constraint $\mathcal{A}.\mathtt{m}(\overline{\tau}, \tau \xrightarrow{pc \cup \mathcal{S}} t_r)$ is added to the constraint set, which will be given a polymorphic instantiation in the closure (see Sections 3.2.6 and 3.2.7). The program counter over the arrow denotes the security context under which this particular method invocation is executing; hence $\mathcal{S}$ is added to the program counter, since the execution of method `m` depends on the object to which the method is being called on. The method type eventually needs to be looked up in the global label table LT. However, since $\mathcal{A}$ may at this point be of an unknown class (such as when `e` is a variable) we postpone this decision until more information is known about $\mathcal{A}$, at constraint closure. The above type constraint records the method call information so it can be propagated in the closure once the concrete class of $\mathcal{A}$ is known.

In (New), *fields* is used to look up the field names in the class `C` (this function is defined in Figure 3.5). New type variables are created to refer to the object being created, and t_r; these are used

to unify the typing of constructor calls with method calls, in order to simplify the proofs, which will become clear in Chapter 4. We also insist on the typing of a series of `null` pointers, for each of the initial object fields; $\{f.\texttt{f} <: \textbf{set}\ \tau\}$ constraints are included to show that the fields will be initially assigned to `null`, the **set** constraint is described in Section 3.2.5. We cannot simply add the types of each argument to the field types, since the constructor may not have this behavior. A constraint is added to capture the call to the constructor, which is similar to a method call. Constraints are also added showing the security context of the object, and that the α-type has the concrete class name of the object being created.

As expected, `Declassify(e,L)` removes `L` from the secrecy labels of `e` in (Declassify).

The type of a $\texttt{read}_{\texttt{L}}(\texttt{e})$ expression contains the security levels of the statement combined with the labels on the file descriptor argument. Note that all read operations are assumed to return integers. A secrecy checking constraint is also added to the constraint set. There are two reasons for this. Firstly, this ensures that low reads are not happening under high guards; as discussed in Section 2.1, this may cause an information leak (note the type of any sub-expression implicitly contains the types of the program counter, a fact easily shown by structural induction on `e`, observing the base cases all add pc to the types). Secondly, if the file descriptor value has a higher label than the channel policy, performing the read may result in a security leak (*e.g.*, two executions that differ only in high inputs may read from different low channels, since the file descriptor for the channel differs).

3.2.2 Statement Typing

The type rules for statements are given in Figure 3.4. In rule (If), the secrecy type of the condition is added to the program counter when typing each branch; fresh type variables are created

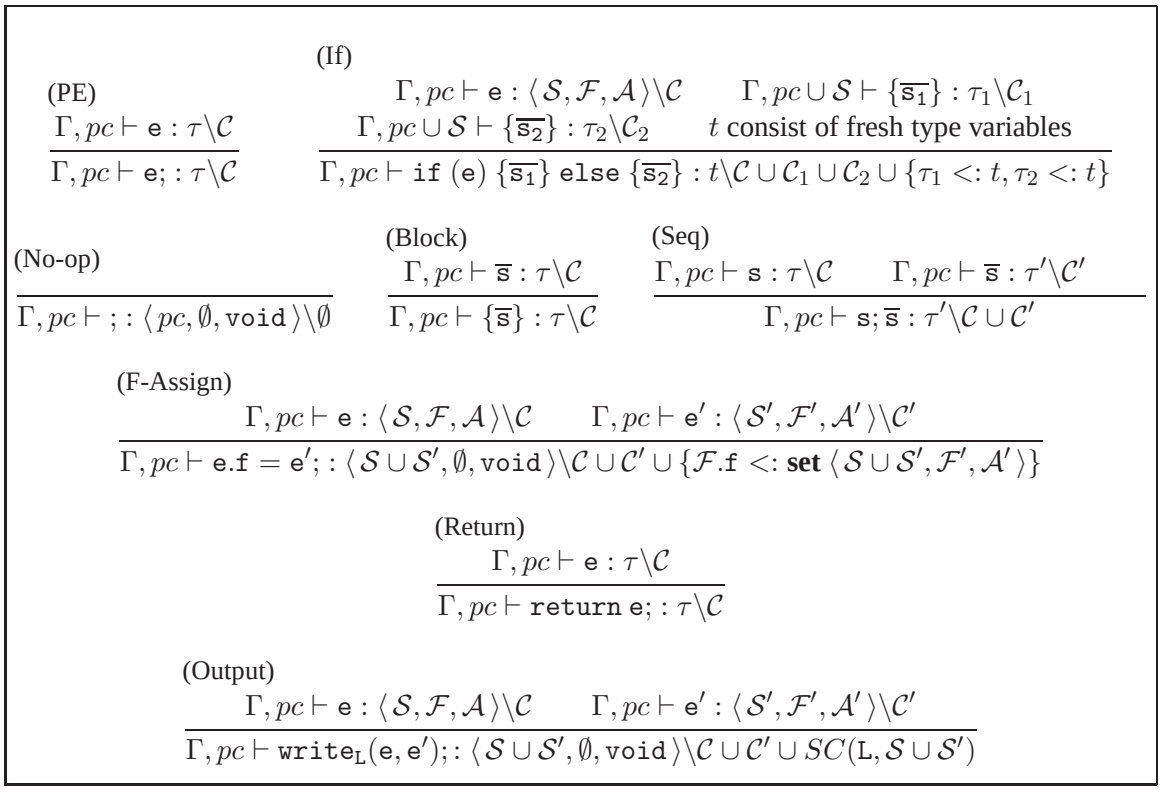


Figure 3.4: Label Type Rules for Statements

to merge the flows from the results of the two branches (we use type variables and constraints here instead of unioning the secrecy types to simplify our noninterference proof in Chapter 4). (F-Assign) adds a **set** constraint to the constraint set to delineate the flow of labels into an object field; these constraints are described in section 3.2.5. Typing a $\mathtt{write_L}(\mathtt{e}, \mathtt{e}')$ statement using (Output) produces a secrecy check constraint to ensure the type of the output aligns with the policy of the channel. The type of the file descriptor is also checked against the policy for the same reasons as `read`, discussed earlier. The rules (PE) and (Return) simply pass along the type of the expression being promoted or returned, respectively. Since ; is a no-op, (No-op) gives it the secrecy type of the program counter, an empty field type, and a `void` α-type.

Fields:

$$\frac{}{fields(\texttt{Object}) = \emptyset} \qquad \frac{}{fields(constants) = \emptyset}$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C f}}; \texttt{K } \overline{\texttt{M}}\} \qquad fields(\texttt{D}) = \overline{\texttt{D g}}}{fields(\texttt{C}) = \overline{\texttt{D g}}, \overline{\texttt{C f}}}$$

Super:

$$\frac{\Gamma, pc \vdash \texttt{e} : \tau \backslash \mathcal{C} \qquad \Gamma, pc \vdash \overline{\texttt{e}} : \overline{\tau} \backslash \overline{\mathcal{C}} \qquad \tau = \langle\, \mathcal{S}, \mathcal{F}, \mathcal{A} \,\rangle}{\Gamma, pc \vdash \texttt{e.super(D, } \overline{\texttt{e}}\texttt{)}; : t_r \backslash \mathcal{C} \cup \overline{\mathcal{C}} \cup \{\texttt{D.K}(\overline{\tau}, \tau \xrightarrow{pc \cup \mathcal{S}} t_r)\}} \text{(Super)}$$

$$\frac{\Gamma, pc \vdash \texttt{e} : \langle\, \mathcal{S}, \mathcal{F}, \mathcal{A} \,\rangle \backslash \mathcal{C}}{\Gamma, pc \vdash \texttt{e.super(Object)}; : \langle\, \mathcal{S}, \emptyset, \texttt{void} \,\rangle \backslash \mathcal{C}} \text{(Super')}$$

Figure 3.5: Auxiliary Definitions for Label Type Rules

3.2.3 Class and Program Typing

Type inference rules for typing programs, classes, and methods are found in Figure 3.6. Typing of a whole program first requires each of the classes to be typed, and these types placed in a label table, LT; then the statements corresponding to `main` are typed.

The body of each method is typed with respect to fresh label variables for the arguments and `this`. As previously noted, methods and constructors are given $\forall$ types so that they may vary polymorphically, and these types are instantiated when computing the constraint closure. For this reason, any free (local) type variables occurring in the typing are found as $\overline{t_l}$, so the type variables are quantified by $\forall \overline{t}$. The function $FTV(\cdot)$ extracts all of the type variables of its argument. These type variables that are local to the method (or constructor) body must be properly instantiated during the constraint closure, so the typing will not mix flows for different calls to the same method. Method typing then fills in the constraint types in the label table, and the constraint $\{\tau <: t_r\}$ is added, since

Method Typing:

$$\frac{\begin{array}{c}\texttt{C}_0 = \texttt{class C extends D } \{\overline{\texttt{C f}}; \texttt{K } \overline{\texttt{M}}\} \qquad \texttt{M} = \texttt{RT m}(\overline{\texttt{C x}})\ \{\overline{\texttt{s}}\} \\ \overline{t_x}, t_t, t_r, s_p \text{ consist of fresh variables.} \\ [\overline{\texttt{x}} : \overline{t_x}, \texttt{this} : t_t], s_p \vdash \overline{\texttt{s}} : \tau \backslash \mathcal{C} \qquad \overline{t_l} = FTV(\mathcal{C} \cup \{\tau <: t_r\}) - FTV(\overline{t_x}, t_t, s_p, t_r)\end{array}}{\vdash_M (\texttt{C}_0, \texttt{M}) : \forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C} \cup \{\tau <: t_r\}}$$

Constructor Typing:

$$\frac{\begin{array}{c}\texttt{C}_0 = \texttt{class C extends D } \{\overline{\texttt{C f}}; \texttt{K } \overline{\texttt{M}}\} \qquad \texttt{K} = \texttt{C}(\overline{\texttt{C x}})\ \{\texttt{super}(\overline{\texttt{e}}); \overline{\texttt{s}}\} \\ \overline{t_x}, t_t, t_r, s_p \text{ consist of fresh variables.} \qquad [\overline{\texttt{x}} : \overline{t_x}, \texttt{this} : t_t], s_p \vdash \texttt{this.super}(\texttt{D}, \overline{\texttt{e}}); \overline{\texttt{s}}; : \tau \backslash \mathcal{C} \\ \overline{t_l} = FTV(\mathcal{C} \cup \{\tau <: t_r\}) - FTV(\overline{t_x}, t_t, s_p, t_r)\end{array}}{\vdash_M (\texttt{C}_0, \texttt{K}) : \forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C} \cup \{\tau <: t_r\}}$$

Class Typing:

$$\frac{\vdash_M (\texttt{C}_0, \texttt{K}) : \kappa_0 \qquad \vdash_M (\texttt{C}_0, \overline{\texttt{M}_0}) : \overline{\kappa}_0 \qquad \ldots}{\vdash_C LT[(\texttt{C}_0, \texttt{K}) : \kappa_0, (\texttt{C}_0, \overline{\texttt{M}_0}) : \overline{\kappa}_0, \ldots]}$$

Program Typing:

$$\frac{\begin{array}{c}\vdash_C LT[(\texttt{C}_0, \texttt{K}) : \kappa_0, (\texttt{C}_0, \overline{\texttt{M}_0}) : \overline{\kappa}_0, \ldots] \qquad \emptyset, \emptyset, u \vdash \overline{\texttt{s}} : \tau \backslash \mathcal{C} \\ \mathit{Closure}(LT[(\texttt{C}_0, \overline{\texttt{M}_0}) : \overline{\kappa}_0, \ldots], \mathcal{C}) \text{ is consistent}\end{array}}{\vdash_P \{\texttt{C}_0, \texttt{C}_1, \ldots\};\ \overline{\texttt{s}} : \tau \backslash \mathcal{C}}$$

Figure 3.6: Label Type Rules for Classes and Programs

t_r appears in the label table as an abstract indication of the return type of the method, which must be bound by the type of the method body, τ.

The constructor rule types constructors in a similar manner to method bodies. The call to `super` and the body of each constructor is typed with respect to fresh label variables for the arguments. Like method typing, these constructor types are then added to the label table. The typing of the call to `super` here is a bit unintuitive; the typing is formulated in this manner for two reasons: to align the typing with methods, and to align the typing with the operational semantics (defined in Chapter 4); both alignments significantly simplify our proofs. The call to `super` here includes the name of the class to which the call is being made, so that `super` calls can be properly

made up the inheritance hierarchy. Unlike in MJ, our semantics does not use stack frames and scoping mechanisms (this simplifies our semantics and proofs), so the name of the constructor gets hard-coded (see Chapter 4). Calls to `super` are typed by (Super) in Figure 3.5, as one would expect given the typing rules (Invoke) and (New). One final caveat is that `e.super(C,e)` must be added to the syntactic definition of `s` in order for this typing to be valid using (Seq); while the type system requires this, we opt not to include this in the definition in Figure 3.1, since it is not a valid statement for a program; it is merely a convenience for our type system, semantics and proof.

Programs are then typed by typing each class definition, which types each method definition. $\overline{\texttt{s}}$, representing `main` is also typed.

3.2.4 Local Variables

In Section 3.1, we mentioned that local variables are removed from our formal language definition. This removal is necessary so that we may define a simpler substitution-style operational semantics instead of a stack-based semantics, which MJ uses. As we shall see in Chapter 4, the formal semantics and proofs are already quite lengthy and complex; the addition of a stack would add another layer of complexity to the formalism with little gain in showing that our approach is correct.

Although we do not formally show their correctness, the addition of local variables to the type inference system is quite simple. Figure 3.7 shows the necessary additions. Firstly, variable declarations must be added to the syntax. Hence, statements `s` are extended to include class and primitive variable definitions. Like in the MJ type system, we introduce a new type rule for sequences (Var-Decl), which handles the variable declaration statement; the type rule simply creates a fresh set of type variables t for the declared variable `x`, then types the remainder of the sequence

Definitions:

$\mathtt{T} ::= \mathtt{C} \mid \mathtt{int} \mid \mathtt{boolean}$

$\mathtt{s} ::= \ldots \mid \mathtt{T\ x;} \mid \mathtt{x = e;}$

Type Rules:

(Var-Decl)

$$\frac{\Gamma[\mathtt{x} : t], pc \vdash \overline{\mathtt{s}} : \tau \backslash \mathcal{C} \qquad t \text{ consists of fresh variables}}{\Gamma, pc \vdash \mathtt{T\ x;}\ \overline{\mathtt{s}} : \tau \backslash \mathcal{C}}$$

(Var-Assign)

$$\frac{\Gamma(\mathtt{x}) = \langle\, s, f, \alpha \,\rangle \qquad \Gamma, pc \vdash \mathtt{e} : \tau \backslash \mathcal{C} \qquad \tau = \langle\, \mathcal{S}, \mathcal{F}, \mathcal{A} \,\rangle}{\Gamma, pc \vdash \mathtt{x = e;} : \langle\, s \cup \mathcal{S}, \emptyset, \mathtt{void} \,\rangle \backslash \mathcal{C} \cup \{\langle\, s, f, \alpha \,\rangle <: \textbf{set}\ \tau\}}$$

(Seq)

$$\frac{\Gamma, pc \vdash \mathtt{s} : \tau \backslash \mathcal{C} \qquad \Gamma, pc \vdash \overline{\mathtt{s}} : \tau' \backslash \mathcal{C}' \qquad \mathtt{s} \neq \mathtt{T\ x}}{\Gamma, pc \vdash \mathtt{s}; \overline{\mathtt{s}} : \tau' \backslash \mathcal{C} \cup \mathcal{C}'}$$

Figure 3.7: Label Type Rules for Local Variables

under a type environment extended with the type for x. This new rule also necessitates the change to (Seq), that the first statement in the typing is not a variable declaration. (Var-Assign) is much like the field assignment typing rule, adding a **set** constraint to the constraint set showing that the type of e flows into the type of x (see Section 3.2.5 for a description of **set** constraints).

3.2.5 Set Constraints

We use **set** constraints for field assignment in (F-Assign), and variable assignment in (Var-Assign), where **set** τ is the type for an assignment. Defining assignments in this manner, and using the (Set) closure rule (in Figure 3.9), ensures a one-directional flow of information from assignments to accesses, never in the opposite direction [SW00, WS01]. This constraint $\tau <:$ **set** τ' means τ is the type of the field/variable, which flows into **set** τ', the type of the assignment. From this constraint, using (Set), the closure contains the constraint $\tau' <: \tau$, indicating that the type of what

is assigned to the field/variable flows into the field/variable type, obtaining the correct type when the field/variable is accessed. Instead of generating this constraint directly in the type rule, the **set** constraint makes the type of the assignment contravariant, allowing closure rules to be applied directly to the **set** constraint. For example, if we have constraints $\tau <: t$ and $t <:$ **set** τ', the closure will contain $\tau <:$ **set** τ'. An example of this contravariance is given below.

Using these **set** constraints ensures that the type of a field access will not flow to the type of a field assignment (a backward flow). With the constraint $t <:$ **set** τ, the closure will generate $\tau <: t$ by (Set), but not $t <: \tau$, which may cause a backward flow. Let us now study the following example that illustrates the need for these constraints, where a backward flow could result in imprecision in the typing.

```
x = read_{low}(fd);
y = x;
y = read_{high}(fd');
```

Suppose in the type environment we have $\Gamma(\texttt{x}) = t_x$ and $\Gamma(\texttt{y}) = t_y$. Since `x` never contains `high` data, the type system should not produce a constraint $\{\texttt{high}\} <: s_x$. Let us examine the typing and closure to see that this cannot occur; we look only at secrecy types and omit any other typings to clearly make the point. The first statement produces the constraint $s_x <:$ **set** $\{\texttt{low}\}$. The second statement produces $s_y <:$ **set** s_x from (Var-Assign). The final statement produces $s_y <:$ **set** $\{\texttt{high}\}$. Hence, by (Set) we have $\{\texttt{low}\} <: s_x$, $s_x <: s_y$, and $\{\texttt{high}\} <: s_y$. By transitivity, we have $\{\texttt{low}\} <: s_y$, but the direction of the flow does *not* allow $\{\texttt{high}\} <: s_x$, which is correct, since `x` never contains `high` data.

The contravariance of **set** is necessary for generating the correct constraints during aliasing. Consider the following example code, along with the relevant constraints the typing generates.

`c2 = c1;`	$f_{c2} <:$ **set** f_{c1}
`c2.x =` $\texttt{read}_{\{\texttt{high}\}}(\texttt{fd}')$;	$f_{c2}.\texttt{x}.\mathsf{S} <:$ **set** $\{\texttt{high}\}$
`z = c1.x;`	$s_z <:$ **set** $f_{c1}.\texttt{x}.\mathsf{S}$

Here, `c2` and `c1` are aliased, so we need to ensure that the closure contains the constraint $\{\texttt{high}\} <: s_z$, where s_z is the secrecy type of `z`. If we did not have the **set** constraint, this would not happen. However, with the **set** constraint, the correct constraints are generated: $f_{c1} <: f_{c2}$, $f_{c1}.\texttt{x}.\mathsf{S} <:$ **set** $\{\texttt{high}\}$, $\{\texttt{high}\} <: f_{c1}.\texttt{x}.\mathsf{S}$, $f_{c1}.\texttt{x}.\mathsf{S} <: s_z$, $\{\texttt{high}\} <: s_z$.

3.2.6 Polymorphic Instantiations

As described in Section 2.6, we require a high degree of polymorphism in order to distinguish security policies of the IO classes and for permitting code re-use across multiple security domains. Since this polymorphism is such an important part of our system, we first give a simplified description of the polymorphism before describing the actual constraint closure in the next section.

We use a form of parametric polymorphism that is closely related to the Cartesian Product Algorithm (CPA) [Age95], and Data-Polymorphic CPA (DCPA) [SW00, WS01]. We now describe how this polymorphism works by looking at typing in our system without the secrecy or field types, since the α-type carries the necessary type information for polymorphic instantiation. Hence, in this section types τ are simply $\mathcal{A}$.

The polymorphic analysis requires a polymorphic type for each method, and an instantiation of this type when a method is invoked. We call each of these polymorphic instantiations a *contour*. As previously noted, method types in the label table are universally quantified, $\forall\overline{\alpha}.\overline{\alpha_l}, \overline{\alpha_x}, \alpha_t \rightarrow \alpha_r \backslash \mathcal{C}$, so they may vary parametrically; $\overline{\alpha_l}$ represent all local type variables in the method body typing, $\overline{\alpha_x}$ represent the method arguments, α_t represents `this`, and α_r represents the return type.

Method constraints $\mathcal{A}.\mathtt{m}(\overline{\tau}, \tau_t \rightarrow \tau_r)$ represent the type of a call site (invocation), and contain the necessary information for polymorphic instantiation of method types. $\mathcal{A}$ is the α-type (representing the class) of the object being invoked with the method `m`; $\overline{\tau}$ are the types of the explicit arguments, τ_t is the type of `this`, and τ_r is the return type.

Using the (Invoke) type rule, a new method constraint is generated at each call-site, containing the types of the object on which the call is being made, all arguments, and a unique return type so that the type of each call-site will be distinct. At constraint closure time, the rule $(Method)$ (in Figure 3.9) ensures that each method constraint generates a unique polymorphic instantiation for every possible concrete class that may be used at each call-site. Hence, a new contour is created for distinct concrete classes on which the call is being made, distinct argument types, and distinct call-sites (represented by the return type).

Let us now consider the polymorphic typing of the example program in Figure 3.8. The class `C` contains a method `m` that takes a `FileIS` as argument. When this method body is typed, we know only that `in` is a `FileIS`, but it may be any subclass of `FileIS`, which may have different types (as described in Section 2.6, it is important for our analysis to distinguish the IO classes, which may have different security policies). Our polymorphic analysis statically approximates which concrete classes `in` may be, based on wherever `m` is invoked. The analysis is expressive enough to distinguish the calls on lines 19 and 20, and therefore the separate the calls to the different `read()` methods. We now go through the typing of this example program in the simplified system with only α-types.

The methods of each class are first typed according to the type rules. We don't give the types of `LowFileIS` and `HighFileIS`, as we are just showing how the type system separates the `read()` calls.

```
1:  class LowFileIS extends FileIS {
2:   int read() { return read∅(0); }
3:  }
4:  class HighFileIS extends FileIS {
5:   int read() { return read{high}(1); }
6:  }
7:  class C extends Object {
8:   int m(FileIS in) {
9:    return in.read();
10:  }
11: }
12: main() {
13:  FileIS hin;
14:  FileIS lin;
15:  C c;
16:  hin = new HighFileIS();
17:  lin = new LowFileIS();
18:  c = new C();
19:  c.m(hin);
20:  c.m(lin);
21: }
```

Figure 3.8: Example Program: Polymorphic Types

The method typing rule in Figure 3.6 types the method body of `m` in class `C` as follows, using rules (Invoke) and (Return).

$$[\texttt{in} : \alpha_{in}, \texttt{this} : \alpha_t] \vdash \texttt{return in.read}(); : \alpha_r \backslash \alpha_{in}.\texttt{read}(\alpha_{in} \rightarrow \alpha_r)$$

This typing produces the following $\forall$-quantified method type, which is put into the label table. Here, α_r is a variable that is local to the typing, α_{in} is the argument type, α_t is the type of `this`, and α_m is the return type.

$$(\texttt{C}, \texttt{m}) = \forall \overline{\alpha}.\alpha_r, \alpha_{in}, \alpha_t \rightarrow \alpha_m \backslash\ \alpha_{in}.\texttt{read}(\alpha_{in} \rightarrow \alpha_r) \cup \{\alpha_r <: \alpha_m\}$$

Now, for typing the `main` of the program, let $\Gamma = [\texttt{hin} : \alpha_h, \texttt{lin} : \alpha_l, \texttt{c} : \alpha_c]$ as produced by the (Var-Decl) rules. The remaining types are as follows.

16: $\Gamma \vdash \texttt{hin} = \texttt{new HighFileIS}(); : \texttt{void} \backslash \{\alpha_h <: \texttt{set } \alpha_4\} \cup \{\texttt{HighFileIS} <: \alpha_4\} \cup \{\texttt{HighFileIS.K}(\alpha_4 \rightarrow \alpha_4')\}$

17: $\Gamma \vdash \texttt{lin} = \texttt{new LowFileIS}(); : \texttt{void} \backslash \{\alpha_l <: \texttt{set } \alpha_5\} \cup \{\texttt{LowFileIS} <: \alpha_5\} \cup \{\texttt{LowFileIS.K}(\alpha_5 \rightarrow \alpha_5')\}$

18: $\Gamma \vdash \texttt{c} = \texttt{new C}(); : \texttt{void} \backslash \{\alpha_c <: \texttt{set } \alpha_3\} \cup \{\texttt{C} <: \alpha_3\} \cup \{\texttt{C.K}(\alpha_3 \rightarrow \alpha_3')\}$

19: $\Gamma \vdash \texttt{c.m(hin)} : \alpha_1 \backslash \alpha_c.\texttt{m}(\alpha_h, \alpha_c \rightarrow \alpha_1)$

20: $\Gamma \vdash \texttt{c.m(lin)} : \alpha_2 \backslash \alpha_c.\texttt{m}(\alpha_l, \alpha_c \rightarrow \alpha_2)$

It is important to observe here that the typing of lines 19 and 20 produce method constraints with unique argument types α_h and α_l, since the arguments differ, and unique return types α_1 and α_2, which distinguish the different call-sites to method m

After typing is complete, the polymorphic instantiations occur during the constraint closure. Firstly, by (Set) and (Trans), $\alpha_3 <: \alpha_c$ and $\texttt{C} <: \alpha_c$ are in the closed constraint set. This shows that a class C flows into the variable c, meaning α_c can be a concrete class C.

Instantiations of the calls to m are now as follows. Since α_c can be a concrete class C according to the constraint $\texttt{C} <: \alpha_c$, using closure rule $(Method)$ on this constraint and $\alpha_c.\texttt{m}(\alpha_h, \alpha_c \rightarrow \alpha_1)$ produces a contour of the method call that is based on the class C and the call-site information recorded in the method constraint. Hence, the type variables occurring in the method type of $(\texttt{C}, \texttt{m})$ in the label table are replaced. The argument and return types in the method constraint replace the variables for argument and return type variables, respectively. The function θ replaces all type variables that are local to the method type, in order to not mix flows with different calls. Since we are distinguishing calls based on the concrete class, method argument, and call-sites, the superscript provides this distinction and makes the local variables specific to this contour. So, applying closure rule $(Method)$ yields the following constraints.

$$[\alpha_r \mapsto \alpha_r^{\texttt{C,m},\alpha_h,\alpha_c,\alpha_1}][\alpha_{in} \mapsto \alpha_h, \alpha_t \mapsto \alpha_c, \alpha_m \mapsto \alpha_1](\alpha_{in}.\texttt{read}(\alpha_{in} \rightarrow \alpha_r) \cup \{\alpha_r <: \alpha_m\})$$

which, after performing the substitutions is

$\alpha_h.\texttt{read}(\alpha_h \rightarrow \alpha_r^{\texttt{C},\texttt{m},\alpha_h,\alpha_c,\alpha_1}) \cup \{\alpha_r^{\texttt{C},\texttt{m},\alpha_h,\alpha_c,\alpha_1} <: \alpha_1\}$

In a similar manner, using closure rule $(Method)$ on constraints $\texttt{C} <: \alpha_c$ and $\alpha_c.\texttt{m}(\alpha_l, \alpha_c \rightarrow \alpha_2)$ produces a different instantiation of the polymorphic type of m, since the method argument and call-site differ.

$[\alpha_r \mapsto \alpha_r^{\texttt{C},\texttt{m},\alpha_l,\alpha_c,\alpha_2}][\alpha_{in} \mapsto \alpha_l, \alpha_t \mapsto \alpha_c, \alpha_m \mapsto \alpha_2](\alpha_{in}.\texttt{read}(\alpha_{in} \rightarrow \alpha_r) \cup \{\alpha_r <: \alpha_m\})$

which, after performing the substitutions is

$\alpha_l.\texttt{read}(\alpha_l \rightarrow \alpha_r^{\texttt{C},\texttt{m},\alpha_l,\alpha_c,\alpha_2}) \cup \{\alpha_r^{\texttt{C},\texttt{m},\alpha_l,\alpha_c,\alpha_2} <: \alpha_2\}$

Hence, after instantiating each of the method invocations with contours, we have the following two method constraints.

$\alpha_h.\texttt{read}(\alpha_h \rightarrow \alpha_r^{\texttt{C},\texttt{m},\alpha_h,\alpha_c,\alpha_1}) \cup \{\alpha_r^{\texttt{C},\texttt{m},\alpha_h,\alpha_c,\alpha_1} <: \alpha_1\}$

$\alpha_l.\texttt{read}(\alpha_l \rightarrow \alpha_r^{\texttt{C},\texttt{m},\alpha_l,\alpha_c,\alpha_2}) \cup \{\alpha_r^{\texttt{C},\texttt{m},\alpha_l,\alpha_c,\alpha_2} <: \alpha_2\}$

Notice that these two type constraints have no type variables in common, as even $\alpha_r^{\texttt{C},\texttt{m},\alpha_h,\alpha_c,\alpha_1}$ and $\alpha_r^{\texttt{C},\texttt{m},\alpha_l,\alpha_c,\alpha_2}$ are different since their superscripts are different. These two constraints show the main goal of our polymorphic analysis, separating the types of different calls to the same method. Since the closure rules will generate $\texttt{HighFileIS} <: \alpha_h$ and $\texttt{LowFileIS} <: \alpha_l$, these two read calls will be distinct, as the contours ensure that their flows will not merge. This will allow the full type system to distinguish the security typings of the high and low input stream objects across method calls.

3.2.7 Label Closure

The closure rules for label constraint sets are given in Figure 3.9 with auxiliary definitions for method closures in Figure 3.10. The rules in Figure 3.9 add new constraints based on transitivity, obvious set propagations, and field labels. (Set) closure rules are discussed in Section 3.2.5. The

$$\frac{\tau <: \textbf{set}\ \tau'}{\tau' <: \tau}(\text{Set}) \qquad \frac{\mathcal{S}_1 <: \mathtt{S} \qquad (\mathtt{S} \cup \mathcal{S}_2 - \mathtt{L}) <: \mathcal{S}_3}{(\mathcal{S}_1 \cup \mathcal{S}_2 - \mathtt{L}) <: \mathcal{S}_3}(\mathcal{S}\text{-Trans})$$

$$\frac{\mathcal{S}_1 \cup \mathcal{S}_2 <: \mathcal{S}_3}{\mathcal{S}_1 <: \mathcal{S}_3 \qquad \mathcal{S}_2 <: \mathcal{S}_3}(\mathcal{S}\text{-Union}) \qquad \frac{\mathcal{F}_1 <: \mathsf{f} \qquad \mathsf{f} <: \mathcal{F}_2}{\mathcal{F}_1 <: \mathcal{F}_2}(\mathcal{F}\text{-Trans})$$

$$\frac{\mathcal{F} <: \mathsf{f} \qquad (\mathsf{f}.\mathtt{f}.\mathsf{S} \cup \mathcal{S}_1 - \mathtt{L}) <: \mathcal{S}_2}{(\mathcal{F}.\mathtt{f}.\mathsf{S} \cup \mathcal{S}_1 - \mathtt{L}) <: \mathcal{S}_2}(\mathcal{S}\text{-Field}) \qquad \frac{\mathcal{F} <: \mathsf{f} \qquad \mathsf{f}.\mathtt{f}.\mathsf{F} <: \mathcal{F}_1}{\mathcal{F}.\mathtt{f}.\mathsf{F} <: \mathcal{F}_1}(\mathcal{F}\text{-Field})$$

$$\frac{\mathcal{F} <: \mathsf{f} \qquad \mathsf{f}.\mathtt{f}.\mathsf{A} <: \mathcal{A}_1}{\mathcal{F}.\mathtt{f}.\mathsf{A} <: \mathcal{A}_1}(\mathcal{A}\text{-Field}) \qquad \frac{\mathsf{f} <: \mathcal{F} \qquad \mathcal{S} <: \mathsf{f}.\mathtt{f}.\mathsf{S}}{\mathcal{S} <: \mathcal{F}.\mathtt{f}.\mathsf{S}}(\mathcal{S}\text{-Field'})$$

$$\frac{\mathsf{f} <: \mathcal{F} \qquad \mathcal{F}' <: \mathsf{f}.\mathtt{f}.\mathsf{F}}{\mathcal{F}' <: \mathcal{F}.\mathtt{f}.\mathsf{F}}(\mathcal{F}\text{-Field'}) \qquad \frac{\mathsf{f} <: \mathcal{F} \qquad \mathcal{A} <: \mathsf{f}.\mathtt{f}.\mathsf{A}}{\mathcal{A} <: \mathcal{F}.\mathtt{f}.\mathsf{A}}(\mathcal{A}\text{-Field'})$$

$$\frac{\mathcal{A}_1 <: \mathsf{i} \qquad \mathsf{i} <: \mathcal{A}_2}{\mathcal{A}_1 <: \mathcal{A}_2}(\mathcal{A}\text{-Trans})$$

$$\frac{\begin{array}{c}\mathtt{C} <: \mathcal{A} \qquad \mathcal{A}.\mathtt{m}(\overline{\tau}, \tau_t \xrightarrow{pc} \tau_r) \qquad mtype(\mathtt{C},\mathtt{m}) = \forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C} \\ \overline{\tau} = \langle\, \mathcal{S}, \mathcal{F}, \mathcal{A} \,\rangle \qquad \tau_t = \langle\, \mathcal{S}_t, \mathcal{F}_t, \mathcal{A}_t \,\rangle \qquad \tau_r = \langle\, \mathcal{S}_r, \mathcal{F}_r, \mathcal{A}_r \,\rangle \\ \overline{t_l'} = \theta(\overline{t_l}, \mathtt{C}, \mathtt{m}, \overline{\mathcal{A}}, \mathcal{A}_t, \mathcal{A}_r)\end{array}}{[\overline{t_l} \mapsto \overline{t_l'}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{\tau}, t_t \mapsto \tau_t, t_r \mapsto \tau_r]\mathcal{C}}(Method)$$

$$\frac{SC(\mathtt{L}, \mathcal{S}_2 \cup (\mathtt{S} \cup \mathcal{S}_3 - \mathtt{L}')) \qquad \mathcal{S}_1 <: \mathtt{S}}{SC(\mathtt{L}, \mathcal{S}_2 \cup (\mathcal{S}_1 \cup \mathcal{S}_3 - \mathtt{L}'))}(\text{SC-Trans})$$

$$\frac{\mathcal{F} <: \mathsf{f} \qquad SC(\mathtt{L}, \mathcal{S}_1 \cup (\mathsf{f}.\mathtt{f}.\mathsf{S} \cup \mathcal{S}_2 - \mathtt{L}'))}{SC(\mathtt{L}, \mathcal{S}_1 \cup (\mathcal{F}.\mathtt{f}.\mathsf{S} \cup \mathcal{S}_2 - \mathtt{L}'))}(\text{SC-Field})$$

Figure 3.9: Label Closure Rules

($\mathcal{S}$-Trans) and ($\mathcal{S}$-Field) rules are non-standard in order to account for declassification using the set difference operator (note L can be empty when there is no declassification). Declassified labels cannot be separated from the type of what they are declassifying.

The closure rule $(Method)$, defined in Figure 3.9 is important for creating polyinstantiations of method types. As discussed in Section 3.2.1, method constraints are added during method invo-

$$\frac{LT(\mathtt{C},\mathtt{m}) = \forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}}{mtype(\mathtt{C},\mathtt{m}) = \forall \overline{t'}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \text{ is not defined in } \overline{\mathtt{M}}}{mtype(\mathtt{C},\mathtt{m}) = mtype(\mathtt{D},\mathtt{m})}$$

$$\theta(t, \mathtt{C}, \mathtt{m}, \overline{\mathcal{A}}, \mathcal{A}_t, \mathcal{A}_r) = t^{\mathtt{C},\mathtt{m},flatten(\overline{\mathcal{A}},\mathcal{A}_t,\mathcal{A}_r)} \qquad flatten(\mathcal{A}^\sigma) = \mathcal{A}$$

$$flatten(x, y, \dots) = flatten(x), flatten(y), \dots$$

Figure 3.10: Label Closure Auxiliary Definitions

cation (also for calls to constructors), when the actual class of the object on which the method is being called may be unknown. Thus, for all constraints $\mathtt{C} <: \mathcal{A}$, where C is a concrete class, the method m is looked up in LT via *mtype*, which returns a typing for that method as found either in C or in a superclass if not defined in C. We then substitute the labels in the *method* constraint into this constraint set from the label table, and replace all local label variables as defined by the function θ.

As described in Section 3.2.6, the manner in which local label variables are replaced defines the contours that distinguish different method invocations. Our definition of θ creates a new contour for each distinct receiver type C, method name m, argument type $\overline{\mathcal{A}}$ ($\mathcal{A}_t$ is the type of this), and return type $\mathcal{A}_r$. This allows calls to be distinguished based on receiver and argument types, as in CPA [Age95], and additionally distinguishes call-sites based on unique program points. Since the (Invoke) type rule creates fresh variables for each method invocation, this serves as a unique marker of the call-site in the program; thus, $\mathcal{A}_r$ is the call-site of the method. Since constructor calls during (New) are similar to method invocations, the analysis can distinguish most object instances via call-sites and constructor arguments.

```
1:  class C extends Object {
2:   int f;
3:   C() { super(); f = 0;}
4:   void put(int a) { this.f = a;}
5:   int get() { return this.f;}
6:  }
7:  main() {
8:   ...
9:   x = new C();
10:  y = new C();
11:  x.put(read_{low}(fd));
12:  y.put(read_{high}(fd'));
13:  write_{low}(x.get(), fd'');
14: }
```

Figure 3.11: Example Program: Object Contours

Consider the example in Figure 3.11. Our analysis produces separate contours for the creation of x and y, where CPA merges them into one. Even though the put calls have different contours, since the types of x and y are of the same class, but are not distinguished as different objects, the CPA analysis cannot determine that x.get() is low. We obtain more precision, so we can correctly identify the flows of data into and out-of abstract objects on the heap.

This precision is similar to that obtained in data-polymorphic CPA analysis [SW00, WS01]; although DCPA includes many optimizations to combine contours whenever possible, while still supporting data polymorphism. The *flatten* function is necessary to merge contours for recursive calls and to ensure the analysis terminates. We discuss the termination of this algorithm in section 3.2.9.

Figure 3.9 also contains transitivity and field type accesses within SC constraints. This is to create the correct check constraints with the proper security labels.

We formally define a constraint closure as follows.

Definition 3.1 (Constraint Closure) *$Closure(LT, \mathcal{C})$ is defined as the least set that includes $\mathcal{C}$ and any constraint that can be derived from $\mathcal{C}$ by the rules of Figure 3.9.*

Notice that the closure of a constraint set is with respect to a label table LT; this is due to the lookup of the method (and constructor) typing in this table. If, some constraint set $\mathcal{C} = Closure(LT, \mathcal{C})$, we will often simply say that $\mathcal{C}$ is *closed*.

3.2.8 Inconsistent Constraints

Inconsistencies in the label constraint sets come from SC constraints, which are security checks that are generated when IO operations are typed. Constraint consistency is defined as follows.

Definition 3.2 (Inconsistent Constraints) *An inconsistent constraint is any constraint $SC(\mathtt{L}_s, \mathtt{L}'_s)$, where $\mathtt{L}'_s \not\subseteq \mathtt{L}_s$.*

Note that consistency of check constraints is defined only on concrete constraint sets, which are formed during the closure after all transitive flows into type variables have been considered. This leads us to the following definition of a consistent constraint set.

Definition 3.3 (Consistent Constraint Set) *A constraint set $\mathcal{C}$ is consistent iff $\mathcal{C}$ does not contain any inconsistent constraints.*

If $Closure(LT, \mathcal{C})$ contains an inconsistent constraint, then the closure is inconsistent, and type inference fails. SC constraints enforce secrecy policies. In the constraint $SC(\mathtt{L}, \mathtt{L}')$, $\mathtt{L}$ is the secrecy policy of the IO channel, and $\mathtt{L}'$ is the set of labels on the data at that point. Proper enforcement of the policy requires the labels on the data to be a subset of the labels on the IO

channel. For example, the constraint $SC(\{\texttt{high}, \texttt{low}\}, \{\texttt{low}\})$ is consistent, with low data flowing to a high channel; $SC(\{\texttt{low}\}, \{\texttt{high}\})$ is inconsistent, since high data is flowing to a low channel.

3.2.9 Typing Complexity and Termination

A potential pitfall of this form of type inference algorithm is non-termination, if contours are continually created for recursive method invocations. Our analysis merges contours for recursive calls, ensuring termination. We now address the complexity of type inference and constraint closure computation.

Inferring types completes in linear time. Closing the constraint set can be exponential in the worst case. This is evident from the definition of θ. t inputs to θ are all flat (i.e. have no superscript), since they are the free type variables that occur when typing method `m` of class `C`. Superscripted variables are only added during the closure. This means t is bounded by n, the size of the program. Since $\overline{\mathcal{A}}, \mathcal{A}_t$, and $\mathcal{A}_r$ are all flattened, the number of possibilities for these values is bounded by the number of concrete classes and the number of fresh variables created in the program, which are each less than n. Thus, in the worst case, we may create up to n^{n^c} contours, where constant c is the maximum length of a superscript `C`,`m`,$\overline{\mathcal{A}}, \mathcal{A}_t$,$\mathcal{A}_r$ (c is bound by the maximum number of arguments that may be passed to a method in the program). This is a weak bound and a large exponential, but nevertheless shows that the analysis terminates. Many optimizations (*e.g.* combining contours and constraint garbage collection) can be performed to make this practical, as shown in [SW00, WS01] and elsewhere; this is out of the scope of the current work.

3.2.10 Modularity

The type inference system provides separate compilation of classes, since type inference can be done separately, and the final global constraint set must be closed and checked for inconsistencies. Classes and methods are analyzed only once, and their types and constraints built into the label table, which may be re-used for any number of programs. This means our system is not entirely modular, since the constraint closure is global. Hence, we assume that the source code is available to be analyzed by the type system; this is not a problem, since Java is not modular, so our type system can be used in the context of the current Java model.

Even though Java is not modular, we now discuss some of the issues related to modularity for our information flow system. Completely modularizing compilation requires trust in compiled libraries to have been correctly analyzed; so libraries would have to be signed by a trusted party, who asserts the correctness of the library. This adds another level of trust to the system, as the signatures must be trustworthy and unforgeable. In addition, complete modularity imposes additional restrictions on the expressiveness on the type system and inheritance hierarchy. In Section 2.6, we motivated the need for our strong CPA-style analysis; modularity would not permit this system, since the creation of contours must be put off until constraint closure time. Modular compilation also imposes the requirement that subclasses conform to the policy of the superclass, since any subclasses may be used in place of the superclass. We intentionally do not impose this restriction, allowing the `InputStream` and `OutputStream` subclasses to contain different policies. This is particularly necessary in our desire to implement default policies for pre-existing IO channels, and thereby achieve backwards compatibility. For example, as described in Section 2.4, the default policy for an input channel (e.g. `FileInputStream`) is low secrecy. If we were to enforce a modular

restriction, any subclasses of `FileInputStream` (e.g. `HighFileIS`) would also have to be low secrecy, since wherever a `FileIS` is declared, the type system forces the `read()` method to return a low value. This would mean we could not create high secrecy input channels, disabling us from defining policies based on IO streams.

3.2.11 Example Typing

To illustrate our type inference algorithm, consider the following program, which reads from low and high input streams, and puts the input into the fields of different objects, and the low data is written back to a low output stream. This is a simplified program of our `HashSet` example in Section 2.6.

```
class HighFileIS extends FileIS {
 int read() { return read_{high}(fd); }
}
class LowFileIS extends FileIS {
 int read() { return read_∅(fd); }
}
class LowFileOS extends FileOS {
 void write(int v) { write_∅(v,fd); }
}
class C extends Object {
 int x;
 C() { super(); x = 0;}
}
void main() {
 FileIS hin = new HighFileIS("high_infile");
 FileIS lin = new LowFileIS("low_infile");
 FileOS lowout = new LowFileOS("low_outfile");

 lowC = new C();
 highC = new C();
 lowC.x = lin.read();
 highC.x = hin.read();
 lowout.write(lowC.x);
 }
```

The relevant constraints in the label table are as follows, where ... indicates additional type variables or constraints, whose inclusion would only clutter our example.

$(\texttt{HighFIS}, \texttt{read}) : \forall t, t_t \xrightarrow{s_p} t_r \backslash \{\langle \ldots \{\texttt{high}\}, , \emptyset, \texttt{int} \rangle <: t_r\} \ldots$

$(\texttt{LowFIS}, \texttt{read}) : \forall t', t'_t \xrightarrow{s'_p} t'_r \backslash \{\langle \ldots \emptyset, , \emptyset, \texttt{int} \rangle <: t'_r\} \ldots$

$(\texttt{LowFOS}, \texttt{write}) : \forall t_v, t''_t \xrightarrow{s''_p} t''_r \backslash SC(\emptyset, \cdots \cup s_v) \cup \ldots$

There are three important things to note here. Firstly, for any call to the `read` method of a `HighFIS` object, the return type will have $\{\texttt{high}\}$ flowing into the secrecy type. Secondly, for any call to the `read` method of a `LowFIS` object, the secrecy type is unlabeled by the input stream. Finally, for any call to the `write` method of a `LowFOS` object, a secrecy check ensures that the data being output is unlabeled.

When each `new C()` expression is typed, (New) gives them each distinct types; the type of `lowC` is t_l, and the type of `highC` is t_h.

Using (Invoke), the typing of `hin.read()` creates a fresh set of variables for the return value, t_{rh}, and generates a method constraint that closure rule $(Method)$, using the above defined label table, instantiates to $\langle \ldots \{\texttt{high}\}, \emptyset, \texttt{int} \rangle <: t_{rh}$, and so $\{\texttt{high}\} <: s_{rh}$ (recall each t is a three-tuple of type variables, $\langle s, f, \alpha \rangle$). By (F-Assign) and the relevant closure rules, we have $f_h.\texttt{x} <:$ **set** t_{rh}, which by (Set) entails $t_{rh} <: f_h.\texttt{x}$, so by transitivity, we have $\{\texttt{high}\} <: f_h.\texttt{x}.\mathsf{S}$. Note that a new type variable is used for each method invocation, so s_{rh} is unique to the call `hin.read()`, so the `high` label does *not* pollute the low call `lin.read()`; therefore, $\{\texttt{high}\}$ does *not* flow into $f_l.\texttt{x}.\mathsf{S}$.

`lowout.write(lowC.x)`; produces a method type, which when instantiated with $(Method)$ gives $SC(\emptyset, \emptyset \cup f_l.\texttt{x}.\mathsf{S})$, since the type of the program counter, the object `lowout`, and the field

fd of lowout are all $\emptyset$, and f_l.x.S was substituted for s_v. Now, we have $\emptyset <: f_l$.x.S from lowC.x = lin.read() and (Set), which leads to the constraint $SC(\emptyset, \emptyset)$, which is consistent.

Suppose we added the statement lowout.write(highC.x); to the program. So, the statement lowout.write(highC.x); yields a method type, which when instantiated by $(Method)$ gives $SC(\emptyset, \emptyset \cup f_h$.x.S). As noted above, we have $\{\texttt{high}\} <: f_h$.x.S, which leads to the constraint $SC(\emptyset, \{\texttt{high}\})$, which is inconsistent.

3.2.12 Integrity

We now discuss how integrity can be added to the formal system, which is uncomplicated, since integrity is a dual to secrecy. Types become four-tuples, $\langle \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle$, where $\mathcal{I}$ represents the integrity type, similar to the secrecy type. A separate integrity program counter is also required to track the integrity context under which an expression is typed. As shown in Chapter 2, read and write operations are expanded to include integrity policies, $\texttt{read}_{(\texttt{L},\texttt{L}')}(\texttt{fd})$ and $\texttt{write}_{(\texttt{L},\texttt{L}')}(\texttt{e}, \texttt{fd})$, where L is a secrecy policy and L′ is an integrity policy. For example, $\texttt{read}_{(\emptyset,\{\texttt{trusted}\})}(\texttt{fd})$ indicates that the inputs from this channel are trusted. Further, an endorsement operation Endorse(e, L) is added to the syntax, which adds labels L to the integrity types of e in the result. This is dual to how declassification removes secrecy types from an expression.

Throughout the typing, wherever secrecy types are unioned ($\cup$), integrity types are intersected ($\cap$). For example, the result of adding trusted data to untrusted data becomes untrusted, which is the intersection of the policies. This reflects the duality relationship between secrecy and integrity. Integrity typing of main must begin with the highest integrity level (*i.e.*, most trusted) program counter, lest all integrity types be forced to low. This again emphasizes the duality, since the program counter of secrecy types begins with the lowest security level.

Finally, integrity policies must be enforced at IO points using check constraints SC. Since integrity is a dual to secrecy, the arguments given to SC are reversed, so $SC(\mathtt{L}', \mathtt{L})$, $\mathtt{L}$ is the integrity policy of the IO channel, and $\mathtt{L}'$ is the set of labels on the data at that point. The check then enforces the relationship that $\mathtt{L} \subseteq \mathtt{L}'$ is required to satisfy the policy. For example, the constraint $SC(\{\mathtt{Trusted}, \mathtt{Classified}\}, \{\mathtt{Trusted}\})$ is consistent, as the data is required to carry at least the trusted label; whereas $SC(\{\mathtt{Trusted}\}, \{\mathtt{Trusted}, \mathtt{Classified}\})$ is inconsistent, since the data must be both trusted and classified.

Chapter 4

Soundness and Noninterference

We now present the formal soundness and noninterference properties for our system that justify its correctness. Soundness means that well-typed programs will not produce any run-time secrecy check failures in our operational semantics. Noninterference is proved to show that for well-typed programs, changes made to high inputs do not affect low outputs. Formal results are proved only for secrecy; the dual properties of integrity can be shown with nearly identical proofs to the secrecy counterparts. Hence, integrity soundness, states that for well-typed programs, no run-time integrity check failures occur; and integrity noninterference states that low integrity inputs do not affect high integrity outputs.

We first provide a brief overview of our proof technique, which is a new method for proving noninterference using a small-step operational semantics that is augmented with a syntactic representation of the type derivation. After we have sufficiently described some necessary assumptions and definitions, we give a more detailed description of the proof technique in section 4.3.

Proving that two runs of a program produce identical low output streams is a challenging task that requires reasoning about the actual execution steps of each run. Indeed, if one execution writes to a low stream, the other execution must do the same, with the same value. However, this process is complicated by the difference of high inputs, which may cause the runs to execute different steps. Therefore, we distinguish the execution steps that both runs must take, from the steps that may differ, which we call *low steps* and *high steps*, respectively. Noninterference is then proved by showing that all low steps executed by one run of the program must be executed by the other run, as long as the low inputs remain the same.

In order to show that two runs produce identical low output streams, we need a way of aligning the two runs as they execute. A normal run-time semantic model only executes a single run, with its set of input values. Since the static type system considers all possible runs of a given program, we can use the type derivation at run-time to provide us with an alignment from one run to another. Therefore, we define a small-step operational semantics that is augmented with a *syntactic* representation of the program's type derivation. So, the entire type derivation (proof tree) is maintained in the semantics, and we leverage the structure of the type derivation tree to show the low behavior of two different executions is the same. The primary reason for using the entire derivation tree, as opposed to just the final type of an expression, is to handle our highly polymorphic methods; in particular, for any low steps that are method calls, both runs must use the same contour for the low-equivalence to hold when the method body is executed. It would be difficult to align the selection of these contours if the type derivations were not present.

Once the operational semantics is defined in this fashion, we prove a subject reduction property, which states that after each reduction step, the syntactic type derivation is a *valid* type derivation for the resulting expression, and the resulting type is the same as before the reduction step.

Since the type system establishes security levels for each expression and statement, using the type derivation in the semantics provides us with the ability to distinguish low and high steps, based on the typing. Using a bisimulation relation between the program runs, we show that for typeable programs, assuming two executions where the low input streams are identical, they each take the same low steps, with possibly differing high steps between. Hence, when the execution finishes, the result is a low-equivalent trace of inputs and outputs.

The remainder of this chapter is structured as follows. Section 4.1 provides some necessary definitions, and Section 4.2 defines a small-step operational semantics that is augmented with syntactic type derivations. After giving these definitions, we provide a description of the proof technique in Section 4.3. The Subject Reduction Lemma and Soundness Theorem appear in Section 4.4, and the Noninterference result for our augmented semantics is in Section 4.5. Section 4.6 concludes with a proof of noninterference for an unaugmented semantics, under which MJ programs may actually execute.

4.1 Assumptions and Definitions

In order to more clearly state our results, we now state some necessary assumptions and definitions for this chapter.

Since we are only proving properties in the secrecy domain, integrity labels and types have been omitted in this chapter. Though our language differentiates expressions and statements, we often simply write *expression* when the distinction is irrelevant.

The noninterference property requires *no* leaks to occur. Therefore, as is standard for noninterference proofs, we assume expressions and statements do not contain any $\texttt{Declassify}(\texttt{e}, \texttt{L})$ subexpressions, which would violate the property that high inputs do not affect low outputs. Though declassification is an explicitly allowed leak of information, it is nevertheless a leak of information, and noninterference will not hold under these circumstances. Consider the following example code, where $\texttt{fd}_\texttt{h}$ is a file descriptor for a high stream and $\texttt{fd}_\texttt{l}$ is a file descriptor for a low stream.

$o.\texttt{h} = \texttt{read}_{\texttt{High}}(\texttt{fd}_\texttt{h});$
$\texttt{write}_{\texttt{Low}}(\texttt{Declassify}(o.\texttt{h}, \texttt{High}), \texttt{fd}_\texttt{l});$

Although the value read from the high input channel is declassified before output to a low channel, the values being written to the low channels will be different if the high inputs are different. Therefore, noninterference cannot hold in the presence of declassification. Section 4.7 presents a more detailed discussion of declassification formalisms.

We proceed by in turn describing some definitions for the semantics and additions to the typing.

4.1.1 Semantic Definitions

Figure 4.1 shows the necessary definitions for the operational semantics. The first several definitions provide a description of the heap, which stores objects. Object identifiers (oids) are memory locations with a concrete security label. This label is used in the proof of noninterference to establish a low-equivalence relationship between heaps in two different runs. Input and output

S	::=	$\{\overline{\mathtt{l}}\}$, where $\overline{\mathtt{l}}$ are unique label names	*Concrete Labels*
F	::=	$\{\overline{\mathtt{f} = v}\}$	*Field Mapping*
ho	::=	$\mathtt{C}, F$	*Heap Objects*
H is a finite partial function from memory locations to heap objects.			*Heap*
loc is a unique memory location in the heap			
o	::=	loc^S	*Object Identifier (oid)*
ι	::=	$(\mathtt{fd}, S) \longrightarrow \overline{\mathtt{c}}$	*Input Streams*
ω	::=	$(\mathtt{fd}, S) \longrightarrow \overline{\mathtt{c}}$	*Output Streams*
v	::=	$\mathtt{C0} \mid o \mid \mathit{CkFail} \mid \mathit{IOErr}$	*Values*
$\mathtt{e}$	::=	$\ldots \mid o \mid \mathit{CkFail} \mid \mathit{IOErr}$	*Expressions*
$\mathtt{s}$	::=	$\ldots \mid \overline{\mathtt{s}} \mid \mathtt{e.super(C, \overline{e})} \mid \mathtt{e.super(Object)}$	*Statements*
ε	::=	$\mathtt{e} \mid \mathtt{s}$	*Expression/Statement*
$\mathcal{H}$	::=	$o \longrightarrow t \backslash \mathcal{C}$	*Heap Environments*
$\mathbf{TD}$	::=	the complete (canonical) type derivation tree of $\Gamma, pc, \mathcal{H} \vdash \varepsilon : \tau \backslash \mathcal{C}$	*Type Derivations*
TDT	::=	$(\mathtt{C}, \mathtt{MK}) \longrightarrow \mathbf{TD}$ where $\mathtt{MK} = \mathtt{m} \mid \mathtt{K}$	*Type Derivation Tables*
$\varepsilon, \mathbf{TD}, \mathcal{C}, H, \iota, \omega \rightsquigarrow \varepsilon', \mathbf{TD}', \mathcal{C}, H', \iota', \omega'$ Assuming a fixed LT, TDT, and CT.			*Reductions of Configurations*

Figure 4.1: Operational Semantics Definitions

streams are defined as functions mapping file descriptors with a *fixed* security level to a stream of integers. Values are either constants, oids, CkFail, which indicates a security check failure, or IOErr, which indicates a mismatch in the program definition of a stream's security level with its run-time counterpart. For example, suppose we have a program that contains $\mathtt{read}_{\mathtt{Low}}(\mathtt{fd}_\mathtt{h})$, but the input stream function defines $\iota(\mathtt{fd}_\mathtt{h}, \mathtt{High})$ ($\mathtt{fd}_\mathtt{h}$ is a file descriptor for an input stream). This mismatch in the security level of the channel will produce an IOErr at run-time.

Expressions are extended from the definitions in Figure 3.1 to add the new values, and statements are extended for calls to `super`. The call to `super` as defined here includes the name of the class to which the call is being made, so that `super` calls can be properly made up the entire inheritance hierarchy, since a call to a constructor will trigger the constructor of all super-classes

up to the root. Unlike in MJ, our semantics does not use stack frames and scoping mechanisms. In MJ, they use a scope to determine what the next class up is. Instead, we hard-code the name of the next class into the `super` syntax during execution. We further allow a statement to also be a list of statements, which makes our proofs simpler.

We introduce the syntactic sugar ε to refer to either an expression or a statement. Heap environments are described in the next section. A type derivation, TD, is a syntactic representation of a complete, canonical, type derivation tree for an expression/statement. Canonical derivations are defined in Definition 4.1 below. Type derivation tables TDT contain the type derivation trees of all method bodies and constructors, so they can be accessed during method and constructor calls; they are also assumed to be canonical. Finally, reductions are on six-tuples, which we call *configurations*; they consist of an expression or statement, a type derivation, a constraint set $\mathcal{C}$, a heap, and input and output streams. The constraint set $\mathcal{C}$ is the largest constraint set from the typing of the whole program, and is invariant during execution. It is a convenient way of accessing all of the constraints, especially during context reductions, when the type derivation tree TD must be taken apart so that the execution may proceed under a reduction context.

We further assume that the program has a fixed, well-typed class table CT (typed via the normal MJ type rules), and a fixed, well-typed label table LT (typed via our label type inference rules), and a fixed type derivation table, TDT, that matches the corresponding types in LT, only containing the complete derivation tree for each method and constructor.

4.1.2 Additional Type Definitions

Typing during the operational semantics reductions requires some additional definitions. The existence of a heap during execution requires some standard type definitions. We introduce a

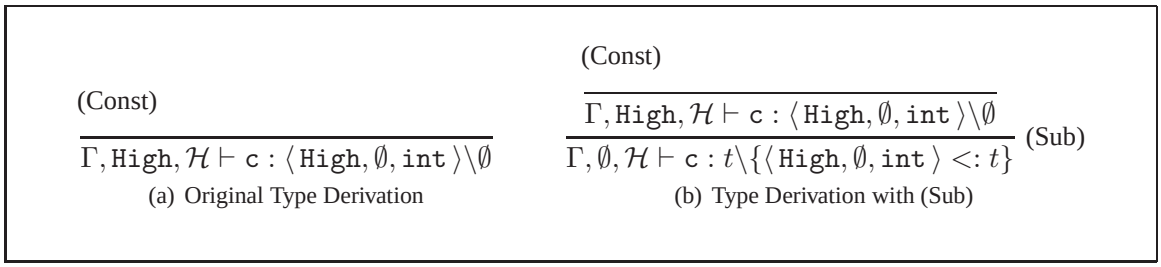
$$\frac{\mathcal{H}(o) = t_o \backslash \mathcal{C}_o}{\Gamma, pc, \mathcal{H} \vdash o : \langle\, pc \cup s_o, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o} \text{(Oid)}$$

$$\frac{\Gamma, pc, \mathcal{H} \vdash \mathtt{e} : \tau \backslash \mathcal{C} \qquad \{\tau <: t\} \subseteq \mathcal{C}' \qquad pc' \subseteq pc \qquad \mathcal{C} \subseteq \mathcal{C}' \qquad \mathcal{C}' \text{ is closed}}{\Gamma, pc', \mathcal{H} \vdash \mathtt{e} : t \backslash \mathcal{C}'} \text{(Sub)}$$

$$\frac{\begin{array}{l} dom(\mathcal{H}) = dom(H) \qquad \forall o \in dom(\mathcal{H}).(\mathcal{H}(o) = t_o \backslash \mathcal{C}_o, \quad H(o) = \mathtt{C}, \{\overline{\mathtt{f} = v}\}, \\ \quad \forall v \in \overline{v}.(\emptyset, \emptyset, \mathcal{H} \vdash v : \tau_v \backslash \mathcal{C}_v, \quad \{f_o.\mathtt{f} <: \textbf{set}\ \tau_v\} \cup \mathcal{C}_v \cup \{\mathtt{C} <: \alpha_o\} \subseteq \mathcal{C}_o)) \end{array}}{\mathcal{H} \vdash H} \text{(Heap)}$$

Figure 4.2: Type Rules for Typing During Reductions

heap environment, $\mathcal{H}$, that maps object identifiers to types with constraint sets; its definition is in Figure 4.1. Type rules are then re-defined under this heap environment, so all rules are of the form $\Gamma, pc, \mathcal{H} \vdash \varepsilon : \tau \backslash \mathcal{C}$; we do not reproduce the already defined typing rules with the heap environment, as their definition is obvious. Additional type rules are given in Figure 4.2. Using (Oid), object identifiers are typed by taking the type from the heap environment and adding the program counter to it. The rule (Sub) is a standard subsumption type rule that allows type variables t to replace a type τ, so long as the constraint set enforces that the type τ flows into t, written ($\tau <: t$). It also allows the secrecy program counter to decrease in the conclusion of a type derivation; hence, we may have a typing in the premise with a high program counter, and a low program counter in the conclusion. This makes sense, as we can add additional constraints to $\mathcal{C}'$ to increase the typing of the conclusion back up to high, as shown in the example use of (Sub) in Figure 4.3. Here, the original typing in Figure 4.3(a) shows the typing of a constant under a high program counter. Figure 4.3(a) shows the same typing with an added (Sub) rule. Notice how the program counter in the final conclusion is empty; nevertheless, the inclusion of $\{\langle \mathtt{High}, \emptyset, \mathtt{int} \,\rangle <: t\}$ in the constraint set provides that `c` will have the same security type as the original, after using the transitivity closure rules.



(a) Original Type Derivation

(b) Type Derivation with (Sub)

Figure 4.3: Example use of (Sub)

The final new type rule in Figure 4.2, (Heap), asserts that a heap H is well-typed in relation to a heap environment $\mathcal{H}$. The only other change to the type rules is in the use of type checking rules as opposed to type inference rules. The type rules presented in Chapter 3 are inference rules. In the type inference system, the generation of fresh variables is required to create the most general typing. When typing during reductions, we are not performing type inference; indeed, we are already assuming a program is well-typed. However, in order to maintain the same type derivation after a reduction step, it is necessary to select the correct type variables in the typing. Therefore, in our proofs we use type checking rules, which differ from the inference rules only in that checking rules have no requirement for fresh type variables. Since the difference between checking and inference rules is so minor, we do not give the full checking rules, as their definitions are obvious from the inference rules presented in Chapter 3.

Now that we have discussed the changes to the type system, we give the following type-based definitions. Definition 4.1 formalizes a canonical type derivation. Canonical derivations allow us to precisely reason about the structure of a typing, and further provide the alignment of derivations we require for proving noninterference, as we shall see. In section 4.4, we will prove that any expression (statement) that has a type derivation also has a canonical derivation.

Definition 4.1 (Canonical Proofs) *A type derivation (proof)* $\Gamma, pc, \mathcal{H} \vdash \varepsilon : \langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C}'$ *is canonical iff every syntax-directed type rule in the derivation is followed by exactly one instance of (Sub).*

As mentioned previously, the typing of a program provides insight into the security level of each expression. Definition 4.2 allows us to use the typing and the constraint set to establish a concrete set of security labels for any secrecy type, and therefore any expression.

Definition 4.2 (Concrete Labels) $Con(\mathcal{S}, \mathcal{C}) = S$*, where* S *is a set containing every concrete label,* $\mathtt{l}$*, such that either*

1. $\mathtt{l} \in \mathcal{S}$*; or*
2. *there exists an* $s \in \mathcal{S}$*, such that* $\mathtt{l} <: s \in \mathcal{C}$*; or*
3. *there exists an* $f.\mathtt{f}.\overline{\mathsf{F}.\mathtt{f}'}.\mathsf{S} \in \mathcal{S}$*, such that* $\mathtt{l} <: f.\mathtt{f}.\overline{\mathsf{F}.\mathtt{f}'}.\mathsf{S} \in \mathcal{C}$

For example, suppose we have the following code, with its type derivation.

$o.\mathtt{h} = \mathtt{read}_{\mathtt{High}}(\mathtt{fd}_{\mathtt{h}})\mathtt{;}$
$\mathtt{return}\ o.\mathtt{h}\mathtt{;}$

Now, according to the type rules (Return) and (Field), $\mathtt{return}\ o.\mathtt{h}$ has a secrecy type s, and based on the read operation and assignment (and closure rules), the constraint set will contain a constraint $\mathtt{High} <: s$. Therefore, based on this definition, we can correctly give the security level of $o.\mathtt{h}$ as $\mathtt{High}$.

4.2 Semantics

We now present a small-step operational semantics for our system. Figure 4.1 shows the necessary definitions for the semantics, which were discussed in Section 4.1.1. The main reduction rules are given in Figure 4.4, Figure 4.5, and Figure 4.6. Reductions under context rules are in Figures 4.7 and 4.8, failure reduction under context rules in Figures 4.9 and 4.10.

In contrast to MJ, we use a substitution-style semantics without stack frames and scoping. This greatly simplifies our semantics and proofs, with the only drawback that we do not have local variables in our formal system (though we show how local variables can be typed in Section 3.2.4). The rules are for the most part standard small-step rules, apart from the addition of the syntactic type derivations. As previously mentioned, these are added so that we may compare two executions for proving noninterference. It is important to recognize that the use of the type derivation in the semantics is *syntactic*; therefore, the semantics rules construct and deconstruct type derivations as a manner of syntax, just like any other reduction may construct a new expression from an old one. We will later prove that these syntactic derivations are, in fact, *valid* type derivations for each expression.

We now proceed with a discussion of the interesting aspects of the semantics. Recall from Section 4.1.1 that the third element of a reduction configuration is a constraint set from the typing of the whole program; we use $\mathcal{C}_\top$ to denote this set. The rules in Figure 4.4 are straight forward apart from the use of $td*$ functions. These functions are used to take the type derivation from before the reduction step and produce the type derivation for the expression after the reduction step; these functions are defined in Section 4.2.1. The rules in Figure 4.5 are defined similarly, with (New-R) as a notable exception. Here, a new object is created on the heap, with all fields set to `null`. The

location of this new heap object is defined by the security level given by the type derivation, via $getref$ and $newref$. Therefore, object locations on the heap are described by the security level of the object when it was created. We need locations to be tagged with security levels in order to provide a low-equivalence relation between heaps across executions in our noninterference proof; hence, any *low* objects will appear in both heaps at the same locations. The (New-R), (Invoke-R), and (Super-R) rules all substitute the values in for arguments in the method (or constructor) bodies. In addition, aligning with MJ, the rule (New-R) inserts a call to the constructor of the superclass before the constructor body of the class is executed. Furthermore, the functions for rebuilding the type derivation trees, $tdnew$, $tdinvoke$, and $tdsuper$, all utilize the $tdsub$ function for building the type derivation of the method or constructor body that appears after the reduction step.

The rules in Figure 4.6 show how input and output statements are executed. The function $conread$ extracts the concrete secrecy label of the file descriptor `fd` to check whether the read should be allowed. Assuming it matches the policy of the channel, using (Input-R), the value is read from the stream, and the type derivation is built via $tdinput$. If, however, the policy does not match, (InFail-R) causes a reduction to $CkFail$. We will later prove a soundness result that this will not happen for well-typed programs; however, since at this level, the type derivations are purely syntactic, such a reduction is allowed by the syntax (the derivation will of course not be valid). The final option is a reduction to $IOErr$, when the security level of the channel does not match that on the read statement. This is a property we *cannot* control statically, as it is a purely dynamic property, akin to such errors as `FileNotFound`. We will assume in our formal results that such reductions do not occur. The rules for reducing write statements are similar in nature, checking the security levels of `c` and `fd` against the security level of the output stream.

Reduction under context rules in Figures 4.7 and 4.8 are straight-forward, where the $ucon$ function takes the type derivation from the context, in order to use it during the context reduction step, and the $dcon$ function reassembles the outer derivation after the step occurs. Since the context reduction may add to the heap and therefore the heap environment, the function $tdheap$ makes these additions to the rest of the type derivation tree. Hence, the type derivation tree remains intact after the reduction, apart from the portion that was actually computed on, and the additions to the heap environment throughout the derivation tree. Failure reduction under context rules in Figures 4.9 and 4.10 simply cause any failure in a sub-computation to propagate outwards, so the entire execution will produce the same failure. Though it increases the number of steps in our reductions, defining the context reduction steps in this manner simplifies the inductive arguments in our proofs over Felleisen-style evaluation contexts [FF86].

(Field-R) $o.\mathtt{f}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow v, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$

where $H(o) = (\mathtt{C}, F),$ and $F(\mathtt{f}) = v$ and $\mathbf{TD}' = \mathit{tdfield}(\mathbf{TD}, v, \mathcal{C}_\top)$

(Op-R) $\mathtt{c} \oplus \mathtt{c}', \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow v, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$

where $v = \mathtt{c} \oplus \mathtt{c}'$ and $\mathbf{TD}' = \mathit{tdop}(\mathbf{TD}, v, \mathcal{C}_\top)$

(Cast-R) $(\mathtt{D})\ o, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow o, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$

where $H(o) = \mathtt{C}, F$ and $\mathtt{C} <: \mathtt{D}$ and $\mathbf{TD}' = \mathit{tdcast}(\mathbf{TD}, o, \mathcal{C}_\top)$

(IfTrue-R) $\mathtt{if}\ (\mathtt{True})\ \{\overline{\mathtt{s_1}}\}\ \mathtt{else}\ \{\overline{\mathtt{s_2}}\}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \{\overline{\mathtt{s_1}}\}, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$

where $\mathbf{TD}' = \mathit{tdiftrue}(\mathbf{TD}, \{\overline{\mathtt{s_1}}\}, \mathcal{C}_\top)$

(IfFalse-R) $\mathtt{if}\ (\mathtt{False})\ \{\overline{\mathtt{s_1}}\}\ \mathtt{else}\ \{\overline{\mathtt{s_2}}\}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \{\overline{\mathtt{s_2}}\}, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$

where $\mathbf{TD}' = \mathit{tdiffalse}(\mathbf{TD}, \{\overline{\mathtt{s_2}}\}, \mathcal{C}_\top)$

(Seq-R) $;\overline{\mathtt{s}}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \overline{\mathtt{s}}, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$

where $\mathbf{TD}' = \mathit{tdseq}(\mathbf{TD}, \overline{\mathtt{s}}, \mathcal{C}_\top)$

(Return-R) $\mathtt{return}\ v;, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow v, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$

where $\mathbf{TD}' = \mathit{tdreturn}(\mathbf{TD}, v, \mathcal{C}_\top)$

(Block-R) $\{\overline{\mathtt{s}}\}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \overline{\mathtt{s}}, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$

where $\mathbf{TD}' = \mathit{tdblock}(\mathbf{TD}, \overline{\mathtt{s}}, \mathcal{C}_\top)$

(Assign-R) $o.\mathtt{f} = v;, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow ;, \mathbf{TD}', \mathcal{C}_\top, H[o \mapsto \mathtt{C}, F'], \iota, \omega$

where $H(o) = \mathtt{C}, F$ and $F' = F[\mathtt{f} = v]$ and $\mathbf{TD}' = \mathit{tdassign}(\mathbf{TD}, \mathcal{C}_\top)$

Figure 4.4: Operational Semantics Reduction Rules

(New-R) $\texttt{new C}(\overline{v}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \varepsilon'; \texttt{return } o;, \mathbf{TD}', \mathcal{C}_\top, H', \iota, \omega$
where $cnbody(\texttt{C}) = (\overline{\texttt{x}}, \texttt{super}(\overline{\texttt{e}}); \overline{\texttt{s}})$ and $\texttt{class C extends D } \{\dots\}$
and $\varepsilon' = [\overline{\texttt{x}} \mapsto \overline{v}, \texttt{this} \mapsto o]\texttt{this.super}(\texttt{D}, \overline{\texttt{e}}); \overline{\texttt{s}}$
and $H' = H[o \mapsto \texttt{C}, F]$ and $F = \{\overline{\texttt{f} = \texttt{null}}\}$
and $o = getref(H, \mathbf{TD}, \mathcal{C}_\top)$ and $\mathbf{TD}' = tdnew(\mathbf{TD}, \texttt{C}, \mathcal{C}_\top)$

(Invoke-R) $o\,.\texttt{m}(\overline{v}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \overline{\texttt{s}'}, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$
where $H(o) = \texttt{C}, F$ and $mbody(\texttt{C}, \texttt{m}) = (\overline{\texttt{x}}, \overline{\texttt{s}})$
and $\overline{\texttt{s}'} = [\overline{\texttt{x}} \mapsto \overline{v}, \texttt{this} \mapsto o]\overline{\texttt{s}}$ and $\mathbf{TD}' = tdinvoke(\mathbf{TD}, \texttt{C}, \texttt{m}, \mathcal{C}_\top)$

(Super-R) $o.\texttt{super}(\texttt{C}, \overline{v}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \varepsilon', \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$
where $cnbody(\texttt{C}) = (\overline{\texttt{x}}, \texttt{super}(\overline{\texttt{e}}); \overline{\texttt{s}})$ and $\texttt{class C extends D } \{\dots\}$
and $\varepsilon' = [\overline{\texttt{x}} \mapsto \overline{v}, \texttt{this} \mapsto o]\texttt{this.super}(\texttt{D}, \overline{\texttt{e}}); \overline{\texttt{s}}$
and $\mathbf{TD}' = tdsuper(\mathbf{TD}, \texttt{C}, \mathcal{C}_\top)$

(Super-R′) $o.\texttt{super}(\texttt{Object}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow ;, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$
and $\mathbf{TD}' = tdsuperobj(\mathbf{TD}, \mathcal{C}_\top)$

$$mbody(\texttt{C}, \texttt{m}) = \begin{cases} (\overline{\texttt{x}}, \overline{\texttt{s}}) & \text{if } \texttt{M} \in \overline{\texttt{M}} \text{ and } \texttt{M} = \texttt{RT m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\ \{\overline{\texttt{s}}\} \\ mbody(\texttt{D}, \texttt{m}) & \text{otherwise} \end{cases}$$
where $\texttt{C}_0 = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\}$

$cnbody(\texttt{C}) = (\overline{\texttt{x}}, \overline{\texttt{s}})$ if $\texttt{K} = \texttt{C}(\overline{\texttt{C}}\ \overline{\texttt{x}})\ \{\texttt{super}(\overline{\texttt{e}}); \overline{\texttt{s}}\}$
where $\texttt{C}_0 = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\}$

getref$(H, \mathbf{TD}, \mathcal{C}_\top)$ = *newref*$(H, Con(s_o, \mathcal{C}_\top))$, where

$$\mathbf{TD} = \dfrac{\dfrac{\overline{\mathbf{TD}_v} \quad \overline{\mathbf{TD}_n}}{\Gamma, pc_2, \mathcal{H} \vdash \texttt{new C}(\overline{v}) : t_o \backslash \dots}\ \text{(New)}}{\Gamma, pc_1, \mathcal{H} \vdash \texttt{new C}(\overline{v}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

newref$(H, S) = o = loc_i^S$ where $i - 1$ is the largest integer, such that $loc_{i-1}^S \in H$

Figure 4.5: Operational Semantics Reduction Rules, continued

(Input-R) $\texttt{read}_{\texttt{L}}(\texttt{fd}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{c}, \mathbf{TD}', \mathcal{C}_\top, H, \iota', \omega$
where $\texttt{L} = S_i$ and $\iota(\texttt{fd}, S_i) = \texttt{c}.\iota'(\texttt{fd}, S_i)$
and $S_f = \mathit{conread}(\mathbf{TD}, \mathcal{C}_\top)$ and $\mathbf{TD}' = \mathit{tdinput}(\mathbf{TD}, \texttt{c}, \mathcal{C}_\top)$
and $S_f \subseteq \texttt{L}$

(Output-R) $\texttt{write}_{\texttt{L}}(\texttt{c}, \texttt{fd}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow ;, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega'$
where $\texttt{L} = S_i$ and $\omega'(\texttt{fd}, S_i) = \texttt{c}.\omega(\texttt{fd}, S_i)$
and $S_f = \mathit{conwrite}(\mathbf{TD}, \mathcal{C}_\top)$ and $\mathbf{TD}' = \mathit{tdoutput}(\mathbf{TD}, \mathcal{C}_\top)$
and $S_f \subseteq \texttt{L}$

(InFail-R) $\texttt{read}_{\texttt{L}}(\texttt{fd}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{CkFail}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$
where $\texttt{L} = S_i$ and $\iota(\texttt{fd}, S_i)$
and $S_f = \mathit{conread}(\mathbf{TD}, \mathcal{C}_\top)$ and $S_f \not\subseteq \texttt{L}$

(OutFail-R) $\texttt{write}_{\texttt{L}}(\texttt{c}, \texttt{fd}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{CkFail}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$
where $\texttt{L} = S_i$ and $\omega(\texttt{fd}, S_i)$
and $S_f = \mathit{conwrite}(\mathbf{TD}, \mathcal{C}_\top)$ and $S_f \not\subseteq \texttt{L}$

(InErr-R) $\texttt{read}_{\texttt{L}}(\texttt{fd}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{IOErr}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$
where $\iota(\texttt{fd}, S_i)$ and $\texttt{L} \neq S_i$

(OutErr-R) $\texttt{write}_{\texttt{L}}(\texttt{c}, \texttt{fd}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{IOErr}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$
where $\omega(\texttt{fd}, S_i)$ and $\texttt{L} \neq S_i$

Figure 4.6: Operational Semantics IO Reduction Rules

(Field-RC)	$\texttt{e.f}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}'\texttt{.f}, \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}', \mathbf{TD}'_1, \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconfield(\mathbf{TD}, \mathbf{TD}'_1, \mathcal{C}_\top)$
(Op-RC)	$\texttt{e}_1 \oplus \texttt{e}_2, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}'_1 \oplus \texttt{e}_2, \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}'_1, \mathbf{TD}'_1, \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconop(\mathbf{TD}, \mathbf{TD}'_1, \mathcal{C}_\top)$
(Op-RC')	$v \oplus \texttt{e}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow v \oplus \texttt{e}', \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}', \mathbf{TD}'_1, \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconop(\mathbf{TD}, \mathbf{TD}'_1, \mathcal{C}_\top)$
(Cast-RC)	$(\texttt{C})\ \texttt{e}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow (\texttt{C})\ \texttt{e}', \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}', \mathbf{TD}'_1, \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconcast(\mathbf{TD}, \mathbf{TD}'_1, \mathcal{C}_\top)$
(If-RC)	$\texttt{if}\ (\texttt{e})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow$ $\texttt{if}\ (\texttt{e}')\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\}, \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}', \mathbf{TD}'_1, \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconif(\mathbf{TD}, \mathbf{TD}'_1, \mathcal{C}_\top)$
(Seq-RC)	$\texttt{s}; \overline{\texttt{s}}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{s}'; \overline{\texttt{s}}, \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{s}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{s}', \mathbf{TD}'_1, \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconseq(\mathbf{TD}, \mathbf{TD}'_1, \mathcal{C}_\top)$
(Return-RC)	$\texttt{return e};, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{return e}', ; \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}', \mathbf{TD}'_1, \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconreturn(\mathbf{TD}, \mathbf{TD}'_1, \mathcal{C}_\top)$
(Assign-RC)	$\texttt{e}_1\texttt{.f} = \texttt{e}_2;, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}'_1\texttt{.f} = \texttt{e}_2;, \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}'_1, \mathbf{TD}'_1, \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconassign(\mathbf{TD}, \mathbf{TD}'_1, \mathcal{C}_\top)$
(Assign-RC')	$o\texttt{.f} = \texttt{e};, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow o\texttt{.f} = \texttt{e}';, \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}', \mathbf{TD}'_1, \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconassign'(\mathbf{TD}, \mathbf{TD}'_1, \mathcal{C}_\top)$

Figure 4.7: Operational Semantics Reductions Under Context

(New-RC)	$\texttt{new C}(\overline{v}, \texttt{e}, \overline{\texttt{e}}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{new C}(\overline{v}, \texttt{e}', \overline{\texttt{e}}), \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}', \mathbf{TD}_1', \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconnew(\mathbf{TD}, \mathbf{TD}_1', \mathcal{C}_\top)$
(Invk-RC)	$\texttt{e.m}(\overline{\texttt{e}}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}'\texttt{.m}(\overline{\texttt{e}}), \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}', \mathbf{TD}_1', \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconinvk(\mathbf{TD}, \mathbf{TD}_1', \mathcal{C}_\top)$
(IArg-RC)	$o\texttt{.m}(\overline{v}, \texttt{e}, \overline{\texttt{e}}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow o\texttt{.m}(\overline{v}, \texttt{e}', \overline{\texttt{e}}), \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}', \mathbf{TD}_1', \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconiarg(\mathbf{TD}, \mathbf{TD}_1', \mathcal{C}_\top)$
(Super-RC)	$o\texttt{.super}(\overline{v}, \texttt{e}, \overline{\texttt{e}}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow$ $o\texttt{.super}(\overline{v}, \texttt{e}', \overline{\texttt{e}}), \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}', \mathbf{TD}_1', \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconsuper(\mathbf{TD}, \mathbf{TD}_1', \mathcal{C}_\top)$
(Input-RC)	$\texttt{read}_\texttt{L}(\texttt{e}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{read}_\texttt{L}(\texttt{e}'), \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}', \mathbf{TD}_1', \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconinput(\mathbf{TD}, \mathbf{TD}_1', \mathcal{C}_\top)$
(Output-RC)	$\texttt{write}_\texttt{L}(\texttt{e}_1, \texttt{e}_2), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{write}_\texttt{L}(\texttt{e}_1', \texttt{e}_2), \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}_1', \mathbf{TD}_1', \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconoutput(\mathbf{TD}, \mathbf{TD}_1', \mathcal{C}_\top)$
(Output-RC')	$\texttt{write}_\texttt{L}(v, \texttt{e}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{write}_\texttt{L}(v, \texttt{e}'), \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{e}', \mathbf{TD}_1', \mathcal{C}_\top, H', \iota', \omega'$ and $\mathbf{TD}_1 = ucon(\mathbf{TD})$ and $\mathbf{TD}' = dconoutput'(\mathbf{TD}, \mathbf{TD}_1', \mathcal{C}_\top)$

Figure 4.8: Operational Semantics Reductions Under Context, continued

(Field-Err)	$\texttt{e.f}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$
(Op-Err)	$\texttt{e}_1 \oplus \texttt{e}_2, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$
(Op-Err')	$v \oplus \texttt{e}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$
(Cast-Err)	$(\texttt{C})\ \texttt{e}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$
(If-Err)	$\texttt{if}\ (\texttt{e})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow$ $\mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$
(Return-Err)	$\texttt{return e;}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$
(Assign-Err)	$\texttt{e}_1\texttt{.f} = \texttt{e}_2\texttt{;}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$
(Assign-Err')	$o\texttt{.f} = \texttt{e;}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$

Figure 4.9: Operational Semantics Failure Reductions Under Context

(New-Err)	$\texttt{new C}(\overline{v}, \texttt{e}, \overline{\texttt{e}}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$
(Invk-Err)	$\texttt{e.m}(\overline{\texttt{e}}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$
(IArg-Err)	$o.\texttt{m}(\overline{v}, \texttt{e}, \overline{\texttt{e}}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$
(Super-Err)	$o.\texttt{super}(\overline{v}, \texttt{e}, \overline{\texttt{e}}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$
(Input-Err)	$\texttt{read}_\texttt{L}(\texttt{e}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$
(Output-Err)	$\texttt{write}_\texttt{L}(\texttt{e}_1, \texttt{e}_2), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$
(Output-Err')	$\texttt{write}_\texttt{L}(v, \texttt{e}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$ where $\texttt{e}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$ and $\mathit{Err} = \mathit{CkFail} \mid \mathit{IOErr}$

Figure 4.10: Operational Semantics Failure Reductions Under Context, Continued

4.2.1 Semantic Functions for TD's

In this section we define the semantic definitions for *syntactically* building type derivation trees. These are the functions used in the reduction rules given in the semantics figures. Note we use ... in type derivations when the remainder of the type derivation tree is inconsequential.

The $td*$ functions given in Definition 4.3 are used in the semantics to build the type derivation after a reduction step. Hence, each of the $td*$ functions take as argument the previous type derivation, $\mathbf{TD}$, the constraint set $\mathcal{C}_\top$ from the original typing, and the arguments necessary for building the new derivation. These functions then return the new type derivation, $\mathbf{TD}'$. Let us now examine $tdop$ as an example. Here, the original type derivation $\mathbf{TD}$ ends in uses of (Op) and (Sub). The result of the reducing this expression produces a new value v, having applied the operation to the two values. Hence, the new type derivation $\mathbf{TD}'$ is a typing of this new value, built so that the type variables used in (Sub) correspond to the same type variables in the previous typing, and the constraint set $\mathcal{C}$ is maintained. Consider the following concrete typing of $\mathbf{TD}$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{c_a} : \langle\, pc, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\,\text{(Const)}}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{c_a} : t_a \backslash \mathcal{C}_a}\,\text{(Sub)} \quad \dfrac{\dfrac{}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{c_b} : \langle\, pc, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\,\text{(Const)}}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{c_b} : t_a \backslash \mathcal{C}_b}\,\text{(Sub)}}{\dfrac{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{c_a} \oplus \mathtt{c_b} : \langle\, s_a \cup s_b, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_a \cup \mathcal{C}_b}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{c_a} \oplus \mathtt{c_b} : t \backslash \mathcal{C}}\,\text{(Sub)}}\,\text{(Op)}$$

When (Op-R) is used to reduce the expression $\mathtt{c_a} \oplus \mathtt{c_b}$, the function $tdop(\mathbf{TD}, v, \mathcal{C}_\top)$ is used to build the new type derivation $\mathbf{TD}'$ as follows.

$$\dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H} \vdash v : \langle\, pc_1, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\,\text{(Const)}}{\Gamma, pc_1, \mathcal{H} \vdash v : t \backslash \mathcal{C}}\,\text{(Sub)}$$

It is important to note here that the final types and constraint remains the same as before the reduction. This allows us to prove a subject reduction lemma over the augmented semantics.

Definition 4.3 ($td*$)

1. $tdfield(\mathbf{TD}, v, \mathcal{C}_\top) = \mathbf{TD}'$

   $$\mathbf{TD} = \dfrac{\dfrac{\cdots}{\Gamma, pc_1, \mathcal{H} \vdash o.\mathtt{f} : t \backslash \mathcal{C}}\ \text{(Sub)}}{}$$

   (a) v is a constant

   $$\mathbf{TD}' = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H} \vdash v : \langle\, pc_1, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_1, \mathcal{H} \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   and $\mathcal{C} \subseteq \mathcal{C}_\top$.

   (b) v is an object identifier

   $$\mathbf{TD}' = \dfrac{\dfrac{\mathcal{H}(v) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_1, \mathcal{H} \vdash v : \langle\, pc_1 \cup s_o, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o}\ \text{(Oid)}}{\Gamma, pc_1, \mathcal{H} \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   and $\mathcal{C} \subseteq \mathcal{C}_\top$.

2. $tdop(\mathbf{TD}, v, \mathcal{C}_\top) = \mathbf{TD}'$

   $$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_a \qquad \mathbf{TD}_b}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{c_a} \oplus \mathtt{c_b} : \langle\, s_a \cup s_b, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_a \cup \mathcal{C}_b}\ \text{(Op)}}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{c_a} \oplus \mathtt{c_b} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   $$\mathbf{TD}_a = \dfrac{\dfrac{\cdots}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{c_a} : t_a \backslash \mathcal{C}_a}\ \text{(Sub)}}{}$$

   $$\mathbf{TD}_b = \dfrac{\dfrac{\cdots}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{c_b} : t_b \backslash \mathcal{C}_b}\ \text{(Sub)}}{}$$

   $$\mathbf{TD}' = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H} \vdash v : \langle\, pc_1, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_1, \mathcal{H} \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   and $\mathcal{C} \subseteq \mathcal{C}_\top$.

3. $tdcast(\mathbf{TD}, o, \mathcal{C}_\top) = \mathbf{TD}'$

   $$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_o}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{D}\ o : t'_o \backslash \mathcal{C}'_o}\ \text{(Cast)}}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{D}\ o : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_o = \dfrac{\dfrac{\mathcal{H}(o) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_3, \mathcal{H} \vdash o : \langle\, pc_3 \cup s_o, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o}\ \text{(Oid)}}{\Gamma, pc_2, \mathcal{H} \vdash o : t'_o \backslash \mathcal{C}'_o}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathcal{H}(o) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_1, \mathcal{H} \vdash o : \langle\, pc_1 \cup s_o, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o}\ \text{(Oid)}}{\Gamma, pc_1, \mathcal{H} \vdash o : t \backslash \mathcal{C}}\ \text{(Sub)}$$

and $\mathcal{C} \subseteq \mathcal{C}_\top$.

4. $tdiftrue(\mathbf{TD}, \{\overline{\mathtt{s_1}}\}, \mathcal{C}_\top) = \mathbf{TD}'$

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_t \qquad \mathbf{TD}_1 \qquad \mathbf{TD}_2}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{if}\ (\mathtt{True})\ \{\overline{\mathtt{s_1}}\}\ \mathtt{else}\ \{\overline{\mathtt{s_2}}\} : \tau \backslash \ldots}\ \text{(If)}}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{if}\ (\mathtt{True})\ \{\overline{\mathtt{s_1}}\}\ \mathtt{else}\ \{\overline{\mathtt{s_2}}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_1 = \dfrac{\dfrac{\ldots}{\Gamma, pc_3, \mathcal{H} \vdash \{\overline{\mathtt{s_1}}\} : \tau_1 \backslash \mathcal{C}'_1}\ \text{(Block)}}{\Gamma, pc_2 \cup s_t, \mathcal{H} \vdash \{\overline{\mathtt{s_1}}\} : t_1 \backslash \mathcal{C}_1}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{\ldots}{\Gamma, pc_3, \mathcal{H} \vdash \{\overline{\mathtt{s_1}}\} : \tau_1 \backslash \mathcal{C}'_1}\ \text{(Block)}}{\Gamma, pc_1, \mathcal{H} \vdash \{\overline{\mathtt{s_1}}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

and $\mathcal{C} \subseteq \mathcal{C}_\top$.

5. $tdseq(\mathbf{TD}, \overline{\mathtt{s}}, \mathcal{C}_\top) = \mathbf{TD}'$

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_n \qquad \mathbf{TD}_s}{\Gamma, pc_2, \mathcal{H} \vdash ; \overline{\mathtt{s}} : t_s \backslash \mathcal{C}_n \cup \mathcal{C}_s}\ \text{(Seq)}}{\Gamma, pc_1, \mathcal{H} \vdash ; \overline{\mathtt{s}} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_s = \dfrac{\dfrac{\ldots}{\Gamma, pc_3, \mathcal{H} \vdash \overline{\mathtt{s}} : \tau_s \backslash \mathcal{C}'_s}\ \text{(Seq)}}{\Gamma, pc_2, \mathcal{H} \vdash \overline{\mathtt{s}} : t_s \backslash \mathcal{C}_s}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{\ldots}{\Gamma, pc_3, \mathcal{H} \vdash \overline{\mathtt{s}} : \tau_s \backslash \mathcal{C}'_s}\ \text{(Seq)}}{\Gamma, pc_1, \mathcal{H} \vdash \overline{\mathtt{s}} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

and $\mathcal{C} \subseteq \mathcal{C}_\top$.

6. $tdreturn(\mathbf{TD}, v, \mathcal{C}_\top) = \mathbf{TD}'$

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_v}{\Gamma, pc_2, \mathcal{H} \vdash \texttt{return}\ v : t_v \backslash \mathcal{C}_v}\ \text{(Return)}}{\Gamma, pc_1, \mathcal{H} \vdash \texttt{return}\ v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

(a) v is a constant

$$\mathbf{TD}' = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H} \vdash v : \langle\, pc_1, \emptyset, \texttt{int}\,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_1, \mathcal{H} \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

and $\mathcal{C} \subseteq \mathcal{C}_\top$.

(b) v is an object identifier

$$\mathbf{TD}' = \dfrac{\dfrac{\mathcal{H}(v) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_1, \mathcal{H} \vdash v : \langle\, pc_1 \cup s_o, f_o, \alpha_o\,\rangle \backslash \mathcal{C}_o}\ \text{(Oid)}}{\Gamma, pc_1, \mathcal{H} \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

and $\mathcal{C} \subseteq \mathcal{C}_\top$.

7. $tdblock(\mathbf{TD}, \overline{\mathtt{s}}, \mathcal{C}_\top) = \mathbf{TD}'$

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_s}{\Gamma, pc_2, \mathcal{H} \vdash \{\overline{\mathtt{s}}\} : t_s \backslash \mathcal{C}_s}\ \text{(Block)}}{\Gamma, pc_1, \mathcal{H} \vdash \{\overline{\mathtt{s}}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_s = \dfrac{\dfrac{\cdots}{\Gamma, pc_3, \mathcal{H} \vdash \overline{\mathtt{s}} : \tau_s \backslash \mathcal{C}'_s}\ \text{(Seq)}}{\Gamma, pc_2, \mathcal{H} \vdash \overline{\mathtt{s}} : t_s \backslash \mathcal{C}_s}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{\cdots}{\Gamma, pc_3, \mathcal{H} \vdash \overline{\mathtt{s}} : \tau_s \backslash \mathcal{C}'_s}\ \text{(Seq)}}{\Gamma, pc_1, \mathcal{H} \vdash \overline{\mathtt{s}} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

and $\mathcal{C} \subseteq \mathcal{C}_\top$.

8. $tdassign(\mathbf{TD}, \mathcal{C}_\top) = \mathbf{TD}'$

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_o \qquad \mathbf{TD}_v}{\begin{array}{c}\Gamma, pc_2, \mathcal{H} \vdash o.\mathtt{f} = v : \langle\, s'_o \cup s_v, \emptyset, \texttt{void}\,\rangle \backslash \mathcal{C}'_o \cup \mathcal{C}_v \cup \\ \{f_o.\mathtt{f} <: \textbf{set}\ \langle\, s'_o \cup s_v, f_v, \alpha_v\,\rangle\}\end{array}}\ \text{(F-Assign)}}{\Gamma, pc_1, \mathcal{H} \vdash o.\mathtt{f} = v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_o = \dfrac{\dfrac{\mathcal{H}(o) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_3, \mathcal{H} \vdash o : \langle\, pc_3 \cup s_o, f_o, \alpha_o\,\rangle \backslash \mathcal{C}_o}\ \text{(Oid)}}{\Gamma, pc_2, \mathcal{H} \vdash o : t'_o \backslash \mathcal{C}'_o}\ \text{(Sub)}$$

$\mathbf{TD}_v =$

$$\frac{\dots}{\Gamma, pc_2, \mathcal{H} \vdash v : t_v \backslash \mathcal{C}_v}\ \text{(Sub)}$$

$\mathcal{H}' = \mathcal{H}[o \mapsto t_o \backslash \mathcal{C}_o \cup \mathcal{C}_v \cup \{f_o.\mathtt{f} <: \textbf{set}\ \langle\, s'_o \cup s_v, f_v, \alpha_v \,\rangle\}]$

$\mathbf{TD}' =$

$$\frac{\dfrac{}{\Gamma, pc_1, \mathcal{H}' \vdash ; : \langle\, pc_1, \emptyset, \mathtt{void} \,\rangle \backslash \emptyset}\ \text{(No-op)}}{\Gamma, pc_1, \mathcal{H}' \vdash ; : t \backslash \mathcal{C}}\ \text{(Sub)}$$

and $\mathcal{C} \subseteq \mathcal{C}_\top$.

9. $tdnew(\mathbf{TD}, \mathtt{C}, \mathcal{C}_\top) = \mathbf{TD}'$

Where

$\mathbf{TD} =$

$$\frac{\dfrac{\overline{\mathbf{TD}_v} \qquad \overline{\mathbf{TD}_n}}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{new\ C}(\overline{v}) : t_o \backslash \dots}\ \text{(New)}}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{new\ C}(\overline{v}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$\overline{\mathbf{TD}_v} =$

$$\frac{\dots}{\Gamma, pc_2, \mathcal{H} \vdash \overline{v} : \overline{t_v} \backslash \overline{\mathcal{C}_v}}\ \text{(Sub)}$$

$\overline{\mathbf{TD}_n} =$

$$\frac{\dots}{\Gamma, pc_2, \mathcal{H} \vdash \overline{\mathtt{null}} : \overline{t_n} \backslash \overline{\mathcal{C}_n}}\ \text{(Sub)}$$

$cnbody(\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{super}(\overline{\mathtt{e}}); \overline{\mathtt{s}})$ and

$\mathbf{TD}_m = TDT(\mathtt{C}, \mathtt{K})$

and $LT(\mathtt{C}, \mathtt{K}) = \forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}_r$ and $\overline{t'_l} = \theta(\overline{t_l}, \mathtt{C}, \mathtt{K}, \overline{\alpha_v}, \alpha_o, \alpha_m)$

$\mathcal{H}' = \mathcal{H}[o \mapsto t_o \backslash \mathcal{C}_o]$ (same t_o that appears in $\mathbf{TD}$ above).

Where $\mathcal{C}_o = \{\overline{f_o.\mathtt{f} <: \textbf{set}\ t_n}\} \cup \overline{\mathcal{C}_n} \cup \{\mathtt{C} <: \alpha_o\}$.

$\mathbf{TD}_o =$

$$\frac{\dfrac{\mathcal{H}'(o) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_2 \cup s_o, \mathcal{H}' \vdash o : \langle\, pc_2 \cup s_o, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o}\ \text{(Oid)}}{\Gamma, pc_2 \cup s_o, \mathcal{H}' \vdash o : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Let $\mathbf{TD}_s = tdsub(\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}};, \overline{v}, o, \mathcal{H}', [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto$

$t_o, t_r \mapsto t_m], \mathbf{TD}_m, tdheap(\overline{\mathbf{TD}_v}, \mathcal{H}'), \mathbf{TD}_o)$

(Note: to simplify the definition of $tdsub$, we are treating `this`, which maps to the object identifier as another argument.)

So,

$$\mathbf{TD}_{ret} = \dfrac{\dfrac{\mathbf{TD}_o}{\Gamma, pc_2 \cup s_o, \mathcal{H}' \vdash \texttt{return}\ o; : t \backslash \mathcal{C}}\ \text{(Return)}}{\Gamma, pc_2 \cup s_o, \mathcal{H}' \vdash \texttt{return}\ o; : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$\mathbf{TD}' =$

$$\dfrac{\dfrac{\mathbf{TD}_s \qquad \mathbf{TD}_{ret}}{\Gamma, pc_2 \cup s_o, \mathcal{H}' \vdash [\overline{\texttt{x}} \mapsto \overline{v}, \texttt{this} \mapsto o]\texttt{this.super}(\texttt{D}, \overline{\texttt{e}}); \overline{\texttt{s}}; \texttt{return}\ o; : t \backslash \mathcal{C}}\ \text{(Seq)}}{\Gamma, pc_1, \mathcal{H}' \vdash [\overline{\texttt{x}} \mapsto \overline{v}, \texttt{this} \mapsto o]\texttt{this.super}(\texttt{D}, \overline{\texttt{e}}); \overline{\texttt{s}}; \texttt{return}\ o; : t \backslash \mathcal{C}}\ \text{(Sub)}$$

and $\mathcal{C} \subseteq \mathcal{C}_\top$.

10. $tdinvoke(\mathbf{TD}, \texttt{C}, \texttt{m}, \mathcal{C}_\top) = \mathbf{TD}'$

Where

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_o \qquad \overline{\mathbf{TD}_v}}{\Gamma, pc_2, \mathcal{H} \vdash o.\texttt{m}(\overline{v}) : t_m \backslash \ldots}\ \text{(Invoke)}}{\Gamma, pc_1, \mathcal{H} \vdash o.\texttt{m}(\overline{v}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_o = \dfrac{\ldots}{\Gamma, pc_2, \mathcal{H} \vdash o : t_o \backslash \mathcal{C}_o}\ \text{(Sub)}$$

$$\overline{\mathbf{TD}_v} = \dfrac{\ldots}{\Gamma, pc_2, \mathcal{H} \vdash \overline{v} : \overline{t_v} \backslash \overline{\mathcal{C}_v}}\ \text{(Sub)}$$

$mbody(\texttt{C}, \texttt{m}) = (\overline{\texttt{x}}, \overline{\texttt{s}})$ and

$\mathbf{TD}_m = TDT(\texttt{C}, \texttt{m})$

and $LT(\texttt{C}, \texttt{m}) = \forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}_r$ and $\overline{t'_l} = \theta(\overline{t_l}, \texttt{C}, \texttt{m}, \overline{\alpha_v}, \alpha_o, \alpha_m)$

Let $\mathbf{TD}_s = tdsub(\overline{\texttt{s}}, \overline{v}, o, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m],$

$\mathbf{TD}_m, \overline{\mathbf{TD}_v}, \mathbf{TD}_o)$

(Note: to simplify the definition of $tdsub$, we are treating `this`, which maps to the object identifier as another argument.)

$$\mathbf{TD}_s = \dfrac{\dfrac{\cdots}{\Gamma, pc_2 \cup s_o, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}} : \tau_s \backslash \mathcal{C}_s}\ (\text{Seq})}{\Gamma, pc_2 \cup s_o, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}} : t'_s \backslash \mathcal{C}'_s}\ (\text{Sub})$$

So, (we can build)

$$\mathbf{TD}' = \dfrac{\dfrac{\cdots}{\Gamma, pc_2 \cup s_o, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}} : \tau_s \backslash \mathcal{C}_s}\ (\text{Seq})}{\Gamma, pc_1, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}} : t \backslash \mathcal{C}}\ (\text{Sub})$$

and $\mathcal{C}'_s \subseteq \mathcal{C}_\top$ and $\mathcal{C} \subseteq \mathcal{C}_\top$.

11. $tdsuper(\mathbf{TD}, \mathtt{C}, \mathcal{C}_\top) = \mathbf{TD}'$

Where

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_o \qquad \overline{\mathbf{TD}_v}}{\Gamma, pc_2, \mathcal{H} \vdash o.\mathtt{super}(\mathtt{C},\overline{v}) : t_m \backslash \ldots}\ (\text{Super})}{\Gamma, pc_1, \mathcal{H} \vdash o.\mathtt{super}(\mathtt{C},\overline{v}) : t \backslash \mathcal{C}}\ (\text{Sub})$$

$$\mathbf{TD}_o = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H} \vdash o : t_o \backslash \mathcal{C}_o}\ (\text{Sub})$$

$$\overline{\mathbf{TD}_v} = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H} \vdash \overline{v} : \overline{t_v} \backslash \overline{\mathcal{C}_v}}\ (\text{Sub})$$

$cnbody(\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{super}(\overline{\mathtt{e}}); \overline{\mathtt{s}})$ and

$\mathbf{TD}_m = TDT(\mathtt{C}, \mathtt{K})$

and $\mathtt{C} <: \mathtt{D}$

and $LT(\mathtt{C}, \mathtt{K}) = \forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}_r$ and $\overline{t'_l} = \theta(\overline{t_l}, \mathtt{C}, \mathtt{m}, \overline{\alpha_v}, \alpha_o, \alpha_m)$

Let $\mathbf{TD}_s = tdsub(\overline{\mathtt{s}}, \overline{v}, o, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m],$

$\mathbf{TD}_m, \overline{\mathbf{TD}_v}, \mathbf{TD}_o)$

(Note: to simplify the definition of $tdsub$, we are treating `this`, which maps to the object identifier as another argument.)

$$\mathbf{TD}_s = \dfrac{\dfrac{\cdots}{\Gamma, pc_2 \cup s_o, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\mathtt{this}.\mathtt{super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}} : \tau_s \backslash \mathcal{C}_s}\ (\text{Seq})}{\Gamma, pc_2 \cup s_o, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\mathtt{this}.\mathtt{super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}} : t'_s \backslash \mathcal{C}'_s}\ (\text{Sub})$$

So, (we can build)

$$\mathbf{TD}' = \dfrac{\dfrac{\cdots}{\Gamma, pc_2 \cup s_o, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}} : \tau_s \backslash \mathcal{C}_s}\ (\text{Seq})}{\Gamma, pc_1, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}} : t \backslash \mathcal{C}}\ (\text{Sub})$$

and $\mathcal{C}'_s \subseteq \mathcal{C}_\top$ and $\mathcal{C} \subseteq \mathcal{C}_\top$.

12. $tdinput(\mathbf{TD}, \mathtt{c}, \mathcal{C}_\top) = \mathbf{TD}'$

$$\mathbf{TD} = \dfrac{\cdots}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{read_L(fd)} : t \backslash \mathcal{C}}\ (\text{Sub})$$

$$\mathbf{TD}' = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{c} : \langle\, pc_1, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\ (\text{Const})}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{c} : t \backslash \mathcal{C}}\ (\text{Sub})$$

and $\mathcal{C} \subseteq \mathcal{C}_\top$.

13. $tdoutput(\mathbf{TD}, \mathcal{C}_\top) = \mathbf{TD}'$

$$\mathbf{TD} = \dfrac{\cdots}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{write_L(c, fd)} : t \backslash \mathcal{C}}\ (\text{Sub})$$

$$\mathbf{TD}' = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{;} : \langle\, pc_1, \emptyset, \mathtt{void} \,\rangle \backslash \emptyset}\ (\text{No-op})}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{;} : t \backslash \mathcal{C}}\ (\text{Sub})$$

and $\mathcal{C} \subseteq \mathcal{C}_\top$.

The $ucon*$ and $dcon*$ functions are used to dis-assemble and re-assemble type derivations for evaluating expressions under a context reduction. The $ucon*$ functions given in Definition 4.4 are used to take apart the type derivation tree and return the internal type derivation of the context. For example, if the left expression of an operator is not yet a value, $uconop$ takes the type derivation and returns the type derivation of this left expression, so that it may be reduced. The $dcon*$ functions given in Definition 4.5 are used to build a type derivation after a context reduction. A reduction step alters some internal part of the type derivation, wherever the context reduction occurs. $dcon*$ functions are used to re-assemble the rest of the type derivation tree after the context reduction.

Definition 4.4 ($ucon*$)

1. $uconop(\mathbf{TD}) = \mathbf{TD}'$

   $$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \mathbf{TD}_2}{\Gamma, pc_2, H \vdash \mathsf{e}_1 \oplus \mathsf{e}_2 : \tau \backslash \ldots}\ \text{(Op)}}{\Gamma, pc_1, H \vdash \mathsf{e}_1 \oplus \mathsf{e}_2 : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   $\mathbf{TD}' = \mathbf{TD}_1$

2. The remaining cases follow a similar structure.

Definition 4.5 ($dcon*$)

1. $dconop(\mathbf{TD}, \mathbf{TD}'_1, \mathcal{C}_\top) = \mathbf{TD}'$

   $$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \mathbf{TD}_2}{\Gamma, pc_2, \mathcal{H} \vdash \mathsf{e}_1 \oplus \mathsf{e}_2 : \tau \backslash \ldots}\ \text{(Op)}}{\Gamma, pc_1, \mathcal{H} \vdash \mathsf{e}_1 \oplus \mathsf{e}_2 : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   $$\mathbf{TD}'_1 = \dfrac{\ldots}{\Gamma, pc_2, \mathcal{H}' \vdash \mathsf{e}'_1 : \ldots}\ \text{(Sub)}$$

   $\mathbf{TD}'_2 = tdheap(\mathbf{TD}_2, \mathcal{H}')$

   $$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}'_1 \qquad \mathbf{TD}'_2}{\Gamma, pc_2, \mathcal{H}' \vdash \mathsf{e}'_1 \oplus \mathsf{e}_2 : \tau' \backslash \ldots}\ \text{(Op)}}{\Gamma, pc_1, \mathcal{H}' \vdash \mathsf{e}'_1 \oplus \mathsf{e}_2 : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   and $\mathcal{C} \subseteq \mathcal{C}_\top$.

2. The remaining cases follow a similar structure.

The $tdheap$ function in Definition 4.6 is used to change the heap environment of a type derivation. This is necessary when a context reduction may add something to the heap environment, and the rest of the type derivation must be updated to use this new heap environment.

Definition 4.6 ($tdheap$)

$tdheap(\mathbf{TD}, \mathcal{H}') = \mathbf{TD}'$

1. (Const)

$$\mathbf{TD} = \dfrac{\dfrac{}{\Gamma, pc_2, \mathcal{H} \vdash v : \langle\, pc_2, \emptyset, \texttt{int} \,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_1, \mathcal{H} \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{}{\Gamma, pc_2, \mathcal{H}' \vdash v : \langle\, pc_2, \emptyset, \texttt{int} \,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_1, \mathcal{H}' \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

2. (Op)

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \mathbf{TD}_2}{\Gamma, pc_2, \mathcal{H} \vdash \texttt{e}_1 \oplus \texttt{e}_2 : \tau \backslash \ldots}\ \text{(Op)}}{\Gamma, pc_1, \mathcal{H} \vdash \texttt{e}_1 \oplus \texttt{e}_2 : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_1' = tdheap(\mathbf{TD}_1, \mathcal{H}')\ \mathbf{TD}_2' = tdheap(\mathbf{TD}_2, \mathcal{H}')$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}_1' \qquad \mathbf{TD}_2'}{\Gamma, pc_2, \mathcal{H}' \vdash \texttt{e}_1 \oplus \texttt{e}_2 : \tau \backslash \ldots}\ \text{(Op)}}{\Gamma, pc_1, \mathcal{H}' \vdash \texttt{e}_1 \oplus \texttt{e}_2 : t \backslash \mathcal{C}}\ \text{(Sub)}$$

3. The remaining cases follow a similar structure.

The $tdsub$ function given in Definition 4.7 is used for building the type derivation after a method invocation step. $\mathbf{TD}_m$ refers to the type derivation of the method body, and $\overline{\mathbf{TD}_v}$ refer to the type derivations of the arguments (including `this`). $tdsub$ also takes as arguments the mappings of the type variables to be replaced in $\mathbf{TD}_m$. The result of this function is the type derivation of the method body, with all of the types of the method arguments replaced in the method body's type derivation.

Definition 4.7 ($tdsub$**)**

$$tdsub(\varepsilon, \overline{v}, [\overline{t_l} \mapsto \overline{t_l'}], [s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t_r'], \mathbf{TD}_m, \overline{\mathbf{TD}_v}) = \mathbf{TD}'$$

1. $\varepsilon = \texttt{x}$

$$\mathbf{TD}_m = \dfrac{\dfrac{}{[\overline{\texttt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \texttt{x} : \langle\, s_x \cup s_p, f_x, \alpha_x \,\rangle \backslash \emptyset}\ \text{(Var)}}{[\overline{\texttt{x}} \mapsto \overline{t_x}] s_p, \emptyset \vdash \texttt{x} : t_m \backslash \mathcal{C}_m}\ \text{(Sub)}$$

$\{\langle\, s_x \cup s_p, f_x, \alpha_x \,\rangle <: t_m\} \subseteq \mathcal{C}_m$

The constraint sets in the conclusion of $\overline{\mathbf{TD}_v}$ are $\overline{\mathcal{C}_v}$.

(a) v is a constant.

$$\mathbf{TD}' = \dfrac{\dfrac{}{\Gamma, pc, \mathcal{H} \vdash v : \langle\, pc, \emptyset, \texttt{int} \,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc, \mathcal{H} \vdash v : t'_m \backslash \mathcal{C}'_m}\ \text{(Sub)}$$

$$t'_m = [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_m$$

$$\mathcal{C}'_m = \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \cup \overline{\mathcal{C}_v})$$

(b) v is an object identifier

$$\mathbf{TD}' = \dfrac{\dfrac{\mathcal{H}(o) = t_o \backslash \mathcal{C}_o}{\Gamma, pc, \mathcal{H} \vdash v : \langle\, pc \cup s_o, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o}\ \text{(Heap)}}{\Gamma, pc, \mathcal{H} \vdash v : t'_m \backslash \mathcal{C}'_m}\ \text{(Sub)}$$

$$t'_m = [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_m$$

$$\mathcal{C}'_m = \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \cup \overline{\mathcal{C}_v})$$

2. $\varepsilon = \mathtt{e}_1 \oplus \mathtt{e}_2$

$$\mathbf{TD}_m = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \mathbf{TD}_2}{[\overline{\mathtt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \mathtt{e}_1 \oplus \mathtt{e}_2 : \langle\, s_1 \cup s_2, \emptyset, \texttt{int} \,\rangle \backslash \mathcal{C}_1 \cup \mathcal{C}_2}\ \text{(Op)}}{[\overline{\mathtt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \mathtt{e}_1 \oplus \mathtt{e}_2 : t_3 \backslash \mathcal{C}_m}\ \text{(Sub)}$$

$$\mathbf{TD}_1 = \dfrac{\cdots}{[\overline{\mathtt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \mathtt{e}_1 : t_1 \backslash \mathcal{C}_1}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\cdots}{[\overline{\mathtt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \mathtt{e}_2 : t_2 \backslash \mathcal{C}_2}\ \text{(Sub)}$$

$$\mathbf{TD}'_1 = \mathit{tdsub}(\mathtt{e}_1, \overline{v}, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r], t_1, \mathcal{C}_1, \mathbf{TD}_1, \overline{\mathbf{TD}_v})$$

That is,

$$\mathbf{TD}'_1 = \dfrac{\cdots}{\Gamma, pc, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_1 : t'_1 \backslash \mathcal{C}'_1}\ \text{(Sub)}$$

$$\mathbf{TD}'_2 = \mathit{tdsub}(\mathtt{e}_1, \overline{v}, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r], t_2, \mathcal{C}_2, \mathbf{TD}_2, \overline{\mathbf{TD}_v})$$

That is,

$$\mathbf{TD}'_2 = \frac{\cdots}{\Gamma, pc, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_2 : t'_2 \backslash \mathcal{C}'_2}\ (\text{Sub})$$

The constraint sets in the conclusion of $\overline{\mathbf{TD}_v}$ are $\overline{\mathcal{C}_v}$.

$$\mathbf{TD}' = \frac{\dfrac{\mathbf{TD}'_1 \qquad \mathbf{TD}'_2}{\Gamma, pc, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_1 \oplus \mathtt{e}_2 : \langle\, s'_1 \cup s'_2, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}'_1 \cup \mathcal{C}'_2}\ (\text{Op})}{\begin{array}{c}\Gamma, pc, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_1 \oplus \mathtt{e}_2 : [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_3 \backslash \\ [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C} \cup \overline{\mathcal{C}_v}\end{array}\ (\text{Sub})$$

3. The remaining cases follow a similar structure.

The *conread* and *conwrite* functions given in Definition 4.8 return the concrete secrecy labels of the expression to be read or written. This is used to decide whether a read or write expression should proceed or result in a check failure.

Definition 4.8 (*con)**

1. $conread(\mathbf{TD}, \mathcal{C}_\top) = S$

   $S = Con(s_f, \mathcal{C}_\top)$

   $$\mathbf{TD} = \frac{\dfrac{\dfrac{}{\Gamma, pc_3, H \vdash \mathtt{fd} : \langle\, pc_3, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\ (\text{Const})}{\Gamma, pc_2, H \vdash \mathtt{fd} : t_f \backslash \mathcal{C}_f}\ (\text{Sub})}{\dfrac{\Gamma, pc_2, H \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{fd}) : \langle\, s_f \cup \mathtt{L}, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_f \cup SC(\mathtt{L}, s_f)}{\Gamma, pc_1, H \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{fd}) : t \backslash \mathcal{C}}\ (\text{Sub})}\ (\text{Input})$$

   $\mathcal{C} \subseteq \mathcal{C}_\top$

2. $conwrite(\mathbf{TD}, \mathcal{C}_\top) = S$

   $S = Con(s_c, \mathcal{C}_\top) \cup Con(s_f, \mathcal{C}_\top)$

   $$\mathbf{TD} = \frac{\dfrac{\mathbf{TD}_c \qquad \mathbf{TD}_f}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{write}_{\mathtt{L}}(\mathtt{c}, \mathtt{fd}) : \langle\, s_c \cup s_f, \emptyset, \mathtt{void} \,\rangle \backslash \mathcal{C}'}\ (\text{Output})}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{write}_{\mathtt{L}}(\mathtt{c}, \mathtt{fd}) : t \backslash \mathcal{C}}\ (\text{Sub})$$

   $$\mathbf{TD}_c = \frac{\dfrac{}{\Gamma, pc_3, \mathcal{H} \vdash \mathtt{c} : \langle\, pc_3, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\ (\text{Const})}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{c} : t_c \backslash \mathcal{C}_c}\ (\text{Sub})$$

$$\mathbf{TD}_f = \dfrac{\dfrac{}{\Gamma, pc_4, \mathcal{H} \vdash \texttt{fd} : \langle\, pc_4, \emptyset, \texttt{int}\,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_2, \mathcal{H} \vdash \texttt{fd} : t_f \backslash \mathcal{C}_f}\ \text{(Sub)}$$

$$\mathcal{C} \subseteq \mathcal{C}_\top$$

4.3 Overview of Proof Technique

Before proceeding with the formal results, we now provide a more detailed overview of the proof technique, in order to assist the reader.

Our noninterference property states that for a typeable program P, any two runs of the program differing only in high input streams will produce the same low output streams, and that the resulting low input streams are also equivalent. The latter condition is necessary since the size of the low input streams after computation may convey secret information, such as if one stream had been read five times, and the other seven; the attacker would know that a change was made to a high input. We specify that the values must be integers for this theorem, as it intuitively doesn't make sense to input or output heap locations (pointers). Since our system is termination-insensitive, both runs of the program are assumed to terminate normally.

As previously stated, the operational semantics is augmented with a syntactic version of the type derivation of each expression. We prove the standard subject reduction property on this semantics; however, here it takes the form of proving that after each reduction step, the syntactic type derivation is actually a *valid* type derivation for the resulting expression, and the resulting type is the same as before the reduction step. Hence, at every reduction step, we have a valid type derivation tree for each expression. After proving this subject reduction property, we can now directly prove a soundness property, that no execution of a well-typed program results in a security check failure. With a valid type derivation at every step of the reduction, and since well-typed

programs do not permit failed checks, we prove that these executions will never take a step to a check failure state.

For proving noninterference, we first distinguish input and output channels that are *low* from those that are *high*. The observational behavior of the low user is defined by an arbitrary set of secrecy labels `Low`. Hence, low data is any data labeled with some set of labels `L`, such that $\texttt{L} \subseteq \texttt{Low}$. High data is any data not observable to the low user, e.g., data labeled with a set of secrecy labels `High`, such that $\texttt{High} \not\subseteq \texttt{Low}$. Note that our results apply for *any* security level `Low`.

Now that we have precisely defined the high and low channels, we use this definition to determine the difference between high and low execution behavior. We distinguish low reduction steps from high reduction steps based on the type derivation of the expression being computed on, such that every reduction step is either a low step or a high step. Since we have the type derivation, we use the function Con to establish the concrete type of an expression, and thereby determine whether the step to be taken is *high* or *low*, based on the security level `Low`. We then show that for typeable programs, assuming two executions where the low input streams are identical, they each take the same low steps, with possibly differing high steps between. Hence, when the execution finishes, the result is a low-equivalent trace of inputs and outputs.

This is shown via bisimulation of the execution of two programs with identical low input streams. Bisimulation is defined on expressions with their type derivation (which is assumed to be canonical). Using the type derivation and Definition 4.2, we establish a notion of high versus low values, based on the type of the value.

The bisimulation relation, $\simeq_{\texttt{Low}}$, states that two expressions are bisimilar, if and only if they have the same structure (e.g. both are assignment statements), and the high labeled values may

differ, while low values in the configurations must be equivalent; it also stipulates that the derivation trees have the same type variable at every use of (Sub). This latter requirement allows us to strictly align the executions, based on the type derivation tree. Since high and low steps are based on typing, this relationship allows us to show that any low steps that are taken must be the same, since the type is the same, anything that is low in one execution must also be low in the other. We further establish bisimulation of the heaps, where data that is low in one heap must be *identical* in the other heap. The final part of the bisimulation asserts the equivalence of low input and output streams.

We then prove that throughout the execution of two runs of a program, we maintain this *low*-bisimularity relationship. Since the bisimulation has the property that low streams are equivalent, maintaining this relationship for the duration of the computation produces noninterference. To accomplish this, we break the computation of each run into a series of alternating *n* high steps, followed by *one* low step, where the number of high steps may also be zero. Hence, an execution of $\rightsquigarrow^*$ is defined as $\rightsquigarrow_h^* \ldots \rightsquigarrow_l \ldots \rightsquigarrow_h^* \ldots \rightsquigarrow_l$. We then prove two lemmas; the low reduction lemma states that for bisimilar expressions, if one run takes a low step, the other run takes the identical low step; the high reduction lemma states that if one run takes a number of high steps, the other will also take some (possibly different) number of high steps, such that the expressions are bisimilar afterwards. Since high steps only read and write from high streams, and only alter high memory locations, and since all of the low steps taken are the same, noninterference follows by these two lemmas and the bisimulation definition.

This proof of noninterference by our augmented semantics allows us to establish noninterference of a unaugmented semantics, under which such programs might actually execute. The augmented semantics is a proof tool only, which we use to establish the noninterference of programs

in a usual semantic definition. Therefore, we define an unaugmented semantics that works directly on expressions (and statements), and is identical to the augmented semantics, only lacking the type derivation. The noninterference result on this semantics is the same as that of the labeled semantics: that if a program is typeable, for two terminating executions of the program that differ only in high input streams, the resulting low input and output streams are equivalent.

Now that we have informally described our proof technique, the subsequent sections describe our formal results.

4.4 Subject Reduction and Soundness

We now show the subject reduction and soundness properties for our type system and semantics. We make the following simplifying assumptions in our proofs, in order to reduce the number of cases. Unless otherwise relevant, constants are assumed to be integers; this simplifies the need to case over booleans and `null`. Constructor calls generally assume that the super-class is not `Object`, since calls to `e.super(Object)` are trivial anyway. We proceed with some necessary definitions and lemmas.

Definition 4.9 formalizes a valid type derivation. We write $\mathbf{TD} \rhd \varepsilon, H, \Gamma, pc, t, \mathcal{C}$, which means $\mathbf{TD}$ is a valid type derivation for expression (or statement) ε and heap H. We include Γ, pc, t, and $\mathcal{C}$ in the definition as a convenience to make our formal statements clearer.

Definition 4.9 (Valid $\mathbf{TD}$) $\mathbf{TD} \rhd \varepsilon, H, \Gamma, pc, t, \mathcal{C}$ *iff* $\mathbf{TD}$ *concludes* $\Gamma, pc, \mathcal{H} \vdash \varepsilon : t \backslash \mathcal{C}$*, and* $\mathcal{C}$ *is consistent, and* $\mathcal{H} \vdash H$.

Lemma 4.10 gives the formal proof that any well-typed expression has a canonical type derivation. This is clear from the (Sub) type rule, where any two uses of (Sub) can be collapsed into

a single instance, and new uses of (Sub) may be inserted by introducing a fresh type variable into the typing and adding $\tau <: t$ to the constraint set, and maintaining the same program counter pc.

Lemma 4.10 (Canonical Proofs) *If* $\Gamma, pc, pc_i, \mathcal{H} \vdash \varepsilon : \tau \backslash \mathcal{C}$*, then there exists a derivation* $\Gamma, pc', pc_i', \mathcal{H} \vdash \varepsilon : t' \backslash \mathcal{C}'$ *that is canonical.*

Proof. By induction on the type derivation of ε. We have two cases.

1. The final rule in the derivation of $\Gamma, pc, pc_i, \mathcal{H} \vdash \varepsilon : \tau \backslash \mathcal{C}$ is (Sub). The premise to this rule is now a derivation of $\Gamma, pc', pc_i', \mathcal{H} \vdash \varepsilon : \tau' \backslash \mathcal{C}'$, where $pc' \subseteq pc$, $pc_i \subseteq pc_i'$, $\mathcal{C} \cup \{\tau' <: \tau\} \subseteq \mathcal{C}'$, and $\mathcal{C}'$ is closed. By induction, this has a canonical derivation, $\Gamma, pc', pc_i', \mathcal{H} \vdash \varepsilon : t \backslash \mathcal{C}''$.

2. The final rule in the derivation of $\Gamma, pc, pc_i, \mathcal{H} \vdash \varepsilon : \tau \backslash \mathcal{C}$ is not (Sub). Then by induction, this final rule's premises have canonical derivations. Hence, $\Gamma, pc, pc_i, \mathcal{H} \vdash \varepsilon : \tau' \backslash \mathcal{C}'$ can be derived using the original rule. Conclude with a use of (Sub), introducing fresh type variables, we have $\Gamma, pc, pc_i, \mathcal{H} \vdash \varepsilon : t' \backslash \mathcal{C}''$, which is consistent, since the original constraint set was consistent, and any typings derived from t' in $\mathcal{C}''$ can also be derived from τ', using transitivity closure rules.

□

The semantic function $tdheap$ is used to add to the heap environment in a type derivation. Lemma 4.11 shows that these type derivations remain valid.

Lemma 4.11 (Heap Extension) *If* $\mathbf{TD} \rhd \varepsilon, H, \Gamma, pc, t, \mathcal{C}$ *and* $\mathcal{H}'$ *extends* $\mathcal{H}$ *and* $\mathcal{H}' \vdash H'$*, and* $\mathbf{TD}' = tdheap(\mathbf{TD}, \mathcal{H}')$*, then* $\mathbf{TD}' \rhd \varepsilon, H', \Gamma, pc, t, \mathcal{C}$.

Proof. By induction on the derivation of $\mathbf{TD}$, using the definition of $tdheap$. Since $\mathcal{H}'$ extends $\mathcal{H}$, we are only adding things to the heap environment. Since the typing was valid without these additions, the typing remains valid with the additions, which will not appear in the derivation. □

The function $tdsub$ is used in the semantics to substitute values and their respective type derivations for variables in an expression and derivation that comes from a method or constructor. Subsitution Lemma 4.12 shows that under the necessary assumptions of the validity of the existing type derivations and the consistency of the closure set from the ($Method$) closure rule, that the type derivation produced by $tdsub$ is valid for the expression with values substituted for arguments.

Lemma 4.12 (Substitution)

If $\mathbf{TD} = tdsub(\varepsilon, \overline{v}, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r], \mathbf{TD}_m, \overline{\mathbf{TD}_v})$, *and* $\mathbf{TD}_m \rhd \varepsilon, \emptyset, [\overline{\mathtt{x}} \mapsto \overline{t_x}], pc, t, \mathcal{C}_m$, *and* $\forall \mathbf{TD}_v \in \overline{\mathbf{TD}_v}$, $\mathbf{TD}_v \rhd v, H, \Gamma, pc', t_v, \mathcal{C}_v$ *and* $pc' \subseteq pc$, *and* $FTV(t_m \backslash \mathcal{C}_m) \subseteq \{\overline{t_l}, s_p, \overline{t_x}, t_r\}$, *and* $Closure([\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \cup \overline{\mathcal{C}_v})$ *is consistent, then* $\mathbf{TD} \rhd [\overline{\mathtt{x}} \mapsto \overline{v}]\varepsilon, H, \Gamma, pc, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_m, Closure(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \cup \overline{\mathcal{C}_v})$.

Proof. By induction on ε and the corresponding $\mathbf{TD}_m$.

1. $\varepsilon = \mathtt{x}$

   Obviously, $\mathtt{x} \in \overline{\mathtt{x}}$, otherwise $\mathbf{TD}_m$ would not be typable.

   So, by $tdsub(\varepsilon, \overline{v}, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r], \mathbf{TD}_m, \overline{\mathbf{TD}_v})$, we have the following.

   $$\mathbf{TD}_m = \dfrac{\dfrac{}{[\overline{\mathtt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \mathtt{x} : \langle\, s_x \cup s_p, f_x, \alpha_x \,\rangle \backslash \emptyset}\ \text{(Var)}}{[\overline{\mathtt{x}} \mapsto \overline{t_x}]s_p, \emptyset \vdash \mathtt{x} : t_m \backslash \mathcal{C}_m}\ \text{(Sub)}$$

   The constraint sets in the conclusion of $\overline{\mathbf{TD}_v}$ are $\overline{\mathcal{C}_v}$.

   We now have two cases for the value of v that corresponds to $\mathtt{x}$.

   (a) v is a constant.

   $$\mathbf{TD} = \dfrac{\dfrac{}{\Gamma, pc, \mathcal{H} \vdash v : \langle\, pc, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc, \mathcal{H} \vdash v : t'_m \backslash \mathcal{C}'_m}\ \text{(Sub)}$$

Where $t'_m = [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_m$ and $\mathcal{C}'_m = Closure(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \cup \overline{\mathcal{C}_v})$.

Since $t_m \in FTV(t_m \backslash \mathcal{C}_m)$, and by assumption $FTV(t_m \backslash \mathcal{C}_m) \subseteq \{\overline{t_l}, s_p, \overline{t_x}, t_r\}$, and since t_m are not in $[s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]$, we must have t_m in $\overline{t_l}$. Hence, there is a t'_m in $\overline{t'_l}$, such that $[t_m \mapsto t'_m]$ is a part of $[\overline{t_l} \mapsto \overline{t'_l}]$.

Since $\mathbf{TD}_m$ is a valid type derivation, according to (Sub), $\{\langle\, s_x \cup s_p, f_x, \alpha_x \,\rangle <: t_m\} \subseteq \mathcal{C}_m$.

Since $[\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \subseteq \mathcal{C}'_m$, applying the substitutions, we have $\{\langle\, s_v \cup pc, f_v, \alpha_v \,\rangle <: t'_m\} \subseteq \mathcal{C}'_m$.

Now, since $\overline{\mathbf{TD}_v}$ are valid type derivations, we have the following.

$$\mathbf{TD}_v = \dfrac{\dfrac{}{\Gamma, pc', \mathcal{H} \vdash v : \langle\, pc', \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc', \mathcal{H} \vdash v : t_v \backslash \mathcal{C}_v}\ \text{(Sub)}$$

Since $\mathbf{TD}_v$ is a valid type derivation, we have $\{\langle\, pc', \emptyset, \mathtt{int} \,\rangle <: t_v\} \subseteq \mathcal{C}_v$.

Since $\mathcal{C}_v \subseteq \mathcal{C}'_m$, and we just showed $\{\langle\, s_v \cup pc, f_v, \alpha_v \,\rangle <: t'_m\} \subseteq \mathcal{C}'_m$, by transitivity rules, we have $\{\langle\, pc' \cup pc, \emptyset, \mathtt{int} \,\rangle <: t'_m\} \subseteq \mathcal{C}'_m$, Since $pc' \subseteq pc$, this is the same as $\{\langle\, pc, \emptyset, \mathtt{int} \,\rangle <: t'_m\} \subseteq \mathcal{C}'_m$. Hence, by (Const) and (Sub), and since by assumption $\mathcal{C}'_m$ is consistent, we conclude that $\mathbf{TD} \rhd [\overline{\mathtt{x}} \mapsto \overline{v}]\varepsilon, H, \Gamma, pc, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_m, Closure(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \cup \overline{\mathcal{C}_v})$.

(b) v is an object identifier.

$$\mathbf{TD} = \dfrac{\dfrac{\mathcal{H}(v) = t_o \backslash \mathcal{C}_o}{\Gamma, pc, \mathcal{H} \vdash v : \langle\, pc \cup s_o, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o}\ \text{(Heap)}}{\Gamma, pc, \mathcal{H} \vdash v : t'_m \backslash \mathcal{C}'_m}\ \text{(Sub)}$$

Where $t'_m = [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_m$ and $\mathcal{C}'_m = Closure(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \cup \overline{\mathcal{C}_v})$.

Since $t_m \in FTV(t_m \backslash \mathcal{C}_m)$, and by assumption $FTV(t_m \backslash \mathcal{C}_m) \subseteq \{\overline{t_l}, s_p, \overline{t_x}, t_r\}$, and since t_m are not in $[s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]$, we must have t_m in $\overline{t_l}$. Hence, there is a t'_m in $\overline{t'_l}$, such that $[t_m \mapsto t'_m]$ is a part of $[\overline{t_l} \mapsto \overline{t'_l}]$.

Since $\mathbf{TD}_m$ is a valid type derivation, according to (Sub), $\{\langle\, s_x \cup s_p, f_x, \alpha_x \,\rangle <: t_m\} \subseteq \mathcal{C}_m$.

Since $[\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \subseteq \mathcal{C}'_m$, applying the substitutions, we have $\{\langle\, s_v \cup pc, f_v, \alpha_v \,\rangle <: t'_m\} \subseteq \mathcal{C}'_m$.

We now have the following.

$$\mathbf{TD}_v = \dfrac{\dfrac{\mathcal{H}(v) = t_o \backslash \mathcal{C}_o}{\Gamma, pc', \mathcal{H} \vdash v : \langle\, pc' \cup s_o, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o}\ \text{(Heap)}}{\Gamma, pc', \mathcal{H} \vdash v : t_v \backslash \mathcal{C}_v}\ \text{(Sub)}$$

Since $\mathbf{TD}_v$ is a valid type derivation, we have $\{\langle\, pc' \cup s_o, f_o, \alpha_o \,\rangle <: t_v\} \subseteq \mathcal{C}_v$.

Since $\mathcal{C}_v \subseteq \mathcal{C}'_m$, and we just showed $\{\langle\, s_v \cup pc, f_v, \alpha_v \,\rangle <: t'_m\} \subseteq \mathcal{C}'_m$, by transitivity rules, we have $\{\langle\, pc' \cup pc, f_o, \mathtt{C} \,\rangle <: t'_m\} \subseteq \mathcal{C}'_m$, and since $pc' \subseteq pc$, this is the same as $\{\langle\, pc \cup s_o, f_o, \alpha_o \,\rangle <: t'_m\} \subseteq \mathcal{C}'_m$.

Hence, by (Oid) and (Sub), and since by assumption $\mathcal{C}'_m$ is consistent, we conclude that $\mathbf{TD} \rhd [\overline{\mathtt{x}} \mapsto \overline{v}]\varepsilon, H, \Gamma, pc, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_m, \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \cup \overline{\mathcal{C}_v})$.

2. $\varepsilon = \mathtt{e}_1 \oplus \mathtt{e}_2$

So, by $\mathit{tdsub}(\varepsilon, \overline{v}, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r], \mathbf{TD}_m, \overline{\mathbf{TD}_v})$, we have the following.

$$\mathbf{TD}_m = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \mathbf{TD}_2}{[\overline{\mathtt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \mathtt{e}_1 \oplus \mathtt{e}_2 : \langle\, s_1 \cup s_2, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_1 \cup \mathcal{C}_2}\ \text{(Op)}}{[\overline{\mathtt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \mathtt{e}_1 \oplus \mathtt{e}_2 : t_m \backslash \mathcal{C}_m}\ \text{(Sub)}$$

$$\mathbf{TD}_1 = \dfrac{\cdots}{[\overline{\mathtt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \mathtt{e}_1 : t_1 \backslash \mathcal{C}_1}\ \text{(Sub)}$$

$\mathbf{TD}_2 =$

$$\frac{\cdots}{[\overline{\mathtt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \mathtt{e}_2 : t_2 \backslash \mathcal{C}_2} \text{ (Sub)}$$

$\mathbf{TD}'_1 = tdsub(\mathtt{e}_1, \overline{v}, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r], \mathbf{TD}_1, \overline{\mathbf{TD}_v})$

That is,

$\mathbf{TD}'_1 =$

$$\frac{\cdots}{\Gamma, pc, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_1 : t'_1 \backslash \mathcal{C}'_1} \text{ (Sub)}$$

$\mathbf{TD}'_2 = tdsub(\mathtt{e}_1, \overline{v}, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r], \mathbf{TD}_2, \overline{\mathbf{TD}_v})$

That is,

$\mathbf{TD}'_2 =$

$$\frac{\cdots}{\Gamma, pc, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_2 : t'_2 \backslash \mathcal{C}'_2} \text{ (Sub)}$$

The constraint sets in the conclusion of $\overline{\mathbf{TD}_v}$ are $\overline{\mathcal{C}_v}$.

$$\mathbf{TD}' = \frac{\dfrac{\mathbf{TD}'_1 \qquad \mathbf{TD}'_2}{\Gamma, pc, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_1 \oplus \mathtt{e}_2 : \langle\, s'_1 \cup s'_2, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}'_1 \cup \mathcal{C}'_2} \text{ (Op)}}{\Gamma, pc, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_1 \oplus \mathtt{e}_2 : t'_m \backslash \mathcal{C}'_m} \text{ (Sub)}$$

Where $t'_m = [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_m$ and $\mathcal{C}'_m = \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \cup \overline{\mathcal{C}_v})$.

By induction,

$\mathbf{TD}'_1 \rhd [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_1, H, \Gamma, pc, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_1, \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r](\mathcal{C}_1) \cup \overline{\mathcal{C}_v})$, and $\mathbf{TD}'_2 \rhd [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_2, H, \Gamma, pc, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_2, \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r](\mathcal{C}_2) \cup \overline{\mathcal{C}_v})$,

In other words, $t'_1 = [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_1$ and $t'_1 = [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_2$, and $\mathcal{C}'_1 = \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r](\mathcal{C}_1) \cup \overline{\mathcal{C}_v})$ and $\mathcal{C}'_2 = \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r](\mathcal{C}_2) \cup \overline{\mathcal{C}_v})$.

Since $t_m \in FTV(t_m \backslash \mathcal{C}_m)$, and by assumption $FTV(t_m \backslash \mathcal{C}_m) \subseteq \{\overline{t_l}, s_p, \overline{t_x}, t_r\}$, and since t_m are not in $[s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]$, we must have t_m in $\overline{t_l}$. Hence, there is a t'_m in $\overline{t'_l}$, such that $[t_m \mapsto t'_m]$ is a part of $[\overline{t_l} \mapsto \overline{t'_l}]$.

Since $\mathbf{TD}_m$ is well-typed, we must have $\{\langle\, s_1 \cup s_2, \emptyset, \texttt{int} \,\rangle <: t_m\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \subseteq \mathcal{C}_m$. Since t'_1 and t'_2 are defined by the substitution as shown above, we have $[\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r](\{\langle\, s_1 \cup s_2, \emptyset, \texttt{int} \,\rangle <: t_m\}) \subseteq \mathcal{C}'_m$.

Since $\mathcal{C}'_m = \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \cup \overline{\mathcal{C}_v})$ we have the following. By $\mathcal{C}_1 \cup \mathcal{C}_2 \subseteq \mathcal{C}_m$, we have $[\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r](\mathcal{C}_1 \cup \mathcal{C}_2) \subseteq \mathcal{C}'_m$. Now, since $\overline{\mathcal{C}_v} \subseteq \mathcal{C}'_m$, and $\mathcal{C}'_m$ is closed, this means $\mathcal{C}'_1 \cup \mathcal{C}'_2 \subseteq \mathcal{C}'_m$.

We can now conclude by (Op) and (Sub) that $\mathbf{TD}' \rhd [\overline{\mathtt{x}} \mapsto \overline{v}]\varepsilon, H, \Gamma, pc, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_m, \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r](\mathcal{C}_m) \cup \overline{\mathcal{C}_v})$.

3. The remaining cases follow similarly, either directly, or by induction and use of the respective case of $tdsub$ and type rule.

□

We can now prove Subject Reduction Lemma 4.13, stating that the type derivation produced after a reduction is valid for the new expression (implying also that the new heap is well-typed), and that the reduced expression has the same Γ, pc, t, and $\mathcal{C}$ as the original expression. In other words, the type is *preserved* by the reduction step.

Lemma 4.13 (Subject Reduction) *If* $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \varepsilon', \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ *and* $\mathbf{TD} \rhd \varepsilon, H, \Gamma, pc, t, \mathcal{C}$*, then* $\mathbf{TD}' \rhd \varepsilon', H', \Gamma, pc, t, \mathcal{C}$.

Proof. By induction on the height of the context derivation tree, with case analysis.

1. (Field-R)

   So, $\varepsilon = o.\mathtt{f}$.

   By (Field-R) and $tdfield(\mathbf{TD}, v, \mathcal{C}_\top) = \mathbf{TD}'$, and since $\mathbf{TD}$ is a valid type derivation for ε, we have the following.

$$\mathbf{TD} = \dfrac{\dfrac{\dfrac{\mathcal{H}(o) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_3, \mathcal{H} \vdash o : \langle\, pc_3 \cup s_o, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o}\,\text{(Oid)}}{\Gamma, pc_2, \mathcal{H} \vdash o : t'_o \backslash \mathcal{C}'_o}\,\text{(Sub)}}{\dfrac{\Gamma, pc_2, \mathcal{H} \vdash o.\mathtt{f} : \langle\, s'_o \cup f'_o.\mathtt{f}.\mathsf{S}, f'_o.\mathtt{f}.\mathsf{F}, f'_o.\mathtt{f}.\mathsf{A} \,\rangle \backslash \mathcal{C}'_o}{\Gamma, pc_1, \mathcal{H} \vdash o.\mathtt{f} : t \backslash \mathcal{C}}\,\text{(Sub)}}\,\text{(Field)}$$

   Where $\mathcal{C}_o \cup \{\langle\, pc_3 \cup s_o, f_o, \alpha_o \,\rangle <: t'_o\} \cup \{\langle\, s'_o \cup f'_o.\mathtt{f}.\mathsf{S}, f'_o.\mathtt{f}.\mathsf{F}, f'_o.\mathtt{f}.\mathsf{A} \,\rangle <: t\} \subseteq \mathcal{C}$.

   According to the definition of $tdfield$, we now have two cases.

   (a) v is a constant

$$\mathbf{TD}' = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H} \vdash v : \langle\, pc_1, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\,\text{(Const)}}{\Gamma, pc_1, \mathcal{H} \vdash v : t \backslash \mathcal{C}}\,\text{(Sub)}$$

   Now, by assumption $\mathcal{H} \vdash H$. Hence, by (Heap), we have $\emptyset, \emptyset, \mathcal{H} \vdash v : t_v \backslash \mathcal{C}_v$. Since v is a constant, and the derivation is canonical, we have the following derivation.

$$\dfrac{\dfrac{}{\emptyset, \emptyset, \mathcal{H} \vdash v : \langle\, \emptyset, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\,\text{(Const)}}{\emptyset, \emptyset, \mathcal{H} \vdash v : t_v \backslash \mathcal{C}_v}\,\text{(Sub)}$$

   Where $\mathcal{C}_v = \{\langle\, \emptyset, \emptyset, \mathtt{int} \,\rangle <: t_v\}$.

   Again by (Heap), we have $\{f_o.\mathtt{f} <: \textbf{set}\ t_v\} \cup \mathcal{C}_v \subseteq \mathcal{C}_o$.

   Since $\mathcal{C}$ is closed, we have the following. By (Set), $\{t_v <: f_o.\mathtt{f}\} \subseteq \mathcal{C}$, so by transitivity closure rules, we have $\{\langle\, \emptyset, \emptyset, \mathtt{int} \,\rangle <: f_o.\mathtt{f}\} \subseteq \mathcal{C}$. From above, we have $\{\langle\, pc_3 \cup s_o, f_o, \alpha_o \,\rangle <: t'_o\} \subseteq \mathcal{C}$, so by ($*$-Field'), we have $\{\langle\, \emptyset, \emptyset, \mathtt{int} \,\rangle <: f'_o.\mathtt{f}\} \subseteq \mathcal{C}$; also, by ($\mathcal{S}$-Union), we have $\{pc_3 <: s'_o\} \subseteq \mathcal{C}$. From above, we have $\{\langle\, s'_o \cup f'_o.\mathtt{f}.\mathsf{S}, f'_o.\mathtt{f}.\mathsf{F}, f'_o.\mathtt{f}.\mathsf{A} \,\rangle <: t\} \subseteq \mathcal{C}$, so by ($*$-Trans), we have $\{\langle\, pc_3 \cup \emptyset, \emptyset, \mathtt{int} \,\rangle <: t\} \subseteq \mathcal{C}$, which is $\{\langle\, pc_3, \emptyset, \mathtt{int} \,\rangle <: t\} \subseteq \mathcal{C}$. Ac-

cording to uses of (Sub), we must have $pc_1 \subseteq pc_3$, hence by ($\mathcal{S}$-Union), $\{\langle\, pc_1, \emptyset, \texttt{int} \,\rangle <: t\} \subseteq \mathcal{C}$. So, we conclude that with (Const) and (Sub), $\mathbf{TD}' \rhd v, H, \Gamma, pc_1, t, \mathcal{C}$.

(b) v is an object identifier

$$\mathbf{TD}' = \dfrac{\dfrac{\mathcal{H}(v) = t'_v \backslash \mathcal{C}'_v}{\Gamma, pc_1, \mathcal{H} \vdash v : \langle\, pc_1 \cup s'_v, f'_v, \alpha'_v \,\rangle \backslash \mathcal{C}'_v}\ \text{(Oid)}}{\Gamma, pc_1, \mathcal{H} \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Now, by assumption $\mathcal{H} \vdash H$. Hence, by (Heap), we have $\emptyset, \emptyset, \mathcal{H} \vdash v : t'_v \backslash \mathcal{C}'_v$. Since v is a constant, and the derivation is canonical, we have the following derivation.

$$\dfrac{\dfrac{\mathcal{H}(v) = t'_v \backslash \mathcal{C}'_v}{\emptyset, \emptyset, \mathcal{H} \vdash v : \langle\, s'_v, f'_v, \alpha'_v \,\rangle \backslash \mathcal{C}'_v}\ \text{(Oid)}}{\emptyset, \emptyset, \mathcal{H} \vdash v : t_v \backslash \mathcal{C}_v}\ \text{(Sub)}$$

Where $\{t'_v <: t_v\} \cup \mathcal{C}'_v \subseteq \mathcal{C}_v$.

Again by (Heap), we have $\{f_o.\texttt{f} <: \textbf{set}\ t_v\} \cup \mathcal{C}_v \subseteq \mathcal{C}_o$.

Since $\mathcal{C}$ is closed, we have the following. By (Set), $\{t_v <: f_o.\texttt{f}\} \subseteq \mathcal{C}$, so by transitivity closure rules, we have $\{t'_v <: f_o.\texttt{f}\} \subseteq \mathcal{C}$. From above, we have $\{\langle\, pc_3 \cup s_o, f_o, \alpha_o \,\rangle <: t'_o\} \subseteq \mathcal{C}$, so by ($*$-Field'), we have $\{t'_v <: f'_o.\texttt{f}\} \subseteq \mathcal{C}$; also, by ($\mathcal{S}$-Union), we have $\{pc_3 <: s'_o\} \subseteq \mathcal{C}$. From above, we have $\{\langle\, s'_o \cup f'_o.\texttt{f}.\mathsf{S}, f'_o.\texttt{f}.\mathsf{F}, f'_o.\texttt{f}.\mathsf{A} \,\rangle <: t\} \subseteq \mathcal{C}$, so by ($*$-Trans), we have $\{\langle\, pc_3 \cup s'_v, f'_v, \alpha'_v \,\rangle <: t\} \subseteq \mathcal{C}$. According to uses of (Sub), we must have $pc_1 \subseteq pc_3$, hence by ($\mathcal{S}$-Union), $\{\langle\, pc_1 \cup s'_v, f'_v, \alpha'_v \,\rangle <: t\} \subseteq \mathcal{C}$. So, we conclude that with (Oid) and (Sub), $\mathbf{TD}' \rhd v, H, \Gamma, pc_1, t, \mathcal{C}$.

2. (Op-R)

So, $\varepsilon = \mathtt{c_a} \oplus \mathtt{c_b}$.

By (Op-R), and $\mathbf{TD}' = tdop(\mathbf{TD}, v, \mathcal{C}_\top)$, and since $\mathbf{TD}$ is a valid type derivation for ε, we have the following.

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_a \qquad \mathbf{TD}_b}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{c_a} \oplus \mathtt{c_b} : \langle\, s_a \cup s_b, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_a \cup \mathcal{C}_b}\ \text{(Op)}}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{c_a} \oplus \mathtt{c_b} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_a = \dfrac{\dfrac{}{\Gamma, pc_3, \mathcal{H} \vdash \mathtt{c_a} : \langle\, pc_3, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{c_a} : t_a \backslash \mathcal{C}_a}\ \text{(Sub)}$$

$$\mathbf{TD}_b = \dfrac{\dfrac{}{\Gamma, pc_4, \mathcal{H} \vdash \mathtt{c_b} : \langle\, pc_4, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{c_b} : t_b \backslash \mathcal{C}_b}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H} \vdash v : \langle\, pc_1, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_1, \mathcal{H} \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Since $\mathcal{C}$ is closed, we have the following. By (Sub), we have $\{\langle\, pc_3, \emptyset, \mathtt{int} \,\rangle <: t_a\} \subseteq \mathcal{C}$. Again by (Sub), $\{\langle\, s_a \cup s_b, \emptyset, \mathtt{int} \,\rangle <: t\} \subseteq \mathcal{C}$, so by ($\mathcal{S}$-Union), $\{\langle\, s_a, \emptyset, \mathtt{int} \,\rangle <: t\} \subseteq \mathcal{C}$. Hence by (*-Trans), $\{\langle\, pc_3, \emptyset, \mathtt{int} \,\rangle <: t\} \subseteq \mathcal{C}$. According to uses of (Sub), we must have $pc_1 \subseteq pc_3$, hence by ($\mathcal{S}$-Union), $\{\langle\, pc_1, \emptyset, \mathtt{int} \,\rangle <: t\} \subseteq \mathcal{C}$. So, we conclude that with (Const) and (Sub), $\mathbf{TD}' \rhd v, H, \Gamma, pc_1, t, \mathcal{C}$.

3. (Cast-R)

   So, $\varepsilon = (\mathtt{D})\ o$.

   By (Cast-R), and $\mathbf{TD}' = tdcast(\mathbf{TD}, o, \mathcal{C}_\top)$, and since $\mathbf{TD}$ is a valid type derivation for ε, we have the following.

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_o}{\Gamma, pc_2, \mathcal{H} \vdash (\mathtt{D})\ o : t'_o \backslash \mathcal{C}'_o}\ \text{(Cast)}}{\Gamma, pc_1, \mathcal{H} \vdash (\mathtt{D})\ o : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_o = \dfrac{\dfrac{\mathcal{H}(o) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_3, \mathcal{H} \vdash o : \langle\, pc_3 \cup s_o, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o}\ \text{(Oid)}}{\Gamma, pc_2, \mathcal{H} \vdash o : t'_o \backslash \mathcal{C}'_o}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathcal{H}(o) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_1, \mathcal{H} \vdash o : \langle\, pc_1 \cup s_o, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o}\ \text{(Oid)}}{\Gamma, pc_1, \mathcal{H} \vdash o : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Since $\mathcal{C}$ is closed, we have the following. According to (Sub), we have $\{\langle\, pc_3 \cup s_o, f_o, \alpha_o \,\rangle <: t'_o\} \subseteq \mathcal{C}$, and again by (Sub), $\{t_o <: t\} \subseteq \mathcal{C}$. So, by (*-Trans), $\{\langle\, pc_3 \cup s_o, f_o, \alpha_o \,\rangle <: t\} \subseteq \mathcal{C}$. According to uses of (Sub), we must have $pc_1 \subseteq pc_3$, hence by ($\mathcal{S}$-Union), $\{\langle\, pc_1 \cup s_o, f_o, \alpha_o \,\rangle <: t\} \subseteq \mathcal{C}$. So, we conclude that with (Oid) and (Sub), $\mathbf{TD}' \rhd o, H, \Gamma, pc_1, t, \mathcal{C}$.

4. (IfTrue-R)

   So $\varepsilon = \texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_1}\}\ \texttt{else}\ \{\overline{\texttt{s}_2}\}$.

   By (IfTrue-R), we have $\texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_1}\}\ \texttt{else}\ \{\overline{\texttt{s}_2}\}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \{\overline{\texttt{s}_1}\}, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$ and $\mathbf{TD}' = tdiftrue(\mathbf{TD}, \{\overline{\texttt{s}_1}\}, \mathcal{C}_\top)$, which entails the following, since $\mathbf{TD}$ is a valid derivation.

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_t \qquad \mathbf{TD}_1 \qquad \mathbf{TD}_2}{\Gamma, pc_2, \mathcal{H} \vdash \texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_1}\}\ \texttt{else}\ \{\overline{\texttt{s}_2}\} : t_i \backslash \mathcal{C}_i}\ \text{(If)}}{\Gamma, pc_1, \mathcal{H} \vdash \texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_1}\}\ \texttt{else}\ \{\overline{\texttt{s}_2}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   Where $\mathcal{C}_i = \mathcal{C}_t \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 <: t_i\} \cup \{\tau_2 <: t_i\}$ and $\mathcal{C}_i \cup \{t_i <: t\} \subseteq \mathcal{C}$.

$$\mathbf{TD}_1 = \dfrac{\dfrac{\cdots}{\Gamma, pc_3, \mathcal{H} \vdash \{\overline{\texttt{s}_1}\} : \tau_1 \backslash \mathcal{C}'_1}\ \text{(Block)}}{\Gamma, pc_2 \cup s_t, \mathcal{H} \vdash \{\overline{\texttt{s}_1}\} : t_1 \backslash \mathcal{C}_1}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\cdots}{\Gamma, pc_2 \cup s_t, \mathcal{H} \vdash \{\overline{\texttt{s}_2}\} : t_2 \backslash \mathcal{C}_2}\ \text{(Sub)}$$

$$\mathbf{TD}_t = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H} \vdash \texttt{True} : t_t \backslash \mathcal{C}_t}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{\cdots}{\Gamma, pc_3, \mathcal{H} \vdash \{\overline{\texttt{s}_1}\} : \tau_1 \backslash \mathcal{C}'_1}\ \text{(Block)}}{\Gamma, pc_1, \mathcal{H} \vdash \{\overline{\texttt{s}_1}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   Since $\mathcal{C}$ is closed, by transitivity rules, we have $\{\tau_1 <: t\} \subseteq \mathcal{C}$. And, by (Sub) rule of $\mathbf{TD}_1$, we know $\mathcal{C}'_1 \subseteq \mathcal{C}_1$, so $\mathcal{C}'_1 \subseteq \mathcal{C}$. By use of (Sub) rules, $pc_1 \subseteq pc_3$. We conclude with (Block) and (Sub), $\mathbf{TD}' \rhd \{\overline{\texttt{s}_1}\}, H, \Gamma, pc_1, t, \mathcal{C}$.

5. (IfFalse-R)

   Follows a similar argument to (IfTrue-R).

6. (Seq-R)

   So, $\varepsilon = ;\overline{\mathtt{s}}$.

   By (Seq-R), and $\mathbf{TD}' = tdseq(\mathbf{TD}, \overline{\mathtt{s}}, \mathcal{C}_\top)$, and since $\mathbf{TD}$ is a valid type derivation for ε, we have the following.

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_n \qquad \mathbf{TD}_s}{\Gamma, pc_2, \mathcal{H} \vdash ;\overline{\mathtt{s}} : t_s \backslash \mathcal{C}_n \cup \mathcal{C}_s}\ \text{(Seq)}}{\Gamma, pc_1, \mathcal{H} \vdash ;\overline{\mathtt{s}} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_s = \dfrac{\dfrac{\cdots}{\Gamma, pc_3, \mathcal{H} \vdash \overline{\mathtt{s}} : \tau_s \backslash \mathcal{C}'_s}\ \text{(Seq)}}{\Gamma, pc_2, \mathcal{H} \vdash \overline{\mathtt{s}} : t_s \backslash \mathcal{C}_s}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{\cdots}{\Gamma, pc_3, \mathcal{H} \vdash \overline{\mathtt{s}} : \tau_s \backslash \mathcal{C}'_s}\ \text{(Seq)}}{\Gamma, pc_1, \mathcal{H} \vdash \overline{\mathtt{s}} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   Since $\mathcal{C}$ is closed, we have the following. According to (Sub), we have $\{\tau_s <: t_s\} \subseteq \mathcal{C}$, and again by (Sub), $\{t_s <: t\} \subseteq \mathcal{C}$. So, by (*-Trans), $\{\tau_s <: t\} \subseteq \mathcal{C}$. According to uses of (Sub), we must have $pc_1 \subseteq pc_3$. So, we conclude that with (Seq) and (Sub), $\mathbf{TD}' \rhd \overline{\mathtt{s}}, H, \Gamma, pc_1, t, \mathcal{C}$.

7. (Return-R)

   So, $\varepsilon = \mathtt{return}\ v$.

   By (Return-R), and $\mathbf{TD}' = tdreturn(\mathbf{TD}, v, \mathcal{C}_\top)$, and since $\mathbf{TD}$ is a valid type derivation for ε, we have the following.

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_v}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{return}\ v : t_v \backslash \mathcal{C}_v}\ \text{(Return)}}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{return}\ v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   According to the definition of $tdreturn$, we now have two cases.

   (a) v is a constant.

   So we have the following.

$$\mathbf{TD}_v = \dfrac{\dfrac{}{\Gamma, pc_3, \mathcal{H} \vdash v : \langle\, pc_3, \emptyset, \texttt{int}\,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_2, \mathcal{H} \vdash v : t_v \backslash \mathcal{C}_v}\ \text{(Sub)}$$

and

$$\mathbf{TD}' = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H} \vdash v : \langle\, pc_1, \emptyset, \texttt{int}\,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_1, \mathcal{H} \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Since $\mathcal{C}$ is closed, we have the following. According to (Sub), we have $\{\langle\, pc_3, \emptyset, \texttt{int}\,\rangle <: t_v\} \subseteq \mathcal{C}$. By (Return) and (Sub), $\{t_v <: t\} \subseteq \mathcal{C}$, so by (*-Trans), $\{\langle\, pc_3, \emptyset, \texttt{int}\,\rangle <: t\} \subseteq \mathcal{C}$. According to uses of (Sub), we must have $pc_1 \subseteq pc_3$, hence by ($\mathcal{S}$-Union), $\{\langle\, pc_1, \emptyset, \texttt{int}\,\rangle <: t\} \subseteq \mathcal{C}$. So, we conclude that with (Const) and (Sub), $\mathbf{TD}' \rhd v, H, \Gamma, pc_1, t, \mathcal{C}$.

(b) v is an object identifier.

So we have the following.

$$\mathbf{TD}_v = \dfrac{\dfrac{\mathcal{H}(v) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_3, \mathcal{H} \vdash v : \langle\, pc_3 \cup s_o, f_o, \alpha_o\,\rangle \backslash \mathcal{C}_o}\ \text{(Oid)}}{\Gamma, pc_2, \mathcal{H} \vdash v : t_v \backslash \mathcal{C}_v}\ \text{(Sub)}$$

and

$$\mathbf{TD}' = \dfrac{\dfrac{\mathcal{H}(v) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_1, \mathcal{H} \vdash v : \langle\, pc_1 \cup s_o, f_o, \alpha_o\,\rangle \backslash \mathcal{C}_o}\ \text{(Oid)}}{\Gamma, pc_1, \mathcal{H} \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Since $\mathcal{C}$ is closed, we have the following. According to (Sub), we have $\{\langle\, pc_3 \cup s_o, f_o, \alpha_o\,\rangle <: t_v\} \subseteq \mathcal{C}$. By (Return) and (Sub), $\{t_v <: t\} \subseteq \mathcal{C}$, so by (*-Trans), $\{\langle\, pc_3 \cup s_o, f_o, \alpha_o\,\rangle <: t\} \subseteq \mathcal{C}$. According to uses of (Sub), we must have $pc_1 \subseteq pc_3$, hence by ($\mathcal{S}$-Union), $\{\langle\, pc_1 \cup s_o, f_o, \alpha_o\,\rangle <: t\} \subseteq \mathcal{C}$. So, we conclude that with (Oid) and (Sub), $\mathbf{TD}' \rhd v, H, \Gamma, pc_1, t, \mathcal{C}$.

8. (Block-R)

So, $\varepsilon = \{\overline{\texttt{s}}\}$

By (Block-R), and $\mathbf{TD}' = tdblock(\mathbf{TD}, \overline{\mathtt{s}}, \mathcal{C}_\top)$, and since $\mathbf{TD}$ is a valid type derivation for ε, we have the following.

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_s}{\Gamma, pc_2, \mathcal{H} \vdash \{\overline{\mathtt{s}}\} : t_s \backslash \mathcal{C}_s}\ \text{(Block)}}{\Gamma, pc_1, \mathcal{H} \vdash \{\overline{\mathtt{s}}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_s = \dfrac{\dfrac{\cdots}{\Gamma, pc_3, \mathcal{H} \vdash \overline{\mathtt{s}} : \tau_s \backslash \mathcal{C}'_s}\ \text{(Seq)}}{\Gamma, pc_2, \mathcal{H} \vdash \overline{\mathtt{s}} : t_s \backslash \mathcal{C}_s}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{\cdots}{\Gamma, pc_3, \mathcal{H} \vdash \overline{\mathtt{s}} : \tau_s \backslash \mathcal{C}'_s}\ \text{(Seq)}}{\Gamma, pc_1, \mathcal{H} \vdash \overline{\mathtt{s}} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Since $\mathcal{C}$ is closed, we have the following. According to (Sub), we have $\{\tau_s <: t_s\} \subseteq \mathcal{C}$, and again by (Sub), $\{t_s <: t\} \subseteq \mathcal{C}$. So, by (*-Trans), $\{\tau_s <: t\} \subseteq \mathcal{C}$. According to uses of (Sub), we must have $pc_1 \subseteq pc_3$. So, we conclude that with (Seq) and (Sub), $\mathbf{TD}' \rhd \overline{\mathtt{s}}, H, \Gamma, pc_1, t, \mathcal{C}$.

9. (Assign-R)

So, $\varepsilon = o.\mathtt{f} = v$.

By (Assign-R) and $tdassign(\mathbf{TD}, \mathcal{C}_\top) = \mathbf{TD}'$, and since $\mathbf{TD}$ is a valid type derivation for ε, we have the following.

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_o \qquad \mathbf{TD}_v}{\Gamma, pc_2, \mathcal{H} \vdash o.\mathtt{f} = v : \langle\, s'_o \cup s_v, \emptyset, \mathtt{void} \,\rangle \backslash \mathcal{C}'_o \cup \mathcal{C}_v \cup \{f'_o.\mathtt{f} <: \textbf{set}\ \langle\, s'_o \cup s_v, f_v, \alpha_v \,\rangle\}}\ \text{(F-Assign)}}{\Gamma, pc_1, \mathcal{H} \vdash o.\mathtt{f} = v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Where $\mathcal{C}'_o \cup \mathcal{C}_v \cup \{f'_o.\mathtt{f} <: \textbf{set}\ \langle\, s'_o \cup s_v, f_v, \alpha_v \,\rangle\} \cup \{\langle\, s'_o \cup s_v, \emptyset, \mathtt{void} \,\rangle <: t\} \subseteq \mathcal{C}$.

$$\mathbf{TD}_o = \dfrac{\dfrac{\mathcal{H}(o) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_3, \mathcal{H} \vdash o : \langle\, pc_3 \cup s_o, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o}\ \text{(Oid)}}{\Gamma, pc_2, \mathcal{H} \vdash o : t'_o \backslash \mathcal{C}'_o}\ \text{(Sub)}$$

Where $\mathcal{C}_o \cup \{\langle\, pc_3 \cup s_o, f_o, \alpha_o \,\rangle <: t'_o\} \subseteq \mathcal{C}'_o$.

$$\mathbf{TD}_v = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H} \vdash v : t_v \backslash \mathcal{C}_v}\ \text{(Sub)}$$

$\mathcal{H}' = \mathcal{H}[o \mapsto t_o \backslash \mathcal{C}''_o]$

where $\mathcal{C}''_o = \mathcal{C}_o \cup \mathcal{C}_v \cup \{f_o.\mathtt{f} <: \textbf{set}\ t_v\}]$

$$\mathbf{TD}' = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H}' \vdash\ ; : \langle\, pc_1, \emptyset, \mathtt{void}\,\rangle \backslash \emptyset}\ \text{(No-op)}}{\Gamma, pc_1, \mathcal{H}' \vdash\ ; : t \backslash \mathcal{C}}\ \text{(Sub)}$$

We first show the heap is well-typed.

By (Assign-R), we have $H' = H[o \mapsto \mathtt{C}, F']$, where $H(o) = \mathtt{C}, F$, and $F' = F[\mathtt{f} = v]$.

In order to show $\mathcal{H}' \vdash H'$, we show that the $\mathtt{f}$ field of o satisfies the (Heap) typing, as for all other oids and fields, the assumption that $\mathcal{H} \vdash H$ suffices.

Since by assumption $\mathcal{H} \vdash H$, we have $\{\mathtt{C} <: \alpha_o\} \subseteq \mathcal{C}_o$. Since $\mathcal{C}''_o = \mathcal{C}_o \cup \mathcal{C}_v \cup \{f_o.\mathtt{f} <: \textbf{set}\ t_v\}]$, we have $\{f_o.\mathtt{f} <: \textbf{set}\ t_v\} \cup \mathcal{C}_v \cup \{\mathtt{C} <: \alpha_o\} \subseteq \mathcal{C}''_o$, hence we conclude $\mathcal{H}' \vdash H'$. Furthermore, all typings via (Heap) on $\mathcal{H}'$ have canonical derivations, since the only change is to v, which has canonical derivation $\mathbf{TD}_v$.

We now show the resulting statement is well-typed.

Since $\mathcal{C}$ is closed, we have the following. From above, we have $\{\langle\, s'_o \cup s_v, \emptyset, \mathtt{void}\,\rangle <: t\} \subseteq \mathcal{C}$, so by ($\mathcal{S}$-Union), $\{\langle\, s'_o, \emptyset, \mathtt{void}\,\rangle <: t\} \subseteq \mathcal{C}$. Since $\mathcal{C}'_o \subseteq \mathcal{C}$ and $\{\langle\, pc_3 \cup s_o, f_o, \alpha_o\,\rangle <: t'_o\} \subseteq \mathcal{C}'_o$, we have $\{\langle\, pc_3 \cup s_o, f_o, \alpha_o\,\rangle <: t'_o\} \subseteq \mathcal{C}$, which by ($\mathcal{S}$-Union) gives $\{pc_3 <: s'_o\} \subseteq \mathcal{C}$. So, by ($\mathcal{S}$-Trans), $\{\langle\, pc_3, \emptyset, \mathtt{void}\,\rangle <: t\} \subseteq \mathcal{C}$. According to uses of (Sub), we must have $pc_1 \subseteq pc_3$, hence by ($\mathcal{S}$-Union), $\{\langle\, pc_1, \emptyset, \mathtt{void}\,\rangle <: t\} \subseteq \mathcal{C}$. Since $\mathcal{H}' \vdash H'$, we conclude that with (No-op) and (Sub), $\mathbf{TD}' \rhd ;, H', \Gamma, pc_1, t, \mathcal{C}$.

10. (New-R)

    So, $\varepsilon = \mathtt{new}\ \mathtt{C}(\overline{v})$.

By (New-R), and $tdnew(\mathbf{TD}, \mathtt{C}, H', \mathcal{C}_\top) = \mathbf{TD}'$, and since $\mathbf{TD}$ is a valid type derivation for ε, we have the following.

$$\mathbf{TD} = \dfrac{\dfrac{\overline{\mathbf{TD}_v} \qquad \overline{\mathbf{TD}_n}}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{new}\ \mathtt{C}(\overline{v}) : t_o \backslash \overline{\mathcal{C}_v} \cup \{\mathtt{C} <: \alpha_o\} \cup \{pc_2 <: s_o\} \cup \{\mathtt{C.K}(\overline{t_v}, t_o \xrightarrow{pc_2 \cup s_o} t_m)\} \cup \{\overline{f_o.\mathtt{f} <: \textbf{set}\ t_n}\} \cup \overline{\mathcal{C}_n}}\ \text{(New)}}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{new}\ \mathtt{C}(\overline{v}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Where $\{\mathtt{C.K}(\overline{t_v}, t_o \xrightarrow{pc_2 \cup s_o} t_m)\} \cup \overline{\mathcal{C}_v} \cup \{\overline{f_o.\mathtt{f} <: \textbf{set}\ t_n}\} \cup \overline{\mathcal{C}_n} \cup \{\mathtt{C} <: \alpha_o\} \cup \{pc_2 <: s_o\} \cup \{t_o <: t\} \subseteq \mathcal{C}$.

$$\overline{\mathbf{TD}_v} = \dfrac{\dots}{\Gamma, pc_2, \mathcal{H} \vdash \overline{v} : \overline{t_v} \backslash \overline{\mathcal{C}_v}}\ \text{(Sub)}$$

$$\overline{\mathbf{TD}_n} = \dfrac{\dfrac{}{\Gamma, pc_3, \mathcal{H} \vdash \overline{\mathtt{null}} : \overline{\langle\, pc_3, \emptyset, \alpha_{null}\,\rangle}}\ \text{(Null)}}{\Gamma, pc_2, \mathcal{H} \vdash \overline{\mathtt{null}} : \overline{t_n} \backslash \overline{\mathcal{C}_n}}\ \text{(Sub)}$$

Where $\overline{\langle\, pc_3, \emptyset, \alpha_{null}\,\rangle} <: \overline{t_n} \in \overline{\mathcal{C}_n}$.

$cnbody(\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{super}(\overline{\mathtt{e}}); \overline{\mathtt{s}})$ and

$\mathbf{TD}_m = TDT(\mathtt{C}, \mathtt{K})$

and $LT(\mathtt{C}, \mathtt{K}) = \forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}_s \cup \{t_s <: t_r\}$ and $\overline{t'_l} = \theta(\overline{t_l}, \mathtt{C}, \mathtt{K}, \overline{\alpha_v}, \alpha_a, \alpha_m)$ and t_s and $\mathcal{C}_s$ are the conclusion of $\mathbf{TD}_m$.

$\mathcal{H}' = \mathcal{H}[o \mapsto t_o \backslash \mathcal{C}_o]$ (same f_o that appears in $\mathbf{TD}$ above).

Where $\mathcal{C}_o = \{\overline{f_o.\mathtt{f} <: \textbf{set}\ t_n}\} \cup \overline{\mathcal{C}_n} \cup \{\mathtt{C} <: \alpha_o\}$.

$$\mathbf{TD}_o = \dfrac{\dfrac{\mathcal{H}'(o) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_2 \cup s_o, \mathcal{H}' \vdash o : \langle\, pc_2 \cup s_o, f_o, \alpha_o\,\rangle \backslash \mathcal{C}_o}\ \text{(Heap)}}{\Gamma, pc_2 \cup s_o, \mathcal{H}' \vdash o : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Let $\mathbf{TD}_s = tdsub(\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}};, \overline{v}, o, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m], \mathbf{TD}_m, tdheap(\overline{\mathbf{TD}_v}, \mathcal{H}'), \mathbf{TD}_o)$

(Note: to simplify the definition of $tdsub$, we are treating `this`, which maps to the object identifier as another argument.)

So,

$$\mathbf{TD}_{ret} = \dfrac{\dfrac{\mathbf{TD}_o}{\Gamma, pc_2 \cup s_o, \mathcal{H}' \vdash \texttt{return}\ o; : t \backslash \mathcal{C}}\ \text{(Return)}}{\Gamma, pc_2 \cup s_o, \mathcal{H}' \vdash \texttt{return}\ o; : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$\mathbf{TD}' =$

$$\dfrac{\dfrac{\mathbf{TD}_s \qquad \mathbf{TD}_{ret}}{\Gamma, pc_2 \cup s_o, \mathcal{H}' \vdash [\overline{\texttt{x}} \mapsto \overline{v}, \texttt{this} \mapsto o]\texttt{this.super}(\texttt{D}, \overline{\texttt{e}}); \overline{\texttt{s}}; \texttt{return}\ o; : t \backslash \mathcal{C}}\ \text{(Seq)}}{\Gamma, pc_1, \mathcal{H}' \vdash [\overline{\texttt{x}} \mapsto \overline{v}, \texttt{this} \mapsto o]\texttt{this.super}(\texttt{D}, \overline{\texttt{e}}); \overline{\texttt{s}}; \texttt{return}\ o; : t \backslash \mathcal{C}}\ \text{(Sub)}$$

and $\mathcal{C}'_s \subseteq \mathcal{C}_\top$ and $\mathcal{C} \subseteq \mathcal{C}_\top$.

Since $\{\texttt{C.K}(\overline{t_v}, t_o \xrightarrow{pc_2 \cup s_o} t_m)\} \cup \overline{\mathcal{C}_v} \cup \{\overline{f_o.\texttt{f} <: \textbf{set}\ t_n}\} \cup \overline{\mathcal{C}_n} \cup \{t_o <: t\} \subseteq \mathcal{C}$ and $\mathcal{C}$ is closed, we have the following.

We first show the new heap is well-typed.

Now, $\mathcal{H}' = \mathcal{H}[o \mapsto t_o \backslash \mathcal{C}_o]$, where $\mathcal{C}_o = \{\overline{f_o.\texttt{f} <: \textbf{set}\ t_n}\} \cup \overline{\mathcal{C}_n} \cup \{\texttt{C} <: \alpha_o\}$. By (New-R), $H' = H[o \mapsto \texttt{C}, F]$, where $F = \{\overline{\texttt{f} = \texttt{null}}\}$.

Since by assumption $\mathcal{H} \vdash H$, it is clear that $\forall o' \neq o \in dom(\mathcal{H}')$, we have $\mathcal{H}'(o') \vdash H'$, since the only change on the heap is adding o. Hence it remains to be shown that $\mathcal{H}' \vdash H'$ holds for o.

As given above, we have canonical derivations $\overline{\mathbf{TD}_n}$. Where $\overline{\langle\, pc_3, \emptyset, \alpha_{null}\,\rangle} <: \overline{t_n} \in \overline{\mathcal{C}_n}$.

By (Null), we have $\emptyset, \emptyset, \mathcal{H}' \vdash \overline{\texttt{null}} : \overline{\langle\, \emptyset, \emptyset, \alpha_{null}\,\rangle} \backslash \emptyset$. Since $\overline{\langle\, pc_3, \emptyset, \alpha_{null}\,\rangle} <: \overline{t_n} \in \overline{\mathcal{C}_n}$, by ($\mathcal{S}$-Union), we have $\overline{\langle\, \emptyset, \emptyset, \alpha_{null}\,\rangle} <: \overline{t_n} \in \overline{\mathcal{C}_n}$, so by (Sub), $\emptyset, \emptyset, \mathcal{H}' \vdash \overline{\texttt{null}} : \overline{t_n} \backslash \overline{\mathcal{C}_n}$, which are canonical derivations.

Since $\mathcal{C}_o = \{\overline{f_o.\texttt{f} <: \textbf{set}\ t_n}\} \cup \overline{\mathcal{C}_n} \cup \{\texttt{C} <: \alpha_o\}$, we conclude by (Heap') that $\mathcal{H}' \vdash H'$; and the heap typings are canonical.

We now showing the resulting statements are well-typed.

Now, $\mathcal{C}_o = \{\overline{f_o.\texttt{f} <: \textbf{set}\ t_n}\} \cup \overline{\mathcal{C}_n}$, hence $\mathcal{C}_o \subseteq \mathcal{C}$; and $\{t_o <: t\} \cup \{pc_2 <: s_o\} \subseteq \mathcal{C}$, so $\mathbf{TD}_o \rhd o, H', \Gamma, pc_2 \cup s_o, t, \mathcal{C}$, so by (Return) and (Sub), $\mathbf{TD}_{ret} \rhd \texttt{return}\ o, H', \Gamma, pc_2, t, \mathcal{C}$.

By (*Method*), we know $[\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m](\mathcal{C}_s \cup \{t_s <: t_r\}) \subseteq \mathcal{C}$.

Now, since $\mathcal{H}'$ extends $\mathcal{H}$ and $\mathcal{H}' \vdash H'$, by Lemma 4.11, $tdheap(\overline{\mathbf{TD}_v}, \mathcal{H}') \rhd \overline{v}, H', \Gamma, pc_2, \overline{t_v}, \overline{\mathcal{C}_v}$ are valid type derivations.

According to the Constructor Typing rule, and the description of $LT(\texttt{C}, \texttt{K})$ above, we have $\overline{t_l} = FTV(\mathcal{C}_s \cup \{t_s <: t_r\}) - FTV(\overline{t_x}, t_t, s_p, t_r)$. Since t_s and $\mathcal{C}_s$ are contained here, we have $FTV(t_r \backslash \mathcal{C}_s) \subseteq \overline{t_l}, \overline{t_x}, t_t, s_p, t_r$.

Now, according to Substitution Lemma 4.12,

$\mathbf{TD}_s \rhd [\overline{\texttt{x}} \mapsto \overline{v}, \texttt{this} \mapsto o]\texttt{this.super}(\texttt{D}, \overline{\texttt{e}}); \overline{\texttt{s}};, H', \Gamma, pc_2 \cup s_o, t'_s, \mathcal{C}'_s$, where

$t'_s = Closure(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m]t_s$ and $\mathcal{C}'_s = Closure(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m](\mathcal{C}_s) \cup \overline{\mathcal{C}_v}$.

Now, since we already showed that $[\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m](\mathcal{C}_s) \subseteq \mathcal{C}$, and since $\overline{\mathcal{C}_v} \subseteq \mathcal{C}$, and $\mathcal{C}$ is closed, this means $\mathcal{C}'_s \subseteq \mathcal{C}$.

Since the resulting types aren't changing in $\mathbf{TD}'$, and $\mathbf{TD}_s$ and $\mathbf{TD}_{ret}$ are valid typings, and $pc_1 \subseteq pc_2 \cup s_o$, by (Seq) and (Sub), we conclude with

$\mathbf{TD}' \rhd [\overline{\texttt{x}} \mapsto \overline{v}, \texttt{this} \mapsto o]\texttt{this.super}(\texttt{D}, \overline{\texttt{e}}); \overline{\texttt{s}}; \texttt{return}\ o;, H', \Gamma, pc_1, t, \mathcal{C}$.

11. (Invoke-R)

So, $\varepsilon = o.\texttt{m}(\overline{v})$.

By (Invoke-R) and $tdinvoke(\mathbf{TD}, \mathtt{C}, \mathtt{m}, \mathcal{C}_\top) = \mathbf{TD}'$, since $\mathbf{TD}$ is a valid type derivation for ε we have the following.

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_o \qquad \overline{\mathbf{TD}_v}}{\Gamma, pc_2, \mathcal{H} \vdash o.\mathtt{m}(\overline{v}) : t_m \backslash \{\alpha_o.\mathtt{m}(\overline{t_v}, t_o \xrightarrow{pc_2 \cup s_o} t_m)\} \cup \mathcal{C}_o \cup \overline{\mathcal{C}_v}}\ \text{(Invoke)}}{\Gamma, pc_1, \mathcal{H} \vdash o.\mathtt{m}(\overline{v}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Where $\{\alpha_o.\mathtt{m}(\overline{t_v}, t_o \xrightarrow{pc_2 \cup s_o} t_m)\} \cup \{t_m <: t\} \cup \mathcal{C}_o \cup \overline{\mathcal{C}_v} \subseteq \mathcal{C}$.

$$\mathbf{TD}_o = \dfrac{\dfrac{\mathcal{H}(o) = t'_o \backslash \mathcal{C}'_o}{\Gamma, pc_3, \mathcal{H} \vdash o : \langle\, pc_3 \cup s'_o, f'_o, \alpha'_o \,\rangle \backslash \mathcal{C}'_o}\ \text{(Heap)}}{\Gamma, pc_2, \mathcal{H} \vdash o : t_o \backslash \mathcal{C}_o}\ \text{(Sub)}$$

Where $\mathcal{C}'_o \cup \{\langle\, pc_3 \cup s'_o, f'_o, \alpha'_o \,\rangle <: t_o\} \subseteq \mathcal{C}_o$.

$$\overline{\mathbf{TD}_v} = \dfrac{\dots}{\Gamma, pc_2, \mathcal{H} \vdash \overline{v} : \overline{t_v} \backslash \overline{\mathcal{C}_v}}\ \text{(Sub)}$$

$mbody(\mathtt{C}, \mathtt{m}) = (\overline{\mathtt{x}}, \overline{\mathtt{s}})$ and

$$\mathbf{TD}_m = TDT(\mathtt{C}, \mathtt{m})$$

$$\mathbf{TD}_m = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \mathbf{TD}_2}{[\overline{\mathtt{x}} \mapsto \overline{t_x}, \mathtt{this} \mapsto t_t], s_p, \emptyset \vdash \overline{\mathtt{s}} : t_2 \backslash \mathcal{C}_1 \cup \mathcal{C}_2}\ \text{(Seq)}}{[\overline{\mathtt{x}} \mapsto \overline{t_x}, \mathtt{this} \mapsto t_t], s_p, \emptyset \vdash \overline{\mathtt{s}} : t_s \backslash \mathcal{C}_s}\ \text{(Sub)}$$

$$\mathbf{TD}_1 = \dfrac{\dots}{[\overline{\mathtt{x}} \mapsto \overline{t_x}, \mathtt{this} \mapsto t_t], s_p, \emptyset \vdash \mathtt{s} : t_1 \backslash \mathcal{C}_1}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\dots}{[\overline{\mathtt{x}} \mapsto \overline{t_x}, \mathtt{this} \mapsto t_t], s_p, \emptyset \vdash \overline{\mathtt{s}'} : t_2 \backslash \mathcal{C}_2}\ \text{(Sub)}$$

$\overline{\mathtt{s}} = \mathtt{s}; \overline{\mathtt{s}'}$

and $LT(\mathtt{C}, \mathtt{m}) = \forall \overline{t_l}.\overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}_s \cup \{t_s <: t_r\}$ and $\overline{t'_l} = \theta(\overline{t_l}, \mathtt{C}, \mathtt{m}, \overline{\alpha_v}, \alpha_o, \alpha_m)$

and $\mathbf{TD}_s = tdsub(\overline{\mathtt{s}}, \overline{v}, o, H, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m],$

$$\mathbf{TD}_m, \overline{\mathbf{TD}_v}, \mathbf{TD}_o)$$

$$\mathbf{TD}_s = \dfrac{\dfrac{\mathbf{TD}'_1 \qquad \mathbf{TD}'_2}{\Gamma, pc_2 \cup s_o, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}} : t'_2 \backslash \mathcal{C}'_1 \cup \mathcal{C}'_2}\ \text{(Seq)}}{\Gamma, pc_2 \cup s_o, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}} : t'_s \backslash \mathcal{C}'_s}\ \text{(Sub)}$$

$$\mathbf{TD}'_1 = \frac{\dots}{\Gamma, pc_2 \cup s_o, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\mathtt{s} : t'_1 \backslash \mathcal{C}'_1} \text{ (Sub)}$$

$$\mathbf{TD}'_2 = \frac{\dots}{\Gamma, pc_2 \cup s_o, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}'} : t'_2 \backslash \mathcal{C}'_2} \text{ (Sub)}$$

$$\mathbf{TD}' = \frac{\dfrac{\mathbf{TD}'_1 \qquad \mathbf{TD}'_2}{\Gamma, pc_2 \cup s_o, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}} : t'_2 \backslash \mathcal{C}'_1 \cup \mathcal{C}'_2} \text{ (Seq)}}{\Gamma, pc_1, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}} : t \backslash \mathcal{C}} \text{ (Sub)}$$

and $\mathcal{C}'_s \subseteq \mathcal{C}_\top$ and $\mathcal{C} \subseteq \mathcal{C}_\top$.

Since $\mathcal{H} \vdash H$, and from the typing of $\mathbf{TD}_o$, we have $\mathcal{H}(o) = t'_o \backslash \mathcal{C}'_o$. so by (Heap), we have $\{\mathtt{C} <: \alpha'_o\} \in \mathcal{C}'_o$. From above, we have $\mathcal{C}'_o \cup \{\langle\, pc_3 \cup s'_o, f'_o, \alpha'_o \,\rangle <: t_o\} \subseteq \mathcal{C}_o$, and since $\mathcal{C}_o \subseteq \mathcal{C}$, and $\mathcal{C}$ is closed, by ($\mathcal{A}$-Trans), we have $\{\mathtt{C} <: \alpha_o\} \in \mathcal{C}$.

Now, since $\{\mathtt{C} <: \alpha_o\} \in \mathcal{C}$ and $\{\alpha_o.\mathtt{m}(\overline{t_v}, t_o \xrightarrow{pc_2 \cup s_o} t_m)\} \cup \{t_m <: t\} \cup \mathcal{C}_o \cup \overline{\mathcal{C}_v} \subseteq \mathcal{C}$, and $\mathcal{C}$ is closed, by closure rule (*Method*), we know $[\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m](\mathcal{C}_s \cup \{t_s <: t_r\}) \subseteq \mathcal{C}$.

According to the Method Typing rule, and the description of $LT(\mathtt{C}, \mathtt{m})$ above, we have $\overline{t_l} = FTV(\mathcal{C}_s \cup \{t_s <: t_r\}) - FTV(\overline{t_x}, t_t, s_p, t_r)$. Since t_s and $\mathcal{C}_s$ are contained here, we have $FTV(t_r \backslash \mathcal{C}_s) \subseteq \overline{t_l}, \overline{t_x}, t_t, s_p, t_r$.

Now, according to Substitution Lemma 4.12,

$\mathbf{TD}_s \rhd [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}}, H, \Gamma, pc_2 \cup s_o, t'_s, \mathcal{C}'_s$, where $t'_s = [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m]t_s$ and $\mathcal{C}'_s = \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m](\mathcal{C}_s) \cup \overline{\mathcal{C}_v})$.

Similarly, by the definition of *tdsub*, we have

$$\mathbf{TD}'_1 = \mathit{tdsub}(\mathtt{s}, \overline{v}, o, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m], \mathbf{TD}_1, \overline{\mathbf{TD}_v}, \mathbf{TD}_o)$$

and

$$\mathbf{TD}'_2 = tdsub(\overline{\mathtt{s}'}, \overline{v}, o, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m],$$
$$\mathbf{TD}_2, \overline{\mathbf{TD}_v}, \mathbf{TD}_o)$$

So, according to Substitution Lemma 4.12, $\mathbf{TD}'_1 \rhd [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\mathtt{s}, H, \Gamma, pc_2 \cup s_o, t'_1, \mathcal{C}'_1$, where $t'_1 = [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m]t_1$ where $\mathcal{C}'_1 = Closure(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m](\mathcal{C}_1) \cup \overline{\mathcal{C}_v})$ and $\mathbf{TD}'_2 \rhd [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}'}, H, \Gamma, pc_2 \cup s_o, t'_2, \mathcal{C}'_2$, where $t'_2 = [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m]t_2$ and $\mathcal{C}'_2 = Closure(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m](\mathcal{C}_2) \cup \overline{\mathcal{C}_v})$

Now, since we already showed that $[\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m](\mathcal{C}_s) \subseteq \mathcal{C}$, and since $\overline{\mathcal{C}_v} \subseteq \mathcal{C}$, and $\mathcal{C}$ is closed, this means $\mathcal{C}'_s \subseteq \mathcal{C}$. Since the use of (Sub) in $\mathbf{TD}_s$ requires $\mathcal{C}'_1 \cup \mathcal{C}'_2 \subseteq \mathcal{C}'_s$, we also have $\mathcal{C}'_1 \subseteq \mathcal{C}$ and $\mathcal{C}'_2 \subseteq \mathcal{C}$.

Now, observing the derivation $\mathbf{TD}_m$, by (Sub), $\mathcal{C}_2 \subseteq \mathcal{C}_s$, and $\{t_2 <: t_s\} \subseteq \mathcal{C}_s$. Hence, $[\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m](\{t_2 <: t_s\}) \subseteq \mathcal{C}'_s$, which is $\{t'_2 <: t'_s\} \subseteq \mathcal{C}'_s$.

From above, we know $[\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m]\{t_s <: t_r\} \subseteq \mathcal{C}$. Which is $\{t'_s <: t_m\} \subseteq \mathcal{C}$.

Since we have $\{t_m <: t\} \subseteq \mathcal{C}$, by transitivity closure rules, $\{t'_s <: t\} \subseteq \mathcal{C}$.

Since $\mathcal{C}'_s \subseteq \mathcal{C}$, we have $\{t'_2 <: t'_s\} \subseteq \mathcal{C}$. So, by transitivity closure rules with the previous constraint, we have $\{t'_2 <: t\} \subseteq \mathcal{C}$.

Now, since $\mathbf{TD}'_1$ and $\mathbf{TD}'_2$ are valid type derivations, and $\{t'_2 <: t\} \subseteq \mathcal{C}$, and $\mathcal{C}'_1 \subseteq \mathcal{C}$ and $\mathcal{C}'_2 \subseteq \mathcal{C}$, and $pc_1 \subseteq pc_2 \cup s_o$. By using (Seq) and (Sub), we conclude that $\mathbf{TD}' \rhd [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}}, H, \Gamma, pc_1, t, \mathcal{C}$.

12. (Super-R)

    Follows a similar argument to (Invoke-R) and (New-R).

13. (Input-R)

    So, $\varepsilon = \mathtt{read}_{\mathtt{L}}(\mathtt{fd})$.

    By (Input-R), we have the following.

    $\mathtt{read}_{\mathtt{L}}(\mathtt{fd}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathtt{c}, \mathbf{TD}', \mathcal{C}_\top, H, \iota', \omega$ and $\mathbf{TD}' = tdinput(\mathbf{TD}, \mathtt{c}, \mathcal{C}_\top)$, which entails the following, since $\mathbf{TD}$ is a valid derivation.

$$\mathbf{TD} = \dfrac{\dfrac{\dfrac{}{\Gamma, pc_3, \mathcal{H} \vdash \mathtt{fd} : \langle\, pc_3, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{fd} : t_f \backslash \mathcal{C}_f}\ \text{(Sub)}}{\dfrac{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{fd}) : \langle\, s_f \cup \mathtt{L}, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_f \cup SC(\mathtt{L}, s_f)}\ \text{(Input)}}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{fd}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

    Where $\{\langle\, pc_3, \emptyset, \mathtt{int} \,\rangle <: t_f\} \subseteq \mathcal{C}_f$ and $\{\langle\, s_f \cup \mathtt{L}, \emptyset, \mathtt{int} \,\rangle <: t\} \cup \mathcal{C}_f \cup SC(\mathtt{L}, s_f) \subseteq \mathcal{C}$.

$$\mathbf{TD}' = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{c} : \langle\, pc_1, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{c} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

    Now, since $\mathcal{C}$ is closed, we have the following. Since $\{\langle\, pc_3, \emptyset, \mathtt{int} \,\rangle <: t_f\} \subseteq \mathcal{C}_f$ and $\{\langle\, s_f \cup \mathtt{L}, \emptyset, \mathtt{int} \,\rangle <: t\} \cup \mathcal{C}_f \cup SC(\mathtt{L}, s_f) \subseteq \mathcal{C}$, by ($\mathcal{S}$-Trans), $\{\langle\, pc_3 \cup \mathtt{L}, \emptyset, \mathtt{int} \,\rangle <: t\} \subseteq \mathcal{C}$, and by ($\mathcal{S}$-Union), $\{\langle\, pc_3, \emptyset, \mathtt{int} \,\rangle <: t\} \subseteq \mathcal{C}$. According to uses of (Sub), we must have $pc_1 \subseteq pc_3$, hence by ($\mathcal{S}$-Union), $\{\langle\, pc_1, \emptyset, \mathtt{int} \,\rangle <: t\} \subseteq \mathcal{C}$. So, we conclude that with (Const) and (Sub), $\mathbf{TD}' \rhd \mathtt{c}, H, \Gamma, pc_1, t, \mathcal{C}$.

14. (Output-R)

So, $\varepsilon = \mathtt{write_L(c, fd)}$.

By (Output-R), we have the following.

$\mathtt{write_L(c, fd)}, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow ;, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega'$ and

$\mathbf{TD}' = tdoutput(\mathbf{TD}, \mathcal{C}_\top)$, which entails the following, since $\mathbf{TD}$ is a valid derivation.

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_c \qquad \mathbf{TD}_f}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{write_L(c, fd)} : \langle\, s_c \cup s_f, \emptyset, \mathtt{void}\,\rangle \backslash \mathcal{C}'}\,\text{(Output)}}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{write_L(c, fd)} : t \backslash \mathcal{C}}\,\text{(Sub)}$$

$$\mathbf{TD}_c = \dfrac{\dfrac{}{\Gamma, pc_3, \mathcal{H} \vdash \mathtt{c} : \langle\, pc_3, \emptyset, \mathtt{int}\,\rangle \backslash \emptyset}\,\text{(Const)}}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{c} : t_c \backslash \mathcal{C}_c}\,\text{(Sub)}$$

$$\mathbf{TD}_f = \dfrac{\dfrac{}{\Gamma, pc_4, \mathcal{H} \vdash \mathtt{fd} : \langle\, pc_4, \emptyset, \mathtt{int}\,\rangle \backslash \emptyset}\,\text{(Const)}}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{fd} : t_f \backslash \mathcal{C}_f}\,\text{(Sub)}$$

Where $\{\langle\, pc_3, \emptyset, \mathtt{int}\,\rangle <: t_c\} \subseteq \mathcal{C}_c$, and $\{\langle\, pc_4, \emptyset, \mathtt{int}\,\rangle <: t_f\} \subseteq \mathcal{C}_f$, and $\mathcal{C}' = \mathcal{C}_c \cup \mathcal{C}_f \cup SC(\mathtt{L}, s_c \cup s_f)$, and $\mathcal{C}' \cup \{\langle\, s_c \cup s_f, \emptyset, \mathtt{void}\,\rangle <: t\} \subseteq \mathcal{C}$.

$$\mathbf{TD}' = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H} \vdash ; : \langle\, pc_1, \emptyset, \mathtt{void}\,\rangle \backslash \emptyset}\,\text{(No-op)}}{\Gamma, pc_1, \mathcal{H} \vdash ; : t \backslash \mathcal{C}}\,\text{(Sub)}$$

Now, since $\mathcal{C}$ is closed, we have the following. Since $\{\langle\, pc_3, \emptyset, \mathtt{int}\,\rangle <: t_c\} \subseteq \mathcal{C}_c$, and $\mathcal{C}' = \mathcal{C}_c \cup \mathcal{C}_f \cup SC(\mathtt{L}, s_c \cup s_f)$, and $\mathcal{C}' \cup \{\langle\, s_c \cup s_f, \emptyset, \mathtt{void}\,\rangle <: t\} \subseteq \mathcal{C}$, by ($\mathcal{S}$-Union) and ($\mathcal{S}$-Trans), we have $\{\langle\, pc_3, \emptyset, \mathtt{void}\,\rangle <: t\} \subseteq \mathcal{C}$. According to uses of (Sub), we must have $pc_1 \subseteq pc_3$, hence by ($\mathcal{S}$-Union), $\{\langle\, pc_1, \emptyset, \mathtt{void}\,\rangle <: t\} \subseteq \mathcal{C}$. So, we conclude that with (No-op) and (Sub), $\mathbf{TD}' \rhd ;, H, \Gamma, pc_1, t, \mathcal{C}$.

15. (Op-RC)

So, $\varepsilon = \mathtt{e_1} \oplus \mathtt{e_2}$.

By (Op-RC), we have $\mathtt{e}_1 \oplus \mathtt{e}_2, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathtt{e}_1' \oplus \mathtt{e}_2, \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$, and

$\mathtt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathtt{e}_1', \mathbf{TD}_1', \mathcal{C}_\top, H', \iota', \omega'$, and $\mathbf{TD}_1 = uconop(\mathbf{TD})$ and

$\mathbf{TD}' = dconop(\mathbf{TD}, \mathbf{TD}_1', \mathcal{C}_\top)$.

Where by $uconop(\mathbf{TD})$, we have

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \mathbf{TD}_2}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{e}_1 \oplus \mathtt{e}_2 : \langle\, s_1 \cup s_2, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_1 \cup \mathcal{C}_2}\ \text{(Op)}}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{e}_1 \oplus \mathtt{e}_2 : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_1 = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{e}_1 : t_1 \backslash \mathcal{C}_1}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{e}_2 : t_2 \backslash \mathcal{C}_2}\ \text{(Sub)}$$

By $dconop(\mathbf{TD}, \mathbf{TD}_1', \mathcal{C}_\top)$, we have

$$\mathbf{TD}_1' = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}' \vdash \mathtt{e}_1' : t_1 \backslash \mathcal{C}_1}\ \text{(Sub)}$$

$$\mathbf{TD}_2' = tdheap(\mathbf{TD}_2, \mathcal{H}')$$

$$\mathbf{TD}_2' = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}' \vdash \mathtt{e}_2 : t_2 \backslash \mathcal{C}_2}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}_1' \qquad \mathbf{TD}_2'}{\Gamma, pc_2, \mathcal{H}' \vdash \mathtt{e}_1' \oplus \mathtt{e}_2 : \langle\, s_1 \cup s_2, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_1 \cup \mathcal{C}_2}\ \text{(Op)}}{\Gamma, pc_1, \mathcal{H}' \vdash \mathtt{e}_1' \oplus \mathtt{e}_2 : t \backslash \mathcal{C}}\ \text{(Sub)}$$

By induction, $\mathbf{TD}_1' \rhd \mathtt{e}_1', H', \Gamma, pc_2, t_1, \mathcal{C}_1$.

Using Lemma 4.11 we have $\mathbf{TD}_2' \rhd \mathtt{e}_2, H', \Gamma, pc_2, t_2, \mathcal{C}_2$. Concluding with (Op) and (Sub), $\mathbf{TD}' \rhd \mathtt{e}_1' \oplus \mathtt{e}_2, H', \Gamma, pc_1, t, \mathcal{C}$.

16. Remaining (*-RC) cases follow a similar argument to (Op-RC).

□

Subject Reduction Lemma 4.13 now directly produces the following soundness result.

Theorem 4.14 (Soundness)

If $\mathbf{TD} \rhd \varepsilon, \emptyset, \emptyset, \emptyset, t, \mathcal{C}$*, then* $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, \emptyset, \iota, \omega \not\rightsquigarrow^* \mathit{CkFail}, \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$.

Proof. By contradiction.

Suppose $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, \iota, \omega \rightsquigarrow^* \mathit{CkFail}, \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$.

Let $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, \iota, \omega \rightsquigarrow^* \varepsilon', \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ be the sequence of reductions immediately before the check failure occurs. By induction on the context derivation tree of the failure step; considering this step is a check failure, we have the following cases.

1. $\varepsilon' = \mathtt{read}_{\mathtt{L}}(\mathtt{fd})$

   Now, by Subject Reduction Lemma 4.13, $\mathbf{TD}'$ is a valid typing for ε', and concludes with the same type and constraint as $\mathbf{TD}$. Thus, we have the following.

   $$\mathbf{TD}' = \dfrac{\dfrac{\dfrac{}{\Gamma, pc, H \vdash \mathtt{fd} : \langle\, pc, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\,(\text{Const})}{\Gamma, pc, H \vdash \mathtt{fd} : t_f \backslash \{\langle\, pc, \emptyset, \mathtt{int} \,\rangle <: t_f\}}\,(\text{Sub})}{\dfrac{\Gamma, pc, H \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{fd}) : \langle\, s_f \cup \mathtt{L}, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_f \cup SC(\mathtt{L}, s_f)}{\Gamma, pc, H \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{fd}) : t \backslash \mathcal{C}}\,(\text{Sub})}\,(\text{Input})$$

   Now, according to (InFail-R), we have $S_f = \mathit{conread}(\mathbf{TD}', \mathcal{C}_\top)$ and $S_f \not\subseteq \mathtt{L}$. Hence, by definition $S_f = \mathit{Con}(s_f, \mathcal{C}_\top)$.

   Now, since $S_f \not\subseteq \mathtt{L}$, there exists some $\mathtt{l} \in S_f$ such that $\mathtt{l} \notin \mathtt{L}$. By Definition 4.2, $\mathtt{l} <: s_f \in \mathcal{C}_\top$.

   Now, $\mathcal{C} \subseteq \mathcal{C}_\top$ and $\mathcal{C}_\top$ is closed, hence by closure rule (SC-Trans), we have $SC(\mathtt{L}, \mathtt{l}) \in \mathcal{C}_\top$. Since by assumption $\mathcal{C}_\top$ is consistent, by Definition 3.2 we have $\mathtt{l} \in \mathtt{L}$. However, we just showed that $\mathtt{l} \notin \mathtt{L}$, a contradiction. Therefore, this reduction step cannot have occurred.

2. $\varepsilon' = \mathtt{write}_{\mathtt{L}}(\mathtt{c}, \mathtt{fd})$ follows in a similar manner to the previous case.

3. $\varepsilon = \mathtt{e}_1 \oplus \mathtt{e}_2$.

So, by (Op-Err), $\mathtt{e}_1 \oplus \mathtt{e}_2, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{CkFail}, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$, where the context reduction is $\mathtt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$, and $\mathbf{TD}_1 = \mathit{ucon}(\mathbf{TD})$. However, by induction, we have $\mathtt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \not\rightsquigarrow \mathit{Err}, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega$, so this cannot have occured, and hence $\mathtt{e}_1 \oplus \mathtt{e}_2, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega \not\rightsquigarrow \mathit{CkFail}, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$.

4. The remaining (*-Err) cases follow a similar inductive reasoning to the previous case.

□

4.5 Noninterference

As stated previously, we prove noninterference by showing that two executions that differ only in high inputs are low-equivalent. This requires a definition of a bisimulation relation on values, expressions and type derivations, heaps, and streams, which specifies that all that is *low* as defined by `Low` (the observational level of a low-observer) and according to the type derivation must be equivalent. We subsequently define low and high reduction steps, based again on `Low` and the type derivation, such that each execution step is either high or low. We give these definitions first, as they are heavily used in the succeeding proofs.

Definition 4.15 formally defines the bisimulation relation. Values are bisimilar if they are both low, with the same set of security labels and the same value, or are both high. Bisimulation of expressions and type derivations is defined using a ("maximal") constraint set in order to determine the security level of values. The bisimulation is then defined inductively on the structure of ε, such that the structure of ε and ε' is the same, and all type variables in the respective type derivations $\mathbf{TD}$ and $\mathbf{TD}'$ are the same, and any values are bisimilar. This strongly correlates the type derivations of $\mathbf{TD}$ and $\mathbf{TD}'$ since the constraint sets $\mathcal{C}_\top$ and $\mathcal{C}'_\top$ must be the same; thus, anything that is considered

high based on one type derivation will be considered high in the other. Bisimulation of heaps establishes that any heap location that is low in *either* heap must exist in the other heap at the same location, with the same type variable, and furthermore that all of the fields of these heaps are also bisimilar. Streams are bisimilar according to their security level in relation to `Low`; any streams that are a subset of `Low` (and thus visible to the low observer) must be equivalent for bisimilarity to hold.

Definition 4.15 (Bisimulation Relation)

1. *(Values)* $S, v \simeq_{\texttt{Low}} S', v'$ *iff either*

   (a) $S \subseteq \texttt{Low}$, $S' \subseteq \texttt{Low}$, $S = S'$, *and* $v = v'$; *or*

   (b) $S \not\subseteq \texttt{Low}$ *and* $S' \not\subseteq \texttt{Low}$

2. *(Expression/Statement and* $\mathbf{TD}$*)* $\varepsilon, \mathbf{TD}, \mathcal{C}_\top \simeq_{\texttt{Low}} \varepsilon', \mathbf{TD}', \mathcal{C}'_\top$ *iff* $\mathcal{C}_\top = \mathcal{C}'_\top$ *and either*

   (a) $\varepsilon = v$, $\varepsilon' = v'$, *and*

   $$\mathbf{TD} = \dfrac{\cdots}{\Gamma, pc, \mathcal{H} \vdash v : t \backslash \mathcal{C}}\ (\text{Sub})$$
   $$\mathbf{TD}' = \dfrac{\cdots}{\Gamma', pc', \mathcal{H}' \vdash v' : t' \backslash \mathcal{C}'}\ (\text{Sub})$$
   $t = t'$, *and* $Con(s, \mathcal{C}_\top), v \simeq_{\texttt{Low}} Con(s', \mathcal{C}'_\top), v'$; *or*

   (b) $\varepsilon = \texttt{e.f}$, $\varepsilon' = \texttt{e}'\texttt{.f}$, *and*

   $$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_o}{\Gamma, pc_2, \mathcal{H} \vdash \texttt{e.f} : t_1 \backslash \mathcal{C}_1}\ (\text{Field})}{\Gamma, pc_1, \mathcal{H} \vdash \texttt{e.f} : t \backslash \mathcal{C}}\ (\text{Sub})$$
   $$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}'_o}{\Gamma', pc'_2, \mathcal{H}' \vdash \texttt{e}'\texttt{.f} : t'_1 \backslash \mathcal{C}'_1}\ (\text{Field})}{\Gamma', pc'_1, \mathcal{H}' \vdash \texttt{e}'\texttt{.f} : t' \backslash \mathcal{C}'}\ (\text{Sub})$$
   $\texttt{e}, \mathbf{TD}_o, \mathcal{C}_\top \simeq_{\texttt{Low}} \texttt{e}', \mathbf{TD}'_o, \mathcal{C}'_\top$, *and , and* $t_1 = t'_1$, *and* $t = t'$; *or*

   (c) $\varepsilon = \texttt{e}_1 \oplus \texttt{e}_2$, $\varepsilon' = \texttt{e}'_1 \oplus \texttt{e}'_2$ *and*

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \mathbf{TD}_2}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{e}_1 \oplus \mathtt{e}_2 : \tau_o \backslash \mathcal{C}_o}\ (\text{Op})}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{e}_1 \oplus \mathtt{e}_2 : t \backslash \mathcal{C}}\ (\text{Sub})$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}'_1 \qquad \mathbf{TD}'_2}{\Gamma', pc'_2, \mathcal{H}' \vdash \mathtt{e}'_1 \oplus \mathtt{e}'_2 : \tau'_o \backslash \mathcal{C}'_o}\ (\text{Op})}{\Gamma', pc'_1, \mathcal{H}' \vdash \mathtt{e}'_1 \oplus \mathtt{e}'_2 : t' \backslash \mathcal{C}'}\ (\text{Sub})$$

$\mathtt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathtt{e}'_1, \mathbf{TD}'_1, \mathcal{C}'_\top$, *and* $\mathtt{e}_2, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathtt{e}'_2, \mathbf{TD}'_2, \mathcal{C}'_\top$, *and* $t = t'$*; or*

(d) $\varepsilon = (\mathtt{C})\ \mathtt{e}_1$, $\varepsilon' = (\mathtt{C})\ \mathtt{e}'_1$

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_o}{\Gamma, pc_2, \mathcal{H} \vdash (\mathtt{C})\ \mathtt{e}_1 : t_o \backslash \mathcal{C}_o}\ (\text{Cast})}{\Gamma, pc_1, \mathcal{H} \vdash (\mathtt{C})\ \mathtt{e}_1 : t \backslash \mathcal{C}}\ (\text{Sub})$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}'_o}{\Gamma', pc'_2, \mathcal{H}' \vdash (\mathtt{C})\ \mathtt{e}'_1 : t'_o \backslash \mathcal{C}'_o}\ (\text{Cast})}{\Gamma', pc'_1, \mathcal{H}' \vdash (\mathtt{C})\ \mathtt{e}'_1 : t' \backslash \mathcal{C}'}\ (\text{Sub})$$

$\mathtt{e}_1, \mathbf{TD}_o, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathtt{e}'_1, \mathbf{TD}'_o, \mathcal{C}'_\top$, *and* $t = t'$*; or*

(e) $\varepsilon = \mathtt{if}\ (\mathtt{e}_1)\ \{\overline{\mathtt{s}_\mathtt{t}}\}\ \mathtt{else}\ \{\overline{\mathtt{s}_\mathtt{f}}\}$, $\varepsilon' = \mathtt{if}\ (\mathtt{e}'_1)\ \{\overline{\mathtt{s}'_\mathtt{t}}\}\ \mathtt{else}\ \{\overline{\mathtt{s}'_\mathtt{f}}\}$

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \mathbf{TD}_t \qquad \mathbf{TD}_f}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{if}\ (\mathtt{e}_1)\ \{\overline{\mathtt{s}_\mathtt{t}}\}\ \mathtt{else}\ \{\overline{\mathtt{s}_\mathtt{f}}\} : t_i \backslash \mathcal{C}_i}\ (\text{If})}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{if}\ (\mathtt{e}_1)\ \{\overline{\mathtt{s}_\mathtt{t}}\}\ \mathtt{else}\ \{\overline{\mathtt{s}_\mathtt{f}}\} : t \backslash \mathcal{C}}\ (\text{Sub})$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}'_1 \qquad \mathbf{TD}'_t \qquad \mathbf{TD}'_f}{\Gamma', pc'_2, \mathcal{H}' \vdash \mathtt{if}\ (\mathtt{e}'_1)\ \{\overline{\mathtt{s}'_\mathtt{t}}\}\ \mathtt{else}\ \{\overline{\mathtt{s}'_\mathtt{f}}\} : t'_i \backslash \mathcal{C}'_i}\ (\text{If})}{\Gamma', pc'_1, \mathcal{H}' \vdash \mathtt{if}\ (\mathtt{e}'_1)\ \{\overline{\mathtt{s}'_\mathtt{t}}\}\ \mathtt{else}\ \{\overline{\mathtt{s}'_\mathtt{f}}\} : t' \backslash \mathcal{C}'}\ (\text{Sub})$$

$\mathtt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathtt{e}'_1, \mathbf{TD}'_1, \mathcal{C}'_\top$, *and* $\{\overline{\mathtt{s}_\mathtt{t}}\}, \mathbf{TD}_t, \mathcal{C}_\top \simeq_{\mathtt{Low}} \{\overline{\mathtt{s}'_\mathtt{t}}\}, \mathbf{TD}'_t, \mathcal{C}'_\top$, *and*

$\{\overline{\mathtt{s}_\mathtt{f}}\}, \mathbf{TD}_f, \mathcal{C}_\top \simeq_{\mathtt{Low}} \{\overline{\mathtt{s}'_\mathtt{f}}\}, \mathbf{TD}'_f, \mathcal{C}'_\top$, *and* $t_i = t'_i$, *and* $t = t'$*; or*

(f) $\varepsilon = \overline{\mathtt{s}}$, $\varepsilon' = \overline{\mathtt{s}'}$,

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \mathbf{TD}_2}{\Gamma, pc_2, \mathcal{H} \vdash \overline{\mathtt{s}} : \tau_s \backslash \mathcal{C}_s}\ (\text{Seq})}{\Gamma, pc_1, \mathcal{H} \vdash \overline{\mathtt{s}} : t \backslash \mathcal{C}}\ (\text{Sub})$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}'_1 \qquad \mathbf{TD}'_2}{\Gamma', pc'_2, \mathcal{H}' \vdash \overline{\mathtt{s}'} : \tau'_s \backslash \mathcal{C}'_s}\ (\text{Seq})}{\Gamma', pc'_1, \mathcal{H}' \vdash \overline{\mathtt{s}'} : t' \backslash \mathcal{C}'}\ (\text{Sub})$$

$\overline{\mathtt{s}} = \mathtt{s}_1; \overline{\mathtt{s}_2}$ *and* $\overline{\mathtt{s}'} = \mathtt{s}'_1; \overline{\mathtt{s}'_2}$.

$\mathtt{s}_1, \mathbf{TD}_1, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathtt{s}'_1, \mathbf{TD}'_1, \mathcal{C}'_\top$, *and* $\overline{\mathtt{s}_2}, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} \overline{\mathtt{s}'_2}, \mathbf{TD}'_2, \mathcal{C}'_\top$, *and* $t = t'$*; or*

(g) $\varepsilon = \texttt{return}\ \texttt{e}_1$, $\varepsilon' = \texttt{return}\ \texttt{e}'_1$

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_1}{\Gamma, pc_2, \mathcal{H} \vdash \texttt{return}\ \texttt{e}_1 : \tau_1 \backslash \mathcal{C}_1}\ \text{(Return)}}{\Gamma, pc_1, \mathcal{H} \vdash \texttt{return}\ \texttt{e}_1 : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}'_1}{\Gamma', pc'_2, \mathcal{H}' \vdash \texttt{return}\ \texttt{e}'_1 : \tau'_1 \backslash \mathcal{C}'_1}\ \text{(Return)}}{\Gamma', pc'_1, \mathcal{H}' \vdash \texttt{return}\ \texttt{e}'_1 : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$\texttt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top \simeq_{\texttt{Low}} \texttt{e}'_1, \mathbf{TD}'_1, \mathcal{C}'_\top$, *and and* $t = t'$*; or*

(h) $\varepsilon = \{\overline{\texttt{s}}\}$, $\varepsilon' = \{\overline{\texttt{s}'}\}$

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_s}{\Gamma, pc_2, \mathcal{H} \vdash \{\overline{\texttt{s}}\} : \tau_s \backslash \mathcal{C}_s}\ \text{(Block)}}{\Gamma, pc_1, \mathcal{H} \vdash \{\overline{\texttt{s}}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}'_s}{\Gamma', pc'_2, \mathcal{H}' \vdash \{\overline{\texttt{s}'}\} : \tau'_s \backslash \mathcal{C}'_s}\ \text{(Block)}}{\Gamma', pc'_1, \mathcal{H}' \vdash \{\overline{\texttt{s}'}\} : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$\overline{\texttt{s}}, \mathbf{TD}_s, \mathcal{C}_\top \simeq_{\texttt{Low}} \overline{\texttt{s}'}, \mathbf{TD}'_s, \mathcal{C}'_\top$, *and* $t = t'$*; or*

(i) $\varepsilon = \texttt{e}_1.\texttt{f} = \texttt{e}_2$, $\varepsilon' = \texttt{e}'_1.\texttt{f} = \texttt{e}'_2$

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \mathbf{TD}_2}{\Gamma, pc_2, \mathcal{H} \vdash \texttt{e}_1.\texttt{f} = \texttt{e}_2 : \tau_a \backslash \mathcal{C}_a}\ \text{(F-Assign)}}{\Gamma, pc_1, \mathcal{H} \vdash \texttt{e}_1.\texttt{f} = \texttt{e}_2 : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}'_1 \qquad \mathbf{TD}'_2}{\Gamma', pc'_2, \mathcal{H}' \vdash \texttt{e}'_1.\texttt{f} = \texttt{e}'_2 : \tau'_a \backslash \mathcal{C}'_a}\ \text{(F-Assign)}}{\Gamma', pc'_1, \mathcal{H}' \vdash \texttt{e}'_1.\texttt{f} = \texttt{e}'_2 : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$\texttt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top \simeq_{\texttt{Low}} \texttt{e}'_1, \mathbf{TD}'_1, \mathcal{C}'_\top$, *and* $\texttt{e}_2, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\texttt{Low}} \texttt{e}'_2, \mathbf{TD}'_2, \mathcal{C}'_\top$, *and* $t = t'$*; or*

(j) $\varepsilon = \texttt{new}\ \texttt{C}(\overline{\texttt{e}})$, $\varepsilon' = \texttt{new}\ \texttt{C}(\overline{\texttt{e}'})$, *and*

$$\mathbf{TD} = \dfrac{\dfrac{\overline{\mathbf{TD}} \qquad \overline{\mathbf{TD}_n}}{\Gamma, pc_2, \mathcal{H} \vdash \texttt{new}\ \texttt{C}(\overline{\texttt{e}}) : t_o \backslash \mathcal{C}_o}\ \text{(New)}}{\Gamma, pc_1, \mathcal{H} \vdash \texttt{new}\ \texttt{C}(\overline{\texttt{e}}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\dfrac{\overline{\mathbf{TD}'} \qquad \overline{\mathbf{TD}'_n}}{\Gamma', pc'_2, \mathcal{H}' \vdash \texttt{new}\ \texttt{C}(\overline{\texttt{e}'}) : t'_o \backslash \mathcal{C}'_o}\ \text{(New)}}{\Gamma', pc'_1, \mathcal{H}' \vdash \texttt{new}\ \texttt{C}(\overline{\texttt{e}'}) : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

and $\texttt{C.K}(\ldots \overset{\cdots}{\longrightarrow} t_m) \in \mathcal{C}_o$, *and* $\texttt{C.K}(\ldots \overset{\cdots}{\longrightarrow} t'_m) \in \mathcal{C}'_o$, *and* $\overline{\texttt{e}}, \overline{\mathbf{TD}}, \mathcal{C}_\top \simeq_{\texttt{Low}} \overline{\texttt{e}'}, \overline{\mathbf{TD}'}, \mathcal{C}'_\top$, *and*

$\overline{\texttt{null}}, \overline{\mathbf{TD}_n}, \mathcal{C}_\top \simeq_{\texttt{Low}} \overline{\texttt{null}'}, \overline{\mathbf{TD}'_n}, \mathcal{C}'_\top$, *and* $t_o = t'_o$, $t_m = t'_m$, *and* $t = t'$*; or*

(k) $\varepsilon = \mathtt{e}_1.\mathtt{m}(\overline{\mathtt{e}})$, $\varepsilon' = \mathtt{e}'_1.\mathtt{m}(\overline{\mathtt{e}'})$, *and*

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \overline{\mathbf{TD}}}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{e}_1.\mathtt{m}(\overline{\mathtt{e}}) : t_m \backslash \mathcal{C}_m}\ (\text{Invoke})}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{e}_1.\mathtt{m}(\overline{\mathtt{e}}) : t \backslash \mathcal{C}}\ (\text{Sub})$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}'_1 \qquad \overline{\mathbf{TD}'}}{\Gamma', pc'_2, \mathcal{H}' \vdash \mathtt{e}'_1.\mathtt{m}(\overline{\mathtt{e}'}) : t'_m \backslash \mathcal{C}'_m}\ (\text{Invoke})}{\Gamma', pc'_1, \mathcal{H}' \vdash \mathtt{e}'_1.\mathtt{m}(\overline{\mathtt{e}'}) : t' \backslash \mathcal{C}'}\ (\text{Sub})$$

$\mathtt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathtt{e}'_1, \mathbf{TD}'_1, \mathcal{C}'_\top$, *and* $\overline{\mathtt{e}}, \overline{\mathbf{TD}}, \mathcal{C}_\top \simeq_{\mathtt{Low}} \overline{\mathtt{e}'}, \overline{\mathbf{TD}'}, \mathcal{C}'_\top$, *and* $t_m = t'_m$, *and* $t = t'$*;*

or

(l) $\varepsilon = \mathtt{e}_1.\mathtt{super}(\mathtt{C}, \overline{\mathtt{e}})$, $\varepsilon' = \mathtt{e}'_1.\mathtt{super}(\mathtt{C}, \overline{\mathtt{e}'})$

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \overline{\mathbf{TD}}}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{e}_1.\mathtt{super}(\mathtt{C}, \overline{\mathtt{e}}) : t_m \backslash \mathcal{C}_m}\ (\text{Super})}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{e}_1.\mathtt{super}(\mathtt{C}, \overline{\mathtt{e}}) : t \backslash \mathcal{C}}\ (\text{Sub})$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}'_1 \qquad \overline{\mathbf{TD}'}}{\Gamma', pc'_2, \mathcal{H}' \vdash \mathtt{e}'_1.\mathtt{super}(\mathtt{C}, \overline{\mathtt{e}'}) : t'_m \backslash \mathcal{C}'_m}\ (\text{Super})}{\Gamma', pc'_1, \mathcal{H}' \vdash \mathtt{e}'_1.\mathtt{super}(\mathtt{C}, \overline{\mathtt{e}'}) : t' \backslash \mathcal{C}'}\ (\text{Sub})$$

$\mathtt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathtt{e}'_1, \mathbf{TD}'_1, \mathcal{C}'_\top$, *and* $\overline{\mathtt{e}}, \overline{\mathbf{TD}}, \mathcal{C}_\top \simeq_{\mathtt{Low}} \overline{\mathtt{e}'}, \overline{\mathbf{TD}'}, \mathcal{C}'_\top$, *and* $t_m = t'_m$, *and* $t = t'$*;*

or

(m) $\varepsilon = o.\mathtt{super}(\mathtt{Object})$, $\varepsilon' = o'.\mathtt{super}(\mathtt{Object})$

(n) $\varepsilon = \mathtt{read}_{\mathtt{L}}(\mathtt{e}_1)$, $\varepsilon' = \mathtt{read}_{\mathtt{L}}(\mathtt{e}'_1)$

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_1}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{e}_1) : \tau_r \backslash \mathcal{C}_r}\ (\text{Input})}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{e}_1) : t \backslash \mathcal{C}}\ (\text{Sub})$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}'_1}{\Gamma', pc'_2, \mathcal{H}' \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{e}'_1) : \tau'_r \backslash \mathcal{C}'_r}\ (\text{Input})}{\Gamma', pc'_1, \mathcal{H}' \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{e}'_1) : t' \backslash \mathcal{C}'}\ (\text{Sub})$$

$\mathtt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathtt{e}'_1, \mathbf{TD}'_1, \mathcal{C}'_\top$, *and* $t = t'$*; or*

(o) $\varepsilon = \mathtt{write}_{\mathtt{L}}(\mathtt{e}_1, \mathtt{e}_2)$, $\varepsilon' = \mathtt{write}_{\mathtt{L}}(\mathtt{e}'_1, \mathtt{e}'_2)$

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \mathbf{TD}_2}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{write}_{\mathtt{L}}(\mathtt{e}_1, \mathtt{e}_2) : \tau_r \backslash \mathcal{C}_r}\ (\text{Input})}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{write}_{\mathtt{L}}(\mathtt{e}_1, \mathtt{e}_2) : t \backslash \mathcal{C}}\ (\text{Sub})$$

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}_1' \qquad \mathbf{TD}_2'}{\Gamma', pc_2', \mathcal{H}' \vdash \mathtt{write_L}(\mathtt{e}_1', \mathtt{e}_2') : \tau_r' \backslash \mathcal{C}_r'}\ \text{(Input)}}{\Gamma', pc_1', \mathcal{H}' \vdash \mathtt{write_L}(\mathtt{e}_1', \mathtt{e}_2') : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$\mathtt{e}_1, \mathbf{TD}_1, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathtt{e}_1', \mathbf{TD}_1', \mathcal{C}_\top'$, *and* $\mathtt{e}_2, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathtt{e}_2', \mathbf{TD}_2', \mathcal{C}_\top'$, *and* $t = t'$*; or*

(p) $\varepsilon = ;, \varepsilon' = ;$

$$\mathbf{TD} = \dfrac{\cdots}{\Gamma, pc_1, \mathcal{H} \vdash ; : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}' = \dfrac{\cdots}{\Gamma', pc_1', \mathcal{H}' \vdash ; : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

and $t = t'$.

3. *(Heaps)* $\mathcal{H}, \mathcal{C}_\top, H \simeq_{\mathtt{Low}} \mathcal{H}', \mathcal{C}_\top', H'$ *iff* $\mathcal{C}_\top = \mathcal{C}_\top'$ *and the following two conditions hold:*

   (a) $\forall o \in dom(H)$ *such that* $H(o) = \mathtt{C}, F$, *and* $F = \{\overline{\mathtt{f} = v}\}$, *and* $\mathcal{H}(o) = t_o \backslash \mathcal{C}_o$, *if* $Con(s_o, \mathcal{C}_\top) \subseteq \mathtt{Low}$, *then* $H'(o) = \mathtt{C}, F'$, *and* $\mathcal{H}'(o) = t_o \backslash \mathcal{C}_o'$, *and* $F = \{\overline{\mathtt{f} = v'}\}$, *and* $Con(s_o, \mathcal{C}_\top') \subseteq \mathtt{Low}$, *and* $\overline{Con(f_o.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top), v} \simeq_{\mathtt{Low}} \overline{Con(f_o.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top'), v'}$.

   (b) $\forall o \in dom(H')$ *such that* $H'(o) = \mathtt{C}, F'$, *and* $F' = \{\overline{\mathtt{f} = v'}\}$, *and* $\mathcal{H}'(o) = t_o \backslash \mathcal{C}_o'$, *if* $Con(s_o', \mathcal{C}_\top') \subseteq \mathtt{Low}$, *then* $H(o) = \mathtt{C}, F$, *and* $\mathcal{H}(o) = t_o \backslash \mathcal{C}_o$, *and* $F = \{\overline{\mathtt{f} = v}\}$, *and* $Con(s_o, \mathcal{C}_\top) \subseteq \mathtt{Low}$, *and* $\overline{Con(f_o.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top), v} \simeq_{\mathtt{Low}} \overline{Con(f_o.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top'), v'}$.

4. *(Output Stream Equality).* $\omega_1(\mathtt{fd}, \mathtt{L}_1) = \omega_2(\mathtt{fd}, \mathtt{L}_2)$ *iff* $\omega_1(\mathtt{fd}, \mathtt{L}_1) = \overline{\mathtt{c}_1}$, $\omega_2(\mathtt{fd}, \mathtt{L}_2) = \overline{\mathtt{c}_2}$, *and* $\mathtt{L}_1 = \mathtt{L}_2$, $|\overline{\mathtt{c}_1}| = |\overline{\mathtt{c}_2}|$, $\forall i < |\overline{\mathtt{c}_1}|, \mathtt{c}_{1i} = \mathtt{c}_{2i}$.

5. *(Input Stream Equality).* $\iota_1(\mathtt{fd}, \mathtt{L}_1) = \iota_2(\mathtt{fd}, \mathtt{L}_2)$ *iff* $\iota_1(\mathtt{fd}, \mathtt{L}_1) = \overline{\mathtt{c}_1}$, $\iota_2(\mathtt{fd}, \mathtt{L}_2) = \overline{\mathtt{c}_2}$, *and* $\mathtt{L}_1 = \mathtt{L}_2$, $|\overline{\mathtt{c}_1}| = |\overline{\mathtt{c}_2}|$, $\forall i < |\overline{\mathtt{c}_1}|, \mathtt{c}_{1i} = \mathtt{c}_{2i}$.

6. *(Output Streams).* $\omega_1 \simeq_{\mathtt{Low}} \omega_2$ *iff* $dom(\omega_1) = dom(\omega_2)$ *and* $\forall (\mathtt{fd}, \mathtt{L}_1) \in dom(\omega_1)$, *either*

   (a) $\mathtt{L}_1 \subseteq \mathtt{Low}$, $\mathtt{L}_2 \subseteq \mathtt{Low}$, *and* $\omega_1(\mathtt{fd}, \mathtt{L}_1) = \omega_2(\mathtt{fd}, \mathtt{L}_2)$*; or*

(b) $\mathtt{L}_1 \not\sqsubseteq \mathtt{Low}$ *and* $\mathtt{L}_2 \not\sqsubseteq \mathtt{Low}$*, and* $(\mathtt{fd}, \mathtt{L}_2) \in dom(\omega_2)$.

7. *(Input Streams).* $\iota_1 \simeq_{\mathtt{Low}} \iota_2$ *iff* $dom(\iota_1) = dom(\iota_2)$ *and* $\forall(\mathtt{fd}, \mathtt{L}_1) \in dom(\iota_1)$*, either*

   (a) $\mathtt{L}_1 \sqsubseteq \mathtt{Low}$*,* $\mathtt{L}_2 \sqsubseteq \mathtt{Low}$*, and* $\iota_1(\mathtt{fd}, \mathtt{L}_1) = \iota_2(\mathtt{fd}, \mathtt{L}_2)$*; or*

   (b) $\mathtt{L}_1 \not\sqsubseteq \mathtt{Low}$ *and* $\mathtt{L}_2 \not\sqsubseteq \mathtt{Low}$*, and* $(\mathtt{fd}, \mathtt{L}_2) \in dom(\iota_2)$.

8. $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \simeq_{\mathtt{Low}} \varepsilon', \mathbf{TD}', \mathcal{C}'_\top, H', \iota', \omega'$ *iff* $\varepsilon, \mathbf{TD}, \mathcal{C}_\top \simeq_{\mathtt{Low}} \varepsilon', \mathbf{TD}', \mathcal{C}'_\top$*, (*$\mathcal{H}$ *and* $\mathcal{H}'$ *are respective to* $\mathbf{TD}$ *and* $\mathbf{TD}'$*)* $\mathcal{H}, \mathcal{C}_\top, H \simeq_{\mathtt{Low}} \mathcal{H}', \mathcal{C}'_\top, H'$*,* $\iota \simeq_{\mathtt{Low}} \iota'$*, and* $\omega \simeq_{\mathtt{Low}} \omega'$.

As previously mentioned, we define low steps and high steps based on the type derivation in the execution step. This is a convenient method of denoting steps that *must* be taken by two bisimilar configurations. They are all based on the concrete labels produced by the type derivation and the constraint set $\mathcal{C}_\top$. For example, a reduction of (Assign-R) is a low step only if the object identifier and the value being written are both low; this means a change is being made to a low heap location, so when reasoning about two runs, the other must make the same change to the same heap location. The reader should be careful to recognize that this is a technique for aligning certain execution steps that have low behavior in terms of what they alter (as in assignment, read, and write), or in what computations they entail (as in conditionals, method calls, and constructor calls). In many cases, two bisimilar configurations will both execute the same high steps, that one might generally consider low behavior; this is inconsequential, as the definitions of low and high steps are merely a tool for establishing low-equivalence of the computation. Indeed, in such instances both configurations will always return to a bisimilar configuration after high steps, as shown later, in Lemma 4.26. Hence, one should not assume that all execution steps shared by both runs are considered low steps.

Definition 4.16 formalizes which steps are low steps, and high steps are simply defined as any step that is not a low step.

Definition 4.16 (Low step)

A low step, denoted $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow_l \varepsilon', \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ *is a step,*

$\varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \varepsilon', \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ *iff either*

1. $\rightsquigarrow$ *is a use of (IfTrue-R)*

   and $\mathbf{TD} =$
   $$\dfrac{\dfrac{\mathbf{TD}_c \qquad \mathbf{TD}_t \qquad \mathbf{TD}_f}{\Gamma, pc_2, \mathcal{H}_1 \vdash \texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\} : t_i \backslash \mathcal{C}_i}\ \text{(If)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$
   and $\mathbf{TD}_c =$
   $$\dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \texttt{True} : t_c \backslash \mathcal{C}_c}\ \text{(Sub)}$$
   and $Con(s_c, \mathcal{C}_\top) \subseteq \texttt{Low}$.

2. $\rightsquigarrow$ *is a use of (IfFalse-R)*

   and $\mathbf{TD} =$
   $$\dfrac{\dfrac{\mathbf{TD}_c \qquad \mathbf{TD}_t \qquad \mathbf{TD}_f}{\Gamma, pc_2, \mathcal{H}_1 \vdash \texttt{if}\ (\texttt{False})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\} : t_i \backslash \mathcal{C}_i}\ \text{(If)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \texttt{if}\ (\texttt{False})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$
   and $\mathbf{TD}_c =$
   $$\dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \texttt{False} : t_c \backslash \mathcal{C}_c}\ \text{(Sub)}$$
   and $Con(s_c, \mathcal{C}_\top) \subseteq \texttt{Low}$.

3. $\rightsquigarrow$ *is a use of (Assign-R)*

   and $\mathbf{TD} =$
   $$\dfrac{\dfrac{\mathbf{TD}_o \qquad \mathbf{TD}_v}{\begin{array}{c}\Gamma, pc_2, \mathcal{H}_1 \vdash o.\texttt{f} = v : \langle\, s_o \cup s_v, \emptyset, \texttt{void} \,\rangle \backslash \mathcal{C}_o \cup \mathcal{C}_v \cup \\ \{f_o.\texttt{f} <: \textbf{\textit{set}} \langle\, s_o \cup s_v, f_v, \alpha_v \,\rangle\}\end{array}}\ \text{(F-Assign)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash o.\texttt{f} = v : t \backslash \mathcal{C}}\ \text{(Sub)}$$
   and $\mathbf{TD}_o =$
   $$\dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash o : t_o \backslash \mathcal{C}_o}\ \text{(Sub)}$$
   and $\mathbf{TD}_v =$
   $$\dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash v : t_v \backslash \mathcal{C}_v}\ \text{(Sub)}$$
   and $Con(s_o, \mathcal{C}_\top) \subseteq \texttt{Low}$ *and* $Con(s_v, \mathcal{C}_\top) \subseteq \texttt{Low}$.

4. $\rightsquigarrow$ *is a use of (Seq-R), or (Block-R)*

   Then $\rightsquigarrow$ *is* not *a low step.*

5. $\rightsquigarrow$ *is a use of* (New-R)

   $$\mathbf{TD} = \dfrac{\dfrac{\overline{\mathbf{TD}_v} \qquad \overline{\mathbf{TD}_n}}{\Gamma, pc_2, \mathcal{H} \vdash \texttt{new C}(\overline{v}) : t_c \backslash \ldots}\ \text{(New)}}{\Gamma, pc_1, \mathcal{H} \vdash \texttt{new C}(\overline{v}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   and $Con(s_c, \mathcal{C}_\top) \subseteq \texttt{Low}$

6. $\rightsquigarrow$ *is a use of* (Invoke-R)

   $$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_o \qquad \overline{\mathbf{TD}_v}}{\Gamma, pc_2, \mathcal{H} \vdash o.\texttt{m}(\overline{v}) : t_m \backslash \ldots}\ \text{(Invoke)}}{\Gamma, pc_1, \mathcal{H} \vdash o.\texttt{m}(\overline{v}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   and $\mathbf{TD}_o = \dfrac{\ldots}{\Gamma, pc_2, \mathcal{H} \vdash o : t_o \backslash \mathcal{C}_o}\ \text{(Sub)}$

   and $Con(s_o, \mathcal{C}_\top) \subseteq \texttt{Low}$

7. $\rightsquigarrow$ *is a use of* (Super-R)

   $$\mathbf{TD} = \dfrac{\dfrac{\overline{\mathbf{TD}_o} \qquad \overline{\mathbf{TD}_v}}{\Gamma, pc_2, \mathcal{H} \vdash o.\texttt{super}(\texttt{D}, \overline{v}) : t_m \backslash \ldots}\ \text{(Super)}}{\Gamma, pc_1, \mathcal{H} \vdash o.\texttt{super}(\texttt{D}, \overline{v}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   and $\mathbf{TD}_o = \dfrac{\ldots}{\Gamma, pc_2, \mathcal{H} \vdash o : t_o \backslash \mathcal{C}_o}\ \text{(Sub)}$

   and $Con(s_o, \mathcal{C}_\top) \subseteq \texttt{Low}$

8. $\rightsquigarrow$ *is a use of* (Input-R)

   and $\texttt{read}_\texttt{L}(\texttt{fd}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \texttt{c}, \mathbf{TD}', \mathcal{C}_\top, H, \iota', \omega$*, and* $S_f = conread(\mathbf{TD}, \mathcal{C}_\top)$*, and* $S_f \subseteq$ Low*.*

9. $\rightsquigarrow$ *is a use of* (Output-R)

   and $\texttt{write}_\texttt{L}(\texttt{c}, \texttt{fd}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow ;, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega'$*, and*

   $S_f = conwrite(\mathbf{TD}, \mathcal{C}_\top)$ *and* $S_f \subseteq \texttt{Low}$*.*

10. $\rightsquigarrow$ *is a use of* (*-RC)

If the context reduction is $\varepsilon_1, \mathbf{TD}_1, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow_l \varepsilon_1', \mathbf{TD}_1', \mathcal{C}_\top, H', \iota', \omega'$.

(In other words, the reduction under context is a low step.)

11. $\rightsquigarrow$ *is the use of (Field-R), (Op-R), (Cast-R), or (Return-R)*

and $\mathbf{TD} = \dfrac{\cdots}{\Gamma, pc, H \vdash \varepsilon : t \backslash \mathcal{C}}$ (Sub)

and $Con(s, \mathcal{C}_\top) \subseteq \texttt{Low}$

Definition 4.17 (High step) *A high step, denoted* $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow_h \varepsilon', \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ *is a step,* $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow \varepsilon', \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$ *iff* $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \not\rightsquigarrow_l \varepsilon', \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$.

Before giving our main lemmas, we first produce several lemmas which are needed in the subsequent proofs. The bisimulation relation is shown to be reflexive, symmetric, and transitive in Lemma 4.18.

Lemma 4.19 shows that for two bisimilar heaps, a new low heap location will be the same on both heaps. Lemma 4.20 shows that if a secrecy type constraint $s <: s'$ appears in the constraint set, then the concrete type of each maintains a subset relationship. Lemma 4.21 shows that a type derivation whose heap environment is increased by $tdheap$ maintains a bisimulation with itself (i.e. nothing changes in the bisimulation relationship of the expression and type derivation).

Lemma 4.18 (Properties of Bisimulation)

1. *(Reflexive)* $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \simeq_{\texttt{Low}} \varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$.
2. *(Symmetric) If* $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \simeq_{\texttt{Low}} \varepsilon', \mathbf{TD}', \mathcal{C}_\top', H', \iota', \omega'$*, then*

   $\varepsilon', \mathbf{TD}', \mathcal{C}_\top', H', \iota', \omega' \simeq_{\texttt{Low}} \varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$.

3. *(Transitive) If* $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \simeq_{\mathtt{Low}} \varepsilon', \mathbf{TD}', \mathcal{C}'_\top, H', \iota', \omega'$*, and*

   $\varepsilon', \mathbf{TD}', \mathcal{C}'_\top, H', \iota', \omega' \simeq_{\mathtt{Low}} \varepsilon'', \mathbf{TD}'', \mathcal{C}''_\top, H'', \iota'', \omega''$*, then*

   $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \simeq_{\mathtt{Low}} \varepsilon'', \mathbf{TD}'', \mathcal{C}''_\top, H'', \iota'', \omega''$

Proof. By induction on the structure of ε_1 and directly by Definition 4.15. □

Lemma 4.19 *If* $S \subseteq \mathtt{Low}$*,* $S' \subseteq \mathtt{Low}$*,* $S = S'$ *and* $\mathcal{H}, \mathcal{C}_\top, H \simeq_{\mathtt{Low}} \mathcal{H}', \mathcal{C}'_\top, H'$*, then* $newref\,(H, S) = newref\,(H', S')$

Proof. By contradiction.

Suppose $newref\,(H, S) = o$, $newref\,(H', S') = o'$, and $o \neq o'$. By the definition of $newref$, $o = loc_i^S$ and $o' = loc_{i'}^{S'}$. Now, by assumption, $S = S'$, so in order to satisfy $o \neq o'$, we must have $i \neq i'$. As per the definition of $newref$, let $i-1$ be the largest integer, such that $loc_{i-1}^S \in dom(H)$, and let $i'-1$ be the largest integer, such that $loc_{i'-1}^{S'} \in dom(H')$. Without loss of generality, assume $i - 1 > i' - 1$.

Now, according to Definition 4.15[3], since $S \subseteq \mathtt{Low}$, and $S' \subseteq \mathtt{Low}$, we must have $loc_{i-1}^S \in dom(H')$ and $loc_{i'-1}^{S'} \in dom(H)$. Since $i - 1 > i' - 1$ and $loc_{i-1}^S \in dom(H')$, then $i' - 1$ is not the largest integer such that $loc_{i'-1}^{S'} \in dom(H')$, a contradiction. Hence, the assumption that $i \neq i'$ is wrong, and therefore the assumption that $o \neq o'$ is also wrong. Hence, $o = o'$.

□

Lemma 4.20 (Concrete Label Relation) *If* $s <: s' \in \mathcal{C}$ *and* $\mathcal{C}$ *is closed, then* $Con(s, \mathcal{C}) \subseteq Con(s', \mathcal{C})$.

Proof. Let $\mathtt{l}$ be any concrete label in $Con(s, \mathcal{C})$. Hence by Definition 4.2, $\mathtt{l} <: s \in \mathcal{C}$. Since by assumption $s <: s' \in \mathcal{C}$ by closure rule ($\mathcal{S}$-Trans), $\mathtt{l} <: s' \in \mathcal{C}$. Then, by Definition 4.2, $\mathtt{l} \in Con(s', \mathcal{C})$. Therefore $Con(s, \mathcal{C}) \subseteq Con(s', \mathcal{C})$. □

Lemma 4.21 *If* $\mathbf{TD}' = tdheap(\mathbf{TD}, \mathcal{H}')$*, and* $\mathbf{TD} \rhd \varepsilon, H, pc, t, \mathcal{C},$*, and* $\mathcal{H}$ *is the heap environment of* $\mathbf{TD}$*, and* $\mathcal{H}'$ *is the heap environment of* $\mathbf{TD}'$*, and* $\mathcal{H}'$ *extends* $\mathcal{H}$*, then for any* $\mathcal{C}_\top$*,* $\varepsilon, \mathbf{TD}, \mathcal{C}_\top \simeq_{\mathtt{Low}} \varepsilon, \mathbf{TD}', \mathcal{C}_\top$.

Proof. By induction on the structure of $\mathbf{TD}$, using Definition 4.15. □

The following Lemma 4.22 is an important lemma needed for proving the method invocation case of the Lemma 4.23; it states that for two sets of bisimilar values (and type derivations), $\overline{v}$ and $\overline{v'}$, if they are both substituted for variables $\overline{\mathtt{x}}$ in the same expression, ε, then the resulting substituted expressions $[\overline{\mathtt{x}} \mapsto \overline{v}]\varepsilon$ and $[\overline{\mathtt{x}} \mapsto \overline{v'}]\varepsilon$ (and their respective derivations) are also bisimilar. Hence, this shows that two bisimilar invocations of the same method produce a bisimilar sequence of statements for further execution.

Lemma 4.22 (Bisimulation of Substitution)

If $\overline{v}, \overline{\mathbf{TD}_v}, \mathcal{C}_\top \simeq_{\mathtt{Low}} \overline{v'}, \overline{\mathbf{TD}'_v}, \mathcal{C}'_\top$*, and* $\mathbf{TD} = tdsub(\varepsilon, \overline{v}, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r], \mathbf{TD}_m, \overline{\mathbf{TD}_v})$*, and* $\mathbf{TD}' = tdsub(\varepsilon, \overline{v'}, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc', \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r], \mathbf{TD}_m, \overline{\mathbf{TD}'_v})$*, and* $\mathcal{C}_a \subseteq \mathcal{C}_\top$*, and* $\mathcal{C}'_a \subseteq \mathcal{C}'_\top$*, and* $\mathcal{C}_\top = \mathcal{C}'_\top$*, then* $[\overline{\mathtt{x}} \mapsto \overline{v}]\varepsilon, \mathbf{TD}, \mathcal{C}_\top \simeq_{\mathtt{Low}} [\overline{\mathtt{x}} \mapsto \overline{v'}]\varepsilon, \mathbf{TD}', \mathcal{C}'_\top$.

Proof.

By induction on the structure of ε.

1. $\varepsilon = \mathtt{x}$.

   Let $v = [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{x}$ and $v' = [\overline{\mathtt{x}} \mapsto \overline{v'}]\mathtt{x}$.

   According to the definition of $tdsub$, we have the following.

$$\mathbf{TD}_m = \dfrac{\dfrac{}{[\overline{\mathtt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \mathtt{x} : \langle\, s_x \cup s_p, f_x, \alpha_x \,\rangle \backslash \emptyset}\ \text{(Var)}}{[\overline{\mathtt{x}} \mapsto \overline{t_x}] s_p, \emptyset \vdash \mathtt{x} : t_m \backslash \mathcal{C}_m}\ \text{(Sub)}$$

$\{\langle\, s_x \cup s_p, f_x, \alpha_x \,\rangle <: t_m\} \subseteq \mathcal{C}_m$

$\mathbf{TD} =$

$$\frac{\cdots}{\Gamma, pc, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{x} : t_a \backslash \mathcal{C}_a} \text{ (Sub)}$$

Where $t_a = [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_m$ and $\mathcal{C}_a = \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \cup \overline{\mathcal{C}_v})$

$\mathbf{TD}' =$

$$\frac{\cdots}{\Gamma', pc', \mathcal{H}' \vdash [\overline{\mathtt{x}} \mapsto \overline{v'}]\mathtt{x} : t'_a \backslash \mathcal{C}'_a} \text{ (Sub)}$$

Where $t'_a = [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc', \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_m$ and $\mathcal{C}'_a = \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc', \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \cup \overline{\mathcal{C}'_v})$

Hence, $t_a = t'_a$, so $\mathbf{TD}$ and $\mathbf{TD}'$ have the same final types.

Assuming $v = [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{x}$ and $v' = [\overline{\mathtt{x}} \mapsto \overline{v'}]\mathtt{x}$. It remains to be shown that $\mathit{Con}(s_a, \mathcal{C}_\top), v \simeq_{\mathtt{Low}} \mathit{Con}(s'_a, \mathcal{C}'_\top), v'$. Since we have already shown that $s_a = s'_a$, and $\mathcal{C}_\top = \mathcal{C}'_\top$, it suffices to show $\mathit{Con}(s_a, \mathcal{C}_\top), v \simeq_{\mathtt{Low}} \mathit{Con}(s_a, \mathcal{C}_\top), v'$.

Now, by assumption that $v, \mathbf{TD}_v, \mathcal{C}_\top \simeq_{\mathtt{Low}} v', \mathbf{TD}'_v, \mathcal{C}'_\top$, we have

$\mathit{Con}(s_v, \mathcal{C}_\top), v \simeq_{\mathtt{Low}} \mathit{Con}(s'_v, \mathcal{C}'_\top), v'$, where $s_v = s'_v$ and $\mathcal{C}_\top = \mathcal{C}'_\top$.

Since $t_a = [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]t_m$ and $\mathcal{C}_a = \mathit{Closure}(LT, [\overline{t_l} \mapsto \overline{t'_l}][s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]\mathcal{C}_m \cup \overline{\mathcal{C}_v})$, and $\{\langle\, s_x \cup s_p, f_x, \alpha_x \,\rangle <: t_m\} \subseteq \mathcal{C}_m$, we have $\{s_v <: s_a\} \subseteq \mathcal{C}_a$. Since $\mathcal{C}_a \subseteq \mathcal{C}_\top$ by assumption, this means $\{s_v <: s_a\} \subseteq \mathcal{C}_\top$ Hence, by Lemma 4.20, we have $\mathit{Con}(s_v, \mathcal{C}_\top) \subseteq \mathit{Con}(s_a, \mathcal{C}_\top)$.

We now have two cases, according to Definition 4.15[1]

(a) $\mathit{Con}(s_a, \mathcal{C}_\top) \subseteq \mathtt{Low}$.

Since $Con(s_v, \mathcal{C}_\top) \subseteq Con(s_a, \mathcal{C}_\top)$, we have $Con(s_v, \mathcal{C}_\top) \subseteq \texttt{Low}$, and since $Con(s_v, \mathcal{C}_\top), v \simeq_{\texttt{Low}} Con(s'_v, \mathcal{C}'_\top), v'$, by Definition 4.15[1] we have $v = v'$. Hence $Con(s_a, \mathcal{C}_\top), v \simeq_{\texttt{Low}} Con(s_a, \mathcal{C}_\top), v'$.

(b) $Con(s_a, \mathcal{C}_\top) \not\subseteq \texttt{Low}$.

Then by Definition 4.15[1], $Con(s_a, \mathcal{C}_\top), v \simeq_{\texttt{Low}} Con(s_a, \mathcal{C}_\top), v'$.

Conclude with Definition 4.15[8] $v, \mathbf{TD}, \mathcal{C}_\top \simeq_{\texttt{Low}} v', \mathbf{TD}', \mathcal{C}'_\top$, which is $[\overline{\texttt{x}} \mapsto \overline{v}]\varepsilon, \mathbf{TD}, \mathcal{C}_\top \simeq_{\texttt{Low}} [\overline{\texttt{x}} \mapsto \overline{v'}]\varepsilon, \mathbf{TD}', \mathcal{C}'_\top$.

2. $\varepsilon = \texttt{e}_1 \oplus \texttt{e}_2$.

According to the definition of $tdsub$, we have the following.

$$\mathbf{TD}_m = \dfrac{\dfrac{\mathbf{TD}_1 \qquad \mathbf{TD}_2}{[\overline{\texttt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \texttt{e}_1 \oplus \texttt{e}_2 : \langle\, s_1 \cup s_2, \emptyset, \texttt{int} \,\rangle \backslash \mathcal{C}_1 \cup \mathcal{C}_2}\ \text{(Op)}}{[\overline{\texttt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \texttt{e}_1 \oplus \texttt{e}_2 : t_m \backslash \mathcal{C}_m}\ \text{(Sub)}$$

$$\mathbf{TD}_1 = \dfrac{\cdots}{[\overline{\texttt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \texttt{e}_1 : t_1 \backslash \mathcal{C}_1}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\cdots}{[\overline{\texttt{x}} \mapsto \overline{t_x}], s_p, \emptyset \vdash \texttt{e}_2 : t_2 \backslash \mathcal{C}_2}\ \text{(Sub)}$$

$$\mathbf{TD}_3 = tdsub(\texttt{e}_1, \overline{v}, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r], \mathbf{TD}_1, \overline{\mathbf{TD}_v})$$

That is,

$$\mathbf{TD}_3 = \dfrac{\cdots}{\Gamma, pc, \mathcal{H} \vdash [\overline{\texttt{x}} \mapsto \overline{v}]\texttt{e}_1 : t_3 \backslash \mathcal{C}_3}\ \text{(Sub)}$$

$$\mathbf{TD}_4 = tdsub(\texttt{e}_1, \overline{v}, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc, \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r], \mathbf{TD}_2, \overline{\mathbf{TD}_v})$$

That is,

$$\mathbf{TD}_4 = \dfrac{\cdots}{\Gamma, pc, \mathcal{H} \vdash [\overline{\texttt{x}} \mapsto \overline{v}]\texttt{e}_2 : t_4 \backslash \mathcal{C}_4}\ \text{(Sub)}$$

The constraint sets in the conclusion of $\overline{\mathbf{TD}_v}$ are $\overline{\mathcal{C}_v}$.

$$\mathbf{TD} = \dfrac{\dfrac{\mathbf{TD}_3 \qquad \mathbf{TD}_4}{\Gamma, pc, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_1 \oplus \mathtt{e}_2 : \langle\, s_3 \cup s_4, \emptyset, \mathtt{int}\,\rangle \backslash \mathcal{C}_3 \cup \mathcal{C}_4}\ \text{(Op)}}{\Gamma, pc, \mathcal{H} \vdash [\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_1 \oplus \mathtt{e}_2 : t_a \backslash \mathcal{C}_a}\ \text{(Sub)}$$

and

$$\mathbf{TD}'_3 = tdsub(\mathtt{e}_1, \overline{v'}, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc', \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r], \mathbf{TD}_1, \overline{\mathbf{TD}'_v})$$

That is,

$$\mathbf{TD}'_3 = \dfrac{\dfrac{\cdots}{\Gamma', pc', \mathcal{H}' \vdash [\overline{\mathtt{x}} \mapsto \overline{v'}]\mathtt{e}_1 : t'_1 \backslash \mathcal{C}'_1}\ \text{(Sub)}}{}$$

$$\mathbf{TD}'_4 = tdsub(\mathtt{e}_1, \overline{v'}, [\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc', \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r], \mathbf{TD}_2, \overline{\mathbf{TD}'_v})$$

That is,

$$\mathbf{TD}'_4 = \dfrac{\dfrac{\cdots}{\Gamma', pc', \mathcal{H}' \vdash [\overline{\mathtt{x}} \mapsto \overline{v'}]\mathtt{e}_2 : t'_2 \backslash \mathcal{C}'_2}\ \text{(Sub)}}{}$$

The constraint sets in the conclusion of $\overline{\mathbf{TD}_v}$ are $\overline{\mathcal{C}_v}$.

$$\mathbf{TD}' = \dfrac{\dfrac{\mathbf{TD}'_3 \qquad \mathbf{TD}'_4}{\Gamma', pc', \mathcal{H}' \vdash [\overline{\mathtt{x}} \mapsto \overline{v'}]\mathtt{e}_1 \oplus \mathtt{e}_2 : \langle\, s'_1 \cup s'_2, \emptyset, \mathtt{int}\,\rangle \backslash \mathcal{C}'_1 \cup \mathcal{C}'_2}\ \text{(Op)}}{\Gamma', pc', \mathcal{H}' \vdash [\overline{\mathtt{x}} \mapsto \overline{v'}]\mathtt{e}_1 \oplus \mathtt{e}_2 : t'_a \backslash \mathcal{C}'_a}\ \text{(Sub)}$$

Now, by induction, $[\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_1, \mathbf{TD}_3, \mathcal{C}_\top \simeq_{\mathtt{Low}} [\overline{\mathtt{x}} \mapsto \overline{v'}]\mathtt{e}_1, \mathbf{TD}'_3, \mathcal{C}_\top$ and $[\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_2, \mathbf{TD}_4, \mathcal{C}_\top \simeq_{\mathtt{Low}} [\overline{\mathtt{x}} \mapsto \overline{v'}]\mathtt{e}_2, \mathbf{TD}'_4, \mathcal{C}_\top$ and since the substitution of variables $[\overline{t_l} \mapsto \overline{t'_l}], [s_p \mapsto pc', \overline{t_x} \mapsto \overline{t_v}, t_r \mapsto t'_r]$ is the same for both $\mathbf{TD}$ and $\mathbf{TD}'$, we have $t_a = t'_a$, and by Definition 4.15[2c], $[\overline{\mathtt{x}} \mapsto \overline{v}]\mathtt{e}_1 \oplus \mathtt{e}_2, \mathbf{TD}, \mathcal{C}_\top \simeq_{\mathtt{Low}} [\overline{\mathtt{x}} \mapsto \overline{v'}]\mathtt{e}_1 \oplus \mathtt{e}_2, \mathbf{TD}', \mathcal{C}'_\top$.

3. The remaining cases follow similarly, either directly, or by induction and use of the respective case of $tdsub$ and bisimulation rule.

□

Lemma 4.23 shows that for two bisimilar expressions, a low step results in the same expression, apart from differing high values. The heaps and input and output streams all remain bisimilar.

Lemma 4.23 (Reduction of Low Security Configurations)

If $\varepsilon_1, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_l \varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2$ *and*

$\varepsilon_1, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\mathtt{Low}} \varepsilon'_1, \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$, *and* $\mathbf{TD}_1 \rhd \varepsilon_1, H_1, \Gamma, pc_1, t, \mathcal{C}$, *and*

$\mathbf{TD}'_1 \rhd \varepsilon'_1, H'_1, \Gamma', pc'_1, t', \mathcal{C}'$, *then* $\varepsilon'_1, \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow_l \varepsilon'_2, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_2, \iota'_2, \omega'_2$, *and*

$\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2 \simeq_{\mathtt{Low}} \varepsilon'_1, \mathbf{TD}'_1, \mathcal{C}'_\top, H'_2, \iota'_2, \omega'_2$.

Proof. By induction on the context derivation tree of $\rightsquigarrow_l$, with case analysis on the last (bottom) reduction rule used.

1. (Field-R) Let $\varepsilon_1 = o.\mathtt{f}$ and $\varepsilon'_1 = o'.\mathtt{f}$.

   By (Field-R), $o.\mathtt{f}, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow v, \mathbf{TD}_2, H_1, \iota_1, \omega_1$

   where $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$, and $H_1(o) = \mathtt{C}, F$, and $F(\mathtt{f}) = v$

   $$\mathbf{TD}_1 = \dfrac{\dfrac{\dfrac{\mathcal{H}_1(o) = t_h \backslash \mathcal{C}_h}{\Gamma, pc_3, \mathcal{H}_1 \vdash o : \langle\, pc \cup s_h, f_h, \alpha_h \,\rangle \backslash \mathcal{C}_h}\ \text{(Oid)}}{\Gamma, pc_2, \mathcal{H}_1 \vdash o : t_o \backslash \mathcal{C}_o}\ \text{(Sub)}}{\dfrac{\Gamma, pc_2, \mathcal{H}_1 \vdash o.\mathtt{f} : \langle\, s_o \cup f_o.\mathtt{f}.\mathsf{S}, f_o.\mathtt{f}.\mathsf{F}, f_o.\mathtt{f}.\mathsf{A} \,\rangle \backslash \mathcal{C}_o}{\Gamma, pc_1, \mathcal{H}_1 \vdash o.\mathtt{f} : t \backslash \mathcal{C}}\ \text{(Sub)}}\ \text{(Field)}$$

   Where $\mathcal{C}_h \cup \{\langle\, pc_3 \cup s_h, f_h, \alpha_h \,\rangle <: t_o\} \cup \{\langle\, s_o \cup f_o.\mathtt{f}.\mathsf{S}, f_o.\mathtt{f}.\mathsf{F}, f_o.\mathtt{f}.\mathsf{A} \,\rangle <: t\} \subseteq \mathcal{C}$.

   $$\mathbf{TD}_2 = \dfrac{\dfrac{\cdots}{\Gamma, pc_1, \mathcal{H}_1 \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}}{}$$

   Again by (Field-R), $o'.\mathtt{f}, \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow v', \mathbf{TD}'_2, H'_1, \iota'_1, \omega'_1$

   where $\mathit{fields}(\mathtt{C}') = \overline{\mathtt{C}'}\ \overline{\mathtt{f}}$, and $H'_1(o') = \mathtt{C}', F'$, and $F(\mathtt{f}') = v'$

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\dfrac{\mathcal{H}'_1(o) = t'_h \backslash \mathcal{C}'_h}{\Gamma', pc'_3, \mathcal{H}'_1 \vdash o' : \langle\, pc' \cup s'_h, f'_h, \alpha'_h \,\rangle \backslash \mathcal{C}'_h}\ \text{(Oid)}}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash o' : t'_o \backslash \mathcal{C}'_o}\ \text{(Sub)}}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash o'.\mathtt{f} : \langle\, s'_o \cup f'_o.\mathtt{f}.\mathsf{S}, f'_o.\mathtt{f}.\mathsf{F}, f'_o.\mathtt{f}.\mathsf{A} \,\rangle \backslash \mathcal{C}'_o}\ \text{(Field)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash o'.\mathtt{f} : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

Where $\mathcal{C}'_h \cup \{\langle\, pc'_3 \cup s'_h, f'_h, \alpha'_h \,\rangle <: t'_o\} \cup \{\langle\, s'_o \cup f'_o.\mathtt{f}.\mathsf{S}, f'_o.\mathtt{f}.\mathsf{F}, f'_o.\mathtt{f}.\mathsf{A} \,\rangle <: t'\} \subseteq \mathcal{C}'$.

$$\mathbf{TD}'_2 = \dfrac{\dots}{\Gamma', pc', \mathcal{H}'_1 \vdash v' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

Now, by Definition 4.16, we have $Con(s, \mathcal{C}_\top) \subseteq \mathtt{Low}$. By assumption and Definition 4.15[2b], we have $t = t'$ and $\mathcal{C}_\top = \mathcal{C}'_\top$, so $Con(s', \mathcal{C}'_\top) \subseteq \mathtt{Low}$.

Now, by ($\mathcal{S}$-Union) and ($\mathcal{S}$-Trans), from the constraints above, we have $\{s_h <: s\} \subseteq \mathcal{C}$, and since by *tdfield*, we have $\mathcal{C} \subseteq \mathcal{C}_\top$, we have $\{s_h <: s\} \subseteq \mathcal{C}_\top$. Therefore by Lemma 4.20, $Con(s_h, \mathcal{C}_\top) \subseteq Con(s, \mathcal{C}_\top)$. Since $Con(s, \mathcal{C}_\top) \subseteq \mathtt{Low}$, we have $Con(s_h, \mathcal{C}_\top) \subseteq \mathtt{Low}$.

Since by assumption $\mathcal{H}_1, \mathcal{C}_\top, H_1 \simeq_{\mathtt{Low}} \mathcal{H}'_1, \mathcal{C}'_\top, H'_1$, this means $o = o'$, and $t_h = t'_h$, and $Con(f_h.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top), v \simeq_{\mathtt{Low}} Con(f_h.\mathtt{f}.\mathsf{S}, \mathcal{C}'_\top), v'$.

By ($\mathcal{S}$-Field) and ($\mathcal{S}$-Trans) closure rules, we have $\{f_h.\mathtt{f}.\mathsf{S} <: s\} \subseteq \mathcal{C}_\top$, hence by Lemma 4.20, $Con(f_h.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top) \subseteq Con(s, \mathcal{C}_\top)$ Since $Con(s, \mathcal{C}_\top) \subseteq \mathtt{Low}$, we have $Con(f_h.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top) \subseteq \mathtt{Low}$. So, by Definition 4.15[1], we have $v = v'$, which entails $Con(s), v \simeq_{\mathtt{Low}} Con(s), v'$ by Definition 4.15[1]. Since $t = t'$, by Definition 4.15[2a], we have $v, \mathbf{TD}_1, \mathcal{C}_\top \simeq_{\mathtt{Low}} v', \mathbf{TD}'_1, \mathcal{C}'_\top$, and thus by Definition 4.15[8], $v, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\mathtt{Low}} v', \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$

2. (Op-R)

   Let $\varepsilon_1 = \mathtt{c_a} \oplus \mathtt{c_b}$ and $\varepsilon'_1 = \mathtt{c'_a} \oplus \mathtt{c'_b}$

   By (Op-R), $\mathtt{c_a} \oplus \mathtt{c_b}, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow v, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1$ where $v = \mathtt{c_a} \oplus \mathtt{c_b}$ and $\mathbf{TD}_2 = tdop(\mathbf{TD}_1, v, \mathcal{C}_\top)$, and the following.

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_a \qquad \mathbf{TD}_b}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{c_a} \oplus \mathtt{c_b} : \langle\, s_a \cup s_b, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_a \cup \mathcal{C}_b}\,(\text{Op})}{\Gamma, pc_1, \mathcal{H}_1 \vdash \mathtt{c_a} \oplus \mathtt{c_b} : t \backslash \mathcal{C}}\,(\text{Sub})$$

$$\mathbf{TD}_a = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{c_a} : t_a \backslash \mathcal{C}_a}\,(\text{Sub})$$

$$\mathbf{TD}_b = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{c_b} : t_b \backslash \mathcal{C}_b}\,(\text{Sub})$$

$$\mathbf{TD}_2 = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H}_1 \vdash v : \langle\, pc_1, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\,(\text{Const})}{\Gamma, pc_1, \mathcal{H}_1 \vdash v : t \backslash \mathcal{C}}\,(\text{Sub})$$

Again by (Op-R), $\mathtt{c}'_\mathtt{a} \oplus \mathtt{c}'_\mathtt{b}, \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow v', \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$ where $v' = \mathtt{c}'_\mathtt{a} \oplus \mathtt{c}'_\mathtt{b}$ and $\mathbf{TD}'_2 = tdop(\mathbf{TD}'_1, v', \mathcal{C}'_\top)$, and the following.

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\mathbf{TD}'_a \qquad \mathbf{TD}'_b}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{c}'_\mathtt{a} \oplus \mathtt{c}'_\mathtt{b} \langle\, s'_a \cup s'_b, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}'_a \cup \mathcal{C}'_b}\,(\text{Op})}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \mathtt{c}'_\mathtt{a} \oplus \mathtt{c}'_\mathtt{b} : t' \backslash \mathcal{C}'}\,(\text{Sub})$$

$$\mathbf{TD}'_a = \dfrac{\cdots}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{c}'_\mathtt{a} : t'_a \backslash \mathcal{C}'_a}\,(\text{Sub})$$

$$\mathbf{TD}'_b = \dfrac{\cdots}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{c}'_\mathtt{b} : t'_b \backslash \mathcal{C}'_b}\,(\text{Sub})$$

$$\mathbf{TD}'_2 = \dfrac{\dfrac{}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash v' : \langle\, pc'_1, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\,(\text{Const})}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash v' : t' \backslash \mathcal{C}'}\,(\text{Sub})$$

By Definition 4.16, we have $Con(s, \mathcal{C}_\top) \subseteq \mathtt{Low}$. Also, by assumption and Definition 4.15[2c], we have $t = t'$ and $\mathcal{C}_\top = \mathcal{C}'_\top$, so $Con(s', \mathcal{C}'_\top) \subseteq \mathtt{Low}$.

By (Sub), and since $\mathcal{C} \subseteq \mathcal{C}_\top$, we have $\{s_a <: s\} \in \mathcal{C}_\top$ and $\{s_b <: s\} \in \mathcal{C}_\top$. Hence, by Lemma 4.20, $Con(s_a, \mathcal{C}_\top) \subseteq Con(s, \mathcal{C}_\top)$ and $Con(s_b, \mathcal{C}_\top) \subseteq Con(s, \mathcal{C}_\top)$. Since $Con(s, \mathcal{C}_\top) \subseteq \mathtt{Low}$, we have $Con(s_a, \mathcal{C}_\top) \subseteq \mathtt{Low}$ and $Con(s_b, \mathcal{C}_\top) \subseteq \mathtt{Low}$.

Similarly, by (Sub), and since $\mathcal{C}' \subseteq \mathcal{C}'_\top$, we have $\{s'_a <: s'\} \in \mathcal{C}_\top$ and $\{s'_b <: s'\} \in \mathcal{C}'_\top$. Hence, by Lemma 4.20, $Con(s'_a, \mathcal{C}_\top) \subseteq Con(s', \mathcal{C}'_\top)$ and $Con(s'_b, \mathcal{C}'_\top) \subseteq Con(s', \mathcal{C}'_\top)$. Since

$Con(s', \mathcal{C}'_\top) \subseteq \texttt{Low}$, we have $Con(s'_a, \mathcal{C}'_\top) \subseteq \texttt{Low}$ and

$Con(s'_b, \mathcal{C}'_\top) \subseteq \texttt{Low}$.

Now, by Definition 4.15[2c], we have $\texttt{c}_\texttt{a}, \mathbf{TD}_a, \mathcal{C}_\top \simeq_{\texttt{Low}} \texttt{c}'_\texttt{a}, \mathbf{TD}'_a, \mathcal{C}'_\top$, and $\texttt{c}_\texttt{b}, \mathbf{TD}_b, \mathcal{C}_\top \simeq_{\texttt{Low}} \texttt{c}'_\texttt{b}, \mathbf{TD}'_b, \mathcal{C}'_\top$, which by Definition 4.15[2a], entails the following. $Con(s_a, \mathcal{C}_\top), \texttt{c}_\texttt{a} \simeq_{\texttt{Low}} Con(s'_a, \mathcal{C}'_\top), \texttt{c}'_\texttt{a}$ and $Con(s_b, \mathcal{C}_\top), \texttt{c}_\texttt{b} \simeq_{\texttt{Low}} Con(s'_b, \mathcal{C}'_\top), \texttt{c}'_\texttt{b}$. Since $Con(s_a, \mathcal{C}_\top) \subseteq \texttt{Low}$ and $Con(s_b, \mathcal{C}_\top) \subseteq \texttt{Low}$, by Definition 4.15[1], we have $\texttt{c}_\texttt{a} = \texttt{c}'_\texttt{a}$ and $\texttt{c}_\texttt{b} = \texttt{c}'_\texttt{b}$, and so $v = v'$. Since $t = t'$ and $\mathcal{C}_\top = \mathcal{C}'_\top$, by Definition 4.15[1] $Con(s, \mathcal{C}_\top), v \simeq_{\texttt{Low}} Con(s', \mathcal{C}'_\top), v'$ Then, by Definition 4.15[2a], we have $v, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\texttt{Low}} v', \mathbf{TD}'_2, \mathcal{C}'_\top$. Conclude by assumption and Definition 4.15[8] $v, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} v', \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$.

3. (Cast-R)

Let $\varepsilon_1 = (\texttt{D})\ o$ and $\varepsilon'_1 = (\texttt{D})\ o'$.

By (Cast-R), $(\texttt{D})\ o, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow o, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1$, where $H_1(o) = \texttt{C}, F$ and $\texttt{C} <: \texttt{D}$ and $\mathbf{TD}_2 = tdcast(\mathbf{TD}_1, o, \mathcal{C}_\top)$

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_o}{\Gamma, pc_2, \mathcal{H}_1 \vdash (\texttt{D})\ o : t_o \backslash \mathcal{C}_o}\ \text{(Cast)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash (\texttt{D})\ o : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\cdots}{\Gamma, pc_1, \mathcal{H}_1 \vdash o : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Again by (Cast-R), $(\texttt{D})\ o', \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow o', \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$, where

$H'_1(o') = \texttt{C}, F'$ and $\texttt{C} <: \texttt{D}$ and $\mathbf{TD}'_2 = tdcast(\mathbf{TD}'_1, o', \mathcal{C}'_\top)$

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\mathbf{TD}'_o}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash (\texttt{D})\ o' : t'_o \backslash \mathcal{C}'_o}\ \text{(Cast)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash (\texttt{D})\ o' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$$\mathbf{TD}'_2 = \dfrac{\cdots}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash o' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

By Definition 4.16, we have $Con(s, \mathcal{C}_\top) \subseteq \texttt{Low}$. By assumption and Definition 4.15[2d], we have $t = t'$ and $\mathcal{C}_\top = \mathcal{C}'_\top$, so $Con(s', \mathcal{C}'_\top) \subseteq \texttt{Low}$.

By (Sub), and since $\mathcal{C} \subseteq \mathcal{C}_\top$, we have $\{s_o <: s\} \in \mathcal{C}_\top$. Hence, by Lemma 4.20, $Con(s_o, \mathcal{C}_\top) \subseteq Con(s, \mathcal{C}_\top)$. Since $Con(s, \mathcal{C}_\top) \subseteq \texttt{Low}$, we have $Con(s_o, \mathcal{C}_\top) \subseteq \texttt{Low}$.

Similarly, by (Sub), and since $\mathcal{C}' \subseteq \mathcal{C}'_\top$, we have $\{s'_o <: s'\} \in \mathcal{C}_\top$. Hence, by Lemma 4.20, $Con(s'_o, \mathcal{C}_\top) \subseteq Con(s', \mathcal{C}'_\top)$. Since $Con(s', \mathcal{C}'_\top) \subseteq \texttt{Low}$, we have $Con(s'_o, \mathcal{C}'_\top) \subseteq \texttt{Low}$.

Now, by Definition 4.15[2d], we have $o, \mathbf{TD}_o, \mathcal{C}_\top \simeq_{\texttt{Low}} o' \mathbf{TD}'_o, \mathcal{C}'_\top$, which by Definition 4.15[2a], entails $Con(s_o, \mathcal{C}_\top), o \simeq_{\texttt{Low}} Con(s'_o, \mathcal{C}'_\top), o'$. Since $Con(s_o, \mathcal{C}_\top) \subseteq \texttt{Low}$, by Definition 4.15[1], we have $o = o'$. Since $t = t'$ and $\mathcal{C}_\top = \mathcal{C}'_\top$, by Definition 4.15[1] $Con(s, \mathcal{C}_\top), o \simeq_{\texttt{Low}} Con(s', \mathcal{C}'_\top), o'$ Then, by Definition 4.15[2a], we have $o, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\texttt{Low}} o', \mathbf{TD}'_2, \mathcal{C}'_\top$. Conclude by assumption and Definition 4.15[8] $o, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} o', \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$.

4. (IfTrue-R)

Let $\varepsilon_1 = \texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\}$, and $\varepsilon'_1 = \texttt{if}\ (v)\ \{\overline{\texttt{s}'_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}'_\texttt{f}}\}$.

By (IfTrue-R), we have

$\texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\}, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \{\overline{\texttt{s}_\texttt{t}}\}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1$

and $\mathbf{TD}_2 = tdiftrue(\mathbf{TD}_1, \{\overline{\texttt{s}_\texttt{t}}\}, \mathcal{C}_\top)$, and the following.

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_c \quad \mathbf{TD}_t \quad \mathbf{TD}_f}{\Gamma, pc_2, \mathcal{H}_1 \vdash \texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\} : t_i \backslash \mathcal{C}_i}\ \text{(If)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_t = \dfrac{\dfrac{\cdots}{\Gamma, pc_3, \mathcal{H}_1 \vdash \{\overline{\texttt{s}_\texttt{t}}\} : \tau_t \backslash \mathcal{C}_a}\ \text{(Block)}}{\Gamma, pc_2 \cup s_c, \mathcal{H}_1 \vdash \{\overline{\texttt{s}_\texttt{t}}\} : t_t \backslash \mathcal{C}_t}\ \text{(Sub)}$$

$$\mathbf{TD}_f = \dfrac{\cdots}{\Gamma, pc_2 \cup s_c, \mathcal{H}_1 \vdash \{\overline{\texttt{s}_\texttt{f}}\} : t_f \backslash \mathcal{C}_f}\ \text{(Sub)}$$

$\mathbf{TD}_c =$

$$\frac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \texttt{True} : t_c \backslash \mathcal{C}_c}\ \text{(Sub)}$$

$\mathbf{TD}_2 =$

$$\frac{\dfrac{\cdots}{\Gamma, pc_3, \mathcal{H}_1 \vdash \{\overline{\texttt{s}_\texttt{t}}\} : \tau_t \backslash \mathcal{C}_a}\ \text{(Block)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \{\overline{\texttt{s}_\texttt{t}}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Again, by (IfTrue-R), we have

$\texttt{if}\ (v)\ \{\overline{\texttt{s}'_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}'_\texttt{f}}\}, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \ldots$. We use $\ldots$, since it is currently unclear whether the then or else branch will be taken, based on the value of v. We have the following.

$$\mathbf{TD}_1 = \frac{\dfrac{\mathbf{TD}'_c \qquad \mathbf{TD}'_t \qquad \mathbf{TD}'_f}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \texttt{if}\ (v)\ \{\overline{\texttt{s}'_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}'_\texttt{f}}\} : t'_i \backslash \mathcal{C}'_i}\ \text{(If)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \texttt{if}\ (v)\ \{\overline{\texttt{s}'_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}'_\texttt{f}}\} : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

By Definition 4.16, we have $Con(s_c, \mathcal{C}_\top) \subseteq \texttt{Low}$. By Definition 4.15[2e], $\texttt{True}, \mathbf{TD}_c, \mathcal{C}_\top \simeq_{\texttt{Low}} v, \mathbf{TD}'_c, \mathcal{C}_\top$, so $Con(s_c, \mathcal{C}_\top), \texttt{True} \simeq_{\texttt{Low}} Con(s'_c, \mathcal{C}'_\top), v$. Now, since $Con(s_c, \mathcal{C}_\top) \subseteq \texttt{Low}$, by Definition 4.15[1], $Con(s'_c, \mathcal{C}'_\top) \subseteq \texttt{Low}$ and $v = \texttt{True}$.

Therefore, by (IfTrue-R), we have the following.

$\texttt{if}\ (v)\ \{\overline{\texttt{s}'_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}'_\texttt{f}}\}, \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \{\overline{\texttt{s}'_\texttt{t}}\}, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$

and $\mathbf{TD}'_2 = tdiftrue(\mathbf{TD}'_1, \{\overline{\texttt{s}'_\texttt{t}}\}, \mathcal{C}'_\top)$, and the following.

$$\mathbf{TD}_1 = \frac{\dfrac{\mathbf{TD}'_c \qquad \mathbf{TD}'_t \qquad \mathbf{TD}'_f}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}'_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}'_\texttt{f}}\} : t'_i \backslash \mathcal{C}'_i}\ \text{(If)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}'_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}'_\texttt{f}}\} : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$\mathbf{TD}'_t =$

$$\frac{\dfrac{\cdots}{\Gamma', pc'_3, \mathcal{H}'_1 \vdash \{\overline{\texttt{s}'_\texttt{t}}\} : \tau'_t \backslash \mathcal{C}'_a}\ \text{(Block)}}{\Gamma', pc'_2 \cup s'_c, \mathcal{H}'_1 \vdash \{\overline{\texttt{s}'_\texttt{t}}\} : t'_t \backslash \mathcal{C}'_t}\ \text{(Sub)}$$

$\mathbf{TD}'_f =$

$$\frac{\cdots}{\Gamma', pc'_2 \cup s'_c, \mathcal{H}'_1 \vdash \{\overline{\texttt{s}'_\texttt{f}}\} : t'_f \backslash \mathcal{C}'_f}\ \text{(Sub)}$$

$\mathbf{TD}'_c =$

$$\frac{\cdots}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \texttt{True} : t'_c \backslash \mathcal{C}'_c}\ \text{(Sub)}$$

$$\mathbf{TD}'_2 = \dfrac{\dfrac{\cdots}{\Gamma', pc'_3, \mathcal{H}'_1 \vdash \{\overline{\mathtt{s}'_\mathtt{t}}\} : \tau'_t \backslash \mathcal{C}'_a}\ \text{(Block)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \{\overline{\mathtt{s}'_\mathtt{t}}\} : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

Now, by Definition 4.15[2e], we have $\{\overline{\mathtt{s}_\mathtt{t}}\}, \mathbf{TD}_t, \mathcal{C}_\top \simeq_{\mathtt{Low}} \{\overline{\mathtt{s}'_\mathtt{t}}\}, \mathbf{TD}'_t, \mathcal{C}'_\top$, and $t = t'$, and $\mathcal{C}_\top = \mathcal{C}'_\top$. Hence, by Definition 4.15[2h], $\{\overline{\mathtt{s}_\mathtt{t}}\}, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} \{\overline{\mathtt{s}'_\mathtt{t}}\}, \mathbf{TD}'_2, \mathcal{C}'_\top$. Conclude by assumption and Definition 4.15[8],

$\{\overline{\mathtt{s}_\mathtt{t}}\}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\mathtt{Low}} \{\overline{\mathtt{s}'_\mathtt{t}}\}, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$.

5. (IfFalse-R) follows similar to (IfTrue-R)

6. (Seq-R)

   By Definition 4.16, this cannot be a low step.

7. (Return-R)

   Let $\varepsilon_1 = \mathtt{return}\ v$ and $\varepsilon'_1 = \mathtt{return}\ v'$

   By (Return-R), $\mathtt{return}\ v; , \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow v, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1$,

   where $\mathbf{TD}_2 = tdreturn(\mathbf{TD}_1, v, \mathcal{C}_\top)$.

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_v}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{return}\ v : t_v \backslash \mathcal{C}_v}\ \text{(Return)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \mathtt{return}\ v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\cdots}{\Gamma, pc_1, \mathcal{H}_1 \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   Again by (Return-R), $\mathtt{return}\ v'; , \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow v', \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$,

   where $\mathbf{TD}'_2 = tdreturn(\mathbf{TD}'_1, v', \mathcal{C}'_\top)$.

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\mathbf{TD}'_v}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{return}\ v' : t'_v \backslash \mathcal{C}'_v}\ \text{(Return)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \mathtt{return}\ v' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$$\mathbf{TD}'_2 = \dfrac{\cdots}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash v' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

Now, by Definition 4.17, we have $Con(s, \mathcal{C}_\top) \subseteq \texttt{Low}$. By assumption and Definition 4.15[2g], we have $t = t'$ and $\mathcal{C}_\top = \mathcal{C}'_\top$, so $Con(s', \mathcal{C}'_\top) \subseteq \texttt{Low}$.

By Definition 4.15[2g], we also have $v, \mathbf{TD}_v, \mathcal{C}_\top \simeq_{\texttt{Low}} v', \mathbf{TD}'_v, \mathcal{C}'_\top$, and hence $Con(s_v, \mathcal{C}_\top), v \simeq_{\texttt{Low}} Con(s'_v, \mathcal{C}'_\top), v'$. By (Sub), we have $\{t_v <: t\}$ and $\{t'_v <: t'\}$, so by Lemma 4.20, we have $Con(s_v, \mathcal{C}_\top) \subseteq Con(s, \mathcal{C}_\top)$ and $Con(s'_v, \mathcal{C}'_\top) \subseteq Con(s', \mathcal{C}'_\top)$, so $Con(s_v, \mathcal{C}_\top) \subseteq \texttt{Low}$ and $Con(s'_v, \mathcal{C}'_\top) \subseteq \texttt{Low}$.

Since $Con(s_v, \mathcal{C}_\top), v \simeq_{\texttt{Low}} Con(s'_v, \mathcal{C}'_\top), v'$, by Definition 4.15[1], $v = v'$. Hence by Definition 4.15[1], $Con(s, \mathcal{C}_\top), v \simeq_{\texttt{Low}} Con(s, \mathcal{C}'_\top), v'$, so by Definition 4.15[2a], $v, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\texttt{Low}} v', \mathbf{TD}'_2, \mathcal{C}'_\top$.

Hence, by Definition 4.15[8], $v, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} v', \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$.

8. (Block-R)

   By Definition 4.16, this cannot be a low step.

9. (Assign-R)

   Let $\varepsilon_1 = o.\texttt{f} = v$ and $\varepsilon'_1 = o'.\texttt{f} = v'$.

   By (Assign-R), $o.\texttt{f} = v;, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow ;, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_1, \omega_1$, where $H_2 = H_1[o \mapsto \texttt{C}, F_2]$, $fields(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{f}}$ and $H(o) = \texttt{C}, F_1$ and $F_2 = F_1[\texttt{f} = v]$ and $\mathbf{TD}_2 = tdassign(\mathbf{TD}_1, \mathcal{C}_\top)$.

   and

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_o \qquad \mathbf{TD}_v}{\Gamma, pc_2, \mathcal{H}_1 \vdash o.\texttt{f} = v : \langle\, s_o \cup s_v, \emptyset, \texttt{void} \,\rangle \backslash \mathcal{C}_o \cup \mathcal{C}_v \cup \{f_o.\texttt{f} <: \textbf{set} \langle\, s_o \cup s_v, f_v, \alpha_v \,\rangle\}}\ \text{(F-Assign)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash o.\texttt{f} = v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   and

$$\mathbf{TD}_o = \dfrac{\dfrac{\mathcal{H}_1(o) = t_h \backslash \mathcal{C}_h}{\Gamma, pc_3, \mathcal{H}_1 \vdash o : \langle\, pc_3 \cup s_h, f_h, \alpha_h \,\rangle \backslash \mathcal{C}_h}\ \text{(Oid)}}{\Gamma, pc_2, \mathcal{H}_1 \vdash o : t_o \backslash \mathcal{C}_o}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H}_2 \vdash \,; : \langle\, pc_1, \emptyset, \texttt{void} \,\rangle \backslash \emptyset}\ \text{(No-op)}}{\Gamma, pc_1, \mathcal{H}_2 \vdash \,; : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$\mathcal{H}_2 = \mathcal{H}_1[o \mapsto t_h \backslash \mathcal{C}_a]$

where $\mathcal{C}_a = \mathcal{C}_h \cup \mathcal{C}_v \cup \{f_h.\texttt{f} <: \textbf{set}\ t_v\}]$

Again by (Assign-R), $o'.\texttt{f} = v'; , \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \,;, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_2, \iota'_1, \omega'_1$, where $H'_2 = H'_1[o' \mapsto \texttt{C}', F'_2]$, and $\mathit{fields}(\texttt{C'}) = \overline{\texttt{C}'}\ \overline{\texttt{f}}$ and $H'_1(o') = \texttt{C}', F'_1$ and $F'_2 = F'_1[\texttt{f} = v']$ and $\mathbf{TD}'_2 = \mathit{tdassign}(\mathbf{TD}'_1, \mathcal{C}'_\top)$.

and

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\mathbf{TD}'_o \qquad \mathbf{TD}'_v}{\begin{array}{c}\Gamma', pc'_2, \mathcal{H}'_1 \vdash o'.\texttt{f} = v' : \langle\, s'_o \cup s'_v, \emptyset, \texttt{void} \,\rangle \backslash \mathcal{C}'_o \cup \mathcal{C}'_v \cup \\ \{f'_o.\texttt{f} <: \textbf{set}\ \langle\, s'_o \cup s'_v, f'_v, \alpha'_v \,\rangle\}\end{array}}\ \text{(F-Assign)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash o'.\texttt{f} = v' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

and

$$\mathbf{TD}'_o = \dfrac{\dfrac{\mathcal{H}'_1(o) = t'_h \backslash \mathcal{C}'_h}{\Gamma', pc'_3, \mathcal{H}'_1 \vdash o' : \langle\, pc'_3 \cup s'_h, f'_h, \alpha'_h \,\rangle \backslash \mathcal{C}'_h}\ \text{(Oid)}}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash o' : t'_o \backslash \mathcal{C}'_o}\ \text{(Sub)}$$

$$\mathbf{TD}'_2 = \dfrac{\dfrac{}{\Gamma', pc'_1, \mathcal{H}'_2 \vdash \,; : \langle\, pc'_1, \emptyset, \texttt{void} \,\rangle \backslash \emptyset}\ \text{(No-op)}}{\Gamma', pc'_1, \mathcal{H}'_2 \vdash \,; : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$\mathcal{H}'_2 = \mathcal{H}'_1[o' \mapsto t'_h \backslash \mathcal{C}'_a]$

where $\mathcal{C}'_a = \mathcal{C}'_h \cup \mathcal{C}'_v \cup \{f'_h.\texttt{f} <: \textbf{set}\ t'_v\}]$

According to Definition 4.16, we have $\mathit{Con}(s_o, \mathcal{C}_\top) \subseteq \texttt{Low}$ and $\mathit{Con}(s_v, \mathcal{C}_\top) \subseteq \texttt{Low}$.

By Definition 4.15[2i,2a], we have $t_o = t'_o$, $t_v = t'_v$, $\mathcal{C}_\top = \mathcal{C}'_\top$, and $\mathit{Con}(s_o, \mathcal{C}_\top), o \simeq_{\texttt{Low}} \mathit{Con}(s'_v, \mathcal{C}'_\top), o$, and $\mathit{Con}(s_v, \mathcal{C}_\top), v \simeq_{\texttt{Low}} \mathit{Con}(s'_v, \mathcal{C}'_\top), v'$. Since $\mathit{Con}(s_o, \mathcal{C}_\top) \subseteq \texttt{Low}$ and $\mathit{Con}(s_v, \mathcal{C}_\top) \subseteq \texttt{Low}$, by Definition 4.15[1], we have $o = o'$ and $v = v'$.

By (Sub), we have $\{pc_3 \cup s_h <: s_o\} \in \mathcal{C}$, and since $\mathcal{C}$ is closed, by ($\mathcal{S}$-Union), $\{s_h <: s_o\} \in \mathcal{C}$. So, by Lemma 4.20, $Con(s_h, \mathcal{C}_\top) \subseteq Con(s_o, \mathcal{C}_\top)$, and since $Con(s_o, \mathcal{C}_\top) \subseteq \texttt{Low}$, we have $Con(s_h, \mathcal{C}_\top) \subseteq \texttt{Low}$. Similarly by (Sub), we have $\{pc_3' \cup s_h' <: s_o'\} \in \mathcal{C}'$, and since $\mathcal{C}'$ is closed, by ($\mathcal{S}$-Union), $\{s_h' <: s_o'\} \in \mathcal{C}'$. So, by Lemma 4.20, $Con(s_h', \mathcal{C}_\top') \subseteq Con(s_o', \mathcal{C}_\top')$, and since $Con(s_o', \mathcal{C}_\top') \subseteq \texttt{Low}$, we have $Con(s_h', \mathcal{C}_\top') \subseteq \texttt{Low}$.

Since $v = v'$, by Definition 4.15[1], $Con(f_h.\texttt{f}.\mathsf{S}, \mathcal{C}_\top), v \simeq_{\texttt{Low}} Con(f_h'.\texttt{f}.\mathsf{S}, \mathcal{C}_\top'), v'$, hence we conclude by Definition 4.15[3] $\mathcal{H}_2, \mathcal{C}_\top, H_2 \simeq_{\texttt{Low}} \mathcal{H}_2', \mathcal{C}_\top', H_2'$.

Since $t = t'$, by Definition 4.15[2p], $;, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\texttt{Low}} ;, \mathbf{TD}_2', \mathcal{C}_\top$, and since $\mathcal{H}_2, \mathcal{C}_\top, H_2 \simeq_{\texttt{Low}} \mathcal{H}_2', \mathcal{C}_\top', H_2'$, concluding with Definition 4.15[8],

$;, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_1, \omega_1 \simeq_{\texttt{Low}} ;, \mathbf{TD}_2', \mathcal{C}_\top, H_2', \iota_1', \omega_1'$.

10. (New-R) Let $\varepsilon_1 = \texttt{new C}(\overline{v})$ and $\varepsilon_1' = \texttt{new C}(\overline{v'})$.

By (New-R), let $\texttt{new C}(\overline{v}), \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_1, \omega_1$.

and

$$\mathbf{TD}_1 = \dfrac{\dfrac{\overline{\mathbf{TD}_v} \qquad \overline{\mathbf{TD}_n}}{\Gamma, pc_2, \mathcal{H}_1 \vdash \texttt{new C}(\overline{v}) : t_o \backslash \ldots}\ (\text{New})}{\Gamma, pc_1, \mathcal{H}_1 \vdash \texttt{new C}(\overline{v}) : t \backslash \mathcal{C}}\ (\text{Sub})$$

$$\overline{\mathbf{TD}_v} = \dfrac{\ldots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \overline{v} : \overline{t_v} \backslash \overline{\mathcal{C}_v}}\ (\text{Sub})$$

Again by (New-R), let $\texttt{new C}(\overline{v'}), \mathbf{TD}_1', \mathcal{C}_\top', H_1', \iota_1', \omega_1' \rightsquigarrow \varepsilon_2', \mathbf{TD}_2', \mathcal{C}_\top', H_2', \iota_1', \omega_1'$.

and

$$\mathbf{TD}_1' = \dfrac{\dfrac{\overline{\mathbf{TD}_v'} \qquad \overline{\mathbf{TD}_n'}}{\Gamma', pc_2', \mathcal{H}_1' \vdash \texttt{new C}(\overline{v'}) : t_o' \backslash \ldots}\ (\text{New})}{\Gamma', pc_1', \mathcal{H}_1' \vdash \texttt{new C}(\overline{v'}) : t' \backslash \mathcal{C}'}\ (\text{Sub})$$

$$\overline{\mathbf{TD}_v'} = \dfrac{\ldots}{\Gamma', pc_2', \mathcal{H}_1' \vdash \overline{v'} : \overline{t_v'} \backslash \overline{\mathcal{C}_v'}}\ (\text{Sub})$$

By assumption and Definition 4.15[2j,2a], we have $\mathcal{C}_\top = \mathcal{C}'_\top$, $t = t'$, $\overline{t_v} = \overline{t'_v}$, $t_m = t'_m$, and $t_o = t'_o$.

By Definition 4.15[2j,2a,1], we have two cases.

(a) $Con(s_o, \mathcal{C}_\top) \not\subseteq \texttt{Low}$

By Definition 4.16, this is not a low step, so this case cannot occur.

(b) $Con(s_o, \mathcal{C}_\top) \subseteq \texttt{Low}$

So, by Definition 4.15[2j,2a,1], $Con(s_o, \mathcal{C}_\top) = Con(s'_o, \mathcal{C}'_\top)$.

We first establish the bisimilarity of the new heaps.

By (New-R), we have $H_2 = H_1[o \mapsto \texttt{C}, F]$, and $\mathit{fields}(\texttt{C}) = \overline{\texttt{C f}}$, and $F = \overline{\texttt{f = null}}$, and $o = \mathit{newref}(H_1, S)$, where $S = Con(s_o, \mathcal{C}_\top)$. According to the definition of newref, we have $o = \mathit{loc}_i^S$.

Again by (New-R), we have $H'_2 = H'_1[o' \mapsto \texttt{C}, F']$, and $\mathit{fields}(\texttt{C}) = \overline{\texttt{C f}}$, and $F' = \overline{\texttt{f = null}}$, and $o' = \mathit{newref}(H'_1, S')$, where $S = Con(s'_o, \mathcal{C}'_\top)$. According to the definition of newref, we have $o' = \mathit{loc}_j^{S'}$.

Now, we have $s_o = s'_o$ and $\mathcal{C}_\top = \mathcal{C}'_\top$ from above, so by Definition 4.2, $S = S'$. Since by assumption $\mathcal{H}, \mathcal{C}_\top, H_1 \simeq_{\texttt{Low}} \mathcal{H}', \mathcal{C}'_\top, H'_1$, by Lemma 4.19, $o = o'$.

By (New-R), we have $\mathbf{TD}_2 = \mathit{tdnew}(\mathbf{TD}_1, \texttt{C}, \mathcal{C}_\top)$ and by the definition of tdnew, we have $\mathcal{H}_2 = \mathcal{H}_1[o \mapsto t_o \backslash \{\overline{f_o.\texttt{f} <: \textbf{set}\ t_n}\} \cup \overline{\mathcal{C}_n} \cup \{\texttt{C} <: \alpha_o\}]$. Again by (New-R), we have $\mathbf{TD}'_2 = \mathit{tdnew}(\mathbf{TD}'_1, \texttt{C}, \mathcal{C}'_\top)$ and by the definition of tdnew, we have $\mathcal{H}'_2 = \mathcal{H}'_1[o' \mapsto t'_o \backslash \{\overline{f'_o.\texttt{f} <: \textbf{set}\ t'_n}\} \cup \overline{\mathcal{C}'_n} \cup \{\texttt{C} <: \alpha'_o\}]$.

Now, we have the following. $H_2 = H_1[o \mapsto \texttt{C}, F]$ and $H'_2 = H'_1[o' \mapsto \texttt{C}, F']$, and $o = o'$, and $t_o = t'_o$, and $F = \overline{\texttt{f = null}}$ and $F' = \overline{\texttt{f = null}}$. Since $f_o = f'_o$ and $\mathcal{C}_\top = \mathcal{C}'_\top$, by Defi-

nition 4.15[1], we have $\overline{Con(f_o.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top), \mathtt{null}} \simeq_{\mathtt{Low}} \overline{Con(f'_o.\mathtt{f}.\mathsf{S}, \mathcal{C}'_\top), \mathtt{null}}$ (in either case, whether $Con(f_o.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top) \subseteq \mathtt{Low}$ or instead if $Con(f_o.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$, $Con(f_o.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top) = Con(f'_o.\mathtt{f}.\mathsf{S}, \mathcal{C}'_\top)$ and $\mathtt{null} = \mathtt{null}$).

Now, by assumption, $\mathcal{H}_1, \mathcal{C}_\top, H_1 \simeq_{\mathtt{Low}} \mathcal{H}'_1, \mathcal{C}'_\top, H'_1$, so by Definition 4.15[3],

$\mathcal{H}_2, \mathcal{C}_\top, H_2 \simeq_{\mathtt{Low}} \mathcal{H}'_2, \mathcal{C}'_\top, H'_2$.

We now establish the bisimularity of the resulting statements.

By (New-R), we have $\mathbf{TD}_2 = tdnew(\mathbf{TD}_1, \mathtt{C}, \mathcal{C}_\top)$, which gives the following.

$cnbody(\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{super}(\overline{\mathtt{e}}); \overline{\mathtt{s}})$ and

$\mathbf{TD}_m = TDT(\mathtt{C}, \mathtt{K})$

and $LT(\mathtt{C}, \mathtt{K}) = \forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}_r$ and $\overline{t_n} = \theta(\overline{t_l}, \mathtt{C}, \mathtt{K}, \overline{\alpha_v}, \alpha_o, \alpha_m)$

$$\mathbf{TD}_o = \dfrac{\dfrac{\mathcal{H}_2(o) = t_o \backslash \mathcal{C}_o}{\Gamma, pc_2 \cup s_o, \mathcal{H}_2 \vdash o : \langle\, pc_2 \cup s_o, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o}\ \text{(Oid)}}{\Gamma, pc_2 \cup s_o, \mathcal{H}_2 \vdash o : t \backslash \mathcal{C}}\ \text{(Sub)}$$

and $\mathbf{TD}_s = tdsub(\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}};, \overline{v}, o, [\overline{t_l} \mapsto \overline{t_n}], [s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m], \mathbf{TD}_m, tdheap(\overline{\mathbf{TD}_v}, \mathcal{H}_2), \mathbf{TD}_o)$

Again by (New-R), we have $\mathbf{TD}'_2 = tdnew(\mathbf{TD}'_1, \mathtt{C}, \mathcal{C}'_\top)$, which gives the following.

$cnbody(\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{super}(\overline{\mathtt{e}}); \overline{\mathtt{s}})$ and

$\mathbf{TD}_m = TDT(\mathtt{C}, \mathtt{K})$

and $LT(\mathtt{C}, \mathtt{K}) = \forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}_r$ and $\overline{t'_n} = \theta(\overline{t_l}, \mathtt{C}, \mathtt{K}, \overline{\alpha'_v}, \alpha'_o, \alpha'_m)$

$$\mathbf{TD}'_o = \dfrac{\dfrac{\mathcal{H}'_2(o') = t'_o \backslash \mathcal{C}'_o}{\Gamma', pc'_2 \cup s'_o, \mathcal{H}'_2 \vdash o' : \langle\, pc'_2 \cup s'_o, f'_o, \alpha'_o \,\rangle \backslash \mathcal{C}'_o}\ \text{(Heap)}}{\Gamma', pc'_2 \cup s'_o, \mathcal{H}'_2 \vdash o' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

and $\mathbf{TD}'_s = tdsub(\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}};, \overline{v'}, o', [\overline{t_l} \mapsto \overline{t'_n}], [s_p \mapsto pc'_2 \cup s'_o, \overline{t_x} \mapsto \overline{t'_v}, t_t \mapsto t'_o, t_r \mapsto t'_m], \mathbf{TD}_m, tdheap(\overline{\mathbf{TD}'_v}, \mathcal{H}'_2), \mathbf{TD}'_o)$.

By Definition 4.15[2j,2a] we have $\overline{t_v} = \overline{t'_v}$, $t_o = t'_o$, and $t_m = t'_m$.

So, we can re-write $\overline{t'_n} = \theta(\overline{t_l}, \mathtt{C}, \mathtt{K}, \overline{\alpha_v}, \alpha_o, \alpha_m)$, that is, $\overline{t'_n} = \overline{t_n}$. Hence, we can re-write $\mathbf{TD}'_s = tdsub(\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}};, \overline{v'}, o', [\overline{t_l} \mapsto \overline{t_n}], [s_p \mapsto pc'_2 \cup s'_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m], \mathbf{TD}_m, tdheap(\overline{\mathbf{TD}'_v}, \mathcal{H}'_2), \mathbf{TD}'_o)$.

So, by Lemma 4.22,

$[\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}};, \mathbf{TD}_s, \mathcal{C}_\top \simeq_{\mathtt{Low}}$

$[\overline{\mathtt{x}} \mapsto \overline{v'}, \mathtt{this} \mapsto o']\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}};, \mathbf{TD}'_s, \mathcal{C}'_\top$.

Now, by Definition 4.15[2j], we have $t = t'$; and as shown before, $Con(s, \mathcal{C}_\top) = Con(s', \mathcal{C}'_\top)$ and $o = o'$, hence by Definition 4.15[1],

$Con(s, \mathcal{C}_\top), o \simeq_{\mathtt{Low}} Con(s', \mathcal{C}'_\top), o'$ and by Definition 4.15[2a], $o, \mathbf{TD}_o, \mathcal{C}_\top \simeq_{\mathtt{Low}} o', \mathbf{TD}'_o, \mathcal{C}'_\top$.

Now, by (New-R), we have

$$\mathbf{TD}_{ret} = \dfrac{\dfrac{\mathbf{TD}_o}{\Gamma, pc_2 \cup s_o, \mathcal{H}_2 \vdash \mathtt{return}\ o; : t \backslash \mathcal{C}}\ \text{(Return)}}{\Gamma, pc_2 \cup s_o, \mathcal{H}_2 \vdash \mathtt{return}\ o; : t \backslash \mathcal{C}}\ \text{(Sub)}$$

and

$$\mathbf{TD}_2 = \dfrac{\dfrac{\mathbf{TD}_s \qquad \mathbf{TD}_{ret}}{\Gamma, pc_2 \cup s_o, \mathcal{H}_2 \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}}; \mathtt{return}\ o; : t \backslash \mathcal{C}}\ \text{(Seq)}}{\Gamma, pc_1, \mathcal{H}_2 \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}}; \mathtt{return}\ o; : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Again, by (New-R), we have

$$\mathbf{TD}_{ret'} = \dfrac{\dfrac{\mathbf{TD}_o}{\Gamma', pc'_2 \cup s'_o, \mathcal{H}'_2 \vdash \mathtt{return}\ o'; : t' \backslash \mathcal{C}'}\ \text{(Return)}}{\Gamma', pc'_2 \cup s'_o, \mathcal{H}'_2 \vdash \mathtt{return}\ o'; : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

and

$$\mathbf{TD}'_2 = \dfrac{\dfrac{\mathbf{TD}'_s \qquad \mathbf{TD}_{ret'}}{\Gamma', pc'_2 \cup s'_o, \mathcal{H}'_2 \vdash [\overline{\mathtt{x}} \mapsto \overline{v'}, \mathtt{this} \mapsto o']\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}}; \mathtt{return}\ o'; : t' \backslash \mathcal{C}'}\ \text{(Seq)}}{\Gamma', pc'_1, \mathcal{H}'_2 \vdash [\overline{\mathtt{x}} \mapsto \overline{v'}, \mathtt{this} \mapsto o']\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}}; \mathtt{return}\ o'; : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

Now, from above we have $[\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}};, \mathbf{TD}_s, \mathcal{C}_\top \simeq_{\mathtt{Low}} [\overline{\mathtt{x}} \mapsto \overline{v'}, \mathtt{this} \mapsto o']\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}};, \mathbf{TD}'_s, \mathcal{C}'_\top$, and

$o, \mathbf{TD}_o, \mathcal{C}_\top \simeq_{\mathtt{Low}} o', \mathbf{TD}'_o, \mathcal{C}'_\top$, and $t = t'$; Hence, by Definition 4.15[2f,2g],

$\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} \varepsilon'_2, \mathbf{TD}'_2, \mathcal{C}'_\top$. Above we showed $\mathcal{H}_2, \mathcal{C}_\top, H_2 \simeq_{\mathtt{Low}} \mathcal{H}'_2, \mathcal{C}'_\top, H'_2$. Therefore, by Definition 4.15[8] we conclude

$\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_1, \omega_1 \simeq_{\mathtt{Low}} \varepsilon'_2, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_2, \iota'_1, \omega'_1$.

11. (Invoke-R) Let $\varepsilon_1 = o.\mathtt{m}(\overline{v})$ and $\varepsilon'_1 = o'.\mathtt{m}(\overline{v'})$.

By (Invoke-R), let $o.\mathtt{m}(\overline{v}), \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1$.

and

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_o \qquad \overline{\mathbf{TD}_v}}{\Gamma, pc_2, \mathcal{H}_1 \vdash o.\mathtt{m}(\overline{v}) : t_m \backslash \ldots}\ \text{(Invoke)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash o.\mathtt{m}(\overline{v}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_o = \dfrac{\dfrac{\mathcal{H}_1(o) = t_h \backslash \mathcal{C}_h}{\Gamma, pc_3, \mathcal{H}_1 \vdash o : \langle\, pc_3 \cup s_h, f_h, \alpha_h \,\rangle \backslash \mathcal{C}_h}\ \text{(Oid)}}{\Gamma, pc_2, \mathcal{H}_1 \vdash o : t_o \backslash \mathcal{C}_o}\ \text{(Sub)}$$

$$\overline{\mathbf{TD}_v} = \dfrac{\ldots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \overline{v} : \overline{t_v} \backslash \overline{\mathcal{C}_v}}\ \text{(Sub)}$$

Again by (Invoke-R), let $o'.\mathtt{m}(\overline{v'}), \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \varepsilon'_2, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$.

and

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\mathbf{TD}'_o \qquad \overline{\mathbf{TD}'_v}}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash o'.\mathtt{m}(\overline{v'}) : t'_m \backslash \ldots}\ \text{(Invoke)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash o'.\mathtt{m}(\overline{v'}) : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$$\mathbf{TD}'_o = \dfrac{\dfrac{\mathcal{H}'_1(o') = t'_h \backslash \mathcal{C}'_h}{\Gamma', pc'_3, \mathcal{H}'_1 \vdash o' : \langle\, pc'_3 \cup s'_h, f'_h, \alpha'_h \,\rangle \backslash \mathcal{C}'_h}\ \text{(Oid)}}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash o' : t'_o \backslash \mathcal{C}'_o}\ \text{(Sub)}$$

$$\overline{\mathbf{TD}'_v} = \dfrac{\ldots}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \overline{v'} : \overline{t'_v} \backslash \overline{\mathcal{C}'_v}}\ \text{(Sub)}$$

By assumption and Definition 4.15[2k,2a,2a], we have $t = t'$, $t_o = t'_o$, $\overline{t_v} = \overline{t'_v}$, and $\mathcal{C}_\top = \mathcal{C}'_\top$.

By Definition 4.15[2k,2a,1], we have two cases.

(a) $Con(s_o, \mathcal{C}_\top) \not\subseteq \texttt{Low}$.

By Definition 4.16, this is not a low step, so this case cannot occur.

(b) $Con(s_o, \mathcal{C}_\top) \subseteq \texttt{Low}$.

So, by Definition 4.15[2k,2a,1], $Con(s_o, \mathcal{C}_\top) = Con(s'_o, \mathcal{C}'_\top)$ and $o = o'$.

Now, according to (Invoke-R), we have $H(o) = \texttt{C}, F$

Now, according to the type derivation $\mathbf{TD}_o$, and the use of the rule (Sub), we must have $\langle\, pc_3 \cup s_h, f_h, \alpha_h \,\rangle <: t_o \in \mathcal{C}_o$, which by closure rule ($\mathcal{S}$-Union) yields $\{s_h <: s_o\} \subseteq \mathcal{C}_o$. According to (Invoke-R), and $tdinvoke$, $\mathcal{C} \subseteq \mathcal{C}_\top$, and since $\mathbf{TD}_o$ is a sub-derivation of $\mathbf{TD}_1$, we have $\mathcal{C}_o \subseteq \mathcal{C}$, so $\mathcal{C}_o \subseteq \mathcal{C}_\top$, and thus $\{s_h <: s_o\} \subseteq \mathcal{C}_\top$. Now, $Con(s_o, \mathcal{C}_\top) \subseteq \texttt{Low}$, and by Lemma 4.20, $Con(s_h, \mathcal{C}_\top) \subseteq Con(s_o, \mathcal{C}_\top)$, so $Con(s_h, \mathcal{C}_\top) \subseteq \texttt{Low}$. Since $\mathcal{H}, \mathcal{C}_\top, H_1 \simeq_{\texttt{Low}} \mathcal{H}', \mathcal{C}'_\top, H'_1$ and $o = o'$, by Definition 4.15[3], $H(o') = \texttt{C}, F'$. (In other words, both o and o' point to C objects.)

By (Invoke-R), we have $\mathbf{TD}_2 = tdinvoke(\mathbf{TD}_1, \texttt{C}, \texttt{m}, \mathcal{C}_\top)$, and

$LT(\texttt{C}, \texttt{m}) = \forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}_r$, and $\overline{t_n} = \theta(\overline{t_l}, \texttt{C}, \texttt{m}, \overline{\alpha_v}, \alpha_o, \alpha_m)$, and

$\mathbf{TD}_s = tdsub(\overline{\texttt{s}}, \overline{v}, o, [\overline{t_l} \mapsto \overline{t_n}], [s_p \mapsto pc_2 \cup s_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m],$

$\mathbf{TD}_m, \overline{\mathbf{TD}_v}, \mathbf{TD}_o)$,

and by $tdinvoke(\mathbf{TD}_1, \texttt{C}, \texttt{m}, \mathcal{C}_\top)$, the constraint set from the final conclusion of $\mathbf{TD}_s$ is $\mathcal{C}_s$, where $\mathcal{C}_s \subseteq \mathcal{C}_\top$.

Again by (Invoke-R), we have $\mathbf{TD}'_2 = tdinvoke(\mathbf{TD}'_1, \texttt{C}, \texttt{m}, \mathcal{C}'_\top)$, and

$LT(\texttt{C}, \texttt{m}) = \forall \overline{t}.\overline{t_l}, \overline{t_x}, t_t \xrightarrow{s_p} t_r \backslash \mathcal{C}_r$, and $\overline{t'_n} = \theta(\overline{t_l}, \texttt{C}, \texttt{m}, \overline{\alpha'_v}, \alpha'_o, \alpha'_m)$, and $\mathbf{TD}'_s = tdsub(\overline{\texttt{s}}, \overline{v'}, o', [\overline{t_l} \mapsto \overline{t'_n}], [s_p \mapsto pc'_2 \cup s'_o, \overline{t_x} \mapsto \overline{t'_v}, t_t \mapsto t'_o, t_r \mapsto t'_m], \mathbf{TD}_m, \overline{\mathbf{TD}'_v}, \mathbf{TD}'_o)$,

and by $tdinvoke(\mathbf{TD}'_1, \mathtt{C}, \mathtt{m}, \mathcal{C}'_\top)$ the constraint set from the final conclusion of $\mathbf{TD}'_s$ is $\mathcal{C}'_s$, where $\mathcal{C}'_s \subseteq \mathcal{C}'_\top$.

By Definition 4.15[2k, 2a] we have $\overline{t} = \overline{t'}$, $t_o = t'_o$, and $t_m = t'_m$

So, we can re-write $\overline{t'_n} = \theta(\overline{t_l}, \mathtt{C}, \mathtt{m}, \overline{\alpha_v}, \alpha_o, \alpha_m)$, that is $\overline{t'_n} = \overline{t_n}$. Hence, we can re-write $\mathbf{TD}'_s = tdsub(\overline{\mathtt{s}}, \overline{v'}, o', [\overline{t_l} \mapsto \overline{t_n}], [s_p \mapsto pc'_2 \cup s'_o, \overline{t_x} \mapsto \overline{t_v}, t_t \mapsto t_o, t_r \mapsto t_m], \mathbf{TD}_m, \overline{\mathbf{TD}'_v}, \mathbf{TD}'_o)$

Now, by Lemma 4.22, $[\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}}, \mathbf{TD}_s, \mathcal{C}_\top \simeq_{\mathtt{Low}} [\overline{\mathtt{x}} \mapsto \overline{v'}, \mathtt{this} \mapsto o']\overline{\mathtt{s}}, \mathbf{TD}'_s, \mathcal{C}'_\top$.

So, we have

$$\mathbf{TD}_s = \dfrac{\dfrac{\mathbf{TD}_a \qquad \mathbf{TD}_b}{[\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}} : t_b \backslash \mathcal{C}_a \cup \mathcal{C}_b}\ \text{(Seq)}}{\Gamma, pc_2 \cup s_o, \mathcal{H}_1 \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}} : t_s \backslash \mathcal{C}_s}\ \text{(Sub)}$$

and by (Invoke-R),

$$\mathbf{TD}_2 = \dfrac{\dfrac{\mathbf{TD}_a \qquad \mathbf{TD}_b}{\Gamma, pc_2 \cup s_o, \mathcal{H}_1 \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}} : t_b \backslash \mathcal{C}_a \cup \mathcal{C}_b}\ \text{(Seq)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

We also have

$$\mathbf{TD}'_s = \dfrac{\dfrac{\mathbf{TD}'_a \qquad \mathbf{TD}'_b}{\Gamma', pc'_2 \cup s'_o, \mathcal{H}'_1 \vdash [\overline{\mathtt{x}} \mapsto \overline{v'}, \mathtt{this} \mapsto o']\overline{\mathtt{s}} : t'_b \backslash \mathcal{C}'_a \cup \mathcal{C}'_b}\ \text{(Seq)}}{\Gamma', pc'_2 \cup s'_o, \mathcal{H}'_1 \vdash [\overline{\mathtt{x}} \mapsto \overline{v'}, \mathtt{this} \mapsto o']\overline{\mathtt{s}} : t'_s \backslash \mathcal{C}'_s}\ \text{(Sub)}$$

and by (Invoke-R),

$$\mathbf{TD}'_2 = \dfrac{\dfrac{\mathbf{TD}'_a \qquad \mathbf{TD}'_b}{\Gamma', pc'_2 \cup s'_o, \mathcal{H}'_1 \vdash [\overline{\mathtt{x}} \mapsto \overline{v'}, \mathtt{this} \mapsto o']\overline{\mathtt{s}} : t'_b \backslash \mathcal{C}'_a \cup \mathcal{C}'_b}\ \text{(Seq)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash [\overline{\mathtt{x}} \mapsto \overline{v'}, \mathtt{this} \mapsto o']\overline{\mathtt{s}} : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

Now, since $[\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}}, \mathbf{TD}_s, \mathcal{C}_\top \simeq_{\mathtt{Low}} [\overline{\mathtt{x}} \mapsto \overline{v'}, \mathtt{this} \mapsto o']\overline{\mathtt{s}}, \mathbf{TD}'_s, \mathcal{C}'_\top$, we have $[\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}}, \mathbf{TD}_a, \mathcal{C}_\top \simeq_{\mathtt{Low}} [\overline{\mathtt{x}} \mapsto \overline{v'}, \mathtt{this} \mapsto o']\overline{\mathtt{s}}, \mathbf{TD}'_a, \mathcal{C}'_\top$, and $[\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}'}, \mathbf{TD}_b, \mathcal{C}_\top \simeq_{\mathtt{Low}} [\overline{\mathtt{x}} \mapsto \overline{v'}, \mathtt{this} \mapsto o']\overline{\mathtt{s}'}, \mathbf{TD}'_b, \mathcal{C}'_\top$. From above, we have $t = t'$, hence By Definition 4.15[2f], we have $[\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} [\overline{\mathtt{x}} \mapsto \overline{v'}, \mathtt{this} \mapsto o']\overline{\mathtt{s}}, \mathbf{TD}'_2, \mathcal{C}'_\top$. In other words,

$\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} \varepsilon_2', \mathbf{TD}_2', \mathcal{C}_\top'$, and therefore we can conclude

$\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\mathtt{Low}} \varepsilon_2', \mathbf{TD}_2', \mathcal{C}_\top', H_1', \iota_1', \omega_1'$.

12. (Super-R)

    Follows similarly to (New-R) and (Invoke-R).

13. (Input-R)

    Let $\varepsilon_1 = \mathtt{read_L(fd)}, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1$ and $\varepsilon_1' = \mathtt{read_L(fd')}, \mathbf{TD}_1', \mathcal{C}_\top', H_1', \iota_1', \omega_1'$

    By (Input-R)

    $\mathtt{read_L(fd)}, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \mathtt{c}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_2, \omega_1$

    where $\mathtt{L} = S_i$ and $\iota_1(\mathtt{fd}, S_i) = \mathtt{c}.\iota_2(\mathtt{fd}, S_i)$, and $S_f = \mathit{conread}(\mathbf{TD}_1, \mathcal{C}_\top)$ and $\mathbf{TD}_2 = \mathit{tdinput}(\mathbf{TD}_1, \mathtt{c}, \mathcal{C}_\top)$ and $S_f \subseteq \mathtt{L}$.

    and

$$\mathbf{TD}_1 = \dfrac{\dfrac{\dfrac{}{\Gamma, pc_3, \mathcal{H} \vdash \mathtt{fd} : \langle\, pc_3, \emptyset, \mathtt{int}\,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{fd} : t_f \backslash \mathcal{C}_f}\ \text{(Sub)}}{\dfrac{\Gamma, pc_2, \mathcal{H} \vdash \mathtt{read_L(fd)} : \langle\, s_f \cup \mathtt{L}, \emptyset, \mathtt{int}\,\rangle \backslash \mathcal{C}_f \cup SC(\mathtt{L}, s_f)}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{read_L(fd)} : t \backslash \mathcal{C}}\ \text{(Sub)}}\ \text{(Input)}$$

    and

$$\mathbf{TD}_2 = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{c} : \langle\, pc_1, \emptyset, \mathtt{int}\,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_1, \mathcal{H} \vdash \mathtt{c} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

    Again by (Input-R)

    $\mathtt{read_L(fd')}, \mathbf{TD}_1', \mathcal{C}_\top', H_1', \iota_1', \omega_1' \rightsquigarrow \mathtt{c'}, \mathbf{TD}_2', \mathcal{C}_\top', H_1', \iota_2', \omega_1'$

    where $\mathtt{L} = S_i$ and $\iota_1'(\mathtt{fd'}, S_i) = \mathtt{c'}.\iota_2'(\mathtt{fd'}, S_i)$, and $S_f' = \mathit{conread}(\mathbf{TD}_1', \mathcal{C}_\top')$ and $\mathbf{TD}_2' = \mathit{tdinput}(\mathbf{TD}_1', \mathtt{c'}, \mathcal{C}_\top')$ and $S_f' \subseteq \mathtt{L}$

    and

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\dfrac{}{\Gamma', pc'_3, \mathcal{H}' \vdash \mathtt{fd}' : \langle\, pc'_3, \emptyset, \mathtt{int}\,\rangle \backslash \emptyset}\,(\text{Const})}{\Gamma', pc'_2, \mathcal{H}' \vdash \mathtt{fd}' : t'_f \backslash \mathcal{C}'_f}\,(\text{Sub})}{\Gamma', pc'_2, \mathcal{H}' \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{fd}') : \langle\, s'_f \cup \mathtt{L}, \emptyset, \mathtt{int}\,\rangle \backslash \mathcal{C}'_f \cup SC(\mathtt{L}, s'_f)}\,(\text{Input})}{\Gamma', pc'_1, \mathcal{H}' \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{fd}') : t' \backslash \mathcal{C}'}\,(\text{Sub})$$

and

$$\mathbf{TD}'_2 = \dfrac{\dfrac{}{\Gamma', pc'_1, \mathcal{H}' \vdash \mathtt{c}' : \langle\, pc'_1, \emptyset, \mathtt{int}\,\rangle \backslash \emptyset}\,(\text{Const})}{\Gamma', pc'_1, \mathcal{H}' \vdash \mathtt{c}' : t' \backslash \mathcal{C}'}\,(\text{Sub})$$

We have two cases.

(a) $\mathtt{L} \not\subseteq \mathtt{Low}$

So by Definition 4.15[7], $\iota_1 \simeq_{\mathtt{Low}} \iota_2$ and $\iota'_1 \simeq_{\mathtt{Low}} \iota'_2$. So, by Lemma 4.18[2,3] $\iota_2 \simeq_{\mathtt{Low}} \iota'_2$.

By Definition of $conread$, we have $S_f = Con(s_f, \mathcal{C}_\top)$, and $S'_f = Con(s'_f, \mathcal{C}'_\top)$. By Definition 4.15[2n,2a], we have $t = t'$, $s_f = s'_f$, and $\mathcal{C}_\top = \mathcal{C}'_\top$, so $S_f = S'_f$,

Since $S_f \not\subseteq \mathtt{Low}$, and $S_f = Con(s_f, \mathcal{C}_\top)$, we have $Con(s_f, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$, which also yields $Con(s'_f, \mathcal{C}'_\top) \not\subseteq \mathtt{Low}$. By (Sub) and ($\mathcal{S}$-Union), we have $\{\mathtt{L} <: s\} \in \mathcal{C}_\top$. So, by Definition 4.2, $Con(s, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$, so $Con(s', \mathcal{C}'_\top) \not\subseteq \mathtt{Low}$. Hence, by Definition 4.15[1], $Con(s, \mathcal{C}_\top), \mathtt{c} \simeq_{\mathtt{Low}} Con(s', \mathcal{C}'_\top), \mathtt{c}'$, and since $t = t'$ and $\mathcal{C}_\top = \mathcal{C}'_\top$, by Definition 4.15[2a], $\mathtt{c}, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathtt{c}', \mathbf{TD}'_2, \mathcal{C}'_\top$. Since $\iota_2 \simeq_{\mathtt{Low}} \iota'_2$, conclude with Definition 4.15[8], $\mathtt{c}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_2, \omega_1 \simeq_{\mathtt{Low}} \mathtt{c}', \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_2, \omega_1$.

(b) $\mathtt{L} \subseteq \mathtt{Low}$

By Definition 4.16, $S_f \subseteq \mathtt{Low}$, and

By Definition of $conread$, we have $S_f = Con(s_f, \mathcal{C}_\top)$, and $S'_f = Con(s'_f, \mathcal{C}'_\top)$. By Definition 4.15[2n,2a], we have $t = t'$, $s_f = s'_f$, and $\mathcal{C}_\top = \mathcal{C}'_\top$, so $S_f = S'_f$,

Since $S_f \subseteq \mathtt{Low}$, and $S_f = Con(s_f, \mathcal{C}_\top)$, we have $Con(s_f, \mathcal{C}_\top) \subseteq \mathtt{Low}$, which also yields $Con(s'_f, \mathcal{C}'_\top) \subseteq \mathtt{Low}$. So, by Definition 4.15[1], $\mathtt{fd} = \mathtt{fd}'$.

Now, by Definition 4.15[7], $\iota_1(\texttt{fd}, \texttt{L}) = \iota_1'(\texttt{fd}', \texttt{L})$, which by Definition 4.15[5] means $\texttt{c}.\iota_2(\texttt{fd}, \texttt{L}) = \texttt{c}'.\iota_2'(\texttt{fd}', \texttt{L})$, and $\texttt{c} = \texttt{c}'$ and $\iota_2(\texttt{fd}, \texttt{L}) = \iota_2'(\texttt{fd}', \texttt{L})$. Hence, by Definition 4.15[7], $\iota_2 \simeq_{\texttt{Low}} \iota_2'$.

Now, since $\texttt{c} = \texttt{c}'$, by Definition 4.15[1], $Con(s, \mathcal{C}_\top), \texttt{c} \simeq_{\texttt{Low}} Con(s', \mathcal{C}_\top'), \texttt{c}'$, and since $t = t'$ and $\mathcal{C}_\top = \mathcal{C}_\top'$, by Definition 4.15[2a], $\texttt{c}, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\texttt{Low}} \texttt{c}', \mathbf{TD}_2', \mathcal{C}_\top'$. Since $\iota_2 \simeq_{\texttt{Low}} \iota_2'$, conclude with Definition 4.15[8], $\texttt{c}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_2, \omega_1 \simeq_{\texttt{Low}} \texttt{c}', \mathbf{TD}_2', \mathcal{C}_\top', H_1', \iota_2', \omega_1$.

14. (Output-R)

Let $\varepsilon_1 = \texttt{write}_\texttt{L}(\texttt{c}, \texttt{fd}), \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1$ and

$\varepsilon_1' = \texttt{write}_\texttt{L}(\texttt{c}', \texttt{fd}'), \mathbf{TD}_1', \mathcal{C}_\top', H_1', \iota_1', \omega_1'$

By (Output-R), $\texttt{write}_\texttt{L}(\texttt{c}, \texttt{fd}), \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow ;, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_2$,

where $\texttt{L} = S_i$, and $\omega_2(\texttt{fd}, S_i) = \texttt{c}.\omega_2(\texttt{fd}, S_i)$, and $S_f = conwrite(\mathbf{TD}_1, \mathcal{C}_\top)$, and $\mathbf{TD}_2 = tdoutput(\mathbf{TD}_1, \mathcal{C}_\top)$, and $S_f \subseteq \texttt{L}$.

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_c \qquad \mathbf{TD}_f}{\Gamma, pc_2, \mathcal{H}_1 \vdash \texttt{write}_\texttt{L}(\texttt{c}, \texttt{fd}) : \langle\, s_c \cup s_f, \emptyset, \texttt{void}\,\rangle \backslash \mathcal{C}_c \cup \mathcal{C}_f}\ \text{(Output)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \texttt{write}_\texttt{L}(\texttt{c}, \texttt{fd}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_c = \dfrac{\dfrac{}{\Gamma, pc_3, \mathcal{H}_1 \vdash \texttt{c} : \langle\, pc_3, \emptyset, \texttt{int}\,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_2, \mathcal{H}_1 \vdash \texttt{c} : t_c \backslash \mathcal{C}_c}\ \text{(Sub)}$$

$$\mathbf{TD}_f = \dfrac{\dfrac{}{\Gamma, pc_4, \mathcal{H}_1 \vdash \texttt{fd} : \langle\, pc_4, \emptyset, \texttt{int}\,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma, pc_2, \mathcal{H}_1 \vdash \texttt{fd} : t_f \backslash \mathcal{C}_f}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H}_1 \vdash ; : \langle\, pc_1, \emptyset, \texttt{void}\,\rangle \backslash \emptyset}\ \text{(No-op)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash ; : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Again by (Output-R), $\texttt{write}_\texttt{L}(\texttt{c}', \texttt{fd}'), \mathbf{TD}_1', \mathcal{C}_\top', H_1', \iota_1', \omega_1' \rightsquigarrow ;, \mathbf{TD}_2', \mathcal{C}_\top', H_1', \iota_1', \omega_2'$, where $\texttt{L} = S_i$, and $\omega_2'(\texttt{fd}', S_i) = \texttt{c}'.\omega_2'(\texttt{fd}', S_i)$, and $S_f' = conwrite(\mathbf{TD}_1', \mathcal{C}_\top')$, and $\mathbf{TD}_2' = tdoutput(\mathbf{TD}_1', \mathcal{C}_\top')$, and $S_f' \subseteq \texttt{L}$.

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\mathbf{TD}'_c \qquad \mathbf{TD}'_f}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{write_L}(\mathtt{c}', \mathtt{fd}') : \langle\, s'_c \cup s'_f, \emptyset, \mathtt{void}\,\rangle \backslash \mathcal{C}'_c \cup \mathcal{C}'_f}\ \text{(Output)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \mathtt{write_L}(\mathtt{c}', \mathtt{fd}') : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$$\mathbf{TD}'_c = \dfrac{\dfrac{}{\Gamma', pc'_3, \mathcal{H}'_1 \vdash \mathtt{c}' : \langle\, pc'_3, \emptyset, \mathtt{int}\,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{c}' : t'_c \backslash \mathcal{C}'_c}\ \text{(Sub)}$$

$$\mathbf{TD}'_f = \dfrac{\dfrac{}{\Gamma', pc'_4, \mathcal{H}'_1 \vdash \mathtt{fd}' : \langle\, pc'_4, \emptyset, \mathtt{int}\,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{fd}' : t'_f \backslash \mathcal{C}'_f}\ \text{(Sub)}$$

$$\mathbf{TD}'_2 = \dfrac{\dfrac{}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash ; : \langle\, pc'_1, \emptyset, \mathtt{void}\,\rangle \backslash \emptyset}\ \text{(No-op)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash ; : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

We have two cases.

(a) $\mathtt{L} \not\subseteq \mathtt{Low}$

So by Definition 4.15[6], $\omega_1 \simeq_{\mathtt{Low}} \omega_2$ and $\omega'_1 \simeq_{\mathtt{Low}} \omega'_2$. So, by Lemma 4.18[2,3] $\omega_2 \simeq_{\mathtt{Low}} \omega'_2$.

By Definition 4.15[2o], we have $t = t'$, and $\mathcal{C}_\top = \mathcal{C}'_\top$, so by Definition 4.15[2p], $;, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} ;, \mathbf{TD}'_2, \mathcal{C}'_\top$. Since $\omega_2 \simeq_{\mathtt{Low}} \omega'_2$, conclude with Definition 4.15[8], $;, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_2 \simeq_{\mathtt{Low}} ;, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega_2$.

(b) $\mathtt{L} \subseteq \mathtt{Low}$

By Definition 4.16, $S_f \subseteq \mathtt{Low}$, and

By Definition of *conwrite*, we have $S_f = Con(s_c, \mathcal{C}_\top) \cup Con(s_f, \mathcal{C}_\top)$, and $S'_f = Con(s'_c, \mathcal{C}'_\top) \cup Con(s'_f, \mathcal{C}'_\top)$. By Definition 4.15[2o,2a], we have $t = t'$, $s_c = s'_c$, $s_f = s'_f$, and $\mathcal{C}_\top = \mathcal{C}'_\top$, so $S_f = S'_f$.

Since $S_f \subseteq \mathtt{Low}$, and $S_f = Con(s_c, \mathcal{C}_\top) \cup Con(s_f, \mathcal{C}_\top)$, we have $Con(s_f, \mathcal{C}_\top) \subseteq \mathtt{Low}$, which also yields $Con(s'_f, \mathcal{C}'_\top) \subseteq \mathtt{Low}$. So, by Definition 4.15[1], $\mathtt{fd} = \mathtt{fd}'$.

By Definition 4.15[2o,2a], $\mathtt{c}, \mathbf{TD}_c, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathtt{c}', \mathbf{TD}'_c, \mathcal{C}'_\top$, so by Definition 4.15[1], we have $Con(s_c, \mathcal{C}_\top), \mathtt{c} \simeq_{\mathtt{Low}} Con(s'_c, \mathcal{C}'_\top), \mathtt{c}'$, and since $S_f \subseteq \mathtt{Low}$, and $S_f = Con(s_c, \mathcal{C}_\top) \cup Con(s_f, \mathcal{C}_\top)$, we have $Con(s_c, \mathcal{C}_\top) \subseteq \mathtt{Low}$, so $\mathtt{c} = \mathtt{c}'$.

Now, by Definition 4.15[6], $\omega_1(\mathtt{fd}, \mathtt{L}) = \omega'_1(\mathtt{fd}', \mathtt{L})$, and since $\mathtt{c} = \mathtt{c}'$ by Definition 4.15[4] $\mathtt{c}.\omega_1(\mathtt{fd}, \mathtt{L}) = \mathtt{c}.\omega'_1(\mathtt{fd}', \mathtt{L})$, which is $\omega_2(\mathtt{fd}, \mathtt{L}) = \omega'_2(\mathtt{fd}', \mathtt{L})$. Hence, by Definition 4.15[6], $\omega_2 \simeq_{\mathtt{Low}} \omega'_2$.

By Definition 4.15[2o], we have $t = t'$, and $\mathcal{C}_\top = \mathcal{C}'_\top$, so by Definition 4.15[2p], $;, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} ;, \mathbf{TD}'_2, \mathcal{C}'_\top$. Since $\omega_2 \simeq_{\mathtt{Low}} \omega'_2$, conclude with Definition 4.15[8], $;, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_2 \simeq_{\mathtt{Low}} ;, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega_2$.

15. (Op-RC)

Let $\varepsilon_1 = \mathtt{e}_a \oplus \mathtt{e}_c$ and $\varepsilon'_1 = \mathtt{e}'_a \oplus \mathtt{e}'_c$

By (Op-RC), $\mathtt{e}_a \oplus \mathtt{e}_c, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \mathtt{e}_b \oplus \mathtt{e}_c, \mathbf{TD}_2, \mathcal{C}_\top, H_b, \iota_b, \omega_b$, where

$\mathtt{e}_a, \mathbf{TD}_a, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \mathtt{e}_b, \mathbf{TD}_b, \mathcal{C}_\top, H_b, \iota_b, \omega_b$, and $\mathbf{TD}_a = ucon(\mathbf{TD}_1)$ and $\mathbf{TD}_2 = dconop(\mathbf{TD}_1, \mathbf{TD}_b, \mathcal{C}_\top)$

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_a \qquad \mathbf{TD}_c}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{e}_a \oplus \mathtt{e}_c : \langle\, s_a \cup s_c, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_a \cup \mathcal{C}_c}\ \text{(Op)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \mathtt{e}_a \oplus \mathtt{e}_b : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_a = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{e}_a : t_a \backslash \mathcal{C}_c}\ \text{(Sub)}$$

$$\mathbf{TD}_c = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{e}_c : t_c \backslash \mathcal{C}_c}\ \text{(Sub)}$$

$$\mathbf{TD}_b = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_b \vdash \mathtt{e}_b : t_a \backslash \mathcal{C}_a}\ \text{(Sub)}$$

$$\mathbf{TD}_d = tdheap(\mathbf{TD}_c, \mathcal{H}_b)$$

$$\mathbf{TD}_d = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_b \vdash \mathtt{e}_c : t_c \backslash \mathcal{C}_c}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\dfrac{\mathbf{TD}_b \qquad \mathbf{TD}_d}{\Gamma, pc_2, \mathcal{H}_b \vdash \mathtt{e}_b \oplus \mathtt{e}_c : \langle\, s_a \cup s_c, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_a \cup \mathcal{C}_c}\ \text{(Op)}}{\Gamma, pc_1, \mathcal{H}_b \vdash \mathtt{e}_b \oplus \mathtt{e}_c : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Again by (Op-RC), $\mathtt{e}'_a \oplus \mathtt{e}'_c, \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \mathtt{e}'_b \oplus \mathtt{e}'_c, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_b, \iota'_b, \omega'_b$, where $\mathtt{e}'_a, \mathbf{TD}'_a, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \mathtt{e}'_b, \mathbf{TD}'_b, \mathcal{C}'_\top, H'_b, \iota'_b, \omega'_b$, and $\mathbf{TD}'_a = ucon(\mathbf{TD}'_1)$ and $\mathbf{TD}'_2 = dconop(\mathbf{TD}'_1, \mathbf{TD}'_b, \mathcal{C}'_\top)$

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\mathbf{TD}'_a \qquad \mathbf{TD}'_c}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{e}'_a \oplus \mathtt{e}'_c : \langle\, s'_a \cup s'_c, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}'_a \cup \mathcal{C}'_c}\ \text{(Op)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \mathtt{e}'_a \oplus \mathtt{e}'_b : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$$\mathbf{TD}'_a = \dfrac{\cdots}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{e}'_a : t'_a \backslash \mathcal{C}'_c}\ \text{(Sub)}$$

$$\mathbf{TD}'_c = \dfrac{\cdots}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{e}'_c : t'_c \backslash \mathcal{C}'_c}\ \text{(Sub)}$$

$$\mathbf{TD}'_b = \dfrac{\cdots}{\Gamma', pc'_2, \mathcal{H}'_b \vdash \mathtt{e}'_b : t'_a \backslash \mathcal{C}'_a}\ \text{(Sub)}$$

$$\mathbf{TD}'_d = tdheap(\mathbf{TD}'_c, \mathcal{H}'_b)$$

$$\mathbf{TD}'_d = \dfrac{\cdots}{\Gamma', pc'_2, \mathcal{H}'_b \vdash \mathtt{e}'_c : t'_c \backslash \mathcal{C}'_c}\ \text{(Sub)}$$

$$\mathbf{TD}'_2 = \dfrac{\dfrac{\mathbf{TD}'_b \qquad \mathbf{TD}'_d}{\Gamma', pc'_2, \mathcal{H}'_b \vdash \mathtt{e}'_b \oplus \mathtt{e}'_c : \langle\, s'_a \cup s'_c, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}'_a \cup \mathcal{C}'_c}\ \text{(Op)}}{\Gamma', pc'_1, \mathcal{H}'_b \vdash \mathtt{e}'_b \oplus \mathtt{e}'_c : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

Since $\mathtt{e}_a \oplus \mathtt{e}_c, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_l \mathtt{e}_b \oplus \mathtt{e}_c, \mathbf{TD}_2, \mathcal{C}_\top, H_b, \iota_b, \omega_b$, by Definition 4.16, $\mathtt{e}_a, \mathbf{TD}_a, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_l \mathtt{e}_b, \mathbf{TD}_b, \mathcal{C}_\top, H_b, \iota_b, \omega_b$ (i.e. the context reduction is a low step). Hence, by induction $\mathtt{e}'_a, \mathbf{TD}'_a, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow_l \mathtt{e}'_b, \mathbf{TD}'_b, \mathcal{C}'_\top, H'_b, \iota'_b, \omega'_b$, and $\mathtt{e}_b, \mathbf{TD}_b, \mathcal{C}_\top, H_b, \iota_b, \omega_b \simeq_{\mathtt{Low}} \mathtt{e}'_b, \mathbf{TD}'_b, \mathcal{C}'_\top, H'_b, \iota'_b, \omega'_b$.

So, by Definition 4.16, $\mathtt{e}'_a \oplus \mathtt{e}'_c, \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow_l \mathtt{e}'_b \oplus \mathtt{e}'_c, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_b, \iota'_b, \omega'_b$.

By Lemma 4.21, $\mathsf{e}_c, \mathbf{TD}_d, \mathcal{C}_\top \simeq_{\texttt{Low}} \mathsf{e}_c, \mathbf{TD}_c, \mathcal{C}_\top$ and $\mathsf{e}'_c, \mathbf{TD}'_d, \mathcal{C}'_\top \simeq_{\texttt{Low}} \mathsf{e}'_c, \mathbf{TD}'_c, \mathcal{C}'_\top$; so by Definition 4.15[2c,8], we have

$\mathsf{e}_b \oplus \mathsf{e}_c, \mathbf{TD}_2, \mathcal{C}_\top, H_b, \iota_b, \omega_b \simeq_{\texttt{Low}} \mathsf{e}'_b \oplus \mathsf{e}'_c, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_b, \iota'_b, \omega'_b$.

16. Remaining (*-RC) cases follow a similar inductive argument to (Op-RC).

□

Before proving the lemma for taking high steps, we first prove two necessary lemmas. Lemma 4.24 shows that a configuration with a type derivation that contains a high program counter at the penultimate typing, will take only high steps. This is unequivocal, since the program counter can only increase in size up the derivation tree. Lemma 4.25 shows that high steps will not change any low heap locations or low streams. These lemmas are both necessary in Lemma 4.26 for proving that bisimilarity is maintained across branching computations. For example, two executions that branch on a high value may take different branches. Since the branching value is high, both branches take only high steps, according to Lemma 4.24, and Lemma 4.25 shows that when the branches complete, their configurations will be bisimilar.

Lemma 4.24 (Deeply High Computation)

If $\varepsilon_1, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1$ terminates, and the penultimate type rule of $\mathbf{TD}$ concludes with $\Gamma_1, pc, H_1 \vdash \varepsilon_1 : \tau \backslash \mathcal{C}$, and $s_o \subseteq pc$, and $Con(s_o, \mathcal{C}_\top) \not\subseteq \texttt{Low}$,then there exists v, $\mathbf{TD}_2$, $\mathcal{C}_\top$, H_2, ι_2, and ω_2, such that $\varepsilon_1, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_h^* v, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2$, and the derivation $\mathbf{TD}_2$ ends with $\Gamma_2, pc_2, H_2 \vdash v : t \backslash \mathcal{C}$, and $Con(s, \mathcal{C}_\top) \not\subseteq \texttt{Low}$.

Proof. By induction on $\rightsquigarrow_h^*$ and the structure of $\varepsilon, \mathbf{TD}, \mathcal{C}_\top$ using Definition 4.17.

Since the penultimate steps all contain a *high* program counter, observing the type rules, this program counter will propagate to all program counters up the derivation tree (program counters can only increase going up the tree). Furthermore, due to sub-typing, in all cases we will have $\{s_o <: s\} \subseteq \mathcal{C}_\top$, making the final type conclusion *high* for the cases which require it. Hence, in every case, based on Definition 4.17, the step will be *high*.

□

Lemma 4.25 (Bisimulation of High Computation)

1. *(one-step) If* $\varepsilon_1, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_h \varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2$*, and* $\mathcal{H}_1$ *and* $\mathcal{H}_2$ *are respective to* $\mathbf{TD}_1$ *and* $\mathbf{TD}_2$*, then* $\mathcal{H}_1, \mathcal{C}_\top, H_1 \simeq_{\mathtt{Low}} \mathcal{H}_2, \mathcal{C}_\top, H_2$*,* $\iota_1 \simeq_{\mathtt{Low}} \iota_2$*, and* $\omega_1 \simeq_{\mathtt{Low}} \omega_2$*.*
2. *(n-steps) If* $\varepsilon_1, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_h^* \varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2$*, and* $\mathcal{H}_1$ *and* $\mathcal{H}_2$ *are respective to* $\mathbf{TD}_1$ *and* $\mathbf{TD}_2$*, then* $\mathcal{H}_1, \mathcal{C}_\top, H_1 \simeq_{\mathtt{Low}} \mathcal{H}_2, \mathcal{C}_\top, H_2$*,* $\iota_1 \simeq_{\mathtt{Low}} \iota_2$*, and* $\omega_1 \simeq_{\mathtt{Low}} \omega_2$*.*

Proof.

1. By induction on the derivation of $\varepsilon_1, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_h \varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2$, using Definition 4.15, and Definition 4.17.

   A *high* step by definition can only alter *high* heap locations, and *high* input and output streams. Hence, taking a *high* step will not alter anything that is *low*, and maintain a bisimularity relationship of the heaps and streams after the reduction step.

2. By induction on the length of $\rightsquigarrow_h^*$, using Lemma 4.25[1].

□

Lemma 4.26 shows that for two bisimilar expressions, a series (possibly different in number) of high steps – where the high steps complete by either reaching a value, or being followed by a

low step – result in bisimilar expressions. As in the low reduction lemma, the heaps and input and output streams all remain bisimilar.

Lemma 4.26 (Reduction of High Security Configurations)

If $\varepsilon_1, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\mathtt{Low}} \varepsilon_1', \mathbf{TD}_1', \mathcal{C}_\top', H_1', \iota_1', \omega_1'$, *and* $\varepsilon_1, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_h^* \varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2$, *and either* ε_2 *is a value, or*

$\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2 \rightsquigarrow_l \varepsilon_3, \mathbf{TD}_3, \mathcal{C}_\top, H_3, \iota_3, \omega_3$, *for some* $\varepsilon_3, \mathbf{TD}_3, H_3, \iota_3$, *and* ω_3, *and assume* $\varepsilon_1, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1$ *and* $\varepsilon_1', \mathbf{TD}_1', \mathcal{C}_\top', H_1', \iota_1', \omega_1'$ *both terminate, and*

$\mathbf{TD}_1 \rhd \varepsilon_1, H_1, \Gamma, pc_1, t, \mathcal{C}$, *and* $\mathbf{TD}_1' \rhd \varepsilon_1', H_1', \Gamma', pc_1', t', \mathcal{C}'$; *then* $\varepsilon_1', \mathbf{TD}_1', \mathcal{C}_\top' H_1', \iota_1', \omega_1' \rightsquigarrow_h^* \varepsilon_2', \mathbf{TD}_2', \mathcal{C}_\top', H_2', \iota_2', \omega_2'$, *and* $\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2 \simeq_{\mathtt{Low}} \varepsilon_2', \mathbf{TD}_2', \mathcal{C}_\top', H_2', \iota_2', \omega_2'$.

Proof. By induction on length of the reduction $\rightsquigarrow_h^*$, and the height of the reduction derivation tree.

Base Case (reflexive): $\varepsilon_1 = \varepsilon_2$, $\mathbf{TD}_1 = \mathbf{TD}_2$, $H_1 = H_2$, $\iota_1 = \iota_2$, and $\omega_1 = \omega_2$. Then, by reflexivity of $\rightsquigarrow^*$, $\varepsilon_1' = \varepsilon_2'$, $\mathbf{TD}_1' = \mathbf{TD}_2'$, $H_1' = H_2'$, $\iota_1' = \iota_2'$, and $\omega_1' = \omega_2'$. The lemma follows by assumption, $\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2 \simeq_{\mathtt{Low}} \varepsilon_2', \mathbf{TD}_2', \mathcal{C}_\top', H_2', \iota_2', \omega_2'$.

Inductive Case: $\varepsilon_1, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_h \varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow_h^* \varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2$.

By induction on the context derivation tree of $\varepsilon_1, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_h \varepsilon, \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega$, with case analysis on the last (bottom) reduction rule used.

1. (Field-R)

   Let $\varepsilon_1 = o.\mathtt{f}$ and $\varepsilon_1' = o'.\mathtt{f}$.

   By (Field-R), $o.\mathtt{f}, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow v, \mathbf{TD}_2, H_1, \iota_1, \omega_1$

   where $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$, and $H_1(o) = \mathtt{C}, F$, and $F(\mathtt{f}) = v$

   $\mathbf{TD}_1 =$

   $$\dfrac{\cdots}{\Gamma, pc_1, \mathcal{H}_1 \vdash o.\mathtt{f} : t \backslash \mathcal{C}}\ (\text{Sub})$$

$\mathbf{TD}_2 =$

$$\frac{\cdots}{\Gamma, pc_1, \mathcal{H}_1 \vdash v : t \backslash \mathcal{C}} \text{ (Sub)}$$

Again by (Field-R), $o'.\mathtt{f}, \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow v', \mathbf{TD}'_2, H'_1, \iota'_1, \omega'_1$

where $fields(\mathtt{C}') = \overline{\mathtt{C}'}\ \overline{\mathtt{f}}$, and $H'_1(o') = \mathtt{C}', F'$, and $F(\mathtt{f}') = v'$

$\mathbf{TD}'_1 =$

$$\frac{\cdots}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash o'.\mathtt{f} : t' \backslash \mathcal{C}'} \text{ (Sub)}$$

$\mathbf{TD}'_2 =$

$$\frac{\cdots}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash v' : t' \backslash \mathcal{C}'} \text{ (Sub)}$$

Now, by Definition 4.17, we have $Con(s, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$. By assumption and Definition 4.15[2b], we have $t = t'$ and $\mathcal{C}_\top = \mathcal{C}'_\top$, so $Con(s', \mathcal{C}'_\top) \not\subseteq \mathtt{Low}$. So, by Definition 4.15[2a], $v, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} v', \mathbf{TD}'_2, \mathcal{C}'_\top$. Hence, by Definition 4.15[8], $v, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\mathtt{Low}} v', \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$.

2. (Op-R)

Let $\varepsilon_1 = \mathtt{c_a} \oplus \mathtt{c_b}$ and $\varepsilon'_1 = \mathtt{c'_a} \oplus \mathtt{c'_b}$

By (Op-R), $\mathtt{c_a} \oplus \mathtt{c_b}, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow v, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1$ where $v = \mathtt{c_a} \oplus \mathtt{c_b}$ and $\mathbf{TD}_2 = tdop(\mathbf{TD}_1, v, \mathcal{C}_\top)$, and the following.

$\mathbf{TD}_1 =$

$$\frac{\dfrac{\mathbf{TD}_a \qquad \mathbf{TD}_b}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{c_a} \oplus \mathtt{c_b} \langle\, s_a \cup s_b, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_a \cup \mathcal{C}_b} \text{ (Op)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \mathtt{c_a} \oplus \mathtt{c_b} : t \backslash \mathcal{C}} \text{ (Sub)}$$

$\mathbf{TD}_a =$

$$\frac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{c_a} : t_a \backslash \mathcal{C}_a} \text{ (Sub)}$$

$\mathbf{TD}_b =$

$$\frac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{c_b} : t_b \backslash \mathcal{C}_b} \text{ (Sub)}$$

$\mathbf{TD}_2 =$

$$\frac{\dfrac{}{\Gamma, pc_1, \mathcal{H}_1 \vdash v : \langle\, pc_1, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset} \text{ (Const)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash v : t \backslash \mathcal{C}} \text{ (Sub)}$$

Again by (Op-R), $\mathtt{c}'_\mathtt{a} \oplus \mathtt{c}'_\mathtt{b}, \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow v', \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$ where $v' = \mathtt{c}'_\mathtt{a} \oplus \mathtt{c}'_\mathtt{b}$ and $\mathbf{TD}'_2 = tdop(\mathbf{TD}'_1, v', \mathcal{C}'_\top)$, and the following.

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\mathbf{TD}'_a \qquad \mathbf{TD}'_b}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{c}'_\mathtt{a} \oplus \mathtt{c}'_\mathtt{b} \langle\, s'_a \cup s'_b, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}'_a \cup \mathcal{C}'_b}\ \text{(Op)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \mathtt{c}'_\mathtt{a} \oplus \mathtt{c}'_\mathtt{b} : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$$\mathbf{TD}'_a = \dfrac{\cdots}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{c}'_\mathtt{a} : t'_a \backslash \mathcal{C}'_a}\ \text{(Sub)}$$

$$\mathbf{TD}'_b = \dfrac{\cdots}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{c}'_\mathtt{b} : t'_b \backslash \mathcal{C}'_b}\ \text{(Sub)}$$

$$\mathbf{TD}'_2 = \dfrac{\dfrac{}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash v' : \langle\, pc'_1, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash v' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

By Definition 4.16, we have $Con(s, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$. By assumption and Definition 4.15[2c], we have $t = t'$ and $\mathcal{C}_\top = \mathcal{C}'_\top$, so $Con(s', \mathcal{C}'_\top) \subseteq \mathtt{Low}$.

Hence, by Definition 4.15[1], $Con(s, \mathcal{C}_\top), v \simeq_\mathtt{Low} Con(s', \mathcal{C}'_\top), v'$. Since $t = t'$ and $\mathcal{C}_\top = \mathcal{C}'_\top$, by Definition 4.15[2a] $v, \mathbf{TD}_2, \mathcal{C}_\top \simeq_\mathtt{Low} v', \mathbf{TD}'_2, \mathcal{C}'_\top$. Conclude by assumption and Definition 4.15[8], $v, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_\mathtt{Low} v', \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$.

3. (Cast-R)

Let $\varepsilon_1 = (\mathtt{D})\ o$ and $\varepsilon'_1 = (\mathtt{D})\ o'$.

By (Cast-R), $(\mathtt{D})\ o, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow o, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1$, where $H_1(o) = \mathtt{C}, F$ and $\mathtt{C} <: \mathtt{D}$ and $\mathbf{TD}_2 = tdcast(\mathbf{TD}_1, o, \mathcal{C}_\top)$

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_o}{\Gamma, pc_2, \mathcal{H}_1 \vdash (\mathtt{D})\ o : t_o \backslash \mathcal{C}_o}\ \text{(Cast)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash (\mathtt{D})\ o : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\cdots}{\Gamma, pc_1, \mathcal{H}_1 \vdash o : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Again by (Cast-R), $(\texttt{D})\ o', \mathbf{TD}_1', \mathcal{C}_\top', H_1', \iota_1', \omega_1' \rightsquigarrow o', \mathbf{TD}_2', \mathcal{C}_\top', H_1', \iota_1', \omega_1'$, where

$H_1'(o') = \texttt{C}, F'$ and $\texttt{C} <: \texttt{D}$ and $\mathbf{TD}_2' = tdcast(\mathbf{TD}_1', o', \mathcal{C}_\top')$

$$\mathbf{TD}_1' = \dfrac{\dfrac{\mathbf{TD}_o'}{\Gamma', pc_2', \mathcal{H}_1' \vdash (\texttt{D})\ o' : t_o' \backslash \mathcal{C}_o'}\ \text{(Cast)}}{\Gamma', pc_1', \mathcal{H}_1' \vdash (\texttt{D})\ o' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$$\mathbf{TD}_2' = \dfrac{\cdots}{\Gamma', pc_1', \mathcal{H}_1' \vdash o' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

By Definition 4.17, we have $Con(s, \mathcal{C}_\top) \not\subseteq \texttt{Low}$. By assumption and Definition 4.15[2d], we have $t = t'$ and $\mathcal{C}_\top = \mathcal{C}_\top'$, so $Con(s', \mathcal{C}_\top') \not\subseteq \texttt{Low}$.

Hence, by Definition 4.15[1], $Con(s, \mathcal{C}_\top), o \simeq_{\texttt{Low}} Con(s', \mathcal{C}_\top'), o'$. Since $t = t'$ and $\mathcal{C}_\top = \mathcal{C}_\top'$, by Definition 4.15[2a] $o, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\texttt{Low}} o', \mathbf{TD}_2', \mathcal{C}_\top'$. Conclude by assumption and Definition 4.15[8], $o, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} o', \mathbf{TD}_2', \mathcal{C}_\top', H_1', \iota_1', \omega_1'$.

4. (IfTrue-R)

Let $\varepsilon_1 = \texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\}$, and $\varepsilon_1' = \texttt{if}\ (v)\ \{\overline{\texttt{s}_\texttt{t}'}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}'}\}$.

By (IfTrue-R), we have

$\texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\}, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \{\overline{\texttt{s}_\texttt{t}}\}, \mathbf{TD}_3, \mathcal{C}_\top, H_1, \iota_1, \omega_1$

and $\mathbf{TD}_3 = tdiftrue(\mathbf{TD}_1, \{\overline{\texttt{s}_\texttt{t}}\}, \mathcal{C}_\top)$, and the following.

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_c \quad \mathbf{TD}_t \quad \mathbf{TD}_f}{\Gamma, pc_2, \mathcal{H}_1 \vdash \texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\} : t_i \backslash \mathcal{C}_i}\ \text{(If)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \texttt{if}\ (\texttt{True})\ \{\overline{\texttt{s}_\texttt{t}}\}\ \texttt{else}\ \{\overline{\texttt{s}_\texttt{f}}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_t = \dfrac{\dfrac{\cdots}{\Gamma, pc_3, \mathcal{H}_1 \vdash \{\overline{\texttt{s}_\texttt{t}}\} : \tau_t \backslash \mathcal{C}_a}\ \text{(Block)}}{\Gamma, pc_2 \cup s_c, \mathcal{H}_1 \vdash \{\overline{\texttt{s}_\texttt{t}}\} : t_t \backslash \mathcal{C}_t}\ \text{(Sub)}$$

$$\mathbf{TD}_f = \dfrac{\cdots}{\Gamma, pc_2 \cup s_c, \mathcal{H}_1 \vdash \{\overline{\texttt{s}_\texttt{f}}\} : t_f \backslash \mathcal{C}_f}\ \text{(Sub)}$$

$$\mathbf{TD}_c = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \texttt{True} : t_c \backslash \mathcal{C}_c}\ \text{(Sub)}$$

$$\mathbf{TD}_3 = \dfrac{\dfrac{\cdots}{\Gamma, pc_3, \mathcal{H}_1 \vdash \{\overline{\mathtt{s_t}}\} : \tau_t \backslash \mathcal{C}_a}\ \text{(Block)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \{\overline{\mathtt{s_t}}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Now, depending on the value of v, we have two cases. We examine the case where $v = \texttt{False}$, and the case where $v = \texttt{True}$ follows similarly.

By (IfFalse-R), we have

$\texttt{if}\ (v)\ \{\overline{\mathtt{s'_t}}\}\ \texttt{else}\ \{\overline{\mathtt{s'_f}}\}, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \{\overline{\mathtt{s'_f}}\}, \mathbf{TD}'_3, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$, and

$\mathbf{TD}'_3 = \mathit{tdiffalse}(\mathbf{TD}'_1, \{\overline{\mathtt{s'_f}}\}, \mathcal{C}'_\top)$, and the following.

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}'_c \quad \mathbf{TD}'_t \quad \mathbf{TD}'_f}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \texttt{if}\ (v)\ \{\overline{\mathtt{s'_t}}\}\ \texttt{else}\ \{\overline{\mathtt{s'_f}}\} : t'_i \backslash \mathcal{C}'_i}\ \text{(If)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \texttt{if}\ (v)\ \{\overline{\mathtt{s'_t}}\}\ \texttt{else}\ \{\overline{\mathtt{s'_f}}\} : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}'_c \quad \mathbf{TD}'_t \quad \mathbf{TD}'_f}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \texttt{if}\ (\texttt{False})\ \{\overline{\mathtt{s'_t}}\}\ \texttt{else}\ \{\overline{\mathtt{s'_f}}\} : t'_i \backslash \mathcal{C}'_i}\ \text{(If)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \texttt{if}\ (\texttt{False})\ \{\overline{\mathtt{s'_t}}\}\ \texttt{else}\ \{\overline{\mathtt{s'_f}}\} :: t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$$\mathbf{TD}'_t = \dfrac{\dfrac{\cdots}{\Gamma', pc'_3, \mathcal{H}'_1 \vdash \{\overline{\mathtt{s'_t}}\} : \tau'_t \backslash \mathcal{C}'_a}\ \text{(Block)}}{\Gamma', pc'_2 \cup s'_c, \mathcal{H}'_1 \vdash \{\overline{\mathtt{s'_t}}\} : t'_t \backslash \mathcal{C}'_t}\ \text{(Sub)}$$

$$\mathbf{TD}'_f = \dfrac{\dfrac{\cdots}{\Gamma', pc'_3, \mathcal{H}'_1 \vdash \{\overline{\mathtt{s'_f}}\} : \tau'_f \backslash \mathcal{C}'_b}\ \text{(Block)}}{\Gamma', pc'_2 \cup s'_c, \mathcal{H}'_1 \vdash \{\overline{\mathtt{s'_f}}\} : t'_f \backslash \mathcal{C}'_f}\ \text{(Sub)}$$

$$\mathbf{TD}'_c = \dfrac{\cdots}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \texttt{False} : t'_c \backslash \mathcal{C}'_c}\ \text{(Sub)}$$

$$\mathbf{TD}'_3 = \dfrac{\dfrac{\cdots}{\Gamma', pc'_3, \mathcal{H}'_1 \vdash \{\overline{\mathtt{s'_f}}\} : \tau'_f \backslash \mathcal{C}'_b}\ \text{(Block)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \{\overline{\mathtt{s'_f}}\} : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

By Definition 4.17, we have $Con(s_c, \mathcal{C}_\top) \not\subseteq \texttt{Low}$. Hence by Definition 4.15[2e, 2a] and assumption, $Con(s'_c, \mathcal{C}'_\top) \not\subseteq \texttt{Low}$.

By assumption, $\{\overline{\mathtt{s_t}}\}, \mathbf{TD}_3, \mathcal{C}_\top, H_1, \iota_1, \omega_1$ terminates, and we have just shown $Con(s_c, \mathcal{C}_\top) \not\subseteq \texttt{Low}$. According to $\mathbf{TD}_3 = \mathit{tdiftrue}(\mathbf{TD}_1, \{\overline{\mathtt{s_t}}\}, \mathcal{C}_\top)$ we know $\mathbf{TD}_3$ has the penultimate typing

$\Gamma, pc_3, \mathcal{H}_1 \vdash \{\overline{\mathtt{s_t}}\} : \ldots$, and observing $\mathbf{TD}_t$ above, by (Sub), we have $s_c \subseteq pc_3$. In other words, the type variable s_c is contained in the program counter of the penultimate typing in $\mathbf{TD}_3$. Hence by Lemma 4.24, $\{\overline{\mathtt{s_t}}\}, \mathbf{TD}_3, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_h^* \varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2$, where $\varepsilon_2 = v_2$ for some v_2, and

$$\mathbf{TD}_2 = \dfrac{\ldots}{\Gamma, pc_1, \mathcal{H}_2 \vdash v_2 : t_2 \backslash \mathcal{C}_2}\ \text{(Sub)}$$

Now, by Lemma 4.25[2], $\mathcal{H}_1, \mathcal{C}_\top, H_1 \simeq_{\mathtt{Low}} \mathcal{H}_2, \mathcal{C}_\top, H_2$, and $\iota_1 \simeq_{\mathtt{Low}} \iota_2$, and $\omega_1 \simeq_{\mathtt{Low}} \omega_2$.

Similarly, by assumption, $\{\overline{\mathtt{s'_f}}\}, \mathbf{TD}'_3, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$ terminates, and we have just shown that $Con(s'_c, \mathcal{C}'_\top) \not\subseteq \mathtt{Low}$. According to $\mathbf{TD}'_3 = tdiffalse(\mathbf{TD}'_1, \{\overline{\mathtt{s'_f}}\}, \mathcal{C}'_\top)$ we know $\mathbf{TD}'_3$ has the penultimate typing $\Gamma', pc'_3, \mathcal{H}'_1 \vdash \{\overline{\mathtt{s'_f}}\} : \ldots$, and observing $\mathbf{TD}'_f$ above, by (Sub), we have $s'_c \subseteq pc'_3$. In other words, the type variable s'_c is contained in the program counter of the penultimate typing in $\mathbf{TD}'_3$. Hence by Lemma 4.24, $\{\overline{\mathtt{s'_f}}\}, \mathbf{TD}'_3, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow_h^* \varepsilon'_2, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_2, \iota'_2, \omega'_2$, where $\varepsilon'_2 = v'_2$ for some v'_2, and

$$\mathbf{TD}'_2 = \dfrac{\ldots}{\Gamma', pc'_1, \mathcal{H}'_2 \vdash v'_2 : t'_2 \backslash \mathcal{C}'_2}\ \text{(Sub)}$$

Now, by Lemma 4.25[2], $\mathcal{H}'_1, \mathcal{C}'_\top, H'_1 \simeq_{\mathtt{Low}} \mathcal{H}'_2, \mathcal{C}'_\top, H'_2$, and $\iota'_1 \simeq_{\mathtt{Low}} \iota'_2$, and $\omega'_1 \simeq_{\mathtt{Low}} \omega'_2$.

Now, by Subject Reduction Lemma 4.13, the final type of $\mathbf{TD}_2$ must be the same as that of $\mathbf{TD}_1$, and the final type of $\mathbf{TD}'_2$ must be the same as that of $\mathbf{TD}'_1$. Hence $t_2 = t$ and $t'_2 = t'$

By (If) and (Sub), we have $\{s_c <: s_i\} \cup \{s_i <: s\} \in \mathcal{C}_\top$, so since $\mathcal{C}_\top$ is closed, $\{s_c <: s\} \in \mathcal{C}_\top$, so by Lemma 4.20, $Con(s_c, \mathcal{C}_\top) \subseteq Con(s, \mathcal{C}_\top)$. Since $Con(s_c, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$, we have $Con(s, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$. Similarly, by (If) and (Sub), we have $\{s'_c <: s'_i\} \cup \{s'_i <: s'\} \in \mathcal{C}'_\top$, so since $\mathcal{C}'_\top$ is closed, $\{s'_c <: s'\} \in \mathcal{C}'_\top$, so by Lemma 4.20, $Con(s'_c, \mathcal{C}'_\top) \subseteq Con(s', \mathcal{C}'_\top)$. Since $Con(s'_c, \mathcal{C}'_\top) \not\subseteq \mathtt{Low}$, we have $Con(s', \mathcal{C}'_\top) \not\subseteq \mathtt{Low}$.

Hence, by Definition 4.15[2a,1], $v_2, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} v_2', \mathbf{TD}_2', \mathcal{C}_\top'$, which is the same as $\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} \varepsilon_2', \mathbf{TD}_2', \mathcal{C}_\top'$.

By Lemma 4.18[2,3], we have $\mathcal{H}_2, \mathcal{C}_\top, H_2 \simeq_{\mathtt{Low}} \mathcal{H}_2', \mathcal{C}_\top', H_2'$, and $\iota_2 \simeq_{\mathtt{Low}} \iota_2'$ and $\omega_2 \simeq_{\mathtt{Low}} \omega_2'$.

Conclude by Definition 4.15[8], $\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2 \simeq_{\mathtt{Low}} \varepsilon_2', \mathbf{TD}_2', \mathcal{C}_\top', H_2', \iota_2', \omega_2'$.

5. (IfFalse-R) follows similar to (IfTrue-R)

6. (Seq-R)

   Let $\varepsilon_1 = ;\overline{\mathtt{s}}$ and $\varepsilon_1' = ;\overline{\mathtt{s}'}$.

   By (Seq-R), we have $; \overline{\mathtt{s}}, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1$, where $\mathbf{TD}_2 = tdseq(\mathbf{TD}_1, \overline{\mathtt{s}}, \mathcal{C}_\top)$

   $$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_n \qquad \mathbf{TD}_s}{\Gamma, pc_2, \mathcal{H}_1 \vdash ;\overline{\mathtt{s}} : t_s \backslash \mathcal{C}_s}\ (\text{Seq})}{\Gamma, pc_1, \mathcal{H}_1 \vdash ;\overline{\mathtt{s}} : t \backslash \mathcal{C}}\ (\text{Sub})$$

   $$\mathbf{TD}_2 = \dfrac{\cdots}{\Gamma, pc_1, \mathcal{H}_1 \vdash \overline{\mathtt{s}} : t \backslash \mathcal{C}}\ (\text{Sub})$$

   Again by (Seq-R), we have $; \overline{\mathtt{s}'}, \mathbf{TD}_1', \mathcal{C}_\top', H_1', \iota_1', \omega_1' \rightsquigarrow \overline{\mathtt{s}'}, \mathbf{TD}_2', \mathcal{C}_\top', H_1', \iota_1', \omega_1'$, where $\mathbf{TD}_2' = tdseq(\mathbf{TD}_1', \overline{\mathtt{s}'}, \mathcal{C}_\top')$

   $$\mathbf{TD}_1' = \dfrac{\dfrac{\mathbf{TD}_n' \qquad \mathbf{TD}_s'}{\Gamma', pc_2', \mathcal{H}_1' \vdash ;\overline{\mathtt{s}'} : t_s' \backslash \mathcal{C}_s'}\ (\text{Seq})}{\Gamma', pc_1', \mathcal{H}_1' \vdash ;\overline{\mathtt{s}'} : t' \backslash \mathcal{C}'}\ (\text{Sub})$$

   $$\mathbf{TD}_2' = \dfrac{\cdots}{\Gamma', pc_1', \mathcal{H}_1' \vdash \overline{\mathtt{s}'} : t' \backslash \mathcal{C}'}\ (\text{Sub})$$

   By Definition 4.15[2f], we have $t = t'$, and $\mathcal{C}_\top = \mathcal{C}_\top'$, and $\overline{\mathtt{s}}, \mathbf{TD}_s, \mathcal{C}_\top \simeq_{\mathtt{Low}} \overline{\mathtt{s}'}, \mathbf{TD}_s', \mathcal{C}_\top'$. Hence by Definition 4.15[2f], we have $\overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} \overline{\mathtt{s}'}, \mathbf{TD}_2', \mathcal{C}_\top'$, and by Definition 4.15[8], $\overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\mathtt{Low}} \overline{\mathtt{s}'}, \mathbf{TD}_2', \mathcal{C}_\top', M_1', \iota_1', \omega_1'$.

We now have two cases, according to Definition 4.17 and Definition 4.16 (the next step must either be high or low).

(a) $\overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_l \varepsilon_3, \mathbf{TD}_3, \mathcal{C}_\top, H_3, \iota_3, \omega_3$.

Conclude with $\overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\mathtt{Low}} \overline{\mathtt{s}'}, \mathbf{TD}'_2, \mathcal{C}'_\top, M'_1, \iota'_1, \omega'_1$.

(b) $\overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_h^* \varepsilon_3, \mathbf{TD}_3, \mathcal{C}_\top, H_3, \iota_3, \omega_3$.

Since $\overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\mathtt{Low}} \overline{\mathtt{s}'}, \mathbf{TD}'_2, \mathcal{C}'_\top, M'_1, \iota'_1, \omega'_1$, by induction

$\overline{\mathtt{s}'}, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow_h^* \varepsilon'_3, \mathbf{TD}'_3, \mathcal{C}'_\top, H'_3, \iota'_3, \omega'_3$, and

$\varepsilon_3, \mathbf{TD}_3, \mathcal{C}_\top, H_3, \iota_3, \omega_3 \simeq_{\mathtt{Low}} \varepsilon'_3, \mathbf{TD}'_3, \mathcal{C}'_\top, M'_3, \iota'_3, \omega'_3$.

7. (Return-R)

Let $\varepsilon_1 = \mathtt{return}\ v$ and $\varepsilon'_1 = \mathtt{return}\ v'$

By (Return-R), $\mathtt{return}\ v; , \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow v, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1$,

where $\mathbf{TD}_2 = tdreturn(\mathbf{TD}_1, v, \mathcal{C}_\top)$.

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_v}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{return}\ v : t_v \backslash \mathcal{C}_v}\ \text{(Return)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \mathtt{return}\ v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\cdots}{\Gamma, pc_1, \mathcal{H}_1 \vdash v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Again by (Return-R), $\mathtt{return}\ v'; , \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow v', \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$,

where $\mathbf{TD}'_2 = tdreturn(\mathbf{TD}'_1, v', \mathcal{C}'_\top)$.

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\mathbf{TD}'_v}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{return}\ v' : t'_v \backslash \mathcal{C}'_v}\ \text{(Return)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \mathtt{return}\ v' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$$\mathbf{TD}'_2 = \dfrac{\cdots}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash v' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

Now, by Definition 4.17, we have $Con(s, \mathcal{C}_\top) \not\subseteq \texttt{Low}$. By assumption and Definition 4.15[2g], we have $t = t'$ and $\mathcal{C}_\top = \mathcal{C}'_\top$, so $Con(s', \mathcal{C}'_\top) \not\subseteq \texttt{Low}$. So, by Definition 4.15[2a], $v, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\texttt{Low}} v', \mathbf{TD}'_2, \mathcal{C}'_\top$. Hence, by Definition 4.15[8], $v, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} v', \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$.

8. (Block-R)

   Let $\varepsilon_1 = \{\overline{\mathtt{s}}\}$ and $\varepsilon'_1 = \{\overline{\mathtt{s}'}\}$.

   By (Block-R), we have $\{\overline{\mathtt{s}}\}, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1$, where $\mathbf{TD}_2 = tdblock(\mathbf{TD}_1, \overline{\mathtt{s}}, \mathcal{C}_\top)$.

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_s}{\Gamma, pc_2, \mathcal{H}_1 \vdash \{\overline{\mathtt{s}}\} : t_s \backslash \mathcal{C}_s}\ \text{(Block)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \{\overline{\mathtt{s}}\} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\cdots}{\Gamma, pc_1, \mathcal{H}_1 \vdash \overline{\mathtt{s}} : t \backslash \mathcal{C}}\ \text{(Sub)}$$

   Again by (Block-R), we have $\{\overline{\mathtt{s}'}\}, \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \overline{\mathtt{s}'}, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$, where $\mathbf{TD}'_2 = tdblock(\mathbf{TD}'_1, \overline{\mathtt{s}'}, \mathcal{C}'_\top)$.

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\mathbf{TD}'_s}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \{\overline{\mathtt{s}'}\} : t'_s \backslash \mathcal{C}'_s}\ \text{(Block)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \{\overline{\mathtt{s}'}\} : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$$\mathbf{TD}'_2 = \dfrac{\cdots}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \overline{\mathtt{s}'} : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

   By Definition 4.15[2h], we have $t = t'$, and $\mathcal{C}_\top = \mathcal{C}'_\top$, and $\overline{\mathtt{s}}, \mathbf{TD}_s, \mathcal{C}_\top \simeq_{\texttt{Low}} \overline{\mathtt{s}'}, \mathbf{TD}'_s, \mathcal{C}'_\top$. Hence by Definition 4.15[2f], we have $\overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\texttt{Low}} \overline{\mathtt{s}'}, \mathbf{TD}'_2, \mathcal{C}'_\top$, and by Definition 4.15[8], $\overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} \overline{\mathtt{s}'}, \mathbf{TD}'_2, \mathcal{C}'_\top, M'_1, \iota'_1, \omega'_1$.

   We now have two cases, according to Definition 4.17 and Definition 4.16 (the next step must either be high or low).

(a) $\overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_l \varepsilon_3, \mathbf{TD}_3, \mathcal{C}_\top, H_3, \iota_3, \omega_3$.

Conclude with $\overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\mathtt{Low}} \overline{\mathtt{s}'}, \mathbf{TD}'_2, \mathcal{C}'_\top, M'_1, \iota'_1, \omega'_1$.

(b) $\overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow^*_h \varepsilon_3, \mathbf{TD}_3, \mathcal{C}_\top, H_3, \iota_3, \omega_3$.

Since $\overline{\mathtt{s}}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \simeq_{\mathtt{Low}} \overline{\mathtt{s}'}, \mathbf{TD}'_2, \mathcal{C}'_\top, M'_1, \iota'_1, \omega'_1$, by induction

$\overline{\mathtt{s}'}, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow^*_h \varepsilon'_3, \mathbf{TD}'_3, \mathcal{C}'_\top, H'_3, \iota'_3, \omega'_3$, and

$\varepsilon_3, \mathbf{TD}_3, \mathcal{C}_\top, H_3, \iota_3, \omega_3 \simeq_{\mathtt{Low}} \varepsilon'_3, \mathbf{TD}'_3, \mathcal{C}'_\top, M'_3, \iota'_3, \omega'_3$.

9. (Assign-R)

Let $\varepsilon_1 = o.\mathtt{f} = v$ and $\varepsilon'_1 = o'.\mathtt{f} = v'$.

By (Assign-R), $o.\mathtt{f} = v; , \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \; ; , \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_1, \omega_1$, where $H_2 = H_1[o \mapsto \mathtt{C}, F_2]$, $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$ and $H(o) = \mathtt{C}, F_1$ and $F_2 = F_1[\mathtt{f} = v]$ and $\mathbf{TD}_2 = \mathit{tdassign}(\mathbf{TD}_1, \mathcal{C}_\top)$.

and

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_o \qquad \mathbf{TD}_v}{\Gamma, pc_2, \mathcal{H}_1 \vdash o.\mathtt{f} = v : \langle\, s_o \cup s_v, \emptyset, \mathtt{void} \,\rangle \backslash \mathcal{C}_o \cup \mathcal{C}_v \cup \{f_o.\mathtt{f} <: \textbf{set} \langle\, s_o \cup s_v, f_v, \alpha_v \,\rangle\}}\ \text{(F-Assign)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash o.\mathtt{f} = v : t \backslash \mathcal{C}}\ \text{(Sub)}$$

and

$$\mathbf{TD}_o = \dfrac{\dfrac{\mathcal{H}_1(o) = t_h \backslash \mathcal{C}_h}{\Gamma, pc_3, \mathcal{H}_1 \vdash o : \langle\, pc_3 \cup s_h, f_h, \alpha_h \,\rangle \backslash \mathcal{C}_h}\ \text{(Oid)}}{\Gamma, pc_2, \mathcal{H}_1 \vdash o : t_o \backslash \mathcal{C}_o}\ \text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H}_2 \vdash \; ; : \langle\, pc_1, \emptyset, \mathtt{void} \,\rangle \backslash \emptyset}\ \text{(No-op)}}{\Gamma, pc_1, \mathcal{H}_2 \vdash \; ; : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$\mathcal{H}_2 = \mathcal{H}_1[o \mapsto t_h \backslash \mathcal{C}_a]$

where $\mathcal{C}_a = \mathcal{C}_h \cup \mathcal{C}_v \cup \{f_h.\mathtt{f} <: \textbf{set}\ t_v\}]$

Again by (Assign-R), $o'.\mathtt{f} = v';, \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow ;, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_2, \iota'_1, \omega'_1$, where $H'_2 = H'_1[o' \mapsto \mathtt{C}', F'_2]$, and $fields(\mathtt{C'}) = \overline{\mathtt{C}'}\ \overline{\mathtt{f}}$ and $H'_1(o') = \mathtt{C}', F'_1$ and $F'_2 = F'_1[\mathtt{f} = v']$ and $\mathbf{TD}'_2 = tdassign(\mathbf{TD}'_1, \mathcal{C}'_\top)$.

and

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\mathbf{TD}'_o \qquad \mathbf{TD}'_v}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash o'.\mathtt{f} = v' : \langle\, s'_o \cup s'_v, \emptyset, \mathtt{void} \,\rangle \backslash \mathcal{C}'_o \cup \mathcal{C}'_v \cup \{f'_o.\mathtt{f} <: \textbf{set} \,\langle\, s'_o \cup s'_v, f'_v, \alpha'_v \,\rangle\}}\ \text{(F-Assign)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash o'.\mathtt{f} = v' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

and

$$\mathbf{TD}'_o = \dfrac{\dfrac{\mathcal{H}'_1(o) = t'_h \backslash \mathcal{C}'_h}{\Gamma', pc'_3, \mathcal{H}'_1 \vdash o' : \langle\, pc'_3 \cup s'_h, f'_h, \alpha'_h \,\rangle \backslash \mathcal{C}'_h}\ \text{(Oid)}}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash o' : t'_o \backslash \mathcal{C}'_o}\ \text{(Sub)}$$

$$\mathbf{TD}'_2 = \dfrac{\dfrac{}{\Gamma', pc'_1, \mathcal{H}'_2 \vdash ; : \langle\, pc'_1, \emptyset, \mathtt{void} \,\rangle \backslash \emptyset}\ \text{(No-op)}}{\Gamma', pc'_1, \mathcal{H}'_2 \vdash ; : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$\mathcal{H}'_2 = \mathcal{H}'_1[o' \mapsto t'_h \backslash \mathcal{C}'_a]$

where $\mathcal{C}'_a = \mathcal{C}'_h \cup \mathcal{C}'_v \cup \{f'_h.\mathtt{f} <: \textbf{set}\ t'_v\}]$

According to Definition 4.15[3], we have two cases.

(a) $Con(s_h, \mathcal{C}_\top) \subseteq \mathtt{Low}$

Now, according to Definition 4.17, either $Con(s_o, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$ or $Con(s_v, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$, so by Definition 4.2, $Con(s_o \cup s_v, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$. By Definition 4.15[2i,2a], we have $t_o = t'_o$, $t_v = t'_v$, and $\mathcal{C}_\top = \mathcal{C}'_\top$. This yields $Con(s'_o \cup s'_v, \mathcal{C}'_\top) \not\subseteq \mathtt{Low}$.

Now, by (Sub), $\{f_o.\mathtt{f} <: \textbf{set}\ \langle\, s_o \cup s_v, f_v, \alpha_v \,\rangle\} \subseteq \mathcal{C}$, again by (Sub) $\{f_h <: f_o\} \subseteq \mathcal{C}$, then by ($\mathcal{S}$-Field), $\{f_h.\mathtt{f}.\mathsf{S} <: \textbf{set}\ s_o \cup s_v\} \subseteq \mathcal{C}$, and since $\mathcal{C}$ is closed, by (Set), $\{s_o \cup s_v <: f_h.\mathtt{f}.\mathsf{S}\} \subseteq \mathcal{C}$; and since by $tdassign$, $\mathcal{C} \subseteq \mathcal{C}_\top$, we have $\{s_o \cup s_v <: f_h.\mathtt{f}.\mathsf{S}\} \subseteq \mathcal{C}_\top$. So, by Lemma 4.20, $Con(s_o \cup s_v, \mathcal{C}_\top) \subseteq Con(f_h.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top)$, and since $Con(s_o \cup s_v, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$, we have $Con(f_h.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$.

Since $s_o = s'_o$, $s_v = s'_v$, and $\mathcal{C}_\top = \mathcal{C}'_\top$, we have $\{s'_o \cup s'_v <: f'_h.\mathtt{f}.\mathsf{S}\} \subseteq \mathcal{C}'_\top$. So, by Lemma 4.20, $Con(s'_o \cup s'_v, \mathcal{C}'_\top) \subseteq Con(f'_h.\mathtt{f}.\mathsf{S}, \mathcal{C}'_\top)$, and since $Con(s'_o \cup s'_v, \mathcal{C}'_\top) \not\subseteq \mathtt{Low}$, we have $Con(f'_h.\mathtt{f}.\mathsf{S}, \mathcal{C}'_\top) \not\subseteq \mathtt{Low}$.

Therefore, by Definition 4.15[1], $Con(f_h.\mathtt{f}.\mathsf{S}, \mathcal{C}_\top), v \simeq_{\mathtt{Low}} Con(f'_h.\mathtt{f}.\mathsf{S}, \mathcal{C}'_\top), v'$, and we conclude by Definition 4.15[3] $\mathcal{H}_2, \mathcal{C}_\top, H_2 \simeq_{\mathtt{Low}} \mathcal{H}'_2, \mathcal{C}'_\top, H'_2$.

(b) $Con(s_h, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$

Then, by Definition 4.15[3], we must have $Con(s'_h, \mathcal{C}'_\top) \not\subseteq \mathtt{Low}$. Otherwise, if this were not the case $Con(s_h, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$ would not hold (since if one location is "low", the other must be also).

Hence, by Definition 4.15[3] $\mathcal{H}_2, \mathcal{C}_\top, H_2 \simeq_{\mathtt{Low}} \mathcal{H}'_2, \mathcal{C}'_\top, H'_2$.

Since $t = t'$, by Definition 4.15[2p], $;, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} ;, \mathbf{TD}'_2, \mathcal{C}_\top$, and since $\mathcal{H}_2, \mathcal{C}_\top, H_2 \simeq_{\mathtt{Low}} \mathcal{H}'_2, \mathcal{C}'_\top, H'_2$, concluding with Definition 4.15[8],

$;, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_1, \omega_1 \simeq_{\mathtt{Low}} ;, \mathbf{TD}'_2, \mathcal{C}_\top, H'_2, \iota'_1, \omega'_1$.

10. (New-R)

Let $\varepsilon_1 = \mathtt{new}\ \mathtt{C}(\overline{v})$ and $\varepsilon'_1 = \mathtt{new}\ \mathtt{C}(\overline{v'})$.

By (New-R), let $\mathtt{new}\ \mathtt{C}(\overline{v}), \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \varepsilon_a, \mathbf{TD}_a, \mathcal{C}_\top, H_a, \iota_1, \omega_1$.

and

$$\mathbf{TD}_1 = \dfrac{\dfrac{\overline{\mathbf{TD}_v} \qquad \overline{\mathbf{TD}_n}}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{new}\ \mathtt{C}(\overline{v}) : t_o \backslash \ldots}\ \text{(New)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \mathtt{new}\ \mathtt{C}(\overline{v}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

Again by (New-R), let $\mathtt{new}\ \mathtt{C}(\overline{v'}), \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \varepsilon'_a, \mathbf{TD}'_a, \mathcal{C}'_\top, H'_a, \iota'_1, \omega'_1$.

and

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\overline{\mathbf{TD}'_v} \qquad \overline{\mathbf{TD}'_n}}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \texttt{new C}(\overline{v'}) : t'_o \backslash \ldots}\ \text{(New)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \texttt{new C}(\overline{v'}) : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

By Definition 4.17, we have $Con(s_o, \mathcal{C}_\top) \not\subseteq \texttt{Low}$. Hence by Definition 4.15[2k, 2a] and assumption, $Con(s'_o, \mathcal{C}'_\top) \not\subseteq \texttt{Low}$.

We first show the new heaps are bisimilar. By (New-R), we have $H_a = H_1[o \mapsto \texttt{C}, F]$ and $\mathcal{H}_a = \mathcal{H}_1[o \mapsto t_o \backslash \mathcal{C}_o]$; and again by (New-R), we have $H'_a = H'_1[o' \mapsto \texttt{C}, F']$ and $\mathcal{H}'_a = \mathcal{H}'_1[o' \mapsto t'_o \backslash \mathcal{C}'_o]$. Since $Con(s_o, \mathcal{C}_\top) \not\subseteq \texttt{Low}$ and $Con(s'_o, \mathcal{C}'_\top) \not\subseteq \texttt{Low}$, and by assumption $\mathcal{H}_1, \mathcal{C}_\top, H_1 \simeq_{\texttt{Low}} \mathcal{H}'_1, \mathcal{C}'_\top, H'_1$, by Definition 4.15[3], $\mathcal{H}_a, \mathcal{C}_\top, H_a \simeq_{\texttt{Low}} \mathcal{H}'_a, \mathcal{C}'_\top, H'_a$.

Now, by assumption, $\varepsilon_a, \mathbf{TD}_a, \mathcal{C}_\top, H_a, \iota_1, \omega_1$ terminates, and we have just shown $Con(s_o, \mathcal{C}_\top) \not\subseteq \texttt{Low}$. According to $tdnew(\mathbf{TD}_1, \texttt{C}, H_a, \mathcal{C}_\top)$, we know the penultimate type rule of $\mathbf{TD}_a$ concludes with $\Gamma, pc_2 \cup s_o, \mathcal{H}_a \vdash \varepsilon_a : \ldots$. Hence by Lemma 4.24,

$\varepsilon_a, \mathbf{TD}_a, \mathcal{C}_\top, H_a, \iota_1, \omega_1 \rightsquigarrow^*_h \varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2$, where $\varepsilon_2 = v_2$ for some v_2, and

$$\mathbf{TD}_2 = \dfrac{\ldots}{\Gamma, pc_1, \mathcal{H}_2 \vdash v_2 : t_2 \backslash \mathcal{C}_2}\ \text{(Sub)}$$

and $Con(s_2, \mathcal{C}_\top) \not\subseteq \texttt{Low}$. Now, by Lemma 4.25[2], $\mathcal{H}_a, \mathcal{C}_\top, H_a \simeq_{\texttt{Low}} \mathcal{H}_2, \mathcal{C}_\top, H_2$, and $\iota_1 \simeq_{\texttt{Low}} \iota_2$, and $\omega_1 \simeq_{\texttt{Low}} \omega_2$.

Similarly, by assumption, $\varepsilon'_a, \mathbf{TD}'_a, \mathcal{C}'_\top, H'_a, \iota'_1, \omega'_1$ terminates, and we have just shown that $Con(s'_o, \mathcal{C}'_\top) \not\subseteq \texttt{Low}$. According to $tdnew(\mathbf{TD}'_1, \texttt{C}, H'_a, \mathcal{C}'_\top)$, we know the penultimate type rule of $\mathbf{TD}'_a$ concludes with $\Gamma', pc'_2 \cup s'_o, \mathcal{H}'_a \vdash \varepsilon'_a : \ldots$.

Hence by Lemma 4.24, $\varepsilon'_a, \mathbf{TD}'_a, \mathcal{C}'_\top, H'_a, \iota'_1, \omega'_1 \rightsquigarrow^*_h \varepsilon'_2, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_2, \iota'_2, \omega'_2$, where $\varepsilon'_2 = v'_2$ for some v'_2, and

$$\mathbf{TD}'_2 = \dfrac{\ldots}{\Gamma', pc'_1, H'_2 \vdash v'_2 : t'_2 \backslash \mathcal{C}'_2}\ \text{(Sub)}$$

and $Con(s_2', \mathcal{C}_\top') \not\subseteq \texttt{Low}$. Now, by Lemma 4.25[2], $\mathcal{H}_a', \mathcal{C}_\top', H_a' \simeq_{\texttt{Low}} \mathcal{H}_2', \mathcal{C}_\top', H_2'$, $\iota_1' \simeq_{\texttt{Low}} \iota_2'$, and $\omega_1' \simeq_{\texttt{Low}} \omega_2'$.

Now, by Subject Reduction Lemma 4.13, the final type of $\mathbf{TD}_2$ must be the same as that of $\mathbf{TD}_1$, and the final type of $\mathbf{TD}_2'$ must be the same as that of $\mathbf{TD}_1'$. Since $\varepsilon_1, \mathcal{C}_\top, \mathbf{TD}_1 \simeq_{\texttt{Low}} \varepsilon_1', \mathcal{C}_\top', \mathbf{TD}_1'$, by Definition 4.15[2k], this means $t_2 = t_2'$. Since $Con(s_2, \mathcal{C}_\top) \not\subseteq \texttt{Low}$ and $Con(s_2', \mathcal{C}_\top') \not\subseteq \texttt{Low}$, by Definition 4.15[2a,1], $v_2, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\texttt{Low}} v_2', \mathbf{TD}_2', \mathcal{C}_\top'$, which is the same as $\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\texttt{Low}} \varepsilon_2', \mathbf{TD}_2', \mathcal{C}_\top'$.

By Lemma 4.18[2,3], we have $\mathcal{H}_2, \mathcal{C}_\top, H_2 \simeq_{\texttt{Low}} \mathcal{H}_2', \mathcal{C}_\top', H_2'$, $\iota_2 \simeq_{\texttt{Low}} \iota_2'$ and $\omega_2 \simeq_{\texttt{Low}} \omega_2'$. Then by Definition 4.15[8], $\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2 \simeq_{\texttt{Low}} \varepsilon_2', \mathbf{TD}_2', \mathcal{C}_\top', H_2', \iota_2', \omega_2'$.

11. (Invoke-R)

Let $\varepsilon_1 = o.\mathtt{m}(\overline{v})$ and $\varepsilon_1' = o'.\mathtt{m}(\overline{v'})$.

By (Invoke-R), let $o.\mathtt{m}(\overline{v}), \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \varepsilon_a, \mathbf{TD}_a, \mathcal{C}_\top, H_1, \iota_1, \omega_1$,

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_o \qquad \overline{\mathbf{TD}_v}}{\Gamma, pc_2, \mathcal{H}_1 \vdash o.\mathtt{m}(\overline{v}) : t_m \backslash \ldots}\ \text{(Invoke)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash o.\mathtt{m}(\overline{v}) : t \backslash \mathcal{C}}\ \text{(Sub)}$$

$$\mathbf{TD}_o = \dfrac{\ldots}{\Gamma, pc_2, \mathcal{H}_1 \vdash o : t_o \backslash \mathcal{C}_o}\ \text{(Sub)}$$

$$\overline{\mathbf{TD}_v} = \dfrac{\ldots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \overline{v} : \overline{t_v} \backslash \overline{\mathcal{C}_v}}\ \text{(Sub)}$$

Again by (Invoke-R), $o'.\mathtt{m}(\overline{v'}), \mathbf{TD}_1', \mathcal{C}_\top', H_1', \iota_1', \omega_1' \rightsquigarrow \varepsilon_a', \mathbf{TD}_a', \mathcal{C}_\top', H_1', \iota_1', \omega_1'$.

$$\mathbf{TD}_1' = \dfrac{\dfrac{\mathbf{TD}_o' \qquad \overline{\mathbf{TD}_v'}}{\Gamma', pc_2', \mathcal{H}_1' \vdash o'.\mathtt{m}(\overline{v'}) : t_m' \backslash \ldots}\ \text{(Invoke)}}{\Gamma', pc_1', \mathcal{H}_1' \vdash o'.\mathtt{m}(\overline{v'}) : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

$$\mathbf{TD}_o' = \dfrac{\ldots}{\Gamma', pc_2', \mathcal{H}_1' \vdash o' : t_o' \backslash \mathcal{C}_o'}\ \text{(Sub)}$$

$$\overline{\mathbf{TD}_v'} = \dfrac{\ldots}{\Gamma', pc_2', \mathcal{H}_1' \vdash \overline{v'} : \overline{t_v'} \backslash \overline{\mathcal{C}_v'}}\ \text{(Sub)}$$

By Definition 4.17, we have $Con(s_o, \mathcal{C}_\top) \not\subseteq \texttt{Low}$. Hence by Definition 4.15[2k, 2a] and assumption, $Con(s'_o, \mathcal{C}'_\top) \not\subseteq \texttt{Low}$.

By assumption, $\varepsilon_a, \mathbf{TD}_a, \mathcal{C}_\top, H_1, \iota_1, \omega_1$ terminates, and we have just shown $Con(s_o, \mathcal{C}_\top) \not\subseteq \texttt{Low}$. According to $tdinvoke(\mathbf{TD}_1, \texttt{C}, \texttt{m}, \mathcal{C}_\top)$, we know the penultimate type rule of $\mathbf{TD}_a$ concludes with $\Gamma, pc_2 \cup s_o, \mathcal{H}_1 \vdash \varepsilon_a : \ldots$. Hence by Lemma 4.24, $\varepsilon_a, \mathbf{TD}_a, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_h^* \varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2$, where $\varepsilon_2 = v_2$ for some v_2, and

$$\mathbf{TD}_2 = \dfrac{\ldots}{\Gamma, pc_1, \mathcal{H}_2 \vdash v_2 : t_2 \backslash \mathcal{C}_2}\ \text{(Sub)}$$

and $Con(s_2, \mathcal{C}_\top) \not\subseteq \texttt{Low}$. Now, by Lemma 4.25[2], $\mathcal{H}_1, \mathcal{C}_\top, H_1 \simeq_{\texttt{Low}} \mathcal{H}_2, \mathcal{C}_\top, H_2$, and $\iota_1 \simeq_{\texttt{Low}} \iota_2$, and $\omega_1 \simeq_{\texttt{Low}} \omega_2$.

Similarly, by assumption, $\varepsilon'_a, \mathbf{TD}'_a, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$ terminates, and we have already shown that $Con(s'_o, \mathcal{C}'_\top) \not\subseteq \texttt{Low}$. According to $tdinvoke(\mathbf{TD}'_1, \texttt{C}, \texttt{m}, \mathcal{C}'_\top)$, we know the penultimate type rule of $\mathbf{TD}'_a$ concludes with $\Gamma', pc'_2 \cup s'_o, \mathcal{H}'_1 \vdash \varepsilon'_a : \ldots$.

Hence by Lemma 4.24, $\varepsilon'_a, \mathbf{TD}'_a, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow_h^* \varepsilon'_2, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_2, \iota'_2, \omega'_2$, where $\varepsilon'_2 = v'_2$ for some v'_2, and

$$\mathbf{TD}'_2 = \dfrac{\ldots}{\Gamma', pc'_1, \mathcal{H}'_2 \vdash v'_2 : t'_2 \backslash \mathcal{C}'_2}\ \text{(Sub)}$$

and $Con(s'_2, \mathcal{C}'_\top) \not\subseteq \texttt{Low}$. Now, by Lemma 4.25[2], $\mathcal{H}'_1, \mathcal{C}'_\top, H'_1 \simeq_{\texttt{Low}} \mathcal{H}'_2, \mathcal{C}'_\top, H'_2$, and $\iota'_1 \simeq_{\texttt{Low}} \iota'_2$, and $\omega'_1 \simeq_{\texttt{Low}} \omega'_2$.

Now, by Subject Reduction Lemma 4.13, the final type of $\mathbf{TD}_2$ must be the same as that of $\mathbf{TD}_1$, and the final type of $\mathbf{TD}'_2$ must be the same as that of $\mathbf{TD}'_1$. Since $\varepsilon_1, \mathcal{C}_\top, \mathbf{TD}_1 \simeq_{\texttt{Low}} \varepsilon'_1, \mathcal{C}'_\top, \mathbf{TD}'_1$, by Definition 4.15[2k], this means $t_2 = t'_2$ and $\mathcal{C}_2 = \mathcal{C}'_2$. Since $Con(s_2, \mathcal{C}_\top) \not\subseteq \texttt{Low}$ and

$Con(s'_2, \mathcal{C}'_\top) \not\subseteq \texttt{Low}$, by Definition 4.15[2a,1], $v_2, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\texttt{Low}} v'_2, \mathbf{TD}'_2, \mathcal{C}'_\top$, which is the same as $\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\texttt{Low}} \varepsilon'_2, \mathbf{TD}'_2, \mathcal{C}'_\top$.

By Lemma 4.18[2,3], we have $\mathcal{H}_2, \mathcal{C}_\top, H_2 \simeq_{\texttt{Low}} \mathcal{H}'_2, \mathcal{C}'_\top, H'_2$, and $\iota_2 \simeq_{\texttt{Low}} \iota'_2$ and $\omega_2 \simeq_{\texttt{Low}} \omega'_2$.

Then by Definition 4.15[8], $\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2 \simeq_{\texttt{Low}} \varepsilon'_2, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_2, \iota'_2, \omega'_2$.

12. (Super-R)

    Follows similarly to (New-R) and (Invoke-R).

13. (Input-R)

    Let $\varepsilon_1 = \texttt{read}_\texttt{L}(\texttt{fd}), \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1$ and $\varepsilon'_1 = \texttt{read}_\texttt{L}(\texttt{fd}'), \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1$

    By (Input-R)

    $\texttt{read}_\texttt{L}(\texttt{fd}), \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \texttt{c}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_2, \omega_1$

    where $\texttt{L} = S_i$ and $\iota_1(\texttt{fd}, S_i) = \texttt{c}.\iota_2(\texttt{fd}, S_i)$, and $S_f = \mathit{conread}(\mathbf{TD}_1, \mathcal{C}_\top)$ and $\mathbf{TD}_2 = \mathit{tdinput}(\mathbf{TD}_1, \texttt{c}, \mathcal{C}_\top)$ and $S_f \subseteq \texttt{L}$.

    and

$$\mathbf{TD}_1 = \dfrac{\dfrac{\dfrac{}{\Gamma, pc_3, \mathcal{H} \vdash \texttt{fd} : \langle\, pc_3, \emptyset, \texttt{int} \,\rangle \backslash \emptyset}\,(\text{Const})}{\Gamma, pc_2, \mathcal{H} \vdash \texttt{fd} : t_f \backslash \mathcal{C}_f}\,(\text{Sub})}{\dfrac{\Gamma, pc_2, \mathcal{H} \vdash \texttt{read}_\texttt{L}(\texttt{fd}) : \langle\, s_f \cup \texttt{L}, \emptyset, \texttt{int} \,\rangle \backslash \mathcal{C}_f \cup SC(\texttt{L}, s_f)}{\Gamma, pc_1, \mathcal{H} \vdash \texttt{read}_\texttt{L}(\texttt{fd}) : t \backslash \mathcal{C}}\,(\text{Sub})}\,(\text{Input})$$

    and

$$\mathbf{TD}_2 = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H} \vdash \texttt{c} : \langle\, pc_1, \emptyset, \texttt{int} \,\rangle \backslash \emptyset}\,(\text{Const})}{\Gamma, pc_1, \mathcal{H} \vdash \texttt{c} : t \backslash \mathcal{C}}\,(\text{Sub})$$

    Again by (Input-R)

    $\texttt{read}_\texttt{L}(\texttt{fd}'), \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \texttt{c}', \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_2, \omega'_1$

where $\mathtt{L} = S_i$ and $\iota_1'(\mathtt{fd}', S_i) = \mathtt{c}'.\iota_2'(\mathtt{fd}', S_i)$, and $S_f' = \mathit{conread}(\mathbf{TD}_1', \mathcal{C}_\top')$ and $\mathbf{TD}_2' = \mathit{tdinput}(\mathbf{TD}_1', \mathtt{c}', \mathcal{C}_\top')$ and $S_f' \subseteq \mathtt{L}$

and

$$\mathbf{TD}_1' = \dfrac{\dfrac{\dfrac{}{\Gamma', pc_3', \mathcal{H}' \vdash \mathtt{fd}' : \langle\, pc_3', \emptyset, \mathtt{int}\,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma', pc_2', \mathcal{H}' \vdash \mathtt{fd}' : t_f' \backslash \mathcal{C}_f'}\ \text{(Sub)}}{\dfrac{\Gamma', pc_2', \mathcal{H}' \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{fd}') : \langle\, s_f' \cup \mathtt{L}, \emptyset, \mathtt{int}\,\rangle \backslash \mathcal{C}_f' \cup SC(\mathtt{L}, s_f')}\ \text{(Input)}}{\Gamma', pc_1', \mathcal{H}' \vdash \mathtt{read}_{\mathtt{L}}(\mathtt{fd}') : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

and

$$\mathbf{TD}_2' = \dfrac{\dfrac{}{\Gamma', pc_1', \mathcal{H}' \vdash \mathtt{c}' : \langle\, pc_1', \emptyset, \mathtt{int}\,\rangle \backslash \emptyset}\ \text{(Const)}}{\Gamma', pc_1', \mathcal{H}' \vdash \mathtt{c}' : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

By Definition 4.17, $S_f \not\subseteq \mathtt{Low}$, and since $S_f \subseteq \mathtt{L}$, we have $\mathtt{L} \not\subseteq \mathtt{Low}$, so by Definition 4.15[7], $\iota_1 \simeq_{\mathtt{Low}} \iota_2$ and $\iota_1' \simeq_{\mathtt{Low}} \iota_2'$. So, by Lemma 4.18[2,3] $\iota_2 \simeq_{\mathtt{Low}} \iota_2'$.

Now, by (Sub), $\{\mathtt{L} <: s\} \subseteq \mathcal{C}$, and since $\mathtt{L} \not\subseteq \mathtt{Low}$, by Definition 4.2 $\mathit{Con}(s, \mathcal{C}_\top) \not\subseteq \mathtt{Low}$. By Definition 4.15[2n], $t = t'$ and $\mathcal{C}_\top = \mathcal{C}_\top'$, so $\mathit{Con}(s', \mathcal{C}_\top') \not\subseteq \mathtt{Low}$, and by Definition 4.15[1], $\mathit{Con}(s, \mathcal{C}_\top), \mathtt{c} \simeq_{\mathtt{Low}} \mathit{Con}(s', \mathcal{C}_\top'), \mathtt{c}'$, so by Definition 4.15[2a], $\mathtt{c}, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathtt{c}', \mathbf{TD}_2', \mathcal{C}_\top'$. Since $\iota_2 \simeq_{\mathtt{Low}} \iota_2'$, conclude with Definition 4.15[8], $\mathtt{c}, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_2, \omega_1 \simeq_{\mathtt{Low}} \mathtt{c}', \mathbf{TD}_2', \mathcal{C}_\top', H_1', \iota_2', \omega_1$.

14. (Output-R)

Let $\varepsilon_1 = \mathtt{write}_{\mathtt{L}}(\mathtt{c}, \mathtt{fd}), \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1$ and

$\varepsilon_1' = \mathtt{write}_{\mathtt{L}}(\mathtt{c}', \mathtt{fd}'), \mathbf{TD}_1', \mathcal{C}_\top', H_1', \iota_1', \omega_1'$

By (Output-R), $\mathtt{write}_{\mathtt{L}}(\mathtt{c}, \mathtt{fd}), \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow ;, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_2$,

where $\mathtt{L} = S_i$, and $\omega_2(\mathtt{fd}, S_i) = \mathtt{c}.\omega_2(\mathtt{fd}, S_i)$, and $S_f = \mathit{conwrite}(\mathbf{TD}_1, \mathcal{C}_\top)$, and $\mathbf{TD}_2 = \mathit{tdoutput}(\mathbf{TD}_1, \mathcal{C}_\top)$, and $S_f \subseteq \mathtt{L}$.

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_c \qquad \mathbf{TD}_f}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{write_L(c, fd)} : \langle\, s_c \cup s_f, \emptyset, \mathtt{void} \,\rangle \backslash \mathcal{C}_c \cup \mathcal{C}_f}\,\text{(Output)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash \mathtt{write_L(c, fd)} : t \backslash \mathcal{C}}\,\text{(Sub)}$$

$$\mathbf{TD}_c = \dfrac{\dfrac{}{\Gamma, pc_3, \mathcal{H}_1 \vdash \mathtt{c} : \langle\, pc_3, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\,\text{(Const)}}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{c} : t_c \backslash \mathcal{C}_c}\,\text{(Sub)}$$

$$\mathbf{TD}_f = \dfrac{\dfrac{}{\Gamma, pc_4, \mathcal{H}_1 \vdash \mathtt{fd} : \langle\, pc_4, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\,\text{(Const)}}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{fd} : t_f \backslash \mathcal{C}_f}\,\text{(Sub)}$$

$$\mathbf{TD}_2 = \dfrac{\dfrac{}{\Gamma, pc_1, \mathcal{H}_1 \vdash ; : \langle\, pc_1, \emptyset, \mathtt{void} \,\rangle \backslash \emptyset}\,\text{(No-op)}}{\Gamma, pc_1, \mathcal{H}_1 \vdash ; : t \backslash \mathcal{C}}\,\text{(Sub)}$$

Again by (Output-R), $\mathtt{write_L(c', fd')}, \mathbf{TD}'_1, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow ;, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_2$, where $\mathtt{L} = S_i$, and $\omega'_2(\mathtt{fd'}, S_i) = \mathtt{c'}.\omega'_2(\mathtt{fd'}, S_i)$, and $S'_f = \mathit{conwrite}(\mathbf{TD}'_1, \mathcal{C}'_\top)$, and $\mathbf{TD}'_2 = \mathit{tdoutput}(\mathbf{TD}'_1, \mathcal{C}'_\top)$, and $S'_f \subseteq \mathtt{L}$.

$$\mathbf{TD}'_1 = \dfrac{\dfrac{\mathbf{TD}'_c \qquad \mathbf{TD}'_f}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{write_L(c', fd')} : \langle\, s'_c \cup s'_f, \emptyset, \mathtt{void} \,\rangle \backslash \mathcal{C}'_c \cup \mathcal{C}'_f}\,\text{(Output)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash \mathtt{write_L(c', fd')} : t' \backslash \mathcal{C}'}\,\text{(Sub)}$$

$$\mathbf{TD}'_c = \dfrac{\dfrac{}{\Gamma', pc'_3, \mathcal{H}'_1 \vdash \mathtt{c'} : \langle\, pc'_3, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\,\text{(Const)}}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{c'} : t'_c \backslash \mathcal{C}'_c}\,\text{(Sub)}$$

$$\mathbf{TD}'_f = \dfrac{\dfrac{}{\Gamma', pc'_4, \mathcal{H}'_1 \vdash \mathtt{fd'} : \langle\, pc'_4, \emptyset, \mathtt{int} \,\rangle \backslash \emptyset}\,\text{(Const)}}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{fd'} : t'_f \backslash \mathcal{C}'_f}\,\text{(Sub)}$$

$$\mathbf{TD}'_2 = \dfrac{\dfrac{}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash ; : \langle\, pc'_1, \emptyset, \mathtt{void} \,\rangle \backslash \emptyset}\,\text{(No-op)}}{\Gamma', pc'_1, \mathcal{H}'_1 \vdash ; : t' \backslash \mathcal{C}'}\,\text{(Sub)}$$

By Definition 4.17, $S_f \not\subseteq \mathtt{Low}$, and since $S_f \subseteq \mathtt{L}$, we have $\mathtt{L} \not\subseteq \mathtt{Low}$. So by Definition 4.15[6], $\omega_1 \simeq_{\mathtt{Low}} \omega_2$ and $\omega'_1 \simeq_{\mathtt{Low}} \omega'_2$. So, by Lemma 4.18[2,3] $\omega_2 \simeq_{\mathtt{Low}} \omega'_2$.

By Definition 4.15[2o], we have $t = t'$, and $\mathcal{C}_\top = \mathcal{C}'_\top$, so by Definition 4.15[2p], $;, \mathbf{TD}_2, \mathcal{C}_\top \simeq_{\mathtt{Low}} ;, \mathbf{TD}'_2, \mathcal{C}'_\top$. Since $\omega_2 \simeq_{\mathtt{Low}} \omega'_2$, conclude with Definition 4.15[8], $;, \mathbf{TD}_2, \mathcal{C}_\top, H_1, \iota_1, \omega_2 \simeq_{\mathtt{Low}} ;, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_1, \iota'_1, \omega_2$.

15. (Op-RC)

Let $\varepsilon_1 = \mathtt{e}_a \oplus \mathtt{e}_c$ and $\varepsilon_1' = \mathtt{e}_a' \oplus \mathtt{e}_c'$

By (Op-RC), $\mathtt{e}_a \oplus \mathtt{e}_c, \mathbf{TD}_1, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \mathtt{e}_b \oplus \mathtt{e}_c, \mathbf{TD}_e, \mathcal{C}_\top, H_b, \iota_b, \omega_b$, where

$\mathtt{e}_a, \mathbf{TD}_a, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \mathtt{e}_b, \mathbf{TD}_b, \mathcal{C}_\top, H_b, \iota_b, \omega_b$, and $\mathbf{TD}_a = ucon(\mathbf{TD}_1)$ and $\mathbf{TD}_e = dconop(\mathbf{TD}_1, \mathbf{TD}_b, \mathcal{C}_\top)$

$$\mathbf{TD}_1 = \dfrac{\dfrac{\mathbf{TD}_a \qquad \mathbf{TD}_c}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{e}_a \oplus \mathtt{e}_c : \langle\, s_a \cup s_c, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_a \cup \mathcal{C}_c}\,(\text{Op})}{\Gamma, pc_1, \mathcal{H}_1 \vdash \mathtt{e}_a \oplus \mathtt{e}_b : t \backslash \mathcal{C}}\,(\text{Sub})$$

$$\mathbf{TD}_a = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{e}_a : t_a \backslash \mathcal{C}_c}\,(\text{Sub})$$

$$\mathbf{TD}_c = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_1 \vdash \mathtt{e}_c : t_c \backslash \mathcal{C}_c}\,(\text{Sub})$$

$$\mathbf{TD}_b = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_b \vdash \mathtt{e}_b : t_a \backslash \mathcal{C}_a}\,(\text{Sub})$$

$$\mathbf{TD}_d = tdheap(\mathbf{TD}_c, \mathcal{H}_b)$$

$$\mathbf{TD}_d = \dfrac{\cdots}{\Gamma, pc_2, \mathcal{H}_b \vdash \mathtt{e}_c : t_c \backslash \mathcal{C}_c}\,(\text{Sub})$$

$$\mathbf{TD}_e = \dfrac{\dfrac{\mathbf{TD}_b \qquad \mathbf{TD}_d}{\Gamma, pc_2, \mathcal{H}_b \vdash \mathtt{e}_b \oplus \mathtt{e}_c : \langle\, s_a \cup s_c, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_a \cup \mathcal{C}_c}\,(\text{Op})}{\Gamma, pc_1, \mathcal{H}_b \vdash \mathtt{e}_b \oplus \mathtt{e}_c : t \backslash \mathcal{C}}\,(\text{Sub})$$

Again by (Op-RC), $\mathtt{e}_a' \oplus \mathtt{e}_c', \mathbf{TD}_1', \mathcal{C}_\top', H_1', \iota_1', \omega_1' \rightsquigarrow \mathtt{e}_b' \oplus \mathtt{e}_c', \mathbf{TD}_e', \mathcal{C}_\top', H_b', \iota_b', \omega_b'$, where

$\mathtt{e}_a', \mathbf{TD}_a', \mathcal{C}_\top', H_1', \iota_1', \omega_1' \rightsquigarrow \mathtt{e}_b', \mathbf{TD}_b', \mathcal{C}_\top', H_b', \iota_b', \omega_b'$, and $\mathbf{TD}_a' = ucon(\mathbf{TD}_1')$ and $\mathbf{TD}_e' = dconop(\mathbf{TD}_1', \mathbf{TD}_b', \mathcal{C}_\top')$

$$\mathbf{TD}_1' = \dfrac{\dfrac{\mathbf{TD}_a' \qquad \mathbf{TD}_c'}{\Gamma', pc_2', \mathcal{H}_1' \vdash \mathtt{e}_a' \oplus \mathtt{e}_c' : \langle\, s_a' \cup s_c', \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_a' \cup \mathcal{C}_c'}\,(\text{Op})}{\Gamma', pc_1', \mathcal{H}_1' \vdash \mathtt{e}_a' \oplus \mathtt{e}_b' : t' \backslash \mathcal{C}'}\,(\text{Sub})$$

$$\mathbf{TD}_a' = \dfrac{\cdots}{\Gamma', pc_2', \mathcal{H}_1' \vdash \mathtt{e}_a' : t_a' \backslash \mathcal{C}_c'}\,(\text{Sub})$$

$\mathbf{TD}'_c =$

$$\frac{\cdots}{\Gamma', pc'_2, \mathcal{H}'_1 \vdash \mathtt{e}'_c : t'_c \backslash \mathcal{C}'_c}\ (\text{Sub})$$

$\mathbf{TD}'_b =$

$$\frac{\cdots}{\Gamma', pc'_2, \mathcal{H}'_b \vdash \mathtt{e}'_b : t'_a \backslash \mathcal{C}'_a}\ (\text{Sub})$$

$\mathbf{TD}'_d = tdheap(\mathbf{TD}'_c, \mathcal{H}'_b)$

$\mathbf{TD}'_d =$

$$\frac{\cdots}{\Gamma', pc'_2, \mathcal{H}'_b \vdash \mathtt{e}'_c : t'_c \backslash \mathcal{C}'_c}\ (\text{Sub})$$

$$\mathbf{TD}'_e = \frac{\dfrac{\mathbf{TD}'_b \qquad \mathbf{TD}'_d}{\Gamma', pc'_2, \mathcal{H}'_b \vdash \mathtt{e}'_b \oplus \mathtt{e}'_c : \langle\, s'_a \cup s'_c, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}'_a \cup \mathcal{C}'_c}\ (\text{Op})}{\Gamma', pc'_1, \mathcal{H}'_b \vdash \mathtt{e}'_b \oplus \mathtt{e}'_c : t' \backslash \mathcal{C}'}\ (\text{Sub})$$

Now, since all reductions terminate, by induction on the height of the context derivation tree and on $\rightsquigarrow_h^*$, we have the following.

$\mathtt{e}_a, \mathbf{TD}_a, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow_h \mathtt{e}_b, \mathbf{TD}_b, \mathcal{C}_\top, H_b, \iota_b, \omega_b \rightsquigarrow_h^* \mathtt{e}_f, \mathbf{TD}_f, \mathcal{C}_\top, H_f, \iota_f, \omega_f$, and

$\mathtt{e}'_a, \mathbf{TD}'_a, \mathcal{C}'_\top, H'_1, \iota'_1, \omega'_1 \rightsquigarrow_h \mathtt{e}'_b, \mathbf{TD}'_b, \mathcal{C}'_\top, H'_b, \iota'_b, \omega'_b \rightsquigarrow_h^* \mathtt{e}'_f, \mathbf{TD}'_f, \mathcal{C}'_\top, H'_f, \iota'_f, \omega'_f$, and

$\mathtt{e}_f, \mathbf{TD}_f, \mathcal{C}_\top, H_f, \iota_f, \omega_f \simeq_{\mathtt{Low}} \mathtt{e}'_f, \mathbf{TD}'_f, \mathcal{C}'_\top, H'_f, \iota'_f, \omega'_f$.

So, by (Op-RC), $\mathtt{e}_b \oplus \mathtt{e}_c, \mathbf{TD}_e, \mathcal{C}_\top, H_b, \iota_b, \omega_b \rightsquigarrow_h^* \mathtt{e}_f \oplus \mathtt{e}_c, \mathbf{TD}_3, \mathcal{C}_\top, H_f, \iota_f, \omega_f$ and again by (Op-RC), $\mathtt{e}'_b \oplus \mathtt{e}'_c, \mathbf{TD}'_e, \mathcal{C}'_\top, H'_b, \iota'_b, \omega'_b \rightsquigarrow_h^* \mathtt{e}'_f \oplus \mathtt{e}'_c, \mathbf{TD}'_3, \mathcal{C}'_\top, H'_f, \iota'_f, \omega'_f$

Now, by assumption $\mathbf{TD}_a \rhd \mathtt{e}_a, \Gamma, pc_2, t_a, \mathcal{C}_a$, so by Subject Reduction Lemma 4.13, we have $\mathbf{TD}_f \rhd \mathtt{e}_f, \Gamma, pc_2, t_a, \mathcal{C}_a$,.

By (Op-RC), we have

$$\mathbf{TD}_3 = \frac{\dfrac{\mathbf{TD}_f \qquad \mathbf{TD}_g}{\Gamma, pc_2, \mathcal{H}_f \vdash \mathtt{e}_f \oplus \mathtt{e}_c : \langle\, s_a \cup s_c, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}_a \cup \mathcal{C}_c}\ (\text{Op})}{\Gamma, pc_1, \mathcal{H}_b \vdash \mathtt{e}_f \oplus \mathtt{e}_c : t \backslash \mathcal{C}}\ (\text{Sub})$$

where $\mathbf{TD}_g = tdheap(\mathbf{TD}_d, \mathcal{H}_f)$

Again by (Op-RC), we have

$$\mathbf{TD}'_3 = \dfrac{\dfrac{\mathbf{TD}'_f \qquad \mathbf{TD}'_g}{\Gamma', pc'_2, \mathcal{H}'_f \vdash \mathsf{e}'_f \oplus \mathsf{e}'_c : \langle\, s'_a \cup s'_c, \emptyset, \mathtt{int} \,\rangle \backslash \mathcal{C}'_a \cup \mathcal{C}'_c}\ \text{(Op)}}{\Gamma', pc'_1, \mathcal{H}'_b \vdash \mathsf{e}'_f \oplus \mathsf{e}'_c : t' \backslash \mathcal{C}'}\ \text{(Sub)}$$

where $\mathbf{TD}'_g = tdheap(\mathbf{TD}'_d, \mathcal{H}'_f)$

Since $\mathsf{e}_f, \mathbf{TD}_f, \mathcal{C}_\top, H_f, \iota_f, \omega_f \simeq_{\mathtt{Low}} \mathsf{e}'_f, \mathbf{TD}'_f, \mathcal{C}'_\top, H'_f, \iota'_f, \omega'_f$, and by Lemma 4.21, $\mathsf{e}_c, \mathbf{TD}_g, \mathcal{C}_\top \simeq_{\mathtt{Low}} \mathsf{e}_c, \mathbf{TD}_d, \mathcal{C}_\top$ and $\mathsf{e}'_c, \mathbf{TD}'_g, \mathcal{C}'_\top \simeq_{\mathtt{Low}} \mathsf{e}'_c, \mathbf{TD}'_d, \mathcal{C}'_\top$; so by Definition 4.15[2c], we have $\mathsf{e}_f \oplus \mathsf{e}_c, \mathbf{TD}_3, \mathcal{C}_\top, H_f, \iota_f, \omega_f \simeq_{\mathtt{Low}} \mathsf{e}'_f \oplus \mathsf{e}'_c, \mathbf{TD}'_3, \mathcal{C}'_\top, H'_f, \iota'_f, \omega'_f$.

We now have two cases, according to Definition 4.17 and Definition 4.16 (the next step must either be high or low).

(a) $\mathsf{e}_f \oplus \mathsf{e}_c, \mathbf{TD}_3, \mathcal{C}_\top, H_f, \iota_f, \omega_f \rightsquigarrow_l \varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2$.

Conclude with $\mathsf{e}_f \oplus \mathsf{e}_c, \mathbf{TD}_3, \mathcal{C}_\top, H_f, \iota_f, \omega_f \simeq_{\mathtt{Low}} \mathsf{e}'_f \oplus \mathsf{e}'_c, \mathbf{TD}'_3, \mathcal{C}'_\top, H'_f, \iota'_f, \omega'_f$.

(b) $\mathsf{e}_f \oplus \mathsf{e}_c, \mathbf{TD}_3, \mathcal{C}_\top, H_f, \iota_f, \omega_f \rightsquigarrow^*_h \varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2$.

Since $\mathsf{e}_f \oplus \mathsf{e}_c, \mathbf{TD}_3, \mathcal{C}_\top, H_f, \iota_f, \omega_f \simeq_{\mathtt{Low}} \mathsf{e}'_f \oplus \mathsf{e}'_c, \mathbf{TD}'_3, \mathcal{C}'_\top, H'_f, \iota'_f, \omega'_f$. By induction, $\mathsf{e}'_f \oplus \mathsf{e}'_c, \mathbf{TD}'_3, \mathcal{C}'_\top, H'_f, \iota'_f, \omega'_f \rightsquigarrow^*_h \varepsilon'_2, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_2, \iota'_2, \omega'_2$ and $\varepsilon_2, \mathbf{TD}_2, \mathcal{C}_\top, H_2, \iota_2, \omega_2 \simeq_{\mathtt{Low}} \varepsilon'_2, \mathbf{TD}'_2, \mathcal{C}'_\top, H'_2, \iota'_2, \omega'_2$.

16. Remaining (*-RC) cases follow a similar inductive argument to (Op-RC).

□

Now that we have proved all of our necessary lemmas for maintaining a bisimulation across both low and high steps, we can state our noninterference theorem. The theorem states that for a well-typed program ($\overline{\mathtt{s}}$ corresponds to `main`), two terminating runs with low-equivalent (bisimilar) initial input and output streams (for normal programs, output streams will initially be empty), then the resulting input and output streams will be low-equivalent.

The theorem assumes the input streams are fixed before execution, but applies to *any* stream of inputs. This differs from the technique employed by O'neill *et. al.* [OCC06] for reasoning about interactive information flow, which defines *user-strategies* to model the behavior of user inputs, allowing the user to observe the trace of previous inputs and outputs before giving the next input. While we do not model IO in this manner, our noninterference theorem is equivalent to their definition of interactive noninterference in the absence of non-deterministic and probabilistic choice. Since our theorem applies to *any* stream of inputs, we can select the particular input streams based on the user-strategies beforehand and achieve the same result.

Theorem 4.27 (Noninterference) *Suppose* $\mathbf{TD} = \emptyset, \emptyset, \emptyset \vdash \overline{\mathtt{s}} : \langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C}$*, and* $\iota_1 \simeq_{\mathtt{Low}} \iota_1'$*, and* $\omega_1 \simeq_{\mathtt{Low}} \omega_1'$*, and* $\overline{\mathtt{s}}, \mathbf{TD}, \mathcal{C}, \emptyset, \iota_1, \omega_1 \rightsquigarrow^* \mathtt{c}, \mathbf{TD}_c, \mathcal{C}, H_c, \iota_c, \omega_c$ *and* $\overline{\mathtt{s}}, \mathbf{TD}, \mathcal{C}, \emptyset, \iota_1', \omega_1' \rightsquigarrow^* \mathtt{c}', \mathbf{TD}_c', \mathcal{C}, H_c', \iota_c', \omega_c'$ *(i.e. both runs terminate), then* $\iota_c \simeq_{\mathtt{Low}} \iota_c'$ *and* $\omega_c \simeq_{\mathtt{Low}} \omega_c'$*. (and* $\mathtt{c} \simeq_{\mathtt{Low}} \mathtt{c}'$*)*

Proof. The theorem follows by repeated use of lemmas 4.26 and 4.23, using Subject Reduction Lemma 4.13.

Now, $\overline{\mathtt{s}}, \mathbf{TD}, \mathcal{C}, \emptyset, \iota_1, \omega_1 \rightsquigarrow^* \mathtt{c}, \mathbf{TD}_c, \mathcal{C}, H_c, \iota_c, \omega_c$ can be written as

$\overline{\mathtt{s}}, \mathbf{TD}, \mathcal{C}, \emptyset, \iota_1, \omega_1 \rightsquigarrow_h^* \varepsilon_2, \mathbf{TD}_2, \mathcal{C}, H_2, \iota_2, \omega_2 \rightsquigarrow_l \varepsilon_3, \mathbf{TD}_3, \mathcal{C}, H_3, \iota_3, \omega_3 \rightsquigarrow_h^* \ldots \rightsquigarrow_h^*$

$\mathtt{c}, \mathbf{TD}_c, \mathcal{C}, H_c, \iota_c, \omega_c$. The reduction $\rightsquigarrow^*$ consists of alternating $\rightsquigarrow_h^*$ and $\rightsquigarrow_l$ steps. The number of high steps in $\rightsquigarrow_h^*$ can also be zero.

Then using Subject Reduction Lemma 4.13 to fulfill the premises, according to Lemmas 4.26 and 4.23,

$\overline{\mathtt{s}}, \mathbf{TD}, \mathcal{C}, \emptyset, \iota_1', \omega_1' \rightsquigarrow_h^* \varepsilon_2', \mathbf{TD}_2', \mathcal{C}, H_2', \iota_2', \omega_2' \rightsquigarrow_l \varepsilon_3', \mathbf{TD}_3', \mathcal{C}, H_3', \iota_3', \omega_3' \rightsquigarrow_h^* \ldots \rightsquigarrow_h^*$

$\mathtt{c}', \mathbf{TD}_c', \mathcal{C}, H_c', \iota_c', \omega_c'$, such that $\varepsilon_2, \mathbf{TD}_2, \mathcal{C}, H_2, \iota_2, \omega_2 \simeq_{\mathtt{Low}} \varepsilon_2', \mathbf{TD}_2', \mathcal{C}, H_2', \iota_2, \omega_2$, and

$\varepsilon_3, \mathbf{TD}_3, \mathcal{C}, H_3, \iota_3, \omega_3 \simeq_{\mathtt{Low}} \varepsilon'_3, \mathbf{TD}'_3, \mathcal{C}, H'_3, \iota_3, \omega_3$, and so forth, until

$\mathtt{c}, \mathbf{TD}_c, \mathcal{C}, H_c, \iota_c, \omega_c \simeq_{\mathtt{Low}} \mathtt{c}', \mathbf{TD}'_c, \mathcal{C}, H'_c, \iota_c, \omega_c$.

Hence, regardless of whether the execution ends in a low or high step, lemmas 4.26 and 4.23 both conclude that the low input and output streams remain equivalent: $\omega_c \simeq_{\mathtt{Low}} \omega'_c$, and $\iota_c \simeq_{\mathtt{Low}} \iota'_c$, and the final values are bisimilar: $\mathtt{c} \simeq_{\mathtt{Low}} \mathtt{c}'$. The theorem follows.

□

4.6 Unaugmented Semantics and Noninterference

In order to fully justify the correctness of our information flow type system, we now describe an unaugmented semantics, under which these programs may actually execute, and prove that well-typed programs are noninterfering.

The unaugmented evaluation relation, $\rightarrow$, is identical to the evaluation via $\rightsquigarrow$, only lacking the type derivations. The heap also takes a slightly different form, M, since it does not align heap locations based on security labels. Other definitions are the same as in Figure 3.1. Reductions in the unaugmented semantics are of the form $\varepsilon_1, M_1, \iota_1, \omega_1 \rightarrow \varepsilon_1, M_2, \iota_2, \omega_2$. The reduction rules are presented in Figure 4.11 and Figure 4.12. The rules for reductions under context are similar to those in the augmented semantics, and are therefore omitted.

The bisimulation in Definition 4.28 shows the relationship between augmented and unaugmented expressions. They are all the same, apart from heap locations, which are related by a partial function β, mapping oids in the augmented heap to oids in the unaugmented heap.

(Field-R)	$o.\mathtt{f}, M, \iota, \omega \rightarrow v, M, \iota, \omega$
	where $M(o) = \mathtt{C}, F,$ and $F(\mathtt{f}) = v$
(Op-R)	$\mathtt{c} \oplus \mathtt{c}', M, \iota, \omega \rightarrow v, M, \iota, \omega$
	where $v = \mathtt{c} \oplus \mathtt{c}'$
(Cast-R)	$(\mathtt{D})\ o, M, \iota, \omega \rightarrow o, M, \iota, \omega$
	where $M(o) = \mathtt{C}, F,$ and $\mathtt{C} <: \mathtt{D}$
(IfTrue-R)	$\mathtt{if}\ (\mathtt{True})\ \{\overline{\mathtt{s_1}}\}\ \mathtt{else}\ \{\overline{\mathtt{s_2}}\}, M, \iota, \omega \rightarrow \{\overline{\mathtt{s_1}}\}, M, \iota, \omega$
(IfFalse-R)	$\mathtt{if}\ (\mathtt{False})\ \{\overline{\mathtt{s_1}}\}\ \mathtt{else}\ \{\overline{\mathtt{s_2}}\}, M, \iota, \omega \rightarrow \{\overline{\mathtt{s_2}}\}, M, \iota, \omega$
(Seq-R)	$;\overline{\mathtt{s}}, M, \iota, \omega \rightarrow \overline{\mathtt{s}}, M, \iota, \omega$
(Return-R)	$\mathtt{return}\ v;, M, \iota, \omega \rightarrow v, M, \iota, \omega$
(Block-R)	$\{\overline{\mathtt{s}}\}, M, \iota, \omega \rightarrow \overline{\mathtt{s}}, M, \iota, \omega$
(Assign-R)	$o.\mathtt{f} = v;, M, \iota, \omega \rightarrow ;, M[o \mapsto \mathtt{C}, F'], \iota, \omega$
	where $M(o) = \mathtt{C}, F,$ and $F' = F[\mathtt{f} = v]$
(New-R)	$\mathtt{new}\ \mathtt{C}(\overline{v}), M, \iota, \omega \rightarrow \overline{\mathtt{s'}}; \mathtt{return}\ o;, M', \iota, \omega$
	where $cnbody(\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{super}(\overline{\mathtt{e}}); \overline{\mathtt{s}})$ and $\mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\ldots\}$
	and $\overline{\mathtt{s'}} = [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}}$
	and $M' = M[o \mapsto \mathtt{C}, F],$ and $F = \{\overline{\mathtt{f} = \mathtt{null}}\}$ and $o = newref(M)$
(Invoke-R)	$o.\mathtt{m}(\overline{v}), M, \iota, \omega \rightarrow \overline{\mathtt{s'}}, M, \iota, \omega$
	where $mbody(\mathtt{C}, \mathtt{m}) = (\overline{\mathtt{x}}, \overline{\mathtt{s}})$ and $\overline{\mathtt{s'}} = [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\overline{\mathtt{s}}$
(Super-R)	$o.\mathtt{super}(\mathtt{C}, \overline{v}), M, \iota, \omega \rightarrow \overline{\mathtt{s'}}, M, \iota, \omega$
	where $cnbody(\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{super}(\overline{\mathtt{e}}); \overline{\mathtt{s}})$ and $\mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\ldots\}$
	and $\overline{\mathtt{s'}} = [\overline{\mathtt{x}} \mapsto \overline{v}, \mathtt{this} \mapsto o]\mathtt{this.super}(\mathtt{D}, \overline{\mathtt{e}}); \overline{\mathtt{s}}$
(Super-R′)	$o.\mathtt{super}(\mathtt{Object}), M, \iota, \omega \rightarrow ;, M, \iota, \omega$

$$mbody(\mathtt{C}, \mathtt{m}) = \begin{cases} (\overline{\mathtt{x}}, \overline{\mathtt{s}}) & \text{if } \mathtt{M} \in \overline{\mathtt{M}} \text{ and } \mathtt{M} = \mathtt{RT}\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\overline{\mathtt{s}}\} \\ mbody(\mathtt{D}, \mathtt{m}) & \text{otherwise} \end{cases}$$

where $\mathtt{C_0} = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\}$

$cnbody(\mathtt{C}) = (\overline{\mathtt{x}}, \overline{\mathtt{s}})$ if $\mathtt{K} = \mathtt{C}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\mathtt{super}(\overline{\mathtt{e}}); \overline{\mathtt{s}}\}$

where $\mathtt{C_0} = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\}$

newref(M) = $o = loc_i$ where $i - 1$ is the largest integer, such that $loc_{i-1} \in M$

Figure 4.11: Unaugmented Operational Semantics Reduction Rules

(Input-R)	$\texttt{read}_{\texttt{L}}(\texttt{fd}), M, \iota, \omega \rightarrow \texttt{c}, M, \iota', \omega$
	where $\texttt{L} = S$ and $\iota(\texttt{fd}, S) = \texttt{c}.\iota'(\texttt{fd}, S)$
(Output-R)	$\texttt{write}_{\texttt{L}}(\texttt{c}, \texttt{fd}), M, \iota, \omega \rightarrow ;, M, \iota, \omega'$
	where $\texttt{L} = S$ and $\omega'(\texttt{fd}, S) = \texttt{c}.\omega(\texttt{fd}, S)$
(InErr-R)	$\texttt{read}_{\texttt{L}}(\texttt{fd}), M, \iota, \omega \rightarrow \mathit{IOErr}, M, \iota, \omega$
	where $\texttt{L} \neq S$ and $\iota(\texttt{fd}, S)$
(OutErr-R)	$\texttt{write}_{\texttt{L}}(\texttt{c}, \texttt{fd}), M, \iota, \omega \rightarrow \mathit{IOErr}, M, \iota, \omega$
	where $\texttt{L} \neq S$ and $\omega(\texttt{fd}, S)$

Figure 4.12: Unaugmented Operational Semantics IO Reduction Rules

Definition 4.28 (Bisimulation of Augmented and Unaugmented)

1. *(Expressions / Statements). For a partial function* $\beta : \{\overline{o}\} \rightharpoonup \{\overline{o'}\}$*, then* $\varepsilon \sim_\beta \varepsilon'$ *iff either*

   (a) $\varepsilon = \varepsilon' = \texttt{CO}$*; or*

   (b) $\varepsilon = \varepsilon' = \mathit{IOErr}$*; or*

   (c) $\varepsilon = o$ *and* $\varepsilon' = \beta(o)$*; or*

   (d) $\varepsilon = \varepsilon' = \texttt{x}$*, for some* $\texttt{x}$*; or*

   (e) $\varepsilon = \varepsilon' = \texttt{this}$*; or*

   (f) $\varepsilon = \texttt{e}_1.\texttt{f}$, $\varepsilon' = \texttt{e}'_1.\texttt{f}$, *and* $\texttt{e}_1 \sim_\beta \texttt{e}'_1$*; or*

   (g) $\varepsilon = \texttt{e}_1 \oplus \texttt{e}_2$, $\varepsilon' = \texttt{e}'_1 \oplus \texttt{e}'_2$, *and* $\texttt{e}_1 \sim_\beta \texttt{e}'_1$, $\texttt{e}_2 \sim_\beta \texttt{e}'_2$*; or*

   (h) $\varepsilon = (\texttt{C})\ \texttt{e}_1$, $\varepsilon' = (\texttt{C})\ \texttt{e}'_1$, *and* $\texttt{e}_1 \sim_\beta \texttt{e}'_1$*; or*

   (i) $\varepsilon = \texttt{if}\ (\texttt{e}_1)\ \{\overline{\texttt{s}_2}\}\ \texttt{else}\ \{\overline{\texttt{s}_3}\}$, $\varepsilon' = \texttt{if}\ (\texttt{e}'_1)\ \{\overline{\texttt{s}'_2}\}\ \texttt{else}\ \{\overline{\texttt{s}'_3}\}$, *and* $\texttt{e}_1 \sim_\beta \texttt{e}'_1$, $\{\overline{\texttt{s}_2}\} \sim_\beta \{\overline{\texttt{s}'_2}\}$, $\{\overline{\texttt{s}_3}\} \sim_\beta \{\overline{\texttt{s}'_3}\}$*; or*

   (j) $\varepsilon = \overline{\texttt{s}}$, $\varepsilon = \overline{\texttt{s}'}$, *and* $\texttt{s} = \texttt{s}_1; \overline{\texttt{s}_2}$, $\texttt{s}' = \texttt{s}'_1; \overline{\texttt{s}'_2}$, $\texttt{s}_1 \sim_\beta \texttt{s}'_1$, $\overline{\texttt{s}_2} \sim_\beta \overline{\texttt{s}'_2}$*; or*

   (k) $\varepsilon = \texttt{return}\ \texttt{e}_1;$, $\varepsilon' = \texttt{return}\ \texttt{e}'_1;$, *and* $\texttt{e}_1 \sim_\beta \texttt{e}'_1$*; or*

   (l) $\varepsilon = \{\overline{\texttt{s}}\}$, $\varepsilon' = \{\overline{\texttt{s}'}\}$, *and* $\overline{\texttt{s}} \sim_\beta \overline{\texttt{s}'}$*; or*

(m) $\varepsilon = \mathtt{e}_1.\mathtt{f} = \mathtt{e}_2;$, $\varepsilon' = \mathtt{e}'_1.\mathtt{f} = \mathtt{e}'_2;$, *and* $\mathtt{e}_1 \sim_\beta \mathtt{e}'_1$, $\mathtt{e}_2 \sim_\beta \mathtt{e}'_2$*; or*

(n) $\varepsilon = \mathtt{new\ C}(\overline{\mathtt{e}})$, $\varepsilon' = \mathtt{new\ C}(\overline{\mathtt{e}'})$, *and* $\overline{\mathtt{e}} \sim_\beta \overline{\mathtt{e}'}$*; or*

(o) $\varepsilon = \mathtt{e}_1.\mathtt{m}(\overline{\mathtt{e}})$, $\varepsilon' = \mathtt{e}'_1.\mathtt{m}(\overline{\mathtt{e}})$, *and* $\mathtt{e}_1 \sim_\beta \mathtt{e}'_1$, $\overline{\mathtt{e}} \sim_\beta \overline{\mathtt{e}'}$*; or*

(p) $\varepsilon = o.\mathtt{super}(\mathtt{C}, \overline{\mathtt{e}})$, $\varepsilon' = o'.\mathtt{super}(\mathtt{C}, \overline{\mathtt{e}'})$, *and* $o \sim_\beta o'$, $\overline{\mathtt{e}} \sim_\beta \overline{\mathtt{e}'}$*; or*

(q) $\varepsilon = o.\mathtt{super}(\mathtt{Object})$, $\varepsilon' = o'.\mathtt{super}(\mathtt{Object})$, *and* $o \sim_\beta o'$*; or*

(r) $\varepsilon = \mathtt{read}_\mathtt{L}(\mathtt{e}_1)$, $\varepsilon' = \mathtt{read}_\mathtt{L}(\mathtt{e}'_1)$, *and* $\mathtt{e}_1 \sim_\beta \mathtt{e}'_1$*; or*

(s) $\varepsilon = \mathtt{write}_\mathtt{L}(\mathtt{e}_1, \mathtt{e}_2)$, $\varepsilon = \mathtt{write}_\mathtt{L}(\mathtt{e}'_1, \mathtt{e}'_2)$, *and* $\mathtt{e}_1 \sim_\beta \mathtt{e}'_1$, $\mathtt{e}_2 \sim_\beta \mathtt{e}'_2$*; or*

(t) $\varepsilon = \varepsilon' = ;$ *; or*

2. *(Heaps)* $M \sim_\beta H$ *iff the following two conditions hold:*

   (a) β *is a bijection between* $dom(\beta)$ *and* $rng(\beta)$*.*

   (b) $dom(\beta) = dom(M)$ *and* $rng(\beta) = dom(H)$

   (c) $\forall o \in dom(M)$*, if* $M(o) = \mathtt{C}, F$*, and* $F = \{\overline{\mathtt{f} = v}\}$*, then* $H(\beta(o)) = \mathtt{C}, F'$*, and* $F' = \{\overline{\mathtt{f} = v'}\}$ *and* $\overline{v} \sim_\beta \overline{v'}$*.*

We prove the equivalence of the evaluations of the augmented and unaugmented semantics in the following lemma.

Lemma 4.29 (Evaluation Equivalence)

If $\varepsilon_1, M_1, \iota_1, \omega_1 \rightarrow \varepsilon_2, M_2, \iota_2, \omega_2$*, and* $\mathbf{TD} \rhd \varepsilon'_1, H'_1, \Gamma, pc, t, \mathcal{C}$ *, and* $\varepsilon_1 \sim_\beta \varepsilon'_1$*, and* $H_1 \sim_\beta M_1$*, then there exists* β' *where* $\beta \subseteq \beta'$*, and there exists* $\mathcal{C}_\top$ *where* $\mathcal{C} \subseteq \mathcal{C}_\top$*, such that* $\varepsilon'_1, \mathbf{TD}, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \rightsquigarrow \varepsilon'_2, \mathbf{TD}', \mathcal{C}_\top, H_2, \iota_2, \omega_2$*, and* $\varepsilon_2 \sim_\beta \varepsilon'_2$*, and* $H_2 \sim_\beta M_2$*.*

Proof. Since $\mathbf{TD} \rhd \varepsilon_1', H_1', \Gamma, pc, t, \mathcal{C}$, by Soundness Theorem 4.14, we have $\varepsilon_1', \mathbf{TD}, \mathcal{C}_\top, H_1, \iota_1, \omega_1 \not\rightsquigarrow \mathit{CkFail}, \mathbf{TD}, \mathcal{C}_\top, H_1, \iota_1, \omega_1$. Conclude by induction on the context reduction tree of $\rightarrow$, using Definition 4.28.

□

The use of the soundness theorem is used in the proof of Lemma 4.29, since the augmented semantics has an a rule that may reduce a configuration to CkFail. However, Soundness Theorem 4.14 shows this will not happen for a well-typed expression, so the evaluation equivalence holds. This also means that in the following noninterference result for the unaugmented semantics, when we assume termination we are assuming only the usual types of non-termination do not occur, since there is no check failure in the semantics. In other words, all reads and writes that are not IOErr will occur, but the following Corollary ensures the low IO streams remain equivalent.

Corollary 4.30 (Noninterference) *Suppose* $\emptyset, \emptyset, \emptyset \vdash \overline{\mathtt{s}} : \langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C}$, *and* $\iota_1 \simeq_{\mathtt{Low}} \iota_1'$, *and* $\omega_1 \simeq_{\mathtt{Low}} \omega_1'$, *and* $\overline{\mathtt{s}}, \emptyset, \iota_1, \omega_1 \rightarrow^* \mathtt{c}, M_2, \iota_2, \omega_2$ *and* $\overline{\mathtt{s}}, \emptyset, \iota_1', \omega_1' \rightarrow^* \mathtt{c}', M_2', \iota_2', \omega_2'$, *then* $\iota_2 \simeq_{\mathtt{Low}} \iota_2'$ *and* $\omega_2 \simeq_{\mathtt{Low}} \omega_2'$.

Proof. Directly by Theorem 4.27 and Lemma 4.29. □

4.7 Formalizing Declassify

In Section 4.1, we discussed that it was necessary to exclude declassification from our proofs of noninterference, since noninterference requires *no* leakage, but declassification is an intentional leak. It is unfortunate to exclude declassification from our formal proofs, since it is an important part of practical programs, as most practical programs will permit some *small* leaks of high security

data. We now discuss some of the difficulties in formalizing declassification, and what can be done to avert bad declassifies.

Formalizing declassification is complicated by the need to distinguish *good* declassifications from *bad* ones, since good versus bad is not well-defined. One must answer the questions: which data can permit some leakage? and how much leakage is allowed? These are questions that only the programmer or end-user can answer about their data, so it is difficult to formalize and prove anything for all programs, since it requires reasoning about programmer *intent*. The issue is further complicated by the arduous task of measuring how much information a piece of code can leak [Mal07].

For our system, we can do two things. Firstly, we can prove that the manner in which our type system handles declassify is consistent with the operational semantics, by proving soundness of a semantics extended with declassify: that no well-typed program will produce a run-time check failure in the augmented operational semantics. We discuss this presently, in Section 4.7.1. Secondly, in Section 2.8.2, we show how declassifications can be restricted to method returns and shown in the API; this allows programmers and end-users to more easily see *where* declassifications may occur, and to inspect these methods to be sure that declassification is warranted. In Section 5.1, we discuss some of the related work on ensuring the correctness of declassification.

4.7.1 Type Soundness with Declassification

In this section, we show how our proofs can be extended to prove type soundness in the presence of declassification. The result is the same as Theorem 4.14, that well-typed programs experience no run-time check failures. This is not to say that the declassification is itself warranted,

rather only that the type system treatment of declassification is consistent with the run-time semantics.

The reason we did not include declassification in our previous soundness proof in Section 4.4 is due to the declassification of objects, and how this relates to noninterference. In our system, declassification of objects is actually a declassification of the object reference (oid), not a mutation of the security level of the object itself. Hence, in a statement `x = Declassify(o,High);` the security level is reduced only for the result of the declassify expression, which will in this case be `x`. Other references to the object will not be affected by this declassify.

The subject reduction proof becomes complicated by the (Oid) type rule (in Figure 4.2), which adds the security level from the heap environment to the typing. So, we cannot prove that the type is preserved across the reduction $\texttt{Declassify}(o, \texttt{High}), \ldots \rightsquigarrow o, \ldots$, since the declassified security label may re-appear after the reduction step, since it was in the heap environment. Hence, we need to change the (Oid) type rule to the following.

$$\frac{\mathcal{H}(o) = t_o \backslash \mathcal{C}_o}{\Gamma, pc, \mathcal{H} \vdash o : \langle\, pc, pc_i, f_o, \alpha_o \,\rangle \backslash \mathcal{C}_o} \text{(Oid')}$$

This simply means that the secrecy type in the heap environment is not put on the type of the oid. The reason we include it in the previous proofs is that it greatly simplifies our noninterference proofs, since we must reason about the low bisimilarity of the heaps. Using (Oid') is, however also sound, since the security type of the heap object will *always* be on the type of the oid (unless it is declassified), since the creation of the oid occurs under the same security context as the new object creation.

Therefore, we can reproduce the subject reduction and soundness results with this new (Oid') type rule in place of (Oid), and the following semantic rule for declassify (the reduction under context rule is omitted, as it is similar to the other context rules in Figure 4.7).

$$\text{(Declassify-R)} \quad \texttt{Declassify}(v, \texttt{L}), \mathbf{TD}, \mathcal{C}_\top, H, \iota, \omega \rightsquigarrow v, \mathbf{TD}', \mathcal{C}_\top, H, \iota, \omega$$
$$\text{where } \mathbf{TD}' = tdDeclassify(\mathbf{TD}, v, \mathcal{C}_\top)$$

We omit these proofs, since they are identical to those of Subject Reduction Lemma 4.13 and Soundness Theorem 4.14, only with these minor changes for declassification. The result is the following soundness theorem.

Theorem 4.31 (Soundness with Declassify)

If $\texttt{Declassify}(\texttt{e}, \texttt{L})$ *is allowed to be a sub-expression of* ε*, and* (Oid') *is used in place of* (Oid)*, and* $\mathbf{TD}$ *is a* $canonical$ *derivation ending in* $\emptyset, \emptyset, \emptyset \vdash \varepsilon : t \backslash \mathcal{C}$*, and* $\mathcal{C} \subseteq \mathcal{C}_\top$*, and* $\mathcal{C}_\top$ *is closed and consistent, then* $\varepsilon, \mathbf{TD}, \mathcal{C}_\top, \iota, \omega \not\rightsquigarrow^* CkFail, \mathbf{TD}', \mathcal{C}_\top, H', \iota', \omega'$*.*

Chapter 5

Related Work

Information flow control was first described by Lampson [Lam73], where he referred to it as the confinement problem. Denning and Denning later demonstrated that a static program analysis could be used to control information flows [DD77]. Since then, static analysis of information flow control systems has been the focus of much research [HR98, VSI96, VS97b, BN02, ABHR99, PC00]; Sabelfeld and Myers present a survey in [SM03]. Much of the literature focuses on proving formal results for small programming languages, though there has been some effort to define working systems. Flow Caml [PS02] is an information flow extension to Core ML. The Jif system provides information flow control for full Java [Mye99b, MZZ+01].

O'Neill *et. al.* describe an information flow security model for interactive IO using a simple imperative language [OCC06]. They demonstrate that a simple type system can be used to obtain noninterference in an interactive setting involving user strategies, then expand the model to incorporate nondeterministic choice and probabalistic noninterference. The focus of their work is to achieve probabalistic noninterference, which is why they consider user-strategies. We can prove our nonin-

terference result in a much simpler way, by considering *any* possible set of input streams a user may define. In the absence of nondeterminism, this is equivalent to considering user-strategies, since our theorem applies to any stream of inputs, including the input streams based on a user-strategy. To our knowledge, this is the only other information flow type system that formally models interactive IO. All other systems consider only a batch input and output model, where all inputs are available prior to program execution, and outputs are only available upon completion.

Jif [Mye99a, Mye99b] is unique as an information flow system since it covers essentially the full Java language, but it lacks a formal analysis. Hence, our approach is fundamentally different, since we are proving the correctness of our system by using a core subset of Java, and have not given an implementation. However, we now discuss how our system compares with Jif, even though the approach is quite different. In Jif, checks on IO channels are intermixed with the many other internal checks within a program (e.g. on function application, or assignment). Our system is designed to reduce the number of checks to IO points only. Jif provides parametric polymorphism and some inference of labels. Programs must be annotated with security labels, including label parameters for polymorphic classes. This creates a backward compatibility issue, where all code must be re-coded to introduce the proper annotations. Additionally, method overloading requires subclass types to conform to the types of the superclass.

In contrast, our type system infers all label types and parametric types, removing the need for additional program annotations. Our label types are inferred for existing code, meaning libraries can be used as is, provided the proper labels and checks are placed on the IO points in the program. Our concrete class analysis [Age95, PC94, WS01] tracks the concrete classes of objects through the program, allowing us to statically determine a conservative approximation of the runtime object.

This means overridden methods in the subclass can have different types from the superclass, and the type system will correctly distinguish the information flow controls on the different objects statically.

Banerjee and Naumann [BN02, BN03] prove a batch-model noninterference property for an information flow type system for a Java-like language using a denotational semantics. They provide an inference extension for libraries that are parameterized by security levels [SBN04]. This form of polymorphism resembles Jif's, requiring annotations in the form of label parameters. They also require polymorphic types for methods must be satisfied by all overriding methods. As mentioned above, we employ a more implicit polymorphism that requires no program modifications, and we prove soundness and interactive noninterference using an extensible operational approach.

Flow Caml [PS02] provides label type inference and parametric polymorphism for an information flow extension to Core ML. They prove soundness of type inference and a batch-model noninterference property. Our type system is significantly different, since it is based on an object-oriented language, which presents unique issues, (e.g. inheritance and dynamic dispatch) that do not arise in a functional language. For example, Flow Caml provides a let-polymorphism over security labels; in Java, this level of polymorphism is insufficient since Java has less immutability. As described in Section 2.6, we require more complex polymorphism to account for the object-oriented features of Java.

Hammer *et. al.* describe how program dependence graphs (PDGs) may be used for information flow control [HKS06]. They use PDGs to capture both data and control dependencies in Java programs, where nodes represent expressions and edges represent dependencies. Information flows can be checked by observing paths in the PDGs; if there is no path in the PDG between two

nodes, than there can be no information flows. Although the technical methods used are significantly different than our approach, the expressive power is roughly similar: our polymorphic types are approximately matched by the context-sensitivity in their analysis for example. Their system incorporates flow-sensitivity and we do not. While it is difficult to compare this approach, since it is radically different from type-based systems, one advantage of the type-based approach is that it provides a much more succinct definition. Further, PDG-based analyses tend not to scale as well to more realistic languages and large applications, and are much less compositional [PCFY07]. In comparison, our type system analyzes each class definition only once, in isolation, and the constraint closure works over a program's global constraint set.

5.1 Declassification

Since declassification is generally held as a necessity of practical information flow systems [SS05, MMG01], many works provide techniques for controlling declassifications, and several formalisms have been given in an effort to define which declassifications are safe, and to prove that programs written under such formal systems are safe with respect to these definitions (*e.g.* [ML97, CM04, BS06, MS04, ZM01]). Sabelfeld and Sands describe many of the issues and challenges of deliberate information leaks [SS05], and compare many of the these works by separating declassification models into "what," "who," "where," and "when" classifications.

In this dissertation, we do not attempt a formal model of declassification. However, the declassification policies described in Section 2.8.2 are related to other works that develop policies for information downgrading. It may be possible to incorporate some of this work on formalizing declassification in our system, which is a topic of future work.

Hicks *et. al.* define a concept of trusted declassification encapsulated by declassifier methods that downgrade information flows [HKMH06]. A global policy defines which declassifiers each principal trusts to downgrade their data. Considering their principal names are like our security labels, their declassifiers are like our top-level declassification policies, so the approach is the same: declassification is restricted to certain methods.

Alternatively, Li and Zdancewic [LZ05a] describe an information flow calculus where the downgrading policy is contained on the data. These policies specify how an integer can be downgraded (*e.g.* `i % 2` declares the parity of `i` can be downgraded). This policy specification corresponds to the dual of our policies: the information labels `L` describes which methods `m` can declassify the data, whereas our policies specify which methods `m` declassify which information labels `L`. While it remains to be seen which technique is preferred in practice, our mechanism follows the object-oriented philosophy, allowing downgrading at the class and method level, and showing it in the API.

5.2 Noninterference Proof Techniques

As shown in Chapter 4, our technique for proving noninterference uses a small-step operational semantics which includes a syntactic type derivation to show the low-equivalence of two program runs. Many works make use of low-equivalence to formalize security properties of information flows [SS01, Dam06] (especially in the concurrency domain [SS00a, BC02, FR02], which we do not consider); however, we believe ours is the first to utilize a syntactic version of the type derivation, and establish low-bisimulation of semantic configurations in this context. Some proofs of noninterference have been given for functional languages using a denotational semantics [HR98, ABHR99]

and a translation to a labeled operational semantics [PC00]; these proof techniques are mostly unrelated to ours. Most noninterference proofs of static type system are much simpler than ours, since they are generally over a small imperative language, without interactive IO. Notable exceptions are discussed below.

The technique used by O'Neill *et. al.* for proving noninterference in interactive IO bears some similarity to ours [OCC06]. They define a low-equivalence relation on execution traces and commands in a small-step operational semantics, and prove that low-equivalence is preserved by the execution of well-typed programs. Since their language and type system are much simpler than ours (lacking methods), they can achieve this result by reasoning only about the type of the current expression. Due to our highly polymorphic method types, we must use the entire type derivation tree to establish low-equivalence. In particular, low-equivalent method calls require identical contours in both runs; this alignment is difficult to achieve without the entire type derivations (see Chapter 4).

Other works employ a technique related to that just described O'Neill *et. al.* to show noninterference of static type systems, but with a batch model of IO. Volpano *et. al.* use a standard big-step operational semantics to prove noninterference by showing that the type system maintains the low-equivalence of the store across execution steps [VSI96, VS97b]. Again, due to the simplicity of the language and type system, they can also get by with using only about the type of the current expression in their proofs. Zdancewic *et. al.* extend this proof technique to a continuation-passing style (CPS) language [ZM02]. Here, the use of CPS allows a semantic program counter (pc) that is monotonically increasing except when a linear continuation is invoked, so they don't have to reason about a stack of pc's. Using a similar low-equivalence relation, the proof shows that whenever the pc is low, both runs take a step that preserves low-equivalence, and when pc is

high, the runs eventually return to low-equivalent configurations. This approach is similar to ours in separating low steps from high steps. However, our operational semantic model does not allow us to define of a monotonically increasing pc; for example, when the branch of a conditional finishes, the pc decreases. In contrast, we use the type derivation to keep track of the program counters, and thereby distinguish low and high steps. While their CPS system has some interesting properties, we prove our results over an actual subset of the Java syntax and use a direct operational semantics, which is much closer to how Java programs actually execute.

Banerjee *et. al.* also prove noninterference by establishing a low-equivalence relation [BN02, BN03]; however, their proof technique is significantly different from ours, since they use a denotational semantics instead of an operational one.

Pottier and Simonet use an alternate proof technique for showing the noninterference property of Flow Caml [PS02, PS03]. They define a non-standard small-step operational semantics for a language Core ML^2, which contains bracketed expressions, $\langle\, e_1 \mid e_2 \,\rangle$, which encapsulate two different high computations. These bracketed expressions are used to show the low-equivalence of the execution, as e_1 and e_2 each represent the high computation that differs between two runs of the program. Since any unbracketed (low) computation must be identical between runs, noninterference follows. Our proof technique is related in that both approaches utilize the type derivation to link the two runs together. Pottier and Simonet achieve this by combining the differing high computation in a single term using bracketed expressions, and the type system works over the two runs simultaneously, allowing low-equivalence to be shown. In our approach, the two runs remain distinct, and we include the type derivation in the semantics to show the runs are low-equivalent, emphasizing the relationship between the type system—which ensures the program security—and

the semantics. This gives us a more standard type system, since they must type programs in an unusual syntax with bracketed expression. Further, our operational semantic model is more faithful to a normal operational semantics; as shown in Section 4.6, by simply stripping the type derivation from the augmented semantics, we achieve a normal operational semantics.

Chapter 6

Conclusion

Security of sensitive information is an important requirement of many computer systems. Applications that manipulate such information must not leak it to unauthorized or unintended locations. Programming language-based mechanisms are a good way to enforce the security requirements of these applications.

This dissertation describes a static information flow type inference system for Middleweight Java and formally proves its correctness. Our type system provides a high level of polymorphism to promote IO-based, end-to-end security policies and code re-use in multiple security contexts. A top-level policy description adds to the usability of our system by clarifying the policy in the API and making it easier to add information flow controls to a program. Changes to Java programs are therefore minor, as only the underlying IO operations change.

The type system is proved correct with soundness and noninterference results for interactive IO. We introduce a new proof technique that uses a syntactic form of a program's type derivation. Proving noninterference requires the alignment of two runs of the program that differ only in high

inputs, ensuring that their low behavior is the same. The complexity of the type system, especially the high degree of polymorphism, necessitate the use of the entire type derivation trees to provide this alignment.

This new approach to IO-based end-to-end security in Java represents a substantial improvement in practical static information flow security and formal proofs justify our approach is correct. Security type inference and easily identifiable policies are a necessary step towards a more usable information flow system. The development of some applications in Jif [HAM06, AS05] and some recent work integrating information flow with SELinux [HRJM07] are evidence that information flow security is moving closer to real applications. We believe our system, and the principles behind it, will help pave the way for a more wide-spread use of information security in programs.

6.1 Future Work

Future directions of this work include implementing the type inference system, and obtaining empirical evidence regarding the usability of our approach. An implementation would require an expansion of the system to include more of Java's language features. For example, exceptions are heavily used in Java, and can introduce new control flows, and therefore new indirect leaks. Other works have already addressed this issue [Mye99a, PS02, VS97a], and we expect it should not be too difficult to add to our system. Other features include loops, switch statements, and additional ways to perform IO.

Different components of a system may have their own security requirements. As briefly mentioned in Chapter 1, our end-to-end security model may be generalized to component interfaces, which is the interaction between distinct system components. An important aspect of program secu-

rity involves the interaction of a process with other processes, and mutual agreement on the security policy between processes. Once a process *A* releases sensitive information to another process *B*, process *A* must rely on process *B* to ensure the security; if process *B* has a different security policy for some sensitive data, this may not happen. By using our end-to-end security policies and a component interface mechanism, we expect a compositional security system is not far off.

Bibliography

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[Age95] Ole Agesen. The cartesian product algorithm. In *Proceedings ECOOP'95*, volume 952 of *LNCS*. SV, 1995.

[AS05] Aslan Askarov and Andrei Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS '05)*, Milan, Italy, September 2005.

[BC02] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.*, 281(1-2):109–130, 2002.

[Ber07] Brian Bergstein. Monster breach teaches familiar lessons. Associated Press, Sept. 2 2007. `http://ap.google.com/article/ALeqM5h05bbusPvl8Xg_3PBZVK3niAQPlw`.

[Bib77] Kenneth J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corporation, Bedford, Massachusetts, April 1977.

[Bis07] Tricia Bishop. Debate growing over data security. *The Baltimore Sun*, Feb. 9 2007. `http://www.baltimoresun.com/news/local/bal-te.bz.encryption09feb09,0,6929836.story?coll=bal-local-headlines`.

[BN02] Anindya Banerjee and David Naumann. Secure information flow and pointer confinement in a java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 253–267, 2002.

[BN03] Anindya Banerjee and David Naumann. Using access control for secure information flow in a java-like language. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 155–169. IEEE Computer Society Press, 2003., 2003.

[BPP03] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory, April 2003.

[BS06] Niklas Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *15th European Symposium on Programming (ESOP)*, pages 180–196, 2006.

[CM04] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209, New York, NY, USA, 2004. ACM Press.

[COW00] Tom Christiansen, Jon Orwant, and Larry Wall. *Programming Perl*. O'Reilly, 3rd edition, July 2000.

[Dam06] Mads Dam. Decidability and proof systems for language-based noninterference relations. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 67–78, New York, NY, USA, 2006. ACM Press.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[FF86] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Language Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), 1986.

[FR02] Riccardo Focardi and Sabina Rossi. Information flow security in dynamic contexts. In *CSFW '02: Proceedings of the 15th IEEE workshop on Computer Security Foundations*, page 307, Washington, DC, USA, 2002. IEEE Computer Society.

[Fre05] French Security Incident Response Team Advisories. Veritas backup exec and netbackup remote file access vulnerability. `http://www.frsirt.com/english/advisories/2005/1387`, August 2005.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[GMPS97] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, 1997.

[HAM06] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. From Languages to Systems: Understanding Practical Application Development in Security-typed Languages. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC 2006)*, Miami, FL, December 11-15 2006.

[HKMH06] Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification: high-level policy for a security-typed language. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 65–74, New York, NY, USA, 2006. ACM Press.

[HKS06] Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for Java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006.

[HO03] Tomoyuki Higuchi and Atsushi Ohori. A static type system for jvm access control. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 227–237, New York, NY, USA, 2003. ACM Press.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, Jan. 1998.

[HRJM07] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference*, Santa Clara, CA, USA, June 2007.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[JIMK03] Timo Jokela, Netta Iivari, Juha Matero, and Minna Karukka. The standard of user-centered design and the standard definition of usability: analyzing iso 13407 against iso 9241-11. In *CLIHC '03: Proceedings of the Latin American conference on Human-computer interaction*, pages 53–60, New York, NY, USA, 2003. ACM Press.

[Kei07] Gregg Keizer. Hackers deface UN site. Computerworld, August 2007. `http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9030318`.

[KSRW04] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*, 2004.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[LMZ03] Peng Li, Yun Mao, and Steve Zdancewic. Information integrity policies. In *Workshop on Formal Aspects in Security and Trust (FAST)*, Sep. 2003.

[LZ05a] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 158–170, New York, NY, USA, 2005. ACM Press.

[LZ05b] Peng Li and Steve Zdancewic. Unifying confidentiality and integrity in downgrading policies. In *Foundations of Computer Security Workshop (FCS)*, 2005.

[Mal07] Pasquale Malacaria. Assessing security threats of looping constructs. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 225–235, New York, NY, USA, 2007. ACM Press.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.

[MMG01] John McLean, Jon Millen, and Virgil Gligor. Non-interference: Who needs it? In P. Ryan, editor, *CSFW '01: Proceedings of the 14th IEEE workshop on Computer Security Foundations*, page 237, Washington, DC, USA, 2001. IEEE Computer Society.

[MS04] Heiko Mantel and David Sands. Controlled Declassification based on Intransitive Noninterference. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems, APLAS 2004*, LNCS 3303, pages 129–145, Taipei, Taiwan, November 4–6 2004. Springer-Verlag.

[Mye99a] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[Mye99b] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, January 1999.

[MZZ+01] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java + information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2001.

[Nie93] Jakob Nielsen. *Usability Engineering*. Academic Press, Inc., San Diego, CA, 1993.

[OCC06] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 190–201, Washington, DC, USA, 2006. IEEE Computer Society.

[PC94] John Plevyak and Andrew Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 324–340, 1994.

[PC00] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 46–57, Montral, Canada, 2000.

[PCFY07] Marco Pistoia, Satish Chandra, Stephen Fink, and Eran Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2), May 2007.

[PS02] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 319–330, Portland, Oregon, January 2002.

[PS03] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, 2003.

[PSS01] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer Verlag, April 2001.

[Rob04] Paul Roberts. Cisco warns of wireless security hole. *Computerworld*, April 2004. `http://www.computerworld.com/securitytopics/security/holes/story/0,10801,92015,00.html`.

[SBN04] Qi Sun, Anindya Banerjee, and David A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *Proc. of the Eleventh International Static Analysis Symposium (SAS)*, volume 3148, pages 84–99. Lecture Notes in Computer Science, Springer-Verlag, August 2004.

[Sec05] Secunia. LineControl Java Client log messages password disclosure. `http://secunia.com/advisories/16817/`, Sept. 2005.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Jounal on Selected Areas in Communications*, 21(1), January 2003.

[Smi07] Gina Smith. 1,500 students' data left exposed. *The State*, Sept. 7 2007. `http://www.thestate.com/crime/story/166087.html`.

[SS00a] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW '00: Proceedings of the 13th IEEE workshop on Computer Security Foundations*, page 200, Washington, DC, USA, 2000. IEEE Computer Society.

[SS00b] Christian Skalka and Scott Smith. Static enforcement of security with types. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 34–45, New York, NY, USA, 2000. ACM Press.

[SS01] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher Order Symbol. Comput.*, 14(1):59–91, 2001.

[SS05] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 255–269, Washington, DC, USA, 2005. IEEE Computer Society.

[ST07] Scott F. Smith and Mark Thober. Improving usability of information flow security in Java. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 11–20, New York, NY, USA, 2007. ACM Press.

[Sun07] Sun Microsystems. Security vulnerability in the Sun Ray Server Software Admin GUI. `http://sunsolve.sun.com/search/document.do?assetkey=1-26-102779-1`, Jan. 2007.

[SW00] Scott Smith and Tiejun Wang. Polyvariant flow analysis with constrained types. In Gert Smolka, editor, *Proceedings of the 2000 European Symposium on Program-*

ming (ESOP'00), volume 1782 of *Lecture Notes in Computer Science*, pages 382–396. Springer Verlag, March 2000.

[Sym05] Symantec Corporation. Symantec brightmail antispam static database password. `http://securityresponse.symantec.com/avcenter/security/Content/2005.05.31a.html`, June 2005.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, January 1998.

[Tan07] William A. Tanenbaum. Sharing business information in a high-risk world. *New York Law Journal*, 2007.

[U.S02] U.S. Federal Trade Commission. Eli lilly settles FTC charges concerning security breach. Press Release, Jan. 18 2002. `http://www.ftc.gov/opa/2002/01/elililly.shtm`.

[U.S04] U.S. Department of Energy: Computer Incident Advisory Capability. CUPS information leak. `http://www.ciac.org/ciac/bulletins/p-014.shtml`, Oct. 2004.

[VS97a] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *CSFW '97: Proceedings of the 10th IEEE workshop on Computer Security Foundations*, page 156, Washington, DC, USA, 1997. IEEE Computer Society.

[VS97b] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.

[Wos03] Nathan Wosnack. Security Advisory - MyTaxexpress 2003. `http://securityvulns.com/docs4288.html`, Mar. 2003.

[WS01] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *European Conference on Object-Oriented Programming(ECOOP'01)*, Budapest, Hungary, June 2001.

[Yam07] Mari Yamaguchi. Japan navy raided over data leak scandal. Associated Press, Aug. 28 2007. `http://www.guardian.co.uk/worldlatest/story/0,,-6879935,00.html`.

[Zag07] Adam Zagorin. Anger over nuclear secrets leak. *Time Magazine*, June 14 2007. `http://www.time.com/time/nation/article/0,8599,1632905,00.html`.

[ZM01] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *CSFW '01: Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, page 5, Washington, DC, USA, 2001. IEEE Computer Society.

[ZM02] Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *Higher Order Symbolic Computation*, 15(2-3):209–234, 2002.

Vita

Mark Andrew Thober was born in Lincoln, Nebraska on January 30, 1978. He was also raised in Lincoln and graduated from Lincoln East High School in 1996. He received his B.S. in Computer Science and Mathematics from Nebraska Wesleyan University in 1999, with highest distinction. He then began his doctoral study at Johns Hopkins University under the guidance of Scott Smith. He now resides in Baltimore, Maryland with his wife Sarah.