Universität des Saarlandes
Faculty of Natural Sciences and Technology I
Max Planck Institut für Software Systeme

Asymmetric Secure Multi-Execution

Masterarbeit in Fach Informatik von / Master's Thesis in Computer Science by
Iulia Mihaela Bastys

angefertigt under der Leitung von / supervised by
Dr. Deepak Garg

begutachtet von / reviewed by
Dr. Deepak Garg
Prof. Dr.-Ing. Christian Hammer

Saarbrücken, 2016

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ______________________ ______________________

(Datum/Date) (Unterschrift/Signature)

Summary

Secure multi-execution (SME) is a promising black-box technique for enforcing information flow properties. Unlike traditional static or dynamic language-based techniques, SME satisfies noninterference (soundness) by construction and is also precise. SME executes a given program twice. In one execution, called the high run, the program receives all inputs, but the program's public outputs are suppressed. In the other execution, called the low run, the program receives only public inputs and declassified or, in the case where no declassification is used, default inputs as a replacement for the secret inputs, but its private outputs are suppressed.

This approach works well in theory, but, in practice, the program might not be prepared to handle the declassified or default inputs, as they may differ a lot from the regular secret inputs, or may contradict some invariant the program carries. As a consequence, the program may produce incorrect outputs or it may crash. To avoid this problem, existing work makes strong assumptions on the ability of the given program to robustly adapt to the declassified or default inputs, limiting the class of programs to which SME applies.

In this thesis, we lift this limitation by presenting Asymmetric-SME (A-SME), a modification of SME that gives up on the pretense that real programs are inherently robust to modified inputs. Instead, A-SME requires a variant of the original program that has been adapted (by the programmer or automatically) to react properly to declassified or default inputs. This variant, which we call the low slice, is used in A-SME as a replacement for the original program in the low run. The original program and its low slice must be related by a semantic correctness criteria, but beyond adhering to this criteria, A-SME offers complete flexibility in the construction of the low slice. We prove A-SME is sound even when the low slice is incorrect, and when the low slice is correct, then A-SME is also precise. Additionally, we show that if a program is policy compliant, then its low slice always exists, at least in theory.

A-SME's solution assumes that the low slice is provided and does not describe a general method for constructing it. Partly addressing this problem, we propose an automatic program transformation for obtaining the low slice in the specific case when the declassification policy blocks some (high) inputs at runtime, or when the (high) inputs are replaced by default values. The behavior of A-SME is the same in both scenarios, i.e. it provides the low run with the optional value `None` instead of the initial value for the high input.

The program transformation leverages an information flow type system for a dependency analysis. The analysis precisely tracks which intermediate outputs are influenced by the inputs assigned `None` and, as a result, the transformation returns a low slice robust to missing

inputs. Assuming the original program can be typed, we prove that the low slice is type-sound (preserves well-typedness) and correct (functionally equivalent to the original program). Additionally, we show that the transformation is monotone, i.e. executing the transformed program on more precise inputs (with fewer `None`s) yields more precise outputs.

On the side, we also improve the state-of-the-art in declassification policies by supporting policies that offer controlled choices to untrustworthy programs.

Acknowledgments

First and foremost I would like to thank my supervisor Dr. Deepak Garg for giving me the opportunity to be part of his group. It was a challenging, but rewarding experience to work with a person whose expertise and knowledge never ceased to surprise me. I owe Deepak much of the scientist I am today. *Thank you so much!*

Second, I would like to thank Prof. Christian Hammer for guiding me towards Deepak's group. If it weren't for his suggestion, I might have never ended up here. And I feel it's a good thing I did. *Vielen Dank!*

My friends in Saarbrüecken have my deepest gratitude and appreciation as they made the difficult moments pass easier and stood by me in all these years. *Mulţumesc mult! Vielen Dank! Efcharistó! Shukraan! Muchas gracias! Teşekkür ederim! Thank you! Ti blagodaram!*

I would have probably given up so many times before if it weren't for the peculiar support of my parents and typical encouragements of my sisters. *Mulţumesc din suflet!*

Last, I would like to thank a special friend, my husband Tomas. I am grateful for all the support he had given me, for saying the right things, for being there. *Aš tave myliu, Tomai!*

Contents

List of Figures

To my parents,

Introduction

Motivation

Secure systems often rely on information flow control (IFC) to ensure that an unreliable application cannot leak sensitive data to public outputs. The standard IFC policy is noninterference, which says that confidential or high inputs must not affect public or low outputs. Traditionally, noninterference and related policies have been enforced using static, dynamic, or hybrid analyses of programs [3, 9, 11, 12, 13, 21, 22, 31], but it is known that such analyses cannot be sound (reject all leaky programs) and precise (accept all leaky programs) simultaneously. Secure multi-execution or SME is a promising recent technique that attains both soundness and precision, at the expense of more computational power [16]. Additionally, SME is a *black-box* monitoring technique that does not require access to the program's source code or binary.

Briefly, SME runs two copies of the same program, called high and low, simultaneously. (We focus here on two security levels, but a generalization to several security levels is possible.) The low run is given only low (public) inputs and its high (secret) outputs are blocked. The high run is given both low and high inputs, but its low outputs are blocked. Neither of the two runs can both see high inputs and produce low outputs, so SME trivially enforces noninterference. Less trivially, it can be shown that if a program is noninterfering semantically, then SME does not change its output behavior, so SME is also precise. For emphasis, we restate the advantages of SME below:

Noninterference by design. Due to the separation of inputs in different runs, neither of the runs can both see high inputs and produce low outputs, hence noninterference is enforced by construction.

Black-box mechanism. SME is language-independent, it does not require access to the program's source code or binary. All it needs is access to the inputs and outputs, allowing any program with a distinguished syntax for inputs and outputs to be used.

Transparency. Running a noninterferent program with SME produces the same output as the original program, however the transparency is achieved *per-level*. Recent results show that precision across output levels can be obtained using barrier synchronization [28, 29].

Whereas SME may sound like the panacea for enforcing noninterference, it does have several drawbacks. First, running two programs instead of a single one obviously requires more computational power. However, SME has been implemented and tested in at least one large application, namely the web browser Firefox [8]. As CPU cores become cheaper, we expect SME to scale better and to be applied to other applications as well. Second, its deployment in practice faces a fundamental issue: Since the low run cannot be provided high inputs, what must it be provided instead? The classical SME design [16] proposes providing *dummy values* like 0 or *null* whenever the low run requires access to a sensitive input. Unfortunately, these values might not be a convenient choice in all cases and they could potentially lead to the program crashing or producing unexpected results.

A similar problem is encountered when declassification is added to SME. Noninterference is too strict for practical applications and, in order for an application to be useful, some release of information is needed. Traditionally, SME offers no support for declassification and, in general, adding declassification to SME is challenging due to the separation of information between the low and high runs. Although promising, the combination of declassification and SME is relatively understudied and the current approaches are to some extent unsatisfactory (see Subsection 3.6). In their seminal work on enforcing declassification policies with SME [34], Vanhoef *et al.* advocate providing *policy-declassified values* in place of high inputs. However, depending on the declassification policy, the high inputs received by the low run of the program might even have a different semantics than the actual high inputs (see Example 2). Consequently, the program must be aware of, and robust to, changes in its high inputs' semantics, otherwise the low run may crash or produce erroneous outputs. This is somewhat contrary to the spirit of SME, which aims to be sound and precise on all (unmodified) programs.

Asymmetric SME (A-SME)

The robustness requirement limits the programs to which SME can be applied in practice. To circumvent this limitation and to broaden the scope of enforcement of declassification policies with SME, we propose a modification of SME that gives up on the original design of executing the *same* program in *both* the high and low runs. Instead, a second program that has been adapted to use declassified inputs (or default inputs in the degenerate scenario where no declassification is required) in place of regular high inputs is used for the low run. This second program, which we call the *low slice*, may be constructed by the programmer or by slicing the original program automatically.

The resulting paradigm is what we call asymmetric SME or A-SME, and we expect it to successfully address the drawbacks of classic SME (with declassification). We reckon the disadvantages of SME and indicate how our method addresses them:

Dummy values and declassification. In the classical SME design, the low run is given a declassified value (or default inputs, such as 0 or *null* when no declassification is allowed) in place of regular high inputs. Since the low slice is designed according to the declassification policy, it will be prepared to handle any potential changes in the input semantics. Hence, by using in the low run a program (the low slice) that is adapted to receive modified high inputs, we solve this problem by default.

Program transformation

A-SME is a framework that enforces declassification policies with SME and which uses, in this regard, a variant of the original program—the low slice—adapted *in some way* to react properly to declassified or default inputs. In this thesis, we do not describe a general algorithm for obtaining the low slice, we mainly assume the low slice is provided by the programmer. However, we do take a step towards specifying how the program must be modified to obtain the low slice in the special case when a default value is provided for the high inputs, or when the declassification policy blocks some (high) inputs at runtime.

Our choice for the default value is the optional value `None`, and `None` is also returned by A-SME whenever the declassification policy ‘refuses’ to provide a value for some (high) inputs. (An optional value can be seen as a variable holding either a value, or no value–`None`.)

We obtain the low slice by means of an automatic program transformation, which given a *source* program, returns a *target* program. The novelty of our approach in program transformation is represented by leveraging an information flow type system for a dependence analysis. The analysis precisely tracks which intermediate outputs are influenced by the inputs assigned `None` and, as a result, the transformation returns a low slice robust to missing inputs.

Improving expressiveness of declassification policies

On the side, we improve the state-of-the-art in declassification policies. More specifically, we improve upon the work of Vanhoef *et al.* [34]. First, we allow declassification to depend on *feedback* from the program and, second, we allow the sensitivity of an input’s *presence* to depend on policy state. We explain these two points below.

Output feedback. We allow policy state to depend on program outputs. This feedback from the program to the policy permits the policy to offer the program controlled choices in what is declassified, without having to introspect into the state of the program. The following examples illustrate this:

Example 1. Consider a data server, which spawns a separate handler process for every client session. A requirement may be that each handler process declassifies (across the network) the data of at most one client, but the process may choose which client that is. With output feedback, the handler process can produce a special high output, seen only by the SME monitor, to name the client whose data the process wants to access. Subsequently, the policy will deny the low run any data not belonging to that client.

Example 2. Consider an outsourced audit process for income tax returns. A significant concern may be subject privacy. Suppose that the process initially reads non-identifying data about all returns (e.g., only gross incomes and pseudonyms of subjects), and then decides which 1% of the returns it wants to audit in detail. With output feedback, we may enforce a very flexible policy without interfering with the audit’s functionality: The low run of the audit process can see (and, hence, leak) the detailed data of *only* 1% of all audit forms, but it can choose *which* forms constitute the 1%.

State-dependent input presence. Like some prior work on SME [8], we consider a reactive setting, where the program being monitored reacts to inputs provided externally. In this setting, the mere *presence* of an input (not just its content) may be sensitive. SME typically handles sensitive input presence by not invoking the low run for an input whose presence is high [8, 34]. Generalizing this, our policies allow the decision of whether an input’s presence is high to depend on the policy state (i.e., on past inputs and outputs). This is useful in some cases, as the following example demonstrates:

Example 3. Consider a news website whose landing page allows the visitor to choose news feeds from topics like politics, sports, and social, and allows the user to interact with the feed by liking news items. When the user clicks one of these topics, its feed is displayed using AJAX, without navigating the user to another page. On the side, untrusted third-party scripts track mouse clicks for page analytics. A privacy-conscious user may want to hide her interaction with certain feeds from the tracking scripts. For example, the occurrence of a mouse click on the politics feed may be sensitive, but a similar click on the sports feed may not be. Thus, the sensitivity of mouse click presence on the page depends on the topic being browsed, making the sensitivity state-dependent.

Contributions

To summarize, in this thesis we make the following contributions:

- We introduce asymmetric SME (A-SME), a variant of SME that uses an additional program (the low slice) *adapted* to process declassified values in the low run (Chapter 3). This expands the set of programs on which declassification policies can be enforced precisely using SME.

- We increase the expressiveness of declassification policies in SME by supporting program feedback and state-dependent input presence (Chapter 2).

- We prove formally that A-SME enforcement is always secure (Section 3.3) and, given a correct low slice, also precise (Section 3.4). We show that if the program conforms to the policy then its low slice always exists, at least in theory (Section 3.5).

- We describe a program transformation based on an information flow type system for obtaining the low slice in the case when A-SME assigns a default value to the high inputs, or when it blocks some inputs at runtime (Chapter 4). We prove the low slice is type-sound (preserves well-typedness) and correct (functionally equivalent to the original program), and that the transformation is monotone (Section 4.5).

Declassification policies

This chapter describes the declassification policies. The first two sections are introductory, with Section 2.1 presenting a brief overview on (stateful) declassification policies and Section 2.2 describing the (reactive) programming model. Section 2.3 defines the declassification policies formally and Section 2.4 illustrates the expressiveness of our policies by means of several examples. We conclude with Section 2.5, by mentioning some related work in the area.

Overview

Secure systems often rely on information flow control (IFC) to ensure that applications do not leak sensitive data to public outputs. The standard IFC policy is non-interference, which says that low or public outputs should not be influenced by high or secret inputs. Under this requirement, even the trivial password authentication would be deemed insecure, as it violates the conservative property via an implicit flow. An incorrect guess at the password will provide a malicious user with the information that the password is not the one she guessed. Although minor, this represents a flow of information from the secret (high) password to the public (low) user, thus violating non-interference.

As a consequence, practical applications require some *intended* release of sensitive information, and in this regard, several approaches that allow for a controlled release of information were suggested: declassify statements [2, 3, 22, 28, 31, 34], condition-based declassification [11, 12, 13], state predicates [7, 15], or external declassification policies [9, 33, 34].

In this thesis, we focus on *reactive* programs and declassification policies specified separately from the monitored program's logic. The rationale for this focus is straightforward: both web and mobile applications are inherently reactive and, due to the open nature of the two platforms, applications cannot be trusted to declassify sensitive information correctly in their own code.

Our declassification policies are stateful, i.e. they maintain a state of their own which is independent from the monitored program's memory and which gets updated on every input *and output*. As in previous work [27, 28, 34], we distinguish between presence and content sensitivity of inputs. Though based on the stateful declassification policies defined by Vanhoef *et al.* [34], our policies improve expressivity, as they allow for output feedback and state-dependent input presence (Section 2.3).

Unlike previous work, we do not assume the program to be annotated with declassify statements and do not consider an additional declassification channel where the declassified values are stored and read from every time the program reaches a declassify annotation. Instead, the value declassified by the policy is sent to the program as a new input.

In Chapter 3 we describe a new method for enforcing the declassification policies. Before delving into the details of our policies and defining them formally, we first introduce the (reactive) programming model.

Programming model

Reactive programs are programs invoked by the runtime when an *input* is available from the program's environment. In response, the program produces a list of *outputs* and this input-output pattern repeats indefinitely. In processing every input, the program may update its internal *memory* and during the next invocation, the runtime passes the updated memory to the program. This allows past inputs to affect the response to future inputs. Reactive programs are a ubiquitous model of computing and web browsers, servers, and OS shells are all examples of reactive programs.

Let Input, Output, and Memory denote the domains of inputs, outputs, and memories for programs, and let $[\tau]$ denote a finite, possibly empty list of elements of type τ.

Definition 1 (Reactive program)**.** A reactive program p is a function of type $\text{Input} \times \text{Memory} \mapsto [\text{Output}] \times \text{Memory}$.

The program p accepts an input and its last memory and produces a list of outputs and an updated memory. We deliberately avoid introducing a syntax for the reactive programs, as the enforcement technique we will present in Chapter 3 is black-box and does not care about the syntax of the program it monitors. It only needs access to the inputs and outputs. Concretely, the program p may be written in any programming language with a distinguished syntax for inputs and outputs.

We use the letters i, I, O, and μ to denote elements of Input, [Input], [Output], and Memory. $p(i,\mu) = (O,\mu')$ means that the program p, when given input i in memory μ, produces the list of outputs O and the new memory μ'. A *run* of the program p, written E, is a finite sequence of the form $(i_1,O_1),\ldots,(i_n,O_n)$. The run means that starting from some initial memory, when the program is invoked sequentially on the inputs $i_1,\ldots,i_n$, it produces the output lists $O_1,\ldots,O_n$, respectively. For $E = (i_1,O_1),\ldots,(i_n,O_n)$, we define its projection to inputs $E|_i = i_1,\ldots,i_n$ and its projection to outputs $E|_o = O_1 +\!\!+ \ldots +\!\!+ O_n$, where ++ denotes list concatenation.

Formally, the semantics of a reactive program p are defined by the judgment $I,\mu \longrightarrow_p E$ (Figure 2.1), which means that program p, when started in initial memory μ and given the sequence of inputs I, produces the run E. Here, $i :: I$ denotes the list obtained by adding element i to the beginning of the list I. Note that if $I,\mu \longrightarrow_p E$, then $E|_i = I$ and $|E| = |I|$.

$$\frac{}{[\,],\mu \longrightarrow_p [\,]}\ \text{R1} \qquad\qquad \frac{p(i,\mu) = (O,\mu') \qquad I,\mu' \longrightarrow_p E}{i :: I,\mu \longrightarrow_p (i,O) :: E}\ \text{R2}$$

Figure 2.1: Reactive semantics.

Policy definition

As we previously mentioned, we consider security policies specified *outside* the program as independent stateful programs, whose state is completely disjoint from the monitored program's memory and is inaccessible to the program directly. The policy's state may be updated on every input and every output and in each state a declassified value may be produced. More formally,

Definition 2 (Policy $\mathscr{D}$)**.** A declassification policy $\mathscr{D}$ is a tuple $(S, \mathsf{upd}^i, \mathsf{upd}^o, \sigma, \pi)$, where:

- S is a possibly infinite set of states. Our policy examples (and metatheorems in Chapter 3) often specify the initial state separately.
- $\mathsf{upd}^i : S \times \text{Input} \to S$ and $\mathsf{upd}^o : S \times [\text{Output}] \to S$ are functions used to update the state on program input and output, respectively.
- $\sigma : S \to \text{Bool}$ specifies whether the *presence* of the last input is low or high. When $\sigma(s) = \mathsf{true}$, the input that caused the state to transition to s has low presence, else it has high presence.
- $\pi : S \to \text{Declassified}$ is the *projection* or *declassification* function that returns the declassified value for a given state. This value is provided as input to the low slice when $\sigma(s) = \mathsf{true}$ (see Chapter 3). Declassified is the domain of declassified values.

Policy examples

In this section we illustrate the expressivity of our policies by means of several realistic examples. We begin with an example taken from Vanhoef *et al.* [34] which highlights the advantage of having the policy state depend on inputs.

Example 4 (Declassification of aggregate inputs)**.** A browsing analytics script running on an interactive webpage records user mouse clicks to help the webpage developer optimize content placement in the future. A desired policy might be to prevent the script from recording every individual click and, instead, release the average coordinates of blocks of 10 mouse clicks. Listing 1 shows an encoding of this policy. The policy's internal state records the number of clicks and the sum of click coordinates in the variables cnt and sum, respectively. The policy's input update function upd^i takes the new coordinate x of a mouse click, and updates both cnt and sum, except on every 10th click, when the avg (average) is updated and cnt and sum are reset. The projection function π simply returns the stored avg. Finally, since the last average can always be declassified, the input presence function σ always returns true. The output update function upd^o is irrelevant for this example and is not shown.

Remark: We do not explicitly pass the internal state of the policy to the functions upd^i, upd^o, σ, and π, nor return it from upd^i and upd^o. This is a convention we use, as the state is implicitly accessible in the policy's state variables, such as cnt, sum, and avg.

The next example illustrates the use of the input presence function σ.

Example 5 (State-dependent input presence)**.** A news website allows the user to browse one of three possible topics: politics, sports, or social. A declassification policy monitoring the mouse

Listing 1 INPUT AGGREGATION

```
Policy state $s$ (local variables):
  cnt : int
  sum : int
  avg : int
Initialization: cnt = 0; sum = 0; avg = 0;
Update functions:
  $\mathsf{upd}^i$(MouseClick $x$) =
    case cnt of
      | 9 → {cnt = 0; avg = (sum + $x$)/10; sum = 0;}
      | _ → {cnt = cnt + 1; sum = sum + $x$;}
Presence decision function:
  $\sigma()$ = true.
Projection function:
  $\pi()$ = avg.
```

clicks can be the following: On the sports page, mouse clicks are not sensitive; on the social page, the average of 10 mouse click coordinates can be declassified (as in Example 4); on the politics page, not even the existence of a mouse click can be revealed.

Listing 2 shows an encoding of this policy. The policy records the current topic being browsed by the user in the state variable st, which may take one of four values: initial, politics, sports, and social. Upon an input (function upd^i), the value of the new policy state is established based on st. For st = sports, the click's coordinate x is stored in the variable last_click. For st = social, the policy mimics the behavior of Example 4, updating a click counter cnt, a click coordinate accumulator sum, and the average avg once in every 10 clicks. Importantly, when st = politics, the policy state is not updated (the input is ignored). A separate component of upd^i not shown here changes st when the user clicks on topic change buttons.

The input presence function σ labels the input to high when st $\in$ {politics, initial} (output is false) and to low otherwise. Hence, when the user is browsing politics, not even the presence of inputs is released.

The projection function π declassifies the last click coordinate last_click when the user is browsing sports and the average of the last block of 10 clicks stored in avg when the user is browsing social topics. The value returned by the projection function is irrelevant when the user is browsing politics or has not chosen a topic (because in those states σ returns high), so these cases are not shown.

The example below illustrates policy state dependence on program output, which allows feedback from the monitored program to the policy.

Example 6 (Output feedback: Data server)**.** Assume a data server handles the data of three clients — Alice, Bob, and Charlie. The policy is that the data of at most one of these clients may be declassified by a server process and the process may choose this one client. An encoding of the policy is shown in Listing 3. The policy tracks the process' choice in variable st, which can take one of the four values: none (choice not yet made), alice, bob, or charlie. To make the choice, the process produces an output specifying a user whose data it wants to declassify. The function upd^o records the server's choice in st if the process has not already made the choice

Listing 2 STATE-DEPENDENT INPUT PRESENCE

```
Policy state $s$ (local variables):
  $\mathsf{st} : \{\mathsf{initial}, \mathsf{sports}, \mathsf{politics}, \mathsf{social}\}$
  $\mathsf{cnt} : \mathsf{int}$
  $\mathsf{sum} : \mathsf{int}$
  $\mathsf{last\_click} : \mathsf{int}$
Initialization: $\mathsf{st} = \mathsf{initial};\ \mathsf{cnt} = 0;\ \mathsf{sum} = 0;\ \mathsf{last\_click} = 0;$
Update functions:
  $\mathsf{upd}^i(\text{MouseClick } x) =$
    case st of
      $|\ \mathsf{sports} \rightarrow \{\mathsf{last\_click} = x;\}$
      $|\ \mathsf{social} \rightarrow$
          case cnt of
            $|\ 10 \rightarrow \{\mathsf{cnt} = 1;\ \mathsf{sum} = x;\}$
            $|\ \_ \rightarrow \{\mathsf{cnt} = \mathsf{cnt} + 1; \mathsf{sum} = \mathsf{sum} + x;\}$
Presence decision function:
  $\sigma() =$
    case st of
      $|\ \mathsf{initial} \rightarrow \mathsf{false}$
      $|\ \mathsf{sports} \rightarrow \mathsf{true}$
      $|\ \mathsf{politics} \rightarrow \mathsf{false}$
      $|\ \mathsf{social} \rightarrow$ case cnt of $|\ 10 \rightarrow \mathsf{true}\ \ |\ \_ \rightarrow \mathsf{false}.$
Projection function:
  $\pi() =$
    case st of
      $|\ \mathsf{sports} \rightarrow \mathsf{last\_click}$
      $|\ \mathsf{social} \rightarrow \mathsf{sum}/10.$
```

(upd^o checks that st = none). When user data is read (i.e., a new input from the file system appears), the input update function upd^i compares st to the user whose data is read. If the two match, the read data d is stored in the policy state variable data, else *null* is stored in data. The projection function π simply declassifies the value stored in data.

The last example also illustrates feedback from the program to the policy.

Example 7 (Output feedback: Audit)**.** Assume an untrusted audit process which is initially provided with pseudonyms and non-sensitive information of several client records. Later it identifies a certain fraction of these records which must be declassified in full for further examination. In addition, we assume the audit process reads exactly 100 records and then selects 1 record to be declassified for further examination. Pseudonyms are simply indices into an array maintained by the policy. An encoding of the corresponding policy is shown in Listing 4. The policy variable count counts the number of records fed to the program so far. While count is less than 100, the input update function upd^i simply stores each input record i of five fields in the array records. When count reaches 100, the output update function upd^o allows the program to provide a single index idx, which identifies the record that must be declassified in full.

Listing 3 OUTPUT FEEDBACK: DATA SERVER

```
Policy state $s$ (local variables):
  $\mathsf{st} : \{\mathsf{none}, \mathsf{alice}, \mathsf{bob}, \mathsf{charlie}\}$
  $\mathsf{data} : \mathsf{file}$
Initialization: $\mathsf{st} = \mathsf{none}$; $\mathsf{data} = \mathit{null}$;
Update functions:
  $\mathsf{upd}^o(\text{RestrictAccessTo } \mathsf{user}) =$
    if $(\mathsf{st} = \mathsf{none})$ then
      case user of
        $\mid \text{Alice} \rightarrow \{\mathsf{st} = \mathsf{alice};\}$
        $\mid \text{Bob} \rightarrow \{\mathsf{st} = \mathsf{bob};\}$
        $\mid \text{Charlie} \rightarrow \{\mathsf{st} = \mathsf{charlie};\}$
  $\mathsf{upd}^i(\text{PrivateData } (\mathsf{user}, \mathsf{d})) =$
    if $(\mathsf{st} = \mathsf{user})$ then $\{\mathsf{data} = \mathsf{d};\}$ else $\{\mathsf{data} = \mathit{null};\}$
Presence decision function:
  $\sigma() = \mathsf{true}$.
Projection function:
  $\pi() = \mathsf{data}$.
```

The full record stored at this index is transferred to the variable declassified, the array records is erased and count is set to ∞ to encode the fact that the process has made its choice.

The projection function π reveals only the index and the gross income of the last input (at index $\mathsf{count} - 1$ in records) while count is not ∞. When count has been set to ∞, the single record chosen by the process is revealed in full through the variable declassified.

Related work

Systems that enforce pure noninterference are overly restrictive and disallow all leaks, be they intentional or not. Previous work focused on proving weaker versions of noninterference for applications that specify intended leaks. The most common approach for specifying controlled release of information is by inserting declassify annotations in the program, but other methods suggest condition-based declassification [11, 12, 13], declassification as state predicates [7, 15], or external declassification policies [21, 33, 34].

Dimensions of declassification. Sabelfeld and Sands [32] survey different methods for representing and enforcing declassification policies and provide a set of four *dimensions* for declassification models. These dimensions — *what*, *where*, *when*, and *who* — have been investigated significantly in literature. Policies often encompass a single dimension, such as *what* in delimited release [31], *where* in gradual release [2], or *who* in the context of faceted values [5], but sometimes also encompass more than one dimension, such as *what* and *where* in localized delimited release [3], or *what* and *who* in decentralized delimited release [22]. Our security policies encompass the *what* and *when* dimensions of declassification. We do not consider pro-

Listing 4 OUTPUT FEEDBACK: AUDIT

```
Policy state $s$ (local variables):
  records : array[100] $*$ array[5]
  count : int
  declassified : array[5]
Initialization: records $= null$; count $= 0$; declassified $= null$;
Update functions:
  $\mathsf{upd}^i(i) =$
    case count of
      | 100 = return;
      | $x$ = {records[$x$] $= i$; count $= x + 1$;}
  $\mathsf{upd}^o(\mathsf{idx}) =$
    case count of
      | 100 = {declassified = records[idx]; records $= null$; count $= \infty$;}
      | _ = return;
Presence decision function:
  $\sigma() =$ true
Projection function:
  $\pi() =$
    case count of
      | $\infty$ = declassified
      | _ = let (idx, name, address, phone, income) = records[count $-$ 1] in (idx, income)
```

grams with explicit declassify commands (in fact, we do not consider any syntax for programs[1]) and, hence, we do not consider the *where* dimension of declassification [28, 31, 34].

Levels of sensitivity. In many applications, not only is the content of a message sensitive, but also its presence. Following the approach of Rafnsson *et al.* [27], we distinguish between sensitivity levels of presence and content of messages. While in the approach of Vanhoef *et al.* [34] the sensitivity levels for input presence are known *a priori*, in our design inputs have high presence by default and policy function σ is the one that decides, based on the policy state, whether the presence of an input should be made visible to a low adversary or not.

Declassification policies as state predicates. Constanzo and Shao [15] use a separation logic for enforcing declassification policies based on Hoare logic, facilitating reasoning on programs specified in C-like, imperative languages. The policies and security guarantees are expressed through state predicates. We believe our policies are at least as expressive as the ones enforced by their system. The policies of Banerjee *et al.* [7] are also specified as state predicates, but enforced through a type system, rather than a program logic.

External policies. In the context of security policies specified separately from code, Li and Zdancewic [21] propose relaxed non-interference, a security property that applies to declassi-

[1] In Chapter 4, when we discuss the program transformation for obtaining the low slice, we introduce an explicit syntax for programs.

fication policies written in a separate language. The policies, expressed as lambda terms over inputs, are treated as security levels and enforced through a type system. The type system can enforce a wide range of declassification policies (also enforceable by A-SME), such as "data must be encrypted before sending it to the network", or "x and y are secrets, but their sum is public" [21]. However, their enforcement mechanism cannot handle policies representing expressions that result from global computations, such as $\lambda x : \texttt{Int}.\lambda p : \texttt{Int}.(x+p) \times p$ [21], while A-SME does.

Swamy *et al.* [33] also define policies separate from the program. Their policies are expressed as security automata in a new language called AIR (automata for information release) and are also enforced by a type system.

For a dynamic enforcement such as SME, Kashyap *et al.* [18] suggest, but do not develop, the idea of writing declassification policies as separate sub-programs. Our work ultimately draws some lineage from this idea. More inspiring were the policies of Vanhoef *et al.* [34] which we use as starting point when designing our declassification policies. However, while specifying the policies as external sub-programs, Vanhoef *et al.* also use declassify annotations in the program. The value to which the expression under a declassify annotation evaluates to is the value stored currently in the policy state (similar to our function π). In contrast, we do not consider any syntax for the programs and our policies send the declassified values as new inputs to the program.

Stateful declassification policies. Swamy *et al.* [33] define stateful policies expressed as security automata. The policies specify release obligations and transition states when one of these obligations is satisfied. When all obligations are fulfilled, then the automaton reaches an accepting state and performs a declassification. However expressive, these policies do not encompass the *when* dimension of declassification (i.e., do not specify *when* information must be released). As we mentioned earlier, the policies are enforced using a type system.

The language Paralocks [9] also supports stateful declassification policies enforced by a type system. There, the policies are represented as sets of Horn clauses, whose antecedents are called locks. Locks are predicates with zero or more parameters and they exhibit two states: opened (true) and closed (false). The type system statically tracks which locks are open and which locks are closed at every program point. As the authors indicate themselves, the Paralocks model can encode simple *what* policies, but satisfying the *what* dimension of declassification is not their main focus.

The conditional declassification policies of Chong and Myers are similar with the ones expressed by Paralocks, but more abstract, and also enforced using a type system [11, 12, 13].

The policies of Vanhoef *et al.* contain a stateful release function *release* used to declassify aggregate information about past inputs. While our policies can also declassify aggregate information about past inputs, in addition, they also allow for output feedback and state-dependent input presence. The output feedback is useful as it gives the program controlled choices in what should be declassified. Considering again Example 7, the policy declassifies the client record, but the audit process is the one that chooses which record should be declassified.

Asymmetric SME

In this chapter we describe the details of our dynamic technique—Asymmetric SME (A-SME)—we use for enforcing the policies described in the previous chapter. The chapter begins with a brief introduction to A-SME (Section 3.1) and a description of the semantics of A-SME (Section 3.2). We continue with proving that A-SME is always secure (Section 3.3) and, given a correct low slice, also precise (Section 3.4). In Section 3.5 we show that if the program conforms to the policy then its low slice exists, at least in theory. Section 3.6 compares A-SME with previous approaches on SME and Section 3.7 concludes with related work.

Overview

A-SME builds on classic SME, but uses different programs in the high and low runs (hence the adjective asymmetric). Classic SME—as described, for example by Vanhoef *et al.* [34]—enforces a declassification policy on a reactive program by maintaining two independent runs of the given program. The first run, called the high run, is invoked on every new input and is provided the new input as-is. The second run, called the low run, is invoked for an input only when the input's presence (as determined by the policy) is low. Additionally, the low run is not given the original input, but a projected (declassified) value obtained from the policy after the policy's state has been updated with the new input. Only high outputs are retained from the high run (these are not visible to the adversary) and only low outputs are retained from the low run (these are visible to the adversary). Since the low run sees only declassified values and the high run does not produce low outputs, it must be the case that the low outputs depend only on declassified values. This enforces a form of noninterference.

The problem with classic SME, which we seek to address by moving to A-SME, is that even though the low and the high runs execute the *same* program, they receive completely different inputs — the high run receives raw inputs, whereas the low run receives inputs created by the declassification policy. This leads to two problems. First, if the programmer is not aware that her program will run with SME, the low run may crash because it may not be prepared to handle the completely different types of the declassified inputs. Fundamentally, it seems impossible for the program to automatically adapt to the different inputs of the high and the low runs, because it gets no indication of which run it is executing in! Second, if the program tries to enforce the declassification policy internally (which a non-malicious program will likely do), then in the low run, the projection function is applied twice — once by the SME monitor and then internally by the program. In contrast, in a run without SME, the function is applied only once. As a consequence, one must assume that the function that implements declassification is idempotent (e.g., in the approach of Vanhoef *et al.* [34], this declassification function is called “project” and it must be idempotent). These two limitations restrict the scenarios in which SME can be used to enforce declassification policies.

To broaden the scope of enforcement of declassification policies with SME, we propose to do away with requirement that the same program be executed in the high and low runs of SME. Instead, we assume that a variant of the program that has been carefully crafted to use declassified inputs (not the raw inputs) exists. This variant, called the low slice, is used in the low run instead of the original program. The resulting paradigm is what we call asymmetric SME or A-SME. Before delving into the details of A-SME and its semantics, we give an intuition for the low slice.

Low slice

For a program p : Input × Memory ↦ [Output] × Memory, the low slice with respect to policy $\mathscr{D}$ is a program p_L : Declassified × Memory ↦ [Output] × Memory that produces the program's low outputs given as inputs values that have been declassified in accordance with policy $\mathscr{D}$. In other words, the low slice is the part of the program that handles only declassified data.

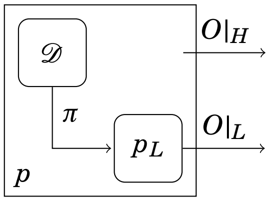


Figure 3.1: Factorization of program p into declassification policy $\mathscr{D}$ and low slice p_L.

A question that arises is why this low slice should even exist? Intuitively, if the program p is compliant with policy $\mathscr{D}$, then its low outputs depend only on the output of the policy $\mathscr{D}$. Hence, *semantically*, p must be equivalent to a program that composes $\mathscr{D}$ with some other function p_L to produce low outputs (Figure 3.1). It is this p_L that we call p's low slice. We formalize this intuition in Section 3.5 by proving that if the program p conforms to $\mathscr{D}$ (in a formal sense), then p_L must exist. However, note that the low slice p_L may not be syntactically extractable from the program p by any automatic transformation, in which case the programmer's help may be needed to construct p_L.

In the next chapter, we describe an automatic program transformation that produces the low slice for the case when the high inputs are assigned a default value, or when the declassification policy blocks some inputs at runtime. Both cases are handled in the same way, by using the optional value `None` as default value — when no declassification is used, and as value returned by function π — when declassification is employed.

A-SME Semantics

A-SME enforces a declassification policy $\mathscr{D}$ over a program p and its low slice p_L, together called an *A-SME-aware program*, written (p, p_L). The semantics of A-SME are defined by the judgment $I, s, \mu_H, \mu_L \models\!\!\Rightarrow^{\mathscr{D}}_{p,p_L} E$ (Figure 3.2), which should be read: “Starting in policy state

s and initial memories μ_H (for the high run) and μ_L (for the low run), the input sequence I produces the run E under A-SME and policy $\mathscr{D}$".

$$\frac{}{[], s, \mu_H, \mu_L \Longmapsto^{\mathscr{D}}_{p,p_L} []}\ \text{A-SME-1}$$

$$\frac{\begin{array}{c} s'' = \mathsf{upd}^i(s,i) \qquad \sigma(s'') = \mathsf{false} \\ p(i,\mu_H) = (O,\mu'_H) \qquad s' = \mathsf{upd}^o(s'',O) \qquad I, s', \mu'_H, \mu_L \Longmapsto^{\mathscr{D}}_{p,p_L} E \end{array}}{i :: I, s, \mu_H, \mu_L \Longmapsto^{\mathscr{D}}_{p,p_L} (i, O|_H) :: E}\ \text{A-SME-2}$$

$$\frac{\begin{array}{c} s'' = \mathsf{upd}^i(s,i) \qquad \sigma(s'') = \mathsf{true} \qquad p_L(\pi(s''),\mu_L) = (O',\mu'_L) \\ p(i,\mu_H) = (O,\mu'_H) \qquad s' = \mathsf{upd}^o(s'',O) \qquad I, s', \mu'_H, \mu'_L \Longmapsto^{\mathscr{D}}_{p,p_L} E \end{array}}{i :: I, s, \mu_H, \mu_L \Longmapsto^{\mathscr{D}}_{p,p_L} (i, O'|_L \mathbin{++} O|_H) :: E}\ \text{A-SME-3}$$

Figure 3.2: Semantics of A-SME.

We define the judgment by induction on the input sequence I. Rule A-SME-1 is the base case: When the input sequence I is empty, so is the run E (when there is no input, a reactive program produces no output). Rules A-SME-2 and A-SME-3 handle the case where an input is available. In both rules, the first available input, i, is given to the policy's input update function upd^i to obtain a new policy state s''. Then, $\sigma(s'')$ is evaluated to determine whether the input's presence is high or low (rules A-SME-2 and A-SME-3, respectively).

If the input's presence is high (rule A-SME-2), then only the high run is executed by invoking p with input i. The outputs O of this high run are used to update the policy state to s' (premise $s' = \mathsf{upd}^o(s'',O)$). After this, the rest of the input sequence is processed inductively (last premise). Importantly, any low outputs in O are discarded. The notation $O|_H$ denotes the subsequence of O containing all outputs on high (protected or non public) channels.

If the input's presence is low (rule A-SME-3), then, in addition to executing the high run and updating the policy state as described above, the low slice p_L is also invoked with the current declassified value $\pi(s'')$ to produce outputs O' and to update the low memory. Only the low outputs in O' ($O'|_L$) are retained. All high outputs in O' are discarded (those come from the high run).

We depict the semantics of A-SME pictorially in Figure 3.3. The dashed arrows denote the case where the input's presence is low (A-SME-3). In that case, the low slice executes with the declassified value returned by the policy function π. The arrow from the output O back to the policy $\mathscr{D}$ represents the output feedback.

Next, we prove formally that A-SME is (1) secure — it enforces policies correctly and has no false negatives (Section 3.3), and (2) precise — if p_L is a correct low slice, then its observable behavior does not change under A-SME (Section 3.4).

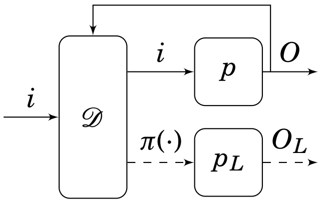


Figure 3.3: Pictorial representation of A-SME semantics.

Security

We prove security of A-SME by showing that a program running under A-SME satisfies a form of noninterference. Roughly, this noninterference says that if we take two different input sequences that result in the same declassified values, then the low outputs of the two runs of the program under A-SME are the same. In other words, the low outputs under A-SME are a function of the declassified values, so an adversary cannot learn more than the declassified values by observing the low outputs. Importantly, the security theorem makes no assumption about the relationship between p and p_L, so security holds even if a leaky program or a program that does not expect declassified values as inputs is provided as p_L.

To formally specify our security criterion, we first define a function $\mathscr{D}^*$ (Figure 3.4) that, given an initial policy state s and a program run E, returns the sequence of values declassified during that run.

$$
\begin{array}{rcll}
\mathscr{D}^*(s,[]) & = & [] & \\
\mathscr{D}^*(s,(i,O)::E) & = & \mathscr{D}^*(\mathsf{upd}^o(s'',O),E) & \text{if } s''=\mathsf{upd}^i(s,i) \text{ and } \sigma(s'')=\mathsf{false} \\
\mathscr{D}^*(s,(i,O)::E) & = & \pi(s''):: \mathscr{D}^*(\mathsf{upd}^o(s'',O),E) & \text{if } s''=\mathsf{upd}^i(s,i) \text{ and } \sigma(s'')=\mathsf{true}
\end{array}
$$

Figure 3.4: Function $\mathscr{D}^*$ returns values declassified by policy $\mathscr{D}$ during a run.

Function $\mathscr{D}^*$ is defined by induction on E and takes into account the update of the policy state due to both inputs and outputs in E. It is similar to a homonym in the approach of Vanhoef *et al.* [34], but adds policy state update due to outputs. Note that $\mathscr{D}^*$ adds the declassified value to the result only when the input presence is low (condition $\sigma(s'')=\mathsf{true}$). Equipped with the function $\mathscr{D}^*$, we state our security theorem.

Theorem 3 (Security, noninterference under $\mathscr{D}$)**.** *Suppose* $I_1,\mu_1 \longrightarrow_p E_1$ *and* $I_2,\mu_2 \longrightarrow_p E_2$ *and* $\mathscr{D}^*(s_1,E_1) = \mathscr{D}^*(s_2,E_2)$*. If* $I_1,s_1,\mu_1,\mu_L \Longmapsto^{\mathscr{D}}_{p,p_L} E'_1$ *and* $I_2,s_2,\mu_2,\mu_L \Longmapsto^{\mathscr{D}}_{p,p_L} E'_2$*, then* $E'_1|_o|_L = E'_2|_o|_L$*.*

Proof. By induction on the length of $I_1 +\!\!+ I_2$. See Appendix A.1, Theorem 22 for details. ■

The theorem says that if for two input sequences I_1, I_2, the two runs E_1, E_2 of a program p result in the same declassified values (condition $\mathscr{D}^*(s_1,E_1) = \mathscr{D}^*(s_2,E_2)$), then the program run under A-SME on I_1, I_2 will produce the same low outputs ($E'_1|_o|_L = E'_2|_o|_L$), for any low slice p_L.

Remark: Note that the precondition of the theorem is an equivalence on E_1 and E_2 obtained by execution under standard (non-A-SME) semantics, but its postcondition is an equivalence on E'_1 and E'_2 obtained by execution under A-SME semantics. This may look a bit odd at first glance, but this is the intended and expected formulation of the theorem. The intuition is that the theorem relates values declassified by the standard semantics to security of the A-SME semantics.

Precision

In the context of SME, precision means that for a non-leaky program, outputs produced under SME are equal to the outputs produced without SME. In general, SME preserves the order of outputs at a given level, but may reorder outputs across levels. For instance, the rule A-SME-3 in Figure 3.2 places all the low outputs $O'|_L$ before the high outputs $O|_H$. So, following prior work [34], we prove precision with respect to each level: We show that the sequence of outputs produced at any level under A-SME is equal to the sequence of outputs produced at the same level in the standard (non-A-SME) execution. Proving precision for high outputs is straightforward for A-SME.

Theorem 4 (Precision for high outputs)**.** *For any programs p and p_L, declassification policy $\mathscr{D}$ with initial state s, and input list I, if $I, \mu_H \longrightarrow_p E$ and $I, s, \mu_H, \mu_L \Longmapsto^{\mathscr{D}}_{p,p_L} E'$, then $E|_o|_H = E'|_o|_H$.*

Proof. From the semantics in Figures 2.1 and 3.2 it can be observed that the high run of A-SME mimics (in input, memory, and outputs) the execution under $\longrightarrow_p$. See Appendix A.2, Theorem 23 for details. ■

To show precision for low outputs, we must assume that the low slice p_L is *correct* with respect to the original program p and the policy $\mathscr{D}$. This assumption is necessary because A-SME uses p_L to produce the low outputs, whereas standard execution uses p to produce them. Recall that the low slice p_L is intended to produce the low outputs of p, given values declassified by policy $\mathscr{D}$. We formalize this intuition in the following correctness criteria for p_L.

Definition 5 (Correct low slice/correct low pair)**.** A program p_L of type Declassified × Memory $\mapsto$ [Output] × Memory and an initial memory μ_L are called a correct low pair (and p_L is called a correct low slice) with respect to policy $\mathscr{D}$, initial state s, program p, and initial memory μ if for all inputs I, if $I, \mu \longrightarrow_p E$ and $\mathscr{D}^*(s, E) = R$ and $R, \mu_L \longrightarrow_{p_L} E'$, then $E|_o|_L = E'|_o|_L$.

Based on this definition, we can now prove precision for low outputs.

Theorem 6 (Precision for low outputs)**.** *For any programs p and p_L, declassification policy $\mathscr{D}$ with initial state s, and input list I, if $I, \mu_H \longrightarrow_p E$ and $I, s, \mu_H, \mu_L \Longmapsto^{\mathscr{D}}_{p,p_L} E'$ and (μ_L, p_L) is a correct low pair with respect to $\mathscr{D}$, s, p, and μ_H, then $E|_o|_L = E'|_o|_L$.*

The proof of this theorem relies on the following easily established lemma.

Lemma 7 (Low simulation)**.** *Let $I, s, \mu_H, \mu_L \Longmapsto^{\mathscr{D}}_{p,p_L} E$ and $\mathscr{D}^*(s, E) = R$. If $R, \mu_L \longrightarrow_{p_L} E'$, then $E|_o|_L = E'|_o|_L$.*

Proof. By induction on I. Intuitively, the low run in A-SME is identical to the given run under $\longrightarrow_{p_L}$ and the high run of A-SME does not contribute any low outputs. See Appendix A.2, Lemma 24 for details. ■

Proof of Theorem 6. Let $R = \mathscr{D}^*(s,E')$ and $R,\mu_L \longrightarrow_{p_L} E''$. By Lemma 7, $E'|_o|_L = E''|_o|_L$. From Definition 5, $E|_o|_L = E''|_o|_L$. By transitivity of equality, we get that $E|_o|_L = E'|_o|_L$. ■

Theorem 8 (Precision). *For any programs p and p_L, declassification policy $\mathscr{D}$ with initial state s, and input list I, if $I,\mu_H \longrightarrow_p E$ and $I,s,\mu_H,\mu_L \overset{\mathscr{D}}{\Longrightarrow}_{p,p_L} E'$, and (μ_L,p_L) is a correct low pair with respect to $\mathscr{D}$, s, p and μ_H, then $E|_o|_L = E'|_o|_L$ and $E|_o|_H = E'|_o|_H$.*

Proof. Immediate from Theorems 4 and 6. ■

Existence of correct low slices

In this section we show that a correct low slice (more specifically, a correct low pair) of a program exists if the program does not leak information beyond what is allowed by the declassification policy.

Definition 9 (No leaks outside declassification). A program p starting from initial memory μ does not leak outside declassification in policy $\mathscr{D}$ and initial state s if for any two input lists I_1,I_2, if $I_1,\mu \longrightarrow_p E_1$ and $I_2,\mu \longrightarrow_p E_2$ and $\mathscr{D}^*(s,E_1) = \mathscr{D}^*(s,E_2)$, then $E_1|_o|_L = E_2|_o|_L$.

Theorem 10 (Existence of correct low slice). *If program p, starting from initial memory μ, does not leak outside declassification in policy $\mathscr{D}$ and initial state s, then there exist p_L and μ_L such that (μ_L,p_L) is a correct low pair with respect to $\mathscr{D}$, s, p, and μ.*

We describe a proof of this theorem. Fix an initial memory μ. Define f,g as follows: If $I,\mu \longrightarrow_p E$, then $f(I) = E|_o|_L$ and $g(I) = \mathscr{D}^*(s,E)$. Then, Definition 9 says that $f(I)$ is a function of $g(I)$, meaning that there exists another function h such that $f(I) = h(g(I))$. Intuitively, for a given sequence of declassification values $R = \mathscr{D}^*(s,E)$, $h(R)$ is the set of low outputs of p.

For lists L_1,L_2, let $L_1 \leq L_2$ denote that L_1 is a prefix of L_2.

Lemma 11 (Monotonicity of h). *If $I_1 \leq I_2$, then $h(g(I_1)) \leq h(g(I_2))$.*

Proof. By definition, $h(g(I_1)) = f(I_1)$ and $h(g(I_2)) = f(I_2)$. So, we need to show that $f(I_1) \leq f(I_2)$. Let $\mu,I_1 \longrightarrow_p E_1$ and $\mu,I_2 \longrightarrow_p E_2$. Since $I_1 \leq I_2$, $E_1|_o|_L \leq E_2|_o|_L$, i.e., $f(I_1) \leq f(I_2)$. ■

We now construct the low slice p_L using h. In the execution of p_L, the low memory μ'_L at any point is the list of declassified values R that have been seen so far. We define:

$$\begin{array}{rcl} \mu_L & = & [] \\ p_L(r,R) & = & (h(R::r) \setminus h(R), R::r). \end{array}$$

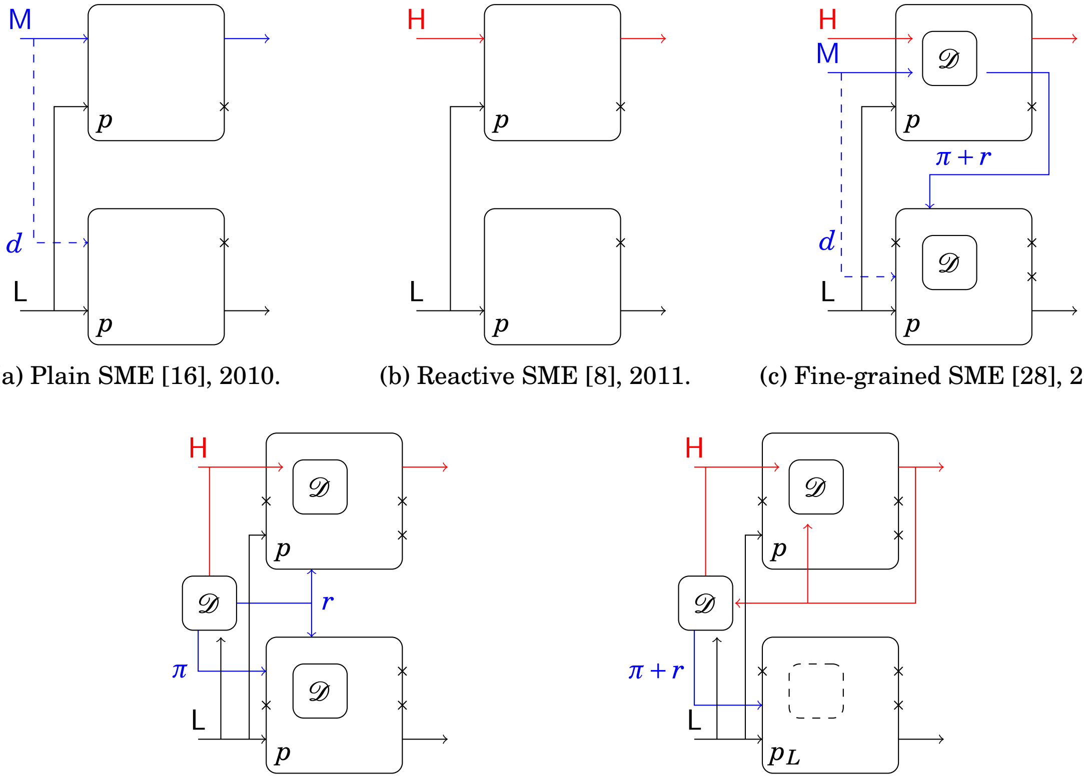
If R is the set of declassified values seen in the past, to produce the low output for a new declassified value r, we simply compute $h(R::r) \setminus h(R)$. By Lemma 11, $h(R) \leq h(R::r)$ when R and $R::r$ are declassified value lists from the same run of p, so $h(R::r) \setminus h(R)$ is well-defined. We then prove the following lemma, which completes the proof.

Lemma 12 (Correctness of construction)**.** (μ_L, p_L) *defined above is a correct low pair for* $\mathscr{D}$*,* s*,* p*, and* μ *if* p*, starting from initial memory* μ*, does not leak outside declassification in* $\mathscr{D}$ *and initial state* s*.*

Proof. See Appendix A.3, Lemma 29 for details. ■

A-SME vs. SME and its variants

In this section, we compare some of the fine points of A-SME with prior work on SME. We often refer to the schemas of Figure 3.5, which summarizes several flavors of SME described in the literature.



(a) Plain SME [16], 2010.

(b) Reactive SME [8], 2011.

(c) Fine-grained SME [28], 2013.

(d) SME with stateful declassification policies [34], 2014.

(e) Our A-SME, 2016.

Figure 3.5: Flavors of SME from literature. Red denotes information at level H, blue denotes information at level M, and black denotes information at level L. d is a default value provided to the low run when it demands an input of higher classification.

Input presence levels. SME was initially designed by Devriese *et al.* [16] to enforce noninterference on sequential programs, not reactive programs (Figure 3.5a). They implicitly assume that all inputs are low presence. Thus, there are only two kinds of inputs—low content/low presence (denoted L) and high content/low presence. Following the approach of Rafnsson *et al.* [27], we call the latter "medium"-level or M-level inputs, reserving high (H) for inputs with high presence.

Bielova *et al.* [8] adapted SME for enforcing noninterference in a reactive setting. Though not explicitly mentioned in their paper, their approach assumes that an input's presence and content are classified at the same level. Consequently, in their work, inputs only have levels H and L (Figure 3.5b). Bielova *et al.* also introduce the idea that for an input with high presence (level H), the low run must not be executed at all and we, as well as Vanhoef *et al.* [34] use this idea. In Bielova *et al.*'s work, an input's presence level is fixed by the channel on which it appears; this static assignment of input presence levels carries into all subsequent work, including that of Vanhoef *et al.* and of Rafnsson and Sabelfeld [28, 29]. Our work relaxes this idea and permits input presence to depend on policy state.

Input totality. Rafnsson and Sabelfeld (Figure 3.5c) consider all three input levels—L, M, and H—for sequential programs with I/O. In their setup, programs demand inputs and can time how long they wait before an input is available. This allows a conceptual distinction between environments that can always provide inputs on demand and environments that cannot. In an asynchronous reactive setting like ours, that of Vanhoef *et al.*, or that of Bielova *et al.*, this distinction is not useful.

Declassification and SME. Early work on SME, including that of Devriese *et al.* [16] and Bielova *et al.* [8], did not consider declassification. Rafnsson and Sabelfeld [28, 29] and Vanhoef *et al.* [34] added support for declassification in the non-reactive and reactive setting, respectively. In the former approach, the declassification policies have two components: a coarse-grained policy ρ specifies the flows allowed between levels statically and is enforced with SME; a fine-grained mechanism allows the high run of the program to declassify data to the low run dynamically. This mechanism routes data from a special M-level output of the high run to an M-level input of the low run. This routing is called the *release channel* and is denoted by $\pi + r$ in Figure 3.5c. Data on the release channel is not monitored by SME and the security theorem for such release is the standard gradual release condition [2], which only says that declassification happens at explicit declassification points of the high run, without capturing *what* is released very precisely. For instance, if Example 4 (see Section 2.4) were implemented in the framework of Rafnsson and Sabelfeld, the only formal security guarantee we would get is that *any* function of the mouse clicks might have been declassified (which is not useful in this example).

In contrast, the security theorem of Vanhoef *et al.*, like ours, captures the declassified information at fine granularity. Their policies declassify high inputs using two different functions—a stateless projection function *project*, which specifies both the presence level of an input and a declassified value, and a stateful release function *release* that can be used to declassify aggregate information about past inputs. The output of the projection function (denoted π in Figure 3.5d) is provided as input to the low run in place of the high input. The decision to pass a projected value to the low run where a high input is normally expected results in the problems mentioned in the overview section of this chapter (Section 3.1), which motivated us to design A-SME. The output of the release function (denoted r) is passed along a release channel similar to the one in Rafnsson and Sabelfeld. We find the use of two different channels redundant and thus we combine *release* and *project* into a single policy function that we call π.

Going beyond the approach of Vanhoef *et al.*, in A-SME, the policy state may depend on program output and the input presence may depend on policy state. As illustrated in Chapter 2, this allows for richer declassification policies.

Totality of the monitored program. Similar to Vanhoef *et al.*, we assume that the (reactive) program being monitored is total and terminates in a finite amount of time. This rules out leaks due to the adversary having the ability to observe lack of progress, also called progress-sensitivity [1, 23]. In contrast, Rafnsson and Sabelfeld do not make this termination assumption. Instead, they (meaningfully) prove progress-sensitive noninterference. This is nontrivial when the adversary has the ability to observe termination on the low run, as a scheduler must be chosen carefully. We believe that the same idea can be applied to both the work of Vanhoef *et al.* and our work if divergent behavior is permitted.

Related work

Secure multi-execution. We discussed prior work on SME in the previous section. Here, we mention some other work on related techniques. Khatiwala *et al.* [19] propose *data sandboxing*, a technique which partitions the program into two slices, a private slice containing the instructions handling sensitive data, and a public slice that contains the remaining instructions and uses system call interposition to control the outputs. The public slice is very similar to our low slice, but Khatiwala *et al.* trust the low slice's correctness for *security* of enforcement, while we do not. Nonetheless, we expect that the slicing method used by Khatiwala *et al.* to construct the public slice can be adapted to construct low slices for use with A-SME.

Capizzi *et al.* [10] introduce *shadow executions* for controlling information flow in an operating system. They suggest running two copies of an application with different sets of inputs: a *public copy*, with access to the network, that is supplied dummy values in place of the user's confidential data, and a *private copy*, with no access to the network, that receives all confidential data from the user.

Zanarini *et al.* [37] introduce *multi-execution monitors*, a combination of SME and monitoring, aimed at reporting any actions that violate a security policy. The multi-execution monitor runs a program in parallel with its SME-enforced version. If the execution is secure, the two programs will run in sync, otherwise, when one version performs an action different from the other, the monitor reports that the program is insecure. However, no support for declassification is provided.

Faceted and sensitive values. Faceted values [5] are a more recent, dynamic mechanism for controlling information flow. They are inspired by SME, but reduce the overhead of SME by simulating the effect of multiple runs in a single run. To do this, they maintain values for different levels (called facets) separately. For a two-level lattice, a faceted value is a pair of values. Declassification corresponds to migrating information from the high facet to the low facet. We expect that in A-SME, the use of the low slice in place of the original program in the low run will result in a reduction of overhead (over SME), comparable to that attained by faceted values.

Jeeves [36] is a new programming model that uses sensitive values for encapsulating a low- and a high-confidentiality view for a given value. Like faceted values, sensitive values are pairs of values. They are parameterized with a level variable which determines the view of the value that should be released to any given sink. Jeeves' policies are represented as declarative rules that describe when a level variable may be set high or low. The policies ensure data confidentiality, but offer no support for declassification. An extension of Jeeves with faceted values [6] can enforce more expressive declassification policies, but output feedback is still not supported.

(Stateful) Declassification policies enforcement mechanisms. As already mentioned in the related work section of the former chapter (Section 2.5), several previous approaches already described methods for enforcing (stateful) declassification policies. However, most of these are static enforcements via type-systems [9, 11, 12, 13, 21, 33], which do not apply in the reactive/dynamic setting of browsers or mobile applications, which are the focus of this thesis.

Generic black-box enforcement. Recently, Ngo *et al.* [26] have shown that black-box techniques based on multi-execution can be used to enforce not just noninterference and declassification policies, but a large subset of what are called hyperproperties [14]. They present a generic construction for enforcing any property in this subset. Superficially, their construction may look similar to A-SME, but it is actually quite different. In particular, their method would enforce noninterference by choosing a second input sequence that results in the same declassified values as the given input sequence to detect if there is any discrepancy in low outputs. A-SME does not use such a construction and is closer in spirit to traditional SME.

Program transformation

This chapter describes an automatic program transformation based on an information flow type system that generates a low slice in the case when the declassification policy blocks some (high) inputs at runtime, or when the (high) inputs are replaced by default values. Section 4.1 gives a short overview of the program transformation, Section 4.2 presents the program syntax and semantics, and Section 4.3 introduces the information flow type system. In Section 4.4 we discuss the transformation in more detail, while in Section 4.5 we prove its type-soundness and correctness. We conclude with Section 4.6 by mentioning some related work in the area.

Overview

Security policies or techniques that enforce them, such as SME, can block access to data, thus preventing an application from receiving certain inputs. Using dummy values such as 0 or *null* to replace the sensitive inputs can prove unfortunate in some cases. Thus, in order to prevent an application from misbehaving (crashing or producing erroneous results), we suggest to replace the dummy values with the optional value `None` and to modify the program to handle correctly the new inputs. An optional value is similar to a variable that holds either a value, or no value. (More details in Section 4.4.) Similarly, the behavior of policy $\mathscr{D}$ blocking some inputs at runtime is represented by function π evaluating to `None` for those inputs whose value is not to be revealed.

The idea of the transformation we propose is to take a *source* program, assign an Option type (which encapsulates an optional value) to the *volatile* inputs (inputs for which the system or the policy might return `None`), and, by tracking how these inputs affect the computation, modify the source program, and return a *target* program. The program transformation leverages an information flow type system for a dependence analysis. The analysis precisely tracks which intermediate outputs are influenced by the inputs assigned `None` and, as a result, the transformation returns a target program (low slice) robust to missing inputs.

Briefly, the program transformation follows four steps. In the first step, the programmer assigns label L to the types of the *non-volatile* inputs, i.e. inputs for which an initial value is always present at runtime, and label H to the types of the volatile inputs. We would like to note that the labels L and H do not necessarily represent security levels that detect illegal flows of information (even though, in this case, the labeling might be confounded with the one made by policy $\mathscr{D}$), but merely the prospect of an input to not be available during runtime, which can possibly disrupt the normal execution of the program. In the second step, we propagate labels L and H throughout the program by means of a flow-sensitive type system. In the third step, we remove the labels from the L-labeled types and, transform the H-labeled types into (non-labeled) Option types. In the fourth and final step, the actual low slice (target program) is obtained by means of a type-directed transformation.

The first two steps are necessary for proving transformation type-soundness and correctness. Type-soundness means well-typedness preservation, i.e. if the source program type-checks to some labeled type, then the target program type-checks to a non-labeled type related to the labeled type. Correctness means that the source and target programs are functionally equivalent, i.e. evaluating the source and target programs under equivalent *complete* memories returns equivalent values. (A memory is complete if values are provided for all volatile inputs.) Finally, we prove that the transformation is monotone, i.e. executing the target program on more precise inputs (with fewer Nones) yields more precise outputs.

Syntax and semantics

In the previous chapters we considered the programs to be black-boxes and did not introduce any specific syntax for them. In this chapter, however, we have to deal with label propagation and for this reason we need to take a closer look at the program structure and make explicit assumptions about it.

We consider a while-language extended with a label upgrade operator • (Figure 4.1a). This operator has no computational effect on the expression e it precedes (rule E-UPGRADE in Figure 4.1b), its only purpose is to upgrade from L to H the label of expression e's type. In Section 4.3 we will explain in more detail the role of this operator and the intuition behind it.

As usual, n ranges over integers, l ranges over memory locations, x ranges over variables, and $\oplus$ ranges over arithmetic operations on expressions. μ is a map from memory locations l and variables x to values v. (It is also possible to have the store for locations separated from the environment for variables. Using a single store μ for both memory locations and variables was simply our design choice.)

Sequence and if-expressions can be encoded in the language, as illustrated below:

$$\begin{array}{rcl} e_1;e_2 & \equiv & \texttt{let}\ x=e_1\ \texttt{in}\ e_2,\ x \notin FV(e_2) \\ \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \equiv & \texttt{case}\ e\ \texttt{of}\ \texttt{inl}\ \_.\ e_1 \quad \texttt{inr}\ \_.\ e_2 \end{array}$$

where $FV(e_2)$ denotes the set of free variables in expression e_2.

As can be seen from the semantic rules in Figure 4.1b, the conditional of the while expression has no side effects. However, this does not make our language less expressive, as a regular while expression while e_1 do e_2 can also be represented as $x := e_1$;while x do $\{e_2; x := e_1\}$.

Label propagation

As we already mentioned, the first step in performing the translation is labeling the inputs' types, or labeling the typing environment: we label each type with labels H or L, depending on whether the corresponding memory location is marked as volatile or not[1]. For example, if

[1] Unlike in Chapter 3, in this chapter we do not explicitly consider the program to be invoked sequentially on a list of inputs. We assume locations are 'assigned' in the initial memory for all the 'expected' inputs, be they volatile or not. If no value is provided for a volatile input, then the corresponding memory location remains empty, with $\bot$ denoting no value is assigned to that location.

$$\begin{array}{ll}
\text{Values} & v ::= n \mid \mathtt{inl}\ v \mid \mathtt{inr}\ v \mid () \\
\text{Expressions} & e ::= v \mid l \mid x \mid e \oplus e \mid l := e \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid \mathtt{inl}\ e \mid \mathtt{inr}\ e \\
 & \quad (\mathtt{case}\ e\ \mathtt{of}\ \mathtt{inl}\ x.\ e_1 \quad \mathtt{inr}\ x.\ e_2) \mid \mathtt{while}\ x\ \mathtt{do}\ e \mid \bullet e
\end{array}$$

(a) Syntax.

E-VAL
$$\frac{}{\langle v,\mu\rangle \Downarrow \langle v,\mu\rangle}$$

E-VAR
$$\frac{x \in dom(\mu)}{\langle x,\mu\rangle \Downarrow \langle \mu(x),\mu\rangle}$$

E-VARLOC
$$\frac{l \in dom(\mu)}{\langle l,\mu\rangle \Downarrow \langle \mu(l),\mu\rangle}$$

E-EXP
$$\frac{\langle e_1,\mu\rangle \Downarrow \langle n_1,\mu_1\rangle \qquad \langle e_2,\mu_1\rangle \Downarrow \langle n_2,\mu_2\rangle}{\langle e_1 \oplus e_2,\mu\rangle \Downarrow \langle n_1 \oplus n_2,\mu_2\rangle}$$

E-ASSIGN
$$\frac{\langle e,\mu\rangle \Downarrow \langle v,\mu'\rangle}{\langle l := e,\mu\rangle \Downarrow \langle (),\mu'[l \mapsto v]\rangle}$$

E-INL
$$\frac{\langle e,\mu\rangle \Downarrow \langle v,\mu'\rangle}{\langle \mathtt{inl}\ e,\mu\rangle \Downarrow \langle \mathtt{inl}\ v,\mu'\rangle}$$

E-INR
$$\frac{\langle e,\mu\rangle \Downarrow \langle v,\mu'\rangle}{\langle \mathtt{inr}\ e,\mu\rangle \Downarrow \langle \mathtt{inr}\ v,\mu'\rangle}$$

E-LET
$$\frac{\begin{array}{c}\langle e_1,\mu\rangle \Downarrow \langle v_1,\mu_1\rangle \\ \langle e_2,\mu_1 \cup \{x \mapsto v_1\}\rangle \Downarrow \langle v_2,\mu_2\rangle\end{array}}{\langle \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2,\mu\rangle \Downarrow \langle v_2,\mu_2 \setminus \{x\}\rangle}$$

E-CASE-INL
$$\frac{\langle e,\mu\rangle \Downarrow \langle \mathtt{inl}\ v,\mu'\rangle \qquad \langle e_1,\mu' \cup \{x \mapsto v\}\rangle \Downarrow \langle v_1,\mu_1\rangle}{\langle \mathtt{case}\ e\ \mathtt{of}\ \mathtt{inl}\ x.\ e_1 \quad \mathtt{inr}\ x.\ e_2,\mu\rangle \Downarrow \langle v_1,\mu_1 \setminus \{x_1\}\rangle}$$

E-CASE-INR
$$\frac{\langle e,\mu\rangle \Downarrow \langle \mathtt{inr}\ v,\mu'\rangle \qquad \langle e_2,\mu' \cup \{x \mapsto v\}\rangle \Downarrow \langle v_2,\mu_2\rangle}{\langle \mathtt{case}\ e\ \mathtt{of}\ \mathtt{inl}\ x.\ e_1 \quad \mathtt{inr}\ x.\ e_2,\mu\rangle \Downarrow \langle v_2,\mu_2 \setminus \{x_2\}\rangle}$$

E-WHILE-TRUE
$$\frac{\mu(x) \neq 0 \qquad \langle e; \mathtt{while}\ x\ \mathtt{do}\ e,\mu\rangle \Downarrow \langle v,\mu'\rangle}{\langle \mathtt{while}\ x\ \mathtt{do}\ e,\mu\rangle \Downarrow \langle v,\mu'\rangle}$$

E-WHILE-FALSE
$$\frac{\mu(x) = 0}{\langle \mathtt{while}\ x\ \mathtt{do}\ e,\mu\rangle \Downarrow \langle (),\mu\rangle}$$

E-UPGRADE
$$\frac{\langle e,\mu\rangle \Downarrow \langle v,\mu'\rangle}{\langle \bullet e,\mu\rangle \Downarrow \langle v,\mu'\rangle}$$

(b) Evaluation rules.

Figure 4.1: Program's syntax and semantics.

memory location l contains a volatile integer, then its type becomes $\mathtt{Int}^H$. A typing environment Γ is a mapping from memory locations l and variables x to labeled types τ (Figure 4.2).

$$\begin{array}{ll}
\text{Base types} & b ::= \mathtt{Int} \mid \mathtt{Unit} \\
\text{Types} & \sigma ::= b \mid \tau_1 + \tau_2 \\
\text{Labeled types} & \tau ::= \sigma^{\ell}
\end{array}$$

Figure 4.2: Syntax of labeled types τ.

We assume the typing environment Γ to be well-formed: Γ contains at most one occurrence of each memory location l and variable x. To preserve well-formedness, variables are renamed tacitly before they are added to the typing environment.

Using a flow-sensitive type system (Figure 4.3), we propagate the labels throughout the program. Although designed for different reasons, the flow-sensitive type system of Hunt and Sands [17] represents the backbone of our type system. We preserve their idea of using two typing environments in the judgment, a fixed one as a reference for typing and a mutable one

INT

$$\frac{}{pc \vdash \Gamma\{n : \mathtt{Int}^{L}\}\Gamma}$$

UNIT

$$\frac{}{pc \vdash \Gamma\{() : \mathtt{Unit}^{L}\}\Gamma}$$

VAR

$$\frac{\Gamma(x) = \tau}{pc \vdash \Gamma\{x : \tau\}\Gamma}$$

VARLOC

$$\frac{\Gamma(l) = \tau}{pc \vdash \Gamma\{l : \tau\}\Gamma}$$

EXP

$$\frac{pc \vdash \Gamma\{e_1 : \mathtt{Int}^{\ell}\}\Gamma_1 \qquad pc \vdash \Gamma_1\{e_2 : \mathtt{Int}^{\ell}\}\Gamma_2}{pc \vdash \Gamma\{e_1 \oplus e_2 : \mathtt{Int}^{\ell}\}\Gamma_2}$$

ASSIGN

$$\frac{pc \vdash \Gamma\{e : \sigma^{\ell}\}\Gamma'}{pc \vdash \Gamma\{l := e : \mathtt{Unit}^{L}\}\Gamma'[l \mapsto \sigma^{pc \sqcup \ell}]}$$

INL

$$\frac{pc \vdash \Gamma\{e : \tau_1\}\Gamma'}{pc \vdash \Gamma\{\mathtt{inl}\ e : (\tau_1 + \tau_2)^{L}\}\Gamma'}$$

INR

$$\frac{pc \vdash \Gamma\{e : \tau_2\}\Gamma'}{pc \vdash \Gamma\{\mathtt{inr}\ e : (\tau_1 + \tau_2)^{L}\}\Gamma'}$$

LET

$$\frac{\begin{array}{c} pc \vdash \Gamma\{e : \tau\}\Gamma' \\ pc \vdash \Gamma', x : \tau\{e' : \tau'\}\Gamma'' \end{array}}{pc \vdash \Gamma\{\mathtt{let}\ x = e\ \mathtt{in}\ e' : \tau'\}\Gamma'' \setminus \{x\}}$$

CASE

$$\frac{pc \vdash \Gamma\{e : (\tau_1 + \tau_2)^{\ell}\}\Gamma' \qquad pc \sqcup \ell \vdash \Gamma', x : \tau_i\{e_i : \sigma^{\ell'}\}\Gamma_i \qquad \ell \sqsubseteq \ell'}{pc \vdash \Gamma\{(\mathtt{case}\ e\ \mathtt{of}\ \mathtt{inl}\ x.\ e_1 \quad \mathtt{inr}\ x.\ e_2) : \sigma^{\ell'}\} \bigsqcup \Gamma_i \setminus \{x\}}$$

WHILE

$$\frac{\Gamma(x) = \mathtt{Int}^{\ell} \qquad pc \sqcup \ell \vdash \Gamma\{e : \tau\}\Gamma}{pc \vdash \Gamma\{\mathtt{while}\ x\ \mathtt{do}\ e : \mathtt{Unit}^{L}\}\Gamma}$$

UPGRADE

$$\frac{pc \vdash \Gamma\{e : \sigma^{L}\}\Gamma'}{pc \vdash \Gamma\{\bullet e : \sigma^{H}\}\Gamma'}$$

Figure 4.3: Flow-sensitive typing rules.

returned at the end of typing. The typing judgment $pc \vdash \Gamma\{e : \tau\}\Gamma'$, where $pc \in \{L, H\}$, should be read: "Given typing environment Γ and program context *pc*, expression e types to labeled type τ and returns (a possibly updated) typing environment Γ'".

Both typing environments are needed in order to keep track of the change in information flow. As in previous work, the security context *pc* tracks the control flow dependencies of the program counter. The memory locations have mutable types and this allows their labels to change depending on the *pc* and on the labels of the expressions assigned. For example, in a high context, all low variables and memory locations that could be redefined get upgraded to a high-labeled type. Similarly, in a low context, all high variables assigned a constant or a non-volatile expression get downgraded to a low type, becoming non-volatile as well.

Mutable labels make all (implicit or explicit) flows possible and from this point of view, our type system is more permissive than the usual flow-insensitive type systems for secure information flow [30, 35]. A typical information flow type system disallows any flow from high to low, as it would be considered a leak. However, we remind the reader that, in our setting, *high* and *low* do not necessarily have the security meaning, they are merely used to distinguish between and track information that is always present at runtime (L) and information that may be missing at runtime (H).

In the following, we discuss the most interesting typing rules and provide arguments to support our design choices.

Rule CASE. Rule CASE ensures that the typing context returned at the end of typing is the same in both branches, no matter which branch was taken at runtime. More specifically, if a label upgrade takes place in one branch of the case expression and this upgrade is not observed in the other branch, the type system ensures that the resulting typing environment preserves the label upgrade. The following example illustrates this idea:

Example 8. Consider the context $\Gamma = \{(x, \texttt{Int}^L), (y, \texttt{Int}^L), (z, \texttt{Int}^H)\}$. Typing the program bellow under initial environment Γ:

```
if $x$ then
  $y := 2z$;
else
  $z := z + x$;
```

will return the typing environment $\Gamma' = \{(x, \texttt{Int}^L), (y, \texttt{Int}^H), (z, \texttt{Int}^H)\}$.

Rule WHILE. The requirement to have the same typing environment at the beginning, as well as at the end of typing, might seem a bit restrictive. However, this constraint is needed in order to prove transformation correctness (Theorem 17).

Rule UPGRADE. The intuition for the upgrade operator $\bullet$ will become clear in the following section (see Example 10). For now, we only mention that the operator is needed to ensure that, in a high context, the values in the RHS of an assignment always have a high-labeled type.

Summary. In brief, when designing the information flow type system for label propagation, we impose several requirements: (1) Assigning in a low context a low variable to a high one downgrades the high variable; (2) Assigning in any context a high variable to a low one upgrades the low variable. In particular, upgrading is allowed when branching on low. Additionally, downgrading is not allowed when branching on high.

Type-directed transformation

We use the label propagation in the previous section (Figure 4.3) to translate an expression with a labeled type into an expression with a non-labeled type. In particular, an expression with an H-labeled type translates into an expression with a non-labeled Option type.

An Option type is a polymorphic union type that encapsulates an optional value. It consists of either an empty constructor None (e.g., to denote that function π has assigned no value to the volatile input), or a constructor which encapsulates the original data type T, written Some T (e.g., to denote that, for the volatile input, function π returns a value of type T). Hence for a type T, $\texttt{Option}(T) \triangleq \texttt{Unit} + T$. Additionally, None and Some e are syntactic sugar for inl () and inr e respectively.

Syntax and semantics

The syntax and semantics of a target program are almost the same as for the source program (Figure 4.1), thus we do not present them again in this section. However, we do discuss the small differences between the two syntaxes. (1) The target program shows no occurrences of the upgrade operator •; this can also be deduced from the rules of the type-directed translation, Figure 4.6. (2) The guard of a `while` expression can have side effects, thus the syntax for the `while` expression is `while` e_1 `do` e_2. The two rules E-WHILE-TRUE and E-WHILE-FALSE for evaluating a `while` expression change accordingly, as shown in Figure 4.4.

$$\textsc{E-While-True}\quad \frac{\langle \bar{e}_1, \bar{\mu}\rangle \Downarrow \langle \bar{v}_1, \bar{\mu}_1\rangle \qquad \bar{v}_1 \neq 0 \qquad \langle \bar{e}_2;\texttt{while}\ \bar{e}_1\ \texttt{do}\ \bar{e}_2, \bar{\mu}_1\rangle \Downarrow \langle \bar{v}_2, \bar{\mu}_2\rangle}{\langle \texttt{while}\ \bar{e}_1\ \texttt{do}\ \bar{e}_2, \bar{\mu}\rangle \Downarrow \langle v_2, \bar{\mu}_2\rangle}$$

$$\textsc{E-While-False}\quad \frac{\langle \bar{e}_1, \bar{\mu}\rangle \Downarrow \langle 0, \bar{\mu}_1\rangle}{\langle \texttt{while}\ \bar{e}_1\ \texttt{do}\ \bar{e}_2, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}_1\rangle}$$

Figure 4.4: Semantics of `while` expression in the target program.

Type syntax

The types of the target language have no labels, thus they exhibit a slightly different syntax than that of the labeled types (Figure 4.5).

$$\begin{array}{rcl} b & ::= & \texttt{Int} \mid \texttt{Unit} \\ \bar{\tau} & ::= & b \mid \bar{\tau}_1 + \bar{\tau}_2 \end{array}$$

Figure 4.5: Non-labeled type syntax.

Target type system

Since label-free types are used for typing a target program, the type system for verifying its soundness has several differences from the one for typing a source program in Figure 4.3. The typing judgment we use is $\vDash \overline{\Gamma}\{e : \bar{\tau}\}\overline{\Gamma}'$ and should be read as: "Target program e, under label-free typing environment $\overline{\Gamma}$, types to label-free type $\bar{\tau}$ and returns (a possibly updated) label-free typing environment $\overline{\Gamma}'$".

One difference already noticeable from the typing judgment is the absence of program counter *pc*. In the type system for the target program, there are no labels, so tracking the *pc* is irrelevant.

Another notable difference is visible in rule N-CASE, where we require the contexts after typing of expressions e_i $(i = 1,2)$ to be the same:

$$\frac{\vDash \overline{\Gamma}\{e : \bar{\tau}_1 + \bar{\tau}_2\}\overline{\Gamma}' \qquad \vDash \overline{\Gamma}', x : \bar{\tau}_i\{e_i : \bar{\tau}\}\overline{\Gamma}'', x : \bar{\tau}_i \qquad i = 1,2}{\vDash \overline{\Gamma}\{\texttt{case}\ e\ \texttt{of inl}\ x.\ e_1 \quad \texttt{inr}\ x.\ e_2 : \bar{\tau}\}\overline{\Gamma}''}\ \textsc{N-Case}$$

Since in the target language, the `while` expression has a slightly different semantics than in the source language, its type rule in the target type system will also be different than the corresponding rule in the source type system:

$$\frac{\vDash \overline{\Gamma}\{e_1 : \texttt{Int}\}\overline{\Gamma} \qquad \vDash \overline{\Gamma}\{e_2 : \bar{\tau}\}\overline{\Gamma}}{\vDash \overline{\Gamma}\{\texttt{while}\ e_1\ \texttt{do}\ e_2 : \texttt{Unit}\}\overline{\Gamma}}\ \textsc{N-While}$$

The other rules are analogous and we do not write them here. The complete target type system is shown in Appendix B.1.

Type transformation

We use operator $[\![\cdot]\!]$ to obtain from a labeled type τ a non-labeled type $\bar{\tau}$. The high-labeled types drop their label and transform to Option types, while the low-labeled types are simply stripped from their label:

$$[\![\sigma]\!] \triangleq \begin{cases} b & \text{if } \sigma = b \\ [\![\tau_1]\!] + [\![\tau_2]\!] & \text{if } \sigma = \tau_1 + \tau_2. \end{cases} \qquad [\![\tau]\!] \triangleq \begin{cases} [\![\sigma]\!] & \text{if } \tau = \sigma^L \\ \texttt{Option}([\![\sigma]\!]) & \text{if } \tau = \sigma^H. \end{cases}$$

Typing environment transformation

We extend the transformation operator $[\![\cdot]\!]$ to typing environments. Thus, for a context Γ, $[\![\Gamma]\!]$ represents the typing environment with the same domain as Γ and where all types have been de-labeled according to the rules above. More formally,

$$[\![\Gamma]\!] \triangleq \{(l, [\![\tau]\!]) \mid (l, \tau) \in \Gamma\}.$$

Memory transformation

Since the (high-labeled) types of the volatile memory locations translate to an Option type, this modification needs to be reflected in the translated memory as well. We extend the operator $[\![\cdot]\!]$ to memories and annotate it with a context Γ:

$$[\![\mu]\!]_\Gamma \triangleq \{(l, [\![v]\!]_\tau) \mid (l, v) \in \mu \wedge (l, \tau) \in \Gamma\} \cup \{(l, \texttt{None}) \mid (l, \bot) \in \mu\}.$$

We assume memory μ to be well-typed with typing environment Γ, meaning that each value v in μ is well typed under Γ and Γ has the same domain as μ, i.e. $dom(\Gamma) = dom(\mu)$.

The memory transformation says that if no value is provided for a volatile memory location l ($\bot$ was 'assigned' to location l), then None is assigned instead to that memory location. Otherwise, assuming value v of type τ is provided, then the location in the transformed memory will contain the transformed value $[\![v]\!]_\tau$. Thus, $[\![\mu]\!]_\Gamma$ is a total function from memory locations l to transformed values $[\![v]\!]_\tau$.

Value transformation

Similarly to memory transformation, the value transformation is annotated with a type and follows the rules below:

$$\begin{aligned} [\![v]\!]_{b^L} &\triangleq v & \qquad [\![\texttt{inl } v]\!]_{(\tau_1+\tau_2)^L} &\triangleq \texttt{inl } [\![v]\!]_{\tau_1} \\ [\![v]\!]_{\sigma^H} &\triangleq \texttt{Some } [\![v]\!]_{\sigma^L} & \qquad [\![\texttt{inr } v]\!]_{(\tau_1+\tau_2)^L} &\triangleq \texttt{inr } [\![v]\!]_{\tau_2} \end{aligned}$$

It can be easily shown that the type of a transformed value corresponds to the type obtained by applying the de-labeling operator $[\![\cdot]\!]$ to the initial type of the value.

Lemma 13 (Type preservation of value transformation). *If* $pc \vdash \cdot\{v : \tau\}\cdot$, *then* $\vDash \cdot\{[\![v]\!]_\tau : [\![\tau]\!]\}\cdot$.

Proof. By induction on the structure of v. The proof is trivial and we do not include it here. ■

Type-directed transformation

The type-directed transformation takes an expression with a labeled type and returns an expression with a non-labeled type. The rules for the translation are presented in Figure 4.6. The judgment $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$ says that "Given typing environment Γ and program context pc, source program e types to τ, translates to target program $\bar{e}$, and returns a (possibly updated) typing environment Γ'".

In the following paragraphs we discuss some of the rules in the transformation we consider need more clarification.

Transformation of `case`**-expression.** The transformation of `case` expression corresponds to two rules in the type-directed translation, as the label of the conditional can be either L or H, depending on whether there was a flow from a volatile input to the conditional or not.

If the label is L (rule T-CASE-L), then there was no such flow and this means that the conditional will always evaluate to an 'existing' value (or 'some' value). The transformation proceeds as follows: we recursively transform the sub-expressions e_1 and e_2 into $\bar{e}_1$ and $\bar{e}_2$, and, in addition, we upgrade in one branch all the memory locations updated solely in the other branch:

$$\texttt{case}\ \bar{e}\ \texttt{of}\ \texttt{inl}\ x.\ \texttt{let}\ y = \bar{e}_1\ \texttt{in}\ \mathsf{upgrade}(S_2); y \quad \texttt{inr}\ x.\ \texttt{let}\ y = \bar{e}_2\ \texttt{in}\ \mathsf{upgrade}(S_1); y$$

$\bar{e}$ is the translation of the conditional and S_1 and S_2 are sets of memory locations statically determined. S_1 (respectively S_2) contains all the memory locations (to be) updated in e_1 (respectively e_2), but not in e_2 (respectively e_1). Function $\mathsf{upgrade}(S)$ alters the memory μ by assigning an optional value to each memory location in S, i.e. for all $l \in S$, $\mu(l) := \texttt{Some}\ \mu(l)$.

Operator $\setminus$ defines the *high complement* of a typing environment Γ_2 with respect to another typing environment Γ_1. Assuming Γ_1 and Γ_2 have the same domain, $\Gamma_1 \setminus \Gamma_2$ returns the memory locations which have a high-labeled type in Γ_1, but a low-labeled type in Γ_2. More formally,

$$\Gamma_1 \setminus \Gamma_2 \triangleq \{l \mid (l, \sigma^H) \in \Gamma_1 \wedge (l, \sigma^L) \in \Gamma_2\}.$$

If the label of the conditional is high (rule T-CASE-H), then there was an information flow from a volatile input to the conditional. If this input is missing at runtime, the conditional evaluates to `None` and the target program assigns `None` to all the memory locations modified in *any* branch. If the conditional does not evaluate to `None`, meaning that some values were given for the volatile inputs influencing the conditional, then the transformation proceeds as for rule T-CASE-L:

$$\begin{aligned}
&\texttt{case}\ \bar{e}\ \texttt{of}\ \texttt{None}.\ (\forall l \in \mathsf{updated}(\bar{e}_1, \bar{e}_2).\ l := \texttt{None}); \texttt{None}\\
&\qquad\qquad \texttt{Some}\ x'.\ \texttt{case}\ x'\ \texttt{of}\ \texttt{inl}\ x.\ \texttt{let}\ y = \bar{e}_1\ \texttt{in}\ \mathsf{upgrade}(S_2); y\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{inr}\ x.\ \texttt{let}\ y = \bar{e}_2\ \texttt{in}\ \mathsf{upgrade}(S_1); y
\end{aligned}$$

Function $\mathsf{updated}(e)$ is a syntactic function that takes as argument a list of expressions and returns an over-approximated list of memory locations that may be modified in e.

For the case when the value of the conditional is `None`, as at least one of the volatile inputs which tainted the conditional was missing, we make the following comment: the conditional does not evaluate to an existing value, so how should we know which of the two branches to

$$\textsc{T-Val}\quad \frac{}{pc \vdash \Gamma\{v : \tau \rightsquigarrow v\}\Gamma} \qquad \textsc{T-Var}\quad \frac{\Gamma(x) = \tau}{pc \vdash \Gamma\{x : \tau \rightsquigarrow x\}\Gamma} \qquad \textsc{T-VarLoc}\quad \frac{\Gamma(l) = \tau}{pc \vdash \Gamma\{l : \tau \rightsquigarrow l\}\Gamma}$$

$$\textsc{T-Exp-H}\quad \frac{pc \vdash \Gamma\{e_1 : \texttt{Int}^H \rightsquigarrow \bar{e}_1\}\Gamma_1 \qquad pc \vdash \Gamma_1\{e_2 : \texttt{Int}^H \rightsquigarrow \bar{e}_2\}\Gamma_2}{\begin{array}{l} pc \vdash \Gamma\{e_1 \oplus e_2 : \texttt{Int}^H \rightsquigarrow \\ \qquad \texttt{let}\ x = \bar{e}_1\ \texttt{in let}\ y = \bar{e}_2\ \texttt{in} \\ \qquad\quad \texttt{case}\ x\ \texttt{of None. None}\quad \texttt{Some}\ x'.\ (\texttt{case}\ y\ \texttt{of None. None}\quad \texttt{Some}\ y'.\ \texttt{Some}\ (x' \oplus y'))\}\Gamma_2 \end{array}}$$

$$\textsc{T-Exp-L}\quad \frac{pc \vdash \Gamma\{e_1 : \texttt{Int}^L \rightsquigarrow \bar{e}_1\}\Gamma_1 \qquad pc \vdash \Gamma_1\{e_2 : \texttt{Int}^L \rightsquigarrow \bar{e}_2\}\Gamma_2}{pc \vdash \Gamma\{e_1 \oplus e_2 : \texttt{Int}^L \rightsquigarrow \bar{e}_1 \oplus \bar{e}_2\}\Gamma_2}$$

$$\textsc{T-Assign}\quad \frac{pc \vdash \Gamma\{e : \sigma^\ell \rightsquigarrow \bar{e}\}\Gamma' \qquad pc \sqsubseteq \ell}{pc \vdash \Gamma\{l := e : \texttt{Unit}^L \rightsquigarrow l := \bar{e}\}\Gamma'[l \mapsto \sigma^\ell]} \qquad \textsc{T-Inl}\quad \frac{pc \vdash \Gamma\{e : \tau_1 \rightsquigarrow \bar{e}\}\Gamma'}{pc \vdash \Gamma\{\texttt{inl}\ e : (\tau_1 + \tau_2)^L \rightsquigarrow \texttt{inl}\ \bar{e}\}\Gamma'}$$

$$\textsc{T-Inr}\quad \frac{pc \vdash \Gamma\{e : \tau_2 \rightsquigarrow \bar{e}\}\Gamma'}{pc \vdash \Gamma\{\texttt{inr}\ e : (\tau_1 + \tau_2)^L \rightsquigarrow \texttt{inr}\ \bar{e}\}\Gamma'} \qquad \textsc{T-Let}\quad \frac{pc \vdash \Gamma\{e_1 : \tau_1 \rightsquigarrow \bar{e}_1\}\Gamma_1 \qquad pc \vdash \Gamma_1, x : \tau_1\{e_2 : \tau_2 \rightsquigarrow \bar{e}_2\}\Gamma_2}{pc \vdash \Gamma\{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow \texttt{let}\ x = \bar{e}_1\ \texttt{in}\ \bar{e}_2\}\Gamma_2 \setminus \{x\}}$$

$$\textsc{T-Case-H}\quad \frac{pc \vdash \Gamma\{e : (\tau_1 + \tau_2)^H \rightsquigarrow \bar{e}\}\Gamma' \qquad H \vdash \Gamma', x : \tau_i\{e_i : \tau \rightsquigarrow \bar{e}_i\}\Gamma_i \qquad i = 1,2 \qquad S_1 = \Gamma_1 \setminus \Gamma_2 \qquad S_2 = \Gamma_2 \setminus \Gamma_1}{\begin{array}{l} pc \vdash \Gamma\{\texttt{case}\ e\ \texttt{of inl}\ x.\ e_1\quad \texttt{inr}\ x.\ e_2 : \tau \rightsquigarrow \\ \qquad \texttt{case}\ \bar{e}\ \texttt{of None.}\ (\forall l \in \mathsf{updated}(\bar{e}_1, \bar{e}_2).\ l := \texttt{None});\ \texttt{None} \\ \qquad\qquad \texttt{Some}\ x'.\ \texttt{case}\ x'\ \texttt{of inl}\ x.\ \texttt{let}\ y = \bar{e}_1\ \texttt{in}\ \mathsf{upgrade}(S_2); y \\ \qquad\qquad\qquad \texttt{inr}\ x.\ \texttt{let}\ y = \bar{e}_2\ \texttt{in}\ \mathsf{upgrade}(S_1); y\} \bigsqcup \Gamma_i \setminus \{x\} \end{array}}$$

$$\textsc{T-Case-L}\quad \frac{pc \vdash \Gamma\{e : (\tau_1 + \tau_2)^L \rightsquigarrow \bar{e}\}\Gamma' \qquad pc \vdash \Gamma', x : \tau_i\{e_i : \tau \rightsquigarrow \bar{e}_i\}\Gamma_i \qquad i = 1,2 \qquad S_1 = \Gamma_1 \setminus \Gamma_2 \qquad S_2 = \Gamma_2 \setminus \Gamma_1}{\begin{array}{l} pc \vdash \Gamma\{\texttt{case}\ e\ \texttt{of inl}\ x.\ e_1\quad \texttt{inr}\ x.\ e_2 : \tau \rightsquigarrow \\ \qquad \texttt{case}\ \bar{e}\ \texttt{of inl}\ x.\ \texttt{let}\ y = \bar{e}_1\ \texttt{in}\ \mathsf{upgrade}(S_2); y\quad \texttt{inr}\ x.\ \texttt{let}\ y = \bar{e}_2\ \texttt{in}\ \mathsf{upgrade}(S_1); y \bigsqcup \Gamma_i \setminus \{x\} \end{array}}$$

$$\textsc{T-While-H}\quad \frac{\Gamma(x) = \texttt{Int}^H \qquad H \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma}{pc \vdash \Gamma\{\texttt{while}\ x\ \texttt{do}\ e : \texttt{Unit}^L \rightsquigarrow \texttt{while}\ (\texttt{case}\ x\ \texttt{of None.}\ (\forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None};)\ 0\quad \texttt{Some}\ y.\ y)\ \texttt{do}\ \bar{e}\}\Gamma}$$

$$\textsc{T-While-L}\quad \frac{\Gamma(x) = \texttt{Int}^L \qquad pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma}{pc \vdash \Gamma\{\texttt{while}\ x\ \texttt{do}\ e : \texttt{Unit}^L \rightsquigarrow \texttt{while}\ x\ \texttt{do}\ \bar{e}\}\Gamma} \qquad \textsc{T-Upgrade}\quad \frac{pc \vdash \Gamma\{e : \sigma^L \rightsquigarrow \bar{e}\}\Gamma'}{pc \vdash \Gamma\{\bullet e : \sigma^H \rightsquigarrow \texttt{Some}\ \bar{e}\}\Gamma'}$$

Figure 4.6: Type-directed transformation.

execute? We decide to not execute any of them and to assign instead None to all the memory locations and variables further influenced by this conditional. The value None we return after the set of assignments is to preserve the Option type of the target program.

Transformation of `while`-expression. The execution of a `while` expression also depends on a branch condition, hence, for its transformation we introduce two rules as well (T-WHILE-L and T-WHILE-H), one for each of the possible labels of the conditional. If the label is L, the transformation proceeds as expected (rule T-WHILE-L), with a recursive transformation of the sub-expression e. However, things are less obvious if the label of the conditional is H.

Naïve approach. If the label is H, then the conditional must have been influenced by a volatile input and the transformed condition must have an Option type. A naïve transformation of the loop with high guard would be to follow the model of the rule T-CASE-H — first case analyze the condition, and second execute the loop body, as depicted in the following flawed rule:

T-WHILE-H-WRONG

$$\frac{\Gamma(x) = \texttt{Int}^{H} \qquad H \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma}{\begin{array}{l} pc \vdash \Gamma\{\texttt{while}\ x\ \texttt{do}\ e : \texttt{Unit}^{L} \rightsquigarrow \\ \qquad \texttt{case}\ x\ \texttt{of}\ \texttt{None}.\ \forall l \in \text{updated}(\bar{e}).\ l := \texttt{None}; \quad \texttt{Some}\ y.\ \texttt{while}\ y\ \texttt{do}\ \bar{e}\}\Gamma \end{array}}$$

The rule is obviously broken, as an upgrade of the conditional in the loop body in a high context might lead to problems when re-evaluating the loop, especially if the high context is influenced by an input for which no value was provided. To be more specific, consider the following example:

Example 9. Assume the initial typing environment is $\Gamma = \{(x, \texttt{Int}^{H}), (y, \texttt{Int}^{H})\}$ and the initial memory is defined such that $x \neq 0$ and y evaluates to None. Further, assume the source program is the one bellow:

```
while x do
  if y then
    x := •0;
  else
    ...
```

If the target program is obtained following the transformation suggested by rule T-WHILE-H-WRONG, then trying to evaluate the loop guard the second time will not be possible. After the first iteration, x will be assigned None, as its value is influenced by the volatile y.

The type system should take into account any such changes, and, hence, the correct rule for the transformation of the loop with high guard should be:

T-WHILE-H

$$\frac{\Gamma(x) = \texttt{Int}^{H} \qquad H \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma}{\begin{array}{l} pc \vdash \Gamma\{\texttt{while}\ x\ \texttt{do}\ e : \texttt{Unit}^{L} \rightsquigarrow \\ \qquad \texttt{while}\ (\texttt{case}\ x\ \texttt{of}\ \texttt{None}.\ (\forall l \in \text{updated}(\bar{e}).\ l := \texttt{None};)\ 0 \quad \texttt{Some}\ y.\ y)\ \bar{e}\}\Gamma \end{array}}$$

It is easy to notice that the problem raised by the previous example is now fixed and the target program successfully type-checks.

Similar to rule T-CASE-H, in rule T-WHILE-H, we assign None to all the memory locations and variables redefined in the loop when the conditional evaluates to None. However, instead of returning None after the set of assignments, we return 0 (zero). The intuition behind this is that if the guard evaluates to None, then the loop should behave as if the guard evaluated to 0 (with the additional 'erasure' of the redefined memory locations).

Upgrade operator

Now that the type system for the transformation was presented and the most unintuitive rules were explained, we can give an intuition for the upgrade operator $\bullet$ by means of an example.

Example 10. Consider the following two programs and assume $\Gamma(x) = \mathtt{Int}^H$. Additionally, $pc = H$ in the second program:

(a) $\mathtt{if}\ x\ \mathtt{then}\ 1\ \mathtt{else}\ 0$ (b) $x := 2$

Since values are typed low (rules INT and UNIT), the value returned by the if expression (in the first example) and the value assigned to x (in the second example) will also be labeled low. However, since the programs are executed in a high context, the translated version of program (a) should return Some 1 or Some 0 if x is not None. Similarly, in example (b), as x will be given an Option type in the transformed memory, its value after the assignment should also reflect the new type.

We introduce the upgrade operator $\bullet$ and leave to the programmer the task to insert it in all the right places. To prevent her from skipping any required annotation, we add rule UPGRADE to the type system (Figure 4.3) and, as a consequence, the type-checking will not succeed unless the operator was inserted in all expected places. Hence, the 'corrected' and type-sound source programs from the previous example will be (a) $\mathtt{if}\ x\ \mathtt{then}\ \bullet 1\ \mathtt{else}\ \bullet 0$ and (b) $x := \bullet 2$.

Transformation type-soundness and correctness

In this section we prove that our transformation is type-sound and semantically correct. In addition, we show that the transformation is also monotone.

Transformation type-soundness

Type-soundness means that if the source program is typed, then the target program is also typed. More specifically, if the source program e types at τ in Γ and returns typing environment Γ', then its corresponding target program $\bar{e}$ types at de-labeled type $[\![\tau]\!]$ in $[\![\Gamma]\!]$ and returns typing environment $[\![\Gamma']\!]$. The following theorem formalizes this statement.

Theorem 14 (Type-soundness)**.** *For any source program e, typing environment Γ, and program context pc, if $pc \vdash \Gamma\{e : \tau\}\Gamma'$ then there exists target program $\bar{e}$ such that $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$ and $\vDash [\![\Gamma]\!]\,\{\bar{e} : [\![\tau]\!]\}\,[\![\Gamma']\!]$.*

Proof. By induction on the structure of type-directed transformation. See Appendix B.2, Theorem 31 for details. ∎

Transformation correctness

Informally, the transformation is correct if the value v to which the source program evaluates to given complete memory μ is equivalent to the value returned by the target program when evaluated under the transformed memory $[\![\mu]\!]_\Gamma$. We represent the equivalence relation between values in Figure 4.7, while in Definition 15 we extend it to stores.

$$\frac{}{n \sim_{\texttt{Int}^L} n} \qquad \frac{}{n \sim_{\texttt{Int}^H} \texttt{Some}\ n} \qquad \frac{}{() \sim_{\texttt{Unit}^L} ()} \qquad \frac{v \sim_{\tau_1} \bar{v}}{\texttt{inl}\ v \sim_{(\tau_1+\tau_2)^L} \texttt{inl}\ \bar{v}}$$

$$\frac{v \sim_{\tau_1} \bar{v}}{\texttt{inl}\ v \sim_{(\tau_1+\tau_2)^H} \texttt{Some inl}\ \bar{v}} \qquad \frac{v \sim_{\tau_2} \bar{v}}{\texttt{inr}\ v \sim_{(\tau_1+\tau_2)^L} \texttt{inr}\ \bar{v}} \qquad \frac{v \sim_{\tau_2} \bar{v}}{\texttt{inr}\ v \sim_{(\tau_1+\tau_2)^H} \texttt{Some inr}\ \bar{v}}$$

Figure 4.7: Value type-equivalence.

Definition 15 (Store equivalence)**.** Two stores μ and $\bar{\mu}$ are equivalent with respect to a typing environment Γ, written $\mu \sim_\Gamma \bar{\mu}$, iff $dom(\mu) = dom(\bar{\mu}) = dom(\Gamma)$ and for all memory locations l (and variables x), $\mu(l) \sim_{\Gamma(l)} \bar{\mu}(l)$.

Lemma 16 (Helper)**.** *For any complete memory μ and typing context Γ, $\mu \sim_\Gamma [\![\mu]\!]_\Gamma$.*

Proof. By induction on the size of μ and from Lemma 13. The proof is trivial and we do not include it here. ■

Theorem 17 (Correctness, equivalence between source and target programs)**.** *For any source program e, typing environment Γ, program context pc, and store μ, if $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$ and $\langle e, \mu\rangle \Downarrow \langle v_f, \mu_f\rangle$, then there exists $\bar{v}_f$ and $\bar{\mu}_f$, such that $\langle \bar{e}, [\![\mu]\!]_\Gamma\rangle \Downarrow \langle \bar{v}_f, \bar{\mu}_f\rangle$ and $v_f \sim_\tau \bar{v}_f$ and $\mu_f \sim_{\Gamma'} \bar{\mu}_f$.*

Proof. From Lemma 16 and by induction on the derivation of the evaluation relation. See Appendix B.3, Theorem 34 for details. ■

Transformation monotonicity

We use the value equivalence relation in Figure 4.7 to specify when a value v is 'more precise' than a target value $\bar{v}$ (Figure 4.8). And, similarly to value equivalence, we extend the relation to memories in Definition 18.

$$\frac{v \sim_\tau \bar{v}}{v \geq_\tau \bar{v}} \qquad\qquad \frac{}{v \geq_{\sigma^H} \texttt{None}}$$

Figure 4.8: v is more precise than $\bar{v}$.

Definition 18 (More precise store.)**.** A store μ is *more precise* than a store $\bar{\mu}$ with respect to a typing environment Γ, written $\mu \geq_\Gamma \bar{\mu}$, iff $dom(\mu) = dom(\bar{\mu}) = dom(\Gamma)$ and, for all memory locations l, $\mu(l) \geq_{\Gamma(l)} \bar{\mu}(l)$.

It is trivial to show now that the value to which a source program evaluates to is more precise than the value to which its corresponding target program evaluates to.

Theorem 19 (Source program more precise)**.** *For any source program e, typing environment* Γ*, program context pc, and stores* μ *and* $\bar{\mu}$ *such that* $\mu \geq_\Gamma \bar{\mu}$*, if* $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$ *and* $\langle e, \mu \rangle \Downarrow \langle v_f, \mu_f \rangle$*, then there exists* $\bar{v}_f$ *and* $\bar{\mu}_f$*, such that* $\langle \bar{e}, \bar{\mu} \rangle \Downarrow \langle \bar{v}_f, \bar{\mu}_f \rangle$ *and* $v_f \geq_\tau \bar{v}_f$ *and* $\mu_f \geq_{\Gamma'} \bar{\mu}_f$*.*

Proof. By induction on the structure of the type-directed transformation. See Appendix B.3, Theorem 35 for details. ■

Similarly to the 'more precise than' relation between values in Figure 4.8, we define a 'more precise than' relation between target values in Figure 4.9, and we extend it to stores in Definition 20.

$$\frac{}{\bar{v} \geq \bar{v}} \qquad \frac{}{\texttt{Some } \bar{v} \geq \texttt{None}} \qquad \frac{\bar{v} \geq \bar{v}'}{\texttt{inl } \bar{v} \geq \texttt{inl } \bar{v}'} \qquad \frac{\bar{v} \geq \bar{v}'}{\texttt{inr } \bar{v} \geq \texttt{inl } \bar{v}'}$$

Figure 4.9: Target value $\bar{v}$ is *more precise than* target value $\bar{v}'$.

Definition 20 (More precise target store.)**.** A target store $\bar{\mu}$ is *more precise than* a target store $\bar{\mu}'$ iff $dom(\bar{\mu}) = dom(\bar{\mu}')$ and, for all memory locations l, $\bar{\mu}(l) \geq \bar{\mu}'(l)$.

Finally, we prove that the transformed program is monotone, i.e. executing the target program with a more precise store (with fewer `None`s) yields more precise outputs. More specifically, we prove that evaluating the target program under memories $\bar{\mu}$ and, respectively, $\bar{\mu}'$, satisfying the property that $\bar{\mu}$ is more precise than $\bar{\mu}'$ ($\bar{\mu} \geq \bar{\mu}'$), returns values $\bar{v}_f$ and memory $\bar{\mu}_f$, respectively, $\bar{v}'_f$ and memory $\bar{\mu}'_f$ satisfying the properties that $\bar{v}_f \geq \bar{v}'_f$ and $\bar{\mu}_f \geq \bar{\mu}'_f$.

Theorem 21 (Monotonicity of $\bar{e}$)**.** *For any source program e, typing context* Γ*, program counter pc, and memories* $\bar{\mu}$*, and* $\bar{\mu}'$ *such that* $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$*, and* $\bar{\mu} \geq \bar{\mu}'$*, if* $\langle \bar{e}, \bar{\mu} \rangle \Downarrow \langle \bar{v}_f, \bar{\mu}_f \rangle$ *and* $\langle \bar{e}, \bar{\mu}' \rangle \Downarrow \langle \bar{v}'_f, \bar{\mu}'_f \rangle$*, then* $\bar{v}_f \geq \bar{v}'_f$ *and* $\bar{\mu}_f \geq \bar{\mu}'_f$*.*

Proof. By induction on the derivation of the type-directed transformation and on the derivation of the evaluation relation. See Appendix B.3, Theorem 37 for details. ■

Related work

Program transformation. Hunt and Sands [17] design a flow-sensitive type system for program transformation that constructs a target program equivalent with the source program which can be typed using a simple flow-insensitive type system. Our type system in Figure 4.3 is inspired by the one in Hunt and Sands' work. However, the transformations and their purposes are quite different: the approach of Hunt and Sands is relevant in a proof-carrying-code setting [17], while our approach aims to make a program robust to missing data.

A program transformation method was earlier suggested by Lam and Chiueh [20], although not by means of a type system, but by using a compiler-based approach. In the context of dynamic taint analyses, Lam and Chiueh propose the GIFT (General dynamic Information Flow Tracking) compiler to automatically add code to propagate and check tags associated with data. Their compiler performs an automated program transformation on C code: it takes C programs and programmer-specified rules for tag manipulation and automatically inserts wrapper functions in order to ensure that the rules are enforced by the program execution. GIFT uses dynamic data and control flow analysis and automates the process of implementing information flow tracking into individual applications. Similarly to our approach where, depending on which inputs are labeled high, different program transformations are possible, in GIFT, the same program can be instrumented differently when used in different applications or systems, in a way that it matches their corresponding security requirements.

Label propagation. Several previous approaches perform label propagation based on information flow type systems [24, 25, 38, 39], while other methods suggest using labeling semantics [4, 15]. Costanzo and Shao [15] use a security-aware operational semantics for propagating labels. Similarly to our approach, they allow for label overwriting, but, in their case, the label is attached to the value, and not to the memory location. Their functions $\mathsf{mark\_vars}(\sigma, C)$ and $\mathsf{modifies}(C)$ have a similar effect as the functions $\mathsf{upgrade}(S)$ and $\mathsf{updated}(e)$ we define. While their approach is used for enforcing declassification policies, we believe that a similar operational semantics can also be adapted to make a program robust to missing data.

Conclusion

In this thesis, we focused on improving secure multi-execution (SME) and its extension with declassification, by presenting Asymmetric SME (A-SME), a modification of SME that gives up on the pretense that real programs are inherently robust to modified inputs. A-SME executes a program and its low slice simultaneously to enforce a broad range of declassification policies. We prove that A-SME is secure, independent of the correctness of the low slice, and also precise if the low slice is semantically correct. Moreover, we show that A-SME does not result in loss of expressiveness: If the original program conforms to the declassification policy, then a correct low slice exists. Additionally, we improve the expressive power of declassification policies considered in literature by allowing feedback from the program, and by allowing input presence sensitivity to depend on policy state. Finally, assuming the program can be typed with an information flow type system, we describe an automatic technique based on an information flow type system for obtaining the low slice in the case when policy $\mathscr{D}$ blocks some inputs at runtime. We prove that the resulting low slice is correct and preserves type correctness, and that the transformation is monotone.

Future work

A-SME can be generalized to arbitrary security lattices. For each lattice level ℓ, a separate projection function π_ℓ could determine the values declassified to the ℓ-run in A-SME. For $\ell \sqsubseteq \ell'$, π_ℓ should reveal less information than $\pi_{\ell'}$, i.e. there should exist some function f such that $\pi_\ell = f \circ \pi_{\ell'}$. Additionally, A-SME would require a different slice of the program for every level ℓ.

Another interesting direction for future work would be to develop an analysis either to verify the correctness of a low slice, or to automatically construct the low slice from a program and a policy (in cases more general than those of Chapter 4). Verification will involve establishing semantic similarity of the composition of the low slice and the policy with a part of the program, which can be accomplished using static methods for relational verification. Automatic construction of the low slice should be feasible using program slicing techniques, at least in some cases.

Viewed from a different perspective, the program transformation can be seen as rewriting a computation with no side effects into a computation that produces side effects. Transforming the high types to Option types encapsulates the use of Option monad. For the Option monad the side effect is represented by the program failing to produce a value of type T and returning instead value `None`. Different monads express different side effects, such as reading or writing memory, or throwing an exception. As future work, it would be interesting to explore whether the transformation could be generalized to arbitrary monads, by rewriting the target type system to use only the monadic operations *bind* and *return*.

Proofs for the results of A-SME

Security of A-SME

Theorem 22 (Security, noninterference under $\mathscr{D}$)**.** *Suppose $I_1,\mu_1 \longrightarrow_p E_1$ and $I_2,\mu_2 \longrightarrow_p E_2$ and $\mathscr{D}^*(s_1,E_1) = \mathscr{D}^*(s_2,E_2)$. If $I_1,s_1,\mu_1,\mu_L \Longmapsto^{\mathscr{D}}_{p,p_L} E'_1$ and $I_2,s_2,\mu_2,\mu_L \Longmapsto^{\mathscr{D}}_{p,p_L} E'_2$, then $E'_1|_o|_L = E'_2|_o|_L$.*

Proof. Let n be the length of $I_1 ++ I_2$. The proof is by induction on n.

Base case: $n = 0$**.** Then $I_1 = I_2 = []$, which implies $E'_1 = E'_2 = []$ and $E'_1|_o|_L = E'_2|_o|_L$.

Inductive step: $n > 0$**.** Then either $I_1 \neq []$ or $I_2 \neq []$. We assume $I_1 \neq []$ (the other case is symmetric). As I_1 is not empty, it must contain at least one element, thus $I_1 = i_1 :: I'_1$.

Let $s''_1 = \mathsf{upd}^i(s_1,i_1)$, $b_1 = \sigma(s''_1)$, $(O_1,\mu'_1) = p(i_1,\mu_1)$, $s'_1 = \mathsf{upd}^o(s''_1,O_1)$, and $E_1 = (i_1,O_1) :: E''_1$, where $I'_1,\mu'_1 \longrightarrow_p E''_1$. We case analyze b_1:

Case: $b_1 = \mathsf{false}$

Then $D^*(s_1,E_1) = \mathscr{D}^*(s'_1,E''_1) = \mathscr{D}^*(s_2,E_2)$. $E'_1 = (i_1,O_1|_H) :: E'''_1$, where $I'_1,s'_1,\mu'_1,\mu_L \Longmapsto^{\mathscr{D}}_{p,p_L} E'''_1$. From the i.h., $E'''_1|_o|_L = E'_2|_o|_L$. Since $O_1|_H|_L = []$, $E'_1|_o|_L = E'_2|_o|_L$.

Case: $b_1 = \mathsf{true}$

Then $\mathscr{D}^*(s_1,E_1) = r_1 ++ \mathscr{D}^*(s'_1,E''_1) = \mathscr{D}^*(s_2,E_2)$, where $r_1 = \pi(s''_1)$. This means that I_2 must be non-empty. Let $I_2 = i_2 :: I'_2$, $s''_2 = \mathsf{upd}^i(s_2,i_2)$, $b_2 = \sigma(s''_2)$, $(O_2,\mu'_2) = p(i_2,\mu_2)$, $s'_2 = \mathsf{upd}^o(s''_2,O_2)$, and $E_2 = (i_2,O_2) :: E''_2$, where $I'_2,\mu'_2 \longrightarrow_p E''_2$.

If $b_2 = \mathsf{false}$, we apply an argument symmetric to the one for $b_1 = \mathsf{false}$. In the following, we discuss the more interesting case, $b_2 = \mathsf{true}$. $\mathscr{D}^*(s_2,E_2) = r_1 :: \mathscr{D}^*(s'_2,E''_2)$, thus $\pi(s''_2) = r_1$. Hence $(O,\mu'_L) = p_L(\pi(s''_1),\mu_L) = p_L(r_1,\mu_L) = p_L(\pi(s''_2),\mu_L)$.

For $j \in \{1,2\}$, $E'_j = (i_j,O|_L :: O_j|_H) :: E'''_j$, where $I'_j,s'_j,\mu'_j,\mu'_L \Longmapsto^{\mathscr{D}}_{p,p_L} E'''_j$. From the i.h., $E'''_1|_o|_L = E'''_2|_o|_L$. Thus $E'_1|_o|_L = E'_2|_o|_L$. ■

Precision of A-SME

Theorem 23 (Precision for high outputs)**.** *For any programs p and p_L, declassification policy $\mathscr{D}$ with initial state s, and input list I, if $I,\mu_H \longrightarrow_p E$ and $I,s,\mu_H,\mu_L \Longmapsto^{\mathscr{D}}_{p,p_L} E'$, then $E|_o|_H = E'|_o|_H$.*

Proof. The proof is by induction on I.

Base case: $I = []$**.** Then $E = E' = []$, hence $E|_o|_H = E'|_o|_H = []$.

Induction step: $I = i :: I'$. Let $s'' = \mathsf{upd}^i(s,i)$, $b = \sigma(s'')$, $(O,\mu'_H) = p(i,\mu_H)$, and $s' = \mathsf{upd}^o(s'',O)$. We case analyze b:

Case: $b = \mathsf{false}$

Then $E = (i,O) :: E''$, where $I',\mu'_H \longrightarrow_p E''$ (from rule R2) and $E' = (i,O|_H) :: E'''$ (from rule A-SME-2), where $I',s',\mu'_H,\mu_L \Longmapsto^{\mathcal{D}}_{p,p_L} E'''$.

From the i.h. applied to I', we get $E''|_o|_H = E'''|_o|_H$. Hence, $E|_o|_H = ((i,O) :: E'')|_o|_H = O|_H ++ E''|_o|_H = O|_H ++ E'''|_o|_H = ((i,O|_H) :: E''')|_o|_H = E'|_o|_H$.

Case: $b = \mathsf{true}$

Let $(O',\mu'_L) = p_L(r,\mu_L)$. Then $E = (i,O) :: E''$, where $I',\mu'_H \longrightarrow_p E''$ (from rule R2) and $E' = (i,O|_H ++ O'|_L) :: E'''$ (from A-SME-3), where $I',s',\mu'_H,\mu'_L \Longmapsto^{\mathcal{D}}_{p,p_L} E'''$.

From the i.h. applied to I', we get $E''|_o|_H = E'''|_o|_H$. Hence, $E|_o|_H = ((i,O) :: E'')|_o|_H = O|_H ++ E''|_o|_H = O|_H ++ E'''|_o|_H = ((i,O|_H ++ O'|_L) :: E''')|_o|_H = E'|_o|_H$. ■

Lemma 24 (Low simulation). *Let $I,s,\mu_H,\mu_L \Longmapsto^{\mathcal{D}}_{p,p_L} E$ and $\mathcal{D}^*(s,E) = R$. If $R,\mu_L \longrightarrow_{p_L} E'$, then $E|_o|_L = E'|_o|_L$.*

Proof. By induction on I.

Base case: $I = []$. Then, $E = R = E' = []$ and $E|_o|_L = [] = E'|_o|_L$.

Induction step: $I = i :: I'$. Let $s'' = \mathsf{upd}^i(s,i)$, $b = \sigma(s'')$, $(O,\mu'_H) = p(i,\mu_H)$ and $s' = \mathsf{upd}^o(s'',O)$. We case analyze b:

Case: $b = \mathsf{false}$

Then, from rule A-SME-2, $E = (i,O|_H) :: E''$, where $I',s',\mu'_H,\mu_L \Longmapsto^{\mathcal{D}}_{p,p_L} E''$. From the definition of $\mathcal{D}^*$, $R = \mathcal{D}^*(s,E) = \mathcal{D}^*(s'',E'')$.

Applying the i.h. to I'/I, s'/s, E''/E, μ'_H/μ_H but the same E', R, and μ_L, we get $E''|_o|_L = E'|_o|_L$. Hence, $E|_o|_L = ((i,O|_H) :: E'')|_o|_L = E''|_o|_L = E'|_o|_L$.

Case: $b = \mathsf{true}$

Let $(O',\mu'_L) = p_L(r,\mu_L)$. Then $E = (i,O|_H ++ O'|_L) :: E''$ (from rule A-SME-3), where $I',s',\mu'_H,\mu'_L \Longmapsto^{\mathcal{D}}_{p,p_L} E''$. From the definition of $\mathcal{D}^*$, $R = \mathcal{D}^*(s,E) = r :: \mathcal{D}^*(s'',E'') = r :: R'$ (where $R' = \mathcal{D}^*(s'',E'')$). From rule R2, $E' = (r,O') :: E'''$, where $R',\mu'_L \longrightarrow_p E'''$.

Applying the i.h. to I'/I, s'/s, E''/E, E'''/E', μ'_H/μ_H, R'/R, and μ'_L/μ_L, we get $E''|_o|_L = E'''|_o|_L$. Hence, $E|_o|_L = ((i,O|_H ++ O'|_L) :: E'')|_o|_L = O'|_L ++ E''|_o|_L = O'|_L ++ E'''|_o|_L = ((r,O') :: E''')|_o|_L = E'|_o|_L$. ■

Theorem 25 (Precision for low outputs). *For any programs p and p_L, declassification policy $\mathcal{D}$ with initial state s and input list I, if $I,\mu_H \longrightarrow_p E$ and $I,s,\mu_H,\mu_L \Longmapsto^{\mathcal{D}}_{p,p_L} E'$ and (μ_L,p_L) is a correct low pair with respect to $\mathcal{D}$, s, p and μ_H, then $E|_o|_L = E'|_o|_L$.*

Proof. Let $R = \mathcal{D}^*(s,E')$ and suppose $R,\mu_L \longrightarrow_{p_L} E''$. By Lemma 24, $E'|_o|_L = E''|_o|_L$. By Definition 5, $E|_o|_L = E''|_o|_L$. Therefore, $E|_o|_L = E''|_o|_L = E'|_o|_L$. ■

Theorem 26 (Precision). *For any programs p and p_L, declassification policy $\mathcal{D}$ with initial state s and input list I, if $I,\mu_H \longrightarrow_p E$ and $I,s,\mu_H,\mu_L \Longmapsto^{\mathcal{D}}_{p,p_L} E'$, and (μ_L,p_L) is a correct low pair with respect to $\mathcal{D}$, s, p and μ_H, then $E|_o|_L = E'|_o|_L$ and $E|_o|_H = E'|_o|_H$.*

Proof. Immediate from Theorems 23 and 25. ■

Existence of correct low slices

Given a program p, we abuse notation and write $p(I,\mu)$ to denote the final memory obtained by executing the input sequence I starting from μ. Formally,

$$\begin{aligned} p([],\mu) &= \mu \\ p(i :: I,\mu) &= \text{let } (\_,\mu') = p(i,\mu) \text{ in } p(I,\mu'). \end{aligned}$$

Similarly, we write $\mathsf{upd}(s,E)$ to denote the final state of the policy obtained by updating s repeatedly with the elements of E. Formally,

$$\begin{aligned} \mathsf{upd}(s,[]) &= s \\ \mathsf{upd}(s,(i,O) :: E) &= \mathsf{upd}(\mathsf{upd}^o(\mathsf{upd}^i(s,i),O),E). \end{aligned}$$

Lemma 27 (Execution factorization). *Let I_1 and I_2 be two input lists such that $I_1 +\!\!+ I_2,\mu \longrightarrow_p E$ and $I_1,\mu \longrightarrow_p E_1$ and $\mu_1 = p(I_1,\mu)$ and $I_2,\mu_1 \longrightarrow_p E_2$. Then $E = E_1 +\!\!+ E_2$.*

Proof. By induction on I_1.

Base case: $I_1 = []$. Here, $E_1 = []$, $I_1 +\!\!+ I_2 = I_2$ and $\mu_1 = p([],\mu) = \mu$. Hence, we know that (a) $I_2,\mu \longrightarrow_p E$ and (b) $I_2,\mu \longrightarrow_p E_2$. Since evaluation is deterministic, $E = E_2 = [] +\!\!+ E_2 = E_1 +\!\!+ E_2$.

Induction step: $I_1 = i_1 :: I_1'$. Let $p(i_1,\mu) = (O_1,\mu')$ and suppose that $I_1' +\!\!+ I_2,\mu' \longrightarrow_p E'$ and $I_1',\mu' \longrightarrow_p E_1'$. Note that $\mu_1 = p(I_1,\mu) = p(i_1 :: I_1',\mu) = p(I_1',\mu')$ by definition of p. Hence, by the i.h. applied to I_1', we know that $E' = E_1' +\!\!+ E_2$. Further, by definition of $\longrightarrow_p$, $E_1 = (i_1,O_1) :: E_1'$ and $E = (i_1,O_1) :: E'$. Hence, $E_1 +\!\!+ E_2 = ((i_1,O_1) :: E_1') +\!\!+ E_2 = (i_1,O_1) :: (E_1' +\!\!+ E_2) = (i_1,O_1) :: E' = E$. ■

Lemma 28 (Update factorization). *Let I_1 and I_2 be two input lists such that $I_1 +\!\!+ I_2,\mu \longrightarrow_p E$ and $I_1,\mu \longrightarrow_p E_1$ and $\mu_1 = p(I_1,\mu)$ and $I_2,\mu_1 \longrightarrow_p E_2$. Then $\mathscr{D}^*(s,E_1 +\!\!+ E_2) = \mathscr{D}^*(s,E_1) +\!\!+ \mathscr{D}^*(\mathsf{upd}(s,E_1),E_2)$.*

Proof. By induction on E_1.

Base case: $E_1 = []$. Then

$$\begin{aligned} &\mathscr{D}^*(s,E_1 +\!\!+ E_2) = \mathscr{D}^*(s,[] +\!\!+ E_2) \\ &\quad = \mathscr{D}^*(s,E_2) \\ &\quad = [] +\!\!+ \mathscr{D}^*(s,E_2) \\ &\quad = \mathscr{D}^*(s,[]) +\!\!+ \mathscr{D}^*(\mathsf{upd}(s,[]),E_2) \\ &\quad = \mathscr{D}^*(s,E_1) +\!\!+ \mathscr{D}^*(\mathsf{upd}(s,E_1),E_2). \end{aligned}$$

Induction step: $E_1 = (i,O) :: E_1'$. Let $s'' = \mathsf{upd}^i(s,i)$, $b = \sigma(s'')$ and $s' = \mathsf{upd}^o(s'',O)$. We case analyze b:

Case: $b = \mathsf{false}$

$$\begin{aligned} &\mathscr{D}^*(s,E_1 +\!\!+ E_2) = \mathscr{D}^*(s,(i,O) :: (E_1' +\!\!+ E_2)) \\ &\quad = \mathscr{D}^*(s',E_1' +\!\!+ E_2) && (b = \mathsf{false}, \text{ definition of } \mathscr{D}^*) \\ &\quad = \mathscr{D}^*(s',E_1') +\!\!+ \mathscr{D}^*(\mathsf{upd}(s'',E_1'),E_2) && \text{(i.h.)} \\ &\quad = \mathscr{D}^*(s,(i,O) :: E_1') +\!\!+ \mathscr{D}^*(\mathsf{upd}(s'',E_1'),E_2) && (b = \mathsf{false}, \text{ definition of } \mathscr{D}^*) \end{aligned}$$

$$\begin{aligned}
&= \mathscr{D}^*(s,E_1) \mathbin{++} \mathscr{D}^*(\mathsf{upd}(s'',E'_1),E_2)\\
&= \mathscr{D}^*(s,E_1) \mathbin{++} \mathscr{D}^*(\mathsf{upd}(\mathsf{upd}^o(s',O),E'_1),E_2)\\
&= \mathscr{D}^*(s,E_1) \mathbin{++} \mathscr{D}^*(\mathsf{upd}(\mathsf{upd}^o(\mathsf{upd}^i(s,i),O),E'_1),E_2)\\
&= \mathscr{D}^*(s,E_1) \mathbin{++} \mathscr{D}^*(\mathsf{upd}(s,(i,O) :: E'_1),E_2) && \text{(Definition of upd)}\\
&= \mathscr{D}^*(s,E_1) \mathbin{++} \mathscr{D}^*(\mathsf{upd}(s,E_1),E_2).
\end{aligned}$$

Case: $b = \mathsf{true}$

$$\begin{aligned}
&\mathscr{D}^*(s,E_1 \mathbin{++} E_2) = \mathscr{D}^*(s,(i,O) :: (E'_1 \mathbin{++} E_2))\\
&\quad = r :: \mathscr{D}^*(s'',E'_1 \mathbin{++} E_2,s'') && (b = \mathsf{true}, \text{definition of } \mathscr{D}^*)\\
&\quad = r :: \mathscr{D}^*(s'',E'_1) \mathbin{++} \mathscr{D}^*(\mathsf{upd}(s'',E'_1),E_2) && \text{(i.h.)}\\
&\quad = \mathscr{D}^*(s,(i,O) :: E'_1) \mathbin{++} \mathscr{D}^*(\mathsf{upd}(s'',E'_1),E_2) && (b = \mathsf{true}, \text{definition of } \mathscr{D}^*)\\
&\quad = \mathscr{D}^*(s,E_1) \mathbin{++} \mathscr{D}^*(\mathsf{upd}(s'',E'_1),E_2)\\
&\quad = \mathscr{D}^*(s,E_1) \mathbin{++} \mathscr{D}^*(\mathsf{upd}(\mathsf{upd}^o(s',O),E'_1),E_2)\\
&\quad = \mathscr{D}^*(s,E_1) \mathbin{++} \mathscr{D}^*(\mathsf{upd}(\mathsf{upd}^o(\mathsf{upd}^i(s,i),O),E'_1),E_2)\\
&\quad = \mathscr{D}^*(s,E_1) \mathbin{++} \mathscr{D}^*(\mathsf{upd}(s,(i,O) :: E'_1),E_2) && \text{(Definition of upd)}\\
&\quad = \mathscr{D}^*(s,E_1) \mathbin{++} \mathscr{D}^*(\mathsf{upd}(s,E_1),E_2). && \blacksquare
\end{aligned}$$

Lemma 29 (Correctness of construction). *(μ_L, p_L) as defined in Section 3.5 is a correct low pair for $\mathscr{D}$, s, p and μ if p, starting from initial memory μ, does not leak outside declassification in $\mathscr{D}$ and initial state s.*

Proof. Let $I,\mu \longrightarrow_p E$ and $\mathscr{D}^*(s,E) = R$ and $R,\mu_L \longrightarrow_{p_L} E'$. We need to prove that $E|_o|_L = E'|_o|_L$. We proceed by induction on I, but in reverse order.

Base case: $I = []$. Then $E = R = E' = []$, hence $E|_o|_L = E'|_o|_L = []$.

Induction step: $I = I_1 :: i$. Let E_1, μ_1, and O_2 be such that $I_1,\mu \longrightarrow_p E_1$ (1), $\mu_1 = p(I_1,\mu)$ (2), and $i,\mu_1 \longrightarrow_p (i,O_2)$ (3).

From Lemma 27 applied to $I,\mu \longrightarrow_p E$ and statements (1)–(3) above, we get that $E = E_1 :: (i,O_2)$ (4).

Let $s_1 = \mathsf{upd}(s,E_1)$ and $R_1 = \mathscr{D}^*(s,E_1)$. From Lemma 28 applied to statement (4), we get $R = \mathscr{D}^*(s,E) = R_1 \mathbin{++} \mathscr{D}^*(s_1,(i,O_2))$ (5).

Let $s'_1 = \mathsf{upd}^i(s_1,i)$ and $b = \sigma(s'_1)$. We case analyze b:

Case: $b = \mathsf{false}$

By definition of $\mathscr{D}^*$, $\mathscr{D}^*(s_1,(i,O_2)) = []$. Thus, from (5) and the definition of R_1, we get $R = \mathscr{D}^*(s,E) = R_1 \mathbin{++} [] = R_1 = \mathscr{D}^*(s,E_1)$.

From the i.h. applied to $I_1/I, E_1/E$ using statement (1) and the assumption $R,\mu_L \longrightarrow_{p_L} E'$, we get $E_1|_o|_L = E'|_o|_L$. Since (μ,p) does not leak outside declassification in $\mathscr{D}$ and initial state s, and $\mathscr{D}^*(s,E) = \mathscr{D}^*(s,E_1)$, we also get $E_1|_o|_L = E|_o|_L$. Hence, $E|_o|_L = E_1|_o|_L = E'|_o|_L$.

Case: $b = \mathsf{true}$

By definition of $\mathscr{D}^*$, $\mathscr{D}^*(s_1,(i,O_2)) = [r]$. Thus, from (5) and the definition of R_1, we get $R = \mathscr{D}^*(s,E) = R_1 \mathbin{++} [r] = R_1 :: r = \mathscr{D}^*(s,E_1) :: r$. We choose E'_1, μ_{L1}, O'_2 such that $R_1,\mu_L \longrightarrow_{p_L} E'_1$ (6), $\mu_{L1} = p_L(R_1,\mu_L)$ (7), and $r,\mu_{L1} \longrightarrow_{p_L} (r,O'_2)$ (8).

Since $R = R_1 :: r$, by Lemma 27 applied to $R,\mu_L \longrightarrow_{p_L} E'$ and (6)–(8), we get that $E' = E'_1 :: (r,O'_2)$ (9).

Applying i.h. to (1), (6), and the definition $R_1 = \mathscr{D}^*(s,E_1)$, we get $E_1|_o|_L = E'_1|_o|_L$. Hence using (4) and (9) it suffices to prove $O_2|_L = O'_2|_L$. For this, we first note that $p_L(R_1,\mu_L) =$

$p_L(R_1,[]) = R_1$ (from definitions of μ_L and p_L). Hence $\mu_{L1} = R_1$ (from (7)) and $r, R_1 \longrightarrow_{p_L} (r, O'_2)$ (from (8)). Expanding the definition of $\longrightarrow_{p_L}$, $(O'_2, \_) = p_L(r, R_1) = (h(R_1 :: r) \setminus h(R_1), R_1 :: r)$.

Thus,$O'_2 = h(R_1 :: r) \setminus h(R_1) = h(R) \setminus h(R_1) = h(\mathscr{D}^*(s,E)) \setminus h(\mathscr{D}^*(s,E_1)) = h(g(I)) \setminus h(g(I_1)) = f(I) \setminus f(I_1)$ (10).

Further, from (4), $O_2|_L = E|_o|_L \setminus E_1|_o|_L$. From the assumption $I, \mu \longrightarrow_p E$, $E|_o|_L = f(I)$ and from (1), $E_1|_o|_L = f(I_1)$. Thus $O_2|_L = f(I) \setminus f(I_1)$. Combining with (10), we get $O'_2 = O_2|_L$. Hence, $O'_2|_L = O_2|_L|_L = O_2|_L$, as needed. ■

Proofs for the program transformation

Type system for target program

$$\overset{\text{N-INT}}{\dfrac{}{\vDash \bar{\Gamma}\{n : \texttt{Int}\}\bar{\Gamma}}} \qquad \overset{\text{N-UNIT}}{\dfrac{}{\vDash \bar{\Gamma}\{() : \texttt{Unit}\}\bar{\Gamma}}} \qquad \overset{\text{N-VAR}}{\dfrac{\bar{\Gamma}(x) = \bar{\tau}}{\vDash \bar{\Gamma}\{x : \bar{\tau}\}\bar{\Gamma}}} \qquad \overset{\text{N-VARLOC}}{\dfrac{\bar{\Gamma}(l) = \bar{\tau}}{\vDash \bar{\Gamma}\{l : \bar{\tau}\}\bar{\Gamma}}}$$

$$\overset{\text{N-EXP}}{\dfrac{\vDash \bar{\Gamma}\{e_1 : \texttt{Int}\}\bar{\Gamma}_1 \qquad \vDash \bar{\Gamma}_1\{e_2 : \texttt{Int}\}\bar{\Gamma}_2}{\vDash \bar{\Gamma}\{e_1 \oplus e_2 : \texttt{Int}\}\bar{\Gamma}_2}} \qquad \overset{\text{N-ASSIGN}}{\dfrac{\vDash \bar{\Gamma}\{e : \bar{\tau}\}\bar{\Gamma}'}{\vDash \bar{\Gamma}\{l := e : \texttt{Unit}\}\bar{\Gamma}'[l \mapsto \bar{\tau}]}} \qquad \overset{\text{N-INL}}{\dfrac{\vDash \bar{\Gamma}\{e : \bar{\tau}_1\}\bar{\Gamma}'}{\vDash \bar{\Gamma}\{\texttt{inl}\ e : \bar{\tau}_1 + \bar{\tau}_2\}\bar{\Gamma}'}}$$

$$\overset{\text{N-INR}}{\dfrac{\vDash \bar{\Gamma}\{e : \bar{\tau}_2\}\bar{\Gamma}'}{\vDash \bar{\Gamma}\{\texttt{inr}\ e : \bar{\tau}_1 + \bar{\tau}_2\}\bar{\Gamma}'}} \qquad \overset{\text{N-LET}}{\dfrac{\vDash \bar{\Gamma}\{e : \bar{\tau}\}\bar{\Gamma}' \qquad \vDash \bar{\Gamma}', x : \bar{\tau}\{e' : \bar{\tau}'\}\bar{\Gamma}''}{\vDash \bar{\Gamma}\{\texttt{let}\ x = e\ \texttt{in}\ e' : \bar{\tau}'\}\bar{\Gamma}'' \setminus \{x\}}}$$

$$\overset{\text{N-CASE}}{\dfrac{\vDash \bar{\Gamma}\{e : \bar{\tau}_1 + \bar{\tau}_2\}\bar{\Gamma}' \qquad \vDash \bar{\Gamma}', x : \bar{\tau}_i\{e_i : \bar{\tau}\}\bar{\Gamma}'', x : \bar{\tau}_i \qquad i = 1,2}{\vDash \bar{\Gamma}\{\texttt{case}\ e\ \texttt{of}\ \texttt{inl}\ x.\ e_1 \quad \texttt{inr}\ x.\ e_2 : \bar{\tau}\}\bar{\Gamma}''}} \qquad \overset{\text{N-WHILE}}{\dfrac{\vDash \bar{\Gamma}\{e_1 : \texttt{Int}\}\bar{\Gamma} \qquad \vDash \bar{\Gamma}\{e_2 : \bar{\tau}\}\bar{\Gamma}}{\vDash \bar{\Gamma}\{\texttt{while}\ e_1\ \texttt{do}\ e_2 : \texttt{Unit}\}\bar{\Gamma}}}$$

Figure B.1: Typing rules for target program.

Transformation type-soundness

Lemma 30 (Context weakening). *If* $\vDash \bar{\Gamma}\{\bar{e} : \bar{\tau}\}\bar{\Gamma}'$ *and* $x \notin dom(\bar{\Gamma})$*, then* $\vDash \bar{\Gamma}, x : \bar{\tau}_x\{\bar{e} : \bar{\tau}\}\bar{\Gamma}', x : \bar{\tau}_x$.

Proof. By induction on the structure of the typing derivation. The proof is standard and we do not include it here. ■

Theorem 31 (Type-soundness). *For any source program* e*, typing environment* Γ*, and program context* pc*, if* $pc \vdash \Gamma\{e : \tau\}\Gamma'$ *then there exists target program* $\bar{e}$ *such that* $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$ *and* $\vDash [\![\Gamma]\!]\,\{\bar{e} : [\![\tau]\!]\}\,[\![\Gamma']\!]$.

The proof of this theorem relies on the following helper lemma.

Lemma 32 (Helper)**.** *For any source program e and typing environment Γ, if $H \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$ then $\vDash \llbracket\Gamma\rrbracket\, \{\forall l \in \mathsf{updated}(\bar{e}).\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma'\rrbracket$.*

Proof. By induction on the structure of e.

Case: $e = v$**.** Then $H \vdash \Gamma\{v : \tau \rightsquigarrow v\}\Gamma$ and $\mathsf{updated}(v) = \emptyset$. Hence $\vDash \llbracket\Gamma\rrbracket\, \{\forall l \in \mathsf{updated}(v).\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma\rrbracket$.

Case: $e = x$**.** Then $H \vdash \Gamma\{x : \Gamma(x) \rightsquigarrow x\}\Gamma$ and $\mathsf{updated}(x) = \emptyset$. Hence $\vDash \llbracket\Gamma\rrbracket\, \{\forall l \in \mathsf{updated}(x).\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma\rrbracket$.

Case: $e = l$**.** Then $H \vdash \Gamma\{l : \Gamma(l) \rightsquigarrow l\}\Gamma$ and $\mathsf{updated}(l) = \emptyset$. Hence $\vDash \llbracket\Gamma\rrbracket\, \{\forall l \in \mathsf{updated}(l).\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma\rrbracket$.

Case: $e = e_1 \oplus e_2$**.** From rule T-EXP-*, $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}_1, \bar{e}_2)$. From i.h., $\vDash \llbracket\Gamma\rrbracket\, \{\forall l \in \mathsf{updated}(\bar{e}_1).\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma_1\rrbracket$ and $\vDash \llbracket\Gamma_1\rrbracket\, \{\forall l \in \mathsf{updated}(\bar{e}_2).\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma_2\rrbracket$. Hence $\vDash \llbracket\Gamma\rrbracket\, \{\forall l \in \mathsf{updated}(\bar{e}_1, \bar{e}_2).\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma_2\rrbracket$.

Case: $e = l := e'$**.** From rule T-ASSIGN, $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}', l)$. From i.h., $\vDash \llbracket\Gamma\rrbracket\, \{\forall l' \in \mathsf{updated}(\bar{e}).l' := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma'\rrbracket$. As $\ell = H$, $\llbracket\sigma^H\rrbracket = \mathtt{Option}(\llbracket\sigma\rrbracket)$. Hence $\vDash \llbracket\Gamma'\rrbracket\, \{l := \mathtt{None} : \mathtt{Unit}\}\, \llbracket\Gamma'\rrbracket\, [l \mapsto \mathtt{Option}(\llbracket\sigma\rrbracket)]$.

Case: $e = \mathtt{inl}\ e'$**.** From rule T-INL, $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}')$. From i.h., $\vDash \llbracket\Gamma\rrbracket\, \{\forall l \in \mathsf{updated}(\bar{e}').\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma'\rrbracket$. Hence $\vDash \llbracket\Gamma\rrbracket\, \{\forall l \in \mathsf{updated}(\mathtt{inl}\ \bar{e}').\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma'\rrbracket$.

Case: $e = \mathtt{inr}\ e'$**.** Similar to the previous case.

Case: $e = \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$**.** From rule T-LET, $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}_1, \bar{e}_2)$. From i.h., $\vDash \llbracket\Gamma\rrbracket\, \{\forall l \in \mathsf{updated}(\bar{e}_1).\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma_1\rrbracket$ and $\vDash \llbracket\Gamma_1\rrbracket, x : \llbracket\tau_1\rrbracket\, \{\forall l \in \mathsf{updated}(\bar{e}_2).\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma_2\rrbracket$. As $x \notin \mathsf{updated}(\bar{e}_2)$, $\vDash \llbracket\Gamma_1\rrbracket\, \{\forall l \in \mathsf{updated}(\bar{e}_2).\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma_2\rrbracket \setminus \{x\}$. Hence $\vDash \llbracket\Gamma\rrbracket\, \{\forall l \in \mathsf{updated}(\bar{e}_1, \bar{e}_2).\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma_2\rrbracket \setminus \{x\}$.

Case: $\mathtt{case}\ e'\ \mathtt{of}\ \mathtt{inl}\ x.\ e_1 \quad \mathtt{inr}\ x.\ e_2$**.** From rule T-CASE-*, $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}', \bar{e}_1, \bar{e}_2)$. But $\mathsf{updated}(\bar{e}_i) = S \cup S_i$, where S denotes the set of memory locations updated in both branches $\bar{e}_1$ and $\bar{e}_2$. Hence $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}') \cup S \cup S_1 \cup S_2$.

From i.h., $\vDash \llbracket\Gamma\rrbracket\, \{\forall l \in \mathsf{updated}(\bar{e}').\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma'\rrbracket$ and $\vDash \llbracket\Gamma'\rrbracket, x : \llbracket\tau_i\rrbracket\, \{\forall l \in \mathsf{updated}(\bar{e}_i).\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma_i\rrbracket$. Using the relation above, we get that

$$\begin{array}{l}
\vDash \llbracket\Gamma'\rrbracket\, \{\forall l \in S \cup S_1.\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma_1\rrbracket \setminus \{x_1\} \\
\vDash \llbracket\Gamma'\rrbracket\, \{\forall l \in S \cup S_2.\ l := \mathtt{None}; 0 : \mathtt{Int}\}\, \llbracket\Gamma_2\rrbracket \setminus \{x_2\}
\end{array}$$

Hence $\vDash \bar{\Gamma}'\{\forall l \in S \cup S_1 \cup S_2.\ l := \mathtt{None}; 0 : \mathtt{Int}\}\bar{\Gamma}''$ for some Γ''. It remains to show that $\bar{\Gamma}'' = \llbracket \bigsqcup \Gamma_i \setminus \{x\} \rrbracket$.

Let $high(\Gamma)$ be the set of all memory locations with a high type in Γ and $low(\Gamma)$ be the set of all memory locations with low type in Γ. Then,

$$\begin{array}{rcl}
\bigsqcup \Gamma_i \setminus \{x\} & = & (\Gamma_1 \setminus \{x\}) \sqcup (\Gamma_2 \setminus \{x\}) \\
& = & (high(\Gamma_1 \setminus \{x\}) \cap low(\Gamma_2 \setminus \{x\})) \cup (high(\Gamma_2 \setminus \{x\}) \cap low(\Gamma_1 \setminus \{x\})) \cup \\
& & (high(\Gamma_1 \setminus \{x\}) \cap high(\Gamma_2 \setminus \{x\})) \cup (low(\Gamma_1 \setminus \{x\}) \cap low(\Gamma_2 \setminus \{x\})) \\
& = & S_1 \cup S_2 \cup S \cup (low(\Gamma_1 \setminus \{x\}) \cap low(\Gamma_2 \setminus \{x\}))
\end{array}$$

Hence $\llbracket\Gamma''\rrbracket = \llbracket \bigsqcup \Gamma_i \setminus \{x\} \rrbracket$.

Case: $\texttt{while}\ x\ \texttt{do}\ e'$. From rule T-WHILE-*, $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}')$. From i.h., $\vDash [\![\Gamma]\!]\,\{\forall l \in \mathsf{updated}(\bar{e}').\ l := \texttt{None}; 0 : \texttt{Int}\}\,[\![\Gamma]\!]$. Hence $\vDash [\![\Gamma]\!]\,\{\forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None}; 0 : \texttt{Int}\}\,[\![\Gamma]\!]$.

Case: $e = \bullet e'$. From rule T-UPGRADE, $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}')$. From i.h. $\vDash [\![\Gamma]\!]\,\{\forall l \in \mathsf{updated}(\bar{e}').\ l := \texttt{None}; 0 : \texttt{Int}\}\,[\![\Gamma']\!]$. Hence $\vDash [\![\Gamma]\!]\,\{\forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None}; 0 : \texttt{Int}\}\,[\![\Gamma']\!]$. ■

Proof of Theorem 31. By induction on the structure of type-directed transformation $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$.

Case: T-VAL. From rule N-VAL and $\tau = \sigma^L$.

Case: T-VAR. From typing environment transformation and rule N-VAR.

Case: T-VARLOC. From typing environment transformation and rule N-VARLOC.

Case: T-EXP-L. From i.h. applied to $pc \vdash \Gamma\{e_1 : \texttt{Int}^L \rightsquigarrow \bar{e}_1\}\Gamma_1$ and, respectively $pc \vdash \Gamma_1\{e_2 : \texttt{Int}^L \rightsquigarrow \bar{e}_2\}\Gamma_2$, we get that $\vDash [\![\Gamma]\!]\,\{\bar{e}_1 : \texttt{Int}\}\,[\![\Gamma_1]\!]$ and, respectively, $\vDash [\![\Gamma_1]\!]\,\{\bar{e}_2 : \texttt{Int}\}\,[\![\Gamma_2]\!]$. By applying rule N-EXP we obtain the desired result:

$$\frac{\vDash [\![\Gamma]\!]\,\{\bar{e}_1 : \texttt{Int}\}\,[\![\Gamma_1]\!] \qquad \vDash [\![\Gamma_1]\!]\,\{\bar{e}_2 : \texttt{Int}\}\,[\![\Gamma_2]\!]}{\vDash [\![\Gamma]\!]\,\{\bar{e}_1 \oplus \bar{e}_2 : \texttt{Int}\}\,[\![\Gamma_2]\!]}\ \text{N-EXP}$$

Case: T-EXP-H. From i.h. applied to $pc \vdash \Gamma\{e_1 : \texttt{Int}^H \rightsquigarrow \bar{e}_1\}\Gamma_1$ and $pc \vdash \Gamma_1\{e_2 : \texttt{Int}^H \rightsquigarrow \bar{e}_2\}\Gamma_2$, we get that $\vDash [\![\Gamma]\!]\,\{\bar{e}_1 : \texttt{Option(Int)}\}\,[\![\Gamma_1]\!]$ and $\vDash [\![\Gamma_1]\!]\,\{\bar{e}_2 : \texttt{Option(Int)}\}\,[\![\Gamma_2]\!]$. The desired result follows from the derivation in Figure B.2.

Case: T-ASSIGN. From i.h. applied to $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$, we get that $\vDash [\![\Gamma]\!]\,\{\bar{e} : [\![\tau]\!]\}\,[\![\Gamma']\!]$. By applying rule N-ASSIGN we obtain the desired result:

$$\frac{\vDash [\![\Gamma]\!]\,\{\bar{e} : [\![\tau]\!]\}\,[\![\Gamma']\!]}{\vDash [\![\Gamma]\!]\,\{l := \bar{e} : \texttt{Unit}\}\,[\![\Gamma']\!]\,[l \mapsto [\![\tau]\!]]}\text{N-ASSIGN}$$

Case: T-INL. From i.h. applied to $pc \vdash \Gamma\{e : \tau_1 \rightsquigarrow \bar{e}\}\Gamma'$, we get that $\vDash [\![\Gamma]\!]\,\{\bar{e} : [\![\tau_1]\!]\}\,[\![\Gamma']\!]$. By applying rule N-INL we obtain the desired result:

$$\frac{\vDash [\![\Gamma]\!]\,\{\bar{e} : [\![\tau_1]\!]\}\,[\![\Gamma']\!]}{\vDash [\![\Gamma]\!]\,\{\texttt{inl}\ \bar{e} : [\![\tau_1]\!] + [\![\tau_2]\!]\}\,[\![\Gamma']\!]}\ \text{N-INL}$$

Case: T-INR. Similar to case T-INL.

Case: T-LET. From i.h. applied to $pc \vdash \Gamma\{e_1 : \tau_1 \rightsquigarrow \bar{e}_1\}\Gamma_1$ and, respectively, $pc \vdash \Gamma_1, x : \tau_1\{e_2 : \tau_2 \rightsquigarrow \bar{e}_2\}\Gamma_2$, we get that $\vDash [\![\Gamma]\!]\,\{\bar{e}_1 : [\![\tau_1]\!]\}\,[\![\Gamma_1]\!]$ and, respectively, $\vDash [\![\Gamma_1]\!], x : [\![\tau_1]\!]\,\{\bar{e}_2 : [\![\tau_2]\!]\}\,[\![\Gamma_2]\!]$. By applying rule N-LET we obtain the desired result:

$$\frac{\vDash [\![\Gamma]\!]\,\{\bar{e}_1 : [\![\tau_1]\!]\}\,[\![\Gamma_1]\!] \qquad \vDash [\![\Gamma_1]\!], x : [\![\tau_1]\!]\,\{\bar{e}_2 : [\![\tau_2]\!]\}\,[\![\Gamma_2]\!]}{\vDash [\![\Gamma]\!]\,\{\texttt{let}\ x = \bar{e}_1\ \texttt{in}\ \bar{e}_2 : [\![\tau_2]\!]\}\,[\![\Gamma_2]\!] \setminus \{x\}}\ \text{N-LET}$$

Case: T-CASE-L. From i.h. applied to $pc \vdash \Gamma\{e : (\tau_1 + \tau_2)^L \rightsquigarrow \bar{e}\}\Gamma'$ and, respectively, $pc \vdash \Gamma', x : \tau_i\{e_i : \tau \rightsquigarrow \bar{e}_i\}\Gamma_i$, we get that $\vDash [\![\Gamma]\!]\,\{\bar{e} : [\![\tau_1]\!] + [\![\tau_2]\!]\}\,[\![\Gamma']\!]$ and, respectively, $\vDash [\![\Gamma']\!], x : [\![\tau_i]\!]\,\{\bar{e}_i : [\![\tau]\!]\}\,[\![\Gamma_i]\!]$, for $i = 1, 2$.

$$
\dfrac{
\vDash \llbracket\Gamma\rrbracket\{\bar{e}_1 : \texttt{Option(Int)}\}\llbracket\Gamma_1\rrbracket
\quad
\dfrac{
\dfrac{\vDash \llbracket\Gamma_1\rrbracket\{\bar{e}_2 : \texttt{Option(Int)}\}\llbracket\Gamma_2\rrbracket}{\vDash \llbracket\Gamma_1\rrbracket, x : \texttt{Option(Int)}\{\bar{e}_2 : \texttt{Option(Int)}\}\bar\Gamma'_2}\,\textsc{Lem. 30}
\quad
\dfrac{
\dfrac{\bar\Gamma''_2(x) = \texttt{Option(Int)}}{\vDash \bar\Gamma''_2\{x : \texttt{Option(Int)}\}\bar\Gamma''_2}\,\text{N-VAR}
\quad
\dfrac{}{\vDash \bar\Gamma''_2, x' : \texttt{Unit}\{\texttt{None} : \texttt{Option(Int)}\}\bar\Gamma''_2, x' : \texttt{Unit}}\,\text{N-VAL}
\quad
\dfrac{
\dfrac{\bar\Gamma''_2(y) = \texttt{Option(Int)}}{\vDash \bar\Gamma''_2, x' : \texttt{Int}\{y : \texttt{Option(Int)}\}\bar\Gamma''_2, x' : \texttt{Int}}\,\text{N-VAR}
\quad
\dfrac{}{\vDash \bar\Gamma''_2, x' : \texttt{Int}\{\texttt{None} : \texttt{Option(Int)}\}\bar\Gamma''_2, x' : \texttt{Int}}\,\text{N-VAL}
\quad
\dfrac{
\dfrac{
\dfrac{\bar\Gamma'''_2(x') = \texttt{Int}}{\vDash \bar\Gamma'''_2\{x' : \texttt{Int}\}\bar\Gamma'''_2}\,\text{N-VAR}
\quad
\dfrac{\bar\Gamma'''_2(y') = \texttt{Int}}{\vDash \bar\Gamma'''_2\{y' : \texttt{Int}\}\bar\Gamma'''_2}\,\text{N-VAR}
}{\vDash \bar\Gamma''_2, x' : \texttt{Int}, y' : \texttt{Int}\{x' \oplus y'\}\bar\Gamma''_2, x' : \texttt{Int}, y' : \texttt{Int}}\,\text{N-EXP}
}{\vDash \bar\Gamma''_2, x' : \texttt{Int}, y' : \texttt{Int}\{\texttt{Some}\ (x' \oplus y')\}\bar\Gamma''_2, x' : \texttt{Int}, y' : \texttt{Int}}\,\text{N-INR}
}{\begin{array}{l}\vdash \bar\Gamma''_2, x' : \texttt{Int}\{\mathsf{case}\ y\ \mathsf{of}\ \texttt{None}.\ \texttt{None}\\ \qquad\qquad \mathsf{Some}\ y'.\ \mathsf{Some}\ (x' \oplus y') : \texttt{Option(Int)}\}\bar\Gamma''_2, x' : \texttt{Int}\end{array}}\,\text{N-CASE}
}{\begin{array}{l}\vDash \bar\Gamma''_2\{\mathsf{case}\ x\ \mathsf{of}\ \texttt{None}.\ \texttt{None}\\ \qquad\qquad \mathsf{Some}\ x'.\ \mathsf{case}\ y\ \mathsf{of}\ \texttt{None}.\ \texttt{None} \quad \mathsf{Some}\ y'.\ \mathsf{Some}\ (x' \oplus y') : \texttt{Option(Int)}\}\bar\Gamma''_2\end{array}}\,\text{N-CASE}
}{\begin{array}{l}\vDash \llbracket\Gamma_1\rrbracket, x : \texttt{Option(Int)}\{\texttt{let}\ y = \bar{e}_2\ \texttt{in}\\ \qquad\qquad \mathsf{case}\ x\ \mathsf{of}\ \texttt{None}.\ \texttt{None}\\ \qquad\qquad\qquad \mathsf{Some}\ x'.\ \mathsf{case}\ y\ \mathsf{of}\ \texttt{None}.\ \texttt{None} \quad \mathsf{Some}\ y'.\ \mathsf{Some}\ (x' \oplus y') : \texttt{Option(Int)}\}\bar\Gamma'_2\end{array}}\,\text{N-LET}
}{\begin{array}{l}\vDash \llbracket\Gamma\rrbracket\{\texttt{let}\ x = \bar{e}_1\ \texttt{in let}\ y = \bar{e}_2\ \texttt{in}\\ \qquad \mathsf{case}\ x\ \mathsf{of}\ \texttt{None}.\ \texttt{None} \quad \mathsf{Some}\ x'.\ (\mathsf{case}\ y\ \mathsf{of}\ \texttt{None}.\ \texttt{None} \quad \mathsf{Some}\ y'.\ \mathsf{Some}\ (x' \oplus y')) : \texttt{Option(Int)}\}\llbracket\Gamma_2\rrbracket\end{array}}\,\text{N-LET}
$$

where $\bar\Gamma'_2 = \llbracket\Gamma_2\rrbracket, x : \texttt{Option(Int)}$, $\bar\Gamma''_2 = \bar\Gamma'_2, y : \texttt{Option(Int)}$, $\bar\Gamma'''_2 = \bar\Gamma''_2, x' : \texttt{Int}, y' : \texttt{Int}$.

Figure B.2: Concluding derivation, case T-EXP-H, Theorem 31.

$$
\dfrac{
\vDash \llbracket\Gamma\rrbracket\{\bar{e} : \llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket\}\llbracket\Gamma'\rrbracket
\quad
\dfrac{
\vDash \llbracket\Gamma'\rrbracket, x : \llbracket\tau_1\rrbracket\{\bar{e}_1 : \llbracket\tau\rrbracket\}\llbracket\Gamma_1\rrbracket
\quad
\dfrac{
\dfrac{}{\vDash \llbracket\Gamma_1\rrbracket, y : \llbracket\tau\rrbracket\{\mathsf{upgrade}(S_2) : \texttt{Unit}\}\bar\Gamma'_1}\,\text{N-ASSIGN}^*
\quad
\dfrac{\bar\Gamma'_1(y) = \llbracket\tau\rrbracket}{\vDash \bar\Gamma'_1\{y : \llbracket\tau\rrbracket\}\bar\Gamma'_1}\,\text{N-VAR}
}{\vDash \llbracket\Gamma_1\rrbracket, y : \llbracket\tau\rrbracket\{\mathsf{upgrade}(S_2); y : \llbracket\tau\rrbracket\}\bar\Gamma'_1}\,\text{N-SEQ}
}{\vDash \llbracket\Gamma'\rrbracket, x : \llbracket\tau_1\rrbracket\{\texttt{let}\ y_1 = \bar{e}_1\ \texttt{in}\ \mathsf{upgrade}(S_2); y_1 : \llbracket\tau\rrbracket\}\bar\Gamma''_1}\,\text{N-LET}
\quad
\dfrac{
\vDash \llbracket\Gamma'\rrbracket, x : \llbracket\tau_2\rrbracket\{\bar{e}_2 : \llbracket\tau_2\rrbracket\}\llbracket\Gamma_2\rrbracket
\quad
\dfrac{
\dfrac{}{\vDash \llbracket\Gamma_2\rrbracket, y : \llbracket\tau\rrbracket\{\mathsf{upgrade}(S_1) : \texttt{Unit}\}\bar\Gamma'_2}\,\text{N-ASSIGN}^*
\quad
\dfrac{\bar\Gamma'_2(y) = \llbracket\tau\rrbracket}{\vDash \bar\Gamma'_2\{y : \llbracket\tau\rrbracket\}\bar\Gamma'_2}\,\text{N-VAR}
}{\vDash \llbracket\Gamma_2\rrbracket, y : \llbracket\tau\rrbracket\{\mathsf{upgrade}(S_1); y : \llbracket\tau\rrbracket\}\bar\Gamma'_2}\,\text{N-SEQ}
}{\vDash \llbracket\Gamma'\rrbracket, x : \llbracket\tau_2\rrbracket\{\texttt{let}\ y = \bar{e}_2\ \texttt{in}\ \mathsf{upgrade}(S_1); y : \llbracket\tau\rrbracket\}\bar\Gamma''_2}\,\text{N-LET}
}{\vDash \llbracket\Gamma\rrbracket\{\mathsf{case}\ \bar{e}\ \texttt{of inl}\ x.\ \texttt{let}\ y = \bar{e}_1\ \texttt{in}\ \mathsf{upgrade}(S_2); y \quad \texttt{inr}\ x.\ \texttt{let}\ y = \bar{e}_2\ \texttt{in}\ \mathsf{upgrade}(S_1); y : \llbracket\tau\rrbracket\}\bar\Gamma''}\,\text{N-CASE}
$$

where $\bar\Gamma''_i = \bar\Gamma'_i \setminus \{y\}$, for $i = 1, 2$.

Rule N-CASE requires $\bar{\Gamma}''_1 \setminus \{x\} = \bar{\Gamma}''_2 \setminus \{x\} = \bar{\Gamma}''$. We prove this below:

From the derivation above, $\Gamma'_1 \setminus \Gamma_1 = S_2$ and $\Gamma'_2 \setminus \Gamma_2 = S_1$. Since $\{x,y\} \not\in S_2$ and $\{x,y\} \not\in S_1$, it follows that:

$$S_1 = (\Gamma'_2 \setminus \{x\}) \setminus (\Gamma_2 \setminus \{x\}) \qquad S_2 = (\Gamma'_1 \setminus \{x\}) \setminus (\Gamma_1 \setminus \{x\}).$$

Let $high(\Gamma)$ be the set of all memory locations with a high type in Γ and $low(\Gamma)$ be the set of all memory locations with low type in Γ. Then, S_1 and S_2 can be written as:

$$S_1 = high(\Gamma'_2 \setminus \{x\}) \cap low(\Gamma_2 \setminus \{x\}) \qquad S_2 = high(\Gamma'_1 \setminus \{x\}) \cap low(\Gamma_1 \setminus \{x\}).$$

In addition, $\Gamma_1 \setminus \Gamma' = S_1 \cup S$ and $\Gamma_2 \setminus \Gamma' = S_2 \cup S$, where S is the set containing the memory locations upgraded in both branches. Hence, $S_1 \cup S = high(\Gamma_1) \cap low(\Gamma')$ and $S_2 \cup S = high(\Gamma_2) \cap low(\Gamma')$. Since $\{x,y\} \not\in \Gamma'$ and $\{x,y\} \not\in \Gamma'$, $S_1 \cup S = high(\Gamma_1 \setminus \{x,y\}) \cap low(\Gamma')$ and $S_2 \cup S = high(\Gamma_2 \setminus \{x,y\}) \cap low(\Gamma')$.

Hence

$$\begin{array}{rcl} (S_1 \cup S) \cup S_2 & = & high(\Gamma_1 \setminus \{x,y\}) \cap low(\Gamma') \cup high(\Gamma'_1 \setminus \{x,y\}) \cap low(\Gamma_1 \setminus \{x,y\}) \\ & = & high(\Gamma'_1 \setminus \{x,y\}) \cap low(\Gamma') \end{array}$$

and

$$\begin{array}{rcl} S_1 \cup (S_2 \cup S) & = & high(\Gamma_2 \setminus \{x,y\}) \cap low(\Gamma') \cup high(\Gamma'_2 \setminus \{x,y\}) \cap low(\Gamma_2 \setminus \{x,y\}) \\ & = & high(\Gamma'_2 \setminus \{x,y\}) \cap low(\Gamma'). \end{array}$$

Hence $high(\Gamma'_1 \setminus \{x,y\}) = high(\Gamma'_2 \setminus \{x,y\})$. Since $dom(\Gamma'_1 \setminus \{x,y\}) = dom(\Gamma'_2 \setminus \{x,y\})$, $\Gamma'_1 \setminus \{x,y\} = \Gamma'_2 \setminus \{x,y\}$. Thus $\bar{\Gamma}'_1 \setminus \{x,y\} = \bar{\Gamma}'_2 \setminus \{x,y\} = \bar{\Gamma}''$, i.e. $\bar{\Gamma}''_i \setminus \{x\} = \bar{\Gamma}''$.

We are left to show that $\bar{\Gamma}'' = [\![\bigsqcup \Gamma_i \setminus \{x\}]\!]$. I.e. we have to show that $high(\Gamma''_1 \setminus \{x\}) = high(\Gamma_1 \setminus \{x\}) \cup high(\Gamma_2 \setminus \{x\})$. But

$$\begin{array}{rcl} high(\Gamma_1 \setminus \{x\}) & \supseteq & high(\Gamma_1 \setminus \{x\}) \cup S_2 \\ & = & high(\Gamma'_1 \setminus \{y,x\}) \\ & = & high(\Gamma''_1 \setminus \{x\}) \end{array}$$

Hence $high(\Gamma_1 \setminus \{x\}) \supseteq high(\Gamma''_1 \setminus \{x\})$. Similarly, $high(\Gamma_2 \setminus \{x\}) \supseteq high(\Gamma''_2 \setminus \{x\})$. But $high(\Gamma''_1 \setminus \{x\}) = high(\Gamma''_2 \setminus \{x\})$ and $dom(\Gamma''_1 \setminus \{x\}) = dom(\Gamma_1 \setminus \{x\}) = dom(\Gamma_2 \setminus \{x\})$.

Hence $high(\Gamma''_1 \setminus \{x\}) = high(\Gamma_1 \setminus \{x\}) \cup high(\Gamma_2 \setminus \{x\})$. Thus $\bar{\Gamma}''_1 \setminus \{x\} = [\![\bigsqcup \Gamma_i \setminus \{x\}]\!]$.

Case: T-CASE-H. From i.h. applied to $pc \vdash \Gamma\{e : (\tau_1 + \tau_2)^H \rightsquigarrow \bar{e}\}\Gamma'$ and $H \vdash \Gamma', x : \tau_i\{e_i : \tau \rightsquigarrow \bar{e}_i\}\Gamma_i$, we get that $\models [\![\Gamma]\!]\,\{\bar{e} : \texttt{Option}([\![\tau_1]\!] + [\![\tau_2]\!])\}\,[\![\Gamma']\!]$ and $\models [\![\Gamma']\!], x : [\![\tau_i]\!]\,\{\bar{e}_i : [\![\tau]\!]\}\,[\![\Gamma_i]\!]$, for $i = 1,2$.

Since $pc \vdash \Gamma\{e : (\tau_1 + \tau_2)^H \rightsquigarrow \bar{e}\}\Gamma'$, it must be the case that $\tau = \sigma^H$, for some σ, hence $[\![\tau]\!] =$

$\mathtt{Option}(\llbracket\sigma\rrbracket)$. The desired result follows from the derivation below:

$$\dfrac{\begin{array}{c}
\vDash \llbracket\Gamma\rrbracket\{\bar{e} : \mathtt{Option}(\llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket)\}\llbracket\Gamma'\rrbracket \\
\dfrac{\cdots}{\vDash \llbracket\Gamma'\rrbracket, x' : \mathtt{Unit}\{(\forall l \in \mathsf{updated}(\bar{e}_1, \bar{e}_2).\ l := \mathtt{None}); \mathtt{None} : \llbracket\tau\rrbracket\}\bar{\Gamma}'''}\ \text{N-SEQ} \\
\dfrac{(\llbracket\Gamma'\rrbracket, x' : \llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket)(x') = \llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket}{\vDash \llbracket\Gamma'\rrbracket, x' : \llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket\{x' : \llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket\}\llbracket\Gamma'\rrbracket, x' : \llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket}\ \text{N-VAR} \\
\dfrac{\begin{array}{c}
\dfrac{\dfrac{\vDash \llbracket\Gamma'\rrbracket, x : \llbracket\tau_1\rrbracket\{\bar{e}_1 : \llbracket\tau\rrbracket\}\llbracket\Gamma_1\rrbracket}{\vDash \llbracket\Gamma'\rrbracket, x' : \llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket, x : \llbracket\tau_1\rrbracket\{\bar{e}_1 : \llbracket\tau\rrbracket\}\bar{\Gamma}'_1}\ \textsc{Lem. 30} \quad \dfrac{\begin{array}{c}\dfrac{}{\vDash \bar{\Gamma}'_1, y : \llbracket\tau\rrbracket\{\mathsf{upgrade}(S_2) : \mathtt{Unit}\}\bar{\Gamma}''_1}\ \text{N-ASSIGN}^* \\ \dfrac{\bar{\Gamma}''_1(y) = \llbracket\tau\rrbracket}{\vDash \bar{\Gamma}''_1\{y : \llbracket\tau\rrbracket\}\bar{\Gamma}''_1}\ \text{N-VAR}\end{array}}{\vDash \bar{\Gamma}'_1, y : \llbracket\tau\rrbracket\{\mathsf{upgrade}(S_2); y : \llbracket\tau\rrbracket\}\bar{\Gamma}''_1}\ \text{N-SEQ}}{\vDash \llbracket\Gamma'\rrbracket, x' : \llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket, x : \llbracket\tau_1\rrbracket\{\mathtt{let}\ y = \bar{e}_1\ \mathtt{in}\ \mathsf{upgrade}(S_2); y : \llbracket\tau\rrbracket\}\bar{\Gamma}'''_1}\ \text{N-LET} \\
\dfrac{\dfrac{\vDash \llbracket\Gamma'\rrbracket, x : \llbracket\tau_2\rrbracket\{\bar{e}_2 : \llbracket\tau\rrbracket\}\llbracket\Gamma_2\rrbracket}{\vDash \llbracket\Gamma'\rrbracket, x' : \llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket, x : \llbracket\tau_2\rrbracket\{\bar{e}_2 : \llbracket\tau\rrbracket\}\bar{\Gamma}'_2}\ \textsc{Lem. 30} \quad \dfrac{\begin{array}{c}\dfrac{}{\vDash \bar{\Gamma}'_2, y : \llbracket\tau\rrbracket\{\mathsf{upgrade}(S_1) : \mathtt{Unit}\}\bar{\Gamma}''_2}\ \text{N-ASSIGN}^* \\ \dfrac{\bar{\Gamma}''_2(y) = \llbracket\tau\rrbracket}{\vDash \bar{\Gamma}''_2\{y : \llbracket\tau\rrbracket\}\bar{\Gamma}''_2}\ \text{N-VAR}\end{array}}{\vDash \bar{\Gamma}'_2, y : \llbracket\tau\rrbracket\{\mathsf{upgrade}(S_1); y : \llbracket\tau\rrbracket\}\bar{\Gamma}''_2}\ \text{N-SEQ}}{\vDash \llbracket\Gamma'\rrbracket, x' : \llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket, x : \llbracket\tau_2\rrbracket\{\mathtt{let}\ y = \bar{e}_2\ \mathtt{in}\ \mathsf{upgrade}(S_1); y : \llbracket\tau\rrbracket\}\bar{\Gamma}'''_2}\ \text{N-LET}
\end{array}}{\vDash \llbracket\Gamma'\rrbracket, x' : \llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket\{\mathtt{case}\ x'\ \mathtt{of}\ \mathtt{inl}\ x.\ \mathtt{let}\ y = \bar{e}_1\ \mathtt{in}\ \mathsf{upgrade}(S_2); y \quad \mathtt{inr}\ x.\ \mathtt{let}\ y = \bar{e}_2\ \mathtt{in}\ \mathsf{upgrade}(S_1); y : \llbracket\tau\rrbracket\}\bar{\Gamma}''}\ \text{N-CASE}
\end{array}}{\begin{array}{l}
\vDash \llbracket\Gamma\rrbracket\{\mathsf{case}\ \bar{e}\ \mathsf{of}\ \mathtt{None}.\ (\forall l \in \mathsf{updated}(\bar{e}_i).\ l := \mathtt{None}); \mathtt{None} \\
\qquad\qquad \mathtt{Some}\ x'.\ (\mathtt{case}\ x'\ \mathtt{of}\ \mathtt{inl}\ x.\ \mathtt{let}\ y = \bar{e}_1\ \mathtt{in}\ \mathsf{upgrade}(S_2); y \quad \mathtt{inr}\ x.\ \mathtt{let}\ y = \bar{e}_2\ \mathtt{in}\ \mathsf{upgrade}(S_1); y) : \llbracket\tau\rrbracket\}\bar{\Gamma}'' \setminus \{x\}
\end{array}}\ \text{N-CASE}$$

where $\bar{\Gamma}'_i = \llbracket\Gamma_i\rrbracket, x' : \llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket$; $\bar{\Gamma}'''_i = \bar{\Gamma}''_i \setminus \{y\}$, $i = 1, 2$.

Proving $\bar{\Gamma}'''_1 \setminus \{x\} = \bar{\Gamma}''_2 \setminus \{x\} = \bar{\Gamma}''$ and $\bar{\Gamma}'' = \llbracket \bigsqcup \Gamma_i \setminus \{x\} \rrbracket$ follows a similar reasoning as in the previous case T-CASE-L. We are left to show that $\bar{\Gamma}''' = \bar{\Gamma}''$.

$\mathsf{updated}(\bar{e}_1, \bar{e}_2) = S \cup S_1 \cup S_2 \cup S'$, where S is defined as previously (the set containing the memory locations upgraded in both branches) and S' is the set containing all the memory locations which get updated in both $\bar{e}_1$ and $\bar{e}_2$, but not upgraded. I.e., S' contains all the memory locations that already have an Option type. Thus, $\vDash \llbracket\Gamma\rrbracket, x' : \mathtt{Unit}\{\forall l \in S \cup S_1 \cup S_2 \cup S'.\ l := \mathtt{None} : \mathtt{Unit}\}\bar{\Gamma}'''$. Since assigning $\mathtt{None}$ to the memory locations in S' does not affect the typing environment, $\vDash \llbracket\Gamma\rrbracket, x' : \mathtt{Unit}\{\forall l \in S \cup S_1 \cup S_2.\ l := \mathtt{None} : \mathtt{Unit}\}\bar{\Gamma}'''$.

As $pc = H$, for the memory locations in S, S_1, and S_2, updating them has the same effect on the typing environment as upgrading them. I.e. $\vDash \llbracket\Gamma'\rrbracket, x' : \llbracket\tau_1\rrbracket + \llbracket\tau_2\rrbracket\{\mathsf{upgrade}(S, S_1, S_2) : \mathtt{Unit}\}\bar{\Gamma}''$. Hence $\bar{\Gamma}''' = \bar{\Gamma}''$ (since $x' \notin S \cup S_1 \cup S_2$).

Case: T-WHILE-L. From i.h. applied to $pc \vdash \Gamma\{e' : \tau \rightsquigarrow \bar{e}'\}\Gamma$, we get that $\vDash \llbracket\Gamma\rrbracket\{\bar{e}' : \llbracket\tau\rrbracket\}\llbracket\Gamma\rrbracket$. By applying rule N-WHILE we obtain the desired result:

$$\dfrac{\llbracket\Gamma\rrbracket(x) = \mathtt{Int} \qquad \vDash \llbracket\Gamma\rrbracket\{\bar{e}' : \llbracket\tau\rrbracket\}\llbracket\Gamma\rrbracket}{\vDash \llbracket\Gamma\rrbracket\{\mathtt{while}\ x\ \mathtt{do}\ \bar{e}' : \mathtt{Unit}\}\llbracket\Gamma\rrbracket}\ \text{N-WHILE}$$

Case: T-WHILE-H. The transformation derivation is of the form

$$\dfrac{\Gamma(x) = \mathtt{Int}^H \qquad H \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma}{\begin{array}{l} pc \vdash \Gamma\{\mathtt{while}\ x\ \mathtt{do}\ e : \mathtt{Unit}^L \rightsquigarrow \\ \qquad \mathtt{while}\ (\mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ (\forall l \in \mathsf{updated}(\bar{e}).\ l := \mathtt{None};)\ 0 \quad \mathtt{Some}\ y.\ y)\ \mathtt{do}\ \bar{e}\}\Gamma \end{array}}\ \text{T-WHILE-H}$$

From i.h. applied to $H \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma$, we get that $\vDash \llbracket\Gamma\rrbracket \{\bar{e} : \llbracket\tau\rrbracket\} \llbracket\Gamma\rrbracket$. The desired result follows from the derivation below:

$$\dfrac{\dfrac{\llbracket\Gamma\rrbracket(x) = \texttt{Option(Int)} \qquad \dfrac{\dfrac{}{\vDash \llbracket\Gamma\rrbracket \{\forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None};\texttt{0} : \texttt{Int}\} \llbracket\Gamma\rrbracket}\ \textsc{Lem. 32}}{\vDash \llbracket\Gamma\rrbracket, y : \texttt{Unit}\{\forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None};\texttt{0} : \texttt{Int}\} \llbracket\Gamma\rrbracket, y : \texttt{Unit}}\ \textsc{Lem. 30} \qquad \dfrac{}{\vDash \llbracket\Gamma\rrbracket, y : \texttt{Int}\{y : \texttt{Int}\} \llbracket\Gamma\rrbracket, y : \texttt{Int}}\ \text{N-VAR}}{\vDash \llbracket\Gamma\rrbracket \{\texttt{case}\ x\ \texttt{of None.}\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None};\texttt{0} \quad \texttt{Some}\ y.\ y : \texttt{Int}\} \llbracket\Gamma\rrbracket}\ \text{N-CASE} \qquad \vDash \llbracket\Gamma\rrbracket \{\bar{e} : \llbracket\tau\rrbracket\} \llbracket\Gamma\rrbracket}{\vDash \llbracket\Gamma\rrbracket \{\texttt{while (case}\ x\ \texttt{of None.}\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None};\texttt{0} \quad \texttt{Some}\ y.\ y\texttt{) do}\ \bar{e} : \texttt{Unit}\} \llbracket\Gamma\rrbracket}\ \text{N-WHILE}$$

Case: T-UPGRADE. From i.h. applied to $pc \vdash \Gamma\{e : \sigma^L \rightsquigarrow \bar{e}\}\Gamma'$, we get that $\vDash \llbracket\Gamma\rrbracket \{\bar{e} : \llbracket\sigma\rrbracket\} \llbracket\Gamma'\rrbracket$. By applying rule N-INL we obtain the desired result:

$$\dfrac{\vDash \llbracket\Gamma\rrbracket \{\bar{e} : \llbracket\sigma\rrbracket\} \llbracket\Gamma'\rrbracket}{\vDash \llbracket\Gamma\rrbracket \{\texttt{Some}\ \bar{e} : \texttt{Option}(\llbracket\sigma\rrbracket)\} \llbracket\Gamma'\rrbracket}\ \text{N-INL}$$

■

Transformation correctness

Lemma 33 (Memory weakening). *If $\langle e,\mu\rangle \Downarrow \langle v_f,\mu_f\rangle$ and $x \notin dom(\mu)$, then $\langle e,\mu \cup \{x \mapsto v'\}\rangle \Downarrow \langle v_f,\mu_f \cup \{x \mapsto v'\}\rangle$.*

Proof. By induction on the derivation of the evaluation relation $\langle e,\mu\rangle \Downarrow \langle v_f,\mu_f\rangle$. The proof is standard and we do not include it here. ■

Theorem 34 (Correctness, equivalence between source and target programs). *For any source program e, typing environment Γ, program context pc, and store μ, if $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$ and $\langle e,\mu\rangle \Downarrow \langle v_f,\mu_f\rangle$, then there exists $\bar{v}_f$ and $\bar{\mu}_f$, such that $\langle \bar{e}, \llbracket\mu\rrbracket_\Gamma\rangle \Downarrow \langle \bar{v}_f,\bar{\mu}_f\rangle$ and $v_f \sim_\tau \bar{v}_f$ and $\mu_f \sim_{\Gamma'} \bar{\mu}_f$.*

Proof. By induction on the derivation of the evaluation relation $\langle e,\mu\rangle \Downarrow \langle v_f,\mu_f\rangle$.

Let $\llbracket\mu\rrbracket_\Gamma = \bar{\mu}$. Then, from Lemma 16, $\mu \sim_\Gamma \bar{\mu}$. Suppose further that $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$ and $\langle e,\mu\rangle \Downarrow \langle v_f,\mu_f\rangle$. We prove that $\langle \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v}_f,\bar{\mu}_f\rangle$, where $v_f \sim_\tau \bar{v}_f$ and $\mu_f \sim_{\Gamma'} \bar{\mu}_f$ by induction on the derivation of $\langle e,\mu\rangle \Downarrow \langle v_f,\mu_f\rangle$ and cases according to the last rule applied.

Case: E-VAL. The transformation derivation is of the form

$$\dfrac{}{pc \vdash \Gamma\{v : \tau \rightsquigarrow v\}\Gamma}\ \text{T-VAL}$$

Suppose that $\langle v,\mu\rangle \Downarrow \langle v,\mu\rangle$. By applying rule E-VAL we get that $\bar{v}_f = v$ and $\bar{\mu}_f = \bar{\mu}$. Hence, $v_f \sim_\tau \bar{v}_f$ and $\mu_f \sim_{\Gamma'} \bar{\mu}_f$ (from hypothesis and since $\Gamma' = \Gamma$).

Case: E-VARLOC. If $l \in dom(\mu)$, then $l \in dom(\bar{\mu})$ (from store equivalence, Definition 15). By applying rule E-VARLOC, we get that $\bar{v}_f = \bar{\mu}(l)$ and $\bar{\mu}_f = \bar{\mu}$. Hence, $\bar{v}_f = \mu(l) \sim_{\Gamma(l)} \bar{\mu}(l) = \bar{v}_f$ and $\mu_f \sim_{\Gamma'} \bar{\mu}_f$ (from hypothesis and since $\Gamma' = \Gamma$).

Case: E-EXP. For the transformation derivation, we distinguish two cases:

- $pc \vdash \Gamma\{e_1 \oplus e_2 : \mathtt{Int}^L \rightsquigarrow \bar{e}\}\Gamma_2$

  The transformation derivation is of the form

$$\frac{pc \vdash \Gamma\{e_1 : \mathtt{Int}^L \rightsquigarrow \bar{e}_1\}\Gamma_1 \qquad pc \vdash \Gamma_1\{e_2 : \mathtt{Int}^L \rightsquigarrow \bar{e}_2\}\Gamma_2}{pc \vdash \Gamma\{e_1 \oplus e_2 : \mathtt{Int}^L \rightsquigarrow \bar{e}_1 \oplus \bar{e}_2\}\Gamma_2}\text{ T-EXP-L}$$

From i.h. applied to: (1) $pc \vdash \Gamma\{e_1 : \mathtt{Int}^L \rightsquigarrow \bar{e}_1\}\Gamma_1$, (2) $\langle e_1,\mu\rangle \Downarrow \langle n_1,\mu_1\rangle$, and (3) $\mu \sim_\Gamma \bar{\mu}$, we get that $\langle \bar{e}_1,\bar{\mu}\rangle \Downarrow \langle \bar{v}_1,\bar{\mu}_1\rangle$, with $n_1 \sim_{\mathtt{Int}^L} \bar{v}_1$ and $\mu_1 \sim_{\Gamma_1} \bar{\mu}_1$. From value equivalence relation (Figure 4.7) it follows that $\bar{v}_1 = n_1$.

From i.h. applied to: (4) $pc \vdash \Gamma_1\{e_2 : \mathtt{Int}^L \rightsquigarrow \bar{e}_2\}\Gamma_2$, (5) $\langle e_2,\mu_1\rangle \Downarrow \langle n_2,\mu_2\rangle$, and (6) $\mu_1 \sim_{\Gamma_1} \bar{\mu}_1$, we get that $\langle \bar{e}_2,\bar{\mu}_1\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_2\rangle$, with $n_2 \sim_{\mathtt{Int}^L} \bar{v}_2$ and $\mu_2 \sim_{\Gamma_2} \bar{\mu}_2$. From value equivalence relation (Figure 4.7) it follows that $\bar{v}_2 = n_2$.

By applying rule E-EXP we get that $\bar{v}_f = n_1 \oplus n_2$ and $\bar{\mu}_f = \bar{\mu}_2$:

$$\frac{\langle \bar{e}_1,\bar{\mu}\rangle \Downarrow \langle n_1,\bar{\mu}_1\rangle \qquad \langle \bar{e}_2,\bar{\mu}_1\rangle \Downarrow \langle n_2,\bar{\mu}_2\rangle}{\langle \bar{e}_1 \oplus \bar{e}_2,\bar{\mu}\rangle \Downarrow \langle n_1 \oplus n_2,\bar{\mu}_2\rangle}\text{ E-EXP}$$

Hence, $v_f = n_1 \oplus n_2 \sim_{\mathtt{Int}^L} n_1 \oplus n_2 = \bar{v}_f$ and $\mu_f = \mu_2 \sim_{\Gamma_2} \bar{\mu}_2 = \bar{\mu}_f$ (from i.h.).

- $pc \vdash \Gamma\{e_1 \oplus e_2 : \mathtt{Int}^H \rightsquigarrow \bar{e}\}\Gamma_2$

  The transformation derivation is of the form

$$\text{T-EXP-H}\quad \frac{pc \vdash \Gamma\{e_1 : \mathtt{Int}^H \rightsquigarrow \bar{e}_1\}\Gamma_1 \qquad pc \vdash \Gamma_1\{e_2 : \mathtt{Int}^H \rightsquigarrow \bar{e}_2\}\Gamma_2}{\begin{array}{l} pc \vdash \Gamma\{e_1 \oplus e_2 : \mathtt{Int}^H \rightsquigarrow \\ \qquad \mathtt{let}\ x = \bar{e}_1\ \mathtt{in}\ \mathtt{let}\ y = \bar{e}_2\ \mathtt{in} \\ \qquad\quad \mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ \mathtt{None} \quad \mathtt{Some}\ x'.\ (\mathtt{case}\ y\ \mathtt{of}\ \mathtt{None}.\ \mathtt{None} \quad \mathtt{Some}\ y'.\ \mathtt{Some}\ (x' \oplus y'))\}\Gamma_2 \end{array}}$$

From i.h. applied to: (1) $pc \vdash \Gamma\{e_1 : \mathtt{Int}^H \rightsquigarrow \bar{e}_1\}\Gamma_1$, (2) $\langle e_1,\mu\rangle \Downarrow \langle n_1,\mu_1\rangle$, and (3) $\mu \sim_\Gamma \bar{\mu}$, we get that $\langle \bar{e}_1,\bar{\mu}\rangle \Downarrow \langle \bar{v}_1,\bar{\mu}_1\rangle$, with $n_1 \sim_{\mathtt{Int}^H} \bar{v}_1$ and $\mu_1 \sim_{\Gamma_1} \bar{\mu}_1$. From value equivalence relation (Figure 4.7) it follows that $\bar{v}_1 = \mathtt{Some}\ n_1$.

From i.h. applied to: (4) $pc \vdash \Gamma_1\{e_2 : \mathtt{Int}^H \rightsquigarrow \bar{e}_2\}\Gamma_2$, (5) $\langle e_2,\mu_1\rangle \Downarrow \langle n_2,\mu_2\rangle$, and (6) $\mu_1 \sim_{\Gamma_1} \bar{\mu}_1$, we get that $\langle \bar{e}_2,\bar{\mu}_1\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_2\rangle$, with $n_2 \sim_{\mathtt{Int}^H} \bar{v}_2$ and $\mu_2 \sim_{\Gamma_2} \bar{\mu}_2$. From value equivalence relation (Figure 4.7) it follows that $\bar{v}_2 = \mathtt{Some}\ n_2$.

From the derivation in Figure B.3 we get that $\bar{v}_f = \mathtt{Some}\ (n_1 \oplus n_2)$ and $\bar{\mu}_f = \bar{\mu}_2$. Hence, $v_f = n_1 \oplus n_2 \sim_{\mathtt{Int}^H} \mathtt{Some}\ (n_1 \oplus n_2) = \bar{v}_f$ (from Figure 4.7) and $\mu_f = \mu_2 \sim_{\Gamma_2} \bar{\mu}_2 = \bar{\mu}_f$.

Case: E-ASSIGN. The transformation derivation is of the form

$$\frac{pc \vdash \Gamma\{e : \sigma^\ell \rightsquigarrow \bar{e}\}\Gamma' \qquad pc \sqsubseteq \ell}{pc \vdash \Gamma\{l := e : \mathtt{Unit}^L \rightsquigarrow l := \bar{e}\}\Gamma'[l \mapsto \sigma^\ell]}\text{ T-ASSIGN}$$

From i.h. applied to: (1) $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$, (2) $\langle e,\mu\rangle \Downarrow \langle v,\mu'\rangle$, and (3) $\mu \sim_\Gamma \bar{\mu}$, we get that $\langle \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}'\rangle$, with $v \sim_\tau \bar{v}$ and $\mu \sim_{\Gamma'} \bar{\mu}'$, where $\tau = \sigma^\ell$.

By applying rule E-ASSIGN, we get that $\bar{v}_f = ()$ and $\bar{\mu}_f = \bar{\mu}'[l \mapsto \bar{v}]$:

$$\frac{\langle \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}'\rangle}{\langle l := \bar{e},\bar{\mu}\rangle \Downarrow \langle (),\bar{\mu}'[l \mapsto \bar{v}]\rangle}\text{ E-ASSIGN}$$

$$\dfrac{\langle \bar{e}_1,\bar{\mu}\rangle \Downarrow \langle \texttt{Some } n_1,\bar{\mu}_1\rangle \qquad \dfrac{\dfrac{\langle \bar{e}_2,\bar{\mu}_1\rangle \Downarrow \langle \texttt{Some } n_2,\bar{\mu}_2\rangle}{\langle \bar{e}_2,\bar{\mu}_1 \cup \{x \mapsto \texttt{Some } n_1\}\rangle \Downarrow \langle \texttt{Some } n_2,\bar{\mu}_2 \cup \{x \mapsto \texttt{Some } n_1\}\rangle}\ \textsc{Lem. 33} \qquad \dfrac{\dfrac{x \in dom(\bar{\mu}'_1)}{\langle x,\bar{\mu}'_1\rangle \Downarrow \langle \texttt{Some } n_1,\bar{\mu}'_1\rangle}\ \text{E-VARLOC} \qquad \dfrac{\dfrac{y \in dom(\bar{\mu}'_2)}{\langle y,\mu'_2\rangle \Downarrow \langle \texttt{Some } n_2,\mu'_2\rangle}\ \text{E-VARLOC} \qquad \dfrac{\dfrac{\dfrac{x' \in dom(\mu'_2 \cup \{y' \mapsto n_2\})}{\langle x',\mu'_2 \cup \{y' \mapsto n_2\}\rangle \Downarrow \langle n_1,\mu'_2 \cup \{y' \mapsto n_2\}\rangle}\ \text{E-VARLOC} \qquad \dfrac{y' \in dom(\mu'_2 \cup \{y' \mapsto n_2\})}{\langle y',\mu'_2 \cup \{y' \mapsto n_2\}\rangle \Downarrow \langle n_2,\mu'_2 \cup \{y' \mapsto n_2\}\rangle}\ \text{E-VARLOC}}{\langle x' \oplus y',\mu'_2 \cup \{y' \mapsto n_2\}\rangle \Downarrow \langle n_1 \oplus n_2,\mu'_2 \cup \{y' \mapsto n_2\}\rangle}\ \text{E-EXP}}{\langle \texttt{Some } (x' \oplus y'),\mu'_2 \cup \{y' \mapsto n_2\}\rangle \Downarrow \langle \texttt{Some } (n_1 \oplus n_2),\mu'_2 \cup \{y' \mapsto n_2\}\rangle}\ \text{E-INR}}{\langle \texttt{case } y \texttt{ of None. None} \quad \texttt{Some } y'\texttt{. Some } (x' \oplus y'),\mu'_2\rangle \Downarrow \langle \texttt{Some } (n_1 \oplus n_2),\mu'_2\rangle}\ \text{E-CASE-INR}}{\begin{array}{l}\langle \texttt{case } x \texttt{ of None. None} \\ \qquad \texttt{Some } x'\texttt{. (case } y \texttt{ of None. None} \quad \texttt{Some } y'\texttt{. Some } (x' \oplus y')),\mu'_1\rangle \Downarrow \langle \texttt{Some } (n_1 \oplus n_2),\mu'_1\rangle\end{array}}\ \text{E-CASE-INR}}{\begin{array}{l}\langle \texttt{let } y = \bar{e}_2 \texttt{ in} \\ \quad \texttt{case } x \texttt{ of None. None} \\ \qquad \texttt{Some } x'\texttt{. (case } y \texttt{ of None. None} \quad \texttt{Some } y'\texttt{. Some } (x' \oplus y')),\bar{\mu}_1 \cup \{x \mapsto \texttt{Some } n_1\}\rangle \Downarrow \langle \texttt{Some } (n_1 \oplus n_2),\mu'_1 \setminus \{y\}\rangle\end{array}}\ \text{E-LET}}{\begin{array}{l}\langle \texttt{let } x = \bar{e}_1 \texttt{ in let } y = \bar{e}_2 \texttt{ in} \\ \quad \texttt{case } x \texttt{ of None. None} \\ \qquad \texttt{Some } x'\texttt{. (case } y \texttt{ of None. None} \quad \texttt{Some } y'\texttt{. Some } (x' \oplus y')),\bar{\mu}\rangle \Downarrow \langle \texttt{Some } (n_1 \oplus n_2),\bar{\mu}_2\rangle\end{array}}\ \text{E-LET}$$

where $\mu'_1 = \bar{\mu}_2 \cup \{x \mapsto \texttt{Some } n_1, y \mapsto \texttt{Some } n_2\}$; $\mu'_2 = \mu'_1 \cup \{x' \mapsto n_1\}$.

Figure B.3: Concluding derivation, case E-EXP, Theorem 34.

Hence $v_f = () \sim_{\texttt{Unit}^L} () = \bar{v}_f$ (from Figure 4.7). We are left to show that $\mu_f = \mu'[l \mapsto v] \sim_{\Gamma'[l \mapsto \sigma^\ell]} \bar{\mu}'[l \mapsto \bar{v}] = \bar{\mu}_f$. From i.h. we know that $\mu' \sim_{\Gamma'} \bar{\mu}'$. As $v \sim_\tau \bar{v}$, $\{l \mapsto v\} \sim_{l:\tau} \{l \mapsto \bar{v}\}$ (from Definition 15). Hence $\mu'[l \mapsto v] \sim_{\Gamma'[l \mapsto \tau]} \bar{\mu}'[l \mapsto \bar{v}]$, i.e. $\mu_f \sim_{\Gamma'[l \mapsto \tau]} \bar{\mu}_f$.

Case: E-INL. The transformation derivation is of the form

$$\dfrac{pc \vdash \Gamma\{e : \tau_1 \rightsquigarrow \bar{e}\}\Gamma'}{pc \vdash \Gamma\{\texttt{inl } e : (\tau_1 + \tau_2)^L \rightsquigarrow \texttt{inl } \bar{e}\}\Gamma'}\ \text{T-INL}$$

From i.h. applied to: (1) $pc \vdash \Gamma\{e : \tau_1 \rightsquigarrow \bar{e}\}\Gamma'$, (2) $\langle e,\mu\rangle \Downarrow \langle v,\mu'\rangle$, and (3) $\mu \sim_\Gamma \bar{\mu}$, we get that $\langle \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}'\rangle$, with $v \sim_{\tau_1} \bar{v}$ and $\mu' \sim_{\Gamma'} \bar{\mu}'$.

By applying rule E-INL we get that $\bar{v}_f = \texttt{inl } \bar{v}$ and $\bar{\mu}_f = \bar{\mu}'$:

$$\dfrac{\langle \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}'\rangle}{\langle \texttt{inl } \bar{e},\bar{\mu}\rangle \Downarrow \langle \texttt{inl } \bar{v},\bar{\mu}'\rangle}\ \text{E-INL}$$

Hence $v_f = \texttt{inl } v \sim_{(\tau_1+\tau_2)^L} \texttt{inl } \bar{v} = \bar{v}_f$ (from Figure 4.7) and $\mu_f = \mu' \sim_{\Gamma'} \bar{\mu}' = \bar{\mu}_f$ (from i.h.).

Case: E-INR. Similar to case $\texttt{inl } e$.

Case: E-LET. The transformation derivation is of the form

$$\dfrac{pc \vdash \Gamma\{e_1 : \tau_1 \rightsquigarrow \bar{e}_1\}\Gamma_1 \qquad pc \vdash \Gamma_1, x : \tau_1\{e_2 : \tau_2 \rightsquigarrow \bar{e}_2\}\Gamma_2}{pc \vdash \Gamma\{\texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2 \rightsquigarrow \texttt{let } x = \bar{e}_1 \texttt{ in } \bar{e}_2\}\Gamma_2 \setminus \{x\}}\ \text{T-LET}$$

From i.h. applied to: (1) $pc \vdash \Gamma\{e_1 : \tau_1 \rightsquigarrow \bar{e}_1\}\Gamma_1$, (2) $\langle e_1, \mu\rangle \Downarrow \langle v_1, \mu_1\rangle$, and (3) $\mu \sim_\Gamma \bar{\mu}$, we get that $\langle \bar{e}_1, \bar{\mu}\rangle \Downarrow \langle \bar{v}_1, \bar{\mu}_1\rangle$, with $v_1 \sim_{\tau_1} \bar{v}_1$ and $\mu_1 \sim_{\Gamma_1} \bar{\mu}_1$.

We are to show that $\mu_1 \cup \{x \mapsto v_1\} \sim_{\Gamma_1, x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$. Since $v_1 \sim_{\tau_1} \bar{v}_1$, $\{x \mapsto v_1\} \sim_{x:\tau_1} \{x \mapsto \bar{v}_1\}$ (Definition 15). Hence $\mu_1 \cup \{x \mapsto v_1\} \sim_{\Gamma_1, x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$ (since $\mu_1 \sim_{\Gamma_1} \bar{\mu}_1$).

From i.h. applied to (4) $pc \vdash \Gamma_1, x : \tau_1\{e_2 : \tau_2 \rightsquigarrow \bar{e}_2\}\Gamma_2$, (5) $\langle e_2, \mu_1 \cup \{x \mapsto v_1\}\rangle \Downarrow \langle v_2, \mu_2\rangle$, and (6) $\mu_1 \cup \{x \mapsto v_1\} \sim_{\Gamma_1, x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$, we get that $\langle \bar{e}_2, \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2, \bar{\mu}_2\rangle$, with $v_2 \sim_{\tau_2} \bar{v}_2$ and $\mu_2 \sim_{\Gamma_2} \bar{\mu}_2$.

By applying rule E-LET, we get that $\bar{v}_f = \bar{v}_2$ and $\bar{\mu}_f = \bar{\mu}_2 \setminus \{x\}$:

$$\dfrac{\langle \bar{e}_1, \bar{\mu}\rangle \Downarrow \langle \bar{v}_1, \bar{\mu}_1\rangle \qquad \langle \bar{e}_2, \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2, \bar{\mu}_2\rangle}{\langle \mathtt{let}\ x = \bar{e}_1\ \mathtt{in}\ \bar{e}_2, \bar{\mu}\rangle \Downarrow \langle \bar{v}_2, \bar{\mu}_2 \setminus \{x\}\rangle}\ \text{E-LET}$$

Hence $v_f = v_2 \sim_{\tau_2} \bar{v}_2 = \bar{v}_f$ (from i.h.) and $\mu_f = \mu_2 \setminus \{x\} \sim_{\Gamma_2 \setminus \{x\}} \bar{\mu}_2 \setminus \{x\} = \bar{\mu}_f$.

Case: E-CASE-INL. For the transformation derivation we distinguish two cases:

- $pc \vdash \Gamma\{e : (\tau_1 + \tau_2)^L \rightsquigarrow \bar{e}\}\Gamma'$

  The transformation derivation is of the form

$$\text{T-CASE-L}\ \dfrac{pc \vdash \Gamma\{e : (\tau_1 + \tau_2)^L \rightsquigarrow \bar{e}\}\Gamma' \qquad pc \vdash \Gamma', x : \tau_i\{e_i : \tau \rightsquigarrow \bar{e}_i\}\Gamma_i \qquad i = 1,2 \qquad S_1 = \Gamma_1 \setminus \Gamma_2 \qquad S_2 = \Gamma_2 \setminus \Gamma_1}{\begin{array}{l} pc \vdash \Gamma\{\mathtt{case}\ e\ \mathtt{of}\ \mathtt{inl}\ x.\ e_1 \quad \mathtt{inr}\ x.\ e_2 : \tau \rightsquigarrow \\ \qquad \mathtt{case}\ \bar{e}\ \mathtt{of}\ \mathtt{inl}\ x.\ \mathtt{let}\ y = \bar{e}_1\ \mathtt{in}\ \mathsf{upgrade}(S_2); y \quad \mathtt{inr}\ x.\ \mathtt{let}\ y = \bar{e}_2\ \mathtt{in}\ \mathsf{upgrade}(S_1); y\}\bigsqcup \Gamma_i \setminus \{x\} \end{array}}$$

We assume e evaluates under rule E-CASE-INL. For the case when it evaluates under rule E-CASE-INR, the proof is analogous.

From i.h. applied to: (1) $pc \vdash \Gamma\{e : (\tau_1 + \tau_2)^L \rightsquigarrow \bar{e}\}\Gamma'$, (2) $\langle e, \mu\rangle \Downarrow \langle \mathtt{inl}\ v_1, \mu_1\rangle$, and (3) $\mu \sim_\Gamma \bar{\mu}$, we get that $\langle \bar{e}, \bar{\mu}\rangle \Downarrow \langle \bar{v}, \bar{\mu}_1\rangle$, with $\mathtt{inl}\ v_1 \sim_{(\tau_1 + \tau_2)^L} \bar{v}$ and $\mu_1 \sim_{\Gamma'} \bar{\mu}_1$. From value equivalence relation (Figure 4.7), $\bar{v} = \mathtt{inl}\ \bar{v}_1$, with $v_1 \sim_{\tau_1} \bar{v}_1$. Hence $\{x \mapsto v_1\} \sim_{x:\tau_1} \{x \mapsto \bar{v}_1\}$ (Definition 15). Thus $\mu_1 \cup \{x \mapsto v_1\} \sim_{\Gamma', x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$.

From i.h. applied to: (4) $pc \vdash \Gamma', x : \tau_1\{e_1 : \tau \rightsquigarrow \bar{e}_1\}\Gamma_1$, (5) $\langle e_1, \mu_1 \cup \{x \mapsto v_1\}\rangle \Downarrow \langle v_2, \mu_2\rangle$, and (6) $\mu_1 \cup \{x \mapsto v_1\} \sim_{\Gamma', x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$, we get that $\langle \bar{e}_1, \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2, \bar{\mu}_2\rangle$, with $v_2 \sim_\tau \bar{v}_2$ and $\mu_2 \sim_{\Gamma_1} \bar{\mu}_2$.

By applying rule E-CASE-INL, we get that $\bar{v}_f = \bar{v}_2$ and $\bar{\mu}_f = \bar{\mu}'_2 \setminus \{x, y\}$:

$$\dfrac{\langle \bar{e}, \bar{\mu}\rangle \Downarrow \langle \mathtt{inl}\ \bar{v}_1, \bar{\mu}_1\rangle \qquad \dfrac{\begin{array}{c} \langle \bar{e}_1, \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2, \bar{\mu}_2\rangle \\ \dfrac{\begin{array}{c} \dfrac{}{\langle \mathsf{upgrade}(S_2), \bar{\mu}_2 \cup \{y \mapsto \bar{v}_2\}\rangle \Downarrow \langle (), \bar{\mu}'_2\rangle}\ \text{E-ASSIGN}^* \\ \dfrac{y \in dom(\bar{\mu}'_2)}{\langle y, \bar{\mu}'_2\rangle \Downarrow \langle \bar{v}_2, \bar{\mu}'_2\rangle}\ \text{E-VARLOC} \end{array}}{\langle \mathsf{upgrade}(S_2); y, \bar{\mu}_2 \cup \{y \mapsto \bar{v}_2\}\rangle \Downarrow \langle \bar{v}_2, \bar{\mu}'_2\rangle}\ \text{E-SEQ} \end{array}}{\langle \mathtt{let}\ y = \bar{e}_1\ \mathtt{in}\ \mathsf{upgrade}(S_2); y, \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2, \bar{\mu}'_2 \setminus \{y\}\rangle}\ \text{E-LET}}{\langle \mathtt{case}\ \bar{e}\ \mathtt{of}\ \mathtt{inl}\ x.\ \mathtt{let}\ y = \bar{e}_1\ \mathtt{in}\ \mathsf{upgrade}(S_2); y \quad \mathtt{inr}\ x.\ \mathtt{let}\ y = \bar{e}_2\ \mathtt{in}\ \mathsf{upgrade}(S_1); y, \bar{\mu}\rangle \Downarrow \langle \bar{v}_2, \bar{\mu}'_2 \setminus \{x, y\}\rangle}\ \text{E-CASE-INL}$$

Thus $v_f = v_2 \sim_\tau \bar{v}_2 = \bar{v}_f$. We are left to show that $\mu_f = \mu_2 \setminus \{x\} \sim_{\sqcup \Gamma_i \setminus \{x\}} \bar{\mu}'_2 \setminus \{y, x\} = \bar{\mu}_f$.

From i.h., $\mu_2 \sim_{\Gamma_1} \bar{\mu}_2$, i.e. for all l, $\mu_2(l) \sim_{\Gamma_1(l)} \bar{\mu}_2(l)$ (Definition 15). S_2 is the set of memory locations that get upgraded in the `inr` branch ($\bar{e}_2$), but not in the `inl` branch ($\bar{e}_1$). I.e. for all

$$\frac{}{\langle \bar{e},\bar{\mu}\rangle \Downarrow \langle \texttt{Some inl } \bar{v}_1,\bar{\mu}_1\rangle}\ \text{I.H.}$$

$$\frac{}{\langle x,\bar{\mu}_1 \cup \{x' \mapsto \texttt{inl } \bar{v}_1\}\rangle \Downarrow \langle \texttt{inl } \bar{v}_1,\bar{\mu}_1 \cup \{x' \mapsto \texttt{inl } \bar{v}_1\}\rangle}\ \text{E-VARLOC}$$

$$\frac{\dfrac{}{\langle \bar{e}_1,\bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_2\rangle}\ \text{I.H.}}{\langle \bar{e}_1,\bar{\mu}_1 \cup \{x' \mapsto \texttt{inl } \bar{v}_1, x \mapsto \bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_2 \cup \{x' \mapsto \texttt{inl } \bar{v}_1\}\rangle}\ \textsc{Lem. 33}$$

$$\begin{array}{ll}
\hline
\langle \mathsf{upgrade}(S_2),\bar{\mu}_2 \cup \{x' \mapsto \texttt{inl } \bar{v}_1, y \mapsto \bar{v}_2\}\rangle \Downarrow \langle (),\bar{\mu}_2' \cup \{x' \mapsto \texttt{inl } \bar{v}_1, y \mapsto \bar{v}_2\}\rangle & \text{E-ASSIGN}^*\\
\qquad y \in dom(\bar{\mu}_2' \cup \{x' \mapsto \texttt{inl } \bar{v}_1, y \mapsto \bar{v}_2\}) & \\
\hline
\langle y,\bar{\mu}_2' \cup \{x' \mapsto \texttt{inl } \bar{v}_1, y \mapsto \bar{v}_2\}\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_2' \cup \{x' \mapsto \texttt{inl } \bar{v}_1, y \mapsto \bar{v}_2\}\rangle & \text{E-VARLOC}\\
\hline
\langle \mathsf{upgrade}(S_2); y,\bar{\mu}_2 \cup \{x' \mapsto \texttt{inl } \bar{v}_1, y \mapsto \bar{v}_2\}\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_2' \cup \{x' \mapsto \texttt{inl } \bar{v}_1, y \mapsto \bar{v}_2\}\rangle & \text{E-SEQ}\\
\hline
\langle \texttt{let } y = \bar{e}_1 \texttt{ in } \mathsf{upgrade}(S_2); y,\bar{\mu}_1 \cup \{x' \mapsto \texttt{inl } \bar{v}_1, x \mapsto \bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_2' \cup \{x' \mapsto \texttt{inl } \bar{v}_1\}\rangle & \text{E-LET}\\
\hline
\langle \texttt{case } x' \texttt{ of inl } x.\ \texttt{let } y = \bar{e}_1 \texttt{ in } \mathsf{upgrade}(S_2); y & \text{E-CASE-INL}\\
\qquad \texttt{inr } x.\ \texttt{let } y = \bar{e}_2 \texttt{ in } \mathsf{upgrade}(S_1); y,\bar{\mu}_1 \cup \{x' \mapsto \texttt{inl } \bar{v}_1\}\rangle \langle \bar{v}_2,\bar{\mu}_2' \cup \{x' \mapsto \texttt{inl } \bar{v}_1\} \setminus \{x_1\}\rangle & \\
\hline
\langle \texttt{case } \bar{e} \texttt{ of None}.\ (\forall l \in \mathsf{updated}(\bar{e}_1,\bar{e}_2).\ l := \texttt{None});\ \texttt{None} & \text{E-CASE-INR}\\
\qquad \texttt{Some } x'.\ \texttt{case } x' \texttt{ of inl } x.\ \texttt{let } y = \bar{e}_1 \texttt{ in } \mathsf{upgrade}(S_2); y \quad \texttt{inr } x.\ \texttt{let } y = \bar{e}_2 \texttt{ in } \mathsf{upgrade}(S_1); y,\bar{\mu}\rangle \langle \bar{v}_2,\bar{\mu}_2' \setminus \{x\}\rangle &
\end{array}$$

Figure B.4: Concluding derivation, case E-CASE-INL, Theorem 34.

$l \in S_2$, $(\Gamma_2 \setminus \Gamma_1)(l) = \sigma^H$ $((\bigsqcup \Gamma_i \setminus \{x\})(l) = \sigma^H)$. Or, put differently, for all $l \in S_2$, $(l, \texttt{Some } \bar{\mu}_2(l)) \in \bar{\mu}_2'$. As $\mu_2(l) \sim_{\sigma^L} \bar{\mu}_2(l)$, it follows that $\mu_2(l) \sim_{\sigma^H} \texttt{Some } \bar{\mu}_2(l)$.

For the other memory locations l, i.e. for all $l \in dom(\mu_2) \setminus S_2$, $(\bigsqcup \Gamma_i \setminus \{x\})(l) = (\Gamma_1 \setminus \{x\})(l)$ and $\mu_2(l) \sim_{\Gamma_1(l)} \bar{\mu}_2(l) = (\bar{\mu}_2' \setminus \{y\})(l)$. I.e. $(\mu_2 \setminus \{x\})(l) \sim_{\Gamma_1(l)} (\bar{\mu}_2' \setminus \{y,x\})(l)$. Hence $\mu_2 \setminus \{x\} \sim_{\sqcup \Gamma_i \setminus \{x\}} \bar{\mu}_2' \setminus \{y,x\}$.

- $pc \vdash \Gamma\{e : (\tau_1 + \tau_2)^H \rightsquigarrow \bar{e}\}\Gamma'$

The transformation derivation is of the form

T-CASE-H

$$\frac{pc \vdash \Gamma\{e : (\tau_1 + \tau_2)^H \rightsquigarrow \bar{e}\}\Gamma' \qquad H \vdash \Gamma', x : \tau_i\{e_i : \tau \rightsquigarrow \bar{e}_i\}\Gamma_i \qquad i = 1,2 \qquad S_1 = \Gamma_1 \setminus \Gamma_2 \qquad S_2 = \Gamma_2 \setminus \Gamma_1}{\begin{array}{l} pc \vdash \Gamma\{\texttt{case } e \texttt{ of inl } x.\ e_1 \quad \texttt{inr } x.\ e_2 : \tau \rightsquigarrow \\ \qquad \texttt{case } \bar{e} \texttt{ of None}.\ (\forall l \in \mathsf{updated}(\bar{e}_1,\bar{e}_2).\ l := \texttt{None});\ \texttt{None} \\ \qquad\qquad \texttt{Some } x'.\ \texttt{case } x' \texttt{ of inl } x.\ \texttt{let } y = \bar{e}_1 \texttt{ in } \mathsf{upgrade}(S_2); y \\ \qquad\qquad\qquad \texttt{inr } x.\ \texttt{let } y = \bar{e}_2 \texttt{ in } \mathsf{upgrade}(S_1); y\} \bigsqcup \Gamma_i \setminus \{x\} \end{array}}$$

From i.h. applied to: (1) $pc \vdash \Gamma\{e : (\tau_1 + \tau_2)^H \rightsquigarrow \bar{e}\}\Gamma'$, (2) $\langle e,\mu\rangle \Downarrow \langle \texttt{inl } v_1,\mu_1\rangle$, and (3) $\mu \sim_\Gamma \bar{\mu}$, and we get $\langle \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}_1\rangle$, with $\texttt{inl } v_1 \sim_{(\tau_1+\tau_2)^H} \bar{v}$ and $\mu_1 \sim_{\Gamma'} \bar{\mu}_1$. From value equivalence relation (Figure 4.7), $\bar{v} = \texttt{Some inl } \bar{v}_1$, with $v_1 \sim_{\tau_1} \bar{v}_1$. Hence $\{x \mapsto v_1\} \sim_{x:\tau_1} \{x \mapsto \bar{v}_1\}$ (Definition 15). Thus $\mu_1 \cup \{x \mapsto v_1\} \sim_{\Gamma',x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$.

From i.h. applied to: (4) $pc \vdash \Gamma', x : \tau_1\{e_1 : \tau \rightsquigarrow \bar{e}_1\}\Gamma_1$, (5) $\langle e_1,\mu_1 \cup \{x \mapsto v_1\}\rangle \Downarrow \langle v_2,\mu_2\rangle$, and (6) $\mu_1 \cup \{x \mapsto v_1\} \sim_{\Gamma',x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$, we get $\langle \bar{e}_1,\bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_2\rangle$, with $v_2 \sim_\tau \bar{v}_2$ and $\mu_2 \sim_{\Gamma_1} \bar{\mu}_2$.

From the derivation in Figure B.4, $\bar{v}_f = \bar{v}_2$ and $\bar{\mu}_f = \bar{\mu}_2' \setminus \{x\}$. Hence, $v_f \sim_\tau \bar{v}_f$ (from i.h.). We are left to show that $\mu_f = \mu_2 \setminus \{x\} \sim_{\sqcup\Gamma_i \setminus \{x\}} \bar{\mu}_2' \setminus \{x\} = \bar{\mu}_f$. The argument is analogous to the previous case. Hence $\mu_f \sim_{\sqcup\Gamma_i \setminus \{x\}} \bar{\mu}_f$.

Case: E-CASE-INR.

Similar to case E-CASE-INL.

Case: E-WHILE-TRUE.

The last step of the derivation has the form:

$$\frac{\mu(x) \neq 0 \qquad \langle e,\mu\rangle \Downarrow \langle v,\mu'\rangle \qquad \langle \mathtt{while}\ x\ \mathtt{do}\ e,\mu'\rangle \Downarrow \langle v',\mu''\rangle}{\langle \mathtt{while}\ x\ \mathtt{do}\ e,\mu\rangle \Downarrow \langle v',\mu''\rangle}\ \text{E-WHILE-TRUE}$$

For the transformation derivation we distinguish two cases:

- $\Gamma(x) = \mathtt{Int}^L$

  The transformation derivation is of the form

$$\frac{\Gamma(x) = \mathtt{Int}^L \qquad pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma}{pc \vdash \Gamma\{\mathtt{while}\ x\ \mathtt{do}\ e : \mathtt{Unit}^L \rightsquigarrow \mathtt{while}\ x\ \mathtt{do}\ \bar{e}\}\Gamma}\ \text{T-WHILE-L}$$

We have to show that if $\langle \mathtt{while}\ x\ \mathtt{do}\ \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v}',\bar{\mu}''\rangle$, then $v' \sim_{\mathtt{Unit}^L} \bar{v}'$ and $\mu'' \sim_\Gamma \bar{\mu}''$. As $\mu \sim_\Gamma \bar{\mu}$ and $\Gamma(x) = \mathtt{Int}^L$, it must be the case that $\bar{\mu}(x) = \mu(x)$. Hence $\bar{\mu}(x) \neq 0$. Thus the target program evaluates under rule E-WHILE-TRUE as well.

From i.h. applied to: (1) $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma$, (2) $\langle e,\mu\rangle \Downarrow \langle v,\mu'\rangle$, and (3) $\mu \sim_\Gamma \bar{\mu}$, we get that $\langle \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}'\rangle$, with $v' \sim_\tau \bar{v}$ and $\mu' \sim_\Gamma \bar{\mu}'$.

From i.h. applied to: (4) $pc \vdash \Gamma\{\mathtt{while}\ x\ \mathtt{do}\ e : \mathtt{Unit}^L \rightsquigarrow \mathtt{while}\ x\ \mathtt{do}\ \bar{e}\}\Gamma$, (5) $\langle \mathtt{while}\ x\ \mathtt{do}\ e,\mu'\rangle \Downarrow \langle v',\mu''\rangle$, and (6) $\mu' \sim_\Gamma \bar{\mu}'$, we get that $\langle \mathtt{while}\ x\ \mathtt{do}\ \bar{e},\bar{\mu}'\rangle \Downarrow \langle \bar{v}',\bar{\mu}''\rangle$, with $v' \sim_{\mathtt{Unit}^L} \bar{v}'$ and $\mu'' \sim_\Gamma \bar{\mu}''$.

By applying rule E-WHILE-TRUE, we get that $\bar{v}_f = \bar{v}'$ and $\bar{\mu}_f = \bar{\mu}''$:

$$\frac{\bar{\mu}(x) \neq 0 \qquad \langle e,\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}'\rangle \qquad \langle \mathtt{while}\ x\ \mathtt{do}\ \bar{e},\bar{\mu}'\rangle \Downarrow \langle \bar{v}',\bar{\mu}''\rangle}{\langle \mathtt{while}\ x\ \mathtt{do}\ \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v}',\bar{\mu}''\rangle}\ \text{E-WHILE-TRUE}$$

Hence $v_f = v' \sim_{\mathtt{Unit}^L} \bar{v}' = \bar{v}_f$ and $\mu_f = \mu'' \sim_\Gamma \bar{\mu}'' = \bar{\mu}_f$.

- $\Gamma(x) = \mathtt{Int}^H$

  The transformation derivation is of the form

$$\frac{\Gamma(x) = \mathtt{Int}^H \qquad H \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma}{\begin{array}{l} pc \vdash \Gamma\{\mathtt{while}\ x\ \mathtt{do}\ e : \mathtt{Unit}^L \rightsquigarrow \\ \qquad \mathtt{while}\ (\mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \mathtt{None};\ 0 \quad \mathtt{Some}\ y.\ y)\ \mathtt{do}\ \bar{e}\}\Gamma \end{array}}\ \text{T-WHILE-H}$$

We have to show that if

$$\langle \mathtt{while}\ (\mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \mathtt{None};\ 0 \quad \mathtt{Some}\ y.\ y)\ \mathtt{do}\ \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v}',\bar{\mu}''\rangle,$$

then $v' \sim_{\mathtt{Unit}^L} \bar{v}'$ and $\mu'' \sim_\Gamma \bar{\mu}''$.

As $\mu \sim_\Gamma \bar{\mu}$ and $\Gamma(x) = \mathtt{Int}^H$, it must be the case that $\bar{\mu}(x) = \mathtt{Some}\ \mu(x)$, with $\mu(x) \neq 0$.

From i.h. applied to: (1) $H \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma$, (2) $\langle e,\mu\rangle \Downarrow \langle v,\mu'\rangle$, and (3) $\mu \sim_\Gamma \bar{\mu}$, we get that $\langle \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}'\rangle$, with $v \sim_\tau \bar{v}$ and $\mu' \sim_\Gamma \bar{\mu}'$.

From i.h. applied to: (4) $pc \vdash \Gamma\{\mathtt{while}\ x\ \mathtt{do}\ e : \mathtt{Unit}^L \rightsquigarrow \mathtt{while}\ (\mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \mathtt{None};\ 0 \quad \mathtt{Some}\ y.\ y)\ \mathtt{do}\ \bar{e}\}\Gamma$, (5) $\langle \mathtt{while}\ x\ \mathtt{do}\ e,\mu'\rangle \Downarrow \langle v',\mu''\rangle$ and (6) $\mu' \sim_\Gamma \bar{\mu}'$, we get that

$\langle \texttt{while}\ (\texttt{case}\ x\ \texttt{of None}.\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None};\ 0 \quad \texttt{Some}\ y.\ y)\ \texttt{do}\ \bar{e}, \bar{\mu}'\rangle \Downarrow \langle \bar{v}', \bar{\mu}''\rangle$, with $v' \sim_{\texttt{Unit}^L} \bar{v}'$ and $\mu'' \sim_\Gamma \bar{\mu}''$.

From the derivation below it follows that $\bar{v}_f = \bar{v}'$ and $\bar{\mu}_f = \bar{\mu}''$:

$$\dfrac{\begin{array}{c}\dfrac{\bar{\mu}(x) = \texttt{Some}\ (\mu(x)) \qquad \dfrac{}{\langle y, \bar{\mu} \cup \{y \mapsto \mu(x)\}\rangle \Downarrow \langle \mu(x), \bar{\mu} \cup \{y \mapsto \mu(x)\}\rangle}\ \text{E-VAR}}{\langle \texttt{case}\ x\ \texttt{of None}.\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None};\ 0 \quad \texttt{Some}\ y.\ y, \bar{\mu}\rangle \Downarrow \langle \mu(x), \bar{\mu}\rangle}\ \text{E-CASE-INR} \\ \langle \bar{e}, \bar{\mu}\rangle \Downarrow \langle \bar{v}, \bar{\mu}'\rangle \\ \langle \texttt{while}\ (\texttt{case}\ x\ \texttt{of None}.\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None};0 \quad \texttt{Some}\ y.\ y)\ \texttt{do}\ \bar{e}, \bar{\mu}'\rangle \Downarrow \langle \bar{v}', \bar{\mu}''\rangle \end{array}}{\langle \texttt{while}\ (\texttt{case}\ x\ \texttt{of None}.\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None};0 \quad \texttt{Some}\ y.\ y)\ \texttt{do}\ \bar{e}, \bar{\mu}\rangle \Downarrow \langle \bar{v}', \bar{\mu}''\rangle}\ \text{E-WHILE-TRUE}$$

Hence $v_f = v' \sim_{\texttt{Unit}^L} \bar{v}' = \bar{v}_f$ and $\mu_f = \mu'' \sim_\Gamma \bar{\mu}'' = \bar{\mu}_f$.

Case: E-WHILE-FALSE.

The last step of the derivation has the form:

$$\dfrac{\mu(x) = 0}{\langle \texttt{while}\ x\ \texttt{do}\ e, \mu\rangle \Downarrow \langle (), \mu\rangle}\ \text{E-WHILE-FALSE}$$

For the transformation derivation we distinguish two cases:

- $\Gamma(x) = \texttt{Int}^L$

  The transformation derivation is of the form

$$\dfrac{\Gamma(x) = \texttt{Int}^L \qquad pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma}{pc \vdash \Gamma\{\texttt{while}\ x\ \texttt{do}\ e : \texttt{Unit}^L \rightsquigarrow \texttt{while}\ x\ \texttt{do}\ \bar{e}\}\Gamma}\ \text{T-WHILE-L}$$

As $\mu \sim_\Gamma \bar{\mu}$ and $\Gamma(x) = \texttt{Int}^L$, it must be the case that $\bar{\mu}(x) = \mu(x)$. Hence $\bar{\mu}(x) = 0$. Thus the target program evaluates under rule E-WHILE-FALSE as well. Thus $v_f = () \sim_{\texttt{Unit}^L} () = \bar{v}_f$ and $\mu_f = \mu \sim_\Gamma \bar{\mu} = \bar{\mu}_f$.

- $\Gamma(x) = \texttt{Int}^H$

  The transformation derivation is of the form

$$\dfrac{\Gamma(x) = \texttt{Int}^H \qquad H \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma}{\begin{array}{l} pc \vdash \Gamma\{\texttt{while}\ x\ \texttt{do}\ e : \texttt{Unit}^L \rightsquigarrow \\ \qquad \texttt{while}\ (\texttt{case}\ x\ \texttt{of None}.\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None};\ 0 \quad \texttt{Some}\ y.\ y)\ \texttt{do}\ \bar{e}\}\Gamma \end{array}}\ \text{T-WHILE-H}$$

As $\mu \sim_\Gamma \bar{\mu}$ and $\Gamma(x) = \texttt{Int}^H$, it must be the case that $\bar{\mu}(x) = \texttt{Some}\ \mu(x)$, with $\mu(x) = 0$. Thus the target program evaluates under rule E-WHILE-FALSE as well:

$$\dfrac{\dfrac{\bar{\mu}(x) = \texttt{Some}\ 0 \qquad \dfrac{}{\langle y, \bar{\mu} \cup \{y \mapsto 0\}\rangle \Downarrow \langle 0, \bar{\mu} \cup \{y \mapsto 0\}\rangle}\ \text{E-VAR}}{\langle \texttt{case}\ x\ \texttt{of None}.\ \forall l \in \mathsf{updated}(\bar{e}_2).\ l := \texttt{None};\ 0 \quad \texttt{Some}\ y.\ y, \bar{\mu}\rangle \Downarrow \langle 0, \bar{\mu}\rangle}\ \text{E-CASE-INR}}{\langle \texttt{while}\ (\texttt{case}\ x\ \texttt{of None}.\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None};\ 0 \quad \texttt{Some}\ y.\ y)\ \texttt{do}\ \bar{e}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}\rangle}\ \text{E-WHILE-FALSE}$$

Hence $v_f = () \sim_{\texttt{Unit}^L} () = \bar{v}_f$ and $\mu_f = \mu \sim_\Gamma \bar{\mu} = \bar{\mu}_f$.

Case: E-UPGRADE.

The transformation derivation is of the form

$$\frac{pc \vdash \Gamma\{e : \sigma^{L} \rightsquigarrow \bar{e}\}\Gamma'}{pc \vdash \Gamma\{\bullet e : \sigma^{H} \rightsquigarrow \texttt{Some}\ \bar{e}\}\Gamma'}\ \text{T-UPGRADE}$$

From i.h. applied to: (1) $pc \vdash \Gamma\{e : \texttt{Int}^{L} \rightsquigarrow \bar{e}\}\Gamma'$, (2) $\langle e, \mu\rangle \Downarrow \langle v, \mu'\rangle$, and (3) $\mu \sim_{\Gamma} \bar{\mu}$, we get that $\langle \bar{e}, \bar{\mu}\rangle \Downarrow \langle \bar{v}, \bar{\mu}'\rangle$, with $v \sim_{\sigma^{L}} \bar{v}$ and $\mu' \sim_{\Gamma'} \bar{\mu}'$.

By applying rule E-INR, we get that $\bar{v}_f = \texttt{Some}\ \bar{v}$ and $\bar{\mu}_f = \bar{\mu}'$:

$$\frac{\langle \bar{e}, \bar{\mu}\rangle \Downarrow \langle \bar{v}, \bar{\mu}'\rangle}{\langle \texttt{Some}\ \bar{e}, \bar{\mu}\rangle \Downarrow \langle \texttt{Some}\ \bar{v}, \bar{\mu}'\rangle}\ \text{E-INR}$$

Hence $v_f = v \sim_{\sigma^{H}} \texttt{Some}\ \bar{v} = \bar{v}_f$ (from Figure 4.7) and $\mu_f = \mu' \sim_{\Gamma'} \bar{\mu}' = \bar{\mu}_f$. ■

Transformation monotonicity

Theorem 35 (Source program more precise). *For any source program e, typing environment Γ, program context pc, and stores μ and $\bar{\mu}$ such that $\mu \geq_{\Gamma} \bar{\mu}$, if $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$ and $\langle e, \mu\rangle \Downarrow \langle v_f, \mu_f\rangle$, then there exists $\bar{v}_f$ and $\bar{\mu}_f$, such that $\langle \bar{e}, \bar{\mu}\rangle \Downarrow \langle \bar{v}_f, \bar{\mu}_f\rangle$ and $v_f \geq_{\tau} \bar{v}_f$ and $\mu_f \geq_{\Gamma'} \bar{\mu}_f$.*

The proof of this theorem relies on the following helper lemma.

Lemma 36 (Helper). *For any source program e, typing environment Γ, and stores μ and $\bar{\mu}$ such that $\mu \geq_{\Gamma} \bar{\mu}$, if $H \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$ and $\langle e, \mu\rangle \Downarrow \langle v_f, \mu_f\rangle$, then $\langle \forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}_f\rangle$ and $\mu_f \geq_{\Gamma'} \bar{\mu}_f$.*

Proof. By induction on the structure of e.

Case: $e = v$**.** From rule T-VAL, $\bar{e} = v$. Then $\mathsf{updated}(v) = \emptyset$. From E-VAL, $\mu_f = \mu$. As $\bar{\mu}_f \geq_{\Gamma} \bar{\mu}$, it follows that $\mu_f \geq_{\Gamma} \bar{\mu}_f$.

Case: $e = l$**.** From rule T-VARLOC, $\bar{e} = l$. Then $\mathsf{updated}(l) = \emptyset$. From E-VARLOC, $\mu_f = \mu$. As $\bar{\mu}_f \geq_{\Gamma} \bar{\mu}$, it follows that $\mu_f \geq_{\Gamma} \bar{\mu}_f$.

Case: $e = e_1 \oplus e_2$**.** From rule T-EXP-*, $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}_1, \bar{e}_2)$.

From i.h. applied to (1) $\langle e_1, \mu\rangle \Downarrow \langle n_1, \mu_1\rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar{e}_1).\ l := \texttt{None}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}_1\rangle$, and (3) $\mu \geq_{\Gamma} \bar{\mu}$, we get $\mu_1 \geq_{\Gamma_1} \bar{\mu}_1$.

From i.h. applied to (4) $\langle e_2, \mu_1\rangle \Downarrow \langle n_2, \mu_2\rangle$, (5) $\langle \forall l \in \mathsf{updated}(\bar{e}_2).\ l := \texttt{None}, \bar{\mu}_1\rangle \Downarrow \langle (), \bar{\mu}_2\rangle$, and (6) $\mu_1 \geq_{\Gamma_1} \bar{\mu}_1$, we get $\mu_2 \geq_{\Gamma_2} \bar{\mu}_2$.

Thus $\langle \forall l \in \mathsf{updated}(\bar{e}_1, \bar{e}_2).\ l := \texttt{None}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}_2\rangle$, with $\bar{\mu}_f = \bar{\mu}_2$. As $\mu_f = \mu_2$, it follows that $\mu_f \geq_{\Gamma_2} \bar{\mu}_f$.

Case: $e = l := e'$**.** From rule T-ASSIGN, $\bar{e} = l := \bar{e}'$. Then $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}', l)$.

From i.h. applied to (1) $\langle e', \mu\rangle \Downarrow \langle v, \mu'\rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar{e}').\ l := \texttt{None}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}'\rangle$, and (3) $\mu \geq_{\Gamma} \bar{\mu}$, we get $\mu' \geq_{\Gamma'} \bar{\mu}'$. I.e. $\langle \forall l' \in \mathsf{updated}(\bar{e}', l).\ l' := \texttt{None}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}'[l \mapsto \texttt{None}]\rangle$.

As $v \geq_{\sigma^{H}} \texttt{None}$, it follows that $(\mu'[l \mapsto v])(l) \geq_{\sigma^{H}} (\bar{\mu}'[l \mapsto \texttt{None}])(l)$. As $\mu_f = \mu'[l \mapsto v]$ and $\bar{\mu}_f = \bar{\mu}'[l \mapsto \texttt{None}]$, it follows that $\mu_f \geq_{\Gamma'[l \mapsto \sigma^{H}]} \bar{\mu}_f$.

Case: $e = \texttt{inl}\ e'$**.** From rule T-INL, $\bar{e} = \texttt{inl}\ \bar{e}'$. Then $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}')$.

From i.h. applied to (1) $\langle e', \mu\rangle \Downarrow \langle v, \mu'\rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar{e}').\ l := \texttt{None}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}'\rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get $\mu' \geq_{\Gamma'} \bar{\mu}'$. Thus $\langle \forall l \in \mathsf{updated}(\texttt{inl}\ \bar{e}').\ l := \texttt{None}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}'\rangle$. As $\mu_f = \mu'$ and $\bar{\mu}_f = \bar{\mu}'$, it follows that $\mu_f \geq_{\Gamma'} \bar{\mu}_f$.

Case: $e = \texttt{inr}\ e'$**.** Similar to the previous case.

Case: $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$**.** From rule T-LET, $\bar{e} = \texttt{let}\ x = \bar{e}_1\ \texttt{in}\ \bar{e}_2$. Then $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}_1, \bar{e}_2)$.

From i.h. applied to (1) $\langle e_1, \mu\rangle \Downarrow \langle v_1, \mu_1\rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar{e}_1).\ l := \texttt{None}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}_1\rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get $\mu_1 \geq_{\Gamma_1} \bar{\mu}_1$.

Let $\bar{v}_1$ be such that $v_1 \geq_{\tau_1} \bar{v}_1$. Hence, $\mu_1 \cup \{x \mapsto v_1\} \geq_{\Gamma_1, x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$.

From i.h. applied to (4) $\langle e_2, \mu_1 \cup \{x \mapsto v_1\}\rangle \Downarrow \langle v_2, \mu_2\rangle$, (5) $\langle \forall l \in \mathsf{updated}(\bar{e}_2).\ l := \texttt{None}, \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}\rangle \Downarrow \langle (), \bar{\mu}_2\rangle$, and (6) $\mu_1 \cup \{x \mapsto v_1\} \geq \Gamma_1, x : \tau_1 \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$, we get $\mu_2 \geq_{\Gamma_2} \bar{\mu}_2$. I.e. $\mu_2 \setminus \{x\} \geq_{\Gamma_2 \setminus \{x\}} \bar{\mu}_2 \setminus \{x\}$.

But $x \notin \mathsf{updated}(\bar{e}_2)$. Thus $\langle \forall l \in \mathsf{updated}(\bar{e}_2).\ l := \texttt{None}, \bar{\mu}_1\rangle \Downarrow \langle (), \bar{\mu}_2 \setminus \{x\}\rangle$. Hence $\langle \forall l \in \mathsf{updated}(\bar{e}_1, \bar{e}_2).\ l := \texttt{None}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}_2 \setminus \{x\}\rangle$. As $\mu_f = \mu_2 \setminus \{x\}$ and $\bar{\mu}_f = \bar{\mu}_2 \setminus \{x\}$, it follows that $\mu_f \geq_{\Gamma_2 \setminus \{x\}} \bar{\mu}_f$.

Case: $e = \mathsf{case}\ e'\ \mathsf{of}\ \texttt{inl}\ x.\ e_1 \quad \texttt{inr}\ x.\ e_2$**.** From rule T-CASE-*, $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}', \bar{e}_1, \bar{e}_2)$.

From i.h. applied to (1) $\langle e', \mu\rangle \Downarrow \langle v, \mu_1\rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar{e}').\ l := \texttt{None}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}_1\rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get $\mu_1 \geq_{\Gamma'} \bar{\mu}_1$.

We assume e evaluates under rule E-CASE-INL. (For the case when it evaluates under rule E-CASE-INR, the proof is analoguous.) Hence, $v = \texttt{inl}\ v_1$. Let $\bar{v}_1$ be such that $v_1 \geq_{\tau_1} \bar{v}_1$. Then $\mu_1 \cup \{x \mapsto v_1\} \geq_{\Gamma', x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$.

From i.h. applied to (4) $\langle e_1, \mu_1 \cup \{x \mapsto v_1\}\rangle \Downarrow \langle v_2, \mu_2\rangle$, (5) $\langle \forall l \in \mathsf{updated}(\bar{e}_1).\ l := \texttt{None}, \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}\rangle \Downarrow \langle (), \bar{\mu}_2\rangle$, and (6) $\mu_1 \cup \{x \mapsto v_1\} \geq_{\Gamma', x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$, we get $\mu_2 \geq_{\Gamma''} \bar{\mu}_2$. Hence, $\mu_2 \setminus \{x\} \geq_{\Gamma'' \setminus \{x\}} \bar{\mu}_2 \setminus \{x\}$.

Since $x \notin \mathsf{updated}(\bar{e}_1)$, it follows that $\langle \forall l \in \mathsf{updated}(\bar{e}_1).\ l := \texttt{None}, \bar{\mu}_1\rangle \Downarrow \langle (), \bar{\mu}_2 \setminus \{x\}\rangle$.

From $\langle \forall l \in \mathsf{updated}(\bar{e}_2).\ l := \texttt{None}, \bar{\mu}_2 \setminus \{x\}\rangle \Downarrow \langle (), \bar{\mu}_3\rangle$, $\bar{\mu}_f = \bar{\mu}_3$. But $\mu_f = \mu_2 \setminus \{x\}$. We show below that $\mu_f = \mu_2 \setminus \{x\} \geq_{\Gamma'' \setminus \{x\}} \bar{\mu}_3 = \bar{\mu}_f$.

For all memory locations $l \in \mathsf{updated}(\bar{e}_2)$, $\mu_f(l) \geq_{\Gamma''(l)} \bar{\mu}_f(l) = \texttt{None}$. For all memory locations $l \in dom(\mu_f) \setminus \mathsf{updated}(\bar{e}_2)$, $\mu_f(l) = (\mu_2 \setminus \{x\})(l) \geq_{\Gamma''(l)} (\bar{\mu}_2 \setminus \{x\})(l) = \bar{\mu}_f(l)$. Thus $\mu_f \geq_{\Gamma'' \setminus \{x\}} \bar{\mu}_f$.

Case: $e = \texttt{while}\ x\ \mathsf{do}\ e'$**.** From rule T-WHILE-*, $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}')$.

We distinguish two subcases:

Subcase: $\mu(x) = 0$**.** Then rule E-WHILE-FALSE applies and $\mu_f = \mu$.

From $\langle \forall l \in \mathsf{updated}(\bar{e}').\ l := \texttt{None}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}'\rangle$, $\bar{\mu}_f = \bar{\mu}'$. We show below that $\mu_f \geq_\Gamma \bar{\mu}_f$.

For all memory locations $l \in \mathsf{updated}(\bar{e}')$, $\mu_f(l) \geq_{\Gamma(l)} \bar{\mu}_f(l) = \texttt{None}$. For all memory locations $l \in dom(\mu_f) \setminus \mathsf{updated}(\bar{e}')$, $\mu_f(l) = \mu(l) \geq_{\Gamma(l)} \bar{\mu}(l) = \bar{\mu}_f(l)$. Hence $\mu_f \geq_\Gamma \bar{\mu}_f$.

Subcase: $\mu(x) \neq 0$**.** Then rule E-WHILE-TRUE applies.

$$\frac{\mu(x) \neq 0 \qquad \langle e', \mu\rangle \Downarrow \langle v, \mu'\rangle \qquad \langle \texttt{while}\ x\ \mathsf{do}\ e', \mu'\rangle \Downarrow \langle v', \mu''\rangle}{\langle \texttt{while}\ x\ \mathsf{do}\ e', \mu\rangle \Downarrow \langle v', \mu''\rangle}\ \text{E-WHILE-TRUE}$$

From i.h. applied to (1) $\langle e', \mu\rangle \Downarrow \langle v, \mu'\rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar{e}').\ l := \texttt{None}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}'\rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get $\mu' \geq_\Gamma \bar{\mu}'$.

Since $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}')$, $\langle \forall l \in \mathsf{updated}(\bar{e}).\ l := \mathtt{None}, \bar{\mu}' \rangle \Downarrow \langle (), \bar{\mu}' \rangle$ and $\bar{\mu}_f = \bar{\mu}'$. Also $\mu_f = \mu''$. For all memory locations $l \in dom(\mu_f) \setminus \mathsf{updated}(\bar{e}')$, $\mu_f(l) = \mu'(l) \geq_{\Gamma(l)} \bar{\mu}'(l) = \bar{\mu}_f(l)$. For all memory locations $l \in \mathsf{updated}(\bar{e}')$, $\bar{\mu}_f(l) = \mathtt{None}$. Hence $\mu_f(l) \geq_{\Gamma(l)} \bar{\mu}_f(l)$. Hence $\mu_f \geq_\Gamma \bar{\mu}_f$.

Case: $e = \bullet e'$**.** From rule T-UPGRADE, $\bar{e} = \mathtt{Some}\ \bar{e}'$. Then $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}')$.

From i.h. applied to (1) $\langle e', \mu \rangle \Downarrow \langle v, \mu' \rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar{e}').\ l := \mathtt{None}, \bar{\mu} \rangle \Downarrow \langle (), \bar{\mu}' \rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get $\mu' \geq_\Gamma \bar{\mu}'$. Thus $\langle \forall l \in \mathsf{updated}(\mathtt{Some}\ \bar{e}').\ l := \mathtt{None}, \bar{\mu} \rangle \Downarrow \langle (), \bar{\mu}' \rangle$. As $\mu_f = \mu'$ and $\bar{\mu}_f = \bar{\mu}'$, it follows that $\mu_f \geq_\Gamma \bar{\mu}_f$. ∎

Proof of Theorem 35. By induction on the structure of the type-directed transformation $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$.

Suppose $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$, $\mu \geq_\Gamma \bar{\mu}$, and $\langle e, \mu \rangle \Downarrow \langle v_f, \mu_f \rangle$. We prove that $\langle \bar{e}, \bar{\mu} \rangle \Downarrow \langle \bar{v}_f, \bar{\mu}_f \rangle$, where $v_f \geq_\tau \bar{v}_f$ and $\mu_f \geq_{\Gamma'} \bar{\mu}_f$ by induction on the derivation of the type-directed transformation $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma'$ and cases according to the last rule applied.

Case: T-VAL. From rule E-VAL it follows that $v_f = v$ and $\mu_f = \mu$, and respectively, $\bar{v}_f = v$ and $\bar{\mu}_f = \bar{\mu}$. Hence $v_f \geq_\tau \bar{v}_f$ and $\mu_f \geq_\Gamma \bar{\mu}_f$.

Case: T-VARLOC. From rule E-VARLOC it follows that $v_f = \mu(l)$ and $\mu_f = \mu$, and respectively, $\bar{v}_f = \bar{\mu}(l)$ and $\bar{\mu}_f = \bar{\mu}$. As $\mu \geq_\Gamma \bar{\mu}$, $\mu(l) \geq_{\Gamma(l)} \bar{\mu}(l)$. Thus $v_f \geq_{\Gamma'(l)} \bar{v}_f$ and $\mu_f \geq_{\Gamma'} \bar{\mu}_f$ ($\Gamma' = \Gamma$).

Case: T-EXP-L. From i.h. applied to: (1) $pc \vdash \Gamma\{e_1 : \mathtt{Int}^L \rightsquigarrow \bar{e}_1\}\Gamma_1$, (2) $\langle e_1, \mu \rangle \Downarrow \langle n_1, \mu_1 \rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get that $\langle \bar{e}_1, \bar{\mu} \rangle \Downarrow \langle \bar{v}_1, \bar{\mu}_1 \rangle$, with $n_1 \geq_{\mathtt{Int}^L} \bar{v}_1$ and $\mu_1 \geq_{\Gamma_1} \bar{\mu}_1$. From the rules in Figures 4.7 and 4.8 it follows that $\bar{v}_1 = n_1$.

From i.h. applied to: (4) $pc \vdash \Gamma_1\{e_2 : \mathtt{Int}^L \rightsquigarrow \bar{e}_2\}\Gamma_2$, (5) $\langle e_2, \mu_1 \rangle \Downarrow \langle n_2, \mu_2 \rangle$, and (6) $\bar{\mu}_1 \geq_{\Gamma_1} \bar{\mu}_1$, we get that $\langle \bar{e}_2, \bar{\mu}_1 \rangle \Downarrow \langle \bar{v}_2, \bar{\mu}_2 \rangle$, with $n_2 \geq_{\mathtt{Int}^L} \bar{v}_2$ and $\mu_2 \geq_{\Gamma_2} \bar{\mu}_2$. From the rules in Figures 4.7 and 4.8 it follows that $\bar{v}_2 = n_2$.

By applying rule E-EXP, we obtain $\bar{v}_f = n_1 \oplus n_2$ and $\bar{\mu}_f = \bar{\mu}_2$:

$$\frac{\langle \bar{e}_1, \bar{\mu} \rangle \Downarrow \langle n_1, \bar{\mu}_1 \rangle \qquad \langle \bar{e}_2, \bar{\mu}_1 \rangle \Downarrow \langle n_2, \bar{\mu}_2 \rangle}{\langle \bar{e}_1 \oplus \bar{e}_2, \bar{\mu} \rangle \Downarrow \langle n_1 \oplus n_2, \bar{\mu}_2 \rangle}\ \text{E-EXP}$$

As $v_f = n_1 \oplus n_2$ and $\mu_f = \mu_2$, it follows that $v_f \geq_{\mathtt{Int}^L} \bar{v}_f$ and $\mu_f \geq_{\Gamma_2} \bar{\mu}_f$.

Case: T-EXP-H. From i.h. applied to: (1) $pc \vdash \Gamma\{e_1 : \mathtt{Int}^H \rightsquigarrow \bar{e}_1\}\Gamma_1$, (2) $\langle e_1, \mu \rangle \Downarrow \langle n_1, \mu_1 \rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get that $\langle \bar{e}_1, \bar{\mu} \rangle \Downarrow \langle \bar{v}_1, \bar{\mu}_1 \rangle$, with $n_1 \geq_{\mathtt{Int}^H} \bar{v}_1$ and $\mu_1 \geq_{\Gamma_1} \bar{\mu}_1$. From the rules in Figures 4.7 and 4.8 it follows that $\bar{v}_1 = \mathtt{Some}\ n_1$ or $\bar{v}_1 = \mathtt{None}$.

From i.h. applied to: (4) $pc \vdash \Gamma_1\{e_2 : \mathtt{Int}^H \rightsquigarrow \bar{e}_2\}\Gamma_2$, (5) $\langle e_2, \mu_1 \rangle \Downarrow \langle n_2, \mu_2 \rangle$, and (6) $\mu_1 \geq_{\Gamma_1} \bar{\mu}_1$, we get that $\langle \bar{e}_2, \bar{\mu}_1 \rangle \Downarrow \langle \bar{v}_2, \bar{\mu}_2 \rangle$, with $n_2 \geq_{\mathtt{Int}^H} \bar{v}_2$ and $\mu_2 \geq_{\Gamma_2} \bar{\mu}_2$. From the rules in Figures 4.7 and 4.8 it follows that $\bar{v}_2 = \mathtt{Some}\ n_2$ or $\bar{v}_2 = \mathtt{None}$. We distinguish four sub-cases:

Subcase: $(\bar{v}_1, \bar{v}_2) = (\mathtt{Some}\ n_1, \mathtt{Some}\ n_2)$**.** The proof is similar to the one for case E-EXP, subcase T-EXP-H, Theorem 34.

Subcase: $(\bar{v}_1, \bar{v}_2) = (\mathtt{Some}\ n_1, \mathtt{None})$**.** From the derivation in Figure B.5 we get $\bar{v}_f = \mathtt{None}$ and $\bar{\mu}_f = \bar{\mu}_2$. As $v_f = \mathtt{Some}\ (n_1 \oplus n_2)$ and $\mu_f = \mu_2$, it follows that $v_f \geq_{\mathtt{Int}^H} \bar{v}_f$ and $\mu_f \geq_{\Gamma_2} \bar{\mu}_f$.

Subcase: $(\bar{v}_1, \bar{v}_2) = (\mathtt{None}, \mathtt{Some}\ n_2)$**.** From the derivation in Figure B.6 we get $\bar{v}_f = \mathtt{None}$ and $\bar{\mu}_f = \bar{\mu}_2$. As $v_f = \mathtt{Some}\ (n_1 \oplus n_2)$ and $\mu_f = \mu_2$, it follows that $v_f \geq_{\mathtt{Int}^H} \bar{v}_f$ and $\mu_f \geq_{\Gamma_2} \bar{\mu}_f$.

Subcase: $(\bar{v}_1, \bar{v}_2) = (\mathtt{None}, \mathtt{None})$**.** Similar to previous case.

$$\dfrac{\langle \bar{e}_1,\bar{\mu}\rangle \Downarrow \langle \texttt{Some}\ n_1,\bar{\mu}_1\rangle \quad \dfrac{\dfrac{\langle \bar{e}_2,\bar{\mu}_1\rangle \Downarrow \langle \texttt{None},\bar{\mu}_2\rangle}{\langle \bar{e}_2,\bar{\mu}_1 \cup \{x \mapsto \texttt{Some}\ n_1\}\rangle \Downarrow \langle \texttt{None},\bar{\mu}_2 \cup \{x \mapsto \texttt{Some}\ n_1\}\rangle}\,\textsc{Lem. 33} \quad \dfrac{\dfrac{\bar{\mu}'_2(x) = \texttt{Some}\ n_1}{\langle x,\bar{\mu}'_2\rangle \Downarrow \langle \texttt{Some}\ n_1,\bar{\mu}'_2\rangle}\,\text{E-VARLOC} \quad \dfrac{\dfrac{(\bar{\mu}'_2 \cup \{x' \mapsto n_1\})(y) = \texttt{None}}{\langle y,\bar{\mu}'_2 \cup \{x' \mapsto n_1\}\rangle \Downarrow \langle \texttt{None},\bar{\mu}'_2 \cup \{x' \mapsto n_1\}\rangle}\,\text{E-VARLOC} \quad \dfrac{}{\langle \texttt{None},\bar{\mu}'_2 \cup \{x' \mapsto n_1\}\rangle \Downarrow \langle \texttt{None},\bar{\mu}'_2 \cup \{x' \mapsto n_1\}\rangle}\,\text{E-VAL}}{\langle \texttt{case}\ y\ \texttt{of None. None}\quad \texttt{Some}\ y'.\ \texttt{Some}\ (x' \oplus y'),\bar{\mu}'_2 \cup \{x' \mapsto n_1\}\rangle \Downarrow \langle \texttt{None},\bar{\mu}'_2 \cup \{x' \mapsto n_1\}\rangle}\,\text{E-CASE-INL}}{\begin{array}{l}\langle \texttt{case}\ x\ \texttt{of None. None}\\ \qquad \texttt{Some}\ x'.\ \texttt{case}\ y\ \texttt{of None. None}\quad \texttt{Some}\ y'.\ \texttt{Some}\ (x' \oplus y'),\bar{\mu}'_2\rangle \Downarrow \langle \texttt{Some}\ (n_1 \oplus n_2),\bar{\mu}'_2\rangle\end{array}}\,\text{E-CASE-INR}}{\begin{array}{l}\langle \texttt{let}\ y = \bar{e}_2\ \texttt{in}\\ \quad \texttt{case}\ x\ \texttt{of None. None}\quad \texttt{Some}\ x'.\ (\texttt{case}\ y\ \texttt{of None. None}\quad \texttt{Some}\ y'.\ \texttt{Some}\ (x' \oplus y')),\bar{\mu}'_1\rangle \Downarrow \langle \texttt{None},\bar{\mu}_2 \cup \{x \mapsto \texttt{Some}\ n_1\}\rangle\end{array}}\,\text{E-LET}}{\begin{array}{l}\langle \texttt{let}\ x = \bar{e}_1\ \texttt{in let}\ y = \bar{e}_2\ \texttt{in}\\ \quad \texttt{case}\ x\ \texttt{of None. None}\quad \texttt{Some}\ x'.\ (\texttt{case}\ y\ \texttt{of None. None}\quad \texttt{Some}\ y'.\ \texttt{Some}\ (x' \oplus y')),\bar{\mu}\rangle \Downarrow \langle \texttt{Some}\ (n_1 \oplus n_2),\bar{\mu}_2\rangle\end{array}}\,\text{E-LET}$$

where $\bar{\mu}'_1 = \bar{\mu}_1 \cup \{x \mapsto \texttt{Some}\ n_1\}$ and $\bar{\mu}'_2 = \bar{\mu}_2 \cup \{x \mapsto \texttt{Some}\ n_1, y \mapsto \texttt{None}\}$.

Figure B.5: Concluding derivation, case T-EXP-H, subcase $(\bar{v}_1,\bar{v}_2)$ = (Some n_1,None), Theorem 35.

Case: T-ASSIGN. From i.h. applied to: (1) $pc \vdash \Gamma\{e' : \tau \rightsquigarrow \bar{e}'\}\Gamma'$, (2) $\langle e',\mu\rangle \Downarrow \langle v,\mu'\rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get that $\langle \bar{e}',\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}'\rangle$, with $v \geq_\tau \bar{v}$ and $\mu' \geq_{\Gamma'} \bar{\mu}'$.

By applying rule E-ASSIGN, we get $\bar{v}_f = ()$ and $\bar{\mu}_f = \bar{\mu}'[l \mapsto \bar{v}]$:

$$\dfrac{\langle \bar{e}',\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}'\rangle}{\langle l := \bar{e}',\bar{\mu}\rangle \Downarrow \langle (),\bar{\mu}'[l \mapsto \bar{v}]\rangle}\,\text{E-ASSIGN}$$

Thus $v_f = () \geq_{\texttt{Unit}^L} () = \bar{v}_f$. We are left to show that $\mu'[l \mapsto v] \geq_{\Gamma'[l \mapsto \tau]} \bar{\mu}'[l \mapsto \bar{v}]$. From i.h. $\mu' \geq_{\Gamma'} \bar{\mu}'$ and $v \geq_\tau \bar{v}$. Hence $\mu'[l \mapsto v](l) \geq_{l:\tau} \bar{\mu}'[l \mapsto \bar{v}](l)$. Since for all $l \in \Gamma'$, $\mu'(l) \geq_{\Gamma'(l)} \bar{\mu}'$, we get that $\mu'[l \mapsto v] \geq_{\Gamma'[l \mapsto \tau]} \bar{\mu}'[l \mapsto \bar{v}]$.

Case: T-INL. From i.h. applied to: (1) $pc \vdash \Gamma\{e' : \tau_1 \rightsquigarrow \bar{e}'\}\Gamma'$, (2) $\langle e',\mu\rangle \Downarrow \langle v,\mu'\rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get that $\langle \bar{e}',\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}'\rangle$, with $v \geq_{\tau_1} \bar{v}$ and $\mu' \geq_{\Gamma'} \bar{\mu}'$.

By applying rule E-INL, we get $\bar{v}_f = \texttt{inl}\ \bar{v}$ and $\bar{\mu}_f = \bar{\mu}'$:

$$\dfrac{\langle \bar{e}',\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}'\rangle}{\langle \texttt{inl}\ \bar{e}',\bar{\mu}\rangle \Downarrow \langle \texttt{inl}\ \bar{v},\bar{\mu}'\rangle}\,\text{E-INL}$$

As $v_f = \texttt{inl}\ v$ and $\mu_f = \mu'$, it follows that $v_f = \texttt{inl}\ v \geq_{(\tau_1+\tau_2)^L} \texttt{inl}\ \bar{v} = \bar{v}_f$ (from Figures 4.7 and 4.8) and $\mu_f = \mu' \geq_{\Gamma'} \bar{\mu}' = \bar{\mu}_f$ (from i.h.).

Case: T-INR. Similar to case T-INL.

Case: T-LET. From i.h. applied to: (1) $pc \vdash \Gamma\{e_1 : \tau_1 \rightsquigarrow \bar{e}_1\}\Gamma_1$, (2) $\langle e_1,\mu\rangle \Downarrow \langle v_1,\mu_1\rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get that $\langle \bar{e}_1,\bar{\mu}\rangle \Downarrow \langle \bar{v}_1,\bar{\mu}_1\rangle$, with $v_1 \geq_{\tau_1} \bar{v}_1$ and $\mu_1 \geq_{\Gamma_1} \bar{\mu}_1$. Thus $\mu_1 \cup \{x \mapsto v_1\} \geq_{\Gamma_1,x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$.

$$\dfrac{\langle \bar{e}_1,\bar{\mu}\rangle \Downarrow \langle \texttt{None},\bar{\mu}_1\rangle \qquad \dfrac{\dfrac{\langle \bar{e}_2,\bar{\mu}_1\rangle \Downarrow \langle \texttt{Some}\ n_2,\bar{\mu}_2\rangle}{\langle \bar{e}_2,\bar{\mu}_1\cup\{x\mapsto\texttt{None}\}\rangle \Downarrow \langle \texttt{Some}\ n_2,\bar{\mu}_2\cup\{x\mapsto\texttt{None}\}\rangle}\ \textsc{Lem. 33}}{\begin{array}{l}\langle \texttt{let}\ y=\bar{e}_2\ \texttt{in}\\ \quad \texttt{case}\ x\ \texttt{of None. None}\\ \qquad\qquad \texttt{Some}\ x'.\ (\texttt{case}\ y\ \texttt{of None. None}\quad \texttt{Some}\ y'.\ \texttt{Some}\ (x'\oplus y')),\bar{\mu}_1'\rangle \Downarrow \langle \texttt{None},\bar{\mu}_2'\cup\{x\mapsto\texttt{Some}\ n_1\}\rangle\end{array}}\ \text{E-LET}}{\begin{array}{l}\langle \texttt{let}\ x=\bar{e}_1\ \texttt{in let}\ y=\bar{e}_2\ \texttt{in}\\ \quad \texttt{case}\ x\ \texttt{of None. None}\\ \qquad\qquad \texttt{Some}\ x'.\ (\texttt{case}\ y\ \texttt{of None. None}\quad \texttt{Some}\ y'.\ \texttt{Some}\ (x'\oplus y')),\bar{\mu}\rangle \Downarrow \langle \texttt{Some}\ (n_1\oplus n_2),\bar{\mu}_2\rangle\end{array}}\ \text{E-LET}$$

where $\bar{\mu}_1' = \bar{\mu}_1 \cup \{x \mapsto \texttt{None}\}$.

Figure B.6: Concluding derivation, case T-EXP-H, subcase $(\bar{v}_1,\bar{v}_2) = (\texttt{None},\texttt{Some}\ n_2)$, Theorem 35.

From i.h. applied to: (4) $pc \vdash \Gamma_1, x : \tau_1\{e_2 : \tau_2 \rightsquigarrow \bar{e}_2\}\Gamma_2$, (5) $\langle e_2, \mu_1 \cup \{x \mapsto v_1\}\rangle \Downarrow \langle v_2, \mu_2\rangle$, and (6) $\mu_1 \cup \{x \mapsto v_1\} \geq_{\Gamma_1,x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$, we get that $\langle \bar{e}_2, \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2, \bar{\mu}_2\rangle$, with $v_2 \geq_{\tau_2} \bar{v}_2$ and $\mu_2 \geq_{\Gamma_2} \bar{\mu}_2$. Hence $\mu_2 \setminus \{x\} \geq_{\Gamma_2\setminus\{x\}} \bar{\mu}_2 \setminus \{x\}$.

By applying rule E-LET we get $\bar{v}_f = \bar{v}_2$ and $\bar{\mu}_f = \bar{\mu}_2 \setminus \{x\}$:

$$\dfrac{\langle \bar{e}_1,\bar{\mu}\rangle \Downarrow \langle \bar{v}_1,\bar{\mu}_1\rangle \qquad \langle \bar{e}_2,\bar{\mu}_1\cup\{x\mapsto\bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_2\rangle}{\langle \texttt{let}\ x=\bar{e}_1\ \texttt{in}\ \bar{e}_2,\mu\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_2\setminus\{x\}\rangle}\ \text{E-LET}$$

As $v_f = v_2$ and $\mu_f = \mu_2 \setminus \{x\}$, it follows that $v_f \geq_{\tau_2} \bar{v}_f$ and $\mu_f \geq_{\Gamma_2\setminus\{x\}} \bar{\mu}_f$.

Case: T-CASE-L. We assume expression e' evaluates under rule E-CASE-INL. When it evaluates under rule E-CASE-INR, the reasoning and proof are analogous.

From i.h. applied to: (1) $pc \vdash \Gamma\{e' : (\tau_1 + \tau_2)^L \rightsquigarrow \bar{e}'\}\Gamma'$, (2) $\langle e', \mu\rangle \Downarrow \langle \texttt{inl}\ v_1, \mu_1\rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get that $\langle \bar{e}', \bar{\mu}\rangle \Downarrow \langle \bar{v}, \bar{\mu}_1\rangle$, with $\texttt{inl}\ v_1 \geq_{(\tau_1+\tau_2)^L} \bar{v}$ and $\mu_1 \geq_{\Gamma'} \bar{\mu}_1$. From the rules in Figures 4.7 and 4.8 it follows that $\bar{v} = \texttt{inl}\ \bar{v}_1$, with $v_1 \geq_{\tau_1} \bar{v}_1$. As $\mu_1 \geq_{\Gamma'} \bar{\mu}_1$, $\mu_1 \cup \{x \mapsto v_1\} \geq_{\Gamma',x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$.

From i.h. applied to: (4) $pc \vdash \Gamma', x : \tau_1\{e_1 : \tau \rightsquigarrow \bar{e}_1\}\Gamma_1$, (5) $\langle e_1, \mu_1 \cup \{x \mapsto v_1\}\rangle \Downarrow \langle v_2, \mu_2\rangle$, and (6) $\mu_1 \cup \{x \mapsto v_1\} \geq_{\Gamma',x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$, we get that $\langle \bar{e}_1, \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2, \bar{\mu}_2\rangle$, with $v_2 \geq_\tau \bar{v}_2$ and $\mu_2 \geq_{\Gamma_1} \bar{\mu}_2$. Hence $\mu_2 \setminus \{x\} \geq_{\Gamma_1\setminus\{x\}} \bar{\mu}_2 \setminus \{x\}$.

From the derivation below it follows that $\bar{v}_f = \bar{v}_2$ and $\bar{\mu}_f = \bar{\mu}_3 \setminus \{x, y\}$:

$$\dfrac{\langle \bar{e}',\bar{\mu}\rangle \Downarrow \langle \texttt{inl}\ \bar{v}_1,\bar{\mu}_1\rangle \qquad \dfrac{\langle \bar{e}_1,\bar{\mu}_1\cup\{x\mapsto\bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_2\rangle \qquad \dfrac{\dfrac{}{\langle \mathsf{upgrade}(S_2),\bar{\mu}_2\cup\{y\mapsto\bar{v}_2\}\rangle \Downarrow \langle (),\bar{\mu}_3\rangle}\ \text{E-ASSIGN}^* \qquad \dfrac{\bar{\mu}_3(y)=\bar{v}_2}{\langle y,\bar{\mu}_3\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_3\rangle}\ \text{E-VAR}}{\langle \mathsf{upgrade}(S_2);y,\bar{\mu}_2\cup\{y\mapsto\bar{v}_2\}\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_3\rangle}\ \text{E-SEQ}}{\langle \texttt{let}\ y=\bar{e}_1\ \texttt{in}\ \mathsf{upgrade}(S_2);y,\bar{\mu}_1\cup\{x\mapsto\bar{v}_1\}\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_3\setminus\{y\}\rangle}\ \text{E-LET}}{\langle \texttt{case}\ \bar{e}'\ \texttt{of inl}\ x.\ \texttt{let}\ y=\bar{e}_1\ \texttt{in}\ \mathsf{upgrade}(S_2);y\quad \texttt{inr}\ x.\ \texttt{let}\ y=\bar{e}_2\ \texttt{in}\ \mathsf{upgrade}(S_1);y,\bar{\mu}\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_3\setminus\{y,x\}\rangle}\ \text{E-CASE-INL}$$

As $v_f = v_2$, $v_f \geq_\tau \bar{v}_f$. Since $\mu_f = \mu_2 \setminus \{x\}$, it remains to show that $\mu_2 \setminus \{x\} \geq_{\bigsqcup\Gamma_i\setminus\{x\}} \bar{\mu}_3 \setminus \{x, y\}$.

If $high(\Gamma)$ represents the set of all memory locations and variables with high type in Γ, and $low(\Gamma)$ represents the set of all memory locations and variables with low type in Γ, then $S_2 = high(\Gamma_2) \setminus low(\Gamma_1)$. Hence for all $l \in S_2$, $\Gamma_1(l) = \sigma^L$ and $\Gamma_2(l) = \sigma^H$.

$$\textsc{E-CASE-INL}\quad \dfrac{\langle \bar{e}',\bar{\mu}\rangle \Downarrow \langle \texttt{None},\bar{\mu}_1\rangle \qquad \dfrac{\dfrac{}{\langle \forall l \in \mathsf{updated}(\bar{e}_1,\bar{e}_2).l := \texttt{None},\bar{\mu}_1\rangle \Downarrow \langle (),\bar{\mu}_2\rangle}\,\textsc{E-ASSIGN}^{*} \qquad \dfrac{}{\langle \texttt{None},\bar{\mu}_2\rangle \Downarrow \langle \texttt{None},\bar{\mu}_2\rangle}\,\textsc{E-VAL}}{\langle \forall l \in \mathsf{updated}(\bar{e}_1,\bar{e}_2).\ l := \texttt{None};\texttt{None},\bar{\mu}_1\rangle \Downarrow \langle \texttt{None},\bar{\mu}_2\rangle}\,\textsc{E-SEQ}}{\begin{array}{l}\langle \texttt{case}\ \bar{e}'\ \texttt{of None.}\ (\forall l \in \mathsf{updated}(\bar{e}_1,\bar{e}_2).\ l := \texttt{None});\ \texttt{None}\\ \qquad\qquad \texttt{Some}\ x'.\ \texttt{case}\ x'\ \texttt{of inl}\ x.\ \texttt{let}\ y = \bar{e}_1\ \texttt{in}\ \mathsf{upgrade}(S_2);y \quad \texttt{inr}\ x.\ \texttt{let}\ y = \bar{e}_2\ \texttt{in}\ \mathsf{upgrade}(S_1);y,\bar{\mu}\rangle \langle \texttt{None},\bar{\mu}_2\rangle\end{array}}$$

Figure B.7: Concluding derivation, case T-CASE-H, subcase $\bar{v} = \texttt{None}$, Theorem 35.

From i.h., $\mu_2 \geq_{\Gamma_1} \bar{\mu}_2$. Hence, for all $l \in dom(\Gamma_1)$, $\mu_2(l) \geq_{\Gamma_1(l)} \bar{\mu}_2(l)$. Thus for all $l \in S_2$, $\mu_2(l) \geq_{\sigma^L} \bar{\mu}_2(l)$, for some σ. From the rules in Figure 4.8 it follows that for all $l \in S_2$, $\mu_2(l) \geq_{\sigma^H}$ $\texttt{Some}\ \bar{\mu}_2(l)$. Upgrading the memory locations from S_2 corresponds to setting the label to H for all the types labeled L in Γ_1, but labeled H in Γ_2, i.e. $\forall l \in S_2$, $\Gamma_1[l \mapsto \sigma^H] = (\Gamma_1 \sqcup \Gamma_2)(l)$, given that $\Gamma_1(l) = \sigma^L$. Thus for all $l \in S_2$, $\mu_2(l) \geq_{(\Gamma_1 \sqcup \Gamma_2)(l)} \texttt{Some}\ \bar{\mu}_2(l) = \bar{\mu}_3(l)$. Hence $\mu_2 \geq_{\bigsqcup \Gamma_i \setminus \{x\}}$ $\bar{\mu}_3 \setminus \{x,y\}$.

Case: T-CASE-H. The transformation derivation is of the form:

$$\textsc{T-CASE-H}\quad \dfrac{pc \vdash \Gamma\{e : (\tau_1+\tau_2)^H \rightsquigarrow \bar{e}\}\Gamma' \qquad H \vdash \Gamma',x:\tau_i\{e_i : \tau \rightsquigarrow \bar{e}_i\}\Gamma_i \qquad i = 1,2 \qquad S_1 = \Gamma_1 \setminus \Gamma_2 \qquad S_2 = \Gamma_2 \setminus \Gamma_1}{\begin{array}{l} pc \vdash \Gamma\{\texttt{case}\ e\ \texttt{of inl}\ x.\ e_1 \quad \texttt{inr}\ x.\ e_2 : \tau \rightsquigarrow \\ \qquad \texttt{case}\ \bar{e}\ \texttt{of None.}\ (\forall l \in \mathsf{updated}(\bar{e}_1,\bar{e}_2).\ l := \texttt{None});\ \texttt{None} \\ \qquad\qquad\qquad \texttt{Some}\ x'.\ \texttt{case}\ x'\ \texttt{of inl}\ x.\ \texttt{let}\ y = \bar{e}_1\ \texttt{in}\ \mathsf{upgrade}(S_2);y \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{inr}\ x.\ \texttt{let}\ y = \bar{e}_2\ \texttt{in}\ \mathsf{upgrade}(S_1);y\}\bigsqcup\Gamma_i \setminus \{x\} \end{array}}$$

We assume expression e' evaluates under rule E-CASE-INL. When it evaluates under rule E-CASE-INR, the reasoning and proof are analogous.

From i.h. applied to: (1) $pc \vdash \Gamma\{e' : (\tau_1+\tau_2)^H \rightsquigarrow \bar{e}'\}\Gamma'$, (2) $\langle e',\mu\rangle \Downarrow \langle \texttt{inl}\ v_1,\mu_1\rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get that $\langle \bar{e}',\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}_1\rangle$, with $\texttt{inl}\ v_1 \geq_{(\tau_1+\tau_2)^H} \bar{v}$ and $\mu_1 \geq_{\Gamma'} \bar{\mu}_1$. From the rules in Figures 4.7 and 4.8 it follows that $\bar{v} = \texttt{Some inl}\ \bar{v}_1$, with $v_1 \geq_{\tau_1} \bar{v}_1$, or $\bar{v} = \texttt{None}$. We distinguish two subcases:

Subcase: $\bar{v} = \texttt{Some inl}\ \bar{v}_1$**.** Proof similar to previous case.

Subcase: $\bar{v} = \texttt{None}$**.** From the derivation in Figure B.7 it follows that $\bar{v}_f = \texttt{None}$ and $\bar{\mu}_f = \bar{\mu}_2$.

Let $\bar{v}_1$ be such that $v_1 \geq_{\tau_1} \bar{v}_1$. Then $\mu_1 \cup \{x \mapsto v_1\} \geq_{\Gamma_1,x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$. From Lemma 36 applied to (4) $H \vdash \Gamma',x : \tau_1\{e_1 : \tau \rightsquigarrow \bar{e}_1\}\Gamma_1$, (5) $\langle e_1,\mu_1 \cup \{x \mapsto v_1\}\rangle \Downarrow \langle v_2,\mu_2\rangle$, and (6) $\mu_1 \cup \{x \mapsto v_1\} \geq_{\Gamma_1,x:\tau_1} \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$, we get that $\langle \forall l \in \mathsf{updated}(\bar{e}_1).\ l := \texttt{None};\texttt{None},\bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\rangle \Downarrow \langle \texttt{None},\bar{\mu}'_1 \cup \{x \mapsto \bar{v}_1\}\rangle$ with $\mu_2 \setminus \{x\} \geq_{\Gamma_1 \setminus \{x\}} \bar{\mu}'_1$. I.e. $\langle \forall l \in \mathsf{updated}(\bar{e}_1).\ l := \texttt{None};\texttt{None},\bar{\mu}_1\rangle \Downarrow \langle \texttt{None},\bar{\mu}'_1\rangle$.

Since $\mu_f = \mu_2$, $\mu_f \geq_{\Gamma_1 \setminus \{x\}} \bar{\mu}'_1$. I.e., for all $l \in \mathsf{updated}(\bar{e}_1)$, $\mu_f(l) \geq_{\Gamma_1 \setminus \{x\}} \bar{\mu}'_1(l) = \texttt{None} = \bar{\mu}_f(l)$.

If we denote by S the set of memory locations updated in both branches, and by S' the set of memory locations upgraded in both branches, then $\mathsf{updated}(\bar{e}_1) = \mathsf{updated}(S,S_1,S')$ and $\mathsf{updated}(\bar{e}_1,\bar{e}_2) = \mathsf{updated}(S,S_1,S_2,S')$. Hence, for all $l \in S_2$, $\Gamma_1(l) = \sigma^L$, for some σ, but $(\Gamma_1 \sqcup \Gamma_2)(l) = \sigma^H$. Also, for all $l \in S_2$, $\mu_f(l) \geq_{(\Gamma_1 \sqcup \Gamma_2)(l)} \texttt{None} = \bar{\mu}_f(l)$. In addition, for all $l \in dom(\mu_f) \setminus (S \cup S_1 \cup S_2 \cup S')$, $\mu_f(l) = \mu_1(l) \geq_{\Gamma'(l)=(\bigsqcup \Gamma_i \setminus \{x\})(l)} = \bar{\mu}_1(l) = \bar{\mu}_f(l)$. Hence $\mu_f \geq_{\bigsqcup \Gamma_i \setminus \{x\}} \bar{\mu}_f$.

Case: T-WHILE-L. The transformation derivation is of the form:

$$\frac{\Gamma(x) = \texttt{Int}^{L} \qquad pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma}{pc \vdash \Gamma\{\texttt{while}\ x\ \texttt{do}\ e : \texttt{Unit}^{L} \rightsquigarrow \texttt{while}\ x\ \texttt{do}\ \bar{e}\}\Gamma}\ \text{T-WHILE-L}$$

As $\mu \geq_\Gamma \bar{\mu}$ and $\Gamma(x) = \texttt{Int}^{L}$, from the rules in Figures 4.7 and 4.8 it follows that $\bar{\mu}(x) = \mu(x)$. For the evaluation rules, we distinguish two cases:

Subcase: E-WHILE-TRUE. The evaluation derivation is of the form:

$$\frac{\mu(x) \neq 0 \qquad \langle e, \mu\rangle \Downarrow \langle v, \mu'\rangle \qquad \langle \texttt{while}\ x\ \texttt{do}\ e, \mu'\rangle \Downarrow \langle v', \mu''\rangle}{\langle \texttt{while}\ x\ \texttt{do}\ e, \mu\rangle \Downarrow \langle v', \mu''\rangle}\ \text{E-WHILE-TRUE}$$

From i.h. applied to: (1) $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma$, (2) $\langle e, \mu\rangle \Downarrow \langle v, \mu'\rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get that $\langle \bar{e}, \bar{\mu}\rangle \Downarrow \langle \bar{v}, \bar{\mu}'\rangle$, with $v \geq_\tau \bar{v}$ and $\mu' \geq_\Gamma \bar{\mu}'$.

From i.h. applied to: (4) $pc \vdash \Gamma\{\texttt{while}\ x\ \texttt{do}\ e : \tau \rightsquigarrow \texttt{while}\ x\ \texttt{do}\ \bar{e}\}\Gamma$, (5) $\langle \texttt{while}\ x\ \texttt{do}\ e, \mu'\rangle \Downarrow \langle v', \mu''\rangle$, and (6) $\mu' \geq_\Gamma \bar{\mu}'$, we get that $\langle \texttt{while}\ x\ \texttt{do}\ \bar{e}, \bar{\mu}'\rangle \Downarrow \langle \bar{v}', \bar{\mu}''\rangle$, with $v' \geq_{\texttt{Unit}^{L}} \bar{v}'$ and $\mu'' \geq_\Gamma \bar{\mu}''$.

By applying rule E-WHILE-TRUE we get that $\bar{v}_f = \bar{v}'$ and $\bar{\mu}_f = \bar{\mu}''$:

$$\frac{\langle x, \bar{\mu}\rangle \Downarrow \langle \mu(x), \bar{\mu}\rangle \qquad \langle \bar{e}, \bar{\mu}\rangle \Downarrow \langle \bar{v}, \bar{\mu}'\rangle \qquad \langle \texttt{while}\ x\ \texttt{do}\ \bar{e}, \bar{\mu}'\rangle \Downarrow \langle \bar{v}', \bar{\mu}''\rangle}{\langle \texttt{while}\ x\ \texttt{do}\ \bar{e}, \bar{\mu}\rangle \Downarrow \langle \bar{v}', \bar{\mu}''\rangle}\ \text{E-WHILE-TRUE}$$

As $v_f = v'$ and $\mu_f = \mu''$, it follows that $v_f \geq_{\texttt{Unit}^{L}} \bar{v}_f$ and $\mu_f \geq_\Gamma \bar{\mu}_f$.

Subcase: E-WHILE-FALSE. By applying rule E-WHILE-FALSE we get that $\bar{v}_f = ()$ and $\bar{\mu}_f = \bar{\mu}$:

$$\frac{\langle \bar{\mu}, x\rangle \Downarrow \langle 0, \bar{\mu}\rangle}{\langle \texttt{while}\ x\ \texttt{do}\ \bar{e}, \bar{\mu}\rangle \Downarrow \langle (), \bar{\mu}\rangle}\ \text{E-WHILE-FALSE}$$

As $v_f = ()$ and $\mu_f = \mu'$, it follows that $v_f \geq_{\texttt{Unit}^{L}} \bar{v}_f$ and $\mu_f \geq_\Gamma \bar{\mu}_f$.

Case: T-WHILE-H. The transformation derivation is of the form:

$$\frac{\Gamma(x) = \texttt{Int}^{H} \qquad H \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma}{\begin{array}{l} pc \vdash \Gamma\{\texttt{while}\ x\ \texttt{do}\ e : \texttt{Unit}^{L} \rightsquigarrow \\ \qquad \texttt{while}\ (\texttt{case}\ x\ \texttt{of None}.\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None};\ 0 \quad \texttt{Some}\ y.\ y)\ \texttt{do}\ \bar{e}\}\Gamma \end{array}}\ \text{T-WHILE-H}$$

As $\mu \geq_\Gamma \bar{\mu}$ and $\Gamma(x) = \texttt{Int}^{L}$, from the rules in Figures 4.7 and 4.8 it follows that $\bar{\mu}(x) = \texttt{Some}\ \mu(x)$ or $\bar{\mu}(x) = \texttt{None}$. For the evaluation rules we distinguish two cases:

Subcase: E-WHILE-TRUE. The evaluation derivation is of the form:

$$\frac{\mu(x) \neq 0 \qquad \langle e, \mu\rangle \Downarrow \langle v, \mu'\rangle \qquad \langle \texttt{while}\ x\ \texttt{do}\ e, \mu'\rangle \Downarrow \langle v', \mu''\rangle}{\langle \texttt{while}\ x\ \texttt{do}\ e, \mu\rangle \Downarrow \langle v', \mu''\rangle}\ \text{E-WHILE-TRUE}$$

- $\bar{\mu}(x) = \texttt{Some}\ \mu(x)$

From i.h. applied to: (1) $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma$, (2) $\langle e, \mu\rangle \Downarrow \langle v, \mu'\rangle$, and (3) $\mu \geq_\Gamma \bar{\mu}$, we get that $\langle \bar{e}, \bar{\mu}\rangle \Downarrow \langle \bar{v}, \bar{\mu}'\rangle$, with $v \geq_\tau \bar{v}$ and $\mu' \geq_\Gamma \bar{\mu}'$.

By applying rule $\texttt{E-WHILE-TRUE}$ we get that $\bar{v}_f = \bar{v}'$ and $\bar{\mu}_f = \bar{\mu}''$:

$$\text{E-WHILE-TRUE}\quad \dfrac{\begin{array}{l}\dfrac{\langle x,\bar{\mu}\rangle \Downarrow \langle \mathtt{Some}\ \mu(x),\bar{\mu}\rangle \qquad \dfrac{(\bar{\mu}\cup\{y\mapsto\mu(x)\})(y)=\mu(x)}{\langle y,\bar{\mu}\cup\{y\mapsto\mu(x)\}\rangle \Downarrow \langle \mu(x),\bar{\mu}\cup\{y\mapsto\mu(x)\}\rangle}\,\text{E-VARLOC}}{\langle \mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ \forall l\in \mathsf{updated}(\bar{e}).\ l:=\mathtt{None};0\quad \mathtt{Some}\ y.\ y,\bar{\mu}\rangle \Downarrow \langle \mu(x),\bar{\mu}\rangle}\,\text{E-CASE-INR}\\ \langle \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}'\rangle\\ \langle \mathtt{while}\ (\mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ \forall l\in \mathsf{updated}(\bar{e}).\ l:=\mathtt{None};0\quad \mathtt{Some}\ y.\ y)\ \mathtt{do}\ \bar{e},\bar{\mu}'\rangle \Downarrow \langle \bar{v}',\bar{\mu}''\rangle\end{array}}{\langle \mathtt{while}\ (\mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ \forall l\in \mathsf{updated}(\bar{e}).\ l:=\mathtt{None};0\quad \mathtt{Some}\ y.\ y)\ \mathtt{do}\ \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v}',\bar{\mu}''\rangle}$$

From i.h. applied to: (4) $pc \vdash \Gamma\{\mathtt{while}\ x\ \mathtt{do}\ e : \tau \rightsquigarrow \mathtt{while}\ (\mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \mathtt{None};0\quad \mathtt{Some}\ y.\ y)\ \mathtt{do}\ \bar{e}\}\Gamma$, (5) $\langle \mathtt{while}\ x\ \mathtt{do}\ e,\mu'\rangle \Downarrow \langle v',\mu''\rangle$, and (6) $\mu' \geq_\Gamma \bar{\mu}'$, we get that $\langle \mathtt{while}\ (\mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ \forall l \in \mathsf{updated}(\bar{e}).\ l := \mathtt{None};0\quad \mathtt{Some}\ y.\ y)\ \mathtt{do}\ \bar{e},\bar{\mu}'\rangle \Downarrow \langle \bar{v}',\bar{\mu}''\rangle$, with $v' \geq_{\mathtt{Unit}^L} \bar{v}'$ and $\mu'' \geq_\Gamma \bar{\mu}''$.

As $v_f = v'$ and $\mu_f = \mu''$, it follows that $v_f \geq_{\mathtt{Unit}^L} \bar{v}_f$ and $\mu_f \geq_\Gamma \bar{\mu}_f$.

- $\bar{\mu}(x) = \mathtt{None}$

From the derivation below, $\bar{v}_f = ()$ and $\bar{\mu}_f = \bar{\mu}'$:

$$\text{E-WHILE-FALSE}\quad \dfrac{\begin{array}{l}\langle x,\bar{\mu}\rangle \Downarrow \langle \mathtt{None},\bar{\mu}\rangle\\ \dfrac{\dfrac{\dfrac{}{\langle \forall l\in \mathsf{updated}(\bar{e}).\ l:=\mathtt{None},\bar{\mu}\rangle \Downarrow \langle (),\bar{\mu}'\rangle}\,\text{E-ASSIGN}^* \qquad \dfrac{}{\langle 0,\bar{\mu}'\rangle \Downarrow \langle 0,\bar{\mu}'\rangle}\,\text{E-VAL}}{\langle \forall l\in \mathsf{updated}(\bar{e}).\ l:=\mathtt{None};0,\bar{\mu}'\rangle \Downarrow \langle 0,\bar{\mu}'\rangle}\,\text{E-SEQ}}{\langle \mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ \forall l\in \mathsf{updated}(\bar{e}).\ l:=\mathtt{None};0\quad \mathtt{Some}\ y.\ y,\bar{\mu}\rangle \Downarrow \langle 0,\bar{\mu}'\rangle}\,\text{E-CASE-INL}\end{array}}{\langle \mathtt{while}\ (\mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ \forall l\in \mathsf{updated}(\bar{e}).l:=\mathtt{None};0\quad \mathtt{Some}\ y.\ y)\ \mathtt{do}\ \bar{e},\bar{\mu}\rangle \Downarrow \langle (),\bar{\mu}'\rangle}$$

We apply Lemma 36 to (4) $H \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma$ (5) $\langle e,\mu\rangle \Downarrow \langle v,\mu'\rangle$, and (6) $\mu \geq_\Gamma \bar{\mu}$, and we get $\langle \forall l \in \mathsf{updated}(\bar{e}).\ l := \mathtt{None};\bar{\mu}\rangle \Downarrow \langle (),\bar{\mu}'\rangle$, with $\mu' \geq_\Gamma \bar{\mu}'$.

As $\mathsf{updated}(\mathtt{while}\ x\ \mathtt{do}\ e) = \mathsf{updated}(\bar{e})$ and $\forall l \in \mathsf{updated}(\bar{e})$, $\bar{\mu}_f(l) = \mathtt{None}$, we have that $\mu_f(l) \geq_{\Gamma(l)} \bar{\mu}_f(l)$. Since for all $l \in dom(\mu_f) \setminus \mathsf{updated}(\bar{e})$, $\mu_f(l) = \mu'(l) \geq_{\Gamma(l)} \bar{\mu}'(l) = \bar{\mu}_f(l)$, $\mu_f \geq_\Gamma \bar{\mu}_f$.

Subcase: E-WHILE-FALSE. The evaluation derivation is of the form:

$$\dfrac{\mu(x) = 0}{\langle \mathtt{while}\ x\ \mathtt{do}\ e,\mu\rangle \Downarrow \langle (),\mu\rangle}\,\text{E-WHILE-FALSE}$$

- $\bar{\mu}(x) = \mathtt{Some}\ 0$

From the derivation below it follows that $\bar{v}_f = ()$ and $\bar{\mu}_f = \bar{\mu}$:

$$\dfrac{\dfrac{\langle x,\bar{\mu}\rangle \Downarrow \langle \mathtt{Some}\ 0,\bar{\mu}\rangle \qquad \dfrac{(\bar{\mu}\cup\{y\mapsto 0\})(x)=0}{\langle y,\bar{\mu}\cup\{y\mapsto 0\}\rangle \Downarrow \langle 0,\bar{\mu}\cup\{y\mapsto 0\}\rangle}\,\text{E-VARLOC}}{\langle \mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ \forall l\in \mathsf{updated}(\bar{e}).\ l:=\mathtt{None};0\quad \mathtt{Some}\ y.\ y,\bar{\mu}\rangle \Downarrow \langle 0,\bar{\mu}\rangle}\,\text{E-CASE-INR}}{\langle \mathtt{while}\ (\mathtt{case}\ x\ \mathtt{of}\ \mathtt{None}.\ \forall l\in \mathsf{updated}(\bar{e}).\ l:=\mathtt{None};0\quad \mathtt{Some}\ y.\ y)\ \mathtt{do}\ \bar{e},\bar{\mu}\rangle \Downarrow \langle (),\bar{\mu}\rangle}\,\text{E-WHILE-FALSE}$$

As $v_f = ()$ and $\mu_f = \mu'$, it follows that $v_f \geq_{\mathtt{Unit}^L} \bar{v}_f$ and $\mu_f \geq_\Gamma \bar{\mu}'_f$.

- $\bar\mu(x) = \texttt{None}$

E-WHILE-FALSE
$\langle x, \bar\mu\rangle \Downarrow \langle \texttt{None}, \bar\mu\rangle$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle \forall l \in \mathsf{updated}(\bar e).\ l := \texttt{None}, \bar\mu\rangle \Downarrow \langle (), \bar\mu'\rangle}\ \text{E-ASSIGN}^* \qquad \dfrac{}{\langle 0, \bar\mu'\rangle \Downarrow \langle 0, \bar\mu'\rangle}\ \text{E-VAL}}{\langle \forall l \in \mathsf{updated}(\bar e).\ l := \texttt{None}; 0, \bar\mu'\rangle \Downarrow \langle 0, \bar\mu'\rangle}\ \text{E-SEQ}}{\langle \texttt{case}\ x\ \texttt{of None}.\ \forall l \in \mathsf{updated}(\bar e).\ l := \texttt{None}; 0 \quad \texttt{Some}\ y.\ y, \bar\mu\rangle \Downarrow \langle 0, \bar\mu'\rangle}\ \text{E-CASE-INL}}{\langle \texttt{while (case}\ x\ \texttt{of None}.\ \forall l \in \mathsf{updated}(\bar e). l := \texttt{None}; 0 \quad \texttt{Some}\ y.\ y)\ \texttt{do}\ \bar e, \bar\mu\rangle \Downarrow \langle (), \bar\mu'\rangle}$$

As $\mu_f = \mu$, it remains to show that $\mu \geq_\Gamma \bar\mu'$. Since for all $l \in \mathsf{updated}(\bar e)$, $\Gamma(l) = \sigma^H$, for some σ, and for all $l \in dom(\mu_f) \setminus \mathsf{updated}(\bar e)$, $\mu_f(l) = \mu(l) \geq_{\Gamma(l)} \bar\mu(l) = \bar\mu_f(l)$, $\mu_f \geq_\Gamma \bar\mu_f$.

Case: T-UPGRADE. From i.h. applied to: (1) $pc \vdash \Gamma\{e : \sigma^L \rightsquigarrow \bar e\}\Gamma$, (2) $\langle e, \mu\rangle \Downarrow \langle v, \mu'\rangle$, and (3) $\mu \geq_\Gamma \bar\mu$, we get that $\langle \bar e, \bar\mu\rangle \Downarrow \langle \bar v, \bar\mu'\rangle$, with $v \geq_{\sigma^L} \bar v$ and $\mu' \geq_{\Gamma'} \bar\mu'$.

By applying rule E-INR we get that $\bar v_f = \texttt{Some}\ \bar v$ and $\bar\mu_f = \bar\mu'$:

$$\dfrac{\langle \bar e, \bar\mu\rangle \Downarrow \langle \bar v, \bar\mu'\rangle}{\langle \texttt{Some}\ \bar e, \bar\mu\rangle \Downarrow \langle \texttt{Some}\ \bar v, \bar\mu'\rangle}\ \text{E-INR}$$

Hence $v \geq_{\sigma^H} \texttt{Some}\ \bar v$ (from Figures 4.7 and 4.8) and $\mu' \geq_{\Gamma'} \bar\mu'$. ∎

Theorem 37 (Monotonicity of $\bar e$)**.** *For any source program e, typing context Γ, program counter pc, and memories $\bar\mu$, and $\bar\mu'$ such that $pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar e\}\Gamma'$, and $\bar\mu \geq \bar\mu'$, if $\langle \bar e, \bar\mu\rangle \Downarrow \langle \bar v_f, \bar\mu_f\rangle$ and $\langle \bar e, \bar\mu'\rangle \Downarrow \langle \bar v'_f, \bar\mu'_f\rangle$, then $\bar v_f \geq \bar v'_f$ and $\bar\mu_f \geq \bar\mu'_f$.*

The proof of this theorem relies on the following helper lemma.

Lemma 38 (Helper)**.** *For any source program e, typing environment Γ, and stores $\bar\mu$ and $\bar\mu'$ such that $\bar\mu \geq \bar\mu'$, if $H \vdash \Gamma\{e : \tau \rightsquigarrow \bar e\}\Gamma'$ and $\langle \bar e, \bar\mu\rangle \Downarrow \langle \bar v_f, \bar\mu_f\rangle$, then $\langle \forall l \in \mathsf{updated}(\bar e).\ l := \texttt{None}, \bar\mu'\rangle \Downarrow \langle (), \bar\mu'_f\rangle$ and $\bar\mu_f \geq \bar\mu'_f$.*

Proof. By induction on the structure of e. The proof is similar with the one for Lemma 36 and we do not include it here.

Case: $e = v$. From rule T-VAL, $\bar e = v$. Then $\mathsf{updated}(v) = \emptyset$. From E-VAL, $\bar\mu_f = \bar\mu$. As $\bar\mu'_f = \bar\mu'$, it follows that $\bar\mu_f \geq \bar\mu'_f$.

Case: $e = l$. From rule T-VARLOC, $\bar e = l$. Then $\mathsf{updated}(l) = \emptyset$. From E-VARLOC, $\bar\mu_f = \bar\mu$. As $\bar\mu'_f = \bar\mu'$, it follows that $\bar\mu_f \geq \bar\mu'_f$.

Case: $e = e_1 \oplus e_2$. From rule T-EXP-*, $\mathsf{updated}(\bar e) = \mathsf{updated}(\bar e_1, \bar e_2)$.

From i.h. applied to (1) $\langle e_1, \mu\rangle \Downarrow \langle n_1, \mu_1\rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar e_1).\ l := \texttt{None};\ 0, \bar\mu\rangle \Downarrow \langle 0, \bar\mu_1\rangle$, and (3) $\mu \geq \bar\mu$, we get $\mu_1 \geq \bar\mu_1$.

From i.h. applied to (4) $\langle e_2, \mu_1\rangle \Downarrow \langle n_2, \mu_2\rangle$, (5) $\langle \forall l \in \mathsf{updated}(\bar e_2).\ l := \texttt{None};\ 0, \bar\mu_1\rangle \Downarrow \langle 0, \bar\mu_2\rangle$, and (6) $\mu_1 \geq \bar\mu_1$, we get $\mu_2 \geq \bar\mu_2$.

Thus $\langle \forall l \in \mathsf{updated}(\bar e_1, \bar e_2).\ l := \texttt{None}; 0, \bar\mu\rangle \Downarrow \langle 0, \bar\mu_2\rangle$, with $\bar\mu_f = \bar\mu_2$. As $\mu_f = \mu_2$, it follows that $\mu_f \geq \bar\mu_f$.

Case: $e = l := e'$. From rule T-ASSIGN, $\bar e = l := \bar e'$. Then $\mathsf{updated}(\bar e) = \mathsf{updated}(\bar e', l)$.

From i.h. applied to (1) $\langle e',\mu\rangle \Downarrow \langle v,\mu'\rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar{e}').\ l := \texttt{None};0,\bar{\mu}\rangle \Downarrow \langle 0,\bar{\mu}'\rangle$, and (3) $\mu \geq \bar{\mu}$, we get $\mu' \geq \bar{\mu}'$. I.e. $\langle \forall l' \in \mathsf{updated}(\bar{e}',l).\ l' := \texttt{None};0,\bar{\mu}\rangle \Downarrow \langle 0,\bar{\mu}'[l \mapsto \texttt{None}]\rangle$.

As $v \geq \texttt{None}$, it follows that $(\mu'[l \mapsto v])(l) \geq (\bar{\mu}'[l \mapsto \texttt{None}])(l)$. As $\mu_f = \mu'[l \mapsto v]$ and $\bar{\mu}_f = \bar{\mu}'[l \mapsto \texttt{None}]$, it follows that $\mu_f \geq \bar{\mu}_f$.

Case: $e = \texttt{inl}\ e'$**.** From rule T-INL, $\bar{e} = \texttt{inl}\ \bar{e}'$. Then $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}')$.

From i.h. applied to (1) $\langle e',\mu\rangle \Downarrow \langle v,\mu'\rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar{e}').\ l := \texttt{None};0\rangle \Downarrow \langle 0,\bar{\mu}'\rangle$, and (3) $\mu \geq \bar{\mu}$, we get $\mu' \geq \bar{\mu}'$. Thus $\langle \forall l \in \mathsf{updated}(\texttt{inl}\ \bar{e}').\ l := \texttt{None};0,\bar{\mu}\rangle \Downarrow \langle 0,\bar{\mu}'\rangle$. As $\mu_f = \mu'$ and $\bar{\mu}_f = \bar{\mu}'$, it follows that $\mu_f \geq \bar{\mu}_f$.

Case: $e = \texttt{inr}\ e'$**.** Similar to the previous case.

Case: $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$**.** From rule T-LET, $\bar{e} = \texttt{let}\ x = \bar{e}_1\ \texttt{in}\ \bar{e}_2$. Then $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}_1,\bar{e}_2)$.

From i.h. applied to (1) $\langle e_1,\mu\rangle \Downarrow \langle v_1,\mu_1\rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar{e}_1).\ l := \texttt{None};0,\bar{\mu}\rangle \Downarrow \langle 0,\bar{\mu}_1\rangle$, and (3) $\mu \geq \bar{\mu}$, we get $\mu_1 \geq \bar{\mu}_1$.

Let $\bar{v}_1$ be such that $v_1 \geq \bar{v}_1$. Hence, $\mu_1 \cup \{x \mapsto v_1\} \geq \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$.

From i.h. applied to (4) $\langle e_2,\mu_1 \cup \{x \mapsto v_1\}\rangle \Downarrow \langle v_2,\mu_2\rangle$, (5) $\langle \forall l \in \mathsf{updated}(\bar{e}_2).\ l := \texttt{None};0,\bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}\rangle \Downarrow \langle 0,\bar{\mu}_2\rangle$, and (6) $\mu_1 \cup \{x \mapsto v_1\} \geq \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$, we get $\mu_2 \geq \bar{\mu}_2$. I.e. $\mu_2 \setminus \{x\} \geq \bar{\mu}_2 \setminus \{x\}$.

But $x \notin \mathsf{updated}(\bar{e}_2)$. Thus $\langle \forall l \in \mathsf{updated}(\bar{e}_2).\ l := \texttt{None};0,\bar{\mu}_1\rangle \Downarrow \langle 0,\bar{\mu}_2 \setminus \{x\}\rangle$. Hence $\langle \forall l \in \mathsf{updated}(\bar{e}_1,\bar{e}_2).\ l := \texttt{None};0,\bar{\mu}\rangle \Downarrow \langle 0,\bar{\mu}_2 \setminus \{x\}\rangle$. As $\mu_f = \mu_2 \setminus \{x\}$ and $\bar{\mu}_f = \bar{\mu}_2 \setminus \{x\}$, it follows that $\mu_f \geq \bar{\mu}_f$.

Case: $e = \texttt{case}\ e'\ \texttt{of}\ \texttt{inl}\ x.\ e_1\quad \texttt{inr}\ x.\ e_2$**.** From rule T-CASE-*, $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}',\bar{e}_1,\bar{e}_2)$.

From i.h. applied to (1) $\langle e',\mu\rangle \Downarrow \langle v,\mu_1\rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar{e}').\ l := \texttt{None};0,\bar{\mu}\rangle \Downarrow \langle 0,\bar{\mu}_1\rangle$, and (3) $\mu \geq \bar{\mu}$, we get $\mu_1 \geq \bar{\mu}_1$.

We assume e evaluates under rule E-CASE-INL. (For the case when it evaluates under rule E-CASE-INR, the proof is analoguous.) Hence, $v = \texttt{inl}\ v_1$. Let $\bar{v}_1$ be such that $v_1 \geq \bar{v}_1$. Then $\mu_1 \cup \{x \mapsto v_1\} \geq \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$.

From i.h. applied to (4) $\langle e_1,\mu_1 \cup \{x \mapsto v_1\}\rangle \Downarrow \langle v_2,\mu_2\rangle$, (5) $\langle \forall l \in \mathsf{updated}(\bar{e}_1).\ l := \texttt{None};0,\bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}\rangle \Downarrow \langle 0,\bar{\mu}_2\rangle$, and (6) $\mu_1 \cup \{x \mapsto v_1\} \geq \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\}$, we get $\mu_2 \geq \bar{\mu}_2$. Hence, $\mu_2 \setminus \{x\} \geq \bar{\mu}_2 \setminus \{x\}$.

Since $x \notin \mathsf{updated}(\bar{e}_1)$, it follows that $\langle \forall l \in \mathsf{updated}(\bar{e}_1).\ l := \texttt{None};0,\bar{\mu}_1\rangle \Downarrow \langle 0,\bar{\mu}_2 \setminus \{x\}\rangle$.

From $\langle \forall l \in \mathsf{updated}(\bar{e}_2).\ l := \texttt{None};0,\bar{\mu}_2 \setminus \{x\}\rangle \Downarrow \langle 0,\bar{\mu}_3\rangle$, $\bar{\mu}_f = \bar{\mu}_3$. But $\mu_f = \mu_2 \setminus \{x\}$. We show below that $\mu_f = \mu_2 \setminus \{x\} \geq \bar{\mu}_3 = \bar{\mu}_f$.

For all memory locations $l \in \mathsf{updated}(\bar{e}_2)$, $\mu_f(l) \geq \bar{\mu}_f(l) = \texttt{None}$. For all memory locations $l \in dom(\mu_f) \setminus \mathsf{updated}(\bar{e}_2)$, $\mu_f(l) = (\mu_2 \setminus \{x\})(l) \geq (\bar{\mu}_2 \setminus \{x\})(l) = \bar{\mu}_f(l)$. Thus $\mu_f \geq \bar{\mu}_f$.

Case: $e = \texttt{while}\ x\ \mathsf{do}\ e'$**.** From rule T-WHILE-*, $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}')$.

We distinguish two subcases:

Subcase: $\mu(x) = 0$**.** Then rule E-WHILE-FALSE applies and $\mu_f = \mu$.

From $\langle \forall l \in \mathsf{updated}(\bar{e}').\ l := \texttt{None};0,\bar{\mu}\rangle \Downarrow \langle 0,\bar{\mu}'\rangle$, $\bar{\mu}_f = \bar{\mu}'$. We show below that $\mu_f \geq \bar{\mu}_f$.

For all memory locations $l \in \mathsf{updated}(\bar{e}')$, $\mu_f(l) \geq \bar{\mu}_f(l) = \texttt{None}$. For all memory locations $l \in dom(\mu_f) \setminus \mathsf{updated}(\bar{e}')$, $\mu_f(l) = \mu(l) \geq \bar{\mu}(l) = \bar{\mu}_f(l)$. Hence $\mu_f \geq \bar{\mu}_f$.

Subcase: $\mu(x) \neq 0$**.** Then rule E-WHILE-TRUE applies.

$$\frac{\mu(x) \neq 0 \qquad \langle e',\mu\rangle \Downarrow \langle v,\mu'\rangle \qquad \langle \texttt{while}\ x\ \mathsf{do}\ e',\mu'\rangle \Downarrow \langle v',\mu''\rangle}{\langle \texttt{while}\ x\ \mathsf{do}\ e',\mu\rangle \Downarrow \langle v',\mu''\rangle}\ \text{E-WHILE-TRUE}$$

From i.h. applied to (1) $\langle e',\mu\rangle \Downarrow \langle v,\mu'\rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar{e}').\ l := \mathtt{None};0,\bar{\mu}\rangle \Downarrow \langle 0,\bar{\mu}'\rangle$, and (3) $\mu \geq \bar{\mu}$, we get $\mu' \geq \bar{\mu}'$.

Since $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}')$, $\langle \forall l \in \mathsf{updated}(\bar{e}).\ l := \mathtt{None};0,\bar{\mu}'\rangle \Downarrow \langle 0,\bar{\mu}'\rangle$ and $\bar{\mu}_f = \bar{\mu}'$. Also $\mu_f = \mu''$. For all memory locations $l \in dom(\mu_f) \setminus \mathsf{updated}(\bar{e}')$, $\mu_f(l) = \mu'(l) \geq \bar{\mu}'(l) = \bar{\mu}_f(l)$. For all memory locations $l \in \mathsf{updated}(\bar{e}')$, $\bar{\mu}(f) = \mathtt{None}$. Hence $\mu_f(l) \geq \bar{\mu}_f(l)$. Hence $\mu_f \geq \bar{\mu}_f$.

Case: $e = \bullet e'$**.** From rule T-UPGRADE, $\bar{e} = \mathtt{Some}\ \bar{e}'$. Then $\mathsf{updated}(\bar{e}) = \mathsf{updated}(\bar{e}')$.

From i.h. applied to (1) $\langle e',\mu\rangle \Downarrow \langle v,\mu'\rangle$, (2) $\langle \forall l \in \mathsf{updated}(\bar{e}').\ l := \mathtt{None};0,\bar{\mu}\rangle \Downarrow \langle 0,\bar{\mu}'\rangle$, and (3) $\mu \geq \bar{\mu}$, we get $\mu' \geq \bar{\mu}'$. Thus $\langle \forall l \in \mathsf{updated}(\mathtt{Some}\ \bar{e}').\ l := \mathtt{None};0\rangle \Downarrow \langle 0,\bar{\mu}'\rangle$. As $\mu_f = \mu'$ and $\bar{\mu}_f = \bar{\mu}'$, it follows that $\mu_f \geq \bar{\mu}_f$. ■

Proof of Theorem 37. By induction on the derivation of the type-directed transformation and on the derivation of the evaluation relation $\langle \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v}_f,\bar{\mu}_f\rangle$. We present the proof only for the more important cases.

Case T-CASE-L. The transformation derivation is of the form:

$$\dfrac{pc \vdash \Gamma\{e : (\tau_1+\tau_2)^L \rightsquigarrow \bar{e}\}\Gamma' \qquad pc \vdash \Gamma', x:\tau_i\{e_i : \tau \rightsquigarrow \bar{e}_i\}\Gamma_i \qquad i = 1,2 \qquad S_1 = \Gamma_1 \setminus \Gamma_2 \qquad S_2 = \Gamma_2 \setminus \Gamma_1}{\begin{array}{l} pc \vdash \Gamma\{\mathtt{case}\ e\ \mathtt{of}\ \mathtt{inl}\ x.\ e_1 \quad \mathtt{inr}\ x.\ e_2 : \tau \rightsquigarrow \\ \qquad \mathtt{case}\ \bar{e}\ \mathtt{of}\ \mathtt{inl}\ x.\ \mathtt{let}\ y = \bar{e}_1\ \mathtt{in}\ \mathsf{upgrade}(S_2);y \quad \mathtt{inr}\ x.\ \mathtt{let}\ y = \bar{e}_2\ \mathtt{in}\ \mathsf{upgrade}(S_1);y \bigsqcup \Gamma_i \setminus \{x\} \end{array}}\ \text{T-CASE-L}$$

From i.h. and Theorem 35, $\langle \bar{e},\bar{\mu}\rangle \Downarrow \langle \bar{v},\bar{\mu}_1\rangle$, $\langle \bar{e},\bar{\mu}'\rangle \Downarrow \langle \bar{v},\bar{\mu}'_1\rangle$ and $\bar{\mu}_1 \geq \bar{\mu}'_1$. We assume $\bar{e}$ evaluates under rule E-CASE-INL, i.e. $\bar{v} = \mathtt{inl}\ v$. For the case when $\bar{e}$ evaluates under rule E-CASE-INR, the proof is analogous.

We are left to show that if $\langle \mathtt{let}\ y = \bar{e}_1;\mathsf{upgrade}(S_2);y\ \mathtt{in}\ ,\bar{\mu}_1 \cup \{x \mapsto v\}\rangle \Downarrow \langle \bar{v}_f,\bar{\mu}_f\rangle$ and $\langle \mathtt{let}\ y = \bar{e}_1;\mathsf{upgrade}(S_2);y\ \mathtt{in}\ ,\bar{\mu}'_1 \cup \{x \mapsto v\}\rangle \Downarrow \langle \bar{v}'_f,\bar{\mu}'_f\rangle$, then $\bar{v}_f \geq \bar{v}'_f$ and $\bar{\mu}_f \geq \bar{\mu}'_f$.

As $\bar{\mu}_1 \geq \bar{\mu}'_1$ and $v = v$, $\bar{\mu}_1 \cup \{x \mapsto v\} \geq \bar{\mu}'_1 \cup \{x \mapsto v\}$.

$$\dfrac{}{\langle \bar{e}_1,\bar{\mu}_1 \cup \{x \mapsto v\}\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_2\rangle}\ \text{I.H.}$$

$$\dfrac{\dfrac{\dfrac{}{\langle \mathsf{upgrade}(S_2),\bar{\mu}_2 \cup \{y \mapsto \bar{v}_2\}\rangle \Downarrow \langle (),\bar{\mu}_3\rangle}\ \text{E-ASSIGN}^* \qquad \dfrac{\bar{\mu}_3(y) = \bar{v}_2}{\langle y,\bar{\mu}_3\rangle \Downarrow \langle \bar{v}_2,\bar{\mu}_3\rangle}\ \text{E-VAR}}{\langle \mathsf{upgrade}(S_2);y,\bar{\mu}_3\rangle \Downarrow \langle \bar{v}_f,\bar{\mu}_3\rangle}\ \text{E-SEQ}}{\langle \mathtt{let}\ y = \bar{e}_1;\mathsf{upgrade}(S_2);y\ \mathtt{in}\ ,\bar{\mu}_1 \cup \{x \mapsto v\}\rangle \Downarrow \langle \bar{v}_f,\bar{\mu}_3 \setminus \{y\}\rangle}\ \text{E-LET}$$

From i.h., $\bar{v}_2 \geq \bar{v}'_2$ and $\bar{\mu}_2 \geq \bar{\mu}'_2$. Hence $\bar{\mu}_2 \cup \{y \mapsto v_2\} \geq \bar{\mu}'_2 \cup \{y \mapsto v'_2\}$. For all $l \in \mathsf{upgrade}(S_2)$, $\bar{\mu}_2(l) = \bar{\mu}'_2(l)$. Thus $\mathtt{Some}\ \bar{\mu}_2(l) = \mathtt{Some}\ \bar{\mu}'_2(l)$. Hence $\bar{\mu}_3 \geq \bar{\mu}'_3$ and $\bar{\mu}_3 \setminus \{y\} \geq \bar{\mu}'_3 \setminus \{y\}$. As $y \notin S_2$, $\bar{\mu}_3(y) = \bar{v}_2 = \bar{v}_f$ and $\bar{\mu}'_3(y) = \bar{v}'_2 = \bar{v}'_f$.

Case: T-CASE-H. The transformation derivation is of the form:

$$\dfrac{pc \vdash \Gamma\{e : (\tau_1+\tau_2)^H \rightsquigarrow \bar{e}\}\Gamma' \qquad H \vdash \Gamma', x:\tau_i\{e_i : \tau \rightsquigarrow \bar{e}_i\}\Gamma_i \qquad i = 1,2 \qquad S_1 = \Gamma_1 \setminus \Gamma_2 \qquad S_2 = \Gamma_2 \setminus \Gamma_1}{\begin{array}{l} pc \vdash \Gamma\{\mathtt{case}\ e\ \mathtt{of}\ \mathtt{inl}\ x.\ e_1 \quad \mathtt{inr}\ x.\ e_2 : \tau \rightsquigarrow \\ \qquad \mathtt{case}\ \bar{e}\ \mathtt{of}\ \mathtt{None}.\ (\forall l \in \mathsf{updated}(\bar{e}_1,\bar{e}_2).\ l := \mathtt{None});\ \mathtt{None} \\ \qquad\qquad \mathtt{Some}\ x'.\ \mathtt{case}\ x'\ \mathtt{of}\ \mathtt{inl}\ x.\ \mathtt{let}\ y = \bar{e}_1\ \mathtt{in}\ \mathsf{upgrade}(S_2);y \\ \qquad\qquad\qquad \mathtt{inr}\ x.\ \mathtt{let}\ y = \bar{e}_2\ \mathtt{in}\ \mathsf{upgrade}(S_1);y\} \bigsqcup \Gamma_i \setminus \{x\} \end{array}}\ \text{T-CASE-H}$$

From i.h., $\langle \bar{e}, \bar{\mu} \rangle \Downarrow \langle \bar{v}, \bar{\mu}_1 \rangle$, $\langle \bar{e}, \bar{\mu}' \rangle \Downarrow \langle \bar{v}', \bar{\mu}'_1 \rangle$, with $\bar{v} \geq \bar{v}'$ and $\bar{\mu}_1 \geq \bar{\mu}'_1$. We distinguish three cases:

Subcase: $\bar{v} = \texttt{Some}\ \bar{v}_1$ **and** $\bar{v}' = \texttt{Some}\ \bar{v}'_1$**, with** $\bar{v}_1 \geq \bar{v}'_1$**.**

The proof is similar with the proof of the corresponding case in Theorem 35.

Subcase: $\bar{v} = \texttt{Some}\ \bar{v}_1$ **and** $\bar{v}' = \texttt{None}$**.**

$\bar{e}$ evaluates in memory $\bar{\mu}$ under rule E-CASE-INR, while in memory $\bar{\mu}'$ it evaluates under rule E-CASE-INL. We assume $\bar{v}_1 = \texttt{inl}\ v_1$, for the case when $\bar{v}_1 = \texttt{inr}\ v_1$, the proof is analogous.

Thus, we are left to show that, if $\langle \texttt{let}\ y = \bar{e}_1\ \texttt{in}\ \mathsf{upgrade}(S_2); y, \bar{\mu} \cup \{x' \mapsto v_1\} \rangle \Downarrow \langle \bar{v}_f, \bar{\mu}_f \rangle$ and $\langle \forall l \in \mathsf{updated}(\bar{e}_1, \bar{e}_2).\ l := \texttt{None};\ \texttt{None}, \bar{\mu}' \rangle \Downarrow \langle \bar{v}'_f, \bar{\mu}'_f \rangle$, then $\bar{v}_f \geq \bar{v}'_f$ and $\bar{\mu}_f \geq \bar{\mu}'_f$.

But $\mathsf{updated}(\bar{e}_1, \bar{e}_2) = \mathsf{updated}(\bar{e}_1) \cup S_2$. From Lemma 38 applied to $\langle \bar{e}_1, \bar{\mu}_1 \cup \{x \mapsto \bar{v}_1\} \rangle \Downarrow \langle \bar{v}_2, \bar{\mu}_2 \rangle$ and $\langle \forall l \in \mathsf{updated}(\bar{e}_1). l := \texttt{None}, \bar{\mu}'_1 \rangle \Downarrow \langle (), \bar{\mu}'_2 \rangle$, we get that $\bar{\mu}_2 \setminus \{x\} \geq \bar{\mu}'_2 \setminus \{x\}$.

For all $l \in S_2$, $\Gamma'(l) = \sigma^L$, for some σ. $\mathsf{upgrade}(S_2)$ means that for all $l \in S_2$, $\bar{\mu}_2\{l \mapsto \texttt{Some}\ \bar{\mu}_2(l)\}$. $\forall l \in S_2. l := \texttt{None}$ means that for all $l \in S_2$, $\bar{\mu}'_2\{l \mapsto \texttt{None}\}$. Hence, for all $l \in S_2$, $\bar{\mu}_f(l) = \bar{\mu}_2\{l \mapsto \bar{\mu}_2(l)\}(l) \geq \bar{\mu}'_2\{l \mapsto \texttt{None}\}(l) = \bar{\mu}'_f(l)$. For all $l \notin S_2$, $\bar{\mu}_f(l) = \bar{\mu}_2(l) \geq \bar{\mu}'_2(l) = \bar{\mu}'_f(l)$. Hence $\bar{\mu}_f \geq \bar{\mu}'_f$. Also, $\bar{v}'_f = \texttt{None}$, hence $\bar{v}_f \geq \bar{v}'_f$.

Subcase: $\bar{v} = \texttt{None}$ **and** $\bar{v}' = \texttt{None}$**.**

$\bar{v}_f = \bar{v}'_f = \texttt{None}$. $\bar{\mu}_1 \geq \bar{\mu}'_1$, hence $\bar{\mu}_f \geq \bar{\mu}'_f$.

Case T-WHILE-L. The transformation derivation is of the form:

$$\frac{\Gamma(x) = \texttt{Int}^L \qquad pc \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma}{pc \vdash \Gamma\{\texttt{while}\ x\ \texttt{do}\ e : \texttt{Unit}^L \rightsquigarrow \texttt{while}\ x\ \texttt{do}\ \bar{e}\}\Gamma}\ \text{T-WHILE-L}$$

As $\Gamma(x) = \texttt{Int}^L$, it follows that $\bar{\mu}(x) = \bar{\mu}'(x)$. We distinguish two subcases:

Subcase: $\bar{\mu}(x) \neq 0$ (rule E-WHILE-TRUE)

From i.h., $\langle \bar{e}, \bar{\mu} \rangle \Downarrow \langle \bar{v}_1, \bar{\mu}_1 \rangle$ and $\langle \bar{e}, \bar{\mu}' \rangle \Downarrow \langle \bar{v}'_1, \bar{\mu}'_1 \rangle$ with $\bar{v}_1 \geq \bar{v}'_1$ and $\bar{\mu}_1 \geq \bar{\mu}'_1$. Also from i.h., $\langle \texttt{while}\ x\ \texttt{do}\ \bar{e}, \bar{\mu}_1 \rangle \Downarrow \langle \bar{v}_2, \bar{\mu}_2 \rangle$ and $\langle \texttt{while}\ x\ \texttt{do}\ e, \bar{\mu}'_1 \rangle \Downarrow \langle \bar{v}'_2, \bar{\mu}'_2 \rangle$ with $\bar{v}_2 \geq \bar{v}'_2$ and $\bar{\mu}_2 \geq \bar{\mu}'_2$.

As $\bar{v}_f = \bar{v}_2$ and $\bar{v}'_f = \bar{v}'_2$, it follows that $\bar{v}_f \geq \bar{v}'_f$. Similarly, $\bar{\mu}_f = \bar{\mu}_2$ and $\bar{\mu}'_f = \bar{\mu}'_2$, hence $\bar{\mu}_f \geq \bar{\mu}'_f$.

Subcase: $\bar{\mu}(x) = 0$ (rule E-WHILE-FALSE)

In this case $\bar{v}_f = \bar{v}'_f = ()$ and $\bar{\mu}_f = \bar{\mu} \geq \bar{\mu}' = \bar{\mu}'_f$.

Case T-WHILE-H. The transformation derivation is of the form:

$$\text{T-WHILE-H}\quad \frac{\Gamma(x) = \texttt{Int}^H \qquad H \vdash \Gamma\{e : \tau \rightsquigarrow \bar{e}\}\Gamma}{\begin{array}{l} pc \vdash \Gamma\{\texttt{while}\ x\ \texttt{do}\ e : \texttt{Unit}^L \rightsquigarrow \\ \qquad \texttt{while}\ (\texttt{case}\ x\ \texttt{of None.}\ (\forall l \in \mathsf{updated}(\bar{e}).\ l := \texttt{None};)\ 0 \quad \texttt{Some}\ y.\ y)\ \texttt{do}\ \bar{e}\}\Gamma \end{array}}$$

We distinguish three subcases:

Subcase: $\bar{\mu}(x) = \texttt{Some}\ \bar{v}$ **and** $\bar{\mu}'(x) = \texttt{Some}\ \bar{v}$

The proof is similar with the proof of the corresponding case in Theorem 35.

Subcase: $\bar{\mu}(x) = \texttt{Some}\ \bar{v}$ **and** $\bar{\mu}'(x) = \texttt{None}$

- $\bar{v} \neq 0$

  Follows from Lemma 38.

- $\bar{v} = 0$

  $\bar{v}_f = ()$ and $\bar{v}_f = \bar{\mu}$. Also $\bar{v}'_f = ()$. As $\bar{\mu} \geq \bar{\mu}'$ and for all $l \in \mathsf{updated}(\bar{e})$, $\bar{\mu} = \bar{\mu}_f \geq \texttt{None} = \bar{\mu}'_f(l)$. Hence $\bar{\mu}_f \geq \bar{\mu}'_f$.

Subcase: $\bar{\mu}(x) = \texttt{None}$ **and** $\bar{\mu}(x) = \texttt{None}$

$\bar{v}_f = () = \bar{v}'_f$. As $\bar{\mu} \geq \bar{\mu}'$ and for all $l \in \mathsf{updated}(\bar{e})$, $\bar{\mu}\{l \mapsto \texttt{None}\}(l) \geq \bar{\mu}'\{l \mapsto \texttt{None}\}(l)$, it follows that $\bar{\mu}_f \geq \bar{\mu}'_f$. ■

Bibliography

[1] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, pages 333–348, 2008.

[2] Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, pages 207–221, 2007.

[3] Aslan Askarov and Andrei Sabelfeld. Localized delimited release: Combining the what and where dimensions of information release. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, PLAS '07, pages 53–60, 2007.

[4] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, pages 113–124, 2009.

[5] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 165–178, 2012.

[6] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '13, pages 15–26, 2013.

[7] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 339–353, 2008.

[8] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. Reactive non-interference for a browser model. In *5th International Conference on Network and System Security, NSS 2011*, pages 97–104, 2011.

[9] Niklas Broberg and David Sands. Paralocks: Role-based information flow control and beyond. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 431–444, 2010.

[10] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing information leaks through shadow executions. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 322–331, 2008.

[11] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004*, pages 198–209, 2004.

[12] Stephen Chong and Andrew C. Myers. Language-based information erasure. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005),*, pages 241–254, 2005.

[13] Stephen Chong and Andrew C. Myers. End-to-end enforcement of erasure and declassification. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008*, pages 98–111, 2008.

[14] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008*, pages 51–65, 2008.

[15] David Costanzo and Zhong Shao. A separation logic for enforcing declarative information flow control policies. In *Principles of Security and Trust - 3rd International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 179–198, 2014.

[16] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P 2010*, pages 109–124, 2010.

[17] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 79–90, 2006.

[18] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 413–428, 2011.

[19] Tejas Khatiwala, Raj Swaminathan, and V. N. Venkatakrishnan. Data sandboxing: A technique for enforcing confidentiality policies. In *22nd Annual Computer Security Applications Conference (ACSAC 2006),*, pages 223–234, 2006.

[20] Lap-Chung Lam and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *22nd Annual Computer Security Applications Conference (ACSAC 2006),*, pages 463–472, 2006.

[21] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, pages 158–170, 2005.

[22] Jonas Magazinius, Aslan Askarov, and Andrei Sabelfeld. Decentralized delimited release. In *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, . Proceedings*, pages 220–237, 2011.

[23] Scott Moore, Aslan Askarov, and Stephen Chong. Precise enforcement of progress-sensitive security. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 881–893, 2012.

[24] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 228–241, 1999.

[25] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, pages 129–142, 1997.

[26] Minh Ngo, Fabio Massacci, Dimiter Milushev, and Frank Piessens. Runtime enforcement of security policies on black box reactive programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, pages 43–54, 2015.

[27] Willard Rafnsson, Daniel Hedin, and Andrei Sabelfeld. Securing interactive programs. In *25th IEEE Computer Security Foundations Symposium, CSF 2012,*, pages 293–307, 2012.

[28] Willard Rafnsson and Andrei Sabelfeld. Secure multi-execution: fine-grained, declassification-aware, and transparent. In *2013 IEEE 26th Computer Security Foundations Symposium, 2013*, pages 33–48, 2013.

[29] Willard Rafnsson and Andrei Sabelfeld. Secure multi-execution: fine-grained, declassification-aware, and transparent. *Journal of Computer Security*, 24(1):39–90, 2016.

[30] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[31] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003*, pages 174–191, 2003.

[32] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005),*, pages 255–269, 2005.

[33] Nikhil Swamy and Michael Hicks. Verified enforcement of stateful information release policies. In *Proceedings of the 2008 Workshop on Programming Languages and Analysis for Security, PLAS 2008*, pages 21–32, 2008.

[34] Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. Stateful declassification policies for event-driven programs. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014*, pages 293–307, 2014.

[35] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, April 14-18, 1997, Proceedings*, pages 607–621, 1997.

[36] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*, pages 85–96, 2012.

[37] Dante Zanarini, Mauro Jaskelioff, and Alejandro Russo. Precise enforcement of confidentiality for reactive systems. In *2013 IEEE 26th Computer Security Foundations Symposium, 2013*, pages 18–32, 2013.

[38] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference (extended abstract). In *Formal Aspects in Security and Trust: Second IFIP TC1 WG1.7 Workshop on Formal Aspects in Security and Trust (FAST), an event of the 18th IFIP World Computer Congress, August 22-27, 2004, Toulouse, France*, pages 27–40, 2004.

[39] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Sec.*, 6(2-3):67–84, 2007.