MODULAR COMPILERS AND THEIR CORRECTNESS PROOFS

BY

WILLIAM LAWRENCE HARRISON

B.A., University of California, Berkeley 1986
M.S., University of California, Davis 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

For Erik Claus Cordes

Abstract

This thesis explores the construction and correctness of modular compilers. *Modular compilation* is a compiler construction technique allowing the construction of compilers for high-level programming languages from reusable compiler building blocks. Modular compilers are defined in terms of denotational semantics based on monads, monad transformers, and a new model of staged computation called *metacomputations*. A novel form of denotational specification called *observational program specification* and related proof techniques are developed to assist in modular compiler verification. It will be demonstrated that the modular compilation framework provides both a level of modularity in compiler proofs as well as a useful organizing principle for such proofs.

Acknowledgments

I wish to thank my family and friends for their stalwart support throughout my time as a graduate student. I owe a special debt of gratitude to my parents, George and Mary Harrison, for their infinite patience, kindness, and generosity. My brother, Robert Harrison, always had sage advice on negotiating academic whitewater. My friend, Erik Cordes, inspired me to undertake this difficult doctoral challenge in the first place.

I must also thank my friend and adviser, Samuel Kamin, for patient insight, having high academic standards with good humor, and occasionally suspending disbelief. I wish to thank the other members of my thesis committee for their help as well: Uday Reddy, Jens Palsberg, and John Gray. I thank the past and present members of the UIUC functional language community for listening to many rambling presentations of earlier versions of this work: Joel Jones, Mattox Beckman, Howard Huang, Miranda Callahan, Jonathan Springer, An Le, Lars Clausen, Robert Hasker, Hanseok Yang, and Brian Howard.

Table of Contents

List of Figures

List of Abbreviations

RCBB Reusable Compiler Building Block

MCG Monadic Code Generator

Chapter 1

Introduction

Compiler complexity is one of the most enduring problems in programming language research. Traditional, hand-written compilers are large, complicated programs which take a long time to write. Given the amount of work that goes into the creation of compilers, it is desirable to reuse as much of this effort as possible. But, the monolithic structure of traditional compilers makes it difficult to modify them, reuse parts of them in new compilers, and prove them correct.

As an example of a traditional, handwritten compiler, consider the GNU GCC-1750 (version 1.0) C++ compiler. The source code for this compiler has 278,949 lines of code in 168 separate files. Certain questions naturally arise about the modularity and correctness of this compiler:

- How would one add or delete a source language feature from this compiler?
- How would one compile one source language feature (e.g., expressions) in a new compiler just as in GCC? Which of the 278,949 lines from which of the 168 files would you choose? In other words, is there any advantage to having the GCC source code when constructing another compiler?
- How would one prove GCC correct? Is there any intelligible relationship between the immense GCC compiler and the semantics of the language it compiles?

The complex, monolithic structure of traditional compilers like GCC makes answering these questions practically impossible.

These problems would be eliminated if compilers could be split up and built from *reusable compiler building blocks* (RCBBs) which compile distinct source language features separately and

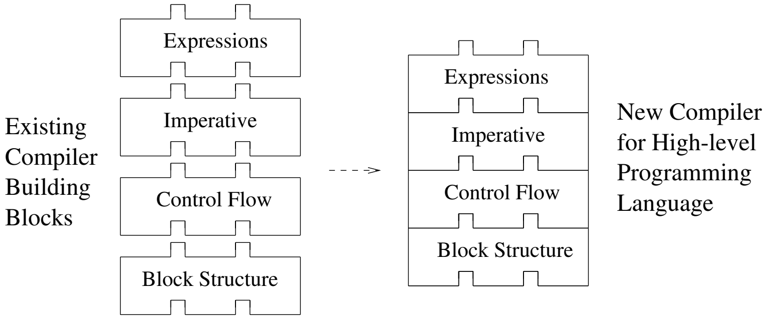


Figure 1.1: Modular Compilers: Existing compiler building blocks combine to make new compiler

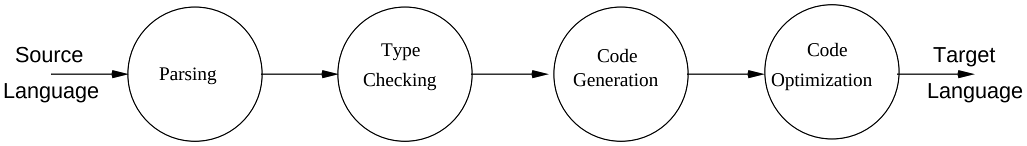


Figure 1.2: Traditional Compilation is Separated into Phases

can then be reassembled *easily* as needed to create new compilers for bigger, high-level programming languages. This novel approach to compiler construction is called *modular compilation*, and it is illustrated in Figure 1.1.

Modular compilers have the following advantages over traditionally constructed compilers:

- They are more easily modified. To change the way a source language feature is compiled, simply change the appropriate reusable compiler building block.

- To add or delete a source language feature, just add or delete the appropriate reusable compiler building block.

- To reuse work from existing modular compilers, simply reuse the desired compiler building blocks.

- Because each compiler building block is specified as a high-level definitional interpreter, it is straightforward (albeit non-trivial) to prove the correctness of reusable compiler building blocks and the compilers constructed from them.

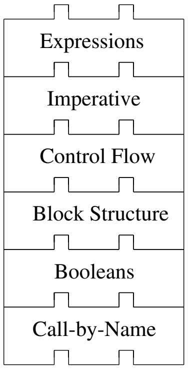
Compilers have traditionally been factored into phases (e.g., parsing, code generation, etc.), while modular compilers are also structured by source language feature (e.g., expressions, procedures, etc.), allowing a "mix and match" approach to compiler construction. It is easy to modify or extend an existing compiler which has been constructed according to this method, and it is also a simple matter to reuse existing work. With this approach, compilers for a language with many features (e.g., expressions, procedures, etc.) are built using compiler building blocks for each specific feature. Each compiler building block compiles a specific feature and can be easily combined with compiler building blocks for other features to provide compilers for non-trivial languages with many features. Furthermore, each compiler building block is reusable and may be used in the construction of compilers for many different languages. Compilers constructed according to this method are modular in that source language features may be added or deleted with ease, allowing the compiler writer to develop compilers at a high level of abstraction. This approach to structuring compilers is completely new in compiler design and is motivated by the insight that the same advantages provided by using the categorical concepts of *monads* and *monad transformers* in structuring interpreters[24, 25, 26, 10, 30, 10, 47] carry over to compilers.

The main accomplishments of this work are:

- Two implementations of reusable compiler building blocks are presented in this thesis, firstly as *metacomputations* in Chapter 2 and secondly as *monadic code generators* in Chapter 3.
- The introduction of *metacomputations* as a semantic model for staged computation. This model provides a modular and extensible style of staging denotational specifications.
- A novel approach to specifying and proving compiler correctness is developed. The correctness of individual RCBBs is specified independently, and under certain preconditions (called *linking conditions*), correct compiler building blocks may be combined into correct compilers, thereby attaining a level of *reusability* in compiler correctness proofs.
- A new form of program specification called *observations* is developed to facilitate reasoning about monadic specifications.

1.1 Modular Compilers

This section presents a number of compilers constructed modularly using "off-the-shelf" reusable compiler building blocks. Although no *implementations* of reusable compiler building blocks have



Source Code:

new g**:intvar in**
 let $f\ =\ \lambda v$: **intvar**. λe: **intexp**. $v := 1 + e\ \ ;\ \ v := (g+1) + e$ **in**
  $f\ g\ (g+1)$

Target Code:

```
<0,0> := 0;               <0,2> := 1;
<0,1> := 1;               <0,1> := <0,1> + <0,2>;
<0,2> := <0,0>;           <0,2> := <0,0>;
<0,3> := 1;               <0,3> := 1;
<0,2> := <0,2> + <0,3>;   <0,2> := <0,2> + <0,3>;
<0,0> := <0,1> + <0,2>;   <0,0> := <0,1> + <0,2>;
<0,1> := <0,0>;           halt;
```

Figure 1.3: Algol Compiler from Reusable Compiler Blocks

been given yet, the purpose of these examples is to illustrate the wide range of programming language features (e.g., procedures, expressions, assignment, booleans, scoping and parameter-passing mechanisms, as well as some optimization techniques) that can be represented as reusable compiler building blocks.

1.1.1 Compiler for Algol

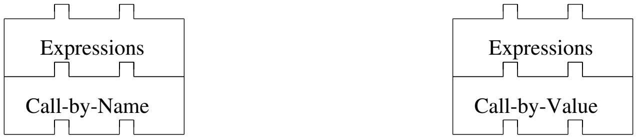
Figure 1.3 presents a compiler for an Algol-like language which was assembled from "off-the-shelf" reusable compiler building blocks. It has imperative features (like assignment), block structure, and call-by-name procedures. Note that, in this example, procedures are compiled with *inlining* rather than with "call" and "return" instructions. This is merely to simplify the presentation—in Section 1.1.5, a reusable compiler building block which compiles to *closed* subroutines (i.e., those with *call* and *return*) is presented.

The compiler in Figure 1.3 and subsequent compilers produce code with *display addresses* of the form `<frame,disp>`. `frame` refers to an activation record[1] on the stack, while `disp` is an offset into that activation record. Together, `frame` and `disp` indicate an address in storage.

1.1.2 Compiler for Functional Language

The next example illustrates how various parameter-passing mechanisms can be represented as reusable compiler building blocks. Figure 1.4 presents two compilers for a simple functional language. Again, these compilers were easily assembled from "off-the-shelf" reusable compiler building

Compiling: `(lambda (i) (+ i i)) (-(+ (+ 1 2) 3))`



```
<0,0> := 1;
<0,1> := 2;
<0,0> := <0,0>+<0,1>;
<0,1> := 3;
<0,0> := <0,0>+<0,1>;
<0,0> := -<0,0>;
<0,1> := 1;
<0,2> := 2;
<0,1> := <0,1>+<0,2>;
<0,2> := 3;
<0,1> := <0,1> + <0,2>;
<0,1> := -<0,1>;
Acc := <0,0> + <0,1>;
```

```
<0,1> := 1;
<0,2> := 2;
<0,1> := <0,1>+<0,2>;
<0,2> := 3;
<0,1> := <0,1>+<0,2>;
<0,0> := -<0,1>;
<0,1> := <0,0>;
<0,2> := <0,0>;
Acc := <0,1>+<0,2>;
```

Figure 1.4: Call-by-Name and Call-by-Value Compiler from Reusable Compiler Blocks

blocks. The compiler on the left has call-by-name procedures, while the compiler on the right has call-by-value. Again, procedures are compiled via inlining.

Note that the code compiled with the call-by-name compiler evaluates the actual parameter `(-(+ 1 2) 3)` twice, every time that the formal parameter `i` is evaluated. This is in accordance with call-by-name parameter passing. In contrast, the code compiled with the call-by-value compiler evaluates the actual argument once, storing the result in `<0,0>`, and uses the stored result twice.

1.1.3 Scoping Mechanisms as Reusable Compiler Building Blocks

The next example shows how different scoping mechanisms—static and dynamic—can be represented as reusable compiler blocks. Figure 1.5 presents two compilers constructed from "off-the-shelf" compiler building blocks. The compiler on the left combines call-by-name procedures with static scoping, while the compiler on the right combines call-by-name with dynamic scoping. As before, we compile procedures via inlining. Note that in the body of `f`, with static scoping, `s` should be 10. This is reflected in the compiled code on the left when 10 is used (boxed in the Figure) for `s` as expected. With dynamic scoping, the most recent definition of `s`—here it is 5—should be used, and the compiled code on the right does just that.

$$\text{Compiling:}\ \ \textbf{let}\ s = 10\ \textbf{in let}\ f = \lambda x.\, x + s\ \textbf{in}$$
$$\textbf{let}\ g = \lambda s.\, f\ (s + 11)\ \textbf{in}$$
$$(g\, 5)$$

Expressions
Static Scope
Expressions
Dynamic Scope

```
<0,0> := 5;
<0,1> := 11;
<0,0> := <0,0>+<0,1>;
<0,1> := 10;
Acc := <0,0>+<0,1>;
```

```
<0,0> := 5;
<0,1> := 11;
<0,0> := <0,0>+<0,1>;
<0,1> := 5;
Acc := <0,0>+<0,1>;
```

Figure 1.5: Static and Dynamic Scope as Reusable Compiler Building Blocks

1.1.4 Optimizing Compiler for Expressions

The next example compiler optimizes the amount of stack space and the number of instructions used in the compilation of expressions. Figure 1.6 displays the results of compiling an expression with two different reusable compiler building blocks. The code on the left was produced by the *non-optimizing* reusable compiler building block for expressions that has been used in all previous examples involving expressions, while the code on the right was compiled by the *optimizing* compiler building block for expressions. The non-optimizing compiler building block stores each intermediate value of an expression on the stack, with the result that more stack locations and machine instructions are produced than are necessary. For example in the compilation of the subexpression `(1+2)`, both `1` and `2` are stored on the stack in locations `<0,4>` and `<0,5>`, respectively, and the next instruction, `<0,2> := <0,4>+<0,5>`, computes the value $1 + 2$ by reading from these two temporary locations. The compiled code on the left in Figure 1.6 clearly involves more locations and instructions than are necessary. Because `1` and `2` are *simple* right-hand sides (i.e., they do not involve any arithmetic operations like addition or negation), it suffices to use the right-hand side `1+2` without storing `1` and `2` on the stack, and doing so would reduce the number of stack locations and machine instructions produced. This is precisely what the optimizing compiler building block does, and the savings in locations used and instructions issued by the optimizing block is clear when comparing the code produced by both blocks.

Compiling: `(((1+2)+(3+4))+((5+6)+(7+8)))`

Expressions
Optimizing
Expressions

```
<0,4> := 1;
<0,5> := 2;
<0,2> := <0,4>+<0,5>;
<0,4> := 3;
<0,5> := 4;
<0,3> := <0,4>+<0,5>;
<0,0> := <0,2>+<0,3>;
<0,4> := 5;
<0,5> := 6;
<0,2> := <0,4>+<0,5>;
<0,4> := 7;
<0,5> := 8;
<0,3> := <0,4> + <0,5>;
<0,1> := <0,2> + <0,3>;
Acc := <0,0> + <0,1>;
```

```
<0,0> := 1+2;
<0,1> := 3+4;
<0,0> := <0,0>+<0,1>;
<0,1> := 5+6;
<0,2> := 7+8;
<0,1> := <0,1>+<0,2>;
Acc := <0,0>+<0,1>;
```

Figure 1.6: Optimizing Compiler Building Block for Expressions

Compiling: `let negat=(lambda x.-x) in (3+(negat (7+9)))`

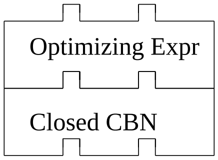
Expressions
Call-by-Name
Optimizing
Expressions
Call-by-Name

```
<0,0> := 3;
<0,3> := 7;
<0,4> := 9;
<0,2> := <0,3>+<0,4>;
<0,1> := -<0,2>;
Acc := <0,0> + <0,1>;
```

```
<0,0> := 7+9;
<0,0> := -<0,0>;
Acc := 3 + <0,0>;
```

Figure 1.7: Call-by-Name Compiler with and without Optimizing Expressions from Reusable Compiler Blocks

Compiling: `let negat=(lambda x.-x) in (3+(negat (7+9)))`



```
        call 101 [103] 0 102;

102:    <0,0> := SBRS;
        Acc := 3+<0,0>;
        halt;

103:    load SBRS,7+9;
        return;

101:    acall 1 1 1 100;

100:    <1,0> := SBRS;
        load SBRS,-<1,0>;
        return;
```

Figure 1.8: Closed Call-by-Name Procedures as Reusable Compiler Building Block

Figure 1.7 presents another example involving the two compiler building blocks for expressions—this time adding procedures.

Certainly, further optimizations could be performed on the code on the right-hand side of Figure 1.6 (e.g., *constant folding*[1] in particular), but the optimizing compiler building block for expressions demonstrates that at least some forms of optimization can occur within reusable compiler building blocks. How code optimization in general fits into the framework for modular compilation presented here remains an open question.

1.1.5 Closed Call-by-Name Procedure Block

Figure 1.8 presents an example compiler combining the optimizing expressions RCBB with the closed call-by-name procedures block. The procedure is compiled into machine language subroutines and subroutine calls.

1.2 Background

This Section outlines the necessary background for this thesis.

1.2.1 Partial Evaluation

Partial evaluation[17] is a program transformation technique which take a program `p(x,y)` and part of its input `x` and returns a specialized version of `p` (denoted $\mathtt{p}_{\mathtt{x}}$). Given `x`, parts of `p` may be evaluated, leaving a simplified program (called the *residual* program). Consider the following program `f` which multiplies its inputs:

```
f(x,y) = if x=0 then 0
         elsif x=1 then y
         elsif even(x) then f(x/2,y)+f(x/2,y)
         else y + f(x-1,y)
```

Partial evaluation of `f(3,y)` yields the program $\mathtt{f}_3(\mathtt{y}) = $ `y + y + y`. It reaches this residual program through the following steps:

$$\mathtt{f}(3,\mathtt{y}) \mapsto \mathtt{y} + \mathtt{f}(2,\mathtt{y}) \mapsto \mathtt{y} + \mathtt{f}(1,\mathtt{y}) + \mathtt{f}(1,\mathtt{y}) \mapsto \mathtt{y} + \mathtt{y} + \mathtt{y}$$

It should be noted that partial evaluation is equality-preserving (i.e., $\mathtt{p}_{\mathtt{x}}(\mathtt{y}) = \mathtt{p}(\mathtt{x},\mathtt{y})$) because only equal terms are substituted for one another, and that it is a source program to source program transformation.

In general, program execution can be divided into two distinct phases: *static* (or compile-time) and *dynamic* (or run-time). The static phase consists of those reductions in the execution which can be made by inspection of the program text alone (hence at compile-time). The dynamic phase consists of those reductions which depend on some part of the program's input unknown at compile-time. Partial evaluation attempts to perform as many static reductions as possible to produce a completely dynamic term.

1.2.2 Monads

Monads were first introduced to Computer Science by Eugenio Moggi to formalize different *notions of computation*[30, 31]. There are many different notions of computation involving combinations of states, continuations, environments, and errors. To illustrate, consider the following ML phrase e:

```
let val x = ref 0 in (x := !x + 1; !x) end
```

Evaluating e first creates a reference `x` to `0`, then increments the current value of `x`; then the dereference `!x` returns the current value `x`. Clearly, e will always return *1* when evaluated. But it would be inaccurate to claim that e is equivalent to the ML expression "`1`", because evaluating e and "`1`" are such vastly different *computations*. Evaluating e involves allocating and deallocating references as well as assigning to and reading from a reference, while evaluating "`1`" only involves returning the value *1*.

Definition 1 presents the usual definition of monads as functional programmers represent them. A *monad* is a type constructor M together with a pair of functions (obeying certain algebraic laws stated below):

$$\star_{\mathsf{M}} : \mathsf{M}\tau \to (\tau \to \mathsf{M}\tau') \to \mathsf{M}\tau'$$
$$\mathbf{unit}_{\mathsf{M}} : \tau \to \mathsf{M}\tau$$

A value of type $\mathsf{M}\tau$ is called a *τ-computation*, the idea being that it yields a value of type τ while also performing some other computation. The $\star_M$ operation generalizes function application in that it determines how the computations associated with monadic values are combined. $\mathbf{unit}_{\mathsf{M}}$ defines how a τ value can be regarded as a τ-computation; it is always a trivial computation.

Definition 1 (Functional Programmer's formulation) *A monad is a type constructor* T *and two polymorphic functions* $\mathbf{unit} : \forall a.a \to \mathsf{T}a$ *and* $\star\ : \forall a,b.\mathsf{T}a \to (a \to \mathsf{T}b) \to \mathsf{T}b$ *such that the following hold:*

$$(\mathbf{unit}\ a) \star k = k\ a \qquad \text{(left unit)}$$
$$x \star \mathbf{unit} = x \qquad \text{(right unit)}$$
$$x \star (\lambda a.(k\ a \star h)) = (x \star k) \star h \qquad \text{(assoc)}$$

Figure 1.9 presents five different monads encapsulating five different notions of computation. The simplest monad is the identity monad Id. Perhaps the best known monad is the state monad

Identity Monad

$$\begin{aligned}
&\mathsf{Id}\, a = a\\
&\mathbf{unit}\, x = x\\
&x \,\star\, f = f\, x
\end{aligned}$$

One State Monad

$$\begin{aligned}
&\mathsf{St}\, a = \mathit{Sto} \to a \times \mathit{Sto}\\
&\mathbf{unit}\, x = \lambda\sigma : \mathit{Sto}.\, \langle x, \sigma\rangle\\
&x \,\star\, f = \lambda\sigma_0 : \mathit{Sto}.\, \mathrm{let}\ \langle y, \sigma_1\rangle = (x\, \sigma_0)\ \mathrm{in}\ (f\, y\, \sigma_1)
\end{aligned}$$

Two State Monad

$$\begin{aligned}
&\mathsf{St2}\, a = \mathit{Sto} \to \mathit{Sto} \to a \times \mathit{Sto} \times \mathit{Sto}\\
&\mathbf{unit}\, x = \lambda\delta : \mathit{Sto}.\lambda\sigma : \mathit{Sto}.\, \langle x, \delta, \sigma\rangle\\
&x \,\star\, f = \lambda\delta_0 : \mathit{Sto}.\lambda\sigma_0 : \mathit{Sto}.\, \mathrm{let}\ \langle y, \delta_1, \sigma_1\rangle = (x\, \delta_0\, \sigma_0)\ \mathrm{in}\ (f\, y\ \delta_1\, \sigma_1)
\end{aligned}$$

Environment+State Monad

$$\begin{aligned}
&\mathsf{EnvSt}\, a = \mathit{Env} \to \mathit{Sto} \to a \times \mathit{Sto}\\
&\mathbf{unit}\, x = \lambda\rho : \mathit{Env}.\lambda\sigma : \mathit{Sto}.\, \langle x, \sigma\rangle\\
&x \,\star\, f = \lambda\rho_0 : \mathit{Env}.\lambda\sigma_0 : \mathit{Sto}.\, \mathrm{let}\ \langle y, \sigma_1\rangle = (x\, \rho_0\, \sigma_0)\ \mathrm{in}\ (f\, y\, \rho_0\, \sigma_1)
\end{aligned}$$

CPS Monad

$$\begin{aligned}
&\mathsf{CPS}\, a = (a \to \mathit{Ans}) \to \mathit{Ans}\\
&\mathbf{unit}\, (x : a) = \lambda(\kappa : a \to \mathit{Ans}).(\kappa\, x)\\
&x \,\star\, f = \lambda\kappa.\, x\, (\lambda y.(f\ y\, \kappa))
\end{aligned}$$

Figure 1.9: ADT Approach to Language Definition, Part 1: Various Monads

St, which represents the notion of a computation as something that modifies a store. The bind operation $\star$ for St handles the bookkeeping of "threading" the store through the computation. The monad $\mathsf{St2}$ has two separate stores so that its computations can cause side effects on each store. The monad EnvSt has both an environment and a store, while the continuation-passing monad CPS has first-class continuations.

Categorical View of Monads

Monads (or triples) are a categorical construction[28, 4] which predates Computer Science by several decades. Definitions 2 and 3 present equivalent categorical formulations of monads, but Definition 3—the Kleisli formulation—corresponds to the "functional programmer's formulation" of monads in Definition 1. For further information on the categorical view of monads, please consult a standard text on Category theory[28, 4].

Definition 2 (Standard formulation) *A monad in a category $\mathcal{C}$ is a triple $\langle \mathsf{T}, \eta, \mu \rangle$ of an endofunctor $\mathsf{T} : \mathcal{C} \to \mathcal{C}$ and two natural transformations $\eta : Id \dot{\to} \mathsf{T}$ and $\mu : \mathsf{T}^2 \dot{\to} \mathsf{T}$ such that the following diagrams commute:*

$$\begin{tikzcd}
\mathsf{T}^3 \arrow[r, "\mathsf{T}\mu"] \arrow[d, "\mu\mathsf{T}"'] & \mathsf{T}^2 \arrow[d, "\mu"] \\
\mathsf{T}^2 \arrow[r, "\mu"] & \mathsf{T}
\end{tikzcd}
\qquad
\begin{tikzcd}
Id\mathsf{T} \arrow[r, "\eta\mathsf{T}"] \arrow[dr, "id"'] & \mathsf{T}^2 \arrow[d, "\mu"] & \mathsf{T}Id \arrow[l, "\mathsf{T}\eta"'] \arrow[dl, "id"] \\
 & \mathsf{T} &
\end{tikzcd}$$

Definition 3 (Kleisli formulation) *A monad in a category $\mathcal{C}$ is a triple $\langle \mathsf{T}, \eta, \_^* \rangle$ of an endofunction T and two families of arrows $\eta_A : A \to \mathsf{T}A$ and $\_^* : (A \to \mathsf{T}B) \to (\mathsf{T}A \to \mathsf{T}B)$ (neither required to be natural) such that, for arrows $f : A \to \mathsf{T}B$ and $g : B \to C$,*

$$\eta_A^* = id_{\mathsf{T}A} \tag{1.1}$$

$$\eta_A \,;\, f^* = f \tag{1.2}$$

$$f^* \mathbin{;} g^* = (f \mathbin{;} g^*)^* \tag{1.3}$$

1.2.3 Monads and the Abstract Data Type Approach to Language Definition

The principal advantage of the monadic approach to language definition is that the underlying denotational *model* can be arbitrarily complex without complicating the denotational *description* unnecessarily. The beauty of the monadic form is that the equations defining $[\![-]\!] : \mathcal{E}xp \to \mathsf{M}a$ can be reinterpreted in a variety of monads M. Monadic semantics separate the *description* of a language from its *denotation*[1]. In this sense, it is similar to *action semantics*[34] and *high-level semantics*[23]. To borrow a term from the language of abstract data types, the monadic semantics of programming languages yields *representationally independent* definitions. This is what prompts some authors (notably Espinosa[10]) to refer to monadic semantics as the "ADT approach to language definition."

Before seeing how monads are used in language definitions, let us first consider standard functional-style language definitions and why they are essentially representationally-dependent. Suppose we wish to define a language of integer expressions containing constants and negation. The standard functional definition might be:

$$[\![-e]\!] \;=\; -[\![e]\!]$$

where $[\![-]\!] : \mathit{Exp} \to \mathit{int}$. However, this definition is inflexible; if the integer expressions were part of a larger language, then this equation would have to change. Consider what would happen if integer expressions were part of a language with assignment. The semantic function might have type $[\![-]\!] : \mathit{Exp} \to \mathit{Sto} \to \mathit{Value} \times \mathit{Sto}$ where Sto is a store used for defining assignment. The equation defining negation would become:

$$[\![-e]\!]\sigma = \text{ let } \langle v, \sigma' \rangle = [\![e]\!]\sigma \text{ in } \langle -v, \sigma' \rangle$$

Although these two definitions of negation *do* the same thing, they *look* entirely different. Even

[1] Lee calls this property *separability*[23].

$$\boxed{[\![-]\!] : \mathcal{E}xp \to int}$$

Functional-style

$[\![i]\!] = i$

$[\![-e]\!] = -[\![e]\!]$

Identity Monad

$[\![i]\!] = \mathbf{unit}(i)$

$[\![-e]\!] = [\![e]\!] \star \lambda v.\mathbf{unit}(-v)$

$$\boxed{[\![-]\!] : \mathcal{E}xp \to Sto \to int \times Sto}$$

Functional-style

$[\![i]\!]\sigma = \langle i, \sigma \rangle$

$[\![-e]\!]\sigma = \text{ let } \langle v, \sigma' \rangle = [\![e]\!]\sigma \text{ in } \langle -v, \sigma' \rangle$

State Monad

$[\![i]\!] = \mathbf{unit}(i)$

$[\![-e]\!] = [\![e]\!] \star \lambda v.\mathbf{unit}(-v)$

$$\boxed{[\![-]\!] : \mathcal{E}xp \to \tau \to Sto \to int \times \tau \times Sto}$$

Functional-style

$[\![i]\!](t : \tau)(\sigma : Sto) = \langle i, t, \sigma \rangle$

$[\![-e]\!]t\,\sigma = \text{ let } \langle v, t', \sigma' \rangle = [\![e]\!]t\,\sigma \text{ in } \langle -v, t', \sigma' \rangle$

Two State Monad

$[\![i]\!] = \mathbf{unit}(i)$

$[\![-e]\!] = [\![e]\!] \star \lambda v.\mathbf{unit}(-v)$

$$\boxed{[\![-]\!] : \mathcal{E}xp \to Env \to Sto \to int \times Sto}$$

Functional-style

$[\![i]\!]\rho\sigma = \langle i, \sigma \rangle$

$[\![-e]\!]\rho\sigma = \text{ let } \langle v, \sigma' \rangle = [\![e]\!]\rho\sigma \text{ in } \langle -v, \sigma' \rangle$

Environment+State Monad

$[\![i]\!] = \mathbf{unit}(i)$

$[\![-e]\!] = [\![e]\!] \star \lambda v.\mathbf{unit}(-v)$

$$\boxed{[\![-]\!] : \mathcal{E}xp \to (int \to Ans) \to Ans}$$

Functional-style

$[\![i]\!](\beta : int \to Ans) = \beta\, i$

$[\![-e]\!]\beta = [\![e]\!](\lambda v : int.\beta\,(-v))$

CPS Monad

$[\![i]\!] = \mathbf{unit}(i)$

$[\![-e]\!] = [\![e]\!] \star \lambda v.\mathbf{unit}(-v)$

Figure 1.10: ADT Approach to Language Definition, Part 2: Functional and Monadic-style

though negation does not use the store, its defining equation must still reflect the existence of the store. In the left column of Figure 1.10, there are a number of other functional-style equations defining negation for denotations with two states, an environment and a state, and continuations. Again, all of these definitions compute "$-e$" the same way—just negating the value of the subexpression e—but they appear completely different because they must all reflect the structure of their underlying denotations. Functional-style denotational definitions are, therefore, necessarily representationally-dependent.

In contrast, monadic-style semantic equations are free from the details of the underlying denotation. To see why this is, consider *the* monadic definition for negation:

$$[\![-e]\!] = [\![e]\!] \ \star \ \lambda v.\mathbf{unit}(-v)$$

This equation defines negation for all monads! Because the monadic unit and bind operations handle the "bookkeeping" of passing the extra computational "stuff" (stores, environments, continuations, etc.), and since negation does not use any of this data, its semantic equation need not refer to them *explicitly*. Thus, monadic language specifications are essentially representationally-independent.

1.2.4 Monad Transformers

Now, suppose we wish to create new monads from existing monads. For example, one might wish to try to create the two-state monad $\mathsf{St2}$ (see Figure 1.9) from the single state monad St. One might expect to get $\mathsf{St2}$ by applying the ordinary state monad twice. Unfortunately, $(\mathsf{St} \circ \mathsf{St})\,\tau$ and $\mathsf{St2}\,\tau$ are very different types. This points to a difficulty with monads: they do not compose in this simple manner.

The key contribution of the work [10, 25] on *monad transformers* is to solve this composition problem. When applied to a monad M, a monad transformer $\mathcal{T}$ creates a new monad M'. For example, the state monad transformer, $\mathcal{T}_{St}\, store$, is shown in Figure 1.11. (Here, the *store* is a type argument, which can be replaced by any value which is to be "threaded" through the computation.) Note that $\mathcal{T}_{St}\, Sto\, \mathsf{Id}$ is identical to the state monad, but here we get a useful notion of composition: $\mathcal{T}_{St}\, Sto\, (\mathcal{T}_{St}\, Sto\, \mathsf{Id})$ is equivalent to the two-state monad $\mathsf{St2}\,\tau$. In addition to

$\boxed{\text{Identity Monad } Id\text{:}}$

$$\mathsf{Id}\,\tau = \tau$$
$$\mathbf{unit}_{\mathsf{Id}}\, x = x$$
$$x \star_{\mathsf{Id}} f = f\, x$$

$\boxed{\text{CPS Monad Transformer } \mathcal{T}_{\mathsf{CPS}}\text{:}}$

$$\mathsf{M}'\tau = \mathcal{T}_{\mathsf{CPS}}\, ans\, \mathsf{M}\,\tau = (\tau \to \mathsf{M}\, ans) \to \mathsf{M}\, ans$$
$$\mathbf{unit}_{\mathsf{M}'}\, x = \lambda\kappa.\, \kappa\, x$$
$$x \star_{\mathsf{M}'} f = \lambda\kappa.\, x(\lambda a. f\, a\, \kappa)$$
$$\mathit{lift}_{\mathsf{M}\tau\to\mathsf{M}'\tau}\, x = \star_{\mathsf{M}}$$
$$\texttt{callcc} \; : ((a \to \mathsf{M}b) \to \mathsf{M}a) \to \mathsf{M}a$$
$$\texttt{callcc}\; f = \lambda\kappa. f(\lambda a.\lambda\_.\kappa\, a)\,\kappa$$

$\boxed{\text{Environment Monad Transformer } \mathcal{T}_{\mathsf{Env}}\text{:}}$

$$\mathsf{M}'\tau = \mathcal{T}_{\mathsf{Env}}\, Env\, \mathsf{M}\,\tau = Env \to \mathsf{M}\tau$$
$$\mathbf{unit}_{\mathsf{M}'}\, x = \lambda\rho : Env.\, \mathbf{unit}_{\mathsf{M}}\, x$$
$$x \star_{\mathsf{M}'} f = \lambda\rho : Env.\, (x\,\rho) \star_{\mathsf{M}} (\lambda a. f\, a\, \rho)$$
$$\mathit{lift}_{\mathsf{M}\tau\to\mathsf{M}'\tau}\, x = \lambda\rho : Env.\, x$$
$$\texttt{rdEnv} : \mathsf{M}'Env = \lambda\rho : Env.\, \mathbf{unit}_{\mathsf{M}}\rho$$
$$\texttt{inEnv}(\rho : Env,\, x : \mathsf{M}'\tau) = \lambda\_.\, (x\,\rho) : \mathsf{M}'\tau$$

$\boxed{\text{State Monad Transformer } \mathcal{T}_{\mathsf{St}}\text{:}}$

$$\mathsf{M}'\tau = \mathcal{T}_{\mathsf{St}}\, \mathit{store}\, \mathsf{M}\,\tau = \mathit{store} \to \mathsf{M}(\tau \times \mathit{store})$$
$$\mathbf{unit}_{\mathsf{M}'}\, x = \lambda\sigma : \mathit{store}.\, \mathbf{unit}_{\mathsf{M}}(x, \sigma)$$
$$x \star_{\mathsf{M}'} f = \lambda\sigma_0 : \mathit{store}.\, (x\,\sigma_0) \star_{\mathsf{M}} (\lambda(a, \sigma_1). f\, a\, \sigma_1)$$
$$\mathit{lift}_{\mathsf{M}\tau\to\mathsf{M}'\tau}\, x = \lambda\sigma.\, x \star_{\mathsf{M}} \lambda y.\, \mathbf{unit}_{\mathsf{M}}(y, \sigma)$$
$$\texttt{update}(\Delta : \mathit{store} \to \mathit{store}) = \lambda\sigma.\, \mathbf{unit}_{\mathsf{M}}(\bullet, \Delta\,\sigma)$$
$$\texttt{getStore} = \lambda\sigma.\, \mathbf{unit}_{\mathsf{M}}(\sigma, \sigma)$$

One element type: `void` such that $\bullet \in$ `void`

Figure 1.11: The Identity Monad, and Environment, CPS, and State Monad Transformers

creating a new monad, monad transformers also define non-proper morphisms (i.e., morphisms other than $\star$ and **unit**) which allow the new data to be manipulated. For example, the state monad transformer $\mathcal{T}_{St}\, Sto$ provides non-proper morphisms `updateSto` and `getSto` to update and return the *Sto* state, respectively. When composing $\mathcal{T}_{St}\, Sto$ with itself, as above, the operations on the "inner" state need to be *lifted* through the outer state monad; this is the main technical issue in [10, 25].

Categorical View of Monad Transformers

There are two equivalent categorical definitions of monad transformers in the literature occurring in Moggi[30] and Espinosa[10]. We summarize Moggi[30] here. The interested reader should consult [30, 10] for further details.

Moggi[30] defines monad transformers (which he called *monad constructors*) as endofunctors on the category of monads $\mathcal{M}on(\mathcal{C})$. The objects of $\mathcal{M}on(\mathcal{C})$ are the monads over the category $\mathcal{C}$. A morphism in $\mathcal{M}on(\mathcal{C})$ from monad (T, η^T, μ^T) to monad (S, η^S, μ^S) is a *monad-morphism* from T to S, where monad-morphism is defined below.

Definition 4 (monad-morphism) *Given monads (T, η^T, μ^T) over $\mathcal{C}$ and (S, η^S, μ^S) over $\mathcal{D}$, a* monad-morphism *is a pair (U, σ), where $U : \mathcal{C} \to \mathcal{D}$ is a functor and $\sigma : T;U \stackrel{\cdot}{\to} U;S$ is a natural transformation such that:*

$$\begin{tikzcd}
UA \arrow[r, "U\eta^T_A"] \arrow[dr, "\eta^S_{UA}"'] & U(TA) \arrow[d, "\sigma_A"'] & U(T^2A) \arrow[l, "U\mu^T_A"'] \arrow[d, "\sigma_{TA}"] \\
 & S(UA) & S(U(TA)) \arrow[d, "S\sigma_A"] \\
 & & S^2(UA) \arrow[ul, "\mu^S_{UA}"]
\end{tikzcd}$$

1.2.5 Modular Interpreters and Monad Transformers

The previous two sections have described how the representational independence of monadic language definitions allows one specification $\llbracket - \rrbracket : \mathcal{L} \to \mathsf{M}(\mathit{value})$ to be reinterpreted in many different monads M, and how monad transformers allow monads to be combined easily. These two facts make it possible to compose monadic language *specifications* as well. Let us say that we have specifications for languages $\mathcal{L}_1$ and $\mathcal{L}_2$:

$$\llbracket - \rrbracket_1 : \mathcal{L}_1 \to \mathsf{M}_1(Values_1) \qquad \llbracket - \rrbracket_2 : \mathcal{L}_2 \to \mathsf{M}_2(Values_2)$$

where $\mathsf{M}_1 = \mathcal{T}_1\,\mathsf{Id}$ and $\mathsf{M}_2 = \mathcal{T}_2\,\mathsf{Id}$. Since M_1 and M_2 are constructed with monad transformers, it is possible to create a "supermonad" $\mathcal{T}_1(\mathcal{T}_2\,\mathsf{Id})$ in which both $\llbracket - \rrbracket_1$ and $\llbracket - \rrbracket_2$ may be reinterpreted. This effectively composes $\llbracket - \rrbracket_1$ and $\llbracket - \rrbracket_2$ into a single specification $\llbracket - \rrbracket_1 + \llbracket - \rrbracket_2$:

$$\begin{aligned}
&\llbracket - \rrbracket_1 + \llbracket - \rrbracket_2 : (\mathcal{L}_1 + \mathcal{L}_2) \to (\mathcal{T}_1(\mathcal{T}_2\,\mathsf{Id}))(Values_1 + Values_2) \\
&\llbracket t \rrbracket_1 + \llbracket t \rrbracket_2 = \begin{cases} \llbracket t \rrbracket_1 : (\mathcal{T}_1(\mathcal{T}_2\,\mathsf{Id}))(Values_1), & \text{if } t \in \mathcal{L}_1 \\ \llbracket t \rrbracket_2 : (\mathcal{T}_1(\mathcal{T}_2\,\mathsf{Id}))(Values_2), & \text{if } t \in \mathcal{L}_2 \end{cases}
\end{aligned} \tag{1.4}$$

$\mathcal{C}[\![e]\!] : \mathsf{M}((\mathit{int} \to \mathsf{M}(\mathit{int})) \to \mathsf{M}(\mathit{int}))$ where $\mathit{Addr} = \mathit{int}$, $\mathit{Sto} = \mathit{Addr} \to \mathit{int}$, and $\mathsf{M} = \mathcal{T}_{\mathsf{St}}\ \mathit{Addr}\ (\mathcal{T}_{\mathsf{St}}\ \mathit{Sto}\ \mathsf{Id})$.

$$\mathcal{C}[\![n]\!] = \mathbf{unit}\ \lambda\beta.\ \beta n$$

$$\mathcal{C}[\![-t]\!] = \mathcal{C}[\![t]\!] \ \star\ \lambda e.\mathbf{unit}\ \Big(\ \lambda\beta.e(\lambda i.\texttt{CreateTemp}(i)\ \star\ \lambda v.\texttt{deAlloc}\ \star\ \lambda\_.\,\beta(-v))\ \Big)$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\texttt{deAlloc} = \texttt{updateA}(\lambda a.a - 1)} \\
\multicolumn{2}{l}{\texttt{allocLoc} = \texttt{rdAddr} \star \lambda a.\texttt{updateA}(\lambda a.a + 1) \star \lambda\_.\mathbf{unit}\, a} \\
\multicolumn{2}{l}{\texttt{CreateTemp}(v) =} \\
\quad \texttt{allocLoc} \star \lambda a. & \texttt{/*allocate Addr*/} \\
\quad \texttt{updateSto}[a \mapsto v] \star \lambda\_. & \texttt{/*store v at a*/} \\
\quad \texttt{rdSto} \star \lambda\sigma. & \texttt{/*get curr. Sto*/} \\
\qquad \mathbf{unit}(\sigma\, a) & \texttt{/*return val at a*/}
\end{array}$$

Figure 1.12: Reusable Compiler Building Block for *Exp*

Note that the composite specification $[\![t]\!]_1 + [\![t]\!]_2$ returns a value in $Values_1$ if t is in $\mathcal{L}_1$ and a value in $Values_2$ if t is in $\mathcal{L}_2$ (and not a pair of values). There is the (mild) proviso that all environment monad transformers ($\mathcal{T}_{\mathsf{Env}}\,\tau$) should be applied after any CPS monad transformers ($\mathcal{T}_{\mathsf{CPS}}\,\tau'$) are applied, because it has been shown[26] that the combinator `inEnv` defined by the environment monad transformer can not be lifted through the CPS monad transformer. In practice, this restriction causes no difficulties whatsoever.

Equational language definitions (such as those defining $[\![-]\!]_i$) are frequently referred to as interpreters[21, 11]. In the preceding paragraph, the pair consisting of the equations defining $[\![-]\!]_1$ and the monad transformer $\mathcal{T}_1$ is called an *interpreter building block*[24, 25, 26] for the language $\mathcal{L}_1$ (and similarly, $\langle[\![-]\!]_2, \mathcal{T}_2\rangle$ is an interpreter building block for $\mathcal{L}_2$). Because a wide variety of programming language features (e.g., variables, assignment, control-flow, procedures, etc.) may be defined as interpreter building blocks, interpreters for languages with many features may be constructed modularly in a "mix and match" fashion using interpreter building blocks, and this gives rise to what Liang, et al., call *modular monadic interpreters*.

1.2.6 Modular Compilers and Monad Transformers

The previous Section outlined how interpreters structured by monad transformers are modular, and compilers based on monad transformers have similar properties. In [13], the authors developed a method of modular compiler construction analogous to the modular interpreter construction in [24, 25, 10]. Compilers are organized in [13] as *reusable compiler building blocks* (RCBB) which can be "mixed and matched" just as the modular interpreter building blocks described in the previous

Scheme output from partial evaluator:

```
(lambda (store add negate read)
 (lambda (a0) (lambda (sto1)
  (cons (cons star 0)
   ((store "Acc" (negate (read 0)))
    ((store 0 (negate (read 0))) ((store 0 1) sto1)))))))
```

Pretty printed version:

```
          0 := 1; 0 := -[0]; Acc := -[0];
```

Figure 1.13: Compiling "$- - 1$"

Section. In fact, the reusable compiler building blocks in [13] are really just "implementation-oriented" interpreter building blocks. That is, extra implementation-level data is introduced using monad transformers, and then type-directed partial evaluation[7, 8] is used to produce a term which can be pretty-printed to produce machine code.

Figure 1.12 contains a RCBB for the arithmetic expression language *Exp* (written in continuation-passing style). It is *implementation-oriented* in that extra implementation-level data has been added by the composite monad transformer $\lambda \mathsf{M}.\mathcal{T}_{\mathsf{St}}\, Addr\, (\mathcal{T}_{\mathsf{St}}\, Sto\, \mathsf{M})$. *Addr* is a "free address" counter state, and *Sto* is a value store. The equations for $\mathcal{C}[\![e]\!]$ define an alternative semantics for expressions in which intermediate values of expressions are stored in the value store. $\mathcal{C}[\![e]\!]$ behaves like typical machine code for an expression, which is reflected by the residual term produced by partial evaluating $\mathcal{C}[\![- - 1]\!]$ in Figure 1.13. The term in the top half of that Figure is simply a sequence of stores and reads from the value store, which can be reasonably pretty-printed as shown in the bottom half of Figure 1.13.

1.2.7 Metacomputations, Metacomputation-style Staging and Modular Compilers

Metacomputations—computations that produce computations—arise naturally in the compilation of programs. Figure 1.14 illustrates this idea. The source language program s is taken as input by the compiler, which produces a target language program t. So, compiling s produces another computation—namely, the computation of t. Observe that there are two entirely distinct notions of computation here: the compilation of s and the execution of t. The reader will recognize

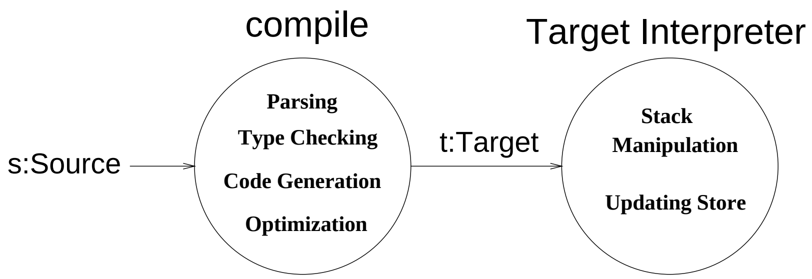


Figure 1.14: Handwritten compiler as metacomputation

this distinction as the classic separation of static from dynamic. Thus, *staging* is an instance of metacomputation.

We can formalize this notion of metacomputation using monads[10, 25, 31, 47] and use the resulting framework as a basis for staging computations. Given a monad M, the *computations* of type a is the type $\mathsf{M}\,a$. So given two monads M and N, the *metacomputations* of type a is the type $\mathsf{M}(\mathsf{N}\,a)$, because the M-computation produces as a value an N-computation. This definition is not superfluous; as we have noted, $\mathsf{M} \circ \mathsf{N}$ is not generally a monad, so metacomputations are generally a different notion altogether from computations.

1.3 Related Work

Structuring denotational semantics with monads and monad transformers was originally proposed by Moggi [30, 31]. Hudak, Liang, and Jones [25], Espinosa [10], and Wadler [47] use monads and monad transformers to create modular, extensible interpreters. Their work shows how interpreters can be developed in a modular way, leaving open the question of whether compilers can be developed similarly. Liang [24, 26] addresses that question, proposing that monadic semantics constructed from monad transformers and monadic specifications provide a modular and extensible basis for semantics-directed compilation. As an example of reasoning in monadic style, he axiomatizes the environment combinators `rdEnv` and `inEnv`, and shows that these axioms hold in any monad constructed with standard monad transformers. He describes an experiment [26] wherein the Glasgow Haskell compiler is re-targeted to the SML/NJ back-end, and he develops several examples of reasoning about monadic specifications. Liang's work is the most closely related to ours, but since he does not compile to machine language, many of the issues we confront do not arise.

Jorring and Scherlis [20] introduced the term "pass separation", which they defined as:

> The idea of pass separation is to introduce intermediate data structure to pass values between two phases of computation, enabling separation of the two phases.

They showed how compilers could be constructed by introducing intermediate data structures into an interpreter and then partially evaluating. Their interpreter had no monadic structure. Also, their derivations were non-automatic, as the introduction and exploitation of intermediate data structure was quite subtle.

Danvy and Vestergaard [8] show how to produce code that "looks like" machine language, by expressing the source language semantics in terms of machine language-like combinators (e.g., "update", "popblock", "push"). When the interpreter is closed over these combinators, partial evaluation of this closed term with respect to a program produces a completely *dynamic* term, composed of a sequence of combinators. These combinators, then, constitute the target language of the compiler.

In Morris[32], Thatcher, et al.,[46], and Wand[48], the correctness of a compiler from source language L to target language $MachLang$ would be expressed by the following diagram:

$$\begin{tikzcd}
L \arrow[r, "compile"] \arrow[d, "source\ semantics"'] & MachLang \arrow[d, "target\ semantics"] \\
M \arrow[r, "encode"] & U
\end{tikzcd}$$

where M and U are the possibly distinct source and target semantic models and *encode* "implements" source denotations with target denotations. Given certain assumptions about these arrows (e.g., *compile* is "syntax-directed" and both semantics are compositional), the compiler is correct if the diagram commutes.

In [43], Reynolds demonstrates how to produce efficient code in a compiler derived from the functor category semantics of an Algol-like language, which was an original inspiration for this study. Our compiler for that language improves on Reynolds in two ways: it is monad-structured—that is, built from interchangeable parts—and it includes jumps and labels where Reynolds simply allowed code duplication and infinite programs.

A syntactic form of metacomputation can be found in the two-level λ-calculus of Nielson[37]. Two-level λ-calculus contains two distinct λ-calculi—representing the static and dynamic *levels*. Expressions of mixed level, then, have strongly separated binding times by definition. Nielson[36] applies two-level λ-calculus to code generation for a typed λ-calculus, and Nielson[37] presents an algorithm for static analysis of a typed λ-calculus which converts one-level specifications into two-level specifications. Mogensen[29] generalizes this algorithm to handle variables of mixed binding times. The present work offers a semantic alternative to the two-level λ-calculus. We formalize distinct levels (in the sense of Nielson[37]) as distinct monads, and the resulting specifications have all of the traditional advantages of monadic specifications (reusability, extensibility, and modularity). While our binding time analysis is not automatic as in [37, 29], we consider a wider range of programming language features than they do.

In [49, 50], Wand presents a combinator-based approach to compilation, in which a continuation semantics is rewritten in terms of special-purpose machine language-like combinators. The denotation of a term according to the resulting semantics is a tree of combinators (with λ-abstractions removed) which can be rotated into a left linear (or almost linear) form through the application of associative and distributive laws. After rotation, this tree resembles machine language code. In [51], Wand extends this work with explicit loops by reintroducing variables in a restricted way resembling labels in machine code. Wand[48] describes the correctness proof of the compilers from [49, 50]. The present work differs from Wand's in that our combinators are constructed semi-automatically via monad transformation and lifting, and thus a number of properties of monadic operations (e.g., the associativity of `bind`) are preserved. Many of Wand's combinators appear to be *ad hoc* versions of `bind`. A further difference is that code generation is automatically performed by partial evaluation rather than by hand. Furthermore, it is not obvious that Wand's original semantics and his machine language-like combinator version of the semantics are related, because the form of the combinator semantics is completely different. The pass separation transformations used here make fairly minimal changes to the standard semantic equations and result in a compilation semantics which is recognizably a more implementation-oriented version of the standard semantics.

Chapter 2

Reusable Compiler Building Blocks as Metacomputations

This chapter concerns the implementation of reusable compiler building blocks as *metacomputations.* A *metacomputation* is "two-phase" computation; that is, a computation which produces another computation. Metacomputation-style staging uses two monads to factor the static and dynamic parts of a language specification, thereby staging the specification and achieving strong binding-time separation. Because metacomputations are defined in terms of monads, they can be constructed modularly and extensibly using monad transformers. A number of language constructs are specified: expressions, control-flow, imperative features, block structure, recursive bindings, and higher-order procedures. Metacomputation-style specification lends itself to semantics-directed compilation, which is demonstrated by creating a modular compiler for a block-structured, imperative while language with higher-order integer-valued functions.

2.1 Why Metacomputations?

Because metacomputations are a natural model for staged computation, it is clearly a reasonable starting point for semantics-directed compilation. In fact, metacomputation-style language specifications have more than an aesthetic advantage over traditional monadic specifications with respect to semantics-directed compilation. This Section describes at a high level why two monads are better than one for modular compilation. Using metacomputations instead of a single monolithic monad simplifies the use of the "code store" (defined below) in the specification of reusable compiler building blocks.

In [13], a technique from denotational semantics[44] is borrowed for modeling jumps, namely storing command continuations in a "code store" and denoting "`jump` L" as "execute the continuation at label L in the code store." Viewing command continuations as machine code is a common technique in semantics-directed compilation[50, 43]. Because the language specifications were in monadic style, it was a simple matter to add label generator and code store states to the underlying monad. Indeed, the primary use for monads in functional programming seems to be that of adding state-like features to purely functional languages and programs[47, 39], and the fact that the monads in [13] are structured with monad transformers made adding the new states simple.

The use of a code store is integral to the modular compilation technique described in [13]. We use it to compile control-flow and procedures, and the presence of the code store in the language specifications allowed substantial improvements over Reynolds[43] (e.g., avoiding infinite programs through jumps and labels). Yet the mixing of static with dynamic data into one "monolithic" monad causes a number of problems with using the code store. Consider the program "**if** b **then** (**if** b' **then** c)". Compiling the outer "**if**" with initial continuation **halt** and label 0 will result in the continuation "$[\![\mathbf{if}\, b'\, \mathbf{then}\, c]\!]; \mathbf{halt}$" being stored at label 0 and the label counter being incremented. The problem here is that trying to compile this continuation via partial evaluation will fail. Why? Because having been *stored* rather than *executed*, it will not have access to the next label 1. Instead, the partial evaluator will try to increment a (dynamic) variable rather than an actual (static) integer, and this will cause an error (e.g., a partial evaluator can evaluate "1+1" but not "x+1" even if x is constant). In [13], the monolithic style specifications forced all static data to be explicitly passed to stored command continuations, although this was at the expense of modularity. In fact to compile **if-then-else**, the snapback operator[40] had to be used. These complications also make reasoning about compilers constructed in [13] difficult. We shall demonstrate in Section 2.2 that using metacomputations results in vastly simpler compiler specifications than in [13] and that this naturally makes them easier to reason about.

$\boxed{\text{Standard:}}$

$\mathsf{Dynam} = \mathsf{Id}$

$$
\begin{aligned}
&[\![-t]\!] : \mathsf{Dynam}(int) = \\
&\quad [\![t]\!] \star_D \lambda i. \\
&\qquad \mathbf{unit}_D\ (-i)
\end{aligned}
$$

$\boxed{\text{Implementation-oriented/Monolithic:}}$

$$
\begin{aligned}
&\mathsf{Dynam} = \mathcal{T}_{\mathsf{Env}}\, Addr\, (\mathcal{T}_{\mathsf{St}}\, Sto\, \mathsf{Id}) \\
&Addr = int, Sto = Addr \to int \\
&\mathtt{Thread}(i : int, a : Addr) = \\
&\quad \mathtt{updateSto}[a \mapsto i] \star_D \lambda\_.\mathtt{rdloc}(a) \\
&\mathtt{rdloc}(a) = \mathtt{getSto} \star_D \lambda\sigma.\mathbf{unit}_D(\sigma a)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{M}ono[\![-t]\!] : \mathsf{Dynam}(int) = \\
&\quad \mathcal{M}ono[\![t]\!] \star_D \lambda i. \\
&\quad \mathtt{rdAddr} \star_D \lambda a. \\
&\quad \mathtt{inAddr}\ (a+1) \\
&\qquad (\mathtt{Thread}(i,a) \star_D \lambda v.\mathbf{unit}_D\ (-v))
\end{aligned}
$$

$\boxed{\text{Metacomputation:}}$

$$
\begin{aligned}
&\mathsf{Dynam} = \mathcal{T}_{\mathsf{St}}\, Sto\, \mathsf{Id} \\
&\mathsf{Static} = \mathcal{T}_{\mathsf{Env}}\, Addr\, \mathsf{Id}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{C}[\![-t]\!] : \mathsf{Static}(\mathsf{Dynam}(int)) = \\
&\quad \mathtt{rdAddr} \star_S \lambda a. \\
&\quad \mathtt{inAddr}\ (a+1) \\
&\qquad (\mathcal{C}[\![t]\!] \star_S \lambda\varphi_t : \mathsf{Dynam}(int). \\
&\qquad\quad \mathbf{unit}_S \boxed{\left(\begin{array}{l} \varphi_t \star_D \lambda i. \\ \mathtt{Thread}(i,a) \star_D \lambda v. \\ \qquad \mathbf{unit}_D(-v) \end{array}\right)}
\end{aligned}
$$

Figure 2.1: Negation, 3 ways

2.2 Metacomputation-style Compiler Architecture

In this section, several compiler building blocks are presented. In Section 2.3, they will be combined to create a compiler. For the first two of these blocks, monolithic versions are also given, drawn from [13], to illustrate why metacomputation is helpful. Of particular importance to the present work, Section 2.2.2 presents the reusable compiler building block for control flow, which demonstrates how metacomputation-based compiler architecture solves the difficulties with the monolithic approach outlined in Section 2.1.

2.2.1 Integer Expressions Compiler Building Block

Consider the standard monadic-style specification of negation[10, 25, 47] displayed in Figure 2.1. To use this as a compiler specification for negation, a more implementation-oriented version is needed, which might be defined informally as:

$$[\![-t]\!] = [\![t]\!] \star_D \lambda i.\ \text{“Store } i \text{ at } a \text{ and return contents of } a\text{”} \star_D \lambda v.\mathbf{unit}_D\ (-v)$$

$$\begin{array}{l}
\mathcal{C}[\![e_1 + e_2]\!] : \mathsf{Static}(\mathsf{Dynam}\, int) = \\
\qquad \texttt{rdAddr} \star_S \lambda a. \\
\qquad \mathcal{C}[\![e_1]\!] \star_S \lambda \varphi_1. \\
\qquad \mathcal{C}[\![e_2]\!] \star_S \lambda \varphi_2. \\
\qquad\qquad \texttt{inAddr}\,(a+2) \\
\qquad\qquad\qquad \mathbf{unit}_S \boxed{\left(\begin{array}{l}
\varphi_1 \star_D \lambda i : int. \\
\varphi_2 \star_D \lambda j : int. \\
\texttt{Thread}(i, a) \star_D \lambda v_1. \\
\texttt{Thread}(j, (a+1)) \star_D \lambda v_2. \\
\quad \mathbf{unit}_D(v_1 + v_2)
\end{array}\right)}
\end{array}$$

Figure 2.2: Specification for Addition

Let us assume that this is written in terms of a monad Dynam with bind and unit operations $\star_D$ and $\mathbf{unit}_D$. Observe that this implementation-oriented definition calculates the same value as the standard definition, but it stores the intermediate value i as well. But where do addresses and storage come from? In [13], they were added to the Dynam monad using monad transformers[10, 25] as in the "Implementation-oriented" specification in Figure 2.1. In that definition, `rdAddr` reads the current top of stack address a, `inAddr` increments the top of stack, and `Thread` stores i at a. The monad (Dynam) is used to construct the domain containing both static and dynamic data.

In the "metacomputation"-style specification, two monads are used, Static, to encapsulate the static data, and Dynam to encapsulate the dynamic data. The meaning of the phrase is a metacomputation: the Static monad produces a computation of the Dynam monad. Clear separation of binding times is thus achieved. (In the examples, the dynamic parts of the computation are set in a box for emphasis.)

Figure 2.2 displays the specification for addition, which is similar to negation. Multiplication and subtraction are defined analogously.

2.2.2 Control-flow Compiler Building Block

We now present an example where separating binding times in specifications with metacomputations has a very significant advantage over the monolithic approach. Consider the three definitions of the conditional **if-then** statement in Figure 2.3. The first is a dual continuation "control-flow" semantics, found commonly in compilers[2]. If B is true, then the first continuation, $[\![c]\!] \star_D \kappa$, is

Control-Flow:

$$\begin{array}{l}
\mathsf{Dynam} = \mathcal{T}_{\mathsf{CPS}}\,\texttt{void}\,\mathsf{Id} \\
\mathit{Bool} = \forall\alpha.\alpha \times \alpha \to \alpha
\end{array}$$

$$\begin{array}{l}
\llbracket \textbf{if } b \textbf{ then } c \rrbracket : \mathsf{Dynam}(\texttt{void}) = \\
\quad \llbracket b \rrbracket \ \star_D \ \lambda B : \mathit{Bool}. \\
\quad \texttt{callcc}\ (\lambda\kappa. \\
\quad\quad B\langle \llbracket c \rrbracket \ \star_D \ \kappa, \kappa\rangle)
\end{array}$$

Implementation-oriented/Monolithic:

$$\begin{array}{l}
\mathsf{Dynam} = \mathcal{T}_{\mathsf{CPS}}\,\texttt{void}\,(\mathcal{T}_{\mathsf{St}}\,\mathit{Label}\,(\mathcal{T}_{\mathsf{St}}\,\mathit{Code}\,\mathsf{Id})) \\
\mathit{Label} = \mathit{int}, \mathit{Code} = \mathit{Label} \to \mathsf{Dynam}\,\texttt{void} \\
\texttt{jump}\,L = \texttt{getCode} \ \star_D \ \lambda\Pi : \mathit{Code}.(\texttt{callcc}\ \lambda\kappa.\Pi\,L) \\
\texttt{newlabel} : \mathsf{Dynam}(\mathit{Label}) = \\
\quad \texttt{getLabel} \ \star_D \ \lambda l : \mathit{Label}. \\
\quad \texttt{updateLabel}[L \mapsto L + 1] \ \star_D \ \lambda\_. \\
\quad \mathbf{unit}_D(l)
\end{array}$$

$$\begin{array}{l}
\mathcal{M}\mathit{ono}\llbracket \textbf{if } b \textbf{ then } c \rrbracket : \mathsf{Dynam}(\texttt{void}) = \\
\quad \mathcal{M}\mathit{ono}\llbracket b \rrbracket \ \star_D \ \lambda B : \mathit{Bool}. \\
\quad \texttt{newlabel} \ \star_D \ \lambda L_\kappa. \\
\quad \texttt{newlabel} \ \star_D \ \lambda L_c. \\
\quad \texttt{callcc}\ (\lambda\kappa. \\
\quad\quad \texttt{newSegment}(L_\kappa, \kappa) \ \star_D \ \lambda\_. \\
\quad\quad \texttt{newSegment}(L_c, \mathcal{M}\mathit{ono}\llbracket c \rrbracket \ \star_D \ (\texttt{jump}\,L_\kappa)) \ \star_D \\
\quad\quad\quad B\langle \texttt{jump}\,L_c, \texttt{jump}\,L_\kappa\rangle)
\end{array}$$

Metacomputation:

$$\begin{array}{l}
\mathsf{Dynam} = \mathcal{T}_{\mathsf{CPS}}\,\texttt{void}\ (\mathcal{T}_{\mathsf{St}}\,\mathit{Code}\,\mathsf{Id}), \mathsf{Static} = \mathcal{T}_{\mathsf{St}}\,\mathit{Label}\,\mathsf{Id} \\
\mathsf{IfThen} : \mathsf{Dynam}(\mathit{Bool}) \times \mathsf{Dynam}(\texttt{void}) \times \mathit{Label} \times \mathit{Label} \to \mathsf{Dynam}(\texttt{void}) \\
\mathsf{IfThen}(\varphi_B, \varphi_c, L_c, L_\kappa) = \\
\quad \varphi_B \ \star_D \ \lambda B : \mathit{Bool}. \\
\quad \texttt{callcc}\ (\lambda\kappa. \\
\quad\quad \texttt{updateCode}[L_\kappa \mapsto \kappa\bullet] \ \star_D \ \lambda\_. \\
\quad\quad \texttt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ \lambda\_.\texttt{jump}\,L_\kappa] \ \star_D \ \lambda\_. \\
\quad\quad\quad B\langle \texttt{jump}\,L_c, \texttt{jump}\,L_\kappa\rangle)
\end{array}$$

$$\begin{array}{l}
\mathcal{C}\llbracket \textbf{if } b \textbf{ then } c \rrbracket : \mathsf{Static}(\mathsf{Dynam}\,\mathit{Void}) = \\
\quad \texttt{newlabel} \ \star_S \ \lambda L_\kappa. \\
\quad \texttt{newlabel} \ \star_S \ \lambda L_c. \\
\quad \mathcal{C}\llbracket b \rrbracket \ \star_S \ \lambda\varphi_B. \\
\quad \mathcal{C}\llbracket c \rrbracket \ \star_S \ \lambda\varphi_c. \\
\quad\quad\quad \mathbf{unit}_S(\mathsf{IfThen}(\varphi_B, \varphi_c, L_c, L_\kappa))
\end{array}$$

Figure 2.3: **if-then**: 3 ways

executed, otherwise c is skipped and just κ is executed. A more implementation-oriented (informal) specification might be:

```
[[if b then c]] =
        [[b]] ⋆_D λB.
        "get two new labels L_c, L_κ" ⋆_D λ⟨L_c, L_κ⟩.
        callcc (λκ.
            "store κ at L_κ, then ([[c]] ⋆_D ("jump to L_κ")) at L_c" ⋆_D λ_.
            B⟨"jump to L_c", "jump to L_κ"⟩)
```

To formalize this specification, a technique from denotational semantics for modeling jumps is used. We introduce a continuation store, *Code*, and a label state *Label*. A jump to label L simply invokes the continuation stored at L. The second definition in Figure 2.3 presents an implementation-oriented specification of **if-then** in monolithic style (that is, where *Code* and *Label* are both added to Dynam). Again, this represents the approach in [13].

One very subtle problem remains: what is "newSegment"? One's first impulse is to define it as a simple update to the *Code* store (i.e., updateCode$[L_\kappa \mapsto \kappa\bullet]$), but here is where the monolithic approach greatly complicates matters. newSegment can not be a simple update to the *Code* store. Because the monolithic specification mixes static and dynamic computation, the continuation κ contains both kinds of computation. But because it is *stored* and not *executed*, κ will not have access to the current label count and any other static data necessary for proper staging. Therefore, newSegment must explicitly pass the current label count and any other static intermediate data structures to the continuation it stores. Furthermore, κ may have effects on the compile-time/static data (like incrementing the label counter) that should be reflected further on in the compilation[1].

The last specification in Figure 2.3 defines **if-then** as a metacomputation and is much simpler than the monolithic-style specification. Observe that Dynam does not include the *Label* store, and so the continuation κ now includes only dynamic computations. Therefore, there is no need to pass in the label count to κ, and so, κ may simply be stored in *Code*. **This is a central advantage of the metacomputation-based specification:** because of the separation of static and dynamic

[1] A full description of newSegment is found in [13].

$$
\begin{array}{l}
\mathcal{C}[\![e_1 \leq e_2]\!] : \mathsf{Static}(\mathsf{Dynam}\, Bool) = \\
\qquad \texttt{rdAddr} \star_S \lambda a. \\
\qquad \texttt{inAddr}\,(a+2) \\
\qquad\; \mathcal{C}[\![e_1]\!] \star_S \lambda \varphi_1. \\
\qquad\; \mathcal{C}[\![e_2]\!] \star_S \lambda \varphi_2. \\
\qquad\quad \mathbf{unit}_S \boxed{\left(\begin{array}{l}
\varphi_1 \star_D \lambda i : int. \\
\varphi_2 \star_D \lambda j : int. \\
\texttt{Thread}(i, a) \star_D \lambda v_1. \\
\texttt{Thread}(j, (a+1)) \star_D \lambda v_2. \\
\quad \mathbf{unit}_D(\lambda\langle \kappa_T, \kappa_F\rangle.((v_1 \leq v_2) \to \kappa_T, \kappa_F))
\end{array}\right)}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{C}[\![\mathbf{while}\; b\; \mathbf{do}\; c]\!] : \mathsf{Static}(\mathsf{Dynam}\, Void) = \\
\quad \texttt{newlabel} \star_S \lambda L_{test}. \\
\quad \texttt{newlabel} \star_S \lambda L_c. \\
\quad \texttt{newlabel} \star_S \lambda L_\kappa. \\
\quad \mathcal{C}[\![b]\!] \star_S \lambda \varphi_B. \\
\quad \mathcal{C}[\![c]\!] \star_S \lambda \varphi_c. \\
\qquad \mathbf{unit}_S \boxed{\left(\begin{array}{l}
\texttt{callcc}\, \lambda\kappa. \\
\;\; \texttt{updateCode}[L_\kappa \mapsto \kappa\bullet] \star_D \lambda\_. \\
\;\; \texttt{updateCode}[L_c \mapsto \varphi_c \star_D \lambda\_.\texttt{jump}\, L_{test}] \star_D \\
\;\; \texttt{updateCode}[L_{test} \mapsto \varphi_B \star_D \lambda B.(B\langle \texttt{jump}\, L_c, \texttt{jump}\, L_\kappa\rangle)] \star_D \lambda\_. \\
\;\;\; \texttt{jump}\, L_{test}
\end{array}\right)}
\end{array}
$$

Figure 2.4: Specification for $\leq$ and **while**

data into two monads, the complications outlined in Section 2.1 associated with storing command continuations in [13] (e.g., explicitly passing static data and use of a *snapback* operator[40]) are *completely* unnecessary.

Figure 2.4 contains the specifications for $\leq$ and **while**, which are very similar to the specifications of addition and **if-then**, respectively, that have been seen already.

2.2.3 Block Structure Compiler Building Block

The block structure language includes **new** x **in** c, which declares a new program variable x in c. The compiler building block for this language appears in Figure 2.5. The static part of this specification allocates a free stack location a, and the program variable x is bound to $\mathbf{unit}_S(a)$ in the current environment ρ. c is then compiled in the updated environment and larger stack $(a+1)$.

$$\mathsf{Static} = \mathcal{T}_{\mathsf{Env}}\, Env\, (\mathcal{T}_{\mathsf{Env}}\, Addr\, \mathsf{Id}),\ \ \mathsf{Dynam} = \mathcal{T}_{\mathsf{St}}\, Sto\, \mathsf{Id}$$

$\mathcal{C}[\![x_{rval}]\!] : \mathsf{Static}(\mathsf{Dynam}(int))$

$\mathcal{C}[\![x_{rval}]\!] = \mathtt{rdEnv} \star_S\ \lambda\rho.(\rho\, x)\ \star_S\ \lambda a.\ \mathbf{unit}_S(\mathtt{rdLoc}(a))$

$\mathcal{C}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!] : \mathsf{Static}(\mathsf{Dynam}\, Void)$

$\mathcal{C}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!] = \mathtt{rdAddr}\ \star_S\ \lambda a.\mathtt{inAddr}\,(a+1)(\mathtt{rdEnv}\ \star_S\ \lambda\rho.\mathtt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_S(a)])\ \mathcal{C}[\![c]\!])$

Figure 2.5: Compiler Building Block for Block Structure

$$\mathsf{Static} = \mathcal{T}_{\mathsf{Env}}\, Env\, \mathsf{Id},\ \ \mathsf{Dynam} = \mathcal{T}_{\mathsf{St}}\, Sto\, \mathsf{Id}$$

$\mathcal{C}[\![c : \mathbf{comm}]\!] : \mathsf{Static}(\mathsf{Dynam}\, Void)$

$\mathcal{C}[\![c_1 ; c_2]\!] = \mathcal{C}[\![c_1]\!]\ \star_S\ \lambda\varphi_{c_1}.\mathcal{C}[\![c_2]\!]\ \star_S\ \lambda\varphi_{c_2}.\mathbf{unit}_S(\varphi_{c_1}\ \star_D\ \lambda\_.\varphi_{c_2})$

$\mathcal{C}[\![x := e]\!] = \mathtt{rdEnv}\ \star_S\ \lambda\rho.(\rho\, x)\ \star_S\ \lambda a.\mathcal{C}[\![e]\!]\ \star_S\ \lambda\varphi_e.\mathbf{unit}_S(\varphi_e\ \star_D\ \lambda i : int.\mathtt{updateSto}[a \mapsto i])$

Figure 2.6: Compiler Building Block for Imperative Features

2.2.4 Imperative Features Compiler Building Block

The simple imperative language includes assignment (:=) and sequencing (;). The compiler building block for this language appears in Figure 2.6. For sequencing, the static part of the specification compiles c_1 and c_2 in succession, while the dynamic part runs them in succession. For assignment, the static part of the specification retrieves the address a for program variable x from the current environment ρ and compiles e, while the dynamic part calculates the value of e and assigns that value to location a.

2.3 Combining Compiler Building Blocks

Figure 2.7 illustrates the process of combining the compiler building blocks for the block structure and control-flow languages. It is important to emphasize that this is much simpler than in [13], in that there is no explicit passing of static data needed. The process is nothing more than applying

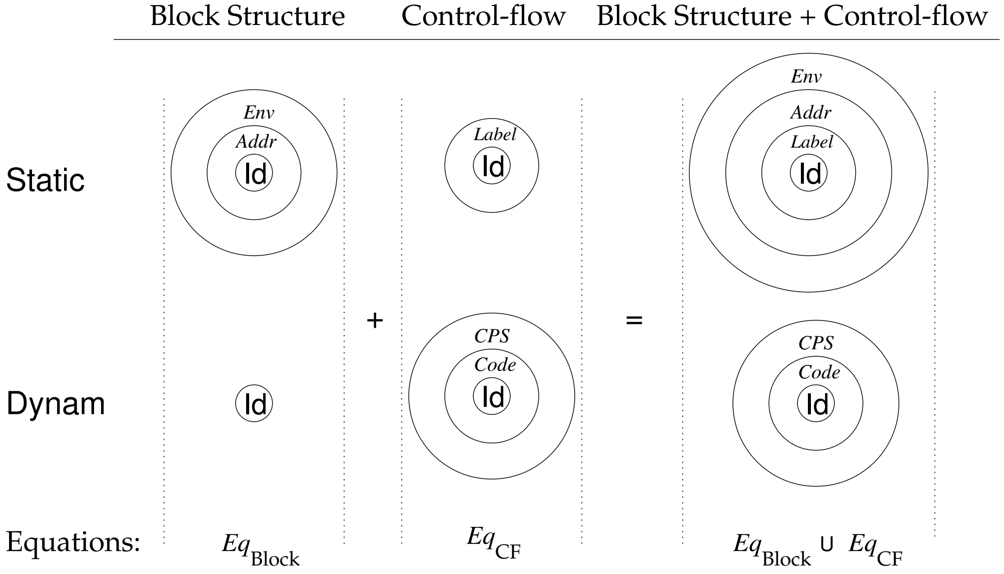


Figure 2.7: Combining Compiler Building Blocks

the appropriate monad transformers to create the Static and Dynam monads for the combined language. Recall that for the block structure language:

$$\mathsf{Static} = \mathcal{T}_{\mathsf{Env}}\, \mathit{Env}\, (\mathcal{T}_{\mathsf{Env}}\, \mathit{Addr}\, \mathsf{Id}),\ \text{and}\ \mathsf{Dynam} = \mathsf{Id}$$

For the control flow language:

$$\mathsf{Static} = \mathcal{T}_{\mathsf{St}}\, \mathit{Label}\, \mathsf{Id},\ \text{and}\ \mathsf{Dynam} = \mathcal{T}_{\mathsf{CPS}}\, \mathtt{void}\, (\mathcal{T}_{\mathsf{St}}\, \mathit{Code}\, (\mathcal{T}_{\mathsf{St}}\, \mathit{Sto}\, \mathsf{Id}))$$

To combine the compiler building blocks for these languages, one simply combines the respective monad transformers:

$$\mathsf{Static} = \mathcal{T}_{\mathsf{Env}}\, \mathit{Env}\, (\mathcal{T}_{\mathsf{Env}}\, \mathit{Addr}\, (\mathcal{T}_{\mathsf{St}}\, \mathit{Label}\, \mathsf{Id})),\ \text{and}\ \mathsf{Dynam} = \mathcal{T}_{\mathsf{CPS}}\, \mathtt{void}\, (\mathcal{T}_{\mathsf{St}}\, \mathit{Code}\, (\mathcal{T}_{\mathsf{St}}\, \mathit{Sto}\, \mathsf{Id}))$$

Now, the specifications for both of the smaller languages, Eq_{Block} and Eq_{CF}, apply for the "larger" Static and Dynam monads, and so the compiler for the combined language is specified by $Eq_{Block} \cup Eq_{CF}$.

Code is generated via type-directed partial evaluation[7] using the method of Danvy and

Compiler:

$\mathsf{Dynam} = \mathcal{T}_{\mathsf{CPS}}\,\texttt{void}\,(\mathcal{T}_{\mathsf{St}}\,\mathit{Code}\,(\mathcal{T}_{\mathsf{St}}\,\mathit{Sto}\,\mathsf{Id}))$, $\mathsf{Static} = \mathcal{T}_{\mathsf{Env}}\,\mathit{Env}\,(\mathcal{T}_{\mathsf{Env}}\,\mathit{Addr}\,(\mathcal{T}_{\mathsf{St}}\,\mathit{Label}\,\mathsf{Id}))$
Language = Expressions ∪ Imperative ∪ Control-flow ∪ Block structure ∪ Booleans
Equations = $Eq_{\text{Expressions}} \cup Eq_{\text{Imperative}} \cup Eq_{\text{Control-flow}} \cup Eq_{\text{Block structure}} \cup Eq_{\text{Booleans}}$

Source Code:

new x **in** **new** y **in**
x := 5; y := 1;
while $(1 \leq x)$ **do**
y := y*x; x := x-1;

Target Code:

```
       0 := 5;                    2:  2 := [1];                 3:  halt;
       1 := 1;                        3 := [0];
       jump 1;                        1 := [2] * [3];
                                      2 := [0];
   1:  2 := 1;                        3 := 1;
       3 := [0];                      0 := [2] - [3];
       BRLEQ [2] [3] 2 3;             jump 1;
```

Figure 2.8: Compiler for While language and example compilation

Vestergaard[8]. Figure 2.8 contains the compiler for the while language, and an example program and its pretty-printed compiled version. All that was necessary was to combine the compiler building blocks developed in this section combined as discussed in Section 2.3.

2.4 The Run-time System for Subroutines

In this Section, the necessary runtime support for the procedure compilation techniques described in Section 2.5 is outlined. This consists mainly of three things:

- displays and display addressing[1, 2]
- stack descriptors[43, 8], and
- the MachLang instructions `call`, `acall`, and `return`.

A *display* (cf. Figure 2.9) is a way of organizing memory access in the runtime system of a compiler. The display D is an array of pointers into memory. Accessing the store location loc is accomplished by first dereferencing the pointer $D[f]$ and then counting up by displacement d. The physical address of loc is, thus, $D[f] + d$. The *display address* of loc is the pair $\langle f, d\rangle$.

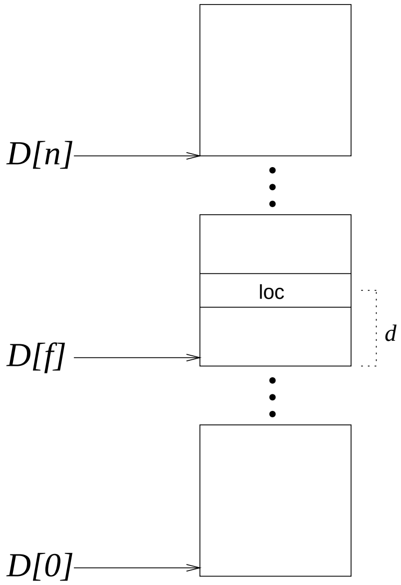


Figure 2.9: Referencing Store Location loc with Display D

It is necessary in the compilation of procedures to *not* have fixed addresses for procedure arguments. Consider the definition of factorial in a suitable functional language:

```
fun fact n = if n=0 then 1 else n*(fact (n-1));
```

In evaluating "`fact 5`", there will be six different invocations of `fact`, each with its own version of the argument `n` (namely, 0, . . . , 5). Generally, the different versions of the formal arguments to procedures must be stored at different addresses, and this is the main purpose of display addressing: the formal argument to a procedure may be stored at display address $\langle f, d \rangle$, but $\langle f, d \rangle$ will generally refer to different actual addresses throughout program execution.

At any point in program execution, if f is the largest number for which the display $D[f]$ is defined and if in the activation record pointed to by $D[f]$ there are a total of d storage locations used, then the run-time stack is said to have *stack shape* $\langle f, d \rangle$. In this situation, $\langle f, d \rangle$ is said to be the *stack descriptor*. Stack descriptors are useful for determining the next free address in store, which is $\langle f, d+1 \rangle$.

In this section, a particular format for activation records is presented, and it is shown how `call`, `acall`, and `return` affect the run-time stack. It is a deliberate design decision to leave unfixed the particular representation of activations, since there are many equivalent such representations and

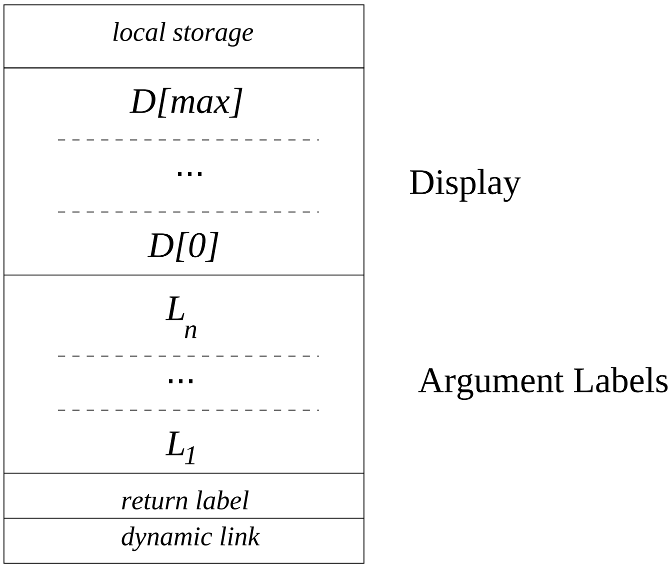


Figure 2.10: Activation Record for `call` L_p f $[L_1, \ldots, L_n]$ L_{ret}

it would diminish the generality of the compilation technique to fix upon any particular one. To aid the reader's understanding, however, a specific representation of activation records and the run-time stack is given below and the actions of `call`, `acall`, and `return` are specified.

Activation Records

Figure 2.10 presents a representation for activation records. Activation records are created by calls:

$$\texttt{call}\ L_p\ f\ [L_1, \ldots, L_n]\ L_{ret}$$

The *dynamic link* points to the topmost activation record before the `call` was executed (i.e., the activation record of the caller), and is used to return from a subroutine call. The *return label* field is L_{ret} in this example and indicates the code to be executed after the subroutine call is through. The "Argument Labels" fields are filled in with the argument labels $L_1, \ldots, L_n$. The "Display" fields contain the *previous* display contents from before the subroutine call. Finally, local storage starts at the top of the activation record, since it may grow or shrink throughout the execution of the call.

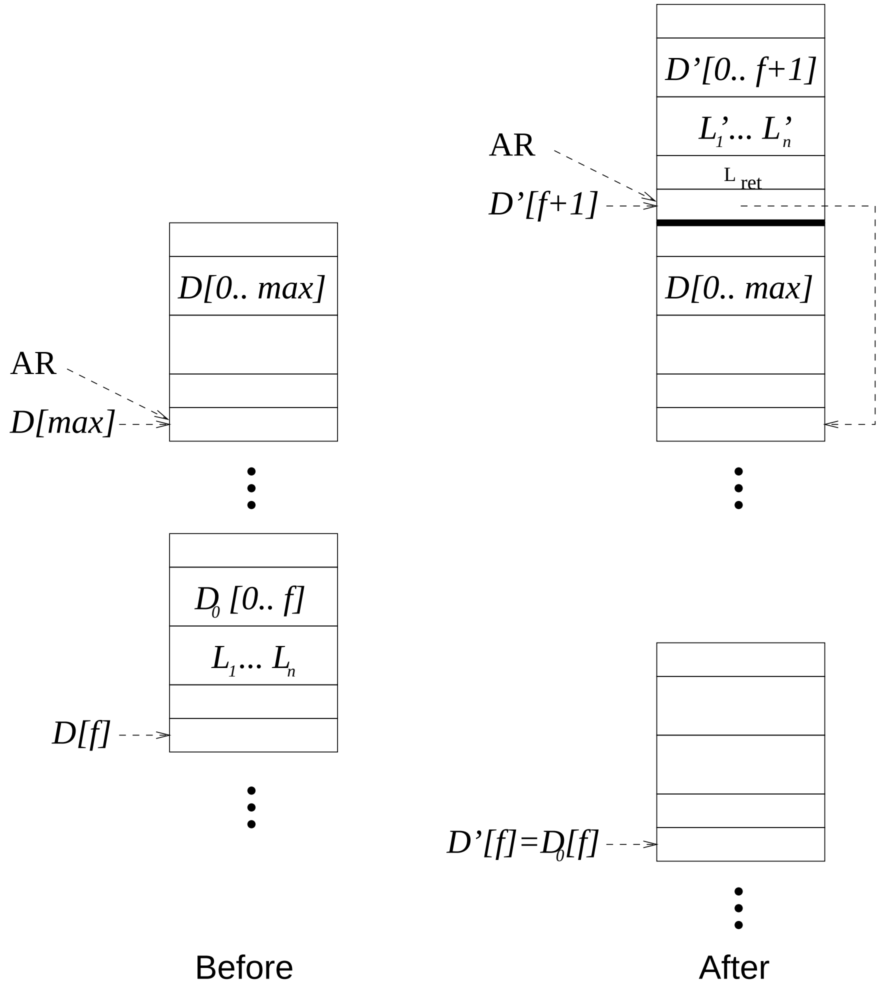


Figure 2.11: The effect on the stack of: `call` $L\ f\ [L'_1 \ldots L'_n]\ L_{ret}$

Effect of `call` and `acall` on the run-time stack

Figure 2.11 presents a "before and after" picture of a call to subroutine i stored at L. i is presumed to have a nesting level $f+1$ (i.e., it may contain references to display locations $\langle h, d\rangle$ for $0 \le h \le f+1$). We assume that there is a register AR which points to the topmost activation in the frame list. Observe that the current display D is stored in the topmost activation.

The effect of `call` $L\, f\, [L'_1 \ldots L'_n]\, L_{ret}$ on the stack configuration labelled Before in Figure 2.11 is:

1. A new activation record is created on top of the stack with the dynamic link set to the current

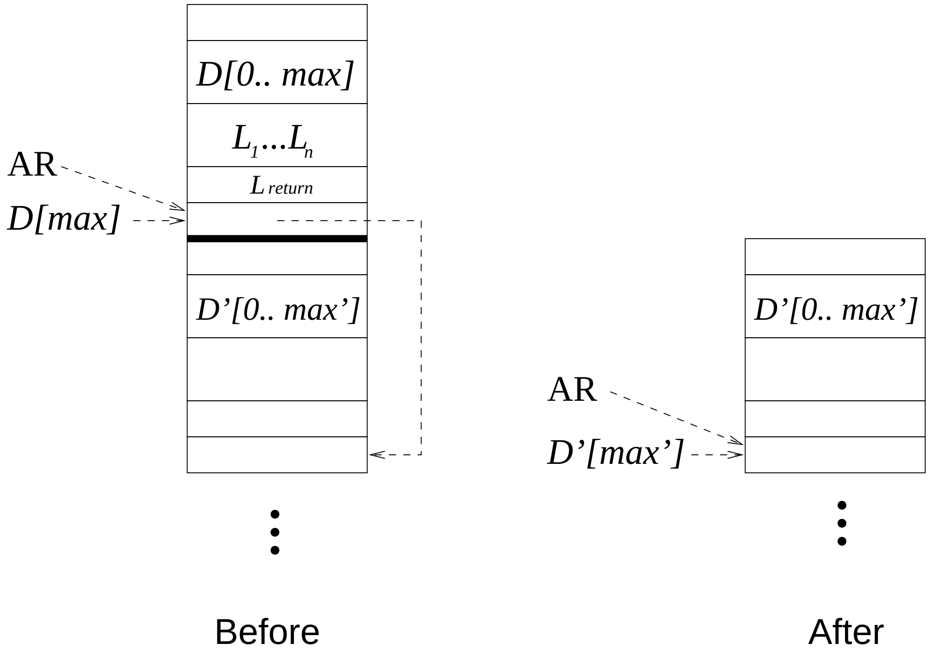


Figure 2.12: The effect on the stack of `return`

activation, an appropriate return label, and argument labels $L'_1 \dots L'_n$.

2. The new display D' is identical to the display D_0 except that it has an additional entry $D'[f+1]$ pointing to the new activation. Observe that D_0 is in the activation pointed to by $D[f]$. D' is inserted into the new activation.

3. AR is set to this new activation, the display is set to D', and control is sent to i.

The effect of `acall` $j\, f\, [L'_1 \dots L'_n]\, L_{ret}$ is identical to the above, except that control is sent to label L_j, which is the jth label pointed to by $D[f]$. The `acall` instruction is used to call procedure arguments. For example, after executing `call` $L\, f\, [L_1 \dots L_n]\, L_{ret}$, the code at L may call the subroutine at L_i using an `acall` instruction.

Effect of `return` on the run-time stack

The effect of `return` is shown in Figure 2.12 and follows these steps:

1. Dereferencing the dynamic link gives the activation of the calling subroutine.

```
        call 101 0 [103] 102

102:    <0,0> := SBRS

103:    <1,1> := #777
        <1,0> := -<1,1>
        ldreg SBRS,-<1,0>
        return

101:    acall 1 1 [] 100

100:    <1,0> := SBRS
        ldreg SBRS,<1,0>
        return

Answer is: <0,0>
```

Figure 2.13: Compiling: `(letclosed id(x) = x in (funcall id --777))`

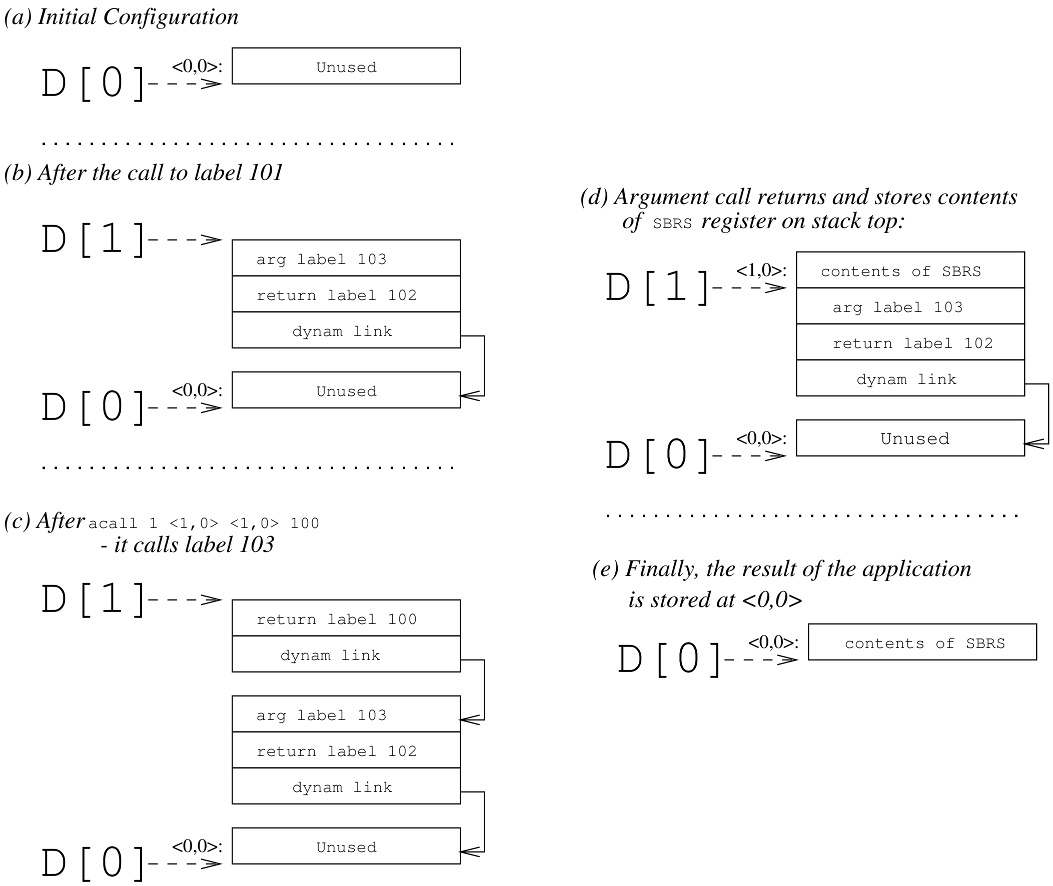
2. The previous display must be restored before returning, so the display is set to D'.

3. AR is set to the activation of the calling subroutine.

4. The stack is popped, and control is sent to L_{return} for `return` and L_i for `ajump` L_i.

Sample Machine Language Program Execution

Figure 2.13 gives an example machine language program which was produced by the compiler block for procedures which is defined in the next section. While that RCBB has not been defined yet, it is instructive to go through the output code to understand how the runtime system works. The particular program compiled in Figure 2.13 is the identity function on integers. Initially, the runtime stack appears as in Figure 2.14(a). Execution begins with the `call 101 0 [103] 102` instruction. This instruction creates a new activation record (cf. Figure 2.14(b)) and passes control to the code at label `101`. The code at label `101` is an `acall` to get the value of the argument `--777`. `acall 1 1 [] 100` calls the first argument label in the activation record pointed to by $D[1]$—that is, `103`. Executing the `acall` creates a new activation record pointed to by $D[1]$, as is shown in Figure 2.14(c). The code at label `103` calculates `--777`, puts the result in a register `SBRS`, and returns. Executing `return` means popping the top activation record off the stack (as in Figure 2.14(d)) and sending control to the return label `100`. The code at `100` essentially just



(a) Initial Configuration

(b) After the call to label 101

(c) After `acall 1 <1,0> <1,0> 100` *- it calls label 103*

(d) Argument call returns and stores contents of SBRS *register on stack top:*

(e) Finally, the result of the application is stored at <0,0>

Figure 2.14: Action of Figure 2.13 on run-time stack

returns again, this time passing control to the code at `102`. Executing the code at `102` simply saves the value of the procedure call in $\langle \mathtt{0}, \mathtt{0} \rangle$. Now, the stack is as portrayed in Figure 2.14(e).

2.5 Compiling Procedures

There are two ways to handle the compilation of procedures generally. The first is to inline (i.e., substitute) the procedure body at each procedure call, in which case the procedure is called *open*. Compiling a procedure application `(funcall (proc (x) (x + x)) 2)` (where `(proc (x) (x + x))` is treated as an open procedure) would inline the procedure body `(x + x)` with argument `2`, thus producing the same target language code as `(2 + 2)` does. As in [43, 13], the compilation semantics for open procedures is *identical* to the usual monadic semantics for the call-by-name λ-calculus. The second method is to translate procedure bodies into target code stored at labels and procedure applications into target language subroutine calls to those labels. Such procedures are called *closed* procedures. Closed procedures are necessary for compiling recursive bindings (since inlining recursive procedures does not terminate). We introduce special syntax, **letopen** and **letclosed**, to distinguish open and closed procedure declarations, respectively.

It will be seen that very few changes need to be made to the standard semantics for procedures (i.e., the lambda calculus semantics) to produce the compilation semantics. One change, though, is that locations in store will now be specified as *display addresses* rather than as the simple fixed addresses used so far. This is to be expected as most traditional compilation schemes for procedures require some means of keeping track of multiple procedure activations[1]. Section 2.4 describes the the runtime system which is assumed here. A procedural language is given—the integer-valued, higher-order function language *FunExp*—to demonstrate our procedural compilation technique. *FunExp* extends integer expressions, and it clarifies our procedure compilation technique to show how the compilation of *FunExp* terms interacts with the compilation of integer expressions. Finally, the compilation semantics for *FunExp* is introduced.

To use display addresses, one need only change the "free address type" from addresses $Addr = int$ to stack descriptors $SD = int \times int$ (for a discussion of stack descriptors, see Section 2.4). The changes to the previous compiler block definitions can be summarized as:

Previous Defn.	Replaced By.
$\mathcal{T}_{\mathsf{Env}}\ Addr$	$\mathcal{T}_{\mathsf{Env}}\ SD$
`rdAddr`	`rdSD`
`inAddr`	`inSD`
$(a + disp)$	$(S + disp)$

where $\langle f, d\rangle + disp = \langle f, d + disp\rangle$. Changing the version of negation in Figure 2.1 to use stack shapes yields:

$$\begin{aligned}\mathcal{C}[\![-e]\!] = \ & \mathtt{rdSD} \star_S \lambda S. \\ & (\mathtt{inSD}\ (S+1)\ \mathcal{C}[\![e]\!]) \star_S \lambda \varphi_e. \\ & \quad \mathbf{unit}_S \left(\begin{array}{l} \varphi_e \star_D \lambda i. \\ \mathtt{Thread}(i, S) \star_D \lambda v. \\ \qquad \mathbf{unit}_D(-v) \end{array} \right)\end{aligned}$$

For a procedure source language, the language of higher-order, integer-valued function expressions *FunExp* is used. Its syntax and typing rules are summarized in Definition 5. To distinguish integer expressions from integer values *int*, the type **intexp** is introduced for the syntactic or phrase type of integer expressions below.

Definition 5 *The Functional Expression language FunExp is:*

$$FunExp ::= \mathbf{funcall}\ f\ e_1 \ldots e_n \ \mid\ \mathbf{letclosed}\ f(x_1 : \varphi_1, \ldots, x_n : \varphi_n) = e\ \mathbf{in}\ e'$$

for types $\varphi ::= \mathbf{intexp} \mid \varphi \rightarrow \varphi$ *and has typing rules:*

$$\frac{\Gamma \vdash f : \varphi_1 \rightarrow \ldots \rightarrow \varphi_n \rightarrow \mathbf{intexp} \quad \Gamma \vdash e_1 : \varphi_1 \ \ldots\ \Gamma \vdash e_n : \varphi_n}{\Gamma \vdash (\mathbf{funcall}\ f\ e_1 \ldots e_n) : \mathbf{intexp}}$$

$$\frac{\Gamma\{x_1 : \varphi_1, \ldots, x_n : \varphi_n\} \vdash e : \mathbf{intexp} \quad \Gamma\{f : \varphi_1 \to \ldots \to \varphi_n \to \mathbf{intexp}\} \vdash e' : \mathbf{intexp}}{\Gamma \vdash (\mathbf{letopen}\ f(x_1 : \varphi_1, \ldots, x_n : \varphi_n) = e\ \mathbf{in}\ e') : \mathbf{intexp}}\ (x_i \text{ distinct})$$

$$\frac{\Gamma\{x_1 : \varphi_1, \ldots, x_n : \varphi_n\} \vdash e : \mathbf{intexp} \quad \Gamma\{f : \varphi_1 \to \ldots \to \varphi_n \to \mathbf{intexp}\} \vdash e' : \mathbf{intexp}}{\Gamma \vdash (\mathbf{letclosed}\ f(x_1 : \varphi_1, \ldots, x_n : \varphi_n) = e\ \mathbf{in}\ e') : \mathbf{intexp}}\ (x_i \text{ distinct})$$

Observe that the only terms of higher type in this language are introduced by a **letopen** or **letclosed**. Because there is only **funcall** rather than the more general function application, all higher-typed terms must be explicitly named.

FunExp extends the language of integer expressions by allowing higher-order function definitions with open and closed procedures and function calls with **funcall**. *FunExp* is a powerful enough language to demonstrate how procedures are compiled within the metacomputation-style framework. As a starting point, the standard semantics of **letopen**, **letclosed**, and **funcall** is considered first, which is nothing but the standard call-by-name semantics for the lambda calculus as was mentioned previously. Then, the compilation semantics for the simplest example of a closed integer-valued procedure is defined—namely, a procedure of type **intexp**. Once it is understood how to introduce `call` and `return` for procedures of type **intexp**, adding arguments is a simple matter.

Definition 6 specifies two helper functions, apply and Lambda which are used throughout this Section. apply and Lambda are syntactic sugar for the usual, call-by-name semantics for application and lambda abstraction. For example, if $\lambda x.f : \mathbf{intexp} \to \mathbf{intexp}$ and $e : \mathbf{intexp}$, then

$$\begin{array}{lclcl} \mathsf{apply}\ [\![\lambda x.f]\!]\ [\![e]\!] & \triangleq & \mathtt{rdEnv} \star \lambda\rho.[\![\lambda x.f]\!] \star \lambda p.\ p(\mathtt{inEnv}\ \rho\ [\![e]\!]) & \triangleq & [\![(\lambda x.f)\ e]\!] \\ \mathsf{Lambda}(x, [\![f]\!]) & \triangleq & \mathtt{rdEnv} \star \lambda\rho.\mathbf{unit}(\lambda c.\mathtt{inEnv}\ \rho[x \mapsto c]\ [\![f]\!]) & \triangleq & [\![\lambda x.f]\!] \end{array}$$

Definition 6 (apply **and** Lambda) *For* $\varphi = \varphi_1 \to \ldots \to \varphi_n \to \mathbf{intexp}$*,*

$$\mathsf{apply}_{\varphi} : \mathsf{M}(\varphi) \to \mathsf{M}(\varphi_1) \to \ldots \to \mathsf{M}(\varphi_n) \to \mathsf{M}(int)$$

$$\begin{aligned}\mathsf{apply}_{\varphi}\, t\, x_1 \ldots x_n = \;& \texttt{rdEnv} \star \lambda\rho. \\ & t \star \lambda p_1. \\ & (p_1(\texttt{inEnv}\, \rho\, x_1)) \star \lambda p_2. \\ & \qquad\qquad \vdots \\ & (p_{n-1}(\texttt{inEnv}\, \rho\, x_{n-1})) \star \lambda p_n. \\ & \quad (p_n(\texttt{inEnv}\, \rho\, x_n))\end{aligned}$$

$$\mathsf{Lambda}(i, x) = \texttt{rdEnv} \star_S \; \lambda\rho.\mathbf{unit}_S(\lambda c.\texttt{inEnv}\, \rho[i \mapsto c]\, x)$$

Definition 7 gives the standard semantics for $FunExp$. It is nothing more than the usual monadic semantics for the call-by-name λ-calculus. Observe that the standard semantics for **letopen** and **letclosed** are identical.

Definition 7 (Standard Semantics of funcall, letopen, and letclosed)

$$[\![\mathbf{letopen}\; f(x_1 : \varphi_1, \ldots, x_n : \varphi_n) = e \;\mathbf{in}\; e']\!] = \texttt{rdEnv} \star \lambda\rho.\; (\texttt{inEnv}\, \rho[f \mapsto [\![\lambda x_1. \ldots \lambda x_n.e]\!]]\; [\![e']\!])$$

$$[\![\mathbf{letclosed}\; f(x_1 : \varphi_1, \ldots, x_n : \varphi_n) = e \;\mathbf{in}\; e']\!] = \texttt{rdEnv} \star \lambda\rho.\; (\texttt{inEnv}\, \rho[f \mapsto [\![\lambda x_1. \ldots \lambda x_n.e]\!]]\; [\![e']\!])$$

$$[\![\mathbf{funcall}\; f\, e_1 \ldots e_n]\!] = \mathsf{apply}_{\varphi}\, [\![f]\!]\, [\![e_1]\!] \ldots [\![e_n]\!]$$

$$[\![\lambda x.f]\!] = \texttt{rdEnv} \star_S \; \lambda\rho.\mathbf{unit}_S(\lambda c.\texttt{inEnv}\, \rho[x \mapsto c]\; [\![f]\!])$$

where $\varphi = \varphi_1 \to \ldots \to \varphi_n \to \mathbf{intexp}, f : \varphi,$ *and* $e_i : \varphi_i$

The defining equations of the compilation semantics for open procedures (given below in Definition 8) are identical to the standard semantics for open procedures. An example compilation is presented in Figure 2.15.

Definition 8 (Compilation Semantics for funcall and letopen)

$$\mathcal{C}[\![\textbf{letopen}\, f(x_1 : \varphi_1, \ldots, x_n : \varphi_n) = e \textbf{ in } e']\!] = \texttt{rdEnv} \;\star_S\; \lambda\rho.\,(\texttt{inEnv}\,\rho[f \mapsto \mathcal{C}[\![\lambda x_1.\ldots\lambda x_n.e]\!]]\,\mathcal{C}[\![e']\!])$$

$$\mathcal{C}[\![\textbf{funcall}\, f\, e_1 \ldots e_n]\!] = \mathsf{apply}_\varphi\, \mathcal{C}[\![f]\!]\, \mathcal{C}[\![e_1]\!] \ldots \mathcal{C}[\![e_n]\!]$$

$$\mathcal{C}[\![\lambda x.f]\!] = \texttt{rdEnv} \;\star_S\; \lambda\rho.\textbf{unit}_S(\lambda c.\texttt{inEnv}\,\rho[x \mapsto c]\,\mathcal{C}[\![f]\!])$$

where $\varphi = \varphi_1 \to \ldots \to \varphi_n \to \textbf{intexp}, f : \varphi,$ *and* $e_i : \varphi_i$

We now consider the compilation of closed procedures into "call and return" code beginning with the simplest case of a closed procedure—that is, a closed procedure of type **intexp**. After that, it is shown how to compile procedures with arguments.

The key to compiling an integer expression e as a closed procedure is that the code for e is stored at a label L_e in the code store, and then a *call* to L_e is made rather than executing the code for e "in place" as before. Furthermore, the value produced by the code for e is returned to the caller via a special register named SBRS (<u>s</u>u<u>b</u>
<u>r</u>outine <u>r</u>eturn<u>s</u>). Semantically, the SBRS register can be modeled equally well by either including it as part the value store *Sto* or by applying the state monad transformer ($\mathcal{T}_{\mathsf{St}}\, int$) to the Dynam monad. Whichever implementation is chosen, it is assumed that there are operations setSBRS : $int \to$ Dynam(void) and getSBRS : Dynam(int) which set and read the current value of the SBRS register, respectively. Following Reynolds[43], source language procedures are distinguished from the target language *subroutines* which represent them. The process of making a subroutine for an integer expression procedure e is as follows:

1. When the subroutine for e is called, it will be executed in a stack with one more activation record, so if the current stack shape is $\langle f, d\rangle$, then compile e in the stack shape $\langle f+1, 0\rangle$. Call π_e : Dynam(int) the result of compiling e.

2. The subroutine for e first executes π_e, then stores the resulting integer in the SBRS register, and executes a return.

Steps (1) and (2) can be neatly summarized as the function $\texttt{mksubr}_{\textbf{intexp}}$, whose specification is found in Definition 9.

Definition 9 (`mksubr` part I)

$$\texttt{mksubr}_{\textbf{intexp}} : SD \rightarrow \mathsf{Static}(\mathsf{Dynam}(int)) \rightarrow \mathsf{Static}(\mathsf{Dynam}(\texttt{void}))$$

$$\texttt{mksubr}_{\textbf{intexp}}\ S\ p = \ (\texttt{inSD}\ S^{+}\ p)\ \star_S\ \lambda f. \ \textbf{unit}_S \begin{pmatrix} f\ \star_D\ \lambda result. \\ (\texttt{setSBRS}\ result)\ \star_D\ \lambda\_. \\ \quad \texttt{return} \end{pmatrix}$$

where $\langle f, d\rangle^{+} = \langle f+1, 0\rangle$.

In the standard semantics for (**letclosed** $x = e$ **in** e') (shown above in Definition 7), the variable x is simply bound to the meaning of e in $[\![e']\!]$. The compilation semantics for (**letclosed** $x = e$ **in** e') binds x to a computation which produces a call to L_e instead. Of course, the compilation semantics must make a subroutine for e and store it at L_e as well. So, the compilation semantics for **letclosed** for procedures of type **intexp** is:

$$
\begin{aligned}
&\mathcal{C}[\![\textbf{letclosed}\ x : \textbf{intexp}\ = e\ \textbf{in}\ e']\!] = \\
&\qquad \texttt{newlabel}\ \star_S\ \lambda L_e. \\
&\qquad \texttt{rdEnv}\ \star_S\ \lambda\rho. \\
&\qquad \texttt{rdSD}\ \star_S\ \lambda S. \\
&\qquad (\texttt{mksubr}_{\textbf{intexp}}\ S\ \mathcal{C}[\![e]\!])\ \star_S\ \lambda\pi_e. \\
&\qquad\quad (\texttt{inEnv}\ \rho[x \mapsto (\texttt{mkcall}_{\textbf{intexp}}\ S\ L_e)] \begin{pmatrix} \mathcal{C}[\![e']\!]\ \star_S\ \lambda\pi_{e'}. \\ \textbf{unit}_S(\texttt{updateCode}[L_e \mapsto \pi_e]\ \star_D\ \lambda\_.\pi_{e'}) \end{pmatrix})
\end{aligned}
$$

In this definition, a new label L_e is generated to store the subroutine for e, then $\texttt{mksubr}_{\textbf{intexp}}$ is used to create the subroutine π_e implementing e. The body of the **letclosed**, e', is then compiled to $\pi_{e'}$ in an environment in which a reference to x in e' will generate a call to L_e. The dynamic part of the metacomputation stores π_e in the code store at label L_e, and then executes the code for e', $\pi_{e'}$. But what is $\texttt{mkcall}_{\textbf{intexp}}\ S\ L_e$?

$\texttt{mkcall}_{\textbf{intexp}}\ S\ L_e$ is a function that must generate a call to L_e whenever x is referenced in e'.

For each reference to x in e', a new return label will have to be generated to represent the different places in the code to which the subroutine for e will return. Furthermore, x may be referenced from within nested definitions (e.g., ($\textbf{letclosed}\ x = e\ \textbf{in}\ \ldots (\textbf{letclosed}\ y = e_0\ \textbf{in}\ x + y) \ldots)$), and so it will be necessary to determine which activation records must be visible from the subroutine for e. If the current stack shape is $\langle f, d \rangle$ when the procedure e is defined, then the first f activation records should be visible during the execution of subroutine π_e. The definition of $\texttt{mkcall}_{\textbf{intexp}}$ is:

Definition 10 (`mkcall` part I)

$$\texttt{mkcall}_{\textbf{intexp}} : SD \rightarrow \mathit{Label} \rightarrow \mathsf{Static}(\mathsf{Dynam}(\mathit{int}))$$

$$\texttt{mkcall}_{\textbf{intexp}}\ \langle f, d \rangle\ L =$$

$$\texttt{newlabel} \star_S\ \lambda L_{ret}.$$

$$\textbf{unit}_S \left(\begin{array}{l} \texttt{callcc}\ \lambda\beta : \mathit{int} \rightarrow \mathsf{Dynam}(\texttt{void}). \\ \quad \texttt{updateCode}[L_{ret} \mapsto (\texttt{getSBRS} \star_D\ \beta)] \star_D\ \lambda\_. \\ \qquad \texttt{call}\ L\ f\ [\,]\ L_{ret} \end{array} \right)$$

Here, $\texttt{mkcall}_{\textbf{intexp}}\ \langle f, d \rangle\ L$ generates a return label L_{ret}. The return code is determined by getting the current continuation β, which will have type $\mathit{int} \rightarrow \mathsf{Dynam}(\texttt{void})$ since this call is being made from within an integer expression, and passing β the value in the SBRS register: $\texttt{getSBRS} \star_D\ \beta$. Finally, the call ($\texttt{call}\ L\ f\ [\,]\ L_{ret}$) is given.

As an example of how all of this fits together, consider the compilation of the simple *FunExp* expression: $\textbf{letclosed}\ i = 1\ \textbf{in}\ (\textbf{funcall}\ i)$. The compiled code for this program should generate a call to a subroutine which returns the value 1 via the SBRS register. By Definition 12:

$$
\begin{aligned}
&\mathcal{C}[\![\textbf{letclosed}\ i = 1\ \textbf{in}\ (\textbf{funcall}\ i)]\!] = \\
&\qquad \texttt{newlabel} \star_S\ \lambda L_i. \\
&\qquad \texttt{rdEnv} \star_S\ \lambda \rho. \\
&\qquad \texttt{rdSD} \star_S\ \lambda S. \\
&\qquad (\texttt{mksubr}_{\textbf{intexp}}\ S\ \mathcal{C}[\![1]\!]) \star_S\ \lambda \pi_i. \\
&\qquad\quad (\texttt{inEnv}\ \rho[i \mapsto (\texttt{mkcall}_{\textbf{intexp}}\ S\ L_i)] \begin{pmatrix} \mathcal{C}[\![\textbf{funcall}\ i]\!] \star_S\ \lambda \pi_{\mathsf{fc}}. \\ \textbf{unit}_S(\texttt{updateCode}[L_i \mapsto \pi_i] \star_D\ \lambda\_.\pi_{\mathsf{fc}}) \end{pmatrix})
\end{aligned} \tag{2.1}
$$

Notice that $\mathcal{C}[\![\textbf{funcall}\ i]\!]$ in Equation 2.1 can be simplified to:

$$
\begin{aligned}
&\texttt{newlabel} \star_S\ \lambda L_{ret}. \\
&\textbf{unit}_S \begin{pmatrix} \texttt{callcc}\ \lambda\beta : int \to \mathsf{Dynam}(\texttt{void}). \\ \quad (\texttt{updateCode}\ L_{ret}\ (\texttt{getSBRS} \star_D\ \beta)) \star_D\ \lambda\_. \\ \qquad \texttt{call}\ L_i\ f\ [\,]\ L_{ret} \end{pmatrix}
\end{aligned} \tag{2.2}
$$

if $S = \langle f, d \rangle$. By Definition 9:

$$
\texttt{mksubr}_{\textbf{intexp}}\ S\ \mathcal{C}[\![1]\!] =
$$

$$
\begin{aligned}
&(\texttt{inSD}\ S^+\ \mathcal{C}[\![1]\!]) \star_S\ \lambda \pi_1 : \mathsf{Dynam}(int). \\
&\textbf{unit}_S \begin{pmatrix} \pi_1 \star_D\ \lambda result : int. \\ (\texttt{setSBRS}\ result) \star_D\ \lambda\_. \\ \quad \texttt{return} \end{pmatrix} = \textbf{unit}_S \begin{pmatrix} (\texttt{setSBRS}\ 1) \star_D\ \lambda\_. \\ \quad \texttt{return} \end{pmatrix}
\end{aligned} \tag{2.3}
$$

$$
\begin{aligned}
&\mathcal{C}[\![\textbf{letclosed}\ i = 1\ \textbf{in}\ (\textbf{funcall}\ i)]\!] = \\
&\quad \mathtt{rdSD} \star_S \lambda\langle f,d\rangle. \\
&\quad \mathtt{newlabel} \star_S \lambda L_i. \\
&\quad \mathtt{newlabel} \star_S \lambda L_{ret}. \\
&\quad \mathbf{unit}_S \left(\begin{array}{l} (\mathtt{updateCode}[L_i \mapsto (\mathtt{setSBRS}\,1) \star_D \lambda\_.\mathtt{return}] \star_D \lambda\_. \\ \left(\begin{array}{l} \mathtt{callcc}\ \lambda\beta : \mathit{int} \to \mathsf{Dynam}(\mathtt{void}). \\ \quad (\mathtt{updateCode}\ L_{ret}\ (\mathtt{getSBRS} \star_D \beta)) \star_D \lambda\_. \\ \qquad \mathtt{call}\ L_i\ f\ [\,]\ L_{ret} \end{array} \right) \end{array} \right)
\end{aligned} \tag{2.4}
$$

Given the initial stack descriptor and label, $\langle 0, 0\rangle$ and 100, and initial (dynamic) continuation β_0, the right hand side of Equation 2.4 reduces to this dynamic semantic term:

$$
\begin{aligned}
&\mathtt{updateCode}[100 \mapsto (\mathtt{setSBRS}\,1) \star_D \lambda\_.\mathtt{return}] \star_D \lambda\_. \\
&\mathtt{updateCode}[101 \mapsto (\mathtt{getSBRS} \star_D \lambda v : \mathit{int}.(\mathtt{setSBRS}\,v) \star_D \beta_0)] \star_D \lambda\_. \\
&\quad \mathtt{call}\ 100\ 0\ [\,]\ 101
\end{aligned} \tag{2.5}
$$

This term may reasonably be pretty-printed as:

```
        call 100 0 [] 101;
100:    ldreg SBRS,1;
        return;
101:    ldreg SBRS,SBRS;
        halt;
```

One detail left out of the previous analysis is that the value of a function call returned in the SBRS register should be saved in a temporary location to avoid having it overwritten by another function call. To this end, savereg is introduced in Definition 11. Any *closed* function calls should be wrapped by a savereg, as is shown in Definition 12.

Definition 11 (savereg) savereg *is used to store the contents of the* SBRS *after the return from a function call:*

$$\mathsf{savereg} : \mathsf{Static}(\mathsf{Dynam}(int)) \to \mathsf{Static}(\mathsf{Dynam}(int))$$

$$\begin{array}{ll}
\mathsf{savereg}\ \mu = & \mathtt{rdSD}\ \star\ \lambda S. \\
& \mu\ \star_S\ \lambda\varphi : \mathsf{Dynam}(int). \\
& \mathbf{unit}_S(\varphi\ \star_D\ \lambda i : int.\mathtt{Thread}(i, S))
\end{array}$$

Definition 12 (Compilation Semantics of letclosed and letrec)

$$\begin{array}{l}
\mathcal{C}\llbracket \textbf{letclosed}\ f(x_1 : \varphi_1, \ldots, x_n : \varphi_n) = e\ \textbf{in}\ e' \rrbracket = \\
\qquad \mathtt{newlabel}\ \star_S\ \lambda L_f. \\
\qquad \mathtt{rdEnv}\ \star_S\ \lambda\rho. \\
\qquad \mathtt{rdSD}\ \star_S\ \lambda S. \\
\qquad (\mathtt{mksubr}_\varphi\ S\ \mathcal{C}\llbracket \lambda x_1. \ldots \lambda x_n.e \rrbracket)\ \star_S\ \lambda\pi_f. \\
\qquad\quad (\mathtt{inEnv}\ \rho[f \mapsto (\mathtt{mkcall}_\varphi\ S\ L_f)] \left(\begin{array}{l} \mathcal{C}\llbracket e' \rrbracket\ \star_S\ \lambda\pi_{e'}. \\ \\ \mathbf{unit}_S(\mathtt{updateCode}[L_f \mapsto \pi_f]\ \star_D\ \lambda\_.\pi_{e'}) \end{array} \right))
\end{array}$$

$$\begin{array}{l}
\mathcal{C}\llbracket \textbf{letrec}\ f(x_1 : \phi_1, \ldots, x_n : \phi_n) = e\ \textbf{in}\ e' \rrbracket = \\
\qquad \mathtt{newlabel}\ \star_S\ \lambda L_f. \\
\qquad \mathtt{rdEnv}\ \star_S\ \lambda\rho. \\
\qquad \mathtt{rdSD}\ \star_S\ \lambda S. \\
\qquad (\mathtt{mksubr}_\phi\ S\ (\mathtt{inEnv}\ \rho[f \mapsto (\mathtt{mkcall}_\phi\ S\ L_f)]\ \mathcal{C}\llbracket \lambda x_1. \ldots \lambda x_n.e \rrbracket))\ \star_S\ \lambda\pi_f. \\
\qquad\quad (\mathtt{inEnv}\ \rho[f \mapsto (\mathtt{mkcall}_\phi\ S\ L_f)] \left(\begin{array}{l} \mathcal{C}\llbracket e' \rrbracket\ \star_S\ \lambda\pi_{e'}. \\ \\ \mathbf{unit}_S(\mathtt{updateCode}[L_f \mapsto \pi_f]\ \star_D\ \lambda\_.\pi_{e'}) \end{array} \right))
\end{array}$$

$$\mathcal{C}\llbracket \textbf{funcall}\ f\, e_1 \ldots e_n \rrbracket = \mathsf{savereg}(\mathsf{apply}_\varphi\, \mathcal{C}\llbracket f \rrbracket\, \mathcal{C}\llbracket e_1 \rrbracket \ldots \mathcal{C}\llbracket e_n \rrbracket)$$

where $\varphi = \varphi_1 \rightarrow \ldots \rightarrow \varphi_n \rightarrow \mathbf{intexp}$,

2.5.1 Adding function arguments

The compilation method described previously can be easily extended to higher types. Consider the *FunExp* expression e:

$$\textbf{letclosed}\ \texttt{p} = (\lambda\texttt{x.b})\ \texttt{in}\ (\textbf{funcall}\ \texttt{p}\ (\texttt{1} + \texttt{2})) * (\textbf{funcall}\ \texttt{p}\ (\texttt{3} + \texttt{4}))$$

where $p : \mathbf{intexp} \rightarrow \mathbf{intexp}$. The compilation of e will involve compiling the arguments `(1+2)` and `(3+4)` into subroutines π_1 and π_2, respectively, using $\texttt{mksubr}_{\mathbf{intexp}}$, and then storing π_1 and π_2 in the code store at new labels L_1 and L_2, respectively. References to `x` within `b` should result in calls to L_1 and L_2 in the subroutine for $(\lambda\texttt{x.b})$. To be more precise, references to `x` within `b` should result in the appearance of appropriate `acall` instructions within the subroutine for $(\lambda\texttt{x.b})$.

To make a call to an argument label, an appropriate `acall` instruction must be generated. In the case of a reference to `x` within `b`, this is accomplished in a manner which is very similar to $\texttt{mkcall}_{\mathbf{intexp}}$, except that an instruction

$$\texttt{acall}\ i\ f\ \textit{arg-label-list return-label}$$

is used instead of a `call`. Since `x` is the first argument to `b`, i will be 1 in this case. Since `x` is of type **intexp**, *arg-label-list* will be `[]`, and for each **funcall** to $(\lambda\texttt{x.b})$, a new return label *return-label* will be generated. This is encapsulated by the helper function $\texttt{mkargcall}_{\mathbf{intexp}}$:

Definition 13 (`mkargcall` part I)

$$\mathtt{mkargcall}_{\textbf{intexp}} : SD \to \mathit{int} \to \mathsf{Static}(\mathsf{Dynam}(\mathit{int}))$$

$$\begin{aligned}\mathtt{mkargcall}_{\textbf{intexp}}\ \langle f, d\rangle\ j = \ & \mathtt{newlabel} \star_S \lambda L_{ret}. \\ & \mathtt{rdSD} \star_S \lambda S'. \\ & \textbf{unit}_S \left(\begin{array}{l} \mathtt{callcc}\ \lambda\beta : \mathit{int} \to \mathsf{Dynam}(\mathtt{void}). \\ \quad (\mathtt{updateCode}\ L_{ret}\ (\mathtt{getSBRS} \star_D \beta)) \star_D \lambda\_. \\ \quad\quad \mathtt{acall}\ j\ f\ [\,]\ L_{ret} \end{array} \right)\end{aligned}$$

($\mathtt{mkargcall}$ $\langle f, d\rangle$ j) produces an `acall` to the i-th argument label in the f-th frame. Recalling the above example $FunExp$ expression e, the appropriate argument call for `x` in `b` is produced by ($\mathtt{mkargcall}_{\textbf{intexp}}$ $\langle f+1, 0\rangle$ j) if e is compiled in a stack with shape $\langle f, d\rangle$. The stack descriptor $\langle f+1, 0\rangle$ signifies that arguments to (λ`x.b`) will be executed in a stack with one additional activation record.

To compile (λ`x.b`) into a subroutine, it is necessary to define $\mathtt{mksubr}_{\varphi}$ for $\varphi = \textbf{intexp} \to \textbf{intexp}$. ($\mathtt{mksubr}_{\textbf{intexp}\to\textbf{intexp}}$ S $\mathcal{C}[\![(\lambda\mathtt{x.b})]\!]$) will apply $\mathcal{C}[\![(\lambda\mathtt{x.b})]\!]$ to an appropriate `mkargcall` for `x`:

$$\begin{aligned} & \mathtt{mksubr}_{\textbf{intexp}\to\textbf{intexp}}\ S\ \mathcal{C}[\![(\lambda\mathtt{x.b})]\!] = \\ & \quad (\mathtt{inSD}\ S^{+}\ [\mathsf{apply}_{\textbf{intexp}\to\textbf{intexp}}\ \mathcal{C}[\![(\lambda\mathtt{x.b})]\!]\ (\mathtt{mkargcall}_{\textbf{intexp}}\ S^{+}\ 1)]) \star_S \lambda\pi_f : \mathsf{Dynam}(\mathit{int}). \\ & \quad \textbf{unit}_S \left(\begin{array}{l} \pi_f \star_D \lambda \mathit{result}. \\ (\mathtt{setSBRS}\ \mathit{result}) \star_D \lambda\_. \\ \quad \mathtt{return} \end{array} \right) \end{aligned}$$

π_f generates the return value corresponding to some function call **funcall** (λ`x.b`) arg, which is then stored in the return register `SBRS`, and then the subroutine returns. The definition of $\mathtt{mksubr}_{\varphi}$ for arbitrary φ is found in Definition 14.

Definition 14 (mksubr part II)

$$\texttt{mksubr}_\varphi : SD \rightarrow \mathsf{Static}(\mathsf{Dynam}(\varphi)) \rightarrow \mathsf{Static}(\mathsf{Dynam}(\texttt{void}))$$

$$\begin{array}{l} \texttt{mksubr}_\varphi\ S\ p = \\ \qquad (\texttt{inSD}\ S^+\ [\mathsf{apply}_\varphi\ p\ (\texttt{mkargcall}_{\varphi_1}\ S^+\ 1) \ldots (\texttt{mkargcall}_{\varphi_n}\ S^+\ n)])\ \star_S\ \lambda f. \\ \qquad \mathbf{unit}_S \left(\begin{array}{l} f\ \star_D\ \lambda result. \\ (\texttt{setSBRS}\ result)\ \star_D\ \lambda\_. \\ \quad \texttt{return} \end{array} \right) \end{array}$$

where $\varphi = \varphi_1 \rightarrow \ldots \rightarrow \varphi_n \rightarrow$ **intexp** *and* $\langle f, d\rangle^+ = \langle f+1, 0\rangle$.

Consider again the example *FunExp* expression e:

$$\textbf{letclosed}\ \texttt{p} = (\lambda\texttt{x.b})\ \texttt{in}\ (\textbf{funcall}\ \texttt{p}\ (\texttt{1}+\texttt{2})) * (\textbf{funcall}\ \texttt{p}\ (\texttt{3}+\texttt{4}))$$

To compile either function call in e, the last thing to do is extend `mkcall` to type **intexp**→**intexp**. The static part of $(\texttt{mkcall}_{\textbf{intexp}\rightarrow\textbf{intexp}}\ \langle f, d\rangle\ L)$ must:

- take the actual argument *arg* (i.e, `(1+2)` or `(3+4)`),

- get the current stack descriptor S,

- compile *arg* into a subroutine π using $(\texttt{mksubr}_{\textbf{intexp}}\ S^+\ arg)$,

- generate new labels L_{ret} and L_1 for the return code and π.

The dynamic part of $(\texttt{mkcall}_{\textbf{intexp}\rightarrow\textbf{intexp}}\ \langle f, d\rangle\ L)$ must:

- store π at L_1 in the code store,

- store the return code at L_{ret}, and

- execute $(\texttt{call}\ L\ f\ [L_1]\ L_{ret})$.

$(\texttt{mkcall}_{\textbf{intexp}\rightarrow\textbf{intexp}}\ \langle f, d\rangle\ L)$ may be neatly encapsulated as:

$$
\begin{array}{l}
\mathsf{Lambda}(1,\\
\quad \texttt{newlabel}\ \star_S\ \lambda L_{ret}.\\
\quad \texttt{rdSD}\ \star_S\ \lambda S'.\\
\quad \texttt{rdEnv}\ \star_S\ \lambda \rho.\\
\quad (\rho\, 1)\ \star_S\ \lambda a_1.\\
\quad (\texttt{mksubr}_{\textbf{intexp}}\ (S')^{+}\ a_1)\ \star_S\ \lambda \pi_1.\\
\quad \texttt{newlabel}\ \star_S\ \lambda L_1.\\
\qquad\qquad \textbf{unit}_S \left(
\begin{array}{l}
\texttt{callcc}\ \lambda\beta : \mathit{int} \rightarrow \mathsf{Dynam}(\texttt{void}).\\
\quad \texttt{updateCode}[L_1 \mapsto \pi_1]\ \star_D\ \lambda\_.\\
\quad (\texttt{updateCode}[L_{ret} \mapsto (\texttt{getSBRS}\ \star_D\ \beta)]\ \star_D\ \lambda\_.\\
\qquad \texttt{call}\ L\ f\ [L_1]\ L_{ret}
\end{array}
\right))
\end{array}
$$

The definition of $\texttt{mkcall}_\varphi$ for arbitrary φ is found in Definition 15. The (quite analogous) definition of $\texttt{mkargcall}_\varphi$ for arbitrary φ is found in Definition 16.

Definition 15 (`mkcall` part II)

$$\texttt{mkcall}_\varphi : SD \to \mathit{Label} \to \mathsf{Static}(\mathsf{Dynam}(\varphi))$$

$$
\begin{array}{l}
\texttt{mkcall}_\varphi\ \langle f, d \rangle\ L = \\
\quad \mathsf{Lambda}(1, \\
\qquad\quad \vdots \\
\qquad\quad \mathsf{Lambda}(n,\ \ \texttt{newlabel}\ \star_S\ \lambda L_{ret}. \\
\qquad\qquad\qquad\qquad \texttt{rdSD}\ \star_S\ \lambda S'. \\
\qquad\qquad\qquad\qquad \texttt{rdEnv}\ \star_S\ \lambda \rho. \\
\qquad\qquad\qquad\qquad (\rho\, 1)\ \star_S\ \lambda a_1. \\
\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad (\rho\, n)\ \star_S\ \lambda a_n. \\
\qquad\qquad\qquad\qquad (\texttt{mksubr}_{\varphi_1}\ (S')^+\ a_1)\ \star_S\ \lambda \pi_1. \\
\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad (\texttt{mksubr}_{\varphi_n}\ (S')^+\ a_n)\ \star_S\ \lambda \pi_n. \\
\qquad\qquad\qquad\qquad \texttt{newlabel}\ \star_S\ \lambda L_1. \\
\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad \texttt{newlabel}\ \star_S\ \lambda L_n. \\
\qquad\qquad\qquad\qquad\qquad \mathbf{unit}_S \left(\begin{array}{l}
\texttt{callcc}\ \lambda \beta : \mathit{int} \to \mathsf{Dynam}(\texttt{void}). \\
\quad \texttt{updateCode}[L_1 \mapsto \pi_1]\ \star_D\ \lambda\_. \\
\qquad\qquad\qquad \vdots \\
\quad \texttt{updateCode}[L_n \mapsto \pi_n]\ \star_D\ \lambda\_. \\
\quad (\texttt{updateCode}[L_{ret} \mapsto (\texttt{getSBRS}\ \star_D\ \beta)]\ \star_D\ \lambda\_. \\
\qquad\quad \texttt{call}\ L\ f\ [L_1, \ldots, L_n]\ L_{ret}
\end{array} \right) \ldots)
\end{array}
$$

where $\varphi = \varphi_1 \to \ldots \to \varphi_n \to \textbf{intexp}$.

Definition 16 (mkargcall part II)

$$\mathtt{mkargcall}_{\varphi} : SD \to int \to \mathsf{Static}(\mathsf{Dynam}(\varphi))$$

$$\begin{array}{l}
\mathtt{mkargcall}_{\varphi}\ \langle f, d\rangle\ j = \\
\quad \mathsf{Lambda}(1, \\
\qquad\quad \vdots \\
\qquad\quad \mathsf{Lambda}(n,\ \ \mathtt{newlabel}\ \star_S\ \lambda L_{ret}. \\
\qquad\qquad\qquad\quad \mathtt{rdSD}\ \star_S\ \lambda S'. \\
\qquad\qquad\qquad\quad \mathtt{rdEnv}\ \star_S\ \lambda\rho. \\
\qquad\qquad\qquad\quad (\rho\, 1)\ \star_S\ \lambda a_1. \\
\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\quad (\rho\, n)\ \star_S\ \lambda a_n. \\
\qquad\qquad\qquad\quad (\mathtt{mksubr}_{\varphi_1}\ (S')^+\ a_1)\ \star_S\ \lambda\pi_1. \\
\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\quad (\mathtt{mksubr}_{\varphi_n}\ (S')^+\ a_n)\ \star_S\ \lambda\pi_n. \\
\qquad\qquad\qquad\quad \mathtt{newlabel}\ \star_S\ \lambda L_1. \\
\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\quad \mathtt{newlabel}\ \star_S\ \lambda L_n. \\
\qquad\qquad\qquad\qquad\quad \mathbf{unit}_S \left(\begin{array}{l}
\mathtt{callcc}\ \lambda\beta : int \to \mathsf{Dynam}(\mathtt{void}). \\
\ \mathtt{updateCode}[L_1 \mapsto \pi_1]\ \star_D\ \lambda\_. \\
\qquad\qquad\quad \vdots \\
\ \mathtt{updateCode}[L_n \mapsto \pi_n]\ \star_D\ \lambda\_. \\
\ (\mathtt{updateCode}[L_{ret} \mapsto (\mathtt{getSBRS}\ \star_D\ \beta)]\ \star_D\ \lambda\_. \\
\qquad \mathtt{acall}\ j\ f\ [L_1, \ldots, L_n]\ L_{ret}
\end{array} \right)) \ldots)
\end{array}$$

where $\varphi = \varphi_1 \to \ldots \to \varphi_n \to \mathbf{intexp}$.

Compiling: `(letopen inc(x) = x+1 in (funcall inc --99))`

```
                  <0,2> := 99;
                  <0,1> := -<0,2>;
                  <0,0> := -<0,1>;
                  <0,1> := 1;

       Answer is: <0,0>+<0,1>
```

Compiling: `(letclosed inc(x) = x+1 in (funcall inc --99))`

```
0:  call 2 0 [4] 3

1:  <1,0> := SBRS
    <1,1> := 1
    ldreg SBRS,SBRS+1
    return

2:  acall 1 1 [] 1 /* call to 4 */

3:  halt           /* Answer in SBRS */

4:  <1,1> := 99
    <1,0> := -<1,1>
    ldreg SBRS,-<1,0>
    return
```

Figure 2.15: Compiling the open and closed versions of the same procedure

2.5.2 Sample Compilations

Figures 2.15 and 2.16 present examples compilations of *FunExp* procedures. Figure 2.15 demonstrates the difference between the compilation of open and closed procedures. The open version of the procedure `inc(x) = x+1` simply inlines the body of the procedure at the call site. In other words, the resulting code is identical to the code that would be produced by $(- - 99 + 1)$. Figure 2.16 shows an example of the compilation of a recursive program.

2.6 Conclusions

This chapter introduced a new form of denotational specification based on metacomputations. Metacomputations are a simple and elegant structure for representing staged computation within

Compiling:

```
letrec
        fact(n):(int->int) = (n=0 ==> 1,(n*(funcall fact (n-1))))
in
        (funcall fact 2)


        call 100  [110]  <0,0> <0,0> 109  /* initial funcall */

        109:    <0,0> := SBRS             /* <0,0> has 2! */
                halt

        110:    ldreg SBRS,#2             /* subr. for original arg. */
                return

        100:    acall 1 1 [] 104          /* gets n for n=0 test */

        104:    brzero SBRS 101 102       /* n=0 test */

        101:    <1,0> := #1               /* return 1 if n=0 */
                jump 103

        102:    acall 1 1 [] 105          /* get n for else clause */

        105:    <1,1> := SBRS             /* store n */
                call 100 0 [108] 106      /* rec. call to fact */

        108:    acall 1 1 [] 107          /* get n for (n-1) arg */

        107:    <2,0> := SBRS             /* subr. calculates (n-1) */
                <2,1> := #1
                ldreg SBRS,<2,0>-<2,1>
                return

        106:    <1,3> := SBRS             /* n*(fact (n-1)) */
                <1,2> := <1,3>
                <1,0> := <1,1>*<1,2>
                jump 103

        103:    ldreg SBRS,<1,0>
                return
```

Figure 2.16: Compiling a **letrec**

the semantics of a programming language. Metacomputation-style specifications use two monads to factor the static and dynamic parts of the specification, thereby staging it and achieving strong binding-time separation iin the sense of [29]. Because metacomputations are defined in terms of monads, they can be constructed modularly and extensibly using monad transformers.

Reusable compiler building blocks (in the sense of Section 1.1) may be implemented as metacomputation-style language specifications. In fact, this chapter presented the first implementation of RCBBs (the second implementation—as monadic code generators—appears in the next chapter). Reusable compiler building blocks were presented for a wide variety of language features: expressions, booleans, imperative, control-flow, block structure, recursive bindings, and higher-order integer-valued functions.

An original inspiration for this study was Reynold's derivation of a compiler from the functor category semantics for Algol[43]. Both the run-time system for subroutines and the target language for the compiler blocks presented here are based on that work. However, one difference is that the target language presented here includes jumps and labels.

Chapter 3

Reusable Compiler Building Blocks as Monadic Code Generators

This chapter concerns the implementation of reusable compiler building blocks as *monadic code generators.* A monadic code generator (MCG) is simply a monadic source language specification which is target language-valued. For each source language feature, a function $\mathsf{compile} : \mathsf{Source} \to \mathsf{M}(\mathsf{TargetLang})$ is defined which specifies how each feature is translated into the target language. Not surprisingly, the definitions of compile closely resemble what are called *translation schemas* or *semantic actions* in traditional work on compilation[1, 35, 2].

The RCBB implementations presented in this chapter differ from those of Chapter 2 in that monadic code generators produce target language code directly without relying on the intermediate process of partial evaluation. Avoiding partial evaluation is an advantage of the straightforward MCG approach to code generation, because the use of a partial evaluator can be complicated in practice. Furthermore, certain optimizations are easier to implement within MCGs than within metacomputation-based definitions because an MCG can inspect target code values (while the dynamic computations produced by metacomputation-based specifications, being functions, may not be inspected). An example of this kind will be seen in the optimizing MCG for expressions presented in Section 3.3. From the point of view of compiler correctness, however, MCGs are one step further removed from the standard semantics of their source language than are the metacomputation-based RCBBs of Chapter 2. This introduces further requirements in the correctness proof of MCG-based modular compilers as will be seen in Chapter 5 where the MCG compiler block implementations are formally related to the metacomputation-based implementations.

MachLang ::=
    NOP|
    S := Rhs |
    MachLang ; MachLang |
    JUMP L |
    ALLOC S |
    DEALLOC S |
    ENDLABEL L MachLang |
    SEGM (L,MachLang) |
    BRLEQ Rhs Rhs |
    MachLang $\diamond$ MachLang $\times$ MachLang |
    call *L f arg-label-list return-label* |
    acall *i f arg-label-list return-label* |
    return

Rhs ::= SRhs | -SRhs | SRhs+SRhs
SRhs ::= $\langle f, d \rangle \in SD$ | $\#c$ for integer c

IntProducer = MachLang $\times$ Rhs $\times$ $\mathcal{S}et(SD)$

TargetLang = MachLang + IntProducer

Figure 3.1: BNF for the Target Language

3.1 Target Language

The target language TargetLang used in this work is presented in Figure 3.1. It is composed of two sublanguages: the machine language MachLang and the integer producer language IntProducer. MachLang is a *three-address code* as one might find in [1, 35, 2]. That is, it contains assignments to locations, sequencing, and jumps. Source language commands are compiled to MachLang programs. Target language right-hand sides Rhs include at most one arithmetic operation as is typical of right-hand sides of intermediate and machine languages[1, 35, 2].

MachLang also contains several commands which are not typically found in three-address code:

- two structured label operators ENDLABEL and SEGM, which allow new labels and code segments to be introduced,
- storage allocation and deallocation operators ALLOC and DEALLOC,
- code-level apply with $\diamond$. This is used to provide labels to a branch and will be discussed in

Section 3.7 (control-flow MCG).

The command (`ENDLABEL` L π) introduces a new label at the end of the MachLang program π. An alternative would be to introduce a `Label(L):` command into MachLang instead, and then "(`ENDLABEL` L π)" would be equivalent to "π `; Label(L)`". The command `SEGM(L,`π`)` defines a new code segment π at label `L`. The commands `ALLOC` and `DEALLOC` explicitly allocate and deallocate stack locations. Although these commands are not typical intermediate code, we shall see in Chapters 4 and 5 that it is convenient for the purpose of relating the source and target language semantics to have these commands.

Source language phrases of type **intexp** are compiled to programs in the integer producer language IntProducer = MachLang × Rhs × $\mathcal{S}et(SD)$. An IntProducer captures the notion of integer-producing code by pairing a machine language right-hand side to a machine language sequence. Machine language sequences do not produce *values* as they are commands, so how does one model source language integer expressions as sequences of machine language instructions? The key is to model an integer expression e as a sequence of machine instructions π_e and a machine language right-hand side rhs_e with the following intended meaning. First π_e is executed, and then rhs_e is evaluated, thereby producing an integer value. π_e may use temporary locations $tmps_e$ to calculate intermediate values of e, and then the evaluation of rhs_e uses these locations prepared by π_e to produce the value for e. When we conjoin the temporaries $tmps_e$ to π_e and rhs_e, we get the integer producer $\langle \pi_e, rhs_e, tmps_e \rangle$.

The simplest example of an IntProducer is the code produced for an integer literal c, which is the triple $\langle \texttt{NOP}, \#\texttt{c}, \{\} \rangle$. The code necessary for calculating an integer constant is a `NOP` ("no operation"), because there are no intermediate values to be stored. The right-hand side is simply the target language constant $\#$`c`, and since no temporary locations were used, the empty set $\{\}$ is the third component.

An IntProducer produced in the compilation of the integer expression "`-777`" is:

$$\langle \texttt{ALLOC}\,\langle 0,0\rangle \ ;\ \langle 0,0\rangle{:=}\#777, -\langle 0,0\rangle, \{\langle 0,0\rangle\}\rangle$$

Operationally, this triple means the following:

$\mathsf{compile}(e : \mathbf{intexp}) : \mathsf{M}(\mathsf{IntProducer})$
$\mathsf{M} = \mathcal{T}_{\mathsf{Env}}\ SD\ \mathsf{Id}$

$$\mathsf{compile}(i) = \mathbf{unit}\langle \mathtt{NOP}, \#i, \{\,\}\rangle$$

$\mathsf{compile}(-e) =$

$$\mathtt{rdSD} \star \lambda S.\mathtt{inSD}\ (S+1) \left(\begin{array}{l} (\mathsf{compile}\, e) \star \lambda\langle \pi_e, rhs_e, tmps_e\rangle . \\ \qquad \mathbf{unit}\langle \pi_e ; \mathtt{ALLOC}\, S ; S{:=}rhs_e ; (\mathsf{pop}\ tmps_e), -S, \{S\}\rangle \end{array} \right)$$

$\mathsf{compile}(e_1 + e_2) =$

$$\begin{array}{l} \mathtt{rdSD} \star \lambda\langle f, d\rangle . \\ \quad \mathtt{inSD}\ (\langle f, d+2\rangle) \left(\begin{array}{l} (\mathsf{compile}\ e_1) \star \lambda\langle \pi_1, rhs_1, tmps_1\rangle . \\ (\mathsf{compile}\ e_2) \star \lambda\langle \pi_2, rhs_2, tmps_2\rangle . \\ \quad \mathbf{unit}\langle \pi_1 ; \mathtt{ALLOC}\,\langle f, d\rangle ; \langle f, d\rangle{:=}rhs_1 ; (\mathsf{pop}\ tmps_1) ; \\ \qquad\quad \pi_2 ; \mathtt{ALLOC}\,\langle f, d+1\rangle ; \langle f, d+1\rangle{:=}rhs_2 ; (\mathsf{pop}\ tmps_2), \\ \qquad\quad \langle f, d\rangle + \langle f, d+1\rangle, \\ \qquad\quad \{\langle f, d\rangle, \langle f, d+1\rangle\}\rangle \end{array} \right) \end{array}$$

where:

$$\begin{array}{l} (\mathsf{pop}\ \{\}) = \mathtt{NOP} \\ (\mathsf{pop}\ \{S_1, \ldots, S_n\}) = \mathtt{DEALLOC}\, S_1\ ;\ \ldots\ ;\ \mathtt{DEALLOC}\, S_n \end{array}$$

Figure 3.2: Monadic Code Generator for *Exp*

1. Execute ALLOC $\langle 0, 0\rangle$; $\langle 0, 0\rangle{:=}\#777$, which first allocates a temporary location $\langle 0, 0\rangle$, and then stores the intermediate value 777 in $\langle 0, 0\rangle$. This code prepares the store for the evaluation of the right-hand side.

2. Evaluate the right-hand side, $-\langle 0, 0\rangle$, which returns the value of the expression.

3. The set of temporary locations $\{\langle 0, 0\rangle\}$ keeps track of the locations which were used in the evaluation of "-777", which may now be deallocated.

Generally, an IntProducer program $\langle \pi_e, rhs_e, tmps_e\rangle$ means operationally:

1. execute π,

2. evaluate the right-hand side rhs_e,

3. deallocate $tmps_e$

```
fun assign S (IP ((SEQ pi),rhs,Slist)) result =
           IP(SEQ(pi@[ALLOC S,ASSIGN(S,rhs),DEALLOC Slist]), result,[S]);

fun AddAssign (IP ((SEQ pi1),rhs1,tmp1)) (IP ((SEQ pi2),rhs2,tmp2)) S1 S2 =
           IP(SEQ(pi1 @ [ALLOC S1,ASSIGN(S1,rhs1),DEALLOC tmp1] @
                  pi2 @ [ALLOC S2,ASSIGN (S2,rhs2),DEALLOC tmp2]),
                        (ADD (stackLoc(S1),stackLoc(S2))),[S1,S2]);

fun compile (e:Expr) =
case e of
(IntLit i) =>   unit (code (IP((SEQ []),(simpRHS (Lit i)),[])))
   | (* N.b., assign and AddAssign do the deallocation *)
(Negate e') =>
         rdSD bind (fn S as (f,d) =>
         (inSD (f,d+1) ((compile e') bind (fn cv as (code c) =>
                   (unit (code (assign S c (NEG (stackLoc S))))))))
   |
(Add (e1,e2)) =>
         rdSD bind (fn S as (f,d) =>
         (inSD (f,d+2)
             ((compile e1) bind (fn cv1 as (code c1) =>
              (compile e2) bind (fn cv2 as (code c2) =>
                  (unit (code (AddAssign c1 c2 (f,d) (f,d+1)))))))));
```

Figure 3.3: ML Monadic Code Generator for *Exp*

3.2 Monadic Code Generator for Expressions

Figure 3.2 presents the MCG for integer expressions and Figure 3.3 shows its implementation in ML. Here, the compiler function compile : M(IntProducer), will generate an IntProducer. Observe, also, that display addresses are used here for the "free address" type, although for this MCG, $Addr = int$ would serve equally well. We add display addresses with the environment monad transformer ($\mathcal{T}_{\mathsf{Env}}\, SD$).

The IntProducer corresponding to an integer literal i requires neither code nor temporary locations, so i is mapped to the triple $\langle \mathtt{NOP}, \mathtt{\#}i, \{\} \rangle$. To compile a negation $-e$, first the top free address S is calculated and subexpression e is compiled in a larger stack $S+1$, producing $\langle \pi_e, rhs_e, tmps_e \rangle$. The MachLang component for $-e$ is:

$$\pi_e \ ; \mathtt{ALLOC}(S) \ ; S{:=}rhs_e \ ; (\mathsf{pop}\, tmps_e)$$

This code executes π_e (thus preparing rhs_e for evaluation), allocates S and stores the value of

rhs_e in S, and then deallocates the temporaries $tmps_e$ used in π_e using pop[1]. After executing that code, the target language right-hand side for negation, $-S$, can be evaluated to produce the value for $-e$. The temporaries used in that code are only $\{S\}$. $\mathsf{compile}(e_1 + e_2)$ is defined similarly to $\mathsf{compile}(-e)$.

3.3 Optimizing Monadic Code Generator for Expressions

While the MCG for expressions presented in the previous section has the advantage of simplicity, it does not always produce space-efficient code because *every* intermediate value of a term is stored in a temporary location. For example, the code produced by the MCG in the previous section for the integer expression "-777" is (if the current stack descriptor is $\langle 0, 0\rangle$ at the time of compilation):

$$\langle \texttt{ALLOC}\,\langle 0,0\rangle \texttt{ ; } \langle 0,0\rangle{:=}\#777, -\langle 0,0\rangle, \{\langle 0,0\rangle\}\rangle$$

Clearly, the use of the location $\langle 0, 0\rangle$ is unnecessary, as a more space-efficient $\mathsf{IntProducer}$ for "-777" is:

$$\langle \texttt{NOP}, \# - 777, \{\}\rangle$$

Figure 3.4 presents an alternative MCG specification for expressions which is considerably more thrifty with regards to storage. By applying the CPS monad transformer, and thereby having access to continuations, we can choose whether or not to store an intermediate value. The specification presented here is reminiscent of Reynolds *usetmp* operator in [43].

A target language right-hand sides (e.g., $\#777$, $-\langle f, d\rangle$, $\langle f_1, d_1\rangle{+}\langle f_2, d_2\rangle$, etc.) may have at most one operation. Define a *simple* right-hand side to be a target language right-hand side with no operations in it—i.e., a literal $\#n$ or a storage reference $\langle f, d\rangle$. If $\langle \pi_e, rhs_e, tmps_e\rangle$ is the $\mathsf{IntProducer}$ generated for expression e where rhs_e is simple, then the code for $-e$ can simply insert a negation

[1] $(\mathsf{pop}\, tmps_e)$ is a sequence of $\texttt{DEALLOC}$ instructions, and although its definition is under-specified in Figure 3.2 because $tmps_e$ is a *set* (i.e., $\mathsf{pop}\,\{\langle 0,1\rangle, \langle 2,5\rangle\}$ could be "$\texttt{DEALLOC}\,\langle 0,1\rangle \texttt{ ; } \texttt{DEALLOC}\,\langle 2,5\rangle$" or "$\texttt{DEALLOC}\,\langle 2,5\rangle \texttt{ ; } \texttt{DEALLOC}\,\langle 0,1\rangle$"), it should be clear that pop could be defined as a function given some total ordering on SD (like the lexicographical order, for example). Because the order of deallocations is irrelevant, such details are unnecessary.

```
compile($e$ : $\mathbf{intexp}$) : M(IntProducer)
M = $\mathcal{T}_{\mathsf{Env}}\ SD\ \mathcal{T}_{\mathsf{CPS}}$ void Id

compile($i$) = $\mathbf{unit}\langle$NOP, #$i$, $\{\ \}\rangle$

compile($-e$) =
 callcc $\lambda\beta$.
      (compile $e$) $\star$ $\lambda\langle\pi_e, rhs_e, tmps_e\rangle$.
       case (simple $rhs_e$) of
      [ $true\ \Rightarrow \beta\langle\pi_e, -rhs_e, tmps_e\rangle$ |
        $false \Rightarrow$ rdSD $\star$ $\lambda S$.
                  inSD($S+1$)
                    $\beta\,\langle\pi_e$ ; ALLOC $S$ ; $S$:=$rhs_e$ ; (pop $tmps_e$), $-S$, $\{S\}\rangle$ ]

compile($e_1 + e_2$) =
 callcc $\lambda\beta$.
   (compile $e_1$) $\star$ $\lambda\langle\pi_1, rhs_1, tmps_1\rangle$.
   case (simple $rhs_1$) of
    $true\ \Rightarrow$
       (compile $e_2$) $\star$ $\lambda\langle\pi_2, rhs_2, tmps_2\rangle$.
        case (simple $rhs_2$) of
       [ $true\ \Rightarrow \beta\langle\pi_1;\pi_2, rhs_1{+}rhs_2, tmps_1 \cup tmps_2\rangle$ |
         $false \Rightarrow$ rdSD $\star$ $\lambda S$.
                  inSD($S+2$)
                    $\beta\,\langle\pi_1;\pi_2;$ALLOC $S+1$ ; $S+1$:=$rhs_2$ ; (pop $tmps_2$), $rhs_1{+}S+1$, $tmps_1 \cup \{S+1\}\rangle$ ]
    $false \Rightarrow$
        rdSD $\star$ $\lambda S$.
        inSD($S+2$)
           (compile $e_2$) $\star$ $\lambda\langle\pi_2, rhs_2, tmps_2\rangle$.
            case (simple $rhs_2$) of
           [ $true\ \Rightarrow \beta\,\langle\pi_1;$ALLOC $S+1$ ; $S+1$:=$rhs_1$ ; (pop $tmps_1$);$\pi_2$, $(S+1) + rhs_2$, $\{S+1\} \cup tmps_2\rangle$
             $false \Rightarrow$
              rdSD $\star$ $\lambda S'$.
              inSD($S'+1$)
                       ( $\pi_1$ ;
                         ALLOC $S$ ;
                         $S$:=$rhs_1$ ;
                  $\beta\,\langle$   (pop $tmps_1$) ;                , $S+S'$, $\{S, S'\}\rangle$
                         $\pi_2$ ;
                         ALLOC $S'$ ;
                         $S'$:=$rhs_2$ ;
                         (pop $tmps_2$) )                                              ]
```

Figure 3.4: Optimizing Monadic Code Generator for *Exp*

in front of rhs_e:

$$\langle \pi_e, -rhs_e, tmps_e \rangle$$

If rhs_e is not simple, then a temporary location must be allocated for rhs_e just as the MCG in Section 3.2 did.

The use of continuations with `callcc` allows control over whether a temporary location is allocated or not. Let $\langle \pi_e, rhs_e, tmps_e \rangle$ be the $\mathsf{IntProducer}$ generated for the subexpression e of the negation $-e$. Then, if rhs_e is simple, the $\mathsf{IntProducer}$ $\langle \pi_e, -rhs_e, tmps_e \rangle$ is passed along to the current continuation β. But if rhs_e is not simple, then a temporary location S is allocated (just as the MCG in the previous section did). The rest of the code is informed of this allocation using $\texttt{inSD}\ (S+1)$:

$$\texttt{inSD}\ (S+1)\ (\beta\ \langle \pi_e\,;\texttt{ALLOC}\,S\,;S{:=}rhs_e\,;(\mathsf{pop}\ tmps_e), -S, \{S\}\rangle)$$

The definition for addition proceeds along similar lines. Figure 3.5 presents example compilations using the ML implementations of the optimizing and non-optimizing MCGs for expressions.

3.4 Monadic Code Generator for Imperative Features

Figure 3.6 presents the MCG for the imperative language block. To compile an assignment "$x := e$", retrieve the address S_x of x in the current environment and compile the right-hand side e, producing $\langle \pi_e, rhs_e, tmps_e \rangle$. The code emitted for the assignment will first execute π_e to prepare rhs_e for evaluation, then update the location S_x by the value of rhs_e, and finally, deallocate any temporaries used by π_e. To compile the command sequence $c_1;c_2$, compile c_1 and c_2, producing $\mathsf{TargetLang}$ commands π_1 and π_2, respectively. The code which is emitted for $c_1;c_2$ simply concatenates the code for c_1 and c_2: π_1 ; π_2. Figure 3.7 presents an ML implementation of the MCG for imperative features.

```
Compiling: -(777+----99)

ALLOC <0,0>
<0,0> := -#99
<0,0> := -<0,0>
<0,0> := -<0,0>
ALLOC <0,1>
<0,1> := -<0,0>
DEALLOC <0,0>
ALLOC <0,0>
<0,0> := #777+<0,1>
DEALLOC <0,1>
rhs=-<0,0>, tmps=[<0,0>]
```

```
Compiling: -(777+----99)

ALLOC <0,1>
<0,1> := #777
ALLOC <0,6>
<0,6> := #99
ALLOC <0,5>
<0,5> := -<0,6>
DEALLOC <0,6>
ALLOC <0,4>
<0,4> := -<0,5>
DEALLOC <0,5>
ALLOC <0,3>
<0,3> := -<0,4>
DEALLOC <0,4>
ALLOC <0,2>
<0,2> := -<0,3>
DEALLOC <0,3>
ALLOC <0,0>
<0,0> := <0,1>+<0,2>
DEALLOC <0,1>
DEALLOC <0,2>
rhs=-<0,0>, tmps=[<0,0>]
```

Figure 3.5: Optimizing vs. Non-optimizing MCG Output from ML Implementation

```
compile($e$ : $\mathbf{comm}$) : M(TargetLang)
M = $\mathcal{T}_{\mathsf{Env}}\ Env$ Id

compile($x := e$) : M(TargetLang) =
      rdEnv $\star$ $\lambda\rho$.
      $(\rho\, x)$ $\star$ $\lambda S_x$.
      compile($e$) $\star$ $\lambda\langle \pi_e, rhs_e, tmps_e \rangle$.
         $\mathbf{unit}(\pi_e\ ;\ S_x{:=}rhs_e\ ;\ (\mathsf{pop}\ tmps_e))$

compile($c_1;c_2$) : M(TargetLang) =
      compile($c_1$) $\star$ $\lambda\pi_1$.
      compile($c_2$) $\star$ $\lambda\pi_2$.
         $\mathbf{unit}(\pi_1\ ;\ \pi_2)$
```

Figure 3.6: Monadic Code Generator for *Imp*

```
fun compile (e:Expr) =
     case e of
     Assign(x,e) =>
           rdEnv bind (fn Rho as (env rho) =>
           (rho x) bind (fn A as (address S) =>
           (compile e) bind (fn R as (code (IP ((SEQ pi_e),rhs,tmps))) =>
                 unit (code(OC(SEQ (pi_e @ [(ASSIGN (S, rhs)),pop tmps])))))))
    |
     Seq(e1,e2) =>
           (compile e1) bind (fn c1 as (code(OC (SEQ pi1))) =>
           (compile e2) bind (fn c2 as (code(OC (SEQ pi2))) =>
                 unit (code(OC (SEQ(pi1 @ pi2))))))
```

Figure 3.7: ML Monadic Code Generator for *Imp*

```
compile($e_1 \leq e_2$) : M($TargetLang$) =
   rdSD $\star$ $\lambda S$.
   inSD $(S+2)$
     compile($e_1$) $\star$ $\lambda\langle \pi_1, rhs_1, tmps_1 \rangle$.
     compile($e_2$) $\star$ $\lambda\langle \pi_2, rhs_2, tmps_2 \rangle$.
       unit($\pi_1$ ; ALLOC($S$) ; $S$:=$rhs_1$ ; $\pi_2$ ; ALLOC($S+1$) ; ($S$+1):=$rhs_2$ ; (pop $tmps_1$) ; (pop $tmps_2$) ; BRLEQ $S$ ($S$+1))
```

Figure 3.8: Monadic Code Generator for *Bool*

```
fun compile (e:Expr) =
case e of

LTEQ(e1,e2) =>
   rdSD bind (fn S =>
   inSD (inc (inc S))
     ((compile e1) bind (fn ic as (code (IP ((SEQ pi1),rhs1,tmps1))) =>
      (compile e2) bind (fn ic as (code (IP ((SEQ pi2),rhs2,tmps2))) =>
        unit (code (OC
          (SEQ (pi1 @ [ALLOC(S), (ASSIGN (S,rhs1))] @
                pi2 @ [ALLOC(inc S), (ASSIGN ((inc S), rhs2)),
                DEALLOC(tmps1@tmps2),
                (BRLEQ ((simpRHS(stackLoc S)),(simpRHS(stackLoc (inc S)))))]))))))))
```

Figure 3.9: ML Monadic Code Generator for *Bool*

3.5 Monadic Code Generator for Boolean Expressions

Figure 3.8 contains the definition of the MCG for the boolean sublanguage. The compilation of the source language predicate $e_1 \leq e_2$ is similar to addition, although, instead of generating an IntProducer, a code sequence ending in a branch is produced. Observe that the target labels are not provided yet to the branch—that is, one might expect the branch produced to look like ($\mathtt{BRLEQ}\ \mathtt{r}_1\ \mathtt{r}_2\ \mathtt{L}_\mathtt{T}\ \mathtt{L}_\mathtt{F}$), where control is given to $\mathtt{L}_\mathtt{T}$ if $\mathtt{r}_1 \leq \mathtt{r}_2$ and to $\mathtt{L}_\mathtt{F}$ otherwise. These labels are provided when the boolean code is used in the control-flow block. See Section 3.7 for further discussion. Figure 3.9 presents an ML implementation of the MCG for boolean features.

3.6 Monadic Code Generator for Block Structure

Figure 3.10 presents the MCG for the block structure language. To compile a new integer variable declaration, ($\mathtt{new}\, x\, \mathtt{in}\, c$), first the current free location S is calculated and the body c is compiled for the larger stack $S+1$, producing the TargetLang command π_c. The code which is emitted first

$$\begin{array}{rcl}
\mathsf{compile}(\mathtt{new}\,x\,\mathtt{in}\,c) & = & \mathtt{rdSD} \star \lambda S.\mathtt{rdEnv} \star \lambda\rho. \\
 & & \quad [\mathtt{inSD}\,(S{+}1)\,(\mathtt{inEnv}\,\rho[x \mapsto (\mathbf{unit}\,S)]\,(\mathsf{compile}\,c))] \star \lambda\pi_c. \\
 & & \quad\quad \mathbf{unit}(\mathtt{ALLOC}(S)\,;\,\pi_c\,;\,\mathtt{DEALLOC}(S))
\end{array}$$

$$\mathsf{compile}(x) \;=\; \mathtt{rdEnv} \star \lambda\rho.(\rho\,x) \star \lambda S.\mathbf{unit}\langle\mathtt{NOP}, S, \{\}\rangle$$

Figure 3.10: Monadic Code Generator for Block Structure

```
fun compile (e:Expr) =
case e of
        NewIntVar (x,c) =>
          rdSD bind (fn S =>
          rdEnv bind (fn rho =>
          (inSD (inc S)
          (inEnv (xEnv (x,(unit (address S))) rho) (compile c)))
                bind (fn D as (code(OC (SEQ pi_c))) =>
                   unit((code(OC (SEQ [ALLOC S] @ pi_c @ [DEALLOC S])))
|
        (Exp x) =>
          rdEnv bind (fn env_rho as (env rho) =>
          (rho x) bind (fn addr as (address Sx) =>
               unit (code(IP (SEQ [],(simpRHS(stackLoc Sx)),[])))))
```

Figure 3.11: ML Monadic Code Generator for Block Structure

allocates the location S, executes π_c, and then deallocates S once outside the scope of the `new`. When the program variable x appears as part of an expression, compile(x) generates an IntProducer which looks up the value of the location S_x associated with x: $\langle\mathtt{NOP}, S_x, \{\}\rangle$. Figure 3.11 presents an ML implementation of the MCG for block structure.

```
compile($\mathbf{if}\ b\ \mathbf{then}\ c$) =
   newlabel $\star$ $\lambda L_{\text{exit}}$.
   newlabel $\star$ $\lambda L_c$.
   compile($b$) $\star$ $\lambda \pi_b$.
   compile($c$) $\star$ $\lambda \pi_c$.
            $\mathbf{unit}$(ENDLABEL $L_{\text{exit}}$ (SEGM[$L_c, \pi_c$ ; JUMP $L_{\text{exit}}$] ; ($\pi_b \diamond \langle$JUMP $L_c$, JUMP $L_{\text{exit}}\rangle$))))
```

```
compile($\mathbf{while}\ b\ \mathbf{do}\ \mathbf{c}$) =
          newlabel $\star$ $\lambda L_{exit}$.
          newlabel $\star$ $\lambda L_c$.
          newlabel $\star$ $\lambda L_{test}$.
          (compile $b$) $\star$ $\lambda \pi_b$.
          (compile $c$) $\star$ $\lambda \pi_c$.
           $\mathbf{unit}$(ENDLABEL $L_{exit}$ (SEGM[$L_c, \pi_c$ ; JUMP $L_{test}$] ; SEGM[$L_{test}, \pi_b \diamond \langle$JUMP$L_c$, JUMP$L_{exit}\rangle$] ; JUMP$L_{test}$))
```

Figure 3.12: Monadic Code Generator for Control Flow

```
fun compile (e:Expr) =
case e of
                IfThen(b,c) =>
                        newlabel bind (fn Lexit =>
                        newlabel bind (fn Lc =>
                        (compile b) bind (fn B as (code (OC (SEQ pi_b))) =>
                        (compile c) bind (fn MP as (code (OC (SEQ pi_c))) =>
                           let val b_code = (diamond ((SEQ pi_b), (JUMP Lc,JUMP Lexit)))
                           in
                                   unit (code(OC(SEQ
                                            ([ENDLABEL (Lexit,[b_code] @
                                                        [(SEGM (Lc, pi_c @ [JUMP Lexit]))])])
                                         )))
                           end
                           ))))
                           end))))
|
                WhileDo(b,c) =>
                       newlabel bind (fn Lexit =>
                       newlabel bind (fn Lc =>
                       newlabel bind (fn Ltest =>
                       (compile b) bind (fn B as (code (OC (SEQ pi_b))) =>
                       (compile c) bind (fn MP as (code (OC (SEQ pi_c))) =>
                          let val b_code = [(diamond ((SEQ pi_b, (JUMP Lc,JUMP Lexit))))]
                          in
                               unit (code (OC (SEQ [ENDLABEL(Lexit,
                                                        [SEGM(Lc,(pi_c@[JUMP Ltest])),
                                                         SEGM(Ltest,b_code),
                                                         (JUMP Ltest)])])))
                          end
                                     )))));
```

Figure 3.13: ML Monadic Code Generator for Control-flow

3.7 Monadic Code Generator for Control Flow

Figure 3.12 presents the MCG for the control-flow block. A conditional "**if** b **then** c" would typically be compiled into machine code of the form:

$\langle$generate new labels $L_{exit}, L_c\rangle$
$\langle$code for $b\rangle$
branch on b to L_c
jump to L_{exit}
$L_c : \langle$code for $c\rangle$
L_{exit} :

The definition of compile(**if** b **then** c) is similar to this translation schema. First, two new labels, L_{exit} and L_c, are generated. Then, b and c are compiled, producing π_b and π_c, respectively. A code segment $\mathtt{SEGM}[L_c, \pi_c\ ;\ \mathtt{JUMP}\, L_{\text{exit}}]$ is created for the code for c which jumps to the exit label L_{exit} when done. This segment is sequenced with the code for the boolean $(\pi_b \diamond \langle \mathtt{JUMP}\, L_c, \mathtt{JUMP}\, L_{\text{exit}}\rangle)$, and then wrapped by an `ENDLABEL` command.

This is the first use of the code-level apply operator $\diamond$, and it was convenient to wait until now to discuss its intended interpretation. Observe that in the above definition for **if-then**, π_b is always code produced by a source language boolean expression, and so its intended interpretation is as a control-flow boolean value (i.e., a value in $\forall\alpha.\alpha \times \alpha \to \alpha$) which chooses between the alternative jumps $\mathtt{JUMP}\, L_c$ and $\mathtt{JUMP}\, L_{\text{exit}}$. The operator $\diamond$ represents the application of the control-flow boolean value signified by π_b to this choice of jumps. This intended interpretation is made formal in Chapter 5.

A translation schema similar to the one above for "**while** b **do** c" is:

```
$\langle$generate new labels $L_{exit}, L_c, L_{test}\rangle$
jump to $L_{test}$
$L_c$ :     $\langle$code for $c\rangle$
            jump to $L_{test}$
$L_{test}$ : $\langle$code for $b\rangle$
            branch on b to $L_c$
            jump to $L_{exit}$
$L_{exit}$ :
```

The control-flow MCG compiles "**while** b **do** c" similarly. First, new labels L_{exit}, L_c, and L_{test} are generated. Then, b and c are compiled to π_b and π_c, respectively. The code for the loop body, π_c ; JUMP L_{test}, is stored at L_c using SEGM, and the code for the loop test, $(\pi_b \diamond \langle \texttt{JUMP}\, L_c, \texttt{JUMP}\, L_{\text{exit}}\rangle)$, is stored at L_{test} using SEGM. The initial jump to the test code, JUMP L_{test} is then emitted. Finally, the exit label L_{exit} is declared using ENDLABEL.

Note that in both of the above definitions, π_b always ends will a branch; that is it has the form: (... ; BRLEQ $\texttt{r}_1$ $\texttt{r}_2$). We use the $\diamond$ operator as a "code-level" apply to pass in the appropriate labels to the branch. The code (... ; BRLEQ $\texttt{r}_1$ $\texttt{r}_2$) $\diamond \langle \texttt{JUMP}\, L_1, \texttt{JUMP}\, L_2\rangle$ is pretty-printed as (... ; BRLEQ $\texttt{r}_1$ $\texttt{r}_2$ $\texttt{L}_1$ $\texttt{L}_2$).

3.8 Monadic Code Generator for Closed Procedures

Figure 3.14 presents the MCG for the *FunExp* language. Here, *FunExp* expressions are compiled into IntProducers. The development of this MCG follows very closely the development of the procedural metacomputation-style compiler block in Section 2.5. Auxiliary functions `mksubr`, `mkcall`, and `mkargcall` analogous to those in Section 2.5 are defined to compile procedures definitions and procedure and procedure argument calls. Figures 3.15 and 3.16 give an ML implementation of this MCG.

$$\mathsf{M} = \mathcal{T}_{\mathsf{Env}}\, \mathit{Env}\, (\mathcal{T}_{\mathsf{Env}} \mathit{SD} (\mathcal{T}_{\mathsf{St}}\, \mathit{Label}\, \mathsf{Id})),$$

$$\mathsf{compile} : \mathit{Expr} \rightarrow \mathsf{M}(\mathsf{IntProducer})$$

$$\mathsf{compile}(v) = \texttt{rdEnv} \,\star\, \lambda\rho.(\rho\, v)$$

$$\mathsf{compile}(i) = \mathbf{unit}\langle \texttt{NOP}, \#i, \{\}\rangle$$

$$\begin{aligned}
&\mathsf{compile}(\mathbf{funcall}\; v) = \mathsf{savereg}(\texttt{rdEnv} \,\star\, (\lambda\rho.(\rho\; v)))\\
&\mathsf{compile}(\lambda n.e) =\\
&\qquad \texttt{rdEnv} \,\star\, \lambda\rho.\\
&\qquad\quad \mathbf{unit}(\lambda c.(\texttt{inEnv}\; \rho[n \mapsto c]\; (\mathsf{compile}\, e)))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{savereg} : \mathsf{M}(\mathsf{IntProducer}) \rightarrow \mathsf{M}(\mathsf{IntProducer})\\
&\mathsf{savereg}\, \varphi = \; \texttt{rdSD} \,\star\, \lambda S.\\
&\qquad\qquad \varphi \,\star_S\; \lambda\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle.\\
&\qquad\qquad \mathbf{unit}_S(\langle \pi \;\texttt{;}\; \texttt{ALLOC}(S) \;\texttt{;}\; S{:=}\mathit{rhs} \;\texttt{;}\; (\mathsf{pop}\, \mathit{tmps}), S, \{S\}\rangle)
\end{aligned}$$

$$\mathsf{compile}(\mathbf{funcall}(f, e)) =$$
$$\mathsf{savereg}\left(\begin{array}{l}
\texttt{rdEnv} \,\star\, \lambda\rho.\\
(\rho\, f) \,\star\, \lambda F : \mathsf{M}(\mathit{IntProd}) \rightarrow \mathsf{M}(\mathit{IntProd}).\\
\qquad F(\mathsf{compile}\, e)
\end{array}\right)$$

$$\begin{aligned}
&\mathsf{compile}(\mathbf{letclosed}\; f = e : \varphi \text{ in } e : \mathbf{intexp})) =\\
&\quad \texttt{rdSD} \,\star\, \lambda S.\\
&\quad (\texttt{mksubr}\; \varphi\; S^{+}\; (\mathsf{compile}\; e)) \,\star\, \lambda\langle \pi_f, \mathit{rhs}, \mathit{tmps}\rangle.\\
&\quad \texttt{newlabel} \,\star\, \lambda L_f.\\
&\quad \texttt{rdEnv} \,\star\, \lambda\rho.\\
&\qquad \texttt{inEnv}\rho[f \mapsto (\texttt{mkcall}\; \varphi\; S\; L_f)] \left(\begin{array}{l}
(\mathsf{compile}\; e) \,\star\, \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle.\\
\qquad\quad \mathbf{unit}\langle \pi_e; \texttt{SEGM}[L_f \,:\; \pi_f], \mathit{rhs}_e, \mathit{tmps}_e\rangle
\end{array}\right)
\end{aligned}$$

Figure 3.14: Code Generator for function expression language

```
fun compile (e:Expr) =
case e of

(Var v) =>
      rdEnv bind (fn env_rho as (env rho) => (rho v))   |

(lambdaN (n,b)) =>
      rdEnv bind (fn rho =>
         unit (Fun (fn c =>
           (inEnv (xEnv (n,c) rho) (compile b)))))   |

(funcall (f,e)) =>
      (savereg (
            rdEnv bind (fn rho_env as (env rho) =>
            (rho f) bind (fn fval as (Fun F) =>
                  (F (compile e)))) )  )  |

(fcall v) =>
       (savereg (rdEnv bind (fn env_rho as (env rho) => (rho v)))) |

(letclosed (f,phi,le,e)) =>
   rdSD bind (fn S =>
   (mksubr phi (newframe S) (compile le)) bind (fn c as (code(IP((SEQ pi_f),rhs,tmps))) =>
   newlabel bind (fn Lf =>
   rdEnv bind (fn rho =>
         (inEnv (xEnv (f,(mkcall phi S Lf)) rho)
              (compile e) bind (fn ip as (code(IP((SEQ pi_e),rhs_e,tmps_e))) =>
                     (unit (code(IP((SEQ (pi_e@[SEGM(Lf,pi_f)])), rhs_e,tmps_e))))
                                        )))))) ;
```

Figure 3.15: ML Implementation of Code Generator for functional expression language

```
fun Lambda (n,x) = rdEnv bind (fn rho => unit(Fun (fn c => inEnv (xEnv (n,c) rho) x)));

(* only for Mf: M(IntProd->IntProd) *)
fun Apply Mf Mx = rdEnv bind (fn rho => Mf bind (fn fv as (Fun f) => f (inEnv rho Mx)));

fun mkargcall phi S i =
case phi of
intexp =>
    newlabel bind (fn Lret =>
    rdSD bind (fn S' =>
         unit (code(IP ((SEQ [ACALL (i,[],S,S',Lret), LABEL Lret]),(simpRHS SBRS),[])))));

fun mksubr phi S a =
case phi of
intexp =>
  (inSD S (a bind (fn ip as (code(IP((SEQ pi),rhs,tmps))) =>
     (unit (code (IP (SEQ(pi @ [(LOADSBRS rhs),(DEALLOC tmps),RETURN]), (simpRHS SBRS),[]))))))) |

(arrow(intexp,intexp)) =>
  (inSD S
        ((Apply a (mkargcall intexp S 1)) bind (fn ip as (code(IP((SEQ pi),rhs,tmps))) =>
        unit(code(IP(SEQ (pi@[LOADSBRS rhs,(DEALLOC tmps),RETURN]),(simpRHS SBRS),[]))))));

fun mkcall phi S Lf =
case phi of
intexp =>
    newlabel bind (fn Lret =>
    rdSD bind (fn S' =>
    rdEnv bind (fn rho_env as (env rho) =>
    unit (code(IP((SEQ [CALL(Lf,[],S,S',Lret), LABEL Lret]), simpRHS(SBRS),[])))))) |

(arrow(intexp,intexp)) =>
(Lambda ((ms 1),
    (newlabel bind (fn Lret =>
    rdSD bind (fn S' =>
    rdEnv bind (fn rho_env as (env rho) =>
    (mksubr intexp (newframe S') (rho (ms 1))) bind (fn ip as (code(IP((SEQ pi),_,[]))) =>
    newlabel bind (fn L1 =>
        unit (code(IP((SEQ [CALL(Lf,[L1],S,S',Lret), SEGM (L1,pi), LABEL Lret]),
                                          simpRHS(SBRS),[])))))))))));
```

Figure 3.16: ML Implementation of mksubr, mkcall, and mkargcall

Chapter 4

Modular Compiler Correctness

Chapters 4 and 5 present a case study in modular compiler verification for the language $\mathsf{Src} = \mathit{Exp}+\mathit{Imp}+\mathit{Block}+\mathit{Bool}+\mathit{Control\text{-}flow}$. This chapter establishes the correctness relation between the standard semantics, $[\![-]\!]$, and compilation semantics, $\mathcal{C}[\![-]\!]$, for Src. Chapter 5 demonstrates a correctness relation between $\mathcal{C}[\![-]\!]$ and the monadic code generator $\mathsf{compile}$ for Src. All semantic definitions of the Src language used in Chapters 4 and 5 are summarized in Appendix A beginning on page 127.

Section 4.1 surveys the challenges involved in specifying and verifying modular compilers. Broadly speaking, these challenges arise from two sources:

- Relating staged computation to unstaged computation requires demonstration that delaying parts of a computation (i.e., staging) does not change the value which is eventually produced. In the context of modular compiler verification, this means relating the monad-valued (unstaged) standard semantics, $[\![-]\!]$, to the metacomputation-valued (staged) compilation semantics, $\mathcal{C}[\![-]\!]$.

- Our language definitions are *representationally-independent*, due to structuring with monads and monad transformers. These language definitions are meant to be interpreted in many different contexts (and, hence, in many different monads). To maintain generality, reasoning about these definitions requires *representationally-independent* correctness specifications that make minimal assumptions about the monads underlying them.

Section 4.2 introduces a novel form of program specification called *observational program specification*, which copes very well with this second challenge. Although observational program specification is used here to specify and verify modular compilers, it is, in fact, a general technique applicable to many monadic contexts. A novel proof technique for observational specifications is presented in Section 4.2.1.

Verifying monadic specifications requires certain basic assumptions about the way that the non-proper monadic combinators (i.e., `rdEnv`, `update`, `callcc` , etc.) interact, and these axioms are listed in Section 4.3. Section 4.4 presents a change to the representation of store which allows scoping of effects through the use of explicit allocation and deallocation of storage. To simplify the task of specifying and proving a correctness relation between $[\![-]\!]$ and $\mathcal{C}[\![-]\!]$, another denotational definition of Src—the *staged standard semantics* $\mathcal{S}[\![-]\!]$— is introduced and both $[\![-]\!]$ and $\mathcal{C}[\![-]\!]$ are formally related to it. Section 4.5 defines the staged standard semantics, $\mathcal{S}[\![-]\!]$, and proves its equivalence with respect to the standard semantics $[\![-]\!]$ in Theorem 2. The staged standard semantics is a means of coping with the first bullet above.

The heart of this case study in modular compiler verification is presented in Section 4.7. The correctness of individual reusable compiler building blocks is specified and verified independently of one another, and given certain requirements called *linking conditions*, correct compiler building blocks may be combined into correct compilers. This achieves a certain level of modularity and reusability in the compiler proofs developed in this style. RCBB correctness specifications involve relating $\mathcal{S}[\![-]\!]$ and $\mathcal{C}[\![-]\!]$, and the major challenge in this endeavor originates in the use of implementation-level data (e.g., labels, code store, etc.) by $\mathcal{C}[\![-]\!]$ which $\mathcal{S}[\![-]\!]$ does not use. The careful use of observational program specification here helps meet this challenge. Section 4.6 lists supporting lemmas.

An overview of the correctness proof of the compiler for $\mathsf{Src} = \mathit{Exp} + \mathit{Imp} + \mathit{Block} + \mathit{Bool} + \mathit{ControlFlow}$ is shown in Figure 4.1. The first two links in Figure 4.1 between $[\![-]\!]$ and $\mathcal{S}[\![-]\!]$ and between $\mathcal{S}[\![-]\!]$ and $\mathcal{C}[\![-]\!]$ are established in Sections 4.5 and 4.7.

$$\begin{tikzcd}
\text{Standard Semantics } [\![t]\!] : \mathsf{M}(Value) \arrow[d, leftrightarrow, "\text{Equivalence Theorem 2}"] \\
\text{Staged Standard Semantics } \mathcal{S}[\![t]\!] : \mathsf{Static}(\mathsf{Dynam}(Value)) \arrow[d, leftrightarrow, "\text{Observational Specifications } (\textbf{Exp-spec},\ldots)"] \\
\text{Compilation Semantics } \mathcal{C}[\![t]\!] : \mathsf{Static}(\mathsf{Dynam}(Value)) \arrow[d, leftrightarrow, "\text{Equivalence of } \mathcal{C}[\![-]\!] \text{ \& } \mathsf{compile} \text{ Theorem 12}"] \\
\text{Monadic Code Generator } \mathsf{compile}(t) : \mathsf{Static}(\mathsf{TargetLang})
\end{tikzcd}$$

Figure 4.1: Src Compiler Correctness

4.1 Modular Compiler Proofs and Linking Conditions

Chapter 2 presents an implementation of RCBBs as staged denotational definitions. Recall that for each phrase type phrase, the *compilation semantics* of phrase

$$\mathcal{C}[\![-]\!] : \mathsf{phrase} \to \mathsf{Static}(\mathsf{Dynam}(Value))$$

was introduced. Each of these phrase types also has a *standard* semantics $[\![-]\!] : \mathsf{phrase} \to \mathsf{M}(Value)$ for an appropriate monad M which defines the usual meaning of a phrase as it is found in [47, 25, 10]. It would be tempting, then, to define the correctness of a RCBB as the equation $\mathcal{C}[\![t]\!] = [\![t]\!]$, but this is clearly unacceptable as it fails to even typecheck. More to the point is that $\mathcal{C}[\![t]\!]$ may use implementation-level data (such as labels, code store, etc.), and the compilation $\varphi_t : \mathsf{Dynam}(Value)$ produced by $\mathcal{C}[\![t]\!]$ may update the store, make jumps, etc., while the standard meaning of $[\![t]\!]$ generally "knows" nothing of this implementation-level data.

Because $\mathcal{C}[\![t]\!] : \mathsf{Static}(\mathsf{Dynam}(Value))$ is a metacomputation, it can be awkward to compare it with $[\![t]\!] : \mathsf{M}(Value)$. For example, the standard semantics for expressions can be interpreted within Dynam, but when the expression language is extended with variables, access to the environment is necessary. But the environment is part of the Static monad, and so it is not clear how to access environments within $[\![-]\!]$. The solution is to *stage* the standard semantics (i.e., make it *metacomputation*-valued rather than monad-valued). In Section 4.5, the *staged standard semantics* for the source language is introduced. While the standard semantics of a term $[\![t]\!] : \mathsf{M}(Value)$ is monad-valued, its staged standard semantics $\mathcal{S}[\![t]\!] : \mathsf{Static}(\mathsf{Dynam}(Value))$ is metacomputation-valued. Theorem 2 proves the equivalence of $\mathcal{S}[\![-]\!]$ with respect to $[\![-]\!]$.

As an example, the RCBB correctness assertion for an integer expression e will require that, if φ_e is the compilation produced by $\mathcal{C}[\![e]\!]$, then φ_e will equal $[\![e]\!]$ when executed in an appropriate store. Just what is meant by "appropriate store" will be defined in Section 4.7.2, but intuitively it means a store whose shape φ_e expects.

Although individual RCBBs may satisfy their individual correctness assertions, combining two correct RCBBs does not necessarily yield a correct compiler. The reason for this is that two RCBBs may update the same shared state and, therefore, may interfere with one another to produce incorrect results. Basic rules specifying correct RCBB interaction, which we call *linking conditions*, must be followed to ensure the "coherence" of the shared data. Consider the RCBBs for expressions, *Exp*, and imperative features, *Imp*, which are summarized in Figures 4.4 (on page 105) and 4.5 (on page 106), respectively. In an assignment "x:=e" in the combined language $Exp + Imp$, the target code resulting from the compilation of e will read, write to the store and allocate and deallocate memory cells in the store. So, it is possible that the code for e could affect the memory cell in which x is kept, and so the resulting code for "x:=e" may not behave correctly.

For the code for "x:=e" to be correct, the *Imp* compiler block must require that the code for e (1) only read from the store existing before it is run, and (2) deallocate any temporary memory cells that are allocated by the code for e. *Any* expression RCBB which meets this requirement, and also satisfies the correctness specification for expression RCBBs, will produce a correct compiler for $Exp + Imp$. Requirements (1) and (2) constitute an informal linking condition between the *Exp* and *Imp* compiler blocks.

Generally speaking, linking conditions are fairly weak requirements about the way that individual RCBBs interact. The linking condition outlined above would be familiar to any compiler writer—it requires merely that code for expressions not update any storage existing when it receives control and that it should pop the stack before it leaves. Furthermore, some linking conditions come essentially for free. For the language $Exp + Imp + Block + Bool + CF$, only the control flow RCBB affects the label and code states (and it affects only those states), and so as a consequence of lifting, any correct control-flow RCBB can be added to a correct modular compiler for $Exp + Imp + Block + Bool$ to achieve a correct compiler for $Exp + Imp + Block + Bool + CF$.

4.2 Observations and Observational Program Specification

The principal advantage of monadic language specifications is that the underlying denotation of a term can be made arbitrarily complex without unnecessarily complicating its denotational definition. This property was referred to in Section 1.2.3 as the "representational independence" of monadic language specifications, and it, in part, makes modular compilation possible. A similar property is required of program specifications (i.e., statements about program properties such as correctness) for modular compilers and reusable compiler building blocks. The reason for this is simple: because any RCBB may appear in many different modular compilers, its correctness specification must be meaningful in each of these compiler contexts. This section introduces a representationally independent style of program specification called *observational program specification.* Although developed to specify and verify modular compilers and RCBBs, observational program specification is really a general form of formal specification which applies to a general monadic context.

As an example, consider the correctness of an imperative construct $\mathsf{p!}$ defined in a monad with a state Sto. Generally, a correctness specification of an imperative feature like this would take the form of a relation $\Re$ between input and output states σ_0 and σ_1, so that $\sigma_0 \Re \sigma_1$ means that the state σ_1 may result from the execution of $\mathsf{p!}$ in σ_0. If $\mathsf{p!}$ were defined in the single state monad $\mathsf{St}\, a = Sto \to a \times Sto$, then the correctness of $\mathsf{p!}$ would be written:

$$\forall \sigma_0 : Sto.\ \sigma_0 \Re (\pi_2(\mathsf{p!}\ \sigma_0)) \tag{4.1}$$

where π_2 is the second projection function $\lambda\langle -, x\rangle.x$. However, if p were reinterpreted in the "Environment+State" monad (cf. Figure 1.9) $\mathsf{EnvSt}\, a = Env \to Sto \to a \times Sto$, then the above correctness specification would be rewritten as:

$$\forall \rho_0 : \mathit{Env}.\ \forall \sigma_0 : \mathit{Sto}.\ \sigma_0 \, \Re \, (\pi_2(\mathsf{p!}\ \rho_0\ \sigma_0)) \tag{4.2}$$

One can see from these two examples that every monad in which p! is interpreted requires a new correctness specification.

Traditional "functional-style" denotational *program* specifications (like 4.1 and 4.2) lack modularity in precisely the same manner and for identical reasons as traditional, functional-style denotational *language* specifications (cf. Section 1.2.3). Because specifications 4.1 and 4.2 rely on the fixed structure of St and EnvSt, respectively, there is no way of reusing them when p! is reinterpreted in another monad; or in other words, they are *representationally-dependent* specifications. If one is reasoning about a particular program defined in a fixed monad, then this representational dependence presents no great obstacle. Nevertheless, it goes against the spirit of the monadic approach to language design to carefully define a language in terms of a monad only to ignore the monad when specifying and verifying programs.

However, the representational dependence of functional-style program specifications *is* a major obstacle to verifying modular compilers. Because reusable compiler building blocks are meant to be used in many different compilers, they must be interpreted in many different monads. The lack of a single statement of correctness makes a functional-style specification and verification of reusable compiler building blocks and modular compilers essentially intractable. Verifying modular compilers constructed from reusable compiler building blocks requires a representationally-independent style of program specification.

The key insight here is that, because the language definitions we use are parameterized by a monad, it is necessary to develop a specification style that is also parameterized by a monad. The first step is to add a new expressed value type **prop**, which plays the role of the traditional (i.e., non-control-flow) boolean type.

Definition 17 (prop) **prop** $=$ `true` $+$ `false`

The correctness condition $(\sigma_0 \, \Re \, \sigma_1)$: **prop** may then be computed value for appropriate stores σ_0

and σ_1:

$$\begin{array}{lcl} \texttt{getSto} \star \lambda\sigma_0. & & \\ \mathsf{p!} \star \lambda\_. & & \mathsf{p!} \star \lambda\_. \\ & = & \\ \texttt{getSto} \star \lambda\sigma_1. & & \quad \mathbf{unit}(\texttt{true}) \\ \quad \mathbf{unit}(\sigma_0 \,\Re\, \sigma_1 : \mathbf{prop}) & & \end{array} \tag{4.3}$$

What does this equation mean? Examining the left-hand side of Equation 4.3, the execution of $\mathsf{p!}$ is couched between two calls to `getSto`, of which the first call returns the input store σ_0 and the second call returns the output store σ_1 resulting from executing $\mathsf{p!}$. Note that σ_1 will reflect any updates to the store made by $\mathsf{p!}$. Finally, the truth-value of the **prop** expression $(\sigma_0 \,\Re\, \sigma_1)$ is returned. The right-hand side of Equation 4.3 executes $\mathsf{p!}$ and then always returns `true`. Observe also that it was necessary to execute $\mathsf{p!}$ on the right-hand side so that identical store updates would occur on both sides of the equation. Equation 4.3 requires that $(\sigma_0 \,\Re\, \sigma_1)$ be `true` for all input and output stores σ_0 and σ_1, respectively, which is precisely what we want.

Equation 4.3 is a representationally independent specification of $\mathsf{p!}$. In the single store monad St, it means precisely the same thing as 4.1, while in the monad EnvSt, 4.3 means exactly the same thing as 4.2. In fact, equation 4.3 makes sense in any monad where $\mathsf{p!}$ makes sense. It is called an *observational* specification because the left-hand side of 4.3 gathers certain data from different stages in the computation (i.e., stores σ_0 and σ_1) and "observes" whether or not $(\sigma_0 \,\Re\, \sigma_1)$ holds.

In observational specifications, a particular kind of computation of type $\mathsf{Dynam}(\mathbf{prop})$ called an "observation" is often useful. An *observation* is a computation which reads (and only reads!) data such as states and environments, and then observes a relation. Observations are utterly innocent computations in that they never change states, fail, or call continuations. We can make this notion of "innocent computations" formal with:

Definition 18 (Innocent Computations) *A computation $\Upsilon : \mathsf{M}(\alpha)$ is innocent, if and only if*

$$\forall x : \mathsf{M}(\tau).\ \Upsilon \star \lambda\_.\ x = x \star \lambda v.\ \Upsilon \star \lambda\_.\ \mathbf{unit}(v) = x$$

Notice that stateful computation can easily lose innocence:

$$\texttt{get} \neq \texttt{update}[\lambda l.l+1] \star \_.\texttt{get} \neq \texttt{get} \star \lambda\sigma.\texttt{update}[\lambda l.l+1] \star \lambda\_.\mathbf{unit}(\sigma)$$

Continuation-manipulating computations can also lose innocence, because, for arbitrary κ_0:

$$\mathbf{unit}(5) \neq \mathbf{unit}(5) \star \lambda v.(\texttt{callcc}\ \lambda\kappa.\kappa_0 7) \star \lambda\_.\mathbf{unit}(v)$$

If Ω produces an error or is non-terminating, then it is not innocent:

$$\mathbf{unit}(5) \neq \mathbf{unit}(5) \star \lambda v.\Omega \star \lambda\_.\mathbf{unit}(v) = \Omega,$$

Some computations are always innocent. For example, any computation constructed from an environment monad's "read" operators (e.g., `rdEnv` and `rdAddr`), an environment monad's "in" operators (e.g., `inEnv` and `inAddr`, assuming their arguments are innocent), or from the "get" operators of a state monad (e.g., `getSto` and `getCode`) are always innocent. Unit computations (i.e., $\mathbf{unit}(x)$, for any x) are also always innocent. Knowing that a computation is innocent is useful in the proofs developed below, not only because an innocent computation commutes with any other computation, but because it can be also be added to any computation without effect. That is, for any arbitrary computations φ_1, φ_2 and innocent computation Υ,

$$\varphi_1 \star \lambda v.\varphi_2 = \Upsilon \star \lambda x.\varphi_1 \star \lambda v.(\Upsilon \star \lambda y.\varphi_2)$$

The values x and y computed by Υ can be used as snapshots to characterize the "before" and "after" behavior of φ_1 just as the states σ_0 and σ_1 computed by `getSto` were used in Equation 4.3 on page 81.

Definition 19 (Observations) *An* observation *is an innocent computation of type* $\mathsf{M}(\mathbf{prop})$.

Definitions 20 and 21 introduces two observations which are used throughout the remainder of this thesis. Definition 22 lists three operations on observations. The first of these, Obs, defines an observational version of if-then-else, while the last two, AND and $\Rightarrow$, create new observations from existing ones.

Definition 20 ($\mathsf{FreshLoc}$) *The observation* $\mathsf{FreshLoc}(a)$ *means that every address above a is unused in the current store.*

$$\mathsf{FreshLoc}(a) = \mathtt{getSto} \star_D \; \lambda\sigma.\mathbf{unit}_D(\forall x \geq a.x \notin dom(\sigma))$$

Definition 21 ($\mathsf{FreshLabel}$) *The observation* $\mathsf{FreshLabel}(L) : \mathsf{Dynam}(\mathbf{prop})$ *means that every label greater than L is undefined in the current code store.*

$$\mathsf{FreshLabel}(L) = \mathtt{getCode} \star_D \; \lambda\Pi.\mathbf{unit}_D(\forall L' \geq L.(L' \notin dom\Pi))$$

Definition 22 (Observe-def) *We define a function* $\mathsf{Obs} : \forall\tau.\mathsf{M}(\mathbf{prop}) \times \mathsf{M}(\tau) \times \mathsf{M}(\tau) \to \mathsf{M}(\tau)$ *that encapsulates the notion of if-then-else for observations, and two functions* $\mathsf{AND}, \Rightarrow : \mathsf{M}(\mathbf{prop}) \times \mathsf{M}(\mathbf{prop}) \to \mathsf{M}(\mathbf{prop})$ *for combining observations:*

$$\begin{aligned}
\mathsf{Obs}(\theta, u, v) &= \theta \star \lambda test.\text{if } test \text{ then } u \text{ else } v \\
\theta_1 \;\mathsf{AND}\; \theta_2 &= \theta_1 \star \lambda p_1.\theta_2 \star \lambda p_2.\mathbf{unit}(p_1 \& p_2) \\
\theta_1 \Rightarrow \theta_2 &= \theta_1 \star \lambda p_1.\theta_2 \star \lambda p_2.\mathbf{unit}(p_1 \supset p_2)
\end{aligned}$$

where & and $\supset$ are the ordinary propositional connectives with the usual truth table definitions.

Notational convention: The double arrow $\Rightarrow$ combines observations, while the cup $\supset$ is the ordinary logical connective of type $\mathbf{prop} \times \mathbf{prop} \to \mathbf{prop}$.

The following theorem gives inference rule-like equations for manipulating observations. Note that in both of the rules below, the first arguments to Obs must always be observations (i.e., innocent). The proof of Theorem 1 demonstrates the soundness of these rules.

Theorem 1 (Observation Introduction and Elimination) *For observations θ, θ_1, θ_2:*

(Introduction) $u = \mathsf{Obs}(\theta, u, u)$

(Elimination) $\theta_1 \Rightarrow \theta_2 = \mathbf{unit}(\mathtt{true})$ *implies* $\mathsf{Obs}(\theta_1, \mathsf{Obs}(\theta_2, u, v), w) = \mathsf{Obs}(\theta_1, u, w)$

$\boxed{\text{Proof of Theorem 1}}$

$\boxed{\text{Case: Observation Introduction}}$

$$\begin{aligned}
\mathsf{Obs}(\theta, u, u) &= \theta \star \lambda test.\ \text{if } test \text{ then } u \text{ else } u \\
&= \theta \star \lambda test.\ u && (test \text{ is } \mathtt{true} \text{ or } \mathtt{false}) \\
&= u && (\theta \text{ is innocent})
\end{aligned}$$

$\boxed{\text{Case: Observation Elimination}}$

$$\begin{aligned}
&\mathsf{Obs}(\theta_1, \mathsf{Obs}(\theta_2, u, v), w) \\
&= \theta_1 \star \lambda t_1.\ \text{if } t_1 \text{ then } [\theta_2 \star \lambda t_2.\ \text{if } t_2 \text{ then } u \text{ else } v] \text{ else } w \\
&= \theta_1 \star \lambda t_1.\ \text{if } t_1 \text{ then } [\theta_2 \star \lambda t_2.\ \text{if } t_1 \supset t_2 \text{ then } u \text{ else } v] \text{ else } w && (\mathtt{true} \supset x = x) \\
&= \theta_1 \star \lambda t_1 . \theta_2 \star \lambda t_2.\ \text{if } t_1 \text{ then } [\text{if } t_1 \supset t_2 \text{ then } u \text{ else } v] \text{ else } w && (\theta_2 \text{ is innocent}) \\
&= \theta_1 \star \lambda t_1 . \theta_2 \star \lambda t_2 . \mathbf{unit}(t_1 \supset t_2) \star \lambda t.\ \text{if } t_1 \text{ then } [\text{if } t \text{ then } u \text{ else } v] \text{ else } w \\
& && (\mathbf{unit}(t_1 \supset t_2) \text{ is innocent}) \\
&= \theta_1 \star \lambda t_1 . \theta_2 \star \lambda t_2 . \mathbf{unit}(\mathtt{true}) \star \lambda t.\ \text{if } t_1 \text{ then } [\text{if } t \text{ then } u \text{ else } v] \text{ else } w \\
& && (\text{supposition}) \\
&= \theta_1 \star \lambda t_1 . \theta_2 \star \lambda t_2.\ \text{if } t_1 \text{ then } u \text{ else } w && (\text{left unit}) \\
&= \theta_1 \star \lambda t_1.\ \text{if } t_1 \text{ then } u \text{ else } w && (\theta_2 \text{ is innocent}) \\
&= \mathsf{Obs}(\theta_1, u, w)
\end{aligned}$$

□Theorem 1.

Lemma 1 *For observation θ,* $\mathsf{Obs}(\theta, x, y) \star f = \mathsf{Obs}(\theta, x \star f, y \star f)$

$\boxed{\text{Proof of Lemma 1}}$

$$\begin{aligned}
& \mathsf{Obs}(\theta, x, y) \star f \\
= \quad & (\theta \star \lambda test.\ \text{if } test \text{ then } x \text{ else } y) \star f \\
= \quad & \theta \star (\lambda test.(\text{if } test \text{ then } x \text{ else } y) \star f) \\
= \quad & \theta \star (\lambda test.\ \text{if } test \text{ then } x \star f \text{ else } y \star f) \\
= \quad & \mathsf{Obs}(\theta, x \star f, y \star f)
\end{aligned}$$

□Lemma 1.

4.2.1 Proof Technique for Observational Specifications

In this section, we outline a proof technique developed for observational specifications which is used repeatedly throughout the remainder of this thesis. It allows for observational properties to be propagated through sequential computations and has something of the flavor of programming logics such as specification and Floyd-Hoare logics[42, 27].

Many of the observational specifications proved later in this chapter have the following form (where $\mathsf{C}[-]$ is a closing context),

$$\mathsf{C}[\mathsf{Obs}(\theta, \varphi_1 \star \lambda v_1 \ldots \star \lambda v_{n-1}.\varphi_n, \gamma)] = \mathsf{C}[\mathsf{Obs}(\theta, \delta_1 \star \lambda v_1 \ldots \star \lambda v_{n-1}.\delta_n, \gamma)] \tag{4.4}$$

The observational property θ serves as a "precondition" (in the same sense as in specification and Floyd-Hoare logics[42, 27]) which will be propagated through the sequence of computations φ_i, converting them to δ_i along the way. Knowing that θ returns `true` at the beginning of $\varphi_1 \star \lambda v_1 \ldots \star \lambda v_{n-1}.\varphi_n$, each φ_i may be viewed as an "observation transformer" analogous to the view of imperative statements as "predicate transformers" in some programming logics. As θ is propagated, it is transformed by the φ_i into intermediate observational properties: $\theta_1, \ldots, \theta_n$. Using

Observation Introduction, the computations φ_i may be guarded with these intermediate properties:

$$\begin{array}{lll} \varphi_1 \star \lambda v_1.\varphi_2 \star \lambda v_2.\ldots \star \lambda v_{n-1}.\varphi_n = & \mathsf{Obs}(\theta_1, \varphi_1 \star \lambda v_1. & \\ & \quad \mathsf{Obs}(\theta_2, \varphi_2 \star \lambda v_2. & \\ & \qquad \vdots & \\ & \qquad \mathsf{Obs}(\theta_n, \varphi_n, \varphi_n) & \\ & \qquad ,\ldots) & \\ & \quad ,\ldots) & \end{array} \tag{4.5}$$

Note that both sides of Equation 4.5 occur within the `true` branch of the "$\mathsf{Obs}(\theta, \ldots, -)$" on the left-hand side of Equation 4.4.

Under the following conditions, the right-hand side of Equation 4.5 may be equated to the right-hand side of Equation 4.4.

1. $\mathsf{Obs}(\theta_i, \varphi_i \star \lambda v_i.\mathsf{Obs}(\theta_{i+1}, x, \ldots), \ldots) = \mathsf{Obs}(\theta_i, \varphi_i \star \lambda v_i.x, \ldots)$

2. $\mathsf{Obs}(\theta_i, \varphi_i \star \lambda v_i.\ldots, \ldots) = \mathsf{Obs}(\theta_i, \delta_i \star \lambda v_i.\ldots, \ldots)$

3. $\theta \Rightarrow \theta_1 = \mathbf{unit}(\mathtt{true})$

Condition 1 allows for later observational properties to be discharged, Condition 2 converts the φ_i into δ_i, and finally, if Condition 3, then the initial condition θ_1 may be discharged. Thus, given Conditions (1)-(3), Equation 4.4 holds.

4.3 The Monadic Interface and Its Properties

The use of monad transformers does not simply create a new monad from an existing one by adding new computational material, but it also defines additional so-called "non-proper" combinators for manipulating data internal to the monad (the "proper" combinators of a monad are just **unit** and $\star$). Before the compiler building blocks can be verified, it must be understood how these combinators interact with one another. In this case study, the non-proper combinators involved are the various "update" and "get" operators added by the state monad transformer, the "read" and "in" operators added by the environment monad transformer, and the "call with current continuation" operator callcc added by the CPS monad transformer. This section introduces axioms characterizing the behavior of these non-proper monadic combinators, as well as the well-known monad laws[25, 10, 30, 47]. No claim is made that these axioms for the non-proper combinators are complete characterizations, but only that they specify what is needed to prove the correctness of the Src compiler and are clearly correct.

The following laws specify the behavior of $\star$ and **unit** in a monad.

Axiom 1 (Monad Laws) *For a monad* M*, the bind* $\star$ *and unit* **unit** *operations have the following properties:*

1. *(left unit):* $(\mathbf{unit}\; a) \star k = k\; a$

2. *(right unit):* $x \star \mathbf{unit} = x$

3. *(associativity):* $x \star (\lambda a.(k\; a \star h)) = (x \star k) \star h$

4. *(associativity):* $x \star (\lambda a.y \star \lambda b.o) = (x \star \lambda a.y) \star \lambda b.o$*, if* a *does not occur free in* o*.*

The update/get laws characterize the behavior of the update and get operators added by a state monad transformer.

Axiom 2 (update/get interactions) *Assume that* update *and* get *are defined in* M *by the state monad transformer* $\mathcal{T}_{\mathsf{St}}\,\tau$ *in the following:*

1. $(\mathtt{update}\,f) \star \lambda\_.(\mathtt{update}\,g) = \mathtt{update}\,f;g$

2. $\mathtt{update}[l \mapsto v] \star \lambda\_.\mathtt{get} \star \lambda\sigma.\mathbf{unit}(\sigma\, l) = \mathtt{update}[l \mapsto v] \star \lambda\_.\mathbf{unit}(v)$

3. $\mathtt{get} \star \lambda\sigma.\mathtt{get} \star \lambda\sigma'.\mathcal{F}(\sigma,\sigma') = \mathtt{get} \star \lambda\sigma.\mathtt{get} \star \lambda\sigma'.\mathcal{F}(\sigma,\sigma)$ *where* $\mathcal{F} : \tau \times \tau \rightarrow \mathsf{M}a$*.*

4. $\mathtt{update}\Delta \star \lambda\_.\mathtt{get} = \mathtt{get} \star \lambda\sigma.\mathtt{update}\Delta \star \lambda\_.\mathbf{unit}(\Delta\sigma)$

5. *For* $\mathcal{F} : \tau \times \tau \rightarrow \mathsf{M}a$*,*

$$\mathtt{get} \star \lambda\sigma.\mathtt{update}\Delta \star \lambda\_.\mathtt{get} \star \lambda\sigma'.\mathcal{F}(\sigma,\sigma') = \mathtt{get} \star \lambda\sigma.\mathtt{update}\Delta \star \lambda\_.\mathcal{F}(\sigma,\Delta\sigma)$$

Rule 2.1 shows how updating by f and then updating by g is the same as just updating by their composition $f;g$. Rule 2.2 states that updating a location l by a value v and then reading from l is that same as updating l by v and just returning v. Rule 2.3 requires that performing two `get` operations in succession retrieves precisely the same value.

Axiom 3 (Callcc Interactions) *The following specify the interactions between* `callcc` *and the other combinators in* $\mathsf{M} = \mathsf{Static} + \mathsf{Dynam}$*:*

1. $\texttt{callcc}\ \lambda\kappa.x \star \lambda v : a.\mathcal{F}(v,\kappa) = x \star \lambda v.\texttt{callcc}\ \lambda\kappa.\mathcal{F}(v,\kappa)$ *where* $\mathcal{F} : a \times (a \to \mathsf{M}b) \to \mathsf{M}b$.

2. *For idempotent* $I : \mathsf{M}(\texttt{void})$ *(i.e.,* $I = I \star \lambda\_.I$*):*

$$(\texttt{callcc}\ \lambda\kappa.x \star \kappa) \star \lambda\_.I = [\texttt{callcc}\ (\lambda\kappa.x \star \lambda_{.}I \star \kappa)] \star \lambda\_.I$$

3. $x \star (\lambda v.\ \texttt{callcc}\ (\lambda\kappa.\kappa v)) = x$

4. $x = \texttt{callcc}\ \lambda\kappa.x \star \kappa$

Note that Axiom 3.2 follows directly from Axiom 3.4 by the following proof:

$$\begin{array}{rcl} & & [\texttt{callcc}\ \lambda\kappa.x \star \kappa] \star \lambda\_.I \\ (\text{Ax } 3.4\ ) & = & x \star \lambda\_.I \\ (\text{idem.}) & = & x \star \lambda\_.I \star \lambda\_.I \\ (\text{Ax } 3.4\ ) & = & [\texttt{callcc}\ \lambda\kappa.x \star \lambda\_.I \star \kappa] \star \lambda\_.I \end{array}$$

Axiom 4 (`update` Interactions) *Assume that* `update` *is added by the state monad transformer* $\mathcal{T}_{\mathsf{St}}\,\tau$*,* `update'/get'` *are added by the (different) state monad transformer* $\mathcal{T}_{\mathsf{St}}\,\tau'$*, and* `rd/in` *are defined by a monad transformer* $\mathcal{T}_{\mathsf{Env}}\,\tau''$*.*

1. $\mathbf{unit}(x) \star \lambda v.(\texttt{update}\,\Delta) \star \lambda\_.\mathcal{F}v = (\texttt{update}\,\Delta) \star \lambda\_.\mathbf{unit}(x) \star \mathcal{F}$ *for any* $x : a$ *and* $\mathcal{F} : a \to \mathsf{M}b$,

2. $\texttt{rd} \star \lambda\rho.(\texttt{update}\,\Delta) \star \lambda\_.(f\,\rho) = (\texttt{update}\,\Delta) \star \lambda\_.\texttt{rd} \star f$ *for any* $x : a$ *and* $\mathcal{F} : \tau'' \to \mathsf{M}a$,

3. *For any* $x : \mathsf{M}a$, $\rho : \tau''$, *and* $\mathcal{F} : a \to \mathsf{M}b$,

$$x \star \lambda v.(\texttt{update}\,\Delta) \star \lambda\_.(\mathcal{F}\,v) = (\texttt{update}\,\Delta) \star \lambda\_.x \star \mathcal{F} \implies$$
$$(\texttt{in}\,\rho\,x) \star \lambda v.(\texttt{update}\,\Delta) \star \lambda\_.(\mathcal{F}\,v) = (\texttt{update}\,\Delta) \star \lambda\_.(\texttt{in}\,\rho\,x) \star \mathcal{F}$$

4. $\texttt{update}\Delta \star \lambda\_.\texttt{update}'\Xi = \texttt{update}'\Xi \star \lambda\_.\texttt{update}\Delta$

5. $(\texttt{in}\;\rho\;x) \star \lambda v.\texttt{update}\Delta = \texttt{in}\;\rho\;(x \star \lambda v.\texttt{update}\Delta)$

Axiom 5 (get Interactions) *Assume that* `get` *is added by the state monad transformer* $\mathcal{T}_{\mathsf{St}}\,\tau$, `update'`/`get'` *are added by the (different) state monad transformer* $\mathcal{T}_{\mathsf{St}}\,\tau'$, *and* `rd`/`in` *are defined by a monad transformer* $\mathcal{T}_{\mathsf{Env}}\,\tau''$.

1. $\mathbf{unit}(x) \star \lambda v.\texttt{get} \star \lambda\sigma.\mathcal{F}(v,\sigma) = \texttt{get} \star \lambda\sigma.\mathbf{unit}(x) \star \lambda v.\mathcal{F}(v,\sigma)$ *for any* $x : a$ *and* $\mathcal{F} : a \times \tau \to \mathsf{M}b$,

2. $\texttt{rd} \star \lambda\rho.\texttt{get} \star \lambda\sigma.\mathcal{F}(\rho,\sigma) = \texttt{get} \star \lambda\sigma.\texttt{rd} \star \lambda\rho.\mathcal{F}(\rho,\sigma)$ *for any* $\mathcal{F} : \tau' \times \tau \to \mathsf{M}b$

3. *For any* $x : \mathsf{M}a$, $\rho : \tau''$, *and* $\mathcal{F} : a \times \tau \to \mathsf{M}b$,

$$x \star \lambda v.\texttt{get} \star \lambda\sigma.\mathcal{F}(v,\sigma) = \texttt{get} \star \lambda\_.x \star \lambda v.\mathcal{F}(v,\sigma) \implies$$
$$(\texttt{in}\,\rho\,x) \star \lambda v.\texttt{get} \star \lambda\sigma.\mathcal{F}(v,\sigma) = \texttt{get} \star \lambda\sigma.(\texttt{in}\,\rho\,x) \star \lambda v.\mathcal{F}(v,\sigma)$$

4. $\texttt{get} \star \lambda\sigma.\texttt{update}'\Delta \star \lambda\_.f\sigma = \texttt{update}'\Delta \star \lambda\_.\texttt{get} \star f$ *for any* $\mathcal{F} : \tau \to \mathsf{M}a$.

5. $\texttt{get} \star \lambda\sigma.\texttt{get}' \star \lambda\sigma'.\mathcal{F}(\sigma,\sigma') = \texttt{get}' \star \lambda\sigma'.\texttt{get} \star \lambda\sigma.\mathcal{F}(\sigma,\sigma')$ *for any* $\mathcal{F} : \tau \times \tau' \to \mathsf{M}a$.

6. $(\texttt{in}\ \rho\ x) \star \lambda v.\texttt{update}\Delta = \texttt{in}\ \rho\ (x \star \lambda v.\texttt{update}\Delta)$

Axiom 6 (Environment Axioms) *The following axioms (due to Liang [26, 24]) characterize the environment combinators* `rdEnv`*,* `rdAddr`*,* `inEnv`*, and* `inAddr`*. Although these laws are stated only for* `rdEnv` *and* `inEnv`*, the corresponding Addr operation may be substituted below.*

1. $(\texttt{inEnv}\ \rho) \circ \mathbf{unit} = \mathbf{unit}$

2. $\texttt{inEnv}\ \rho\ (c_1 \star \lambda v.c_2) = (\texttt{inEnv}\ \rho\ c_1) \star \lambda v.(\texttt{inEnv}\ \rho\ c_2)$

3. $\texttt{inEnv}\ \rho\ \texttt{rdEnv} = \mathbf{unit}\rho$

4. $\texttt{inEnv}\ \rho\ (\texttt{inEnv}\ \rho'\ e) = \texttt{inEnv}\ \rho'\ e$

4.4 Scoping of Effects with Explicit Allocation and Deallocation

In this section, a new representation of the *Sto* state is introduced which allows restrictions to be placed on the scope of side-effects. In previous chapters, *Sto* has been represented as a function of type $Addr \to int$. While this suffices for the generation of code in Chapter 2, it complicates the statement and verification of modular compiler correctness because it is difficult to limit the scope of effects with that representation of *Sto*.

Why? Consider the dynamic part of the definition of $\mathcal{C}[\![-t]\!]$ in Figure 2.1 on page 25:

$$\begin{aligned}&\varphi_t \ \star_D\ \lambda i.\\ &\texttt{Thread}(i, a)\ \star_D\ \lambda v.\\ &\quad\ \ \mathbf{unit}_D(-v)\end{aligned}$$

After the execution of $\texttt{Thread}(i, a)$, the contents of address a will be i in the global *Sto* state (unless a is reused in the compilation of some other part of the program being compiled). As will be seen in Section 4.7.2, the correctness specification for $\mathcal{C}[\![- : Exp]\!]$ requires something like, given certain preconditions on the *Sto* state, the above dynamic term *equals* $[\![-t]\!]$. Clearly then, some limit must be placed on the scope of effects from the use of temporary storage for such an equality to hold because $[\![-t]\!]$ has no side-effects on *Sto*.

One way to limit the scope of effects is to explicitly allocate and deallocate temporary storage, as in:

$$\begin{array}{l}
\varphi_t \star_D \lambda i. \\
\texttt{Alloc}(a) \star_D \lambda\_. \\
\texttt{Thread}(i, a) \star_D \lambda v. \\
\texttt{deAlloc}(a) \star_D \lambda\_. \\
\qquad \mathbf{unit}_D(-v)
\end{array}$$

The representation of store is changed to be a set of address/value pairs $Sto \subset \{\langle a, v\rangle \ : \ a \in Addr \ \& \ v \in int\}$ defining a partial function from *Addr* to *int*. If $\texttt{Alloc}(a)$ adds $\langle a, 0\rangle$ to the current store and $\texttt{deAlloc}(a)$ removes any $\langle a, v\rangle$ from the current store, then as long as a is not in the domain of the current store before the execution of the above dynamic computation, then it will appear to have no affect on the global store at all. This method of restricting the scope of effects is not new—it is inspired by the Algol functor category semantics of Reynolds[41].

The compilation semantics $\mathcal{C}[\![- : \mathsf{Src}]\!]$ used in the compiler correctness result (displayed in Figures 4.4, 4.5, 4.6, 4.7, 4.8, and 4.9) uses this new representation of *Sto*, as do all of the other semantic definitions presented in Chapters 4 and 5. They are repeated in full in Appendix A.

Definition 23 (Storage Operators) *The following operators are used to store, read, allocate,* *and deallocate to and from addresses in the new representation of Sto.*

$$\texttt{store}(a,i) = \ \texttt{getSto} \star_D \lambda\sigma. \left(\begin{array}{l} \textit{if } \langle a,-\rangle \in \sigma \textit{ then} \\ \quad \texttt{updateSto}(\lambda\sigma.(\sigma\backslash\langle a,-\rangle) \cup \langle a,i\rangle) \\ \textit{else } \perp_{\mathsf{Dynam(void)}} \end{array} \right)$$

$$\texttt{read}(a) = \ \texttt{getSto} \star_D \lambda\sigma.[\text{if } \langle a,v\rangle \in \sigma \text{ then } \mathbf{unit}_D(v) \text{ else } \perp_{\mathsf{Dynam(void)}}]$$

$$\texttt{Alloc}(a) = \texttt{getSto} \star_D \lambda\sigma.\ \textit{if } a \in \sigma \textit{ then } \perp_{\mathsf{Dynam(void)}} \textit{ else } \texttt{updateSto}[\sigma^* \mapsto \sigma^* \cup \{\langle a,0\rangle\}]$$

$$\texttt{deAlloc}(a) = \texttt{getSto} \star_D \lambda\sigma.\ \textit{if } a \notin \sigma \textit{ then } \perp_{\mathsf{Dynam(void)}} \textit{ else } \texttt{updateSto}[\sigma^* \mapsto \sigma^* \setminus \{\langle a,-\rangle\}]$$

4.5 The Staged Standard Semantics

Because $\mathcal{C}[\![t]\!] : \mathsf{Static}(\mathsf{Dynam}(\mathit{Value}))$ is a metacomputation, it can be awkward to compare it with $[\![t]\!] : \mathsf{M}(\mathit{Value})$. For example, the standard semantics for expressions can be interpreted within Dynam, but when the expression language is extended with variables, access to the environment is necessary. But the environment is part of the Static monad, and so it is not clear how to access environments within $[\![-]\!]$. The solution is to *stage* the standard semantics; that is, make it *metacomputation*-valued rather than monad-valued. In this section, the *staged standard semantics* for the source language (displayed in Figure 4.2) is introduced. While the standard semantics of a term $[\![t]\!] : \mathsf{M}(\mathit{Value})$ is monad-valued, its staged standard semantics $\mathcal{S}[\![t]\!] : \mathsf{Static}(\mathsf{Dynam}(\mathit{Value}))$ is metacomputation-valued. The standard semantics of the source language is displayed in Figure 4.3. Theorem 2 proves the equivalence of $\mathcal{S}[\![-]\!]$ with respect to $[\![-]\!]$.

A combinator `unquote` is introduced in the following definition. `unquote` is used to "run" a metacomputation by first evaluating the static part and then evaluating the dynamic part.

Definition 24 (unquote) *For* $\mathsf{M} = \mathsf{Static} + \mathsf{Dynam}$*, the function* $\texttt{unquote} : \mathsf{M}(\mathsf{M}(a)) \to \mathsf{M}(a)$ *is defined as:*

$$\texttt{unquote}(x : \mathsf{M}(\mathsf{M}(a))) = x \star \lambda i.i$$

$$\mathcal{S}\llbracket - \rrbracket : \mathsf{Src} \to \mathsf{Static}(\mathsf{Dynam}(\texttt{void} + \mathit{int} + \mathbf{bool}))$$

$$\mathcal{S}\llbracket i \rrbracket = \mathbf{unit}_S(\mathbf{unit}_D(i))$$

$$\mathcal{S}\llbracket -e \rrbracket = \mathcal{S}\llbracket e \rrbracket \;\star_S\; \lambda\varphi_e : \mathsf{Dynam}(\mathit{int}).\; \mathbf{unit}_S(\varphi_e \;\star_D\; \lambda i.\mathbf{unit}_D(-i))$$

$$\mathcal{S}\llbracket x \rrbracket = \texttt{rdEnv} \;\star_S\; \lambda\rho.(\rho\, x) \;\star_S\; \lambda a : \mathit{Addr}.\; \mathbf{unit}_S(\texttt{read}(a))$$

$$\mathcal{S}\llbracket c_1 \,;\, c_2 \rrbracket = \mathcal{S}\llbracket c_1 \rrbracket \;\star_S\; \lambda\varphi_1.\mathcal{S}\llbracket c_2 \rrbracket \;\star_S\; \lambda\varphi_2.\; \mathbf{unit}_S(\varphi_1 \;\star_D\; \lambda\_.\varphi_2)$$

$$\begin{aligned}\mathcal{S}\llbracket x := e \rrbracket = \;& \texttt{rdEnv} \;\star_S\; \lambda\rho.\\ & \quad (\rho\, x) \;\star_S\; \lambda a : \mathit{Addr}.\\ & \qquad \mathcal{S}\llbracket e \rrbracket \;\star_S\; \lambda\varphi_e : \mathsf{Dynam}(\mathit{int}).\; \mathbf{unit}_S(\varphi_e \;\star_D\; \lambda i : \mathit{int}.\texttt{store}(a, i))\end{aligned}$$

$$\begin{aligned}&\mathcal{S}\llbracket \mathbf{new}\; x \;\mathbf{in}\; c \rrbracket = \\ &\quad \texttt{rdAddr} \;\star_S\; \lambda a.\\ &\quad \texttt{inAddr}\,(a+1) \begin{pmatrix} \texttt{rdEnv} \;\star_S\; \lambda\rho. \\ \texttt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_S(a)])\; \mathcal{S}\llbracket c \rrbracket \end{pmatrix} \;\star_S\; \lambda\varphi_c.\\ &\qquad \mathbf{unit}_S(\texttt{Alloc}(a) \;\star_D\; \lambda\_.\varphi_c \;\star_D\; \lambda\_.\texttt{deAlloc}(a))\end{aligned}$$

$$\begin{aligned}\mathcal{S}\llbracket e_1 \;\mathsf{leq}\; e_2 \rrbracket = \;& \mathcal{S}\llbracket e_1 \rrbracket \;\star_S\; \lambda\varphi_1 : \mathsf{Dynam}(\mathit{int}).\\ & \mathcal{S}\llbracket e_2 \rrbracket \;\star_S\; \lambda\varphi_2 : \mathsf{Dynam}(\mathit{int}).\\ & \mathbf{unit}_S \begin{pmatrix} \varphi_1 \;\star_D\; \lambda v_1 : \mathit{int}. \\ \varphi_2 \;\star_D\; \lambda v_2 : \mathit{int}. \\ \quad \mathbf{unit}_D(\lambda\langle\kappa_T, \kappa_F\rangle.(v_1 \le v_2 \to \kappa_T, \kappa_F)) \end{pmatrix}\end{aligned}$$

$$\begin{aligned}\mathcal{S}\llbracket \mathbf{if}\; b \;\mathbf{then}\; c \rrbracket = \;& \mathcal{S}\llbracket b \rrbracket \;\star_S\; \lambda\varphi_b : \mathsf{Dynam}(\mathbf{bool}).\\ & \mathcal{S}\llbracket c \rrbracket \;\star_S\; \lambda\varphi_c : \mathsf{Dynam}(\texttt{void}).\\ & pp \quad \mathbf{unit}_S(\mathsf{IfThen}(\varphi_b, \varphi_c))\end{aligned}$$

where $\mathsf{IfThen} : \mathsf{Dynam}(\mathbf{bool}) \times \mathsf{Dynam}(\texttt{void}) \to \mathsf{Dynam}(\texttt{void})$ is defined by:

$$\mathsf{IfThen}(\varphi_b, \varphi_c) = \varphi_b \;\star_D\; \lambda\beta : \mathbf{bool}.\texttt{callcc}\; \lambda\kappa.\beta\langle \varphi_c \;\star_D\; \kappa, \kappa\,\bullet\rangle$$

$$\begin{aligned}\mathcal{S}\llbracket \mathbf{while}\; b \;\mathbf{do}\; c \rrbracket = \;& \mathcal{S}\llbracket b \rrbracket \;\star_S\; \lambda B : \mathsf{Dynam}(\mathbf{bool}).\\ & \mathcal{S}\llbracket c \rrbracket \;\star_S\; \lambda\varphi_c : \mathsf{Dynam}(\texttt{void}).\\ & \quad \mathbf{unit}_S(\mathsf{dynwhile}(B, \varphi_c))\end{aligned}$$

where:

$\mathsf{dynwhile} : \mathsf{Dynam}(\mathbf{bool}) \times \mathsf{Dynam}(\texttt{void}) \to \mathsf{Dynam}(\texttt{void})$ is defined:

$$\mathsf{dynwhile}(B, \varphi) = B \;\star_D\; \lambda\beta : \mathbf{bool}.\texttt{callcc}\; \lambda\kappa.\beta\langle \varphi \;\star_D\; \lambda\_.\mathsf{dynwhile}(B, \varphi), \kappa\,\bullet\rangle$$

Figure 4.2: Src Staged Standard Semantics

$$[\![-]\!] : \mathsf{Src} \to \mathsf{Static} + \mathsf{Dynam}(\mathtt{void} + \mathit{int} + \mathbf{bool})$$

$$[\![i]\!] = \mathbf{unit}(i)$$

$$[\![-e]\!] = [\![e]\!] \star \lambda i.\mathbf{unit}(-i)$$

$$[\![x]\!] = \mathtt{rdEnv} \star \lambda\rho.(\rho\, x) \star \lambda a : \mathit{Addr}.\ \mathtt{read}(a)$$

$$[\![c_1 \ ; c_2]\!] = [\![c_1]\!] \star \lambda\_.[\![c_2]\!]$$

$$[\![x := e]\!] = \mathtt{rdEnv} \star \lambda\rho.(\rho\, x) \star \lambda a : \mathit{Addr}.[\![e]\!] \star \lambda i : \mathit{int}.\ \mathtt{store}(a, i)$$

$$\begin{array}{rl} [\![\mathbf{new}\ x\ \mathbf{in}\ c]\!] = & \mathtt{rdAddr} \star \lambda a. \\ & \mathtt{inAddr}\,(a+1) \left(\begin{array}{l} \mathtt{rdEnv} \star \lambda\rho. \\ \mathtt{inEnv}\,(\rho[x \mapsto \mathbf{unit}(a)]) \\ \quad \mathtt{Alloc}(a) \star \lambda\_.[\![c]\!] \star \lambda\_.\mathtt{deAlloc}(a) \end{array} \right) \end{array}$$

$$[\![e_1\ \ \mathsf{leq}\ \ e_2]\!] = [\![e_1]\!] \star \lambda v_1.[\![e_2]\!] \star \lambda v_2.\mathbf{unit}(\lambda\langle\kappa_T, \kappa_F\rangle.(v_1 \leq v_2 \to \kappa_T, \kappa_F))$$

$$[\![\mathbf{if}\ b\ \mathbf{then}\ c]\!] = \mathsf{IfThen}([\![b]\!], [\![c]\!])$$

$$[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = \mathsf{dynwhile}([\![b]\!], [\![c]\!])$$

Figure 4.3: Src Standard Semantics

Intuitively, $\texttt{unquote}(x)$ runs the static and dynamic phases of x in order:

$$\texttt{unquote}(x) = \text{“first stage of } x\text{”} \ \star \ \lambda i. \text{“second stage of } x\text{”}$$

The effect of $(\texttt{unquote}\,\mu)$ is to evaluate the static part of a metacomputation, thereby producing the dynamic part, which is then evaluated. N.b., `unquote` is the same as monadic **join**.

Two useful, straightforward consequences of the definition of $\mathcal{S}[\![-]\!]$ and Axiom 6.2 are that, for any $t, t' : \mathsf{Src}$ and $\mathcal{F} : \mathsf{Static}\, a \to \mathsf{Static}\, b \to \mathsf{Static}\, c$,

$$\begin{aligned} \mathcal{S}[\![t]\!] \ \star \ \lambda\varphi_t.\mathcal{S}[\![t']\!] \ \star \ \lambda\varphi_{t'}.(\mathcal{F}\,\varphi_t\,\varphi_{t'}) &= \mathcal{S}[\![t']\!] \ \star \ \lambda\varphi_{t'}.\mathcal{S}[\![t]\!] \ \star \ \lambda\varphi_t.(\mathcal{F}\,\varphi_t\,\varphi_{t'}) && (4.6) \\ \mathcal{S}[\![t]\!] \ \star \ \lambda\varphi_t.\mathcal{S}[\![t]\!] \ \star \ \lambda\widehat{\varphi}_t.(\mathcal{F}\,\varphi_t\,\widehat{\varphi}_t) &= \mathcal{S}[\![t]\!] \ \star \ \lambda\varphi_t.(\mathcal{F}\,\varphi_t\,\varphi_t) && (4.7) \end{aligned}$$

Intuitively, Axiom 6.2 guarantees that $\mathcal{S}[\![t]\!]$ and $\mathcal{S}[\![t']\!]$ receive the same *Env* and *Addr* environments as “input.” This, combined with the fact that $\mathcal{S}[\![t]\!]$ and $\mathcal{S}[\![t']\!]$ are innocent, implies that they commute.

The relationship between the standard semantics $[\![-]\!]$ and the staged standard semantics $\mathcal{S}[\![-]\!]$ is given in Theorem 2. This theorem states intuitively that running the static and dynamic parts of $\mathcal{S}[\![t]\!]$ as distinct phases is equivalent to running $[\![t]\!]$ without distinguishing static from dynamic. Note that this theorem is proved in the monad $\mathsf{M} = \mathsf{Static} + \mathsf{Dynam}$, which is created by applying all of the monad transformers from Static and Dynam.

Theorem 2 (Equivalence of Staged Standard Semantics) *In the monad* $\mathsf{M} = \mathsf{Static}{+}\mathsf{Dynam}$,

$$[\![t]\!] = \texttt{unquote}\ \mathcal{S}[\![t]\!]$$

Theorem 2 is proved in Appendix B beginning on page 137.

4.6 Various Lemmas

There are a number of lemmas used in the correctness proof of the source language $Exp + Imp + ControlFlow + Boolean + Block$ (specifically in the proofs of Theorems 3, 4, 5, 6, 7, 8, and 9 later in this chapter). They are gathered into this section and their proofs are found in Appendix B. On first reading, it may be helpful to skip this section.

The following lemma follows directly from Axiom 4.

Lemma 2 ($\texttt{Alloc}/\texttt{deAlloc}$ commutes) *It expresses how* $\texttt{Alloc}(a)$ *and* $\texttt{deAlloc}(a)$ *commute with other combinators. The lemma is stated for* $\texttt{Alloc}$ *only, although the following are true for* $\texttt{deAlloc}$ *as well.*

1. $\mathbf{unit}(x) \star \lambda v.\texttt{Alloc}(a) \star \lambda\_.(f\,v) = \texttt{Alloc}(a) \star \lambda\_.\mathbf{unit}(x) \star f$

2. $\texttt{rdEnv} \star \lambda\rho.\texttt{Alloc}(a) \star \lambda\_.(f\,\rho) = \texttt{Alloc}(a) \star \lambda\_.\texttt{rdEnv} \star f$

3. $x \star \lambda v.\texttt{Alloc}(a) \star \lambda\_.(f\,v) = \texttt{Alloc}(a) \star \lambda\_.x \star f \implies$
$$(\texttt{inEnv}\,\rho\,x) \star \lambda v.\texttt{Alloc}(a) \star \lambda\_.(f\,v) = \texttt{Alloc}(a) \star \lambda\_.(\texttt{inEnv}\,\rho\,x) \star f$$

4. $\texttt{rdAddr} \star \lambda\alpha.\texttt{Alloc}(a) \star \lambda\_.(f\,\alpha) = \texttt{Alloc}(a) \star \lambda\_.\texttt{rdAddr} \star f$

5. $x \star \lambda v.\texttt{Alloc}(a) \star \lambda\_.(f\,v) = \texttt{Alloc}(a) \star \lambda\_.x \star f \implies$
$$(\texttt{inAddr}\,\alpha\,x) \star \lambda v.\texttt{Alloc}(a) \star \lambda\_.(f\,v) = \texttt{Alloc}(a) \star \lambda\_.(\texttt{inAddr}\,\alpha\,x) \star f$$

The following lemma states that a fresh address may be allocated, stored to, and discarded without effect.

Lemma 3 *For* $\mathcal{F} : int \to \mathsf{M}\,b$*,*

$$\begin{aligned}&\mathsf{Obs}(\mathsf{FreshLoc}(a), \texttt{Alloc}(a) \star \lambda\_.\texttt{Thread}(i,a) \star \lambda v.\texttt{deAlloc}(a) \star \lambda\_.\mathcal{F}\,v, \gamma)\\ &\quad = \mathsf{Obs}(\mathsf{FreshLoc}(a), \mathcal{F}\,i, \gamma)\end{aligned}$$

Lemma 3 is proved in Appendix B beginning on page 147.

Lemma 4 states that, if a is a fresh location in the store, then $a+1$ will be fresh after a is allocated.

Lemma 4 (Discharging FreshLoc After Alloc) $\mathsf{FreshLoc}(a)$ *discharges* $\mathsf{FreshLoc}(a+1)$ *after* $\mathtt{Alloc}(a)$*:*

$$\begin{aligned}&\mathsf{Obs}(\mathsf{FreshLoc}(a), \mathtt{Alloc}(a) \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLoc}(a+1), x, y) \star_D \lambda v.z, \gamma)\\ &\qquad = \mathsf{Obs}(\mathsf{FreshLoc}(a), \mathtt{Alloc}(a) \star_D \lambda\_.x \star_D \lambda v.z, \gamma)\end{aligned}$$

Lemma 4 is proved in Appendix B beginning on page 152.

Lemma 5 (FreshLoc and updateCode Interactions)

$$\begin{aligned}&\mathsf{Obs}(\mathsf{FreshLoc}(a), \mathtt{updateCode}(f) \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLoc}(a), x, y) \star_D \lambda v.z, \gamma)\\ &\qquad = \mathsf{Obs}(\mathsf{FreshLoc}(a), \mathtt{updateCode}(f) \star_D \lambda\_.x \star_D \lambda v.z, \gamma)\end{aligned}$$

Lemma 5 is proved in Appendix B beginning on page 156.

Lemma 6 (FreshLabel and updateCode Interactions) *If* $L < L'$*, then*

1. $\mathsf{Obs}(\mathsf{FreshLabel}(L'), \mathtt{updateCode}[L \mapsto \pi] \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y), \gamma)$

$$= \mathsf{Obs}(\mathsf{FreshLabel}(L'), \mathtt{updateCode}[L \mapsto \pi] \star_D \lambda\_.x, \gamma)$$

2. $\mathsf{Obs}(\mathsf{FreshLabel}(L), \mathtt{updateCode}[L \mapsto \pi] \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L+1), x, y), \gamma)$

$$= \mathsf{Obs}(\mathsf{FreshLabel}(L), \mathtt{updateCode}[L \mapsto \pi] \star_D \lambda\_.x, \gamma)$$

Lemma 7 *Observations commute with* `callcc` *:*

$$\mathtt{callcc}\ \lambda\kappa.\mathsf{Obs}(\theta, \mathcal{F}\kappa, \mathcal{F}'\kappa) = \mathsf{Obs}(\theta, (\mathtt{callcc}\ \lambda\kappa.\mathcal{F}\kappa), (\mathtt{callcc}\ \lambda\kappa.\mathcal{F}'\kappa))$$

$\boxed{\text{Proof of Lemma 7:}}$

$$\begin{aligned}
&\mathtt{callcc}\ \lambda\kappa.\mathsf{Obs}(\theta, \mathcal{F}\kappa, \mathcal{F}'\kappa) \\
&= \mathtt{callcc}\ \lambda\kappa.\theta \star \lambda \mathit{test}.\ \text{if } \mathit{test} \text{ then } \mathcal{F}\kappa \text{ else } \mathcal{F}'\kappa \\
&= \theta \star \lambda \mathit{test}.\mathtt{callcc}\ \lambda\kappa.\ \text{if } \mathit{test} \text{ then } \mathcal{F}\kappa \text{ else } \mathcal{F}'\kappa \qquad \text{(Axiom 3.1)} \\
&= \theta \star \lambda \mathit{test}.\ \text{if } \mathit{test} \text{ then } (\mathtt{callcc}\ \lambda\kappa.\mathcal{F}\kappa) \text{ else } (\mathtt{callcc}\ \lambda\kappa.\mathcal{F}'\kappa) \\
&= \mathsf{Obs}(\theta, (\mathtt{callcc}\ \lambda\kappa.\mathcal{F}\kappa), (\mathtt{callcc}\ \lambda\kappa.\mathcal{F}'\kappa))
\end{aligned}$$

□ Lemma 7

Lemma 8 shows that the domain of the value store (i.e., the store shape [41]) is not changed by the dynamic part of $\mathcal{C}[\![c : \mathbf{comm}]\!]$.

Lemma 8 (Commands Preserve Store Shape) *Commands preserve store shape:*

$$\begin{array}{l}
\mathcal{S}[\![c : \mathbf{comm}]\!] \star_S \lambda\delta_c. \\
\mathbf{unit}_S \left(\begin{array}{l}
\mathsf{getSto} \star_D \lambda\sigma_0. \\
\delta_c \star_D \lambda\_. \\
\mathsf{getSto} \star_D \lambda\sigma_1. \\
\quad \mathbf{unit}_D(\mathsf{dom}(\sigma_0) = \mathsf{dom}(\sigma_1))
\end{array} \right)
\end{array}
= \begin{array}{l}
\mathcal{S}[\![c : \mathbf{comm}]\!] \star_S \lambda\delta_c. \\
\quad \mathbf{unit}_S(\delta_c \star_D \lambda\_.\mathbf{unit}_D(\mathtt{true}))
\end{array}$$

$$\begin{array}{l}
\mathcal{C}[\![c : \mathbf{comm}]\!] \star_S \lambda\varphi_c. \\
\mathbf{unit}_S \left(\begin{array}{l}
\mathsf{getSto} \star_D \lambda\sigma_0. \\
\varphi_c \star_D \lambda\_. \\
\mathsf{getSto} \star_D \lambda\sigma_1. \\
\quad \mathbf{unit}_D(\mathsf{dom}(\sigma_0) = \mathsf{dom}(\sigma_1))
\end{array} \right)
\end{array}
= \begin{array}{l}
\mathcal{C}[\![c : \mathbf{comm}]\!] \star_S \lambda\varphi_c. \\
\quad \mathbf{unit}_S(\varphi_c \star_D \lambda\_.\mathbf{unit}_D(\mathtt{true}))
\end{array}$$

Lemma 8 is proved in Appendix B beginning on page 167.

Lemma 9 specifies how the compilation φ_t affects the part of the code store within the range of the allocated labels.

Lemma 9 (Discharging Label Freshness)

$$
\begin{array}{l}
\texttt{getLabel} \star_S \lambda L. \\
\mathcal{C}[\![t]\!] \star_S \lambda\varphi_t. \\
\texttt{getLabel} \star_S \lambda L'. \\
\quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l} \varphi_t \star_D \lambda v. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array}\right), z))
\end{array}
$$

$$
\begin{array}{rl}
= & \texttt{getLabel} \star_S \lambda L. \\
 & \mathcal{C}[\![t]\!] \star_S \lambda\varphi_t. \\
 & \texttt{getLabel} \star_S \lambda L'. \\
 & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L), \varphi_t \star_D \lambda v.x, z))
\end{array}
$$

Proof of Lemma 9 is included in Appendix B beginning on page 171.

Lemma 10 specifies the least labels, L and L', that can satisfy FreshLabel before and after the compilation of t, φ_t. Most Src phrase types (e.g., *Exp*) have no affect on the code store, and proving this lemma for those cases is typically elementary. It follows from Lemma 9.

Lemma 10 (Compilations Preserve Label Freshness)

$$
\begin{aligned}
&\mathcal{C}[\![t]\!] \star_S \lambda\varphi_t.\\
&\texttt{getLabel} \star_S \lambda L'.\\
&\quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L'), \varphi_t \star_D \lambda v.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y), -)\\
&\qquad = \mathcal{C}[\![t]\!] \star_S \lambda\varphi_t.\\
&\qquad\quad \texttt{getLabel} \star_S \lambda L'.\\
&\qquad\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L'), \varphi_t \star_D \lambda v.x, -)
\end{aligned}
$$

Lemma 10 is proved in Appendix B beginning on page 191.

4.7 Observational Correctness of Src

This section presents the compiler correctness specification relating the standard staged semantics for Src, $\mathcal{S}[\![-]\!]$, to a compiler for the source language constructed from reusable compiler building blocks for each sublanguage. This relationship is entitled *observational correctness* in part to distinguish it from the issue of overall modular compiler correctness. Individual RCBBs are given observation-style specifications characterizing their correctness independently of the other sublanguages. It is demonstrated that each building block in the compilation semantics with explicit allocation from Section 4.4 satisfies these specifications. All of the proofs from this section are included in Appendix B. This constitutes the middle link in the "road map" of modular compiler verification case study outlined in Figure 4.1.

Figures 4.4, 4.5, 4.7, 4.8, and 4.9 contain the reusable compiler building blocks for the source language Src. These differ from the RCBBs developed in Chapter 2 in exactly two respects. Firstly, we assume now that the value store *Sto* has the representation described in Section 4.4, because the scoping of side-effects provided by that representation makes reasoning about RCBB correctness simpler. Secondly, we remove addition from the *Exp* RCBB to simplify the presentation here, but the correctness of addition is handled analogously to negation. The semantic definitions for

Src are summarized in Appendix A, and in particular, the official compilation semantics for Src is summarized in Appendix A.3 on page 132.

Also included here is a new RCBB $\mathsf{CF}[\![e : \mathit{Exp}]\!] : \mathsf{Static}(\mathsf{Dynam}(\mathit{int}))$ for expressions which performs constant-folding[1, 2], which is a code optimization that evaluates constant expressions (i.e., those without variables) at compile-time. $\mathsf{CF}[\![-]\!]$ is shown to fulfill the same specification as the *Exp* block (called **Exp-spec**). Because the proofs of the specifications for *Imp*, *ControlFlow*, *Boolean*, and *Block* only assume **Exp-spec**, we get immediately that the compiler using the constant-folding $\mathsf{CF}[\![-]\!]$ instead of the *Exp* block defined in Chapter 2 is also correct. It will be seen that the structuring of both compilers for Src affords some modularity in the compiler correctness proofs.

4.7.1 Monad Expressions

A notational convention for monads constructed from monad transformers allows more general RCBB specifications to be expressed easily and clearly. A monad $\mathsf{M} = \mathcal{T}_1\, a_1\, (\ldots \mathcal{T}_n\, a_n\, \mathsf{Id})$ may be abbreviated as a sum of its component parts (called hereafter a *monad expression*): $a_1 + \ldots + a_n$. This convention applies when

1. a_i is always used in this thesis as part of a particular monad transformer (e.g., *Label* has always appeared with $\mathcal{T}_{\mathsf{St}}$)

2. liftings exist to combine the $\mathcal{T}_i\, a_i$ so that the axioms in Section 4.3 hold.

To avoid confusion, the above a_i will be annotated with a subscript denoting the kind of monad transformer with which it is associated, so for example, $\mathit{Addr}_{\mathsf{Env}} + \mathit{Label}_{\mathsf{St}}$ stands for both:

$$\mathcal{T}_{\mathsf{Env}}\, \mathit{Addr}\, (\mathcal{T}_{\mathsf{St}}\, \mathit{Label}\, \mathsf{Id}) \text{ and } \mathcal{T}_{\mathsf{St}}\, \mathit{Label}\, (\mathcal{T}_{\mathsf{Env}}\, \mathit{Addr}\, \mathsf{Id})$$

because *Addr* and *Label* have always appeared heretofore with $\mathcal{T}_{\mathsf{Env}}$ and $\mathcal{T}_{\mathsf{St}}$, respectively. Because well-known liftings exist such that `updateLabel`, `getLabel`, `rdAddr`, and `inAddr` behave as the relevant axioms from Section 4.3 dictate (see Section 1.2.4 and Liang, et al.,[25] for further details), the order of monad transformer application in this example is irrelevant. In those cases where the order of monad transformer applications does matter, $a_1 + \ldots + a_n$ stands for only those monads for which appropriate liftings exist. So in the case of $\mathcal{T}_{\mathsf{Env}}\, a$ and $\mathcal{T}_{\mathsf{CPS}}\, \texttt{void}$ for example, $a + CPS$ stands for only $\mathcal{T}_{\mathsf{Env}}\, a\, (\mathcal{T}_{\mathsf{CPS}}\, \texttt{void}\, \mathsf{Id})$ and not $\mathcal{T}_{\mathsf{CPS}}\, \texttt{void}\, (\mathcal{T}_{\mathsf{Env}}\, a\, \mathsf{Id})$.

Furthermore, monad expressions can be ordered such that:

$$a_1 + \ldots + a_n \preceq a'_1 + \ldots + a'_m \Leftrightarrow \{a_1, \ldots, a_n\} \subseteq \{a'_1, \ldots, a'_m\}$$

We say that a computation φ is *in the monad* M, if, and only if, $\mathsf{M} = a_1 + \ldots + a_n$ is the least monad according to $\preceq$ such that φ is written in terms of the combinators defined by the monad transformers associated with $a_1, \ldots, a_n$. For example, the computation **rdAddr** is in the monad $\mathsf{M} = \mathcal{T}_{\mathsf{Env}}\, \mathit{Addr}\, \mathsf{Id} = \mathit{Addr}_{\mathsf{Env}}$, but not in the monad $\mathcal{T}_{\mathsf{Env}}\, \mathit{Addr}\, (\mathcal{T}_{\mathsf{St}}\, \mathit{Label}\, \mathsf{Id}) = \mathit{Addr}_{\mathsf{Env}} + \mathit{Label}_{\mathsf{St}}$. A metacomputation μ is *in the monads* S *and* D if its static part is in S and its dynamic part is in D.

We say that a monadic language specification $\{\mathsf{Mng}[\![t_1]\!] = e_1, \ldots, \mathsf{Mng}[\![t_n]\!] = e_n\}$ is *in the monad* M if, and only if, M is the least upper bound according to $\preceq$ of the monads M_i, where e_i is in M_i. This generalizes to metacomputation-based language specifications similarly.

4.7.2 Expressions

The correctness specification for expressions will compare the integer value computed ultimately by the staged standard semantics $\mathcal{S}[\![e]\!] : \mathsf{Static}(\mathsf{Dynam}(\mathit{int}))$ with the value produced by $\mathcal{C}[\![e]\!] : \mathsf{Static}(\mathsf{Dynam}(\mathit{int}))$. Correctness for the expressions block means that these computations should, under certain conditions, produce the same integer. Yet, it is too strong to simply require that $\mathcal{C}[\![e]\!] = \mathcal{S}[\![e]\!]$ since the dynamic computation produced by $\mathcal{C}[\![e]\!]$ (the *compilation* φ_e) involves allocating and deallocating memory cells and reading and writing from the store and is, therefore, a different kind of computation from the dynamic computation produced by $\mathcal{S}[\![e]\!]$ (i.e., the *interpretation* δ_e). For example, φ_e may fail (i.e., return $\bot_{\mathsf{Dynam}(\texttt{void})}$) if it is executed in a store for which it was not compiled. Consider if φ_e were produced by $(\texttt{inAddr}\ 0\ \mathcal{C}[\![e]\!])$ but is executed in the store $\{\langle 0, 99\rangle\}$. φ_e may allocate the memory cell 0, which would cause it to fail. In contrast, δ_e will always produce the same integer, no matter what store it is executed in, because it does not depend on the store. Therefore, care must be taken to ensure that φ_e and δ_e are only compared when φ_e in executed in an "appropriate" store. Observation-based specification allows us to do this easily.

So, what is an "appropriate" store for φ_e? If e is compiled when a is the first free address, then φ_e may use as temporaries any address $a' \geq a$, and so φ_e should execute correctly in a store σ where none of φ_e's temporaries are allocated. That is, if $\forall a' \geq a. \langle a', - \rangle \notin \sigma$, then σ is an appropriate store to execute φ_e. This requirement was encapsulated as the observation $\mathsf{FreshLoc}(a)$ in Definition 20. The correctness requirement for expressions will require that, if φ_e is executed in an appropriate store, then φ_e will be *identical* to δ_e. Why identical? Because φ_e takes pains to deallocate every memory cell that it allocates, and so it should return the same store in which it was executed. This specification can be written in observation-style as:

$$\varphi_e = \mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_e, \varphi_e) \tag{4.8}$$

But this is still incomplete as it has free variables (namely, φ_e, δ_e, and a).

To achieve the general form of the correctness specification, first observe that:

$$\begin{aligned} \mathcal{C}[\![e]\!] = \;& \texttt{rdAddr} \star_S \lambda a. \\ & \mathcal{S}[\![e]\!] \star_S \lambda \delta_e. \\ & \mathcal{C}[\![e]\!] \star_S \lambda \varphi_e. \\ & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi_e, \varphi_e)) \end{aligned} \tag{4.9}$$

by the "innocence" of $\texttt{rdAddr}$ and $\mathcal{S}[\![e]\!]$, observation introduction, and the right unit law. What Equation 4.8 requires is that:

$$\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi_e, \varphi_e) = \mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_e, \varphi_e)$$

or, in other words, that the right-hand side of Equation 4.9 be equal to:

$$\begin{aligned}&\texttt{rdAddr} \star_S \lambda a.\\&\mathcal{S}[\![e]\!] \star_S \lambda\delta_e.\\&\mathcal{C}[\![e]\!] \star_S \lambda\varphi_e.\\&\quad\mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_e, \varphi_e))\end{aligned} \tag{4.10}$$

Hence, the correct requirement for expressions is that $\mathcal{C}[\![e]\!]$ be equal to (4.10). This correctness statement, **Exp-spec**, is summarized in Specification 1.

Specification 1 (Exp-spec) *Let $\mathcal{E}[\![-]\!] : Exp \to \mathsf{S}(\mathsf{D}(int))$ where $Addr \preceq \mathsf{S}$ and $Sto \preceq \mathsf{D}$. $\mathcal{E}[\![-]\!]$ is a correct RCBB for expressions, if*

$$\begin{aligned}\mathcal{E}[\![e]\!] = \;&\texttt{rdAddr} \star_S \lambda a.\\&\mathcal{S}[\![e]\!] \star_S \lambda\delta_e.\\&\mathcal{E}[\![e]\!] \star_S \lambda\varphi_e.\\&\qquad\mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_e, \varphi_e))\end{aligned}$$

Theorem 3 *$\mathcal{C}[\![- : Exp]\!]$ satisfies* **Exp-spec***.*

Proof of Theorem 3 is included in Appendix B beginning on page 198.

4.7.3 Imperative

The assignment lemma is used in the proof of the the imperative block specification 2.

Lemma 11 (Assignment Lemma) *Under appropriate conditions, the compilation produced for an assignment, φ, is identical to the dynamic part of $\mathcal{S}[\![x{:=}e]\!]$:*

$$Sto \preceq \mathsf{Dynam} \quad Addr \preceq \mathsf{Static}$$

$$\mathcal{C}[\![i]\!] = \mathbf{unit}_S(\mathbf{unit}_D(i))$$

$$\begin{array}{l}
\mathcal{C}[\![-t]\!] : \mathsf{Static}(\mathsf{Dynam}(int)) = \\
\quad\quad \texttt{rdAddr} \star_S \ \lambda a. \\
\quad\quad \texttt{inAddr}\ (a+1) \\
\quad\quad\quad\quad (\mathcal{C}[\![t]\!] \star_S \ \lambda \varphi_t : \mathsf{Dynam}(int). \\
\quad\quad\quad\quad\quad \mathbf{unit}_S(\mathsf{Negate}(\varphi_t, a))
\end{array}
\qquad
\begin{array}{rl}
\mathsf{Negate}(\varphi_t, a) = & \varphi_t \star_D \ \lambda i. \\
& \texttt{Alloc}(a) \star_D \ \lambda \_. \\
& \texttt{Thread}(i, a) \star_D \ \lambda v. \\
& \texttt{deAlloc}(a) \star_D \ \lambda \_. \\
& \quad\quad \mathbf{unit}_D(-v)
\end{array}$$

Figure 4.4: Compilation Semantics for the Expression Block of Src with explicit (de)allocation

$$\begin{array}{rl}
\mathcal{C}[\![x{:=}e]\!] = & \texttt{rdAddr} \star_S \ \lambda a. \\
& \mathcal{S}[\![x{:=}e]\!] \star_S \ \lambda \delta. \\
& \mathcal{C}[\![x{:=}e]\!] \star_S \ \lambda \varphi. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta, \varphi))
\end{array}$$

Lemma 11 is proved in Appendix B on page 195.

Imp-spec states that, under the right circumstances, the compilation produced for an *Imp* term c behaves just like the dynamic part of $\mathcal{S}[\![c]\!]$, if the code store is ignored. The right circumstances here are when both $\mathsf{FreshLoc}(a)$ and $\mathsf{FreshLabel}(L)$ produce `true`. Ignoring the code store is accomplished by initializing it at the end of the dynamic phase with $\texttt{init}_{CS}$.

Specification 2 (Imp-spec) *Let* $Env + Addr + Label \preceq \mathsf{S}$, $CPS + Sto + Code \preceq \mathsf{D}$, *and* $\mathcal{I}[\![-]\!]$:

$$Env \preceq \mathsf{Static} \quad Sto \preceq \mathsf{Dynam}$$

$$\begin{array}{l}
\mathcal{C}[\![c_1;c_2]\!] : \mathsf{Static}(\mathsf{Dynam}\,\mathtt{void}) = \\
\qquad \mathcal{C}[\![c_1]\!] \star_S \ \lambda\varphi_{c_1}. \\
\qquad \mathcal{C}[\![c_2]\!] \star_S \ \lambda\varphi_{c_2}. \\
\qquad\quad \mathbf{unit}_S(\varphi_{c_1} \star_D \ \lambda\_.\varphi_{c_2})
\end{array}
\qquad
\begin{array}{l}
\mathcal{C}[\![x := e]\!] : \mathsf{Static}(\mathsf{Dynam}\,\mathtt{void}) = \\
\qquad \mathtt{rdEnv} \star_S \ \lambda\rho. \\
\qquad (\rho\, x) \star_S \ \lambda a. \\
\qquad\quad \mathcal{C}[\![e]\!] \star_S \ \lambda\varphi_e. \\
\qquad\qquad \mathbf{unit}_S(\varphi_e \star_D \ \lambda i : int.\mathtt{store}(a, i))
\end{array}$$

Figure 4.5: Compilation Semantics for Imperative Features

$Imp \to \mathsf{S}(\mathsf{D}(\mathtt{void}))$*, then* $\mathcal{I}[\![-]\!]$ *is a correct RCBB for imperative features, if*

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathcal{I}[\![c]\!] \star_S \ \lambda\varphi_c.\mathbf{unit}_S(\varphi_c \star_D \ \lambda\_.\mathtt{init}_{CS})} \\
= & \mathtt{rdAddr} \star_S \ \lambda a. \\
 & \mathtt{getLabel} \star_S \ \lambda L. \\
 & \mathcal{S}[\![c]\!] \star_S \ \lambda\delta_c. \\
 & \mathcal{I}[\![c]\!] \star_S \ \lambda\varphi_c. \\
 & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a)\ \mathsf{AND}\ \mathsf{FreshLabel}(L), \delta_c, \varphi_c) \star_D \ \lambda\_.\mathtt{init}_{CS})
\end{array}$$

Specification 3 (Linking Condition for Exp+Imp) Imp-spec *assumes that the Exp compiler building block it uses satisfies* **Exp-spec**.

Theorem 4 $\mathcal{C}[\![- : Imp]\!]$ *satisfies* **Imp-spec**.

Theorem 4 is proved in Appendix B beginning on page 204.

4.7.4 The Constant Folding Expression Block

In this section, a new RCBB $\mathsf{CF}[\![-]\!]$ for expressions which performs constant-folding is introduced. Constant-folding is a standard code optimization[1, 2], which evaluates constant expressions at compile-time. Figure 4.6 has the constant-folding compiler block for expressions. What $\mathsf{CF}[\![-]\!]$ does is simple: if e contains no variables, then it can be evaluated at compile-time. To evaluate e at compile-time, simply apply the the standard semantics for constant expressions (i.e., the usual

$$Addr \preceq \mathsf{Static} \quad Sto \preceq \mathsf{Dynam}$$

$$\begin{array}{llll}
\mathsf{constexp}(-) : Exp + \mathcal{V}ar \to \{\texttt{true}, \texttt{false}\} & & & \\
\mathsf{constexp}(e) = & \text{case } e \text{ of} & i & \Longrightarrow \texttt{true} \\
 & & e_1 + e2 & \Longrightarrow \mathsf{constexp}(e_1) \,\&\, \mathsf{constexp}(e_2) \\
 & & -e' & \Longrightarrow \mathsf{constexp}(e') \\
 & & x & \Longrightarrow \texttt{false}
\end{array}$$

$$\begin{array}{lll}
\mathsf{CF}[\![-e]\!] = & \text{case } \mathsf{constexp}(e) \text{ of} & \\
 & \texttt{true} \Longrightarrow & [\![-e]\!] \star_S \lambda i.\mathbf{unit}_S(\mathbf{unit}_D(i)) \\
 & \texttt{false} \Longrightarrow & \texttt{rdAddr} \star_S \lambda a.\texttt{inAddr}\,(a+1) \\
 & & \quad (\mathsf{CF}[\![e]\!] \star_S \lambda\varphi_e.\mathbf{unit}_S(\mathsf{Negate}(\varphi_e, a))
\end{array}$$

Figure 4.6: Constant-Folding Compilation Semantics for *Exp*

monadic semantics for expressions: $[\![-e]\!] = [\![e]\!] \star \lambda i.\mathbf{unit}(-i))$ in the Static monad and boost the resulting integer v to the dynamic phase with $\mathbf{unit}_S(\mathbf{unit}_D(v))$. If e is not constant, then do the same thing you would have in the RCBB in Figure 4.4.

The following lemma states that "boosting" the standard semantics for a constant expression gives the same result as the standard staged semantics. Let $\mathsf{boost} : \mathsf{Static}(\tau) \to \mathsf{Static}(\mathsf{Dynam}(\tau))$ be defined as:

$$\mathsf{boost}(x : \mathsf{Static}(\tau)) = x \star_S \lambda v : \tau.\ \mathbf{unit}_S(\mathbf{unit}_D(v))$$

The boost operation is similar to the `lift` operation associated with a monad transformer, but it differs in that boost is metacomputation-valued. The name boost has been chosen to avoid confusion with `lift`.

Lemma 12 (boost-lemma) *For any constant integer expression e:*

$$\mathsf{boost}\,[\![e]\!] = \mathcal{S}[\![e]\!]$$

Proof of **boost-lemma** is included in the Appendix B beginning on page 213.

$$Env + Addr \preceq \mathsf{Static} \quad Sto \preceq \mathsf{Dynam}$$

$$\begin{array}{ll}
\mathcal{C}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!] = & \mathtt{rdEnv}\ \star_S\ \lambda\rho. \\
& \mathtt{rdAddr}\ \star_S\ \lambda a. \\
& \quad [\mathtt{inEnv}\ (\rho[x \mapsto \mathbf{unit}_S(a)])\ (\mathtt{inAddr}\ (a+1)\ \mathcal{C}[\![c]\!])]\ \star_S\ \lambda\varphi_c : \mathsf{Dynam}(\mathtt{void}). \\
& \qquad \mathbf{unit}_S(\mathtt{Alloc}(a)\ \star_D\ \lambda\_.\varphi_c\ \star_D\ \lambda\_.\mathtt{deAlloc}(a))
\end{array}$$

$$\mathcal{C}[\![x_{rval}]\!] = \mathtt{rdEnv}\ \star_S\ \lambda\rho.(\rho\, x)\ \star_S\ \lambda a.\ \mathbf{unit}_S(\mathtt{read}(a))$$

Figure 4.7: Compilation Semantics for Block Structure

Theorem 5 $\mathsf{CF}[\![-]\!]$ *satisfies* **Exp-spec**.

4.7.5 Block Structure

As in **Imp-spec**, masking out the code store is accomplished by initializing it at the end of the dynamic phase with $\mathtt{init}_{CS}$.

Specification 4 (Block-spec) *Let* $Env + Addr + Label \preceq \mathsf{S}$, $CPS + Sto + Code \preceq \mathsf{D}$, *and* $\mathcal{N}[\![-]\!] : Block \to \mathsf{S}(\mathsf{D}(\mathtt{void}))$, *then* $\mathcal{N}[\![-]\!]$ *is a correct RCBB for block structure, if*

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{N}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!]\ \star_S\ \lambda\varphi.\mathbf{unit}_S(\varphi\ \star_D\ \lambda\_.\mathtt{init}_{CS})} \\
= & \mathtt{rdAddr}\ \star_S\ \lambda a. \\
& \mathtt{getLabel}\ \star_S\ \lambda L. \\
& \mathcal{S}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!]\ \star_S\ \lambda\delta. \\
& \mathcal{N}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!]\ \star_S\ \lambda\varphi. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a)\ \mathsf{AND}\ \mathsf{FreshLabel}(L), \delta, \varphi)\ \star_D\ \lambda\_.\mathtt{init}_{CS})
\end{array}$$

Theorem 6 $\mathcal{C}[\![- : Block]\!]$ *satisfies* **Block-spec**.

Theorem 6 is proved in Appendix B beginning on page 214.

$$Addr + Label \preceq \mathsf{Static} \quad Code + Sto \preceq \mathsf{Dynam}$$

$$\begin{array}{l}
\mathcal{C}[\![e_1 \leq e_2]\!] : \mathsf{Static}(\mathsf{Dynam}(\mathbf{bool})) = \\
\quad \texttt{rdAddr} \star_S \ \lambda a. \\
\quad \texttt{inAddr}\ (a+2) \\
\qquad (\mathcal{C}[\![e_1]\!] \ \star_S \ \lambda \varphi_1 : \mathsf{Dynam}(int). \\
\qquad (\mathcal{C}[\![e_2]\!] \ \star_S \ \lambda \varphi_2 : \mathsf{Dynam}(int). \\
\qquad\quad \mathbf{unit}_S(\mathsf{Lteq}(\varphi_1, \varphi_2, a))
\end{array}$$

$$\begin{array}{l}
\mathsf{Lteq}(\varphi_1, \varphi_2, a) = \\
\quad \varphi_1 \ \star_D \ \lambda i. \\
\quad \varphi_2 \ \star_D \ \lambda j. \\
\quad \texttt{Alloc}(a) \ \star_D \ \lambda\_. \\
\quad \texttt{Alloc}(a+1) \ \star_D \ \lambda\_. \\
\quad \texttt{Thread}(i, a) \ \star_D \ \lambda v_1. \\
\quad \texttt{Thread}(i, a+1) \ \star_D \ \lambda v_2. \\
\quad \texttt{deAlloc}(a) \ \star_D \ \lambda\_. \\
\quad \texttt{deAlloc}(a+1) \ \star_D \ \lambda\_. \\
\qquad \mathbf{unit}_D(\lambda\langle\kappa_T, \kappa_F\rangle.(v_1 \leq v_2 \ \rightarrow \ \kappa_T, \kappa_F))
\end{array}$$

$$\begin{array}{l}
\mathbf{if}\ (b_1\, \mathbf{or}\, b_2)\ \mathbf{then}\ c \ \equiv_{\text{syn. sugar}} \ \mathbf{if}\ b_1\ \mathbf{then}\ c\ \mathbf{else}\ (\mathbf{if}\ b_2\ \mathbf{then}\ c) \\
\mathbf{if}\ (b_1\, \mathbf{or}\, b_2)\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \ \equiv_{\text{syn. sugar}} \ \mathbf{if}\ b_1\ \mathbf{then}\ c_1\ \mathbf{else}\ (\mathbf{if}\ b_2\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2)
\end{array}$$

Figure 4.8: Compilation Semantics for the Boolean Block of Src

4.7.6 Booleans

Specification 5 (Bool-spec) *Let* $Addr + Label \preceq \mathsf{S}$*,* $CPS + Sto + Code \preceq \mathsf{D}$*, and* $\mathcal{B}[\![-]\!] : Bool \rightarrow \mathsf{S}(\mathsf{D}(\texttt{void}))$*, then* $\mathcal{B}[\![-]\!]$ *is a correct RCBB for booleans, if*

$$\begin{array}{rl}
\mathcal{B}[\![b]\!] = & \texttt{rdAddr} \ \star_S \ \lambda a. \\
& \texttt{getLabel} \ \star_S \ \lambda L. \\
& \mathcal{S}[\![b]\!] \ \star_S \ \lambda \delta_b. \\
& \mathcal{B}[\![b]\!] \ \star_S \ \lambda \varphi_b. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a)\ \mathsf{AND}\ \mathsf{FreshLabel}(L), \delta_b, \varphi_b)
\end{array}$$

Theorem 7 $\mathcal{C}[\![- : Bool]\!]$ *satisfies* **Bool-spec**.

Theorem 7 is proved in Appendix B beginning on page 219.

$$\mathit{Label} \preceq \mathsf{Static} \quad \mathit{Code} + \mathit{CPS} \preceq \mathsf{Dynam}$$

```
$\mathcal{C}[\![\textbf{if } b \textbf{ then } c]\!] =$
    newlabel $\star_S$ $\lambda L_\kappa$.
    newlabel $\star_S$ $\lambda L_c$.
    $\mathcal{C}[\![b]\!]$ $\star_S$ $\lambda\varphi_b$.
    $\mathcal{C}[\![c]\!]$ $\star_S$ $\lambda\varphi_c$.
        $\textbf{unit}_S(\mathsf{IfThenPS}(\varphi_b, \varphi_c, L_c, L_\kappa))$
```

```
$\mathsf{IfThenPS}(\varphi_b, \varphi_c, L_c, L_\kappa) =$
    $\varphi_b$ $\star_D$ $\lambda\beta$.
    callcc $\lambda\kappa$.
        updateCode$[L_\kappa \mapsto \kappa\bullet]$ $\star_D$ $\lambda\_$.
        updateCode$[L_c \mapsto \varphi_c$ $\star_D$ $\lambda\_$.jump $L_\kappa]$ $\star_D$ $\lambda\_$.
            $\beta\langle$jump $L_c$, jump $L_\kappa\rangle$
```

```
$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] =$
  newlabel $\star_S$ $\lambda L_\kappa$.
  newlabel $\star_S$ $\lambda L_c$.
  newlabel $\star_S$ $\lambda L_{test}$.
  $\mathcal{C}[\![b]\!]$ $\star_S$ $\lambda\varphi_b$.
  $\mathcal{C}[\![c]\!]$ $\star_S$ $\lambda\varphi_c$.
    $\textbf{unit}_S(\mathsf{WhilePS}(\varphi_b, \varphi_c, L_{test}, L_c, L_\kappa))$
```

```
$\mathsf{WhilePS}(\varphi_b, \varphi_c, L_{test}, L_c, L_\kappa) =$
    callcc $\lambda\kappa$.
        updateCode$[L_\kappa \mapsto \kappa\bullet]$ $\star_D$ $\lambda\_$.
        updateCode$[L_c \mapsto \varphi_c$ $\star_D$ (jump $L_{test}$)$]$ $\star_D$ $\lambda\_$.
        updateCode$[L_{test} \mapsto \varphi_b$ $\star_D$ $\lambda\beta.\beta\langle L_c, L_{test}\rangle]$ $\star_D$ $\lambda\_$.
            jump $L_{test}$
```

Figure 4.9: Compilation Semantics of the Control-Flow Block of Src

4.7.7 The Simple Control-Flow Language

Specification 6 (Linking Condition for Bool+CF) *The booleans produced by boolean RCBB $\mathcal{B}[\![-]\!]$ are* parametric. *That is, for $\beta : \forall\alpha.\alpha \times \alpha \to \alpha$: $\beta\langle a, b\rangle = a$ or $\beta\langle a, b\rangle = b$.*

Theorem 8 *$\mathcal{C}[\![- : Bool]\!]$ satisfies Specification 6.*

Please note that Theorem 8 holds by inspection. That is, for any two integers v_1, v_2, the boolean $\lambda\langle\kappa_T, \kappa_F\rangle.(v_1 \leq v_2 \;\to\; \kappa_T, \kappa_F)$ is always parametric.

Specification 7 (Control-Flow-spec) *Let $Addr + Label \preceq \mathsf{S}$, $CPS + Sto + Code \preceq \mathsf{D}$, and*

$\mathcal{K}[\![-]\!] : \mathit{ControlFlow} \to \mathsf{S}(\mathsf{D}(\mathtt{void}))$. *Then* $\mathcal{K}[\![-]\!]$ *is a correct RCBB for the control-flow block, if*

$$\begin{aligned}
&\mathcal{K}[\![cf]\!] \ \star_S \ \lambda\varphi.\mathbf{unit}_S(\varphi \ \star_D \ \lambda\_.\mathtt{init}_{CS}) \\
&\qquad = \ \mathtt{rdAddr} \ \star_S \ \lambda a. \\
&\qquad\quad\ \ \mathtt{getLabel} \ \star_S \ \lambda L. \\
&\qquad\quad\ \ \mathcal{S}[\![cf]\!] \ \star_S \ \lambda\delta. \\
&\qquad\quad\ \ \mathcal{K}[\![cf]\!] \ \star_S \ \lambda\varphi. \\
&\qquad\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a)\ \mathsf{AND}\ \mathsf{FreshLabel}(L), \delta, \varphi) \ \star_D \ \lambda\_.\mathtt{init}_{CS})
\end{aligned}$$

Theorem 9 $\mathcal{C}[\![- : \mathit{ControlFlow}]\!]$ *satisfies* **Control-Flow-spec**.

Theorem 9 is proved in Appendix B beginning on page 220.

4.7.8 Observational Correctness Relation

Specification 8 formalizes the correctness relation between the standard staged semantics $\mathcal{S}[\![- : \mathsf{Src}]\!]$ and reusable compiler building blocks for the sublanguages. This relation is referred to as *observational correctness.* The observational correctness of $\mathcal{C}[\![- : \mathsf{Src}]\!]$ (Theorem 10) corresponds to the middle link of Figure 4.1. Theorem 11 demonstrates that the compiler formed by expression block $\mathcal{C}[\![- : \mathit{Imp} + \mathit{Block} + \mathit{Bool} + \mathit{ControlFlow}]\!]$ with the constant-folding expression block $\mathsf{CF}[\![-]\!]$ is also observationally correct.

Specification 8 (Observational Correctness for Src Compiler)

$$\frac{\begin{array}{l}
\mathcal{E}[\![-]\!] \ \textit{satisfies } \textbf{Exp-spec} \textit{ and linking condition (Spec. 3)} \\
\mathcal{I}[\![-]\!] \ \textit{satisfies } \textbf{Imp-spec} \\
\mathcal{N}[\![-]\!] \ \textit{satisfies } \textbf{Block-spec} \\
\mathcal{B}[\![-]\!] \ \textit{satisfies } \textbf{Bool-spec} \textit{ and linking condition (Spec. 6)} \\
\mathcal{K}[\![-]\!] \ \textit{satisfies } \textbf{Control-Flow-spec}
\end{array}}{\mathcal{E}[\![-]\!] + \mathcal{I}[\![-]\!] + \mathcal{N}[\![-]\!] + \mathcal{B}[\![-]\!] + \mathcal{K}[\![-]\!] \ \textit{is an } \text{observationally correct } \textit{compiler for } \mathsf{Src}}$$

Theorem 10 follows from Theorems 3, 4, 6, 7, and 9.

Theorem 10 $\mathcal{C}[\![- : \mathsf{Src}]\!]$ *is a correct compiler for* Src.

Theorem 11 follows from Theorems 5, 4, 6, 7, and 9.

Theorem 11 $\mathsf{CF}[\![- : Exp]\!] + \mathcal{C}[\![- : Imp + Block + Bool + ControlFlow]\!]$ *is a correct compiler for* Src.

4.8 Conclusions

The structure of the proof of the Src compiler involved three parts:

1. The individual compiler building block-level specifications and verifications: **Exp-spec** (Specification 1), **Imp-spec** (Specification 2), **Block-spec** (Specification 4), **Bool-spec** (Specification 5), and **Control-Flow-spec** (Specification 7).

2. Linking Conditions. There were two linking conditions—**Imp-spec** assumed that **Exp-spec** held of *Exp* and **Control-Flow-spec** required that the booleans produced by *Bool* are parametric. Neither of these was difficult to prove.

The specifications in (1) relate the meanings of a term $t : \mathsf{Src}$ according to the staged standard semantics $\mathcal{S}[\![-]\!]$ and the compilation semantics $\mathcal{C}[\![-]\!]$. The main idea in these specifications is to require the value or effect produced by the compiled code (i.e., the compilation φ_t produced by $\mathcal{C}[\![t]\!]$) to be identical to that produced by δ_t (the dynamic computation produced by $\mathcal{S}[\![t]\!]$). The main challenge here was to "mask out" the effects on the implementation-level data used exclusively by $\mathcal{C}[\![t]\!]$. Observational program specification was used in this endeavor.

While this method of compiler verification does provide some level of modularity and reusability in compiler proofs, it does not provide *unrestrained* reusability. One can not combine arbitrary, correct compiler building blocks together into a correct compiler. For example, **Imp-spec** must mask out the code store with $\mathtt{init}_{CS}$ because in the Src language, **while-do** and **if-then** are also commands. While in modular compiler and interpreter construction, adding language building blocks together is quite elementary[14, 13, 25, 10], compiler specifications and verifications are far more delicate[1].

However, this methodology provides a certain level of modularity in compiler proofs. To prove Theorem 11, it was clear what (small) part of the overall proof had to be reproved. It was necessary to show only that CF satisfies **Exp-spec**. This experiment shows that the modularization of the language semantics using monads, monad transformers, and metacomputations provides a useful guide for organizing a compiler proof. This is a substantial benefit compared with previous efforts in this area[32, 46, 48].

[1] Of course, combining arbitrary language building blocks does not necessarily produce a sensible compiler or interpreter, even if combining the language building blocks is a simple matter. This fact is often ignored in the literature.

Chapter 5

Equivalence of $\mathcal{C}[\![-]\!]$ and `compile`

Two implementations of reusable compiler building blocks have been presented so far in the thesis. In Chapter 2, compiler building blocks were implemented as metacomputation-valued definitional interpreters. Using a standard technique from semantics-directed compilation [5, 8, 13, 14, 16, 18, 19, 38], code generation is accomplished through type-directed partial evaluation[7, 8] of the definitional interpreter. The second RCBB implementation—monadic code generators—can be viewed as code-valued interpreters. Monadic code generators produce target language code directly without the intermediate step of partial evaluation.

An overview of the correctness proof of the compiler for $\mathsf{Src} = \mathit{Exp} + \mathit{Imp} + \mathit{Block} + \mathit{Bool} + \mathit{ControlFlow}$ is shown in Figure 5.1. The first two links between $[\![-]\!]$ and $\mathcal{S}[\![-]\!]$ and between $\mathcal{S}[\![-]\!]$ and $\mathcal{C}[\![-]\!]$ were established in Chapter 4. This chapter concerns the last correspondence between $\mathcal{C}[\![-]\!]$ and $\mathsf{compile}$ (defined in Figures 5.4 and 5.5). First, a formal semantics $\mathcal{M}[\![-]\!]$ for the target language $\mathsf{TargetLang}$ is given in Section 5.1. Next, Section 5.3 demonstrates a few lemmas helpful to the proof of the main result. Finally, the main theorem establishing the equivalence of $\mathcal{C}[\![-]\!]$ and $\mathsf{compile}$ is shown.

Is there some reason for relating the metacomputation-based RCBB implementations of Chapter 2 to the monadic code generator RCBB implementations of Chapter 3 other than academic rigor? There are three principal reasons:

- The strength of the correctness relation between $\mathcal{C}[\![-]\!]$, $\mathsf{compile}$, and $\mathcal{M}[\![-]\!]$ demonstrates how well the metacomputation idea models staging.
- Doing so avoids the "pretty-printing" problem in semantics-directed compilation based on

$$\begin{tikzcd}
\text{Standard Semantics } [\![t]\!] : \mathsf{M}(Value) \arrow[d, leftrightarrow, "\text{Equivalence Theorem 2}"'] \\
\text{Staged Standard Semantics } \mathcal{S}[\![t]\!] : \mathsf{Static}(\mathsf{Dynam}(Value)) \arrow[d, leftrightarrow, "\text{Observational Specifications (}\textbf{Exp-spec}\text{,\ldots)}"'] \\
\text{Compilation Semantics } \mathcal{C}[\![t]\!] : \mathsf{Static}(\mathsf{Dynam}(Value)) \arrow[d, leftrightarrow, "\text{Equivalence of } \mathcal{C}[\![-]\!] \text{ \& } \mathsf{compile} \text{ Theorem 12}"'] \\
\text{Monadic Code Generator } \mathsf{compile}(t) : \mathsf{Static}(\mathsf{TargetLang}))
\end{tikzcd}$$

Figure 5.1: Src Compiler Correctness

partial evaluation. Until now, it has been tacitly assumed that the output from the type-directed partial evaluator resembles machine code, but how can one be sure of this?

- It helps to clarify the implicit run-time model—that is, how does one know that combining two RCBBs produces a sensible compiler?

Traditional semantics-directed compilation based on partial evaluation[5, 8, 14, 16, 17, 18, 19, 23, 38] applies a partial evaluator to a semantic language definition to generate code. The residualized term is purported to resemble machine language code so closely that it may be pretty-printed as machine code. This begs the question of whether the residualized term is *guaranteed* to resemble machine code, and hence, calls into question the viability of partial evaluation-based semantics-directed compilation. The metacomputation-based RCBB implementations of Chapter 2 are susceptible to this argument as well, because it was implicitly assumed that the terms residualized by type-directed partial evaluation resemble TargetLang code.

Formally relating $\mathcal{C}[\![-]\!]$ to compile establishes that partial evaluation is not essential to the approach to compiler construction advocated in this thesis, because the pretty-printing problem may be completely side-stepped. The monadic code generator RCBB implementations (i.e., compile) may be seen as "the" definitive compiler building blocks, and the metacomputation-based compilation semantics can be then viewed as an intermediate step in their verification. It may be possible

```
MachLang::=
        NOP|
        S := Rhs|
        MachLang; MachLang|
        JUMP L |
        ALLOC Addr |
        DEALLOC Addr |
        ENDLABEL L MachLang|
        SEGM (L,MachLang) |
        BRLEQ Addr Addr |
        MachLang ◇ MachLang × MachLang |

Rhs ::= SRhs | -SRhs | SRhs+SRhs
SRhs ::= a ∈ Addr | #c for integer c

IntProducer = MachLang × Rhs × Set(Addr)

TargetLang= MachLang+ IntProducer
```

Figure 5.2: BNF for the Target Language

to solve the pretty-printing problem with the kind of type-based analysis that is found in languages with explicit staging like MetaML[45], but this is left for future research.

Another reason to relate $\mathcal{C}[\![-]\!]$ and compile is that it helps clarify which compiler building blocks may be combined sensibly. A monadic code generator RCBB implementation is necessarily strongly wedded to a particular machine language, which is, in turn, wedded to a particular run-time model. The target language in Chapter 3 used a stack organized into stack frames, while the the target language in the present Chapter uses a stack of integers. One could imagine monadic code generator RCBBs which use heap-based allocation, and the target language for such an RCBB would contain a malloc-like operator. It is a necessary (although not sufficient) condition for two RCBBs to combine sensibly that combining their target languages makes sense as well.

Definition 25 *The semantics of the target language:*

$$\mathcal{M}[\![\texttt{NOP}]\!] = \mathbf{unit}_D(\bullet)$$

$$\mathcal{M}[\![a{:=}rhs]\!] = \mathcal{M}[\![rhs]\!] \;\star_D\; \lambda i : int.\texttt{store}(a, i)$$

$$\texttt{store}(a, i) = \;\texttt{getSto} \;\star_D\; \lambda\sigma. \\ \left(\begin{array}{l} \text{if } \langle a, - \rangle \in \sigma \text{ then} \\ \qquad \texttt{updateSto}(\lambda\sigma.(\sigma \backslash \langle a, - \rangle) \cup \langle a, i \rangle) \\ \text{else} \\ \qquad \bot_{\mathsf{Dynam}(\texttt{void})} \end{array} \right)$$

$$\mathcal{M}[\![\pi_1 \;\texttt{;}\; \ldots \;\texttt{;}\; \pi_n]\!] = \mathcal{M}[\![\pi_1]\!] \;\star_D\; \lambda\_.\ldots \;\star_D\; \lambda\_.\mathcal{M}[\![\pi_n]\!]$$

$$\mathcal{M}[\![\texttt{SEGM(}L, \pi\texttt{)}]\!] = \texttt{updateCode}[L \mapsto \mathcal{M}[\![\pi]\!]]$$

$$\mathcal{M}[\![\texttt{JUMP}\;\; L]\!] = \texttt{jump}\; L$$

$$\mathcal{M}[\![\texttt{ENDLABEL L}\; \pi]\!] = \texttt{callcc}\; (\lambda\kappa.\texttt{updateCode}[L \mapsto \kappa\bullet] \;\star_D\; \lambda\_.\mathcal{M}[\![\pi]\!])$$

$$\mathcal{M}[\![\texttt{ALLOC}(a)]\!] = \;\texttt{getSto} \;\star_D\; \lambda\sigma. \\ \left(\begin{array}{l} \text{if } \langle a, - \rangle \in \sigma \text{ then} \\ \qquad \bot_{\mathsf{Dynam}(\texttt{void})} \\ \text{else} \\ \qquad \texttt{updateSto}(\lambda\sigma.\, \sigma \cup \langle a, 0 \rangle) \end{array} \right)$$

$$\mathcal{M}[\![\texttt{DEALLOC}(a)]\!] = \;\texttt{getSto} \;\star_D\; \lambda\sigma. \\ \left(\begin{array}{l} \text{if } \langle a, - \rangle \in \sigma \text{ then} \\ \qquad \texttt{updateSto}(\lambda\sigma.\, \sigma \backslash \langle a, - \rangle) \\ \text{else} \\ \qquad \bot_{\mathsf{Dynam}(\texttt{void})} \end{array} \right)$$

$$\mathcal{M}[\![\langle \pi, rhs, tmps \rangle]\!] = \mathcal{M}[\![\pi]\!] \;\star_D\; \lambda\_.\; \mathcal{M}[\![rhs]\!] \;\star_D\; \lambda i : int.\; \mathcal{M}[\![(\mathsf{pop}\; tmps)]\!] \;\star_D\; \lambda\_.\mathbf{unit}_D(i)$$

$$\begin{array}{rl} \mathcal{M}[\![\texttt{BRLEQ}\; a\; a']\!] = & \texttt{read}(a) \;\star_D\; \lambda v. \\ & \texttt{read}(a') \;\star_D\; \lambda v'. \\ & \texttt{DEALLOC}(a) \;\star_D\; \lambda\_. \\ & \texttt{DEALLOC}(a') \;\star_D\; \lambda\_. \\ & \quad \mathbf{unit}_D(\lambda\langle x, y \rangle.v \leq v' \;\rightarrow\; x, y) \end{array}$$

$$\texttt{read}(a) = \texttt{getSto} \;\star_D\; \lambda\sigma.[\text{if } \langle a, v \rangle \in \sigma \text{ then } \mathbf{unit}_D(v) \text{ else } \bot_{\mathsf{Dynam}(\texttt{void})}]$$

$$\begin{array}{rl} \mathcal{M}[\![\diamond\; \pi_b\; \langle \pi_T, \pi_F \rangle]\!] = & \mathcal{M}[\![\pi_b]\!] \;\star_D\; \lambda b : \mathsf{Dynam}(\texttt{void}) \times \mathsf{Dynam}(\texttt{void}) \rightarrow \mathsf{Dynam}(\texttt{void}). \\ & \quad b\, \langle \mathcal{M}[\![\pi_T]\!], \mathcal{M}[\![\pi_F]\!] \rangle \end{array}$$

Figure 5.3: Formal Semantics of the Target Language

5.1 The Target Language and Its Semantics

Figure 5.3 presents the formal semantics $\mathcal{M}[\![-]\!]$ of the target language $\mathsf{TargetLang}$. The type of $\mathcal{M}[\![-]\!]$ is:

$$\mathcal{M}[\![t : \mathsf{TargetLang}]\!] : \mathsf{Dynam}(int + \mathtt{void} + \mathsf{Dynam}(\mathtt{void}) \times \mathsf{Dynam}(\mathtt{void}) \rightarrow \mathsf{Dynam}(\mathtt{void}))$$

The meanings of $\mathtt{NOP}$, $a{:=}rhs$, and π_1 ; π_2 are just what they are in the standard semantics for skip, assignment, and sequencing, except that $a{:=}rhs$ assigns directly to an address, so no environment look-up is necessary. $\mathcal{M}[\![\mathtt{SEGM}(L, \pi)]\!]$ updates the code store at label L with the meaning of the stored code $\mathcal{M}[\![\pi]\!]$. $\mathtt{JUMP}\, L$ retrieves the code store and executes the segment stored at L. ($\mathtt{ENDLABEL}\, L \;\; \pi$) gets the current continuation, stores it in the code store, and then executes π. Any jumps to L inside of π will have the effect of jumping out of π. The meaning of $\mathtt{ALLOC}(a)$ looks to see if a is a free address, and if so, adds it to the current store. Otherwise, $\mathtt{ALLOC}(a)$ fails. $\mathtt{DEALLOC}(a)$ is similar. The meanings of $\mathtt{ALLOC}(a)$ and $\mathtt{DEALLOC}(a)$ are identical to the combinators $\mathtt{Alloc}(a)$ and $\mathtt{deAlloc}(a)$ defined in Chapter 4. The meaning of an $\mathsf{IntProducer}$ $\langle \pi, rhs, tmps \rangle$ is (1) execute the code $\mathcal{M}[\![\pi]\!]$, (2) evaluate the right-hand side $\mathcal{M}[\![rhs]\!]$ producing an integer i, (3) deallocate all of the temporaries used in π, and (4) return i. The branch on less-than-or-equal $\mathcal{M}[\![\mathtt{BRLEQ}\; a\; a']\!]$ gets the current values at addresses a and a' (call them v and v', respectively), deallocates a and a', then returns the function which chooses x or y from $\langle x, y \rangle$ based on whether $v \leq v'$. Finally, the meaning of the "code apply" operator $\diamond$ is to evaluate its first argument, producing a choice function b, and then to apply b to the pair of meanings of the true and false branches.

5.2 The Correctness Relation between $\mathcal{C}[\![-]\!]$, $\mathsf{compile}$, and $\mathcal{M}[\![-]\!]$

Before beginning with the details of the chapter, it may be helpful to examine at a high level the equivalence theorem:

Theorem 12 (Relating $\mathcal{C}[\![-]\!]$, $\mathsf{compile}$, and $\mathcal{M}[\![-]\!]$) *For any well-typed term t in the source lan-*

$\mathsf{compile} : \mathsf{Src} \to \mathsf{M}(\mathsf{TargetLang})$
$\mathsf{M} = \mathsf{Static}$

$(\mathsf{pop}\ \{\}) = \mathtt{NOP}$
$(\mathsf{pop}\ \{a_1, \ldots, a_n\}) = \mathtt{DEALLOC}\,a_1\ ;\ \ldots\ ;\ \mathtt{DEALLOC}\,a_n$

$\mathsf{compile}(i) = \mathbf{unit}\langle \mathtt{NOP}, \mathtt{\#}i, \{\ \}\rangle$

$$\mathsf{compile}(-e) = \\ \qquad \mathtt{rdAddr} \star \lambda a.\mathtt{inAddr}\,(a+1) \left(\begin{array}{l} (\mathsf{compile}\,e) \star \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle. \\ \quad \mathbf{unit}\langle \pi_e\,;\mathtt{ALLOC}\,a\,;a{:=}\mathit{rhs}_e\,;(\mathsf{pop}\ \mathit{tmps}_e), -a, \{a\}\rangle \end{array} \right)$$

$$\begin{array}{ll} \mathsf{compile}(x := e) = & \mathtt{rdEnv} \star \lambda\rho. \\ & (\rho\,x) \star \lambda a_x. \\ & \mathsf{compile}(e) \star \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle. \\ & \ \mathbf{unit}(\pi_e\ ;\ a_x{:=}\mathit{rhs}_e\ ;\ (\mathsf{pop}\ \mathit{tmps}_e)) \end{array}$$

$\mathsf{compile}(c_1;c_2) = \mathsf{compile}(c_1) \star \lambda\pi_1.\mathsf{compile}(c_2) \star \lambda\pi_2.\mathbf{unit}(\pi_1\ ;\ \pi_2)$

$$\begin{array}{ll} \mathsf{compile}(\mathtt{new}\,x\,\mathtt{in}\,c) = & \mathtt{rdAddr} \star \lambda a.\mathtt{rdEnv} \star \lambda\rho. \\ & \quad [\mathtt{inAddr}\,(a{+}1)\ (\mathtt{inEnv}\,\rho[x \mapsto (\mathbf{unit}\,a)]\ (\mathsf{compile}\,c))] \star \lambda\pi_c. \\ & \qquad \mathbf{unit}(\mathtt{ALLOC}(a)\ ;\ \pi_c\ ;\ \mathtt{DEALLOC}(a)) \end{array}$$

$\mathsf{compile}(x) = \mathtt{rdEnv} \star \lambda\rho.(\rho\ x) \star \lambda a.\mathbf{unit}\langle \mathtt{NOP}, a, \{\}\rangle$

Figure 5.4: Monadic Code Generator for Src (part 1)

$$
\begin{array}{l}
\mathsf{compile}(e_1 \;\mathsf{leq}\; e_2) = \\
\quad \texttt{rdAddr} \,\star\, \lambda a. \\
\quad \texttt{inAddr}\;(a+2) \\
\qquad \mathsf{compile}(e_1) \,\star\, \lambda\langle \pi_1, \mathit{rhs}_1, \mathit{tmps}_1\rangle. \\
\qquad \mathsf{compile}(e_2) \,\star\, \lambda\langle \pi_2, \mathit{rhs}_2, \mathit{tmps}_2\rangle. \\
\qquad\quad \mathbf{unit} \left(\begin{array}{l} \pi_1 \;;\; \texttt{ALLOC}(a) \;;\; a{:=}\mathit{rhs}_1 \;;\; \pi_2 \;;\; \texttt{ALLOC}(a+1) \;; \\ \quad (a{+}1){:=}\mathit{rhs}_2 \;;\; (\mathsf{pop}\,\mathit{tmps}_1) \;;\; (\mathsf{pop}\,\mathit{tmps}_2) \;;\; \texttt{BRLEQ}\,a\,(a{+}1)) \end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{compile}(\mathbf{if}\; b\; \mathbf{then}\; c) = \\
\quad \texttt{newlabel} \,\star\, \lambda L_{\mathrm{exit}}. \\
\quad \texttt{newlabel} \,\star\, \lambda L_c. \\
\quad \mathsf{compile}(b) \,\star\, \lambda \pi_b. \\
\quad \mathsf{compile}(c) \,\star\, \lambda \pi_c. \\
\qquad \mathbf{unit}(\texttt{ENDLABEL}\; L_{\mathrm{exit}} \;\; (\texttt{SEGM}[L_c, \pi_c \;;\; \texttt{JUMP}\; L_{\mathrm{exit}}] \;;\; (\pi_b \diamond \langle \texttt{JUMP}\; L_c, \texttt{JUMP}\; L_{\mathrm{exit}}\rangle)))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{compile}(\mathbf{while}\; b\; \mathbf{do}\; c) = \\
\quad \texttt{newlabel} \,\star\, \lambda L_{test}. \\
\quad \texttt{newlabel} \,\star\, \lambda L_c. \\
\quad \texttt{newlabel} \,\star\, \lambda L_{exit}. \\
\quad (\mathsf{compile}\, b) \,\star\, \lambda \pi_b. \\
\quad (\mathsf{compile}\, c) \,\star\, \lambda \pi_c. \\
\qquad \mathbf{unit}(\texttt{ENDLABEL}\; L_{exit} \left(\begin{array}{l} \texttt{SEGM}[L_c, \pi_c \;;\; \texttt{JUMP}\; L_{test}] \;; \\ \texttt{SEGM}[L_{test}, \pi_b \diamond \langle \texttt{JUMP} L_c, \texttt{JUMP} L_{exit}\rangle] \;; \\ \texttt{JUMP} L_{test} \end{array} \right))
\end{array}
$$

Figure 5.5: Monadic Code Generator for Src (part 2)

guage $\mathsf{Src} = \mathit{Exp} + \mathit{Imp} + \mathit{ControlFlow} + \mathit{Block} + \mathit{Bool}$,

$$\mathcal{C}[\![t]\!] = \mathsf{compile}(t) \ \star_S \ \lambda \pi_t.\mathbf{unit}_S(\mathcal{M}[\![\pi_t]\!])$$

The theorem states that the formal meaning of the code produced for well-typed t by $\mathsf{compile}$, $\mathcal{M}[\![\pi_t]\!]$, is identical to the compilation $\varphi_t : \mathsf{Dynam}(\mathit{Value})$. Theorem 12 is an extremely strong relation, which demonstrates how well the metacomputation idea characterizes staging semantically.

5.3 Technical Lemmas

Lemma 13 specifies the interactions between the combinators `store`, `read`, `Alloc`, and `deAlloc`. This lemma is used in the proof of Lemma 14. It follows directly from Axiom 2.

Lemma 13 *Given $a, a' : \mathit{Addr}$ such that $a \neq a'$:*

i. $\mathtt{store}(a,v) \ \star_D \ \lambda\_.\mathtt{deAlloc}(a') = \mathtt{deAlloc}(a') \ \star_D \ \lambda\_.\mathtt{store}(a,v)$

ii. $\mathtt{Alloc}(a) \ \star_D \ \lambda\_.\mathtt{deAlloc}(a') = \mathtt{deAlloc}(a') \ \star_D \ \lambda\_.\mathtt{Alloc}(a)$

iii. $\mathtt{Alloc}(a) \ \star_D \ \lambda\_.\mathtt{read}(a') \ \star_D \ \lambda v.(f\ v) = \mathtt{read}(a') \ \star_D \ \lambda v.\mathtt{Alloc}(a) \ \star_D \ \lambda\_.(f\ v)$

In the meaning of an expression according to the compilation semantics $\mathcal{C}[\![e]\!]$, temporary addresses are deallocated as soon as they are used. Consider the definition of compilation Negate (the dynamic part $\mathcal{C}[\![-e]\!]$):

$$\begin{aligned}\mathsf{Negate}(\varphi_e, a) = \ & \varphi_e \ \star_D \ \lambda i. \\ & \mathtt{Alloc}(a) \ \star_D \ \lambda\_. \\ & \mathtt{Thread}(i,a) \ \star_D \ \lambda v. \\ & \mathtt{deAlloc}(a) \ \star_D \ \lambda\_. \\ & \qquad \mathbf{unit}_D(-v)\end{aligned}$$

Here, $\mathtt{deAlloc}(a)$ occurs right after the value i is stored at a. Consider an $\mathsf{IntProducer}$ for a negation:

$$\langle \pi_e \, ; \mathtt{ALLOC}\, a \, ; a{:=}\mathit{rhs}_e \, ; (\mathsf{pop}\ \mathit{tmps}_e), -a, \{a\} \rangle$$

The temporary address a is not deallocated until the right-hand side $-a$ is used. This has the effect that deallocations in $\mathsf{IntProducers}$ can occur later than in corresponding compilations produced by $\mathcal{C}[\![-]\!]$. Lemma 14 shows that this has no effect on the value produced.

Lemma 14 (Separability)

$$
\begin{array}{l}
\texttt{rdAddr} \star_S \lambda a. \\
\quad (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(e)) \star_S \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle. \\
\qquad \mathbf{unit}_S(\mathcal{M}[\![\langle \pi_e \ ; \texttt{ALLOC}(a) \ ; a{:=}\mathit{rhs}_e \ ; \mathsf{pop}(\mathit{tmps}_e), -a, \{a\}\rangle]\!]) \\
\qquad\qquad\qquad\qquad = \texttt{rdAddr} \star_S \lambda a. \\
\qquad\qquad\qquad\qquad\quad (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(e)) \star_S \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle. \\
\qquad\qquad\qquad\qquad\qquad \mathbf{unit}_S(\mathsf{Negate}(\mathcal{M}[\![\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle]\!], a))
\end{array}
$$

Proof of Lemma 14 is included in Appendix C beginning on page 246.

Lemma 15 relates the compilation produced for an **if** statement with the corresponding $\mathsf{TargetLang}$ code produced by $\mathsf{compile}$.

Lemma 15 *For all* $\pi_b, \pi_c : \mathsf{MachLang}$ (*where* π_b *is produced by* $\mathsf{compile}(b : \textbf{boolexp})$), *and* L_c, L_{exit} : *Label:*

$$
\begin{array}{l}
\mathsf{IfThen}(\mathcal{M}[\![B]\!], \mathcal{M}[\![\pi_c]\!], L_c, L_{exit}) = \\
\quad \mathcal{M}[\![(\texttt{ENDLABEL}\ L_{\mathrm{exit}}\ \ (\texttt{SEGM}(L_c,\ \pi_c \ ; \texttt{JUMP}\ L_{\mathrm{exit}}) \ ; \pi_b \diamond \langle \texttt{JUMP}L_c, \texttt{JUMP}L_{\mathrm{exit}}\rangle))]\!]
\end{array}
$$

Proof of Lemma 15 is included in Appendix C beginning on page 254.

5.4 Proof of Main Theorem

This section presents the equivalence between $\mathcal{C}[\![-]\!]$ and $\mathsf{compile}$ in Theorem 12.

Definition 26 *For a set of variables* V *and an environment* ρ : *env, define* $V \cap \rho$ *to be the* *environment:*

$$\lambda x.\text{if } x \in V \text{ then } \ (\rho\, x) \ \text{ else } \ \bot$$

Definition 27 *For address a : Addr and environment ρ : env, define $\rho < a$ if and only if, for every variable x, if $(\rho\, x) = \mathbf{unit}(a')$ for some address a' : Addr, then $a' < a$.*

Definition 28 (Relation $\Re$) *Define relation $\Re(a^* : Addr, \rho^* : env, t : \mathsf{Src})$ as:*

$$(\mathsf{FreeVars}(t) \cap \rho^*) < a^* \Longrightarrow$$

$$\texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ \mathcal{C}[\![t]\!]) = \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ (\mathsf{compile}(t)\ \star_S\ \lambda\pi_t.\mathbf{unit}_S(\mathcal{M}[\![\pi_t]\!])))$$

Lemma 16 ($\Re$-lemma) $\forall a^* : Addr, \rho^* : env, t : \mathsf{Src}.\ \Re(a^*, \rho^*, t)$.

The proof of **$\Re$-lemma** is included in Appendix C beginning on page 256.

Theorem 12 (Relating $\mathcal{C}[\![-]\!]$ and $\mathcal{M}[\![-]\!]$) *For any well-typed term t in the source language $\mathsf{Src} = Exp + Imp + ControlFlow + Block + Bool$,*

$$\mathcal{C}[\![t]\!] = \mathsf{compile}(t)\ \star_S\ \lambda\pi_t.\mathbf{unit}_S(\mathcal{M}[\![\pi_t]\!])$$

$\boxed{\text{Proof of Theorem 12}}$

Let ρ^* be the empty environment $(\lambda x.\ \bot)$ and a^* be the address 0. It is clear that: $\mathsf{FreeVars}(t) \cap \rho^*) < a^*$. By Lemma 16, it is also known that $\Re(a^*, \rho^*, t)$. Therefore:

$$\texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ \mathcal{C}[\![t]\!]) = \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ (\mathsf{compile}(t)\ \star_S\ \lambda\pi_t.\mathbf{unit}_S(\mathcal{M}[\![\pi_t]\!])))$$

Because t is well-typed, it contains no free variables, and so the `inEnv` and `inAddr` can be dropped from both sides. Thus:

$$\mathcal{C}[\![t]\!] = \mathsf{compile}(t)\ \star_S\ \lambda\pi_t.\mathbf{unit}_S(\mathcal{M}[\![\pi_t]\!])$$

□

Chapter 6

Conclusions

The author's central research objective is to develop methods of structuring software that make large-scale software systems more maintainable, modifiable, and reliable. This dissertation focuses on building modular compilers for high-level programming languages and proving them correct. Compilers have traditionally been factored into phases (e.g., parsing, code generation, etc.), while compilers constructed according to the method developed here are also structured by source language feature (e.g., expressions, procedures, etc.), yielding a "mix and match" approach to compiler construction. It is quite easy to modify or extend an existing compiler which has been constructed according to this method, and it is also a simple matter to reuse existing work.

The main contributions of the present work are:

- **Modular Compilation.** This dissertation concerns the construction of modular compilers for high-level programming languages from reusable compiler building blocks. With this approach, compilers for a language with many features (e.g., expressions, procedures, etc.) are built using compiler building blocks for each specific feature. Each compiler building block compiles a specific feature and can be easily combined with compiler building blocks for other features to provide compilers for non-trivial languages with many features. Furthermore, each compiler building block is reusable and may be used in the construction of compilers for many different languages. Compilers constructed according to this method are modular in that source language features may be added or deleted with ease, allowing the compiler writer to develop compilers at a very high level of abstraction. This approach to structuring compilers is completely new in compiler design and was motivated by the insight that the same advantages provided by using the categorical concepts of *monads* and *monad transformers* in structuring interpreters carry over to compilers. To date, compiler building blocks have been developed for such features as expressions, call-by-value and call-by-name procedures, recursive bindings, dynamic binding, control flow, and block structure, and a number of compilers have been constructed using these building blocks.

- **Metacomputation-style Staging.** A metacomputation is a formal semantic model of staged computation. Metacomputation-style language specifications factor the compile-time and run-time parts of a specification into separate monads. Metacomputation-style specifications retain the modularity, extensibility, and reusability of monadic specifications and closely resemble semantic versions of the translation schemas found in traditional hand-written compilers, while remaining close to the usual intended semantics of the language. In a sense, metacomputation-style reconciles the informal approach to code generation taken in traditional hand-written compilers with semantics-based compilation.

  Metacomputations provide a new modular and extensible style of staging denotational specifications. Staging in language specifications, or distinguishing compile-time parts of a specification from run-time parts, is important in semantics-based compilation because it substantially reduces the need for a sophisticated partial evaluator. In the context of modular compilation, staging also makes the construction and combination of compiler building blocks simpler.

- **Two Implementations of Reusable Compiler Building Blocks.** Reusable compiler building blocks for language features such as expressions, imperative, control-flow, block structure, booleans, open/closed procedures, call-by-name, call-by-value, dynamic binding, recursion were implemented in both metacomputation-style and as monadic code generators.

- **Novel Approach to Compiler Correctness.** Given certain preconditions, correct compiler building blocks may be combined into correct compilers, thereby attaining a level of *reusability* in compiler correctness proofs. While this method of compiler verification does provide some level of modularity and reusability in compiler proofs, it does not provide *unrestrained* reusability. That is, one can not combine arbitrary, correct compiler building blocks together into a correct compiler.

  However, this methodology provided a certain level of modularity in compiler proofs. This case study in compiler verification shows that the modularization of the language semantics using monads, monad transformers, and metacomputations does provide a useful guide for organizing a compiler proof. This is a substantial benefit compared with previous efforts in this area[32, 46, 48].

- **Observational Program Specification.** This is a form of monadic program specification which makes minimal assumptions about the monad in which its specifications are interpreted. It was particularly valuable in the specification and verification of modular compilers, whose building blocks are intended to be interpreted in many different monadic contexts.

Appendix A

Semantic Definitions of Src

The following definitions are used throughout this Appendix.

Operations on the store:

$$\texttt{store}(a,i) = \texttt{getSto} \star_D \lambda\sigma. \left(\begin{array}{l} \textit{if } \langle a,-\rangle \in \sigma \textit{ then} \\ \qquad \texttt{updateSto}(\lambda\sigma.(\sigma\backslash\langle a,-\rangle) \cup \langle a,i\rangle) \\ \textit{else} \\ \qquad \bot_{\mathsf{Dynam(\texttt{void})}} \end{array} \right)$$

$\texttt{read}(a) = \texttt{getSto} \star_D \lambda\sigma.[\text{if } \langle a,v\rangle \in \sigma \text{ then } \mathbf{unit}_D(v) \text{ else } \bot_{\mathsf{Dynam(\texttt{void})}}]$

$\texttt{Alloc}(a) = \texttt{getSto} \star_D \lambda\sigma. \text{ if } a \in \sigma \text{ then } \bot_{\mathsf{Dynam(\texttt{void})}} \text{ else } \texttt{updateSto}[\sigma^* \mapsto \sigma^* \cup \{\langle a,0\rangle\}]$

$\texttt{deAlloc}(a) = \texttt{getSto} \star_D \lambda\sigma. \text{ if } a \notin \sigma \text{ then } \bot_{\mathsf{Dynam(\texttt{void})}} \text{ else } \texttt{updateSto}[\sigma^* \mapsto \sigma^* \setminus \{\langle a,-\rangle\}]$

Control-flow operations:

$\mathsf{dynwhile} : \mathsf{Dynam}(\mathbf{bool}) \times \mathsf{Dynam}(\texttt{void}) \to \mathsf{Dynam}(\texttt{void})$ is defined:

$\mathsf{dynwhile}(B,\varphi) = \mathsf{fix}(\lambda dw.B \star_D \lambda\beta : \mathbf{bool}.\texttt{callcc}\ \lambda\kappa.\beta\langle\varphi \star_D \lambda\_.dw(B,\varphi), \kappa\bullet\rangle)$

$\mathsf{IfThen} : \mathsf{Dynam}(\mathbf{bool}) \times \mathsf{Dynam}(\texttt{void}) \to \mathsf{Dynam}(\texttt{void})$ is defined by:

$\mathsf{IfThen}(\varphi_b,\varphi_c) = \varphi_b \star_D \lambda\beta : \mathbf{bool}.\texttt{callcc}\ \lambda\kappa.\beta\langle\varphi_c \star_D \kappa, \kappa\bullet\rangle$

$\texttt{jump } L = \texttt{getCode} \star_D \lambda\Pi.\texttt{callcc}\ \lambda\kappa. \text{ if } \langle L,\pi\rangle \in \mathit{dom}(\Pi) \text{ then } \pi \text{ else } \bot_{\mathsf{Dynam(\texttt{void})}}$

In the following definitions of IfThenPS and WhilePS, the "PS" refers to these definitions being *pass-separated* in the sense of Jørring and Scherlis[20].

$$
\begin{array}{ll}
\mathsf{IfThenPS}(\varphi_b, \varphi_c, L_c, L_\kappa) = & \varphi_b \ \star_D \ \lambda\beta. \\
& \quad \mathtt{callcc}\ \lambda\kappa. \\
& \quad \mathtt{updateCode}[L_\kappa \mapsto \kappa\bullet] \ \star_D \ \lambda\_. \\
& \quad \mathtt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ \lambda\_.\mathtt{jump}\, L_\kappa] \ \star_D \ \lambda\_. \\
& \qquad \beta\langle \mathtt{jump}\, L_c, \mathtt{jump}\, L_\kappa \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{WhilePS}(\varphi_b, \varphi_c, L_{test}, L_c, L_\kappa) = \\
\qquad\qquad \mathtt{callcc}\ \lambda\kappa. \\
\qquad\qquad\quad \mathtt{updateCode}[L_\kappa \mapsto \kappa\bullet] \ \star_D \ \lambda\_. \\
\qquad\qquad\quad \mathtt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ (\mathtt{jump}\, L_{test})] \ \star_D \ \lambda\_. \\
\qquad\qquad\quad \mathtt{updateCode}[L_{test} \mapsto \varphi_b \ \star_D \ \lambda\beta.\beta\langle \mathtt{jump}\ L_c, \mathtt{jump}\ L_\kappa \rangle] \ \star_D \ \lambda\_. \\
\qquad\qquad\qquad \mathtt{jump}\, L_{test}
\end{array}
$$

A.1 Standard Semantics $[\![-]\!]$ for Src

$$[\![-]\!] : \mathsf{Src} \to \mathsf{Static} + \mathsf{Dynam}(\mathtt{void} + \mathit{int} + \mathbf{bool})$$

$$[\![i]\!] = \mathbf{unit}(i)$$

$$[\![-e]\!] = [\![e]\!] \ \star\ \lambda i.\mathbf{unit}(-i)$$

$$[\![x]\!] = \mathtt{rdEnv} \ \star\ \lambda\rho.(\rho\, x) \ \star\ \lambda a : \mathit{Addr}.\ \mathtt{read}(a)$$

$$[\![c_1 \ ;\, c_2]\!] = [\![c_1]\!] \ \star\ \lambda\_.[\![c_2]\!]$$

$$[\![x := e]\!] = \mathtt{rdEnv} \ \star\ \lambda\rho.(\rho\, x) \ \star\ \lambda a : \mathit{Addr}.[\![e]\!] \ \star\ \lambda i : \mathit{int}.\ \mathtt{store}(a, i)$$

$$\begin{array}{ll} [\![\mathbf{new}\ x\ \mathbf{in}\ c]\!] = & \mathtt{rdAddr} \ \star\ \lambda a. \\ & \qquad\qquad \mathtt{inAddr}\,(a+1) \left(\begin{array}{l} \mathtt{rdEnv} \ \star\ \lambda\rho. \\ \mathtt{inEnv}\,(\rho[x \mapsto \mathbf{unit}(a)]) \\ \quad \mathtt{Alloc}(a) \ \star\ \lambda\_.[\![c]\!] \ \star\ \lambda\_.\mathtt{deAlloc}(a) \end{array} \right) \end{array}$$

$$[\![e_1 \ \ \mathsf{leq} \ \ e_2]\!] = [\![e_1]\!] \ \star\ \lambda v_1.[\![e_2]\!] \ \star\ \lambda v_2.\mathbf{unit}(\lambda\langle\kappa_T, \kappa_F\rangle.(v_1 \leq v_2 \to \kappa_T, \kappa_F))$$

$$[\![\mathbf{if}\ b\ \mathbf{then}\ c]\!] = \mathsf{IfThen}([\![b]\!], [\![c]\!])$$

$$[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = \mathsf{dynwhile}([\![b]\!], [\![c]\!])$$

A.2 Staged Standard Semantics $\mathcal{S}[\![-]\!]$ for Src

$$\mathcal{S}[\![-]\!] : \mathsf{Src} \to \mathsf{Static}(\mathsf{Dynam}(\mathtt{void} + int + \mathbf{bool}))$$

$$\mathcal{S}[\![i]\!] = \mathbf{unit}_S(\mathbf{unit}_D(i))$$

$$\mathcal{S}[\![-e]\!] = \mathcal{S}[\![e]\!] \ \star_S \ \lambda\varphi_e : \mathsf{Dynam}(int).\ \mathbf{unit}_S(\varphi_e \ \star_D \ \lambda i.\mathbf{unit}_D(-i))$$

$$\mathcal{S}[\![x]\!] = \mathtt{rdEnv} \ \star_S \ \lambda\rho.(\rho\, x) \ \star_S \ \lambda a : Addr.\ \mathbf{unit}_S(\mathtt{read}(a))$$

$$\begin{aligned}
\mathcal{S}[\![c_1 \ ; c_2]\!] &= \mathcal{S}[\![c_1]\!] \ \star_S \ \lambda\varphi_1.\mathcal{S}[\![c_2]\!] \ \star_S \ \lambda\varphi_2.\ \mathbf{unit}_S(\varphi_1 \ \star_D \ \lambda\_.\varphi_2) \\
\mathcal{S}[\![x := e]\!] &= \ \mathtt{rdEnv} \ \star_S \ \lambda\rho. \\
&\qquad (\rho\, x) \ \star_S \ \lambda a : Addr. \\
&\qquad\qquad \mathcal{S}[\![e]\!] \ \star_S \ \lambda\varphi_e : \mathsf{Dynam}(int).\ \mathbf{unit}_S(\varphi_e \ \star_D \ \lambda i : int.\mathtt{store}(a, i))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{S}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!] = \\
&\quad \mathtt{rdAddr} \ \star_S \ \lambda a. \\
&\quad \mathtt{inAddr}\,(a+1) \left(\begin{array}{l} \mathtt{rdEnv} \ \star_S \ \lambda\rho. \\ \mathtt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_S(a)])\ \mathcal{S}[\![c]\!] \end{array} \right) \star_S \ \lambda\varphi_c. \\
&\qquad \mathbf{unit}_S(\mathtt{Alloc}(a) \ \star_D \ \lambda\_.\varphi_c \ \star_D \ \lambda\_.\mathtt{deAlloc}(a))
\end{aligned}$$

$$
\begin{aligned}
\mathcal{S}[\![e_1 \ \mathsf{leq} \ \ e_2]\!] = \ & \mathcal{S}[\![e_1]\!] \ \star_S \ \lambda\varphi_1 : \mathsf{Dynam}(int). \\
& \mathcal{S}[\![e_2]\!] \ \star_S \ \lambda\varphi_2 : \mathsf{Dynam}(int). \\
& \mathbf{unit}_S \left(\begin{array}{l} \varphi_1 \ \star_D \ \lambda v_1 : int. \\ \varphi_2 \ \star_D \ \lambda v_2 : int. \\ \quad \mathbf{unit}_D(\lambda\langle\kappa_T, \kappa_F\rangle.(v_1 \leq v_2 \rightarrow \kappa_T, \kappa_F)) \end{array} \right)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}[\![\mathbf{if}\ b\ \mathbf{then}\ c]\!] = \ & \mathcal{S}[\![b]\!] \ \star_S \ \lambda\varphi_b : \mathsf{Dynam}(\mathbf{bool}). \\
& \mathcal{S}[\![c]\!] \ \star_S \ \lambda\varphi_c : \mathsf{Dynam}(\mathtt{void}). \\
& \mathbf{unit}_S(\mathsf{IfThen}(\varphi_b, \varphi_c))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = \ & \mathcal{S}[\![b]\!] \ \star_S \ \lambda B : \mathsf{Dynam}(\mathbf{bool}). \\
& \mathcal{S}[\![c]\!] \ \star_S \ \lambda\varphi_c : \mathsf{Dynam}(\mathtt{void}). \\
& \mathbf{unit}_S(\mathsf{dynwhile}(B, \varphi_c))
\end{aligned}
$$

A.3 Compilation Semantics $\mathcal{C}[\![-]\!]$ for Src

$$\mathcal{C}[\![-]\!] : \mathsf{Src} \to \mathsf{Static}(\mathsf{Dynam}(\texttt{void} + \mathit{int} + \mathbf{bool}))$$

$$\mathcal{C}[\![i]\!] = \mathbf{unit}_S(\mathbf{unit}_D(i))$$

$$\begin{array}{l}
\mathcal{C}[\![-e]\!] = \ \texttt{rdAddr} \ \star_S \ \lambda a. \\
\qquad\qquad (\texttt{inAddr} \ (a+1) \ \mathcal{C}[\![e]\!]) \ \star_S \ \lambda \varphi_e : \mathsf{Dynam}(\mathit{int}). \\
\qquad\qquad\qquad\qquad \mathbf{unit}_S(\mathsf{Negate}(\varphi_e, a)) \\
\qquad \text{where: } \mathsf{Negate}(\varphi_e, a) = \ \varphi_e \ \star_D \ \lambda i. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{Alloc}(a) \ \star_D \ \lambda\_. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{Thread}(i, a) \ \star_D \ \lambda v. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{deAlloc}(a) \ \star_D \ \lambda\_. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{unit}_D(-v)
\end{array}$$

$$\mathcal{C}[\![x := e]\!] = \texttt{rdEnv} \ \star_S \ \lambda\rho.(\rho\, x) \ \star_S \ \lambda a.\mathcal{C}[\![e]\!] \ \star_S \ \lambda\varphi_e.\mathbf{unit}_S(\varphi_e \ \star_D \ \lambda i : \mathit{int}.\texttt{store}(a, i))$$

$$\mathcal{C}[\![c_1;c_2]\!] = \mathcal{C}[\![c_1]\!] \ \star_S \ \lambda\varphi_{c_1}.\mathcal{C}[\![c_2]\!] \ \star_S \ \lambda\varphi_{c_2}.\mathbf{unit}_S(\varphi_{c_1} \ \star_D \ \lambda\_.\varphi_{c_2})$$

$$\begin{array}{l}
\mathcal{C}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!] = \ \texttt{rdEnv} \ \star_S \ \lambda\rho. \\
\qquad\qquad\qquad\quad \texttt{rdAddr} \ \star_S \ \lambda a. \\
\qquad\qquad\qquad\qquad [\texttt{inEnv} \ (\rho[x \mapsto \mathbf{unit}_S(a)]) \ (\texttt{inAddr} \ (a+1) \ \mathcal{C}[\![c]\!])] \ \star_S \ \lambda\varphi_c : \mathsf{Dynam}(\texttt{void}). \\
\qquad\qquad\qquad\qquad\quad \mathbf{unit}_S(\texttt{Alloc}(a) \ \star_D \ \lambda\_.\varphi_c \ \star_D \ \lambda\_.\texttt{deAlloc}(a))
\end{array}$$

$$\mathcal{C}[\![x_{rval}]\!] = \texttt{rdEnv} \ \star_S \ \lambda\rho.(\rho\, x) \ \star_S \ \lambda a.\ \mathbf{unit}_S(\texttt{read}(a))$$

$$\begin{array}{l}
\mathcal{C}[\![e_1 \mathsf{leq} e_2]\!] = \texttt{rdAddr} \ \star_S \ \lambda a.\texttt{inAddr} \ (a+2) \ (\mathcal{C}[\![e_1]\!] \ \star_S \ \lambda\varphi_1 \mathcal{C}[\![e_2]\!] \ \star_S \ \lambda\varphi_2.\mathbf{unit}_S(\mathsf{Lteq}(\varphi_1, \varphi_2, a))) \\
\qquad \text{where: } \mathsf{Lteq}(\varphi_1, \varphi_2, a) = \ \varphi_1 \ \star_D \ \lambda i. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \varphi_2 \ \star_D \ \lambda j. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{Alloc}(a) \ \star_D \ \lambda\_. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{Alloc}(a+1) \ \star_D \ \lambda\_. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{Thread}(i, a) \ \star_D \ \lambda v_1. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{Thread}(i, a+1) \ \star_D \ \lambda v_2. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{deAlloc}(a) \ \star_D \ \lambda\_. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{deAlloc}(a+1) \ \star_D \ \lambda\_. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{unit}_D(\lambda\langle\kappa_T, \kappa_F\rangle.(v_1 \leq v_2 \ \to \ \kappa_T, \kappa_F))
\end{array}$$

$$
\begin{aligned}
\mathcal{C}[\![\textbf{if } b \textbf{ then } c]\!] = \; & \texttt{newlabel} \star_S \; \lambda L_\kappa . \\
& \texttt{newlabel} \star_S \; \lambda L_c . \\
& \mathcal{C}[\![b]\!] \; \star_S \; \lambda \varphi_b . \\
& \mathcal{C}[\![c]\!] \; \star_S \; \lambda \varphi_c . \\
& \qquad \mathbf{unit}_S(\mathsf{IfThenPS}(\varphi_b, \varphi_c, L_c, L_\kappa)) \\
\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \; & \texttt{newlabel} \star_S \; \lambda L_\kappa . \\
& \texttt{newlabel} \star_S \; \lambda L_c . \\
& \texttt{newlabel} \star_S \; \lambda L_{test} . \\
& \mathcal{C}[\![b]\!] \; \star_S \; \lambda \varphi_b . \\
& \mathcal{C}[\![c]\!] \; \star_S \; \lambda \varphi_c . \\
& \qquad \mathbf{unit}_S(\mathsf{WhilePS}(\varphi_b, \varphi_c, L_{test}, L_c, L_\kappa))
\end{aligned}
$$

A.4 Code Generator compile for Src

$$\mathsf{compile} : \mathsf{Src} \to \mathsf{Static}(\mathsf{TargetLang})$$

$$\mathsf{compile}(i) = \mathbf{unit}\langle \texttt{NOP}, \texttt{\#}i, \{\,\}\rangle$$

$$\mathsf{compile}(-e) =$$

$$\texttt{rdAddr} \star \lambda a.\texttt{inAddr}\,(a+1) \left(\begin{array}{l} (\mathsf{compile}\, e) \;\star\; \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle. \\ \quad \mathbf{unit}\langle \pi_e \,;\, \texttt{ALLOC}\, a \,;\, a{:=}\mathit{rhs}_e \,;\, (\mathsf{pop}\ \mathit{tmps}_e), -a, \{a\}\rangle \end{array} \right)$$

$$\begin{array}{rl} \mathsf{compile}(x := e) = & \texttt{rdEnv} \star \lambda\rho. \\ & (\rho\, x) \;\star\; \lambda a_x. \\ & \mathsf{compile}(e) \;\star\; \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle. \\ & \;\mathbf{unit}(\pi_e \;;\; a_x{:=}\mathit{rhs}_e \;;\; (\mathsf{pop}\ \mathit{tmps}_e)) \end{array}$$

$$\mathsf{compile}(c_1;c_2) = \mathsf{compile}(c_1) \;\star\; \lambda\pi_1.\mathsf{compile}(c_2) \;\star\; \lambda\pi_2.\mathbf{unit}(\pi_1 \;;\; \pi_2)$$

$$\begin{array}{rl} \mathsf{compile}(\texttt{new}\, x\, \texttt{in}\, c) = & \texttt{rdAddr} \;\star\; \lambda a.\texttt{rdEnv} \;\star\; \lambda\rho. \\ & \quad [\texttt{inAddr}\,(a{+}1)\ (\texttt{inEnv}\,\rho[x \mapsto (\mathbf{unit}\, a)]\ (\mathsf{compile}\, c))] \;\star\; \lambda\pi_c. \\ & \quad\; \mathbf{unit}(\texttt{ALLOC}(a) \;;\; \pi_c \;;\; \texttt{DEALLOC}(a)) \end{array}$$

$$\mathsf{compile}(x) = \texttt{rdEnv} \;\star\; \lambda\rho.(\rho\ x) \;\star\; \lambda a.\mathbf{unit}\langle \texttt{NOP}, a, \{\}\rangle$$

$$\begin{array}{rl} \mathsf{compile}(e_1\ \mathsf{leq}\ e_2) = & \texttt{rdAddr} \star \lambda a. \\ & \texttt{inAddr}\ (a+2) \\ & \;\; \mathsf{compile}(e_1) \;\star\; \lambda\langle \pi_1, \mathit{rhs}_1, \mathit{tmps}_1\rangle. \\ & \;\; \mathsf{compile}(e_2) \;\star\; \lambda\langle \pi_2, \mathit{rhs}_2, \mathit{tmps}_2\rangle. \\ & \;\;\; \mathbf{unit} \left(\begin{array}{l} \pi_1 \;;\; \texttt{ALLOC}(a) \;;\; a{:=}\mathit{rhs}_1 \;;\; \pi_2 \;;\; \texttt{ALLOC}(a+1) \;; \\ \; (a{+}1){:=}\mathit{rhs}_2 \;;\; (\mathsf{pop}\ \mathit{tmps}_1) \;;\; (\mathsf{pop}\ \mathit{tmps}_2) \;;\; \texttt{BRLEQ}\, a\, (a{+}1)) \end{array} \right) \end{array}$$

$$
\begin{array}{l}
\mathsf{compile}(\textbf{if } b \textbf{ then } c) = \\
\quad \texttt{newlabel} \ \star \ \lambda L_{\text{exit}}. \\
\quad \texttt{newlabel} \ \star \ \lambda L_c. \\
\quad \mathsf{compile}(b) \ \star \ \lambda \pi_b. \\
\quad \mathsf{compile}(c) \ \star \ \lambda \pi_c. \\
\qquad \textbf{unit}(\texttt{ENDLABEL}\ L_{\text{exit}} \ \ (\texttt{SEGM}[L_c, \pi_c \ ; \ \texttt{JUMP}\ L_{\text{exit}}] \ ; \ (\pi_b \diamond \langle \texttt{JUMP}\ L_c, \texttt{JUMP}\ L_{\text{exit}} \rangle)))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{compile}(\textbf{while } b \textbf{ do } c) = \\
\quad \texttt{newlabel} \ \star \ \lambda L_{test}. \\
\quad \texttt{newlabel} \ \star \ \lambda L_c. \\
\quad \texttt{newlabel} \ \star \ \lambda L_{exit}. \\
\quad (\mathsf{compile}\, b) \ \star \ \lambda \pi_b. \\
\quad (\mathsf{compile}\, c) \ \star \ \lambda \pi_c. \\
\qquad \textbf{unit}(\texttt{ENDLABEL}\ L_{exit} \left(\begin{array}{l}
\texttt{SEGM}[L_c, \pi_c \ ; \ \texttt{JUMP}\ L_{test}] \ ; \\
\texttt{SEGM}[L_{test}, \pi_b \diamond \langle \texttt{JUMP} L_c, \texttt{JUMP} L_{exit} \rangle] \ ; \\
\texttt{JUMP} L_{test}
\end{array} \right))
\end{array}
$$

A.5 Semantics $\mathcal{M}[\![-]\!]$ for Target Language TargetLang

$$\mathcal{M}[\![-]\!] : \mathsf{TargetLang} \rightarrow \mathsf{Dynam}(\mathtt{void} + \mathit{int} + \mathbf{bool})$$

$$\mathcal{M}[\![\mathtt{NOP}]\!] = \mathbf{unit}_D(\bullet)$$

$$\mathcal{M}[\![a{:=}\mathit{rhs}]\!] = \mathcal{M}[\![\mathit{rhs}]\!] \ \star_D \ \lambda i : \mathit{int}.\mathtt{store}(a, i)$$

$$\mathcal{M}[\![\pi_1 \ \mathtt{;} \ \ldots \ \mathtt{;} \ \pi_n]\!] = \mathcal{M}[\![\pi_1]\!] \ \star_D \ \lambda\_. \ldots \ \star_D \ \lambda\_.\mathcal{M}[\![\pi_n]\!]$$

$$\mathcal{M}[\![\mathtt{ALLOC}(a)]\!] = \mathtt{Alloc}(a)$$

$$\mathcal{M}[\![\mathtt{DEALLOC}(a)]\!] = \mathtt{deAlloc}(a)$$

$$\mathcal{M}[\![\langle \pi, \mathit{rhs}, \mathit{tmps} \rangle]\!] = \mathcal{M}[\![\pi]\!] \ \star_D \ \lambda\_. \ \mathcal{M}[\![\mathit{rhs}]\!] \ \star_D \ \lambda i : \mathit{int}. \ \mathcal{M}[\![(\mathsf{pop} \ \mathit{tmps})]\!] \ \star_D \ \lambda\_.\mathbf{unit}_D(i)$$

$$\mathcal{M}[\![\mathtt{SEGM(}L, \pi\mathtt{)}]\!] = \mathtt{updateCode}[L \mapsto \mathcal{M}[\![\pi]\!]]$$

$$\mathcal{M}[\![\mathtt{JUMP} \ \ L]\!] = \mathtt{jump} \ L$$

$$\mathcal{M}[\![\mathtt{ENDLABEL \ L} \ \pi]\!] = \mathtt{callcc} \ (\lambda\kappa.\mathtt{updateCode}[L \mapsto \kappa\bullet] \ \star_D \ \lambda\_.\mathcal{M}[\![\pi]\!])$$

$$\begin{aligned}
\mathcal{M}[\![\mathtt{BRLEQ} \ a \ a']\!] = \ & \mathtt{read}(a) \ \star_D \ \lambda v. \\
& \mathtt{read}(a') \ \star_D \ \lambda v'. \\
& \mathtt{deAlloc}(a) \ \star_D \ \lambda\_. \\
& \mathtt{deAlloc}(a') \ \star_D \ \lambda\_. \\
& \quad \mathbf{unit}_D(\lambda\langle x, y\rangle.v \leq v' \ \rightarrow \ x, y)
\end{aligned}$$

$$\mathcal{M}[\![\diamond \ \pi_b \ \langle \pi_T, \pi_F \rangle]\!] = \mathcal{M}[\![\pi_b]\!] \ \star_D \ \lambda b.b \, \langle \mathcal{M}[\![\pi_T]\!], \mathcal{M}[\![\pi_F]\!] \rangle$$

Appendix B

Proofs from Chapter 4

B.1 Proof of Theorem 2

Theorem 2 (Equivalence of Standard Staged Semantics) *In the monad* $\mathsf{M} = \mathsf{Static{+}Dynam}$*,*

$$[\![t]\!] = \texttt{unquote}\,\mathcal{S}[\![t]\!]$$

where $\texttt{unquote}\,x = x \;\star_M\; id$.

$\boxed{\text{Proof of Theorem 2}}$

By induction on t.

$$\boxed{\text{Case: } t \text{ is constant } i}$$

$$\texttt{unquote}\,\mathcal{S}[\![i]\!] = (\mathbf{unit}_M(\mathbf{unit}_M(i))) \;\star_M\; id =_{\text{left unit}} \mathbf{unit}_M(i)$$

$$\boxed{\text{Case: } t \text{ is negation } -e}$$

$$\begin{aligned}\texttt{unquote}\,\mathcal{S}[\![-e]\!] &= (\mathcal{S}[\![-e]\!]) \;\star_M\; id \\ &= (\mathcal{S}[\![e]\!] \;\star_M\; \lambda\varphi_e.\, \mathbf{unit}_M(\varphi_e \;\star_M\; \lambda i.\mathbf{unit}_M(-i))) \;\star_M\; id\end{aligned}$$

$\boxed{\text{By associativity:}}$

$$= \mathcal{S}[\![e]\!] \;\star_M\; \lambda\varphi_e.\, [\mathbf{unit}_M(\varphi_e \;\star_M\; \lambda i.\mathbf{unit}_M(-i)) \;\star_M\; id]$$

By left unit:

$$= \mathcal{S}\llbracket e \rrbracket \star_M \lambda\varphi_e.\ [\varphi_e \star_M \lambda i.\mathbf{unit}_M(-i)]$$

By associativity:

$$= [\mathcal{S}\llbracket e \rrbracket \star_M \lambda\varphi_e.\ \varphi_e] \star_M \lambda i.\mathbf{unit}_M(-i)$$

By induction hypothesis:

$$= \llbracket e \rrbracket \star_M \lambda i.\mathbf{unit}_M(-i) = \llbracket -e \rrbracket$$

Case: t is variable x

$$\texttt{unquote}\ \mathcal{S}\llbracket x \rrbracket = [\texttt{rdEnv} \star_M \lambda\rho.(\rho\, x) \star_M \lambda a : \mathit{Addr}.\mathbf{unit}_M(\texttt{read}(a))] \star_M \mathit{id}$$

By associativity:

$$= \texttt{rdEnv} \star_M \lambda\rho.(\rho\, x) \star_M \lambda a.[\mathbf{unit}_M(\texttt{read}(a)) \star_M \mathit{id}]$$

By left unit:

$$= \texttt{rdEnv} \star_M \lambda\rho.(\rho\, x) \star_M \lambda a.\texttt{read}(a) \ = \ \llbracket x \rrbracket$$

Case: t is the command $c_1\ ;\ c_2$

$$\texttt{unquote}(\mathcal{S}\llbracket c_1\ ;\ c_2 \rrbracket)$$

Unfolding definitions of `unquote` and $\mathcal{S}\llbracket c_1\ ;\ c_2 \rrbracket$:

$$= [\mathcal{S}\llbracket c_1 \rrbracket \star_M \lambda\varphi_1.\mathcal{S}\llbracket c_2 \rrbracket \star_M \lambda\varphi_2.\ \mathbf{unit}_M(\varphi_1 \star_D \lambda\_.\varphi_2)] \star_M \mathit{id}$$

By associativity:

$$= \mathcal{S}\llbracket c_1 \rrbracket \star_M \lambda\varphi_1.\mathcal{S}\llbracket c_2 \rrbracket \star_M \lambda\varphi_2.\ [\mathbf{unit}_M(\varphi_1 \star_D \lambda\_.\varphi_2) \star_M \mathit{id}]$$

By left unit:

$$= \mathcal{S}\llbracket c_1 \rrbracket \star_M \lambda\varphi_1.\mathcal{S}\llbracket c_2 \rrbracket \star_M \lambda\varphi_2.\ \varphi_1 \star_D \lambda\_.\varphi_2$$

Because $\mathcal{S}\llbracket c_2 \rrbracket$ is innocent:

$$= \mathcal{S}\llbracket c_1 \rrbracket \ \star_M \ \lambda\varphi_1.\varphi_1 \ \star_D \ \lambda\_.\mathcal{S}\llbracket c_2 \rrbracket \ \star_M \ \lambda\varphi_2.\varphi_2$$

By associativity:

$$= (\mathcal{S}\llbracket c_1 \rrbracket \ \star_M \ \lambda\varphi_1.\varphi_1) \ \star_D \ \lambda\_.(\mathcal{S}\llbracket c_2 \rrbracket \ \star_M \ \lambda\varphi_2.\varphi_2)$$

By induction hypothesis:

$$= \llbracket c_1 \rrbracket \ \star_D \ \lambda\_.\llbracket c_2 \rrbracket = \llbracket c_1 \ ; \ c_2 \rrbracket$$

Case: t is the command $x := e$

$$\texttt{unquote}\ \mathcal{S}\llbracket x := e \rrbracket =$$

$$[\texttt{rdEnv} \ \star_M \ \lambda\rho.(\rho\, x) \ \star_M \ \lambda a : \mathit{Addr}.\ \mathcal{S}\llbracket e \rrbracket \ \star_M \ \lambda\varphi_e.\ \mathbf{unit}_M(\varphi_e \ \star_M \ \lambda i.\texttt{store}(a, i))] \ \star_M \ \mathit{id}$$

By associativity:

$$= \texttt{rdEnv} \ \star_M \ \lambda\rho.(\rho\, x) \ \star_M \ \lambda a.\ [\mathcal{S}\llbracket e \rrbracket \ \star_M \ \lambda\varphi_e.\ \mathbf{unit}_M(\varphi_e \ \star_M \ \lambda i.\texttt{store}(a, i))] \ \star_M \ \mathit{id}$$

By associativity:

$$= \texttt{rdEnv} \ \star_M \ \lambda\rho.(\rho\, x) \ \star_M \ \lambda a.\ \mathcal{S}\llbracket e \rrbracket \ \star_M \ \lambda\varphi_e.\ [\mathbf{unit}_M(\varphi_e \ \star_M \ \lambda i.\texttt{store}(a, i)) \ \star_M \ \mathit{id}]$$

By left unit:

$$= \texttt{rdEnv} \ \star_M \ \lambda\rho.(\rho\, x) \ \star_M \ \lambda a.\ \mathcal{S}\llbracket e \rrbracket \ \star_M \ \lambda\varphi_e.\ [\varphi_e \ \star_M \ \lambda i.\texttt{store}(a, i)]$$

By associativity:

$$= \texttt{rdEnv} \ \star_M \ \lambda\rho.(\rho\, x) \ \star_M \ \lambda a.\ [\mathcal{S}\llbracket e \rrbracket \ \star_M \ \lambda\varphi_e.\ \varphi_e] \ \star_M \ \lambda i.\texttt{store}(a, i)$$

By inductive hypothesis:

$$= \texttt{rdEnv} \ \star_M \ \lambda\rho.(\rho\, x) \ \star_M \ \lambda a.\ \llbracket e \rrbracket \ \star_M \ \lambda i.\texttt{store}(a, i)$$

$$= \llbracket x := e \rrbracket$$

Case: t is $e_1 \ \mathsf{leq} \ e_2$

$\texttt{unquote}\,\mathcal{S}[\![e_1 \ \mathsf{leq}\ e_2]\!]$

Unfolding the definitions of `unquote` and $\mathcal{S}[\![e_1 \ \mathsf{leq}\ e_2]\!]$:

$$= \mathcal{S}[\![e_1 \ \mathsf{leq}\ e_2]\!] \ \star\ id$$

$$= \left(\begin{array}{ll} \multicolumn{2}{l}{\mathcal{S}[\![e_1]\!] \ \star\ \lambda\varphi_1 : \mathsf{M}(int).} \\ \multicolumn{2}{l}{\mathcal{S}[\![e_2]\!] \ \star\ \lambda\varphi_2 : \mathsf{M}(int).} \\ \mathbf{unit} & \left(\begin{array}{l} \varphi_1 \ \star\ \lambda v_1 : int. \\ \varphi_2 \ \star\ \lambda v_2 : int. \\ \quad \mathbf{unit}(\lambda\langle\kappa_T,\kappa_F\rangle.(v_1 \leq v_2 \rightarrow \kappa_T,\kappa_F)) \end{array} \right) \end{array} \right) \star\ id$$

By associativity:

$$\begin{array}{lll} = & \multicolumn{2}{l}{\mathcal{S}[\![e_1]\!] \ \star\ \lambda\varphi_1.} \\ & \multicolumn{2}{l}{\mathcal{S}[\![e_2]\!] \ \star\ \lambda\varphi_2.} \\ & \mathbf{unit} & \left(\begin{array}{l} \varphi_1 \ \star\ \lambda v_1. \\ \varphi_2 \ \star\ \lambda v_2. \\ \quad \mathbf{unit}(\lambda\langle\kappa_T,\kappa_F\rangle.(v_1 \leq v_2 \rightarrow \kappa_T,\kappa_F)) \end{array} \right) \star\ id \end{array}$$

By left unit:

$$= \mathcal{S}[\![e_1]\!] \ \star\ \lambda\varphi_1.\mathcal{S}[\![e_2]\!] \ \star\ \lambda\varphi_2.\varphi_1 \ \star\ \lambda v_1.\varphi_2 \ \star\ \lambda v_2.\mathbf{unit}(\lambda\langle\kappa_T,\kappa_F\rangle.(v_1 \leq v_2 \rightarrow \kappa_T,\kappa_F))$$

Because $\mathcal{S}[\![e_2]\!]$ is innocent:

$$= \mathcal{S}[\![e_2]\!] \ \star\ \lambda\varphi_2.\mathcal{S}[\![e_1]\!] \ \star\ \lambda\varphi_1.\varphi_1 \ \star\ \lambda v_1.\varphi_2 \ \star\ \lambda v_2.\mathbf{unit}(\lambda\langle\kappa_T,\kappa_F\rangle.(v_1 \leq v_2 \rightarrow \kappa_T,\kappa_F))$$

By associativity:

$$= \mathcal{S}[\![e_2]\!] \ \star\ \lambda\varphi_2.(\mathcal{S}[\![e_1]\!] \ \star\ \lambda\varphi_1.\varphi_1) \ \star\ \lambda v_1.\varphi_2 \ \star\ \lambda v_2.\mathbf{unit}(\lambda\langle\kappa_T,\kappa_F\rangle.(v_1 \leq v_2 \rightarrow \kappa_T,\kappa_F))$$

By the induction hypothesis:

$$= \mathcal{S}\llbracket e_2 \rrbracket \ \star\ \lambda\varphi_2.\llbracket e_1 \rrbracket \ \star\ \lambda v_1.\varphi_2 \ \star\ \lambda v_2.\mathbf{unit}(\lambda\langle\kappa_T,\kappa_F\rangle.(v_1 \leq v_2 \to \kappa_T,\kappa_F))$$

$\boxed{\text{Because } \llbracket e_1 \rrbracket \text{ is innocent:}}$

$$= \llbracket e_1 \rrbracket \ \star\ \lambda v_1.\mathcal{S}\llbracket e_2 \rrbracket \ \star\ \lambda\varphi_2.\varphi_2 \ \star\ \lambda v_2.\mathbf{unit}(\lambda\langle\kappa_T,\kappa_F\rangle.(v_1 \leq v_2 \to \kappa_T,\kappa_F))$$

$\boxed{\text{By associativity:}}$

$$= \llbracket e_1 \rrbracket \ \star\ \lambda v_1.(\mathcal{S}\llbracket e_2 \rrbracket \ \star\ \lambda\varphi_2.\varphi_2) \ \star\ \lambda v_2.\mathbf{unit}(\lambda\langle\kappa_T,\kappa_F\rangle.(v_1 \leq v_2 \to \kappa_T,\kappa_F))$$

$\boxed{\text{By the induction hypothesis:}}$

$$= \llbracket e_1 \rrbracket \ \star\ \lambda v_1.\llbracket e_2 \rrbracket \ \star\ \lambda v_2.\mathbf{unit}(\lambda\langle\kappa_T,\kappa_F\rangle.(v_1 \leq v_2 \to \kappa_T,\kappa_F))$$

$$= \llbracket e_1 \ \textsf{leq}\ e_2 \rrbracket$$

$\boxed{\text{Case: } t \text{ is the command: } \mathbf{new}\ x\ \mathbf{in}\ c}$

$$\texttt{unquote}\ \mathcal{S}\llbracket \mathbf{new}\ x\ \mathbf{in}\ c \rrbracket$$

$$= \left[\begin{array}{l} \left(\begin{array}{l} \texttt{rdAddr} \star_M \lambda a. \\ \quad \texttt{inAddr}\,(a+1) \left(\begin{array}{l} \texttt{rdEnv} \star_M \lambda\rho. \\ \texttt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_M(a)])\ \mathcal{S}\llbracket c \rrbracket \end{array}\right) \end{array}\right) \star_M \lambda\varphi_c. \\ \quad \mathbf{unit}_M(\texttt{Alloc}(a) \star_M \lambda\_.\varphi_c \star_M \lambda\_.\texttt{deAlloc}(a)) \end{array}\right] \star_M id$$

$\boxed{\text{By associativity:}}$

$$= \begin{array}{l} \left(\begin{array}{l} \texttt{rdAddr} \star_M \lambda a. \\ \quad \texttt{inAddr}\,(a+1) \left(\begin{array}{l} \texttt{rdEnv} \star_M \lambda\rho. \\ \texttt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_M(a)])\ \mathcal{S}\llbracket c \rrbracket \end{array}\right) \end{array}\right) \star_M \lambda\varphi_c. \\ \quad [\mathbf{unit}_M(\texttt{Alloc}(a) \star_M \lambda\_.\varphi_c \star_M \lambda\_.\texttt{deAlloc}(a)) \star_M id] \end{array}$$

$\boxed{\text{By left unit:}}$

$$= \begin{array}{l} \left(\begin{array}{l} \texttt{rdAddr} \star_M \lambda a. \\ \quad \texttt{inAddr}\,(a+1) \left(\begin{array}{l} \texttt{rdEnv} \star_M \lambda\rho. \\ \texttt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_M(a)])\ \mathcal{S}\llbracket c \rrbracket \end{array}\right) \end{array}\right) \star_M \lambda\varphi_c. \\ \quad \texttt{Alloc}(a) \star_M \lambda\_.\varphi_c \star_M \lambda\_.\texttt{deAlloc}(a) \end{array}$$

Observe that the computation φ_c produced by $\mathcal{S}[\![c]\!]$ has no calls to `rdEnv`, `rdAddr`, `inAddr`, or `inEnv`, so the scope of $\texttt{inAddr}\,(a+1)$ and $\texttt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_M(a)])$ may be extended over $\texttt{Alloc}(a) \star_M \lambda\_.\varphi_c \star_M \lambda\_.\texttt{deAlloc}(a)$ by Axioms 5 and 6:

$$\begin{aligned}
= \ & \texttt{rdAddr} \star_M \lambda a.\, \texttt{inAddr}\,(a+1) \\
& \quad \texttt{rdEnv} \star_M \lambda\rho.\, \texttt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_M(a)]) \\
& \quad\quad \mathcal{S}[\![c]\!] \star_M \lambda\varphi_c.\texttt{Alloc}(a) \star_M \lambda\_.\varphi_c \star_M \lambda\_.\texttt{deAlloc}(a)
\end{aligned}$$

By Lemma 2:

$$\begin{aligned}
= \ & \texttt{rdAddr} \star_M \lambda a.\, \texttt{inAddr}\,(a+1) \\
& \quad \texttt{rdEnv} \star_M \lambda\rho.\, \texttt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_M(a)]) \\
& \quad\quad \texttt{Alloc}(a) \star_M \lambda\_.\mathcal{S}[\![c]\!] \star_M \lambda\varphi_c.\varphi_c \star_M \lambda\_.\texttt{deAlloc}(a)
\end{aligned}$$

By associativity:

$$\begin{aligned}
= \ & \texttt{rdAddr} \star_M \lambda a.\, \texttt{inAddr}\,(a+1) \\
& \quad \texttt{rdEnv} \star_M \lambda\rho.\, \texttt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_M(a)]) \\
& \quad\quad \texttt{Alloc}(a) \star_M \lambda\_.[\mathcal{S}[\![c]\!] \star_M \lambda\varphi_c.\varphi_c] \star_M \lambda\_.\texttt{deAlloc}(a)
\end{aligned}$$

By induction hypothesis:

$$\begin{aligned}
= \ & \texttt{rdAddr} \star_M \lambda a.\, \texttt{inAddr}\,(a+1) \qquad\qquad = [\![\mathbf{new}\ x\ \mathbf{in}\ c]\!] \\
& \quad \texttt{rdEnv} \star_M \lambda\rho.\, \texttt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_M(a)]) \\
& \quad\quad \texttt{Alloc}(a) \star_M \lambda\_.[\![c]\!] \star_M \lambda\_.\texttt{deAlloc}(a)
\end{aligned}$$

Case: t is the command: **if** b **then** c

$$\texttt{unquote}(\mathcal{S}[\![\mathbf{if}\ b\ \mathbf{then}\ c]\!]) = \begin{pmatrix} \mathcal{S}[\![b]\!] \star_M \lambda\varphi_b : \mathsf{M}(\mathit{bool}). \\ \mathcal{S}[\![c]\!] \star_M \lambda\varphi_c : \mathsf{M}(\texttt{void}). \\ \mathbf{unit}_M(\mathsf{IfThen}(\varphi_b, \varphi_c)) \end{pmatrix} \star_M \lambda\varphi.\varphi$$

By associativity and right unit:

$$\begin{array}{ll}
= \mathcal{S}[\![b]\!] \star_M \ \lambda\varphi_b : \mathsf{M}(\mathit{bool}). & = \mathcal{S}[\![b]\!] \star_M \ \lambda\varphi_b : \mathsf{M}(\mathit{bool}). \\
\quad \mathcal{S}[\![c]\!] \star_M \ \lambda\varphi_c : \mathsf{M}(\mathtt{void}). & \quad \mathcal{S}[\![c]\!] \star_M \ \lambda\varphi_c : \mathsf{M}(\mathtt{void}). \\
\qquad (\mathbf{unit}_M(\mathsf{IfThen}(\varphi_b, \varphi_c)) \star_M \ \lambda\varphi.\varphi) & \qquad \mathsf{IfThen}(\varphi_b, \varphi_c)
\end{array}$$

Definition of IfThen

$$= \mathcal{S}[\![b]\!] \star_M \ \lambda\varphi_b.\ \mathcal{S}[\![c]\!] \star_M \ \lambda\varphi_c.\ \varphi_b \star_M \ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.\beta\langle\varphi_c \star_M \ \kappa, \kappa\,\bullet\rangle$$

Equation 4.6 on page 95:

$$\begin{aligned}
&= \mathcal{S}[\![c]\!] \star_M \ \lambda\varphi_c.\ \mathcal{S}[\![b]\!] \star_M \ \lambda\varphi_b.\ \varphi_b \star_M \ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.\beta\langle\varphi_c \star_M \ \kappa, \kappa\,\bullet\rangle \\
&= \mathcal{S}[\![c]\!] \star_M \ \lambda\varphi_c.\ [\mathcal{S}[\![b]\!] \star_M \ \lambda\varphi_b.\ \varphi_b] \star_M \ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.\beta\langle\varphi_c \star_M \ \kappa, \kappa\,\bullet\rangle \\
&=_{IH} \mathcal{S}[\![c]\!] \star_M \ \lambda\varphi_c.\ [\![b]\!] \star_M \ \lambda\beta.\ \mathtt{callcc}\ \lambda\kappa.\beta\langle\varphi_c \star_M \ \kappa, \kappa\,\bullet\rangle \\
&= [\![b]\!] \star_M \ \lambda\beta.\ \mathcal{S}[\![c]\!] \star_M \ \lambda\varphi_c.\ \mathtt{callcc}\ \lambda\kappa.\beta\langle\varphi_c \star_M \ \kappa, \kappa\,\bullet\rangle
\end{aligned}$$

Because β is parametric, either $\beta\langle\varphi_c \star_M \ \kappa, \kappa\,\bullet\rangle = \varphi_c \star_M \ \kappa$, or $\beta\langle\varphi_c \star_M \ \kappa, \kappa\,\bullet\rangle = \kappa\bullet$.

Case 1. $\beta\langle\varphi_c \star_M \ \kappa, \kappa\,\bullet\rangle = \varphi_c \star_M \ \kappa$

$$\begin{array}{ll}
[\![b]\!] \star_M \ \lambda\beta : \mathit{bool}. & = [\![b]\!] \star_M \ \lambda\beta : \mathit{bool}. \\
\mathcal{S}[\![c]\!] \star_M \ \lambda\varphi_c : \mathsf{M}(\mathtt{void}). & \quad \mathcal{S}[\![c]\!] \star_M \ \lambda\varphi_c : \mathsf{M}(\mathtt{void}). \\
\qquad \mathtt{callcc}\ \lambda\kappa.\beta\langle\varphi_c \star_M \ \kappa, \kappa\,\bullet\rangle & \qquad \mathtt{callcc}\ \lambda\kappa.\varphi_c \star_M \ \kappa
\end{array}$$

From $x = \mathtt{callcc}\ \lambda\kappa.x \star \kappa$ (Axiom 3.4) and the induction hypothesis:

$$= \ [\![b]\!] \star_M \ \lambda\beta : \mathit{bool}.\mathcal{S}[\![c]\!] \star_M \ \lambda\varphi_c.\varphi_c \ = [\![b]\!] \star_M \ \lambda\beta : \mathit{bool}.[\![c]\!]$$

$$\begin{array}{lll} = [\![b]\!] \star_M \lambda\beta : \mathit{bool}. & = [\![b]\!] \star_M \lambda\beta : \mathit{bool}. & = [\![\textbf{if } b \textbf{ then } c]\!] \\ \quad \texttt{callcc}\ \lambda\kappa.[\![c]\!] \star_M \kappa & \quad \texttt{callcc}\ \lambda\kappa. & \\ & \qquad \beta\langle [\![c]\!] \star_M \kappa, \kappa \bullet\rangle & \end{array}$$

Case 2. $\beta\langle \varphi_c \star_M \kappa, \kappa \bullet\rangle = \kappa \bullet$

Because $\mathcal{S}[\![c]\!]$ is innocent, and the first component of $\langle \varphi_c \star_M \kappa, \kappa \bullet\rangle$ is ignored by β:

$$\begin{array}{ll} [\![b]\!] \star_M \lambda\beta : \mathit{bool}. & = [\![b]\!] \star_M \lambda\beta : \mathit{bool}. \\ \mathcal{S}[\![c]\!] \star_M \lambda\varphi_c : \mathsf{M}(\texttt{void}). & \qquad \texttt{callcc}\ \lambda\kappa.\beta\langle [\![c]\!] \star_M \kappa, \kappa \bullet\rangle \\ \quad \texttt{callcc}\ \lambda\kappa.\beta\langle \varphi_c \star_M \kappa, \kappa \bullet\rangle & \end{array}$$

Case: t is the command: **while** b **do** c

First, define the functional $\mathsf{dynwhile}_n$ as:

$$\mathsf{dynwhile}_0\ \varphi_b\ \varphi_c = \perp$$

$$\mathsf{dynwhile}_{n+1}\ \varphi_b\ \varphi_c = \varphi_b \star \lambda\beta.\texttt{callcc}\ \lambda\kappa.\beta\langle \varphi_c \star \lambda\_.(\mathsf{dynwhile}_n\ \varphi_b\ \varphi_c), \kappa\bullet\rangle$$

Claim 1 $\mathsf{dynwhile}_{n+1}\ (\mathcal{S}[\![b]\!] \star_M \lambda\varphi_b.\varphi_b)\ \varphi_c = \mathcal{S}[\![b]\!] \star_M \lambda\varphi_b.\mathsf{dynwhile}_{n+1}\ \varphi_b\ \varphi_c$

Claim 2 *The induction hypothesis implies the following:*

$$\begin{array}{rl} \mathsf{dynwhile}_{n+1}\ [\![b]\!]\ [\![c]\!] = & \mathcal{S}[\![b]\!] \star \lambda\varphi_b.\mathcal{S}[\![c]\!] \star \lambda\varphi_c. \\ & \quad \varphi_b \star \lambda\beta.\texttt{callcc}\ \lambda\kappa.\beta\langle \varphi_c \star \lambda\_.(\mathsf{dynwhile}_n\ \varphi_b\ \varphi_c), \kappa\bullet\rangle \end{array}$$

Given Claim 2, this case follows easily:

$$\begin{aligned}
\llbracket \textbf{while } b \textbf{ do } c \rrbracket &= \mathsf{dynwhile}\ \llbracket b \rrbracket\ \llbracket c \rrbracket \\
&= \bigsqcup_{n \in \omega} \mathsf{dynwhile}_{n+1}\ \llbracket b \rrbracket\ \llbracket c \rrbracket \\
&= \bigsqcup_{n \in \omega} \mathcal{S}\llbracket b \rrbracket \ \star\ \lambda \varphi_b.\, \mathcal{S}\llbracket c \rrbracket \ \star\ \lambda \varphi_c.\, \mathsf{dynwhile}_n(\varphi_b, \varphi_c) \\
&= \mathcal{S}\llbracket b \rrbracket \ \star\ \lambda \varphi_b.\, \mathcal{S}\llbracket c \rrbracket \ \star\ \lambda \varphi_c.\, \mathsf{dynwhile}(\varphi_b, \varphi_c) \\
&= \mathtt{unquote}\ \mathcal{S}\llbracket \textbf{while } b \textbf{ do } c \rrbracket
\end{aligned}$$

Proof of Claim 1 Assume $n \geq 0$.

$$\mathsf{dynwhile}_{n+1}\ (\mathcal{S}\llbracket b \rrbracket \ \star\ \lambda \varphi_b.\varphi_b)\ \varphi_c$$

$$= \mathcal{S}\llbracket b \rrbracket \ \star\ \lambda \varphi_b.\varphi_b \ \star\ \lambda \beta.\mathtt{callcc}\ \lambda \kappa.\beta\langle \varphi_c \ \star\ \lambda\_.(\mathsf{dynwhile}_n\ (\mathcal{S}\llbracket b \rrbracket \ \star\ \lambda \varphi_b.\varphi_b)\ \varphi_c), \kappa \bullet \rangle$$

Assume WLOG that $\beta\langle x, y \rangle = x$ (done otherwise):

$$= \mathcal{S}\llbracket b \rrbracket \ \star\ \lambda \varphi_b.\varphi_b \ \star\ \lambda \beta.\mathtt{callcc}\ \lambda \kappa.\varphi_c \ \star\ \lambda\_.(\mathsf{dynwhile}_n\ (\mathcal{S}\llbracket b \rrbracket \ \star\ \lambda \varphi'_b.\varphi'_b)\ \varphi_c)$$

Inductive hypothesis:

$$= \mathcal{S}\llbracket b \rrbracket \ \star\ \lambda \varphi_b.\varphi_b \ \star\ \lambda \beta.\mathtt{callcc}\ \lambda \kappa.\varphi_c \ \star\ \lambda\_.\mathcal{S}\llbracket b \rrbracket \ \star\ \lambda \varphi'_b.(\mathsf{dynwhile}_n\ \varphi'_b\ \varphi_c)$$

Innocence of $\mathcal{S}\llbracket b \rrbracket$:

$$= \mathcal{S}\llbracket b \rrbracket \ \star\ \lambda \varphi_b.\mathcal{S}\llbracket b \rrbracket \ \star\ \lambda \varphi'_b.\varphi_b \ \star\ \lambda \beta.\mathtt{callcc}\ \lambda \kappa.\varphi_c \ \star\ \lambda\_.(\mathsf{dynwhile}_n\ \varphi'_b\ \varphi_c)$$

Because $\mathcal{S}\llbracket b \rrbracket$ depends only on the *Addr* and *Env* environments, by Axiom 6.2:

$$= \mathcal{S}\llbracket b \rrbracket \ \star\ \lambda \varphi_b.\varphi_b \ \star\ \lambda \beta.\mathtt{callcc}\ \lambda \kappa.\varphi_c \ \star\ \lambda\_.(\mathsf{dynwhile}_n\ \varphi_b\ \varphi_c)$$

$$= \mathcal{S}\llbracket b \rrbracket \ \star\ \lambda \varphi_b.\mathsf{dynwhile}_{n+1}\ \varphi_b\ \varphi_c$$

□Claim 1.

$\boxed{\text{Proof of Claim 2}}$ Assume $n \geq 0$.

$$\mathsf{dynwhile}_{n+1}\ \llbracket b \rrbracket\ \llbracket c \rrbracket$$

$$=_{ih}\ \mathsf{dynwhile}_{n+1}\ (\mathcal{S}\llbracket b \rrbracket\ \star\ id)\ (\mathcal{S}\llbracket c \rrbracket\ \star\ id)$$

$$=\ (\mathcal{S}\llbracket b \rrbracket\ \star\ \lambda\varphi_b.\varphi_b)\ \star\ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.$$
$$\qquad \beta\langle(\mathcal{S}\llbracket c \rrbracket\ \star\ id)\ \star\ \lambda\_.(\mathsf{dynwhile}_n\ (\mathcal{S}\llbracket b \rrbracket\ \star\ \lambda\widehat{\varphi}_b.\widehat{\varphi}_b)\ (\mathcal{S}\llbracket c \rrbracket\ \star\ id)), \kappa\bullet\rangle$$

$$=\ \mathcal{S}\llbracket b \rrbracket\ \star\ \lambda\varphi_b.\varphi_b\ \star\ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.$$
$$\qquad \beta\langle(\mathcal{S}\llbracket c \rrbracket\ \star\ id)\ \star\ \lambda\_.(\mathsf{dynwhile}_n\ (\mathcal{S}\llbracket b \rrbracket\ \star\ \lambda\widehat{\varphi}_b.\widehat{\varphi}_b)\ (\mathcal{S}\llbracket c \rrbracket\ \star\ id)), \kappa\bullet\rangle$$

By Claim 1:

$$=\ \mathcal{S}\llbracket b \rrbracket\ \star\ \lambda\varphi_b.\varphi_b\ \star\ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.$$
$$\qquad \beta\langle(\mathcal{S}\llbracket c \rrbracket\ \star\ id)\ \star\ \lambda\_.\mathcal{S}\llbracket b \rrbracket\ \star\ \lambda\widehat{\varphi}_b.(\mathsf{dynwhile}_n\ \widehat{\varphi}_b\ (\mathcal{S}\llbracket c \rrbracket\ \star\ id)), \kappa\bullet\rangle$$

Because $\mathcal{S}\llbracket b \rrbracket$ is innocent, β is parametric, and Axiom 3.4:

$$=\ \mathcal{S}\llbracket b \rrbracket\ \star\ \lambda\varphi_b.\mathcal{S}\llbracket b \rrbracket\ \star\ \lambda\widehat{\varphi}_b.\varphi_b\ \star\ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.$$
$$\qquad \beta\langle(\mathcal{S}\llbracket c \rrbracket\ \star\ id)\ \star\ \lambda\_.(\mathsf{dynwhile}_n\ \widehat{\varphi}_b\ (\mathcal{S}\llbracket c \rrbracket\ \star\ id)), \kappa\bullet\rangle$$

From Equation 4.7 on page 95:

$$=\ \mathcal{S}\llbracket b \rrbracket\ \star\ \lambda\varphi_b.\varphi_b\ \star\ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.$$
$$\qquad \beta\langle(\mathcal{S}\llbracket c \rrbracket\ \star\ id)\ \star\ \lambda\_.(\mathsf{dynwhile}_n\ \varphi_b\ (\mathcal{S}\llbracket c \rrbracket\ \star\ id)), \kappa\bullet\rangle$$

As with the case of $\mathcal{S}\llbracket b \rrbracket$ above:

$$=\ \mathcal{S}\llbracket b \rrbracket\ \star\ \lambda\varphi_b.\mathcal{S}\llbracket c \rrbracket\ \star\ \lambda\varphi_c.\mathcal{S}\llbracket c \rrbracket\ \star\ \lambda\widehat{\varphi}_c.\varphi_b\ \star\ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.$$
$$\qquad \beta\langle\varphi_c\ \star\ \lambda\_.(\mathsf{dynwhile}_n\ \varphi_b\ \widehat{\varphi}_c), \kappa\bullet\rangle$$

From Equation 4.7 on page 95:

$$= \mathcal{S}\llbracket b \rrbracket\ \star\ \lambda\varphi_b.\mathcal{S}\llbracket c \rrbracket\ \star\ \lambda\varphi_c.\varphi_b\ \star\ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.\beta\langle\varphi_c\ \star\ \lambda\_.(\mathsf{dynwhile}_n\ \varphi_b\ \varphi_c), \kappa\bullet\rangle$$

□Theorem 2 (Equivalence of Standard Staged Semantics).

B.2 Proof of Lemma 3

Lemma 3 *For $\mathcal{F} : int \to \mathsf{M}\, b$,*

$$\mathsf{Obs}(\mathsf{FreshLoc}(a), \mathtt{Alloc}(a) \star \lambda\_.\mathtt{Thread}(i,a) \star \lambda v.\mathtt{deAlloc}(a) \star \lambda\_.\mathcal{F}\, v, \gamma)$$
$$= \mathsf{Obs}(\mathsf{FreshLoc}(a), \mathcal{F}\, i, \gamma)$$

Proof of Lemma 3:

$$\mathsf{Obs}(\mathsf{FreshLoc}(a), \mathtt{Alloc}(a) \star \lambda\_.\mathtt{Thread}(i,a) \star \lambda v.\mathtt{deAlloc}(a) \star \lambda\_.\mathcal{F}\, v, \gamma)$$

$$\begin{aligned}
= \ & \mathtt{getSto} \star \lambda\sigma. \\
& \quad \text{if } \ \forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma) \ \text{ then} \\
& \qquad \mathtt{Alloc}(a) \star \lambda\_.\mathtt{Thread}(i,a) \star \lambda v.\mathtt{deAlloc}(a) \star \lambda\_.\mathcal{F}\, v \\
& \quad \text{else } \ \gamma
\end{aligned}$$

Definition of $\mathtt{Alloc}(a)$:

$$\begin{aligned}
= \ & \mathtt{getSto} \star \lambda\sigma. \\
& \quad \text{if } \ \forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma) \ \text{ then} \\
& \qquad \mathtt{getSto} \star \lambda\sigma_1.(\text{if } a \in \mathsf{dom}(\sigma_1) \text{ then } \ \bot \ \text{ else } \mathtt{updateSto}[\sigma^* \mapsto \sigma^* \cup \{\langle a, 0\rangle\}]) \star \lambda\_. \\
& \qquad \mathtt{Thread}(i,a) \star \lambda v.\mathtt{deAlloc}(a) \star \lambda\_.\mathcal{F}\, v \\
& \quad \text{else } \ \gamma
\end{aligned}$$

Consequence of Axiom 2.3:

$$\begin{aligned}
= \ & \mathtt{getSto} \star \lambda\sigma. \\
& \quad \text{if } \ \forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma) \ \text{ then} \\
& \qquad (\text{if } a \in \mathsf{dom}(\sigma) \text{ then } \ \bot \ \text{ else } \mathtt{updateSto}[\sigma^* \mapsto \sigma^* \cup \{\langle a, 0\rangle\}]) \star \lambda\_. \\
& \qquad \mathtt{Thread}(i,a) \star \lambda v.\mathtt{deAlloc}(a) \star \lambda\_.\mathcal{F}\, v \\
& \quad \text{else } \ \gamma
\end{aligned}$$

$= \texttt{getSto} \star \lambda\sigma.$

if $\forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma)$ then

$\texttt{updateSto}[\sigma^* \mapsto \sigma^* \cup \{\langle a, 0\rangle\}] \star \lambda\_.$

$\texttt{Thread}(i, a) \star \lambda v.\texttt{deAlloc}(a) \star \lambda\_.\mathcal{F}\, v$

else γ

Definition of $\texttt{Thread}(i, a)$:

$= \texttt{getSto} \star \lambda\sigma.$

if $\forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma)$ then

$\texttt{updateSto}[\sigma^* \mapsto \sigma^* \cup \{\langle a, 0\rangle\}] \star \lambda\_.$

$\texttt{store}(a, i) \star \lambda\_.$

$\texttt{read}(a) \star \lambda v.\texttt{deAlloc}(a) \star \lambda\_.\mathcal{F}\, v$

else γ

Definition of $\texttt{store}(a, i)$:

$= \texttt{getSto} \star \lambda\sigma.$

if $\forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma)$ then

$\texttt{updateSto}[\sigma^* \mapsto \sigma^* \cup \{\langle a, 0\rangle\}] \star \lambda\_.$

$\texttt{getSto} \star_D \lambda\sigma_2.$

(if $\langle a, -\rangle \in \sigma_2$ then $\texttt{updateSto}[\sigma^* \mapsto (\sigma^* \setminus \langle a, -\rangle) \cup \langle a, i\rangle]$ else $\bot$) $\star \lambda\_.$

$\texttt{read}(a) \star \lambda v.\texttt{deAlloc}(a) \star \lambda\_.\mathcal{F}\, v$

else γ

Letting $\Delta_1 = [\sigma^* \mapsto \sigma^* \cup \{\langle a, 0\rangle\}]$ and by Axiom 2.5:

$= \texttt{getSto} \star \lambda\sigma.$

if $\forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma)$ then

$\texttt{updateSto}\Delta_1 \star \lambda\_.$

(if $\langle a, -\rangle \in (\Delta_1\sigma_2)$ then $\texttt{updateSto}[\sigma^* \mapsto (\sigma^* \setminus \langle a, -\rangle) \cup \langle a, i\rangle]$ else $\bot$) $\star \lambda\_.$

$\texttt{read}(a) \star \lambda v.\texttt{deAlloc}(a) \star \lambda\_.\mathcal{F}\, v$

else γ

$\forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma)$ implies $\langle a, - \rangle \in [\sigma^* \mapsto \sigma^* \cup \{\langle a, 0 \rangle\}]\sigma$, so simplifying:

$$
\begin{array}{l}
= \texttt{getSto} \ \star \ \lambda\sigma. \\
\quad \text{if } \ \forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma) \ \text{ then} \\
\qquad \texttt{updateSto}\Delta_1 \ \star \ \lambda\_. \\
\qquad \texttt{updateSto}[\sigma^* \mapsto (\sigma^* \setminus \langle a, - \rangle) \cup \langle a, i \rangle] \ \star \ \lambda\_. \\
\qquad \texttt{read}(a) \ \star \ \lambda v.\texttt{deAlloc}(a) \ \star \ \lambda\_.\mathcal{F}\, v \\
\quad \text{else } \ \gamma
\end{array}
$$

Letting $\Delta_2 = [\sigma^* \mapsto (\sigma^* \setminus \langle a, - \rangle) \cup \langle a, i \rangle]$, then by Axiom 2.1:

$$
\begin{array}{l}
= \texttt{getSto} \ \star \ \lambda\sigma. \\
\quad \text{if } \ \forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma) \ \text{ then} \\
\qquad \texttt{updateSto}(\Delta_1 \ ; \Delta_2) \ \star \ \lambda\_. \\
\qquad \texttt{read}(a) \ \star \ \lambda v.\texttt{deAlloc}(a) \ \star \ \lambda\_.\mathcal{F}\, v \\
\quad \text{else } \ \gamma
\end{array}
$$

Definition of $\texttt{read}(a)$:

$$
\begin{array}{l}
= \texttt{getSto} \ \star \ \lambda\sigma. \\
\quad \text{if } \ \forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma) \ \text{ then} \\
\qquad \texttt{updateSto}(\Delta_1 \ ; \Delta_2) \ \star \ \lambda\_. \\
\qquad \texttt{getSto} \ \star \ \lambda\sigma_3.(\text{if } \langle a, v \rangle \in \sigma_3 \text{ then } \mathbf{unit}(v) \text{ else } \ \bot) \ \star \ \lambda v. \\
\qquad \texttt{deAlloc}(a) \ \star \ \lambda\_.\mathcal{F}\, v \\
\quad \text{else } \ \gamma
\end{array}
$$

Definition of $\texttt{read}(a)$:

$$
\begin{array}{l}
= \texttt{getSto} \ \star \ \lambda\sigma. \\
\quad \text{if } \ \forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma) \ \text{ then} \\
\qquad \texttt{updateSto}(\Delta_1 \ ; \Delta_2) \ \star \ \lambda\_. \\
\qquad \texttt{getSto} \ \star \ \lambda\sigma_3.(\text{if } \langle a, v \rangle \in \sigma_3 \text{ then } \mathbf{unit}(v) \text{ else } \ \bot) \ \star \ \lambda v. \\
\qquad \texttt{deAlloc}(a) \ \star \ \lambda\_.\mathcal{F}\, v \\
\quad \text{else } \ \gamma
\end{array}
$$

$\boxed{\text{By Axiom 2.5:}}$

$$\begin{array}{l}
= \mathtt{getSto} \star \lambda\sigma. \\
\quad \text{if } \forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma) \text{ then} \\
\qquad \mathtt{updateSto}(\Delta_1;\Delta_2) \star \lambda\_. \\
\qquad (\text{if } \langle a, v\rangle \in (\Delta_1;\Delta_2)\sigma \text{ then } \mathbf{unit}(v) \text{ else } \bot) \star \lambda v. \\
\qquad \mathtt{deAlloc}(a) \star \lambda\_.\mathcal{F}\,v \\
\quad \text{else } \gamma
\end{array}$$

$$\begin{array}{l}
= \mathtt{getSto} \star \lambda\sigma. \\
\quad \text{if } \forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma) \text{ then} \\
\qquad \mathtt{updateSto}(\Delta_1;\Delta_2) \star \lambda\_. \\
\qquad \mathbf{unit}(i) \star \lambda v. \\
\qquad \mathtt{deAlloc}(a) \star \lambda\_.\mathcal{F}\,v \\
\quad \text{else } \gamma
\end{array}$$

$\boxed{\text{By left unit:}}$

$$\begin{array}{l}
= \mathtt{getSto} \star \lambda\sigma. \\
\quad \text{if } \forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma) \text{ then} \\
\qquad \mathtt{updateSto}(\Delta_1;\Delta_2) \star \lambda\_.\mathtt{deAlloc}(a) \star \lambda\_.\mathcal{F}\,i \\
\quad \text{else } \gamma
\end{array}$$

$\boxed{\text{Letting } \Delta_3 = [\sigma^* \mapsto \sigma^* \setminus \{\langle a, -\rangle\}], \text{ then by Axiom 2.5: :}}$

$$\begin{array}{l}
= \mathtt{getSto} \star \lambda\sigma. \\
\quad \text{if } \forall a^* \geq a.a^* \notin \mathsf{dom}(\sigma) \text{ then} \\
\qquad \mathtt{updateSto}(\Delta_1;\Delta_2) \star \lambda\_. \\
\qquad (\text{if } a \notin (\Delta_1;\Delta_2)\sigma \text{ then } \bot_{\mathsf{Dynam(void)}} \text{ else } \mathtt{updateSto}\Delta_3) \star \lambda\_.\mathcal{F}\,i \\
\quad \text{else } \gamma
\end{array}$$

$\boxed{\text{Simplifying and by Axiom 2.1:}}$

$$\begin{aligned}
= \;& \texttt{getSto} \star \lambda\sigma. \\
& \quad \text{if } \; \forall a^* \geq a . a^* \notin \mathsf{dom}(\sigma) \; \text{ then} \\
& \qquad \texttt{updateSto}(\Delta_1; \Delta_2; \Delta_3) \, \star \, \lambda\_. \, \star \, \lambda\_. \mathcal{F}\, i \\
& \quad \text{else } \; \gamma
\end{aligned}$$

$\boxed{\text{Because } \Delta_1; \Delta_2; \Delta_3 = id\text{:}}$

$$\begin{aligned}
= \;& \texttt{getSto} \star \lambda\sigma. \\
& \quad \text{if } \; \forall a^* \geq a . a^* \notin \mathsf{dom}(\sigma) \; \text{ then} \\
& \qquad \mathcal{F}\, i \\
& \quad \text{else } \; \gamma \\
= \;& \mathsf{Obs}(\mathsf{FreshLoc}(a), \mathcal{F}\, i, \gamma)
\end{aligned}$$

□Lemma 3

B.3 Proof of Lemma 4

Lemma 4 (Discharging FreshLoc After Alloc) *If* $\mathsf{FreshLoc}(a)$*, then* $\mathsf{FreshLoc}(a+1)$ *after* $\mathtt{Alloc}(a)$*:*

$$\begin{aligned}&\mathsf{Obs}(\mathsf{FreshLoc}(a), (\mathtt{Alloc}(a) \ \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLoc}(a+1), x, y) \ \star_D \ \lambda v.z), \gamma)\\ &\qquad = \mathsf{Obs}(\mathsf{FreshLoc}(a), (\mathtt{Alloc}(a) \ \star_D \ \lambda\_.x \ \star_D \ \lambda v.z), \gamma)\end{aligned}$$

Proof of Lemma 4

$\mathsf{Obs}(\mathsf{FreshLoc}(a), (\mathtt{Alloc}(a) \ \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLoc}(a+1), x, y) \ \star_D \ \lambda v.z), \gamma)$

Unfolding the definitions of the leftmost Obs and $\mathsf{FreshLoc}$:

$$\begin{aligned}= &\left(\begin{array}{l}\mathtt{getSto} \ \star_D \ \lambda\sigma.\\ \quad \mathbf{unit}_D(\forall a' \geq a. a' \notin \mathsf{dom}(\sigma))\end{array}\right) \star_D \ \lambda \mathit{fresh}.\\ &\text{if } \ \mathit{fresh} \ \text{ then}\\ &\qquad \mathtt{Alloc}(a) \ \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLoc}(a+1), x, y) \ \star_D \ \lambda v.z\\ &\text{else } \ \gamma\end{aligned}$$

Associativity and left unit:

$$\begin{aligned}= \ &\mathtt{getSto} \ \star_D \ \lambda\sigma.\\ &\quad \text{if } \ \forall a' \geq a. a' \notin \mathsf{dom}(\sigma) \ \text{ then}\\ &\qquad \mathtt{Alloc}(a) \ \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLoc}(a+1), x, y) \ \star_D \ \lambda v.z\\ &\quad \text{else } \ \gamma\end{aligned}$$

Associativity, left unit, and unfolding the definition of `Alloc`:

$$
\begin{array}{l}
= \ \mathtt{getSto} \ \star_D \ \lambda\sigma. \\
\quad \text{if } \ \forall a' \geq a.a' \notin \mathsf{dom}(\sigma) \ \text{ then} \\
\qquad \left(\begin{array}{l}
\mathtt{getSto} \ \star_D \ \lambda\sigma^*. \\
\quad \text{if } \ a \in \mathsf{dom}(\sigma^*) \ \text{ then} \\
\qquad \bot_{\mathsf{Dynam}(\mathtt{void})} \\
\quad \text{else } \ \mathtt{updateSto}[\sigma_0 \mapsto \sigma_0 \cup \{\langle a, 0\rangle\}]
\end{array} \right) \ \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLoc}(a+1), x, y) \ \star_D \ \lambda v.z \\
\quad \text{else } \ \gamma
\end{array}
$$

Because $\mathtt{getSto} \ \star_D \ \lambda\sigma_0.\mathtt{getSto} \ \star_D \ \lambda\sigma_1.\mathbf{unit}_D(\sigma_0 = \sigma_1) = \mathbf{unit}_D(\mathtt{true})$:

$$
\begin{array}{l}
= \ \mathtt{getSto} \ \star_D \ \lambda\sigma. \\
\quad \text{if } \ \forall a' \geq a.a' \notin \mathsf{dom}(\sigma) \ \text{ then} \\
\qquad \left(\begin{array}{l}
\quad \text{if } \ a \in \mathsf{dom}(\sigma) \ \text{ then} \\
\qquad \bot_{\mathsf{Dynam}(\mathtt{void})} \\
\quad \text{else } \ \mathtt{updateSto}[\sigma_0 \mapsto \sigma_0 \cup \{\langle a, 0\rangle]\}
\end{array} \right) \ \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLoc}(a+1), x, y) \ \star_D \ \lambda v.z \\
\quad \text{else } \ \gamma
\end{array}
$$

Because $\forall a' \geq a.a' \notin \mathsf{dom}(\sigma) \Longrightarrow a \notin \mathsf{dom}(\sigma)$:

$$
\begin{array}{l}
= \ \mathtt{getSto} \ \star_D \ \lambda\sigma. \\
\quad \text{if } \ \forall a' \geq a.a' \notin \mathsf{dom}(\sigma) \ \text{ then} \\
\qquad \mathtt{updateSto}[\sigma_0 \mapsto \sigma_0 \cup \{\langle a, 0\rangle\}] \ \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLoc}(a+1), x, y) \ \star_D \ \lambda v.z \\
\quad \text{else } \ \gamma
\end{array}
$$

Unfolding the definitions of the remaining Obs and $\mathsf{FreshLoc}$:

$$\begin{array}{l} = \ \texttt{getSto} \ \star_D \ \lambda\sigma. \\ \quad \text{if } \ \forall a' \geq a. a' \notin \mathsf{dom}(\sigma) \ \text{ then} \\ \quad\quad \texttt{updateSto}[\sigma_0 \mapsto \sigma_0 \cup \{\langle a,0\rangle\}] \ \star_D \ \lambda\_. \\ \quad\quad \left(\begin{array}{l}\texttt{getSto} \ \star_D \ \lambda\sigma'. \\ \quad \mathbf{unit}_D(\forall a' \geq a+1. a' \notin \mathsf{dom}(\sigma'))\end{array}\right) \star_D \ \lambda \mathit{fresh}'. \\ \quad\quad\quad (\text{if } \mathit{fresh}' \text{ then } x \text{ else } y) \ \star_D \ \lambda v.z \\ \quad \text{else } \ \gamma \end{array}$$

Because $\texttt{getSto} \star \lambda\sigma.\texttt{updateSto}\Delta \star \lambda\_.\texttt{getSto} \star f = \texttt{getSto} \star \lambda\sigma.\texttt{updateSto}\Delta \star \lambda\_.f(\Delta\sigma)$:

$$\begin{array}{l} = \ \texttt{getSto} \ \star_D \ \lambda\sigma. \\ \quad \text{if } \ \forall a' \geq a. a' \notin \mathsf{dom}(\sigma) \ \text{ then} \\ \quad\quad \texttt{updateSto}[\sigma_0 \mapsto \sigma_0 \cup \{\langle a,0\rangle\}] \ \star_D \ \lambda\_. \\ \quad\quad\quad \mathbf{unit}_D(\forall a' \geq a+1. a' \notin \mathsf{dom}(\sigma \cup \{\langle a,0\rangle\})) \ \star_D \ \lambda \mathit{fresh}'. \\ \quad\quad\quad (\text{if } \mathit{fresh}' \text{ then } x \text{ else } y) \ \star_D \ \lambda v.z \\ \quad \text{else } \ \gamma \end{array}$$

Left unit:

$$\begin{array}{l} = \ \texttt{getSto} \ \star_D \ \lambda\sigma. \\ \quad \text{if } \ \forall a' \geq a. a' \notin \mathsf{dom}(\sigma) \ \text{ then} \\ \quad\quad \texttt{updateSto}[\sigma_0 \mapsto \sigma_0 \cup \{\langle a,0\rangle\}] \ \star_D \ \lambda\_. \\ \quad\quad\quad (\text{if } \forall a' \geq a+1. a' \notin \mathsf{dom}(\sigma \cup \{\langle a,0\rangle\}) \text{ then } x \text{ else } y) \ \star_D \ \lambda v.z \\ \quad \text{else } \ \gamma \end{array}$$

$\boxed{\text{Because } \forall a' \geq a.a' \notin \mathsf{dom}(\sigma) \Longrightarrow \forall a' \geq a+1.a' \notin \mathsf{dom}(\sigma \cup \{\langle a, 0\rangle\}):}$

$$
\begin{array}{ll}
= & \texttt{getSto} \star_D \lambda\sigma. \\
& \quad \text{if } \forall a' \geq a.a' \notin \mathsf{dom}(\sigma) \text{ then} \\
& \qquad \texttt{updateSto}[\sigma_0 \mapsto \sigma_0 \cup \{\langle a, 0\rangle\}] \star_D \lambda\_.x \star_D \lambda v.z \\
& \quad \text{else } \gamma
\end{array}
$$

$\boxed{\text{Working backwards:}}$

$$= \mathsf{Obs}(\mathsf{FreshLoc}(a), (\texttt{Alloc}(a) \star_D \lambda\_.x \star_D \lambda v.z), \gamma)$$

B.4 Proof of Lemma 5

Lemma 5 (FreshLoc and `updateCode` Interactions)

$$\begin{aligned}&\mathsf{Obs}(\mathsf{FreshLoc}(a), \mathtt{updateCode}(f) \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLoc}(a), x, y) \star_D \lambda v.z, \gamma)\\ &\qquad = \mathsf{Obs}(\mathsf{FreshLoc}(a), \mathtt{updateCode}(f) \star_D \lambda\_.x \star_D \lambda v.z, \gamma)\end{aligned}$$

$\boxed{\text{Proof of Lemma 5:}}$

$$\mathsf{Obs}(\mathsf{FreshLoc}(a), \mathtt{updateCode}(f) \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLoc}(a), x, y) \star_D \lambda v.z, \gamma)$$

$\boxed{\text{Unfolding definition of } \mathsf{Obs}\text{:}}$

$$= \mathsf{Obs}\left(\begin{array}{l}\mathsf{FreshLoc}(a),\\ \quad\left(\begin{array}{l}\mathtt{updateCode}(f) \star_D \lambda\_.\\ \mathsf{FreshLoc}(a) \star_D \lambda test.\\ \text{ if } test \text{ then } x \text{ else } y\end{array}\right) \star_D \lambda v.z,\\ \quad\gamma\end{array}\right)$$

$\boxed{\text{Unfolding definition of } \mathsf{FreshLoc}\text{:}}$

$$= \mathsf{Obs}\left(\begin{array}{l}\mathsf{FreshLoc}(a),\\ \quad\left(\begin{array}{l}\mathtt{updateCode}(f) \star_D \lambda\_.\\ \mathtt{getSto} \star_D \lambda\sigma.\\ \mathbf{unit}_D(\forall x \geq a.x \notin Dom(\sigma)) \star_D \lambda test.\\ \text{ if } test \text{ then } x \text{ else } y\end{array}\right) \star_D \lambda v.z,\\ \quad\gamma\end{array}\right)$$

$\boxed{\text{By Axiom 5.4:}}$

$$= \mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLoc}(a), \\ \quad \left(\begin{array}{l} \texttt{getSto} \ \star_D \ \lambda\sigma. \\ \texttt{updateCode}(f) \ \star_D \ \lambda\_. \\ \mathbf{unit}_D(\forall x \geq a.x \notin \mathit{Dom}(\sigma)) \ \star_D \ \lambda \mathit{test}. \\ \ \text{if } \mathit{test} \text{ then } x \text{ else } y \end{array}\right) \star_D \ \lambda v.z, \\ \quad \gamma \end{array}\right)$$

$\boxed{\text{By Axiom 4.1:}}$

$$= \mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLoc}(a), \\ \quad \left(\begin{array}{l} \texttt{getSto} \ \star_D \ \lambda\sigma. \\ \mathbf{unit}_D(\forall x \geq a.x \notin \mathit{Dom}(\sigma)) \ \star_D \ \lambda \mathit{test}. \\ \texttt{updateCode}(f) \ \star_D \ \lambda\_. \\ \ \text{if } \mathit{test} \text{ then } x \text{ else } y \end{array}\right) \star_D \ \lambda v.z, \\ \quad \gamma \end{array}\right)$$

$\boxed{\text{Folding definition of } \mathsf{FreshLoc}(a)\text{:}}$

$$= \mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLoc}(a), \\ \quad \left(\begin{array}{l} \mathsf{FreshLoc}(a) \ \star_D \ \lambda \mathit{test}. \\ \texttt{updateCode}(f) \ \star_D \ \lambda\_. \\ \ \text{if } \mathit{test} \text{ then } x \text{ else } y \end{array}\right) \star_D \ \lambda v.z, \\ \quad \gamma \end{array}\right)$$

$\boxed{\text{Case analysis of } \mathit{test}\text{:}}$

$$= \mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLoc}(a), \\ \quad \left(\begin{array}{l} \mathsf{FreshLoc}(a) \ \star_D \ \lambda \mathit{test}. \\ \ \text{if } \mathit{test} \text{ then } \texttt{updateCode}(f) \ \star_D \ \lambda\_.x \text{ else } \texttt{updateCode}(f) \ \star_D \ \lambda\_.y \end{array}\right) \star_D \ \lambda v.z, \\ \quad \gamma \end{array}\right)$$

$\boxed{\text{Folding definition of } \mathsf{Obs}\text{:}}$

$$= \mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLoc}(a), \\ \quad \mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLoc}(a), \\ \quad \texttt{updateCode}(f) \ \star_D \ \lambda\_.x, \\ \quad \texttt{updateCode}(f) \ \star_D \ \lambda\_.y \end{array}\right) \star_D \ \lambda v.z, \\ \quad \gamma \end{array}\right)$$

$\boxed{\text{Observation Elimination:}}$

$$= \mathsf{Obs}(\mathsf{FreshLoc}(a), \texttt{updateCode}(f) \ \star_D \ \lambda\_.x \ \star_D \ \lambda v.z, \gamma)$$

□ Lemma 5

B.5 Proof of Lemma 6

Lemma 6 (FreshLabel and updateCode Interactions) If $L < L'$, then

1. $$\mathsf{Obs}(\mathsf{FreshLabel}(L'), \mathtt{updateCode}[L \mapsto \pi] \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y), \gamma)$$
$$= \mathsf{Obs}(\mathsf{FreshLabel}(L'), \mathtt{updateCode}[L \mapsto \pi] \star_D \lambda\_.x, \gamma)$$

2. $$\mathsf{Obs}(\mathsf{FreshLabel}(L), \mathtt{updateCode}[L \mapsto \pi] \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L+1), x, y), \gamma)$$
$$= \mathsf{Obs}(\mathsf{FreshLabel}(L), \mathtt{updateCode}[L \mapsto \pi] \star_D \lambda\_.x, \gamma)$$

$\boxed{\text{Proof of Lemma 6.1:}}$

$$\mathsf{Obs}(\mathsf{FreshLabel}(L'), \mathtt{updateCode}[L \mapsto \pi] \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y), \gamma)$$

$\boxed{\text{Unfolding definition of Obs:}}$

$$\begin{aligned}
= \quad & \mathsf{FreshLabel}(L') \star_D \lambda \mathit{test}. \\
& \text{if } \mathit{test} \text{ then} \\
& \quad \mathtt{updateCode}[L \mapsto \pi] \star_D \lambda\_. \\
& \quad \mathsf{FreshLabel}(L') \star_D \lambda \mathit{test}^*. \\
& \quad\ (\text{if } \mathit{test}^* \text{ then } x \text{ else } y) \\
& \text{else}\gamma
\end{aligned}$$

$\boxed{\text{Unfolding definition of FreshLabel}(L')\text{:}}$

$$\begin{aligned}
= \quad & \left(\begin{array}{l} \mathtt{getCode} \star_D \lambda\Pi. \\ \quad \mathbf{unit}_D(\forall L^* \geq L'.L^* \notin \mathit{dom}(\Pi)) \end{array} \right) \star_D \lambda \mathit{test}. \\
& \text{if } \mathit{test} \text{ then} \\
& \quad \mathtt{updateCode}[L \mapsto \pi] \star_D \lambda\_. \\
& \quad \left(\begin{array}{l} \mathtt{getCode} \star_D \lambda\Pi^*. \\ \quad \mathbf{unit}_D(\forall L^* \geq L'.L^* \notin \mathit{dom}(\Pi^*)) \end{array} \right) \star_D \lambda \mathit{test}^*. \\
& \quad\ (\text{if } \mathit{test}^* \text{ then } x \text{ else } y) \\
& \text{else}\gamma
\end{aligned}$$

From Associativity (Axiom 1.3):

```
=   getCode $\star_D$ $\lambda\Pi$.
    $\mathbf{unit}_D(\forall L^* \geq L'.L^* \notin dom(\Pi))$ $\star_D$ $\lambda test$.
    if $test$ then
        updateCode$[L \mapsto \pi]$ $\star_D$ $\lambda\_$.
        getCode $\star_D$ $\lambda\Pi^*$.
        $\mathbf{unit}_D(\forall L^* \geq L'.L^* \notin dom(\Pi^*))$ $\star_D$ $\lambda test^*$.
          (if $test^*$ then $x$ else $y$)
    else$\gamma$
```

From Axiom 2.4 and because $L < L'$, $\forall L^* \geq L'.L^* \notin dom(\Pi^*) \Leftrightarrow \forall L^* \geq L'.L^* \notin dom([L \mapsto \pi]\Pi^*)$:

```
=   getCode $\star_D$ $\lambda\Pi$.
    $\mathbf{unit}_D(\forall L^* \geq L'.L^* \notin dom(\Pi))$ $\star_D$ $\lambda test$.
    if $test$ then
        getCode $\star_D$ $\lambda\Pi^*$.
        updateCode$[L \mapsto \pi]$ $\star_D$ $\lambda\_$.
        $\mathbf{unit}_D(\forall L^* \geq L'.L^* \notin dom([L \mapsto \pi]\Pi^*))$ $\star_D$ $\lambda test^*$.
          (if $test^*$ then $x$ else $y$)
    else$\gamma$
```

From the innocence of getCode:

```
=   getCode $\star_D$ $\lambda\Pi$.
    $\mathbf{unit}_D(\forall L^* \geq L'.L^* \notin dom(\Pi))$ $\star_D$ $\lambda test$.
    getCode $\star_D$ $\lambda\Pi^*$.
    if $test$ then
        updateCode$[L \mapsto \pi]$ $\star_D$ $\lambda\_$.
        $\mathbf{unit}_D(\forall L^* \geq L'.L^* \notin dom([L \mapsto \pi]\Pi^*))$ $\star_D$ $\lambda test^*$.
          (if $test^*$ then $x$ else $y$)
    else$\gamma$
```

From Axiom 5.1, `getCode` commutes with $\mathbf{unit}_D(\forall L^* \geq L'.L^* \notin dom(\Pi))$:

```
=   getCode $\star_D$ $\lambda\Pi.$
    getCode $\star_D$ $\lambda\Pi^*.$
    $\mathbf{unit}_D(\forall L^* \geq L'.L^* \notin dom(\Pi))$ $\star_D$ $\lambda test.$
    if $test$ then
        updateCode$[L \mapsto \pi]$ $\star_D$ $\lambda\_.$
        $\mathbf{unit}_D(\forall L^* \geq L'.L^* \notin dom([L \mapsto \pi]\Pi^*))$ $\star_D$ $\lambda test^*.$
          (if $test^*$ then $x$ else $y$)
    else$\gamma$
```

From Axiom 2.3, Π^* can be replaced by Π:

```
=   getCode $\star_D$ $\lambda\Pi.$
    getCode $\star_D$ $\lambda\Pi^*.$
    $\mathbf{unit}_D(\forall L^* \geq L'.L^* \notin dom(\Pi))$ $\star_D$ $\lambda test.$
    if $test$ then
        updateCode$[L \mapsto \pi]$ $\star_D$ $\lambda\_.$
        $\mathbf{unit}_D(\forall L^* \geq L'.L^* \notin dom([L \mapsto \pi]\Pi))$ $\star_D$ $\lambda test^*.$
          (if $test^*$ then $x$ else $y$)
    else$\gamma$
```

Guarded by $test$, $\forall L^* \geq L'.L^* \notin dom(\Pi)$ is `true`:

```
=   getCode $\star_D$ $\lambda\Pi.$
    getCode $\star_D$ $\lambda\Pi^*.$
    $\mathbf{unit}_D(\forall L^* \geq L'.L^* \notin dom(\Pi))$ $\star_D$ $\lambda test.$
    if $test$ then
        updateCode$[L \mapsto \pi]$ $\star_D$ $\lambda\_.$
        $\mathbf{unit}_D(\texttt{true}))$ $\star_D$ $\lambda test^*.$
          (if $test^*$ then $x$ else $y$)
    else$\gamma$
```

$\boxed{\text{Left unit law and innocence of } \texttt{getCode}\text{:}}$

$$
\begin{aligned}
= \quad & \texttt{getCode} \star_D \ \lambda\Pi. \\
& \mathbf{unit}_D(\forall L^* \geq L'.L^* \notin dom(\Pi)) \ \star_D \ \lambda test. \\
& \text{if } test \text{ then} \\
& \qquad \texttt{updateCode}[L \mapsto \pi] \ \star_D \ \lambda\_.x \\
& \text{else}\gamma
\end{aligned}
$$

$\boxed{\text{Folding back definitions of } \mathsf{FreshLabel}(L') \text{ and } \mathsf{Obs}\text{:}}$

$$
= \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), \texttt{updateCode}[L \mapsto \pi] \ \star_D \ \lambda\_.x, \gamma)
$$

□ Lemma 6.1

Proof of Lemma 6.2:

$$\mathsf{Obs}(\mathsf{FreshLabel}(L), \mathtt{updateCode}[L \mapsto \pi] \ \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L+1), x, y), \gamma)$$

Unfolding definition of Obs:

$$\begin{array}{ll}
= & \mathsf{FreshLabel}(L) \ \star_D \ \lambda test. \\
 & \text{if } test \text{ then} \\
 & \qquad \mathtt{updateCode}[L \mapsto \pi] \ \star_D \ \lambda\_. \\
 & \qquad \mathsf{FreshLabel}(L+1) \ \star_D \ \lambda test^*. \\
 & \qquad \quad (\text{if } test^* \text{ then } x \text{ else } y) \\
 & \text{else}\gamma
\end{array}$$

Unfolding definition of $\mathsf{FreshLabel}(L')$:

$$\begin{array}{ll}
= & \left(\begin{array}{l} \mathtt{getCode} \ \star_D \ \lambda\Pi. \\ \quad \mathbf{unit}_D(\forall L^* \geq L.L^* \notin dom(\Pi)) \end{array} \right) \star_D \ \lambda test. \\
 & \text{if } test \text{ then} \\
 & \qquad \mathtt{updateCode}[L \mapsto \pi] \ \star_D \ \lambda\_. \\
 & \qquad \left(\begin{array}{l} \mathtt{getCode} \ \star_D \ \lambda\Pi^*. \\ \quad \mathbf{unit}_D(\forall L^* \geq L+1.L^* \notin dom(\Pi^*)) \end{array} \right) \star_D \ \lambda test^*. \\
 & \qquad \quad (\text{if } test^* \text{ then } x \text{ else } y) \\
 & \text{else}\gamma
\end{array}$$

From Associativity (Axiom 1.3):

```
$=$  getCode $\star_D$ $\lambda\Pi.$
     $\mathbf{unit}_D(\forall L^* \geq L.L^* \notin \mathit{dom}(\Pi))$ $\star_D$ $\lambda \mathit{test}.$
     if $\mathit{test}$ then
          updateCode$[L \mapsto \pi]$ $\star_D$ $\lambda\_.$
          getCode $\star_D$ $\lambda\Pi^*.$
          $\mathbf{unit}_D(\forall L^* \geq L+1.L^* \notin \mathit{dom}(\Pi^*))$ $\star_D$ $\lambda \mathit{test}^*.$
            (if $\mathit{test}^*$ then $x$ else $y$)
     else$\gamma$
```

From Axiom 2.4 and because $L < L+1$, $\forall L^* \geq L.L^* \notin \mathit{dom}(\Pi^*) \Leftrightarrow \forall L^* \geq L+1.L^* \notin \mathit{dom}([L \mapsto \pi]\Pi^*)$, we can change the second observation:

```
$=$  getCode $\star_D$ $\lambda\Pi.$
     $\mathbf{unit}_D(\forall L^* \geq L'.L^* \notin \mathit{dom}(\Pi))$ $\star_D$ $\lambda \mathit{test}.$
     if $\mathit{test}$ then
          getCode $\star_D$ $\lambda\Pi^*.$
          updateCode$[L \mapsto \pi]$ $\star_D$ $\lambda\_.$
          $\mathbf{unit}_D(\forall L^* \geq L+1.L^* \notin \mathit{dom}([L \mapsto \pi]\Pi^*))$ $\star_D$ $\lambda \mathit{test}^*.$
            (if $\mathit{test}^*$ then $x$ else $y$)
     else$\gamma$
```

From the innocence of `getCode`:

```
$=$  $\texttt{getCode} \star_D \lambda\Pi.$
     $\mathbf{unit}_D(\forall L^* \geq L.L^* \notin \mathit{dom}(\Pi)) \star_D \lambda \mathit{test}.$
     $\texttt{getCode} \star_D \lambda\Pi^*.$
     if $\mathit{test}$ then
          $\texttt{updateCode}[L \mapsto \pi] \star_D \lambda\_.$
          $\mathbf{unit}_D(\forall L^* \geq L+1.L^* \notin \mathit{dom}([L \mapsto \pi]\Pi^*)) \star_D \lambda \mathit{test}^*.$
            (if $\mathit{test}^*$ then $x$ else $y$)
     else$\gamma$
```

From Axiom 5.1, $\texttt{getCode}$ commutes with $\mathbf{unit}_D(\forall L^* \geq L.L^* \notin \mathit{dom}(\Pi))$:

```
$=$  $\texttt{getCode} \star_D \lambda\Pi.$
     $\texttt{getCode} \star_D \lambda\Pi^*.$
     $\mathbf{unit}_D(\forall L^* \geq L.L^* \notin \mathit{dom}(\Pi)) \star_D \lambda \mathit{test}.$
     if $\mathit{test}$ then
          $\texttt{updateCode}[L \mapsto \pi] \star_D \lambda\_.$
          $\mathbf{unit}_D(\forall L^* \geq L+1.L^* \notin \mathit{dom}([L \mapsto \pi]\Pi^*)) \star_D \lambda \mathit{test}^*.$
            (if $\mathit{test}^*$ then $x$ else $y$)
     else$\gamma$
```

From Axiom 2.3, Π^* can be replaced by Π:

```
$=$  $\texttt{getCode} \star_D \lambda\Pi.$
     $\texttt{getCode} \star_D \lambda\Pi^*.$
     $\mathbf{unit}_D(\forall L^* \geq L.L^* \notin \mathit{dom}(\Pi)) \star_D \lambda \mathit{test}.$
     if $\mathit{test}$ then
          $\texttt{updateCode}[L \mapsto \pi] \star_D \lambda\_.$
          $\mathbf{unit}_D(\forall L^* \geq L+1.L^* \notin \mathit{dom}([L \mapsto \pi]\Pi)) \star_D \lambda \mathit{test}^*.$
            (if $\mathit{test}^*$ then $x$ else $y$)
     else$\gamma$
```

Note that $(\forall L^* \geq L.L^* \notin \mathit{dom}(\Pi)) \supset (\forall L^* \geq L+1.L^* \notin \mathit{dom}([L \mapsto \pi]\Pi))$:

$$
\begin{aligned}
= \quad & \texttt{getCode} \ \star_D \ \lambda\Pi.\\
& \texttt{getCode} \ \star_D \ \lambda\Pi^*.\\
& \mathbf{unit}_D(\forall L^* \geq L'.L^* \notin \mathit{dom}(\Pi)) \ \star_D \ \lambda \mathit{test}.\\
& \text{if } \mathit{test} \text{ then}\\
& \qquad \texttt{updateCode}[L \mapsto \pi] \ \star_D \ \lambda\_.\\
& \qquad \mathbf{unit}_D(\texttt{true})) \ \star_D \ \lambda \mathit{test}^*.\\
& \qquad\quad (\text{if } \mathit{test}^* \text{ then } x \text{ else } y)\\
& \text{else}\gamma
\end{aligned}
$$

Left unit law and innocence of `getCode`:

$$
\begin{aligned}
= \quad & \texttt{getCode} \ \star_D \ \lambda\Pi.\\
& \mathbf{unit}_D(\forall L^* \geq L'.L^* \notin \mathit{dom}(\Pi)) \ \star_D \ \lambda \mathit{test}.\\
& \text{if } \mathit{test} \text{ then}\\
& \qquad \texttt{updateCode}[L \mapsto \pi] \ \star_D \ \lambda\_.x\\
& \text{else}\gamma
\end{aligned}
$$

Folding back definitions of $\mathsf{FreshLabel}(L')$ and Obs:

$$
= \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), \texttt{updateCode}[L \mapsto \pi] \ \star_D \ \lambda\_.x, \gamma)
$$

□ Lemma 5

B.6 Proof of Lemma 8

Lemma 8 (Commands Preserve Store Shape)

$$\begin{array}{l}\mathcal{S}[\![c : \mathbf{comm}]\!] \star_S \lambda\delta_c. \\ \quad \mathbf{unit}_S \left(\begin{array}{l} \mathtt{getSto} \star_D \lambda\sigma_0. \\ \delta_c \star_D \lambda\_. \\ \mathtt{getSto} \star_D \lambda\sigma_1. \\ \quad \mathbf{unit}_D(\mathsf{dom}(\sigma_0) = \mathsf{dom}(\sigma_1)) \end{array} \right)\end{array} = \begin{array}{l}\mathcal{S}[\![c : \mathbf{comm}]\!] \star_S \lambda\delta_c. \\ \quad \mathbf{unit}_S(\delta_c \star_D \lambda\_.\mathbf{unit}_D(\mathtt{true}))\end{array}$$

$$\begin{array}{l}\mathcal{C}[\![c : \mathbf{comm}]\!] \star_S \lambda\varphi_c. \\ \quad \mathbf{unit}_S \left(\begin{array}{l} \mathtt{getSto} \star_D \lambda\sigma_0. \\ \varphi_c \star_D \lambda\_. \\ \mathtt{getSto} \star_D \lambda\sigma_1. \\ \quad \mathbf{unit}_D(\mathsf{dom}(\sigma_0) = \mathsf{dom}(\sigma_1)) \end{array} \right)\end{array} = \begin{array}{l}\mathcal{C}[\![c : \mathbf{comm}]\!] \star_S \lambda\varphi_c. \\ \quad \mathbf{unit}_S(\varphi_c \star_D \lambda\_.\mathbf{unit}_D(\mathtt{true}))\end{array}$$

Proof of Lemma 8

This proof proceeds by induction on terms. The only case which is not obvious by inspection is that of "**new** x **in** c"—all other commands do no allocation or deallocation of memory cells, and so do not change the store shape.

$$\begin{array}{l}\mathcal{S}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!] \star_S \lambda\delta. \\ \quad \mathbf{unit}_S \left(\begin{array}{l} \mathtt{getSto} \star_D \lambda\sigma_0. \\ \delta \star_D \lambda\_. \\ \mathtt{getSto} \star_D \lambda\sigma_1. \\ \quad \mathbf{unit}_D(\mathsf{dom}(\sigma_0) = \mathsf{dom}(\sigma_1)) \end{array} \right)\end{array}$$

$\boxed{\text{Unfolding definition of } \mathcal{S}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!]\text{:}}$

$$
\begin{array}{l}
= \left(\begin{array}{l}
\texttt{rdAddr} \star_S \lambda a. \\
\quad \texttt{inAddr}\,(a+1) \left(\begin{array}{l}
\texttt{rdEnv} \star_S \lambda \rho. \\
\texttt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_S(a)])\ \mathcal{S}[\![c]\!]
\end{array} \right) \star_S \lambda \delta_c. \\
\qquad \mathbf{unit}_S(\texttt{Alloc}(a) \star_D \lambda\_.\delta_c \star_D \lambda\_.\texttt{deAlloc}(a))
\end{array} \right) \star_S \lambda \delta. \\
\quad \mathbf{unit}_S \left(\begin{array}{l}
\texttt{getSto} \star_D \lambda \sigma_0. \\
\delta \star_D \lambda\_. \\
\texttt{getSto} \star_D \lambda \sigma_1. \\
\quad \mathbf{unit}_D(\mathsf{dom}(\sigma_0) = \mathsf{dom}(\sigma_1))
\end{array} \right)
\end{array}
$$

$\boxed{\text{By left unit:}}$

$$
\begin{array}{l}
= \texttt{rdAddr} \star_S \lambda a. \\
\quad \texttt{inAddr}\,(a+1) \left(\begin{array}{l}
\texttt{rdEnv} \star_S \lambda \rho. \\
\texttt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_S(a)])\ \mathcal{S}[\![c]\!]
\end{array} \right) \star_S \lambda \delta_c. \\
\quad \mathbf{unit}_S \left(\begin{array}{l}
\texttt{getSto} \star_D \lambda \sigma_0. \\
(\texttt{Alloc}(a) \star_D \lambda\_.\delta_c \star_D \lambda\_.\texttt{deAlloc}(a)) \star_D \lambda\_. \\
\texttt{getSto} \star_D \lambda \sigma_1. \\
\quad \mathbf{unit}_D(\mathsf{dom}(\sigma_0) = \mathsf{dom}(\sigma_1))
\end{array} \right)
\end{array}
$$

By innocence of $\mathbf{unit}_D(\mathtt{true})$:

$$
\begin{array}{rl}
= & \mathtt{rdAddr} \star_S \lambda a. \\
& \mathtt{inAddr}\,(a+1) \left(\begin{array}{l} \mathtt{rdEnv} \star_S \lambda\rho. \\ \mathtt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_S(a)])\, \mathcal{S}[\![c]\!] \end{array} \right) \star_S \lambda\delta_c. \\
& \mathbf{unit}_S \left(\begin{array}{l} \mathtt{getSto} \star_D \lambda\sigma_0. \\ (\mathtt{Alloc}(a) \star_D \lambda\_.(\delta_c \star_D \lambda\_.\mathbf{unit}_D(\mathtt{true})) \star_D \lambda\_.\mathtt{deAlloc}(a)) \star_D \lambda\_. \\ \mathtt{getSto} \star_D \lambda\sigma_1. \\ \quad \mathbf{unit}_D(\mathsf{dom}(\sigma_0) = \mathsf{dom}(\sigma_1)) \end{array}\right)
\end{array}
$$

By the inductive hypothesis for c:

$$
\begin{array}{rl}
= & \mathtt{rdAddr} \star_S \lambda a. \\
& \mathtt{inAddr}\,(a+1) \left(\begin{array}{l} \mathtt{rdEnv} \star_S \lambda\rho. \\ \mathtt{inEnv}\,(\rho[x \mapsto \mathbf{unit}_S(a)])\, \mathcal{S}[\![c]\!] \end{array} \right) \star_S \lambda\delta_c. \\
& \mathbf{unit}_S \left(\begin{array}{l} \mathtt{getSto} \star_D \lambda\sigma_0. \\ (\mathtt{Alloc}(a) \star_D \lambda\_. \left[\begin{array}{l} \mathtt{getSto} \star_D \lambda\sigma'_0. \\ \delta_c \star_D \lambda\_. \\ \mathtt{getSto} \star_D \lambda\sigma'_1. \\ \quad \mathbf{unit}_D(\mathsf{dom}(\sigma'_0) = \mathsf{dom}(\sigma'_1)) \end{array} \right] \star_D \lambda\_\mathtt{deAlloc}(a)) \star_D \lambda\_. \\ \mathtt{getSto} \star_D \lambda\sigma_1. \\ \quad \mathbf{unit}_D(\mathsf{dom}(\sigma_0) = \mathsf{dom}(\sigma_1)) \end{array}\right)
\end{array}
$$

Case 1: $a \in \sigma_0$. In this case, the dynamic result on both sides of the equation in the lemma (i.e., the terms within $\mathbf{unit}_S(\ldots)$) are $\bot_{\mathsf{Dynam}}(\mathtt{void})$.

Case 2:
$$
\begin{array}{rl}
a \notin \sigma_0 & \Longrightarrow\ \sigma'_0 = \sigma_0 \cup \{\langle a, 0\rangle\} =_{\mathrm{IH}} \sigma'_1 \\
& \Longrightarrow\ \sigma_1 = \sigma'_1 \setminus \{\langle a, -\rangle\} = \sigma_0 \\
& \Longrightarrow\ \mathsf{dom}(\sigma_0) = \mathsf{dom}(\sigma_1)
\end{array}
$$

Observe that the proof for $\mathcal{C}[\![-]\!]$ is completely analogous to the proof for the staged semantics

$\mathcal{S}[\![-]\!]$.

□Lemma 8.

B.7 Proof of Lemma 9

Lemma 9 (Discharging Label Freshness)

$$\begin{array}{l}
\texttt{getLabel} \star_S \lambda L. \\
\mathcal{C}[\![t]\!] \star_S \lambda\varphi_t. \\
\texttt{getLabel} \star_S \lambda L'. \\
\quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l} \varphi_t \star_D \lambda v. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array}\right), z))
\end{array}$$

$$\begin{array}{rl}
= & \texttt{getLabel} \star_S \lambda L. \\
& \mathcal{C}[\![t]\!] \star_S \lambda\varphi_t. \\
& \texttt{getLabel} \star_S \lambda L'. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L), \varphi_t \star_D \lambda v.x, z))
\end{array}$$

where $\texttt{getLabel} : \mathsf{Static}(Label)$ is the $\texttt{get}$ operator added by the $\mathcal{T}_{\mathsf{St}}\, Label$ monad transformer.

Proof of Lemma 9.

This proof proceeds by induction on terms. The cases which do not involve control-flow (and hence labels and code store) follow easily.

Case $t = c_1 \,;\, c_2$

$$\begin{array}{l}
\texttt{getLabel} \star_S \lambda L. \\
\mathcal{C}[\![c_1 \,;\, c_2]\!] \star_S \lambda\varphi_t. \\
\texttt{getLabel} \star_S \lambda L'. \\
\quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l} \varphi_t \star_D \lambda v. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array}\right), z))
\end{array}$$

$$\boxed{\text{By definition of } \mathcal{C}[\![c_1\ ;\ c_2]\!] \text{ and the innocence of } \texttt{getLabel}\text{:}}$$

$$\begin{aligned}
= \ & \texttt{getLabel} \star_S \lambda L. \\
& \mathcal{C}[\![c_1]\!] \star_S \lambda \varphi_1. \\
& \texttt{getLabel} \star_S \lambda L_1. \\
& \mathcal{C}[\![c_2]\!] \star_S \lambda \varphi_2. \\
& \texttt{getLabel} \star_S \lambda L'. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L), \varphi_1 \star_D \lambda\_.\varphi_2 \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y), z))
\end{aligned}$$

$$\boxed{\text{By Observation Introduction:}}$$

$$\begin{aligned}
= \ & \texttt{getLabel} \star_S \lambda L. \\
& \mathcal{C}[\![c_1]\!] \star_S \lambda \varphi_1. \\
& \texttt{getLabel} \star_S \lambda L_1. \\
& \mathcal{C}[\![c_2]\!] \star_S \lambda \varphi_2. \\
& \texttt{getLabel} \star_S \lambda L'. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L), \varphi_1 \star_D \lambda\_.\mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLabel}(L_1), \\ \quad \varphi_2 \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y), \\ \quad \varphi_2 \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array}\right), z))
\end{aligned}$$

By Observation Introduction:

$$
\begin{aligned}
= \;\; & \texttt{getLabel} \star_S \lambda L. \\
& \mathcal{C}[\![c_1]\!] \star_S \lambda \varphi_1. \\
& \texttt{getLabel} \star_S \lambda L_1. \\
& \mathcal{C}[\![c_2]\!] \star_S \lambda \varphi_2. \\
& \texttt{getLabel} \star_S \lambda L'. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}\begin{pmatrix}
\mathsf{FreshLabel}(L), & & \\
& \mathsf{Obs}\begin{pmatrix}
\mathsf{FreshLabel}(L), & \\
& \varphi_1 \star_D \lambda\_.\mathsf{Obs}\begin{pmatrix}
\mathsf{FreshLabel}(L_1), \\
\varphi_2 \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y), \\
\varphi_2 \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y)
\end{pmatrix}, \\
& \varphi_1 \star_D \lambda\_.\mathsf{Obs}\begin{pmatrix}
\mathsf{FreshLabel}(L_1), \\
\varphi_2 \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y), \\
\varphi_2 \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y)
\end{pmatrix}
\end{pmatrix} & , \\
& z &
\end{pmatrix})
\end{aligned}
$$

$\boxed{\text{By Induction Hypothesis for } c_2\text{:}}$

$$
\begin{array}{ll}
= & \texttt{getLabel} \star_S \lambda L. \\
& \mathcal{C}[\![c_1]\!] \star_S \lambda \varphi_1. \\
& \texttt{getLabel} \star_S \lambda L_1. \\
& \mathcal{C}[\![c_2]\!] \star_S \lambda \varphi_2. \\
& \texttt{getLabel} \star_S \lambda L'. \\
& \quad \mathbf{unit}_S(\mathsf{Obs} \left(\begin{array}{l}
\mathsf{FreshLabel}(L), \\
\quad \mathsf{Obs} \left(\begin{array}{l}
\mathsf{FreshLabel}(L), \\
\quad \varphi_1 \star_D \lambda\_.\mathsf{Obs} \left(\begin{array}{l}
\mathsf{FreshLabel}(L_1), \\
\quad \varphi_2 \star_D \lambda\_.x, \\
\quad \varphi_2 \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y)
\end{array} \right), \\
\quad \varphi_1 \star_D \lambda\_.\mathsf{Obs} \left(\begin{array}{l}
\mathsf{FreshLabel}(L_1), \\
\quad \varphi_2 \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y), \\
\quad \varphi_2 \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y)
\end{array} \right)
\end{array} \right), \\
\quad z
\end{array} \right))
\end{array}
$$

By Induction Hypothesis for c_1:

$$
\begin{aligned}
= {} & \texttt{getLabel} \star_S \lambda L.\\
& \mathcal{C}[\![c_1]\!] \star_S \lambda\varphi_1.\\
& \texttt{getLabel} \star_S \lambda L_1.\\
& \mathcal{C}[\![c_2]\!] \star_S \lambda\varphi_2.\\
& \texttt{getLabel} \star_S \lambda L'.\\
& \quad \mathbf{unit}_S(\mathsf{Obs}\left(\begin{array}{l}
\mathsf{FreshLabel}(L),\\
\quad \mathsf{Obs}\left(\begin{array}{l}
\mathsf{FreshLabel}(L),\\
\quad \varphi_1 \star_D \lambda\_.\varphi_2 \star_D \lambda\_.x,\\
\quad \varphi_1 \star_D \lambda\_.\mathsf{Obs}\left(\begin{array}{l}
\mathsf{FreshLabel}(L_1),\\
\quad \varphi_2 \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y),\\
\quad \varphi_2 \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y)
\end{array}\right)
\end{array}\right),\\
\quad z
\end{array}\right))
\end{aligned}
$$

By Observation Elimination:

$$
\begin{aligned}
= {} & \texttt{getLabel} \star_S \lambda L.\\
& \mathcal{C}[\![c_1]\!] \star_S \lambda\varphi_1.\\
& \texttt{getLabel} \star_S \lambda L_1.\\
& \mathcal{C}[\![c_2]\!] \star_S \lambda\varphi_2.\\
& \texttt{getLabel} \star_S \lambda L'.\\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L), \varphi_1 \star_D \lambda\_.\varphi_2 \star_D \lambda\_.x, z))
\end{aligned}
$$

$$
\begin{aligned}
= {} & \texttt{getLabel} \star_S \lambda L.\\
& \mathcal{C}[\![c_1 \, ; c_2]\!] \star_S \lambda\varphi_t.\\
& \texttt{getLabel} \star_S \lambda L'.\\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L), \varphi_t \star_D \lambda\_.x, z))
\end{aligned}
$$

$\boxed{\text{Case } t = \textbf{if } b \textbf{ then } c}$

$$\begin{array}{l}
\texttt{getLabel} \star_S \ \lambda L. \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } c]\!] \star_S \ \lambda \varphi_t. \\
\texttt{getLabel} \star_S \ \lambda L'. \\
\quad \textbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l} \varphi_t \star_D \ \lambda v. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array} \right), z))
\end{array}$$

$\boxed{\text{By the definition of } \mathcal{C}[\![\textbf{if } b \textbf{ then } c]\!]\text{:}}$

$$\begin{array}{rl}
= & \texttt{getLabel} \star_S \ \lambda L. \\
& \texttt{newlabel} \star_S \ \lambda L_\kappa. \\
& \texttt{newlabel} \star_S \ \lambda L_c. \\
& \mathcal{C}[\![b]\!] \star_S \ \lambda \varphi_b. \\
& \mathcal{C}[\![c]\!] \star_S \ \lambda \varphi_c. \\
& \texttt{getLabel} \star_S \ \lambda L'. \\
& \quad \textbf{unit}_S(\mathsf{Obs} \left(\begin{array}{l} \mathsf{FreshLabel}(L), \\ \qquad \left(\begin{array}{l} \mathsf{IfThenPS}(\varphi_b, \varphi_c, L_c, L_\kappa) \star_D \ \lambda\_. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array} \right), \\ \qquad\quad z \end{array} \right))
\end{array}$$

$$\boxed{\text{By the innocence of } \mathcal{C}[\![b]\!]\text{:}}$$

$$\begin{aligned}
= \ & \texttt{getLabel} \star_S \ \lambda L. \\
& \mathcal{C}[\![b]\!] \star_S \ \lambda \varphi_b. \\
& \texttt{newlabel} \star_S \ \lambda L_\kappa. \\
& \texttt{newlabel} \star_S \ \lambda L_c. \\
& \mathcal{C}[\![c]\!] \star_S \ \lambda \varphi_c. \\
& \texttt{getLabel} \star_S \ \lambda L'. \\
& \quad \mathbf{unit}_S(\mathsf{Obs} \left(\begin{array}{l}
\mathsf{FreshLabel}(L), \\
\qquad \left(\begin{array}{l}
\mathsf{IfThenPS}(\varphi_b, \varphi_c, L_c, L_\kappa) \ \star_D \ \lambda\_. \\
\quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y)
\end{array} \right), \\
\qquad\qquad z
\end{array} \right))
\end{aligned}$$

$$\boxed{\text{By the innocence of } \texttt{getLabel}\text{:}}$$

$$\begin{aligned}
= \ & \texttt{getLabel} \star_S \ \lambda L. \\
& \mathcal{C}[\![b]\!] \star_S \ \lambda \varphi_b. \\
& \texttt{getLabel} \star_S \ \lambda L_0. \\
& \texttt{newlabel} \star_S \ \lambda L_\kappa. \\
& \texttt{newlabel} \star_S \ \lambda L_c. \\
& \texttt{getLabel} \star_S \ \lambda L_1. \\
& \mathcal{C}[\![c]\!] \star_S \ \lambda \varphi_c. \\
& \texttt{getLabel} \star_S \ \lambda L'. \\
& \quad \mathbf{unit}_S(\mathsf{Obs} \left(\begin{array}{l}
\mathsf{FreshLabel}(L), \\
\qquad \left(\begin{array}{l}
\mathsf{IfThenPS}(\varphi_b, \varphi_c, L_c, L_\kappa) \ \star_D \ \lambda\_. \\
\quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y)
\end{array} \right), \\
\qquad\qquad z
\end{array} \right))
\end{aligned}$$

We note here that by construction: $L \leq L_0 (= L_\kappa) < L_c (= L_0 + 1) < L_1 (= L_0 + 2) \leq L'$.

For the sake of clarity, the following context will be abbreviated by "$\ddots$" for the remainder of this proof:

$$
\begin{aligned}
\mathit{context}[\circ] = \;& \texttt{getLabel} \star_S \lambda L.\\
& \mathcal{C}[\![b]\!] \star_S \lambda \varphi_b.\\
& \texttt{getLabel} \star_S \lambda L_0.\\
& \texttt{newlabel} \star_S \lambda L_\kappa.\\
& \texttt{newlabel} \star_S \lambda L_c.\\
& \texttt{getLabel} \star_S \lambda L_1.\\
& \mathcal{C}[\![c]\!] \star_S \lambda \varphi_c.\\
& \texttt{getLabel} \star_S \lambda L'.\\
& \quad \mathbf{unit}_S(\circ)
\end{aligned}
$$

$\boxed{\text{Unfolding } \mathsf{IfThenPS}(\varphi_b, \varphi_c, L_c, L_\kappa)\text{:}}$

$$
\begin{aligned}
= \;& \ddots\\
& \mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l} \left(\begin{array}{l} \varphi_b \star_D \lambda\beta.\\ \texttt{callcc}\ \lambda\kappa.\\ \quad \texttt{updateCode}[L_\kappa \mapsto \kappa\bullet] \star_D \lambda\_.\\ \quad \texttt{updateCode}[L_c \mapsto \varphi_c \star_D \lambda\_.\texttt{jump}\, L_\kappa] \star_D \lambda\_.\\ \quad\quad \beta\langle \texttt{jump}\, L_c, \texttt{jump}\, L_\kappa \rangle \end{array} \right) \\ \quad\quad\quad \star_D \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array} \right), z)
\end{aligned}
$$

Focusing now on $\texttt{jump}\, L_c$ and $\texttt{jump}\, L_\kappa$ within the above expression, they may be "unrolled" using Axioms 2.1 and 2.2.

“Unrolling” $\mathtt{jump}\,L_c$ and $\mathtt{jump}\,L_\kappa$ using Axioms 2.1 and 2.2:

$$= \ddots$$

$$\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l} \left(\begin{array}{l} \varphi_b \ \star_D \ \lambda\beta. \\ \mathtt{callcc}\ \lambda\kappa. \\ \quad \mathtt{updateCode}[L_\kappa \mapsto \kappa\bullet] \ \star_D \ \lambda\_. \\ \quad \mathtt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ \lambda\_.\mathtt{jump}\,L_\kappa] \ \star_D \ \lambda\_. \\ \qquad \beta\langle \varphi_c \ \star_D \ \lambda\_.\kappa\bullet, \kappa\bullet\rangle \end{array} \right) \\ \qquad\qquad \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array} \right), z)$$

Subcase: $\beta\langle \varphi_c \ \star_D \ \lambda\_.\kappa\bullet, \kappa\bullet\rangle = \varphi_c \ \star_D \ \lambda\_.\kappa\bullet$:

$$= \ddots$$

$$\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l} \left(\begin{array}{l} \varphi_b \ \star_D \ \lambda\beta. \\ \mathtt{callcc}\ \lambda\kappa. \\ \quad \mathtt{updateCode}[L_\kappa \mapsto \kappa\bullet] \ \star_D \ \lambda\_. \\ \quad \mathtt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ \lambda\_.\mathtt{jump}\,L_\kappa] \ \star_D \ \lambda\_. \\ \qquad \varphi_c \ \star_D \ \lambda\_.\kappa\bullet \end{array} \right) \\ \qquad\qquad \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array} \right), z)$$

By Axiom 3.2 where $\kappa_0 = \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \ \star_D \ \kappa$:

$$= \ddots$$

$$\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l} \varphi_b \ \star_D \ \lambda\beta. \\ \mathtt{callcc}\ \lambda\kappa. \\ \quad \mathtt{updateCode}[L_\kappa \mapsto \kappa_0\bullet] \ \star_D \ \lambda\_. \\ \quad \mathtt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ \lambda\_.\mathtt{jump}\,L_\kappa] \ \star_D \ \lambda\_. \\ \qquad \varphi_c \ \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \ \star_D \ \kappa \end{array} \right), z)$$

By Observation Introduction:

$$= \ddots$$

$$\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l} \varphi_b \ \star_D \ \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \quad \texttt{updateCode}[L_\kappa \mapsto \kappa_0 \bullet] \ \star_D \ \lambda\_. \\ \quad \texttt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ \lambda\_.\texttt{jump}\, L_\kappa] \ \star_D \ \lambda\_. \\ \qquad\qquad \mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLabel}(L_1), \\ \quad \varphi_c \ \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \ \star_D \ \kappa, \\ \quad \varphi_c \ \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \ \star_D \ \kappa \end{array}\right) \end{array}\right), z)$$

Let $bad_0 = \varphi_c \ \star_D \ \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \ \star_D \ \kappa$, then by the induction hypothesis for c:

$$= \ddots$$

$$\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l} \varphi_b \ \star_D \ \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \quad \texttt{updateCode}[L_\kappa \mapsto \kappa_0 \bullet] \ \star_D \ \lambda\_. \\ \quad \texttt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ \lambda\_.\texttt{jump}\, L_\kappa] \ \star_D \ \lambda\_. \\ \qquad \mathsf{Obs}(\mathsf{FreshLabel}(L_1), \varphi_c \ \star_D \ \lambda\_.x \ \star_D \ \kappa, bad_0) \end{array}\right), z)$$

Observation Introduction:

$$= \ddots$$

$$\mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLabel}(L), \\ \quad \left(\begin{array}{l} \varphi_b \ \star_D \ \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \quad \texttt{updateCode}[L_\kappa \mapsto \kappa_0\bullet] \ \star_D \ \lambda\_. \\ \qquad \mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLabel}(L_c), \\ \quad \left(\begin{array}{l} \texttt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ \lambda\_.\texttt{jump}\,L_\kappa] \ \star_D \ \lambda\_. \\ \mathsf{Obs}(\mathsf{FreshLabel}(L_1), \varphi_c \ \star_D \ \lambda\_.x \ \star_D \ \kappa, bad_0) \end{array}\right), \\ \quad \left(\begin{array}{l} \texttt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ \lambda\_.\texttt{jump}\,L_\kappa] \ \star_D \ \lambda\_. \\ \mathsf{Obs}(\mathsf{FreshLabel}(L_1), \varphi_c \ \star_D \ \lambda\_.x \ \star_D \ \kappa, bad_0) \end{array}\right) \end{array}\right) \end{array}\right), \\ \quad z \end{array}\right)$$

Since $L_c = L_0 + 1$ and $L_1 = L_0 + 2$, by Lemma 6.2

$$= \ddots$$

$$\mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLabel}(L), \\ \quad \left(\begin{array}{l} \varphi_b \ \star_D \ \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \quad \texttt{updateCode}[L_\kappa \mapsto \kappa_0\bullet] \ \star_D \ \lambda\_. \\ \qquad \mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLabel}(L_c), \\ \quad \left(\begin{array}{l} \texttt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ \lambda\_.\texttt{jump}\,L_\kappa] \ \star_D \ \lambda\_. \\ \varphi_c \ \star_D \ \lambda\_.x \ \star_D \ \kappa \end{array}\right), \\ \quad \left(\begin{array}{l} \texttt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ \lambda\_.\texttt{jump}\,L_\kappa] \ \star_D \ \lambda\_. \\ \mathsf{Obs}(\mathsf{FreshLabel}(L_1), \varphi_c \ \star_D \ \lambda\_.x \ \star_D \ \kappa, bad_0) \end{array}\right) \end{array}\right) \end{array}\right), \\ \quad z \end{array}\right)$$

For the sake of readability, use the following abbreviations:

$$\begin{array}{rcl}
\mathit{good}_1 & = & \left(\begin{array}{l} \mathtt{updateCode}[L_c \mapsto \varphi_c \;\star_D\; \lambda\_.\mathtt{jump}\, L_\kappa] \;\star_D\; \lambda\_. \\ \quad \varphi_c \;\star_D\; \lambda\_.x \;\star_D\; \kappa \end{array} \right) \\
\mathit{bad}_1 & = & \left(\begin{array}{l} \mathtt{updateCode}[L_c \mapsto \varphi_c \;\star_D\; \lambda\_.\mathtt{jump}\, L_\kappa] \;\star_D\; \lambda\_. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L_1), \varphi_c \;\star_D\; \lambda\_.x \;\star_D\; \kappa, \mathit{bad}_0) \end{array} \right)
\end{array}$$

$$= \ddots$$

$$\mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLabel}(L), \\ \quad \left(\begin{array}{l} \varphi_b \;\star_D\; \lambda\beta. \\ \mathtt{callcc}\; \lambda\kappa. \\ \quad \mathtt{updateCode}[L_\kappa \mapsto \kappa_0 \bullet] \;\star_D\; \lambda\_. \\ \quad\quad \mathsf{Obs}(\mathsf{FreshLabel}(L_c), \mathit{good}_1, \mathit{bad}_1) \end{array} \right), \\ \quad z \end{array} \right)$$

Since $L_\kappa = L_0$ and $L_c = L_0 + 1$, by Lemma 6.2

$$= \ddots$$

$$\mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLabel}(L), \\ \quad \left(\begin{array}{l} \varphi_b \;\star_D\; \lambda\beta. \\ \mathtt{callcc}\; \lambda\kappa. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L_\kappa), \mathit{good}_2, \mathit{bad}_2) \end{array} \right), \\ \quad z \end{array} \right)$$

where:

$$\begin{array}{rcl}
\mathit{good}_2 & = & \left(\begin{array}{l}
\texttt{updateCode}[L_\kappa \mapsto \kappa_0\bullet] \ \star_D \ \lambda\_. \\
\texttt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ \lambda\_.\texttt{jump}\, L_\kappa] \ \star_D \ \lambda\_. \\
\quad \varphi_c \ \star_D \ \lambda\_.x \ \star_D \ \kappa
\end{array}\right) \\
\mathit{bad}_2 & = & \left(\begin{array}{l}
\texttt{updateCode}[L_\kappa \mapsto \kappa_0\bullet] \ \star_D \ \lambda\_. \\
\quad \mathsf{Obs}(\mathsf{FreshLabel}(L_c), \mathit{good}_1, \mathit{bad}_1)
\end{array}\right)
\end{array}$$

$\boxed{\text{From Lemma 7, } \texttt{callcc} \text{ commutes with } \mathsf{Obs}\text{:}}$

$$\begin{array}{l}
= \ \ddots \\
\qquad \mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l}
\varphi_b \ \star_D \ \lambda\beta. \\
\quad \mathsf{Obs}(\mathsf{FreshLabel}(L_\kappa), \mathit{good}_3, \mathit{bad}_3)
\end{array}\right), z)
\end{array}$$

where:

$$\begin{array}{rcl}
\mathit{good}_3 & = & \left(\begin{array}{l}
\texttt{callcc}\, \lambda\kappa. \\
\texttt{updateCode}[L_\kappa \mapsto \kappa_0\bullet] \ \star_D \ \lambda\_. \\
\texttt{updateCode}[L_c \mapsto \varphi_c \ \star_D \ \lambda\_.\texttt{jump}\, L_\kappa] \ \star_D \ \lambda\_. \\
\quad \varphi_c \ \star_D \ \lambda\_.x \ \star_D \ \kappa
\end{array}\right) \\
\mathit{bad}_3 & = & \left(\begin{array}{l}
\texttt{callcc}\, \lambda\kappa. \\
\texttt{updateCode}[L_\kappa \mapsto \kappa_0\bullet] \ \star_D \ \lambda\_. \\
\quad \mathsf{Obs}(\mathsf{FreshLabel}(L_c), \mathit{good}_1, \mathit{bad}_1)
\end{array}\right)
\end{array}$$

$\boxed{\text{Observation Introduction:}}$

$$= \ddots \; \mathsf{Obs}(\mathsf{FreshLabel}(L), \mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLabel}(L), \\ \quad \left(\begin{array}{l} \varphi_b \ \star_D \ \lambda\beta. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L_\kappa), good_3, bad_3) \end{array}\right), \\ \quad \left(\begin{array}{l} \varphi_b \ \star_D \ \lambda\beta. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L_\kappa), good_3, bad_3) \end{array}\right) \end{array}\right), z)$$

$\boxed{\text{By the induction hypothesis for } b\text{:}}$

$$= \ddots \; \mathsf{Obs}(\mathsf{FreshLabel}(L), \mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLabel}(L), \\ \quad \varphi_b \ \star_D \ \lambda\beta.good_3, \\ \quad \left(\begin{array}{l} \varphi_b \ \star_D \ \lambda\beta. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L_\kappa), good_3, bad_3) \end{array}\right) \end{array}\right), z)$$

$\boxed{\text{By Observation Elimination:}}$

$$= \ddots \; \mathsf{Obs}(\mathsf{FreshLabel}(L), \varphi_b \ \star_D \ \lambda\beta.good_3, z)$$

Therefore, done with subcase $\beta\langle \varphi_c \ \star_D \ \lambda\_.\kappa\bullet, \kappa\bullet\rangle = \varphi_c \ \star_D \ \lambda\_.\kappa\bullet$.

Subcase: $\beta\langle\varphi_c \;\star_D\; \lambda\_.\kappa\bullet, \kappa\bullet\rangle = \kappa\bullet$:

$$
= \;\ddots
$$

$$
\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\left(\begin{array}{l}
\varphi_b \;\star_D\; \lambda\beta. \\
\texttt{callcc}\;\lambda\kappa. \\
\quad \texttt{updateCode}[L_\kappa \mapsto \kappa\bullet] \;\star_D\; \lambda\_. \\
\quad \texttt{updateCode}[L_c \mapsto \varphi_c \;\star_D\; \lambda\_.\mathsf{jump}\,L_\kappa] \;\star_D\; \lambda\_. \\
\qquad \kappa\bullet
\end{array}\right) \\
\qquad\qquad \star_D\; \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y)\right), z)
$$

By Axiom 3.2 where $\kappa_0 = \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \;\star_D\; \kappa$:

$$
= \;\ddots
$$

$$
\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l}
\varphi_b \;\star_D\; \lambda\beta. \\
\texttt{callcc}\;\lambda\kappa. \\
\quad \texttt{updateCode}[L_\kappa \mapsto \kappa_0\bullet] \;\star_D\; \lambda\_. \\
\quad \texttt{updateCode}[L_c \mapsto \varphi_c \;\star_D\; \lambda\_.\mathsf{jump}\,L_\kappa] \;\star_D\; \lambda\_. \\
\qquad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \;\star_D\; \lambda\_.\kappa\bullet
\end{array}\right), z)
$$

By Observation Introduction:

$$= \ddots$$

$$\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l}
\varphi_b \star_D \lambda\beta. \\
\texttt{callcc}\ \lambda\kappa. \\
\quad \texttt{updateCode}[L_\kappa \mapsto \kappa_0\bullet] \star_D \lambda\_. \\
\quad \texttt{updateCode}[L_c \mapsto \varphi_c \star_D \lambda\_.\texttt{jump}\, L_\kappa] \star_D \lambda\_. \\
\quad \mathsf{Obs}\left(\begin{array}{l}
\mathsf{FreshLabel}(L_1), \\
\quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \star_D \lambda\_.\kappa\bullet, \\
\quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \star_D \lambda\_.\kappa\bullet
\end{array}\right)
\end{array}\right), z)$$

Recall that, by construction, $L_1 \leq L'$, and so

$$\mathsf{FreshLabel}(L_1) \Rightarrow \mathsf{FreshLabel}(L') = \mathbf{unit}_D(\texttt{true})$$

By Observation Elimination:

$$= \ddots$$

$$\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l}
\varphi_b \star_D \lambda\beta. \\
\texttt{callcc}\ \lambda\kappa. \\
\quad \texttt{updateCode}[L_\kappa \mapsto \kappa_0\bullet] \star_D \lambda\_. \\
\quad \texttt{updateCode}[L_c \mapsto \varphi_c \star_D \lambda\_.\texttt{jump}\, L_\kappa] \star_D \lambda\_. \\
\quad \mathsf{Obs}\left(\begin{array}{l}
\mathsf{FreshLabel}(L_1), \\
\quad x \star_D \lambda\_.\kappa\bullet, \\
\quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \star_D \lambda\_.\kappa\bullet
\end{array}\right)
\end{array}\right), z)$$

This subcase now proceeds precisely as the previous subcase.

$\boxed{\text{Case } t = \textbf{while } b \textbf{ do } c}$

$$\begin{array}{l}
\texttt{getLabel} \star_S \ \lambda L. \\
\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] \ \star_S \ \lambda \varphi_t. \\
\texttt{getLabel} \star_S \ \lambda L'. \\
\quad \textbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L), [\varphi_t \ \star_D \ \lambda v.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y)], z))
\end{array}$$

$\boxed{\text{By the definition of } \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] \text{ and the right unit law:}}$

$$\begin{array}{rl}
= & \texttt{getLabel} \star_S \ \lambda L. \\
& \texttt{newlabel} \star_S \ \lambda L_\kappa. \\
& \texttt{newlabel} \star_S \ \lambda L_c. \\
& \texttt{newlabel} \star_S \ \lambda L_{test}. \\
& \mathcal{C}[\![b]\!] \ \star_S \ \lambda \varphi_b. \\
& \mathcal{C}[\![c]\!] \ \star_S \ \lambda \varphi_c. \\
& \texttt{getLabel} \star_S \ \lambda L'. \\
& \quad \textbf{unit}_S(\mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLabel}(L), \\ \qquad \left(\begin{array}{l} \mathsf{WhilePS}(\varphi_b, \varphi_c, L_{test}, L_c, L_\kappa) \ \star_D \ \lambda\_. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array} \right), \\ \qquad\qquad z \end{array} \right))
\end{array}$$

By the innocence of `getLabel`:

$$
\begin{aligned}
= \;& \texttt{getLabel} \star_S \lambda L. \\
& \texttt{newlabel} \star_S \lambda L_\kappa. \\
& \texttt{newlabel} \star_S \lambda L_c. \\
& \texttt{newlabel} \star_S \lambda L_{test}. \\
& \texttt{getLabel} \star_S \lambda L_1. \\
& \mathcal{C}[\![b]\!] \star_S \lambda \varphi_b. \\
& \texttt{getLabel} \star_S \lambda L_2. \\
& \mathcal{C}[\![c]\!] \star_S \lambda \varphi_c. \\
& \texttt{getLabel} \star_S \lambda L'. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}\left(\begin{array}{l}
\mathsf{FreshLabel}(L), \\
\qquad \left(\begin{array}{l}
\mathsf{WhilePS}(\varphi_b, \varphi_c, L_{test}, L_c, L_\kappa) \star_D \lambda\_. \\
\quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y)
\end{array}\right), \\
\qquad\qquad z
\end{array}\right))
\end{aligned}
$$

For the sake of clarity, the following context will be abbreviated by “$\ddots$” for the remainder of this proof:

$$
\begin{aligned}
\mathit{context}[\circ] = \;& \texttt{getLabel} \star_S \lambda L. \\
& \texttt{newlabel} \star_S \lambda L_\kappa. \\
& \texttt{newlabel} \star_S \lambda L_c. \\
& \texttt{newlabel} \star_S \lambda L_{test}. \\
& \texttt{getLabel} \star_S \lambda L_1. \\
& \mathcal{C}[\![b]\!] \star_S \lambda \varphi_b. \\
& \texttt{getLabel} \star_S \lambda L_2. \\
& \mathcal{C}[\![c]\!] \star_S \lambda \varphi_c. \\
& \texttt{getLabel} \star_S \lambda L'. \\
& \quad \mathbf{unit}_S(\circ)
\end{aligned}
$$

$\boxed{\text{By the definition of } \mathsf{WhilePS}(\varphi_b, \varphi_c, L_{test}, L_c, L_\kappa)\text{:}}$

$$= \ddots$$

$$\mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLabel}(L),\\ \quad\left(\begin{array}{l}\left(\begin{array}{l}\texttt{callcc}\ \lambda\kappa.\\ \quad \texttt{updateCode}[L_\kappa \mapsto \kappa\bullet]\ \star_D\ \lambda\_.\\ \quad \texttt{updateCode}[L_c \mapsto \varphi_c\ \star_D\ (\texttt{jump}\, L_{test})]\ \star_D\ \lambda\_.\\ \quad \texttt{updateCode}[L_{test} \mapsto \varphi_b\ \star_D\ \lambda\beta.\beta\langle L_c, L_{test}\rangle]\ \star_D\ \lambda\_.\\ \qquad \texttt{jump}\, L_{test}\end{array}\right)\star_D\ \lambda\_.\\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y)\end{array}\right),\\ \quad z\end{array}\right)$$

Focusing now on $\texttt{jump}\, L_{test}$ within the above expression, it may be "unrolled" into one of the following (using Axioms 2.1 and 2.2):

$$\begin{array}{l}\varphi_b\ \star_D\ \lambda\_.\kappa\bullet\\ \varphi_b\ \star_D\ \lambda\_.\varphi_c\ \star_D\ \lambda\_.\varphi_b\ \star_D\ \lambda\_.\kappa\bullet\\ \varphi_b\ \star_D\ \lambda\_.\varphi_c\ \star_D\ \lambda\_.\varphi_b\ \star_D\ \lambda\_.\varphi_c\ \star_D\ \lambda\_.\varphi_b\ \star_D\ \lambda\_.\kappa\bullet\\ \varphi_b\ \star_D\ \lambda\_.\varphi_c\ \star_D\ \lambda\_.\varphi_b\ \star_D\ \lambda\_.\varphi_c\ \star_D\ \lambda\_.\varphi_b\ \star_D\ \lambda\_.\varphi_c\ \star_D\ \lambda\_.\varphi_b\ \star_D\ \lambda\_.\kappa\bullet\\ \qquad\qquad\qquad\qquad\qquad\vdots\end{array}$$

which may be neatly summarized (with a slight abuse of language) as the regular expression:

$$[\varphi_b\ \star_D\ \lambda\_.\varphi_c]^*\ \star_D\ \lambda\_.\varphi_b\ \star_D\ \lambda\_.\kappa\bullet$$

The "smallest" case, $\varphi_b\ \star_D\ \lambda\_.\kappa\bullet$, is handled in precisely the same manner as the $\beta = \pi_2$ case for the **if-then** term, the other cases are handled analogously to the $\beta = \pi_1$ case. But what if the loop does not terminate? In this case, $\texttt{jump}\, L_{test} = \perp_{\mathsf{Dynam}(\texttt{void})}$.

$\boxed{\text{If } \texttt{jump}\, L_{test} = \bot_{\mathsf{Dynam}(\texttt{void})}\text{:}}$

$$= \ddots$$

$$\mathsf{Obs}\left(\begin{array}{l}
\mathsf{FreshLabel}(L),\\
\quad\left(\begin{array}{l}
\left(\begin{array}{l}
\texttt{callcc}\ \lambda\kappa.\\
\quad \texttt{updateCode}[L_\kappa \mapsto \kappa\bullet]\ \star_D\ \lambda\_.\\
\quad \texttt{updateCode}[L_c \mapsto \varphi_c\ \star_D\ \lambda\_.(\texttt{jump}\, L_{test})]\ \star_D\ \lambda\_.\\
\quad \texttt{updateCode}[L_{test} \mapsto \varphi_b\ \star_D\ \lambda\beta.\beta\langle L_c, L_{test}\rangle]\ \star_D\ \lambda\_.\\
\qquad \bot_{\mathsf{Dynam}(\texttt{void})}
\end{array}\right) \star_D\ \lambda\_.\\
\quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y)
\end{array}\right),\\
\quad z
\end{array}\right)$$

$$= \ddots$$

$$\mathsf{Obs}(\mathsf{FreshLabel}(L), \bot_{\mathsf{Dynam}(\texttt{void})}\ \star_D\ \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y), z)$$

$\boxed{\text{Because } \bot_{\mathsf{Dynam}(\texttt{void})}\ \star_D\ \lambda\_.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) = \bot_{\mathsf{Dynam}(\texttt{void})}\ \star_D\ \lambda\_.x\text{:}}$

$$= \ddots$$

$$\mathsf{Obs}(\mathsf{FreshLabel}(L), \bot_{\mathsf{Dynam}(\texttt{void})}\ \star_D\ \lambda\_.x, z)$$

□Lemma 9

B.8 Proof of Lemma 10

Lemma 10 (Compilations Preserve Label Freshness)

$$\begin{array}{l}
\mathcal{C}[\![t]\!] \star_S \lambda\varphi_t. \\
\texttt{getLabel} \star_S \lambda L'. \\
\quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L'), \varphi_t \star_D \lambda v.\mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y), -) \\
\qquad\qquad\qquad = \mathcal{C}[\![t]\!] \star_S \lambda\varphi_t. \\
\qquad\qquad\qquad\quad \texttt{getLabel} \star_S \lambda L'. \\
\qquad\qquad\qquad\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L'), \varphi_t \star_D \lambda v.x, -)
\end{array}$$

Proof of Lemma 10

First we note that, for any $L : \mathit{Label}$, the following equivalence holds:

$$\mathsf{FreshLabel}(L) = \mathbf{unit}_D(\texttt{true}) \Leftrightarrow \forall L^* : \mathit{Label}.\ (\mathbf{unit}_D(L \leq L^*) \Rightarrow \mathsf{FreshLabel}(L^*))$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall L^* : \mathit{Label}.\ \mathbf{unit}_D(L \leq L^*) \Rightarrow \mathsf{FreshLabel}(L^*)} \\
\Leftrightarrow & \forall L^* : \mathit{Label}. \\
& \quad \mathbf{unit}_D(L \leq L^*) \star_D \lambda p. \\
& \quad \texttt{getCode} \star_D \lambda\Pi. \\
& \qquad \mathbf{unit}_D(p \supset \forall L' \geq L^*.L' \notin \mathit{dom}\Pi) \\
\Leftrightarrow & \forall L^* : \mathit{Label}. \\
& \quad \texttt{getCode} \star_D \lambda\Pi. \\
& \qquad \mathbf{unit}_D(L \leq L^* \supset \forall L' \geq L^*.L' \notin \mathit{dom}\Pi) \\
\Leftrightarrow & \forall L^* : \mathit{Label}. \\
& \quad \texttt{getCode} \star_D \lambda\Pi. \\
& \qquad \mathbf{unit}_D(\forall L' \geq L.L' \notin \mathit{dom}\Pi) \\
\Leftrightarrow & \forall L^* : \mathit{Label}.\mathsf{FreshLabel}(L)
\end{array}$$

$$\Leftrightarrow \quad \mathsf{FreshLabel}(L)$$

Restating Lemma 9:

$$\begin{array}{l}
\texttt{getLabel} \star_S \lambda L. \\
\mathcal{C}[\![t]\!] \star_S \lambda \varphi_t. \\
\texttt{getLabel} \star_S \lambda L'. \\
\quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L), \left(\begin{array}{l} \varphi_t \star_D \lambda v. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array} \right), z))
\end{array}$$

$$\begin{array}{rl}
= & \texttt{getLabel} \star_S \lambda L. \\
& \mathcal{C}[\![t]\!] \star_S \lambda \varphi_t. \\
& \texttt{getLabel} \star_S \lambda L'. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L), \varphi_t \star_D \lambda v.x, z))
\end{array}$$

Using the above equivalence:

$$\begin{array}{l}
\forall L^* : \mathit{Label}. \\
\texttt{getLabel} \star_S \lambda L. \\
\mathcal{C}[\![t]\!] \star_S \lambda \varphi_t. \\
\texttt{getLabel} \star_S \lambda L'. \\
\quad \mathbf{unit}_S(\mathsf{Obs}(\mathbf{unit}_D(L^* \geq L) \Rightarrow \mathsf{FreshLabel}(L^*), \left(\begin{array}{l} \varphi_t \star_D \lambda v. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array} \right), z))
\end{array}$$

$$\begin{array}{rl}
= & \texttt{getLabel} \star_S \lambda L. \\
& \mathcal{C}[\![t]\!] \star_S \lambda \varphi_t. \\
& \texttt{getLabel} \star_S \lambda L'. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathbf{unit}_D(L^* \geq L) \Rightarrow \mathsf{FreshLabel}(L^*), \varphi_t \star_D \lambda v.x, z))
\end{array}$$

Picking a particular L^* at which the previous equality holds (namely, $L^* = L'$):

$$\begin{array}{l} \texttt{getLabel} \;\star_S\; \lambda L. \\ \mathcal{C}[\![t]\!] \;\star_S\; \lambda\varphi_t. \\ \texttt{getLabel} \;\star_S\; \lambda L'. \\ \quad \mathbf{unit}_S(\mathsf{Obs}(\mathbf{unit}_D(L' \geq L) \Rightarrow \mathsf{FreshLabel}(L'), \begin{pmatrix} \varphi_t \;\star_D\; \lambda v. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{pmatrix}, z)) \end{array}$$

$$\begin{array}{rl} = & \texttt{getLabel} \;\star_S\; \lambda L. \\ & \mathcal{C}[\![t]\!] \;\star_S\; \lambda\varphi_t. \\ & \texttt{getLabel} \;\star_S\; \lambda L'. \\ & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathbf{unit}_D(L' \geq L) \Rightarrow \mathsf{FreshLabel}(L'), \varphi_t \;\star_D\; \lambda v.x, z)) \end{array}$$

By construction, $L' \geq L$, and hence:

$$\begin{array}{l} \texttt{getLabel} \;\star_S\; \lambda L. \\ \mathcal{C}[\![t]\!] \;\star_S\; \lambda\varphi_t. \\ \texttt{getLabel} \;\star_S\; \lambda L'. \\ \quad \mathbf{unit}_S(\mathsf{Obs}(\mathbf{unit}_D(\texttt{true}) \Rightarrow \mathsf{FreshLabel}(L'), \begin{pmatrix} \varphi_t \;\star_D\; \lambda v. \\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{pmatrix}, z)) \end{array}$$

$$\begin{array}{rl} = & \texttt{getLabel} \;\star_S\; \lambda L. \\ & \mathcal{C}[\![t]\!] \;\star_S\; \lambda\varphi_t. \\ & \texttt{getLabel} \;\star_S\; \lambda L'. \\ & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathbf{unit}_D(\texttt{true}) \Rightarrow \mathsf{FreshLabel}(L'), \varphi_t \;\star_D\; \lambda v.x, z)) \end{array}$$

Therefore:

$$\begin{array}{l}
\texttt{getLabel} \star_S \lambda L.\\
\mathcal{C}[\![t]\!] \star_S \lambda \varphi_t.\\
\texttt{getLabel} \star_S \lambda L'.\\
\quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L'), \left(\begin{array}{l} \varphi_t \star_D \lambda v.\\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array}\right), z))
\end{array}$$

$$\begin{array}{rl}
= & \texttt{getLabel} \star_S \lambda L.\\
& \mathcal{C}[\![t]\!] \star_S \lambda \varphi_t.\\
& \texttt{getLabel} \star_S \lambda L'.\\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathbf{unit}_D(\mathsf{FreshLabel}(L'), \varphi_t \star_D \lambda v.x, z))
\end{array}$$

Because **`getLabel`** is innocent and L has been eliminated from the rest of the term:

$$\begin{array}{l}
\mathcal{C}[\![t]\!] \star_S \lambda \varphi_t.\\
\texttt{getLabel} \star_S \lambda L'.\\
\quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLabel}(L'), \left(\begin{array}{l} \varphi_t \star_D \lambda v.\\ \quad \mathsf{Obs}(\mathsf{FreshLabel}(L'), x, y) \end{array}\right), z))
\end{array}$$

$$\begin{array}{rl}
= & \mathcal{C}[\![t]\!] \star_S \lambda \varphi_t.\\
& \texttt{getLabel} \star_S \lambda L'.\\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathbf{unit}_D(\mathsf{FreshLabel}(L'), \varphi_t \star_D \lambda v.x, z))
\end{array}$$

□Lemma 10.

B.9 Proof of Lemma 11

Lemma 11 (Assignment Lemma) *Under appropriate conditions, the compilation produced for an assignment, φ, is identical to the dynamic part of $\mathcal{S}[\![x{:=}e]\!]$:*

$$\begin{aligned}\mathcal{C}[\![x{:=}e]\!] = \ & \texttt{rdAddr} \star_S \lambda a. \\ & \mathcal{S}[\![x{:=}e]\!] \star_S \lambda\delta. \\ & \mathcal{C}[\![x{:=}e]\!] \star_S \lambda\varphi. \\ & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta, \varphi))\end{aligned}$$

$\boxed{\text{Proof of Lemma 11}}$

$$\begin{aligned}\mathcal{C}[\![x{:=}e]\!] = \ & \texttt{rdEnv} \star_S \lambda\rho. \\ & (\rho\, x) \star_S \lambda a_x : \mathit{Addr}. \\ & \mathcal{C}[\![e]\!] \star_S \lambda\varphi_e. \\ & \quad \mathbf{unit}_S(\varphi_e \star_S \lambda i.\texttt{store}(a_x, i))\end{aligned}$$

$\boxed{\text{By the innocence of } \texttt{rdAddr} \text{ and } \mathcal{S}[\![x{:=}e]\!]\text{, and observation introduction:}}$

$$\begin{aligned} = \ & \texttt{rdAddr} \star_S \lambda a. \\ & \mathcal{S}[\![x{:=}e]\!] \star_S \lambda\delta_c. \\ & \texttt{rdEnv} \star_S \lambda\rho. \\ & (\rho\, x) \star_S \lambda a_x : \mathit{Addr}. \\ & \mathcal{C}[\![e]\!] \star_S \lambda\varphi_e. \\ & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi_e \star_S \lambda i.\texttt{store}(a_x, i), \varphi_e \star_S \lambda i.\texttt{store}(a_x, i)))\end{aligned}$$

By the innocence of $\mathcal{S}\llbracket x{:=}e \rrbracket$, it commutes with $(\rho\, x)$ and $\texttt{rdEnv}$:

$$
\begin{aligned}
= \ & \texttt{rdAddr} \star_S \ \lambda a. \\
& \texttt{rdEnv} \star_S \ \lambda \rho. \\
& (\rho\, x) \star_S \ \lambda a_x : \mathit{Addr}. \\
& \mathcal{S}\llbracket e \rrbracket \star_S \ \lambda \delta_e. \\
& \mathcal{C}\llbracket e \rrbracket \star_S \ \lambda \varphi_e. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi_e \star_S \ \lambda i.\texttt{store}(a_x, i), \varphi_e \star_S \ \lambda i.\texttt{store}(a_x, i)))
\end{aligned}
$$

By Lemma 1 on page 85:

$$
\begin{aligned}
= \ & \texttt{rdAddr} \star_S \ \lambda a. \\
& \texttt{rdEnv} \star_S \ \lambda \rho. \\
& (\rho\, x) \star_S \ \lambda a_x : \mathit{Addr}. \\
& \mathcal{S}\llbracket e \rrbracket \star_S \ \lambda \delta_e. \\
& \mathcal{C}\llbracket e \rrbracket \star_S \ \lambda \varphi_e. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi_e, \varphi_e) \star_S \ \lambda i.\texttt{store}(a_x, i))
\end{aligned}
$$

Assuming **Exp-spec**:

$$
\begin{aligned}
= \ & \texttt{rdAddr} \star_S \ \lambda a. \\
& \texttt{rdEnv} \star_S \ \lambda \rho. \\
& (\rho\, x) \star_S \ \lambda a_x : \mathit{Addr}. \\
& \mathcal{S}\llbracket e \rrbracket \star_S \ \lambda \delta_e. \\
& \mathcal{C}\llbracket e \rrbracket \star_S \ \lambda \varphi_e. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_e, \varphi_e) \star_S \ \lambda i.\texttt{store}(a_x, i)) \\
= \ & \texttt{rdAddr} \star_S \ \lambda a. \\
& \texttt{rdEnv} \star_S \ \lambda \rho. \\
& (\rho\, x) \star_S \ \lambda a_x : \mathit{Addr}. \\
& \mathcal{S}\llbracket e \rrbracket \star_S \ \lambda \delta_e. \\
& \mathcal{C}\llbracket e \rrbracket \star_S \ \lambda \varphi_e. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_e \star_S \ \lambda i.\texttt{store}(a_x, i), \varphi_e \star_S \ \lambda i.\texttt{store}(a_x, i)))
\end{aligned}
$$

Folding back with the definitions of $\mathcal{S}\llbracket x{:=}e \rrbracket$ and $\mathcal{C}\llbracket x{:=}e \rrbracket$:

$$
\begin{aligned}
= \;& \texttt{rdAddr} \star_S \; \lambda a. \\
& \texttt{rdEnv} \star_S \; \lambda \rho. \\
& (\rho\, x) \; \star_S \; \lambda a_x : Addr. \\
& \mathcal{S}[\![x{:=}e]\!] \; \star_S \; \lambda \delta_c. \\
& \mathcal{C}[\![x{:=}e]\!] \; \star_S \; \lambda \varphi_c. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_c, \varphi_c))
\end{aligned}
$$

□Lemma 11

B.10 Proof of Theorem 3

Theorem 3 **Exp-spec** *holds for the expression block* $\mathcal{C}\llbracket - : Exp \rrbracket$

$$\begin{array}{rl} \mathcal{C}\llbracket e \rrbracket = & \texttt{rdAddr} \star_S \lambda a. \\ & \mathcal{S}\llbracket e \rrbracket \star_S \lambda \delta_e. \\ & \mathcal{C}\llbracket e \rrbracket \star_S \lambda \varphi_e. \\ & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_e, \varphi_e)) \end{array}$$

Proof of Theorem 3:

By induction on terms.

Because $\mathcal{C}\llbracket i \rrbracket = \mathcal{S}\llbracket i \rrbracket$ for integer constant i, the theorem holds trivially.

Case: e is a negation:

By the right unit law, and then, by the innocence of `rdAddr` and observation introduction:

$$\begin{array}{rlrl} \mathcal{C}\llbracket -e \rrbracket = & \mathcal{C}\llbracket -e \rrbracket \star_S \lambda \varphi. & = & \texttt{rdAddr} \star_S \lambda a. \\ & \quad \mathbf{unit}_S(\varphi) & & \mathcal{C}\llbracket -e \rrbracket \star_S \lambda \varphi. \\ & & & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi, \varphi)) \end{array}$$

Unfolding the definition of $\mathcal{C}\llbracket -e \rrbracket$:

$$\begin{array}{rl} = & \texttt{rdAddr} \star_S \lambda a. \\ & \left(\begin{array}{l} (\texttt{inAddr}\ (a+1)\ \mathcal{C}\llbracket e \rrbracket) \star_S \lambda \varphi_e. \\ \quad \mathbf{unit}_S(\mathsf{Negate}(\varphi_e, a)) \end{array} \right) \star_S \lambda \varphi. \\ & \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi, \varphi)) \end{array}$$

By the induction hypothesis and simplification:

$$
\begin{aligned}
= {} & \texttt{rdAddr} \star_S \lambda a. \\
& \left(\begin{array}{l} (\texttt{inAddr}\ (a+1) \left[\begin{array}{l} \mathcal{S}[\![e]\!] \star_S \lambda\delta_e. \\ \mathcal{C}[\![e]\!] \star_S \lambda\gamma. \\ \qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a+1), \delta_e, \gamma)) \end{array} \right]) \star_S \lambda\varphi_e. \\ \quad \mathbf{unit}_S(\mathsf{Negate}(\varphi_e, a)) \end{array} \right) \star_S \lambda\varphi. \\
& \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi, \varphi))
\end{aligned}
$$

Using $(\texttt{inAddr}\ a\ x) \star \lambda y.\mathbf{unit}(z) = \texttt{inAddr}\ a\ (x \star \lambda y.\mathbf{unit}(z))$ and associativity :

$$
\begin{aligned}
= {} & \texttt{rdAddr} \star_S \lambda a. \\
& \texttt{inAddr}\ (a+1) \\
& \quad \mathcal{S}[\![e]\!] \star_S \lambda\delta_e. \\
& \quad \mathcal{C}[\![e]\!] \star_S \lambda\gamma. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a+1), \delta_e, \gamma)) \star_S \lambda\varphi_e. \\
& \quad \mathbf{unit}_S(\mathsf{Negate}(\varphi_e, a)) \star_S \lambda\varphi. \\
& \qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi, \varphi))
\end{aligned}
$$

Using the left unit law twice:

$$
\begin{aligned}
= {} & \texttt{rdAddr} \star_S \lambda a. \\
& \texttt{inAddr}\ (a+1) \\
& \quad \mathcal{S}[\![e]\!] \star_S \lambda\delta_e. \\
& \quad \mathcal{C}[\![e]\!] \star_S \lambda\gamma. \\
& \qquad \mathbf{unit}_S(\mathsf{Obs} \left(\begin{array}{l} \mathsf{FreshLoc}(a), \\ \qquad \mathsf{Negate}(\mathsf{Obs}(\mathsf{FreshLoc}(a+1), \delta_e, \gamma), a), \\ \qquad\qquad \mathsf{Negate}(\mathsf{Obs}(\mathsf{FreshLoc}(a+1), \delta_e, \gamma), a) \end{array} \right))
\end{aligned}
$$

$\boxed{\text{Unfolding definition of } \mathsf{Negate}\text{:}}$

$$
\begin{array}{l}
= \ \texttt{rdAddr} \ \star_S \ \lambda a. \\
\quad \texttt{inAddr} \ (a+1) \\
\qquad \mathcal{S}\llbracket e \rrbracket \ \star_S \ \lambda \delta_e. \\
\qquad \mathcal{C}\llbracket e \rrbracket \ \star_S \ \lambda \gamma. \\
\qquad\qquad \mathbf{unit}_S(\mathsf{Obs} \left(
\begin{array}{l}
\mathsf{FreshLoc}(a), \\
\qquad \left(
\begin{array}{l}
\mathsf{Obs}(\mathsf{FreshLoc}(a+1), \delta_e, \gamma) \ \star_D \ \lambda i. \\
\texttt{Alloc}(a) \ \star_D \ \lambda\_. \\
\texttt{Thread}(i, a) \ \star_D \ \lambda v. \\
\texttt{deAlloc}(a) \ \star_D \ \lambda\_. \\
\qquad \mathbf{unit}_D(-v)
\end{array}
\right), \\
\qquad\quad \mathsf{Negate}(\mathsf{Obs}(\mathsf{FreshLoc}(a+1), \delta_e, \gamma), a)
\end{array}
\right))
\end{array}
$$

$\boxed{\text{By observation elimination:}}$

$$
\begin{array}{l}
= \ \texttt{rdAddr} \ \star_S \ \lambda a. \\
\quad \texttt{inAddr} \ (a+1) \\
\qquad \mathcal{S}\llbracket e \rrbracket \ \star_S \ \lambda \delta_e. \\
\qquad \mathcal{C}\llbracket e \rrbracket \ \star_S \ \lambda \gamma. \\
\qquad\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \left(
\begin{array}{l}
\delta_e \ \star_D \ \lambda i. \\
\texttt{Alloc}(a) \ \star_D \ \lambda\_. \\
\texttt{Thread}(i, a) \ \star_D \ \lambda v. \\
\texttt{deAlloc}(a) \ \star_D \ \lambda\_. \\
\qquad \mathbf{unit}_D(-v)
\end{array}
\right), \mathsf{Negate}(\mathsf{Obs}(\mathsf{FreshLoc}(a+1), \delta_e, \gamma), a)))
\end{array}
$$

$\boxed{\text{By the innocence of } \delta_e\text{:}}$

$$
\begin{aligned}
= \ & \texttt{rdAddr} \star_S \ \lambda a.\\
& \texttt{inAddr}\ (a+1)\\
& \quad \mathcal{S}[\![e]\!] \ \star_S \ \lambda\delta_e.\\
& \quad \mathcal{C}[\![e]\!] \ \star_S \ \lambda\gamma.\\
& \qquad \mathbf{unit}_S(\delta_e \ \star_D \ \lambda i.(\mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLoc}(a),\\ \qquad \left(\begin{array}{l}\texttt{Alloc}(a) \ \star_D \ \lambda\_.\\ \texttt{Thread}(i,a) \ \star_D \ \lambda v.\\ \texttt{deAlloc}(a) \ \star_D \ \lambda\_.\\ \qquad \mathbf{unit}_D(-v)\end{array}\right),\\ \qquad\qquad \mathsf{Negate}(\mathsf{Obs}(\mathsf{FreshLoc}(a+1),\delta_e,\gamma),a)\end{array}\right)))
\end{aligned}
$$

$\boxed{\text{By Lemma 3, a new memory cell may be used and discarded without effect:}}$

$$
\begin{aligned}
= \ & \texttt{rdAddr} \star_S \ \lambda a.\\
& \texttt{inAddr}\ (a+1)\\
& \quad \mathcal{S}[\![e]\!] \ \star_S \ \lambda\delta_e.\\
& \quad \mathcal{C}[\![e]\!] \ \star_S \ \lambda\gamma.\\
& \qquad \mathbf{unit}_S(\delta_e \ \star_D \ \lambda i.(\mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLoc}(a),\\ \qquad \mathbf{unit}_D(-i),\\ \qquad\qquad \mathsf{Negate}(\mathsf{Obs}(\mathsf{FreshLoc}(a+1),\delta_e,\gamma),a)\end{array}\right)))
\end{aligned}
$$

$\boxed{\text{Because of the innocence of } \delta_e\text{:}}$

$$\begin{aligned} = {} & \texttt{rdAddr} \star_S \lambda a. \\ & \quad \texttt{inAddr}\ (a+1) \\ & \qquad \mathcal{S}[\![e]\!] \star_S \lambda\delta_e. \\ & \qquad \mathcal{C}[\![e]\!] \star_S \lambda\gamma. \\ & \qquad\quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_e \star_D \lambda i.\mathbf{unit}_D(-i), \mathsf{Negate}(\mathsf{Obs}(\mathsf{FreshLoc}(a+1), \delta_e, \gamma), a))) \end{aligned}$$

$\boxed{\text{Innocence of } \mathcal{S}[\![e]\!]\text{:}}$

$$\begin{aligned} = {} & \texttt{rdAddr} \star_S \lambda a. \\ & \quad \mathcal{S}[\![e]\!] \star_S \lambda\delta_e. \\ & \quad \texttt{inAddr}\ (a+1) \\ & \qquad \mathcal{C}[\![e]\!] \star_S \lambda\gamma. \\ & \qquad\quad \mathbf{unit}_S(\mathsf{Obs}\left(\begin{array}{l} \mathsf{FreshLoc}(a), \\ \qquad \delta_e \star_D \lambda i.\mathbf{unit}_D(-i), \\ \qquad\qquad \mathsf{Negate}(\mathsf{Obs}(\mathsf{FreshLoc}(a+1), \delta_e \star_D \lambda i.\mathbf{unit}_D(-i), \gamma), a) \end{array}\right)) \end{aligned}$$

$\boxed{\text{Left unit:}}$

$$\begin{aligned} = {} & \texttt{rdAddr} \star_S \lambda a. \\ & \quad \mathcal{S}[\![e]\!] \star_S \lambda\delta_e. \\ & \quad \left(\begin{array}{l} \texttt{inAddr}\ (a+1) \\ \mathcal{C}[\![e]\!] \star_S \lambda\gamma. \\ \quad \mathbf{unit}_D(\mathsf{Negate}(\mathsf{Obs}(\mathsf{FreshLoc}(a+1), \delta_e \star_D \lambda i.\mathbf{unit}_D(-i), \gamma), a)) \end{array}\right) \star_S \lambda\varphi. \\ & \qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_e \star_D \lambda i.\mathbf{unit}_D(-i), \varphi)) \end{aligned}$$

Working backwards from inductive hypothesis, definition of Negate, and left unit:

$$\begin{array}{ll}
\begin{array}{ll}
= & \mathtt{rdAddr} \star_S \ \lambda a. \\
 & \mathcal{S}[\![e]\!] \star_S \ \lambda \delta_e. \\
 & \mathcal{C}[\![-e]\!] \star_S \ \lambda \varphi. \\
 & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_e \star_D \ \lambda i.\mathbf{unit}_D(-i), \varphi))
\end{array}
&
\begin{array}{ll}
= & \mathtt{rdAddr} \star_S \ \lambda a. \\
 & \mathcal{S}[\![-e]\!] \star_S \ \lambda \delta. \\
 & \mathcal{C}[\![-e]\!] \star_S \ \lambda \varphi. \\
 & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta, \varphi))
\end{array}
\end{array}$$

□

B.11 Proof of Theorem 4

Theorem 4 *For terms c of the Imp block:*

$$\begin{aligned}
&\mathcal{C}\llbracket c\rrbracket \star_S \lambda\varphi_c.\mathbf{unit}_S(\varphi_c \star_D \lambda\_.\mathtt{init}_{CS})\\
&= \mathtt{rdAddr} \star_S \lambda a.\\
&\quad \mathtt{getLabel} \star_S \lambda L.\\
&\quad \mathcal{S}\llbracket c\rrbracket \star_S \lambda\delta_c.\\
&\quad \mathcal{C}\llbracket c\rrbracket \star_S \lambda\varphi_c.\\
&\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a)\ \mathsf{AND}\ \mathsf{FreshLabel}(L), \delta_c, \varphi_c) \star_D \lambda\_.\mathtt{init}_{CS})
\end{aligned}$$

Proof of Theorem 4

By induction on terms.

Let $\mathsf{Pre}(a, L)$ stand for the observation $\mathsf{FreshLoc}(a)$ AND $\mathsf{FreshLabel}(L)$.

Case 1: c is $x{:=}e$.

$$\begin{aligned}
\mathcal{C}\llbracket x{:=}e\rrbracket \star_S \lambda\varphi_c.\mathbf{unit}_S(\varphi_c \star_D \lambda\_.\mathtt{init}_{CS}) = \;& \mathtt{rdEnv} \star_S \lambda\rho.\\
& (\rho\, x) \star_S \lambda a_x : \mathit{Addr}.\\
& \mathcal{C}\llbracket e\rrbracket \star_S \lambda\varphi_e.\\
& \quad \mathbf{unit}_S(\varphi_e \star_S \lambda i.\mathtt{store}(a_x, i) \star_D \lambda\_.\mathtt{init}_{CS})
\end{aligned}$$

By the innocence of `rdAddr`, `getLabel` and $\mathcal{S}\llbracket x{:=}e\rrbracket$, and observation introduction:

$$\begin{aligned}
&= \mathtt{rdAddr} \star_S \lambda a.\\
&\quad \mathtt{getLabel} \star_S \lambda L.\\
&\quad \mathcal{S}\llbracket x{:=}e\rrbracket \star_S \lambda\delta_c.\\
&\quad \mathtt{rdEnv} \star_S \lambda\rho.\\
&\quad (\rho\, x) \star_S \lambda a_x : \mathit{Addr}.\\
&\quad \mathcal{C}\llbracket e\rrbracket \star_S \lambda\varphi_e.\\
&\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \varphi_e \star_S \lambda i.\mathtt{store}(a_x, i), \varphi_e \star_S \lambda i.\mathtt{store}(a_x, i)) \star_D \lambda\_.\mathtt{init}_{CS})
\end{aligned}$$

By the innocence of $\mathcal{S}[\![e]\!]$, it commutes with $(\rho\, x)$ and **`rdEnv`**:

$$
\begin{aligned}
= \;& \mathtt{rdAddr} \star_S \lambda a. \\
& \mathtt{getLabel} \star_S \lambda L. \\
& \mathtt{rdEnv} \star_S \lambda \rho. \\
& (\rho\, x) \star_S \lambda a_x : \mathit{Addr}. \\
& \mathcal{S}[\![e]\!] \star_S \lambda \delta_e. \\
& \mathcal{C}[\![e]\!] \star_S \lambda \varphi_e. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \varphi_e \star_S \lambda i.\mathtt{store}(a_x, i), \varphi_e \star_S \lambda i.\mathtt{store}(a_x, i)) \star_D \lambda\_.\mathtt{init}_{CS})
\end{aligned}
$$

By Lemma 1 on page 85:

$$
\begin{aligned}
= \;& \mathtt{rdAddr} \star_S \lambda a. \\
& \mathtt{getLabel} \star_S \lambda L. \\
& \mathtt{rdEnv} \star_S \lambda \rho. \\
& (\rho\, x) \star_S \lambda a_x : \mathit{Addr}. \\
& \mathcal{S}[\![e]\!] \star_S \lambda \delta_e. \\
& \mathcal{C}[\![e]\!] \star_S \lambda \varphi_e. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \varphi_e, \varphi_e) \star_S \lambda i.\mathtt{store}(a_x, i) \star_D \lambda\_.\mathtt{init}_{CS})
\end{aligned}
$$

Assuming **Exp-spec**:

$$
\begin{aligned}
= \;& \mathtt{rdAddr} \star_S \lambda a. \\
& \mathtt{getLabel} \star_S \lambda L. \\
& \mathtt{rdEnv} \star_S \lambda \rho. \\
& (\rho\, x) \star_S \lambda a_x : \mathit{Addr}. \\
& \mathcal{S}[\![e]\!] \star_S \lambda \delta_e. \\
& \mathcal{C}[\![e]\!] \star_S \lambda \varphi_e. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \delta_e, \varphi_e) \star_S \lambda i.\mathtt{store}(a_x, i) \star_D \lambda\_.\mathtt{init}_{CS})
\end{aligned}
$$

$$
\begin{aligned}
= {} & \mathtt{rdAddr} \star_S \lambda a. \\
& \mathtt{getLabel} \star_S \lambda L. \\
& \mathtt{rdEnv} \star_S \lambda \rho. \\
& (\rho\, x) \star_S \lambda a_x : \mathit{Addr}. \\
& \mathcal{S}\llbracket e \rrbracket \star_S \lambda \delta_e. \\
& \mathcal{C}\llbracket e \rrbracket \star_S \lambda \varphi_e. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \delta_e \star_S \lambda i.\mathtt{store}(a_x, i), \varphi_e \star_S \lambda i.\mathtt{store}(a_x, i)) \star_D \lambda\_.\mathtt{init}_{CS})
\end{aligned}
$$

Folding back with the definitions of $\mathcal{S}\llbracket x{:=}e \rrbracket$ and $\mathcal{C}\llbracket x{:=}e \rrbracket$:

$$
\begin{aligned}
= {} & \mathtt{rdAddr} \star_S \lambda a. \\
& \mathtt{getLabel} \star_S \lambda L. \\
& \mathtt{rdEnv} \star_S \lambda \rho. \\
& (\rho\, x) \star_S \lambda a_x : \mathit{Addr}. \\
& \mathcal{S}\llbracket x{:=}e \rrbracket \star_S \lambda \delta_c. \\
& \mathcal{C}\llbracket x{:=}e \rrbracket \star_S \lambda \varphi_c. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \delta_c, \varphi_c) \star_D \lambda\_.\mathtt{init}_{CS})
\end{aligned}
$$

Case 2: c is c_1 ; c_2.

$$
\begin{aligned}
\mathcal{C}\llbracket c_1 \,;\, c_2 \rrbracket \star_S \lambda \varphi_c.\mathbf{unit}_S(\varphi_c \star_D \lambda\_.\mathtt{init}_{CS}) = {} & \mathcal{C}\llbracket c_1 \rrbracket \star_S \lambda \varphi_1. \\
& \mathcal{C}\llbracket c_2 \rrbracket \star_S \lambda \varphi_2. \\
& \quad \mathbf{unit}_S(\varphi_1 \star_D \lambda\_.\varphi_2 \star_D \lambda\_.\mathtt{init}_{CS})
\end{aligned}
$$

By the innocence of `rdAddr`, `getLabel` and $\mathcal{S}\llbracket c_i \rrbracket$, and observation introduction:

$$
\begin{aligned}
= {} & \mathtt{rdAddr} \star_S \lambda a. \\
& \mathtt{getLabel} \star_S \lambda L. \\
& \mathcal{S}\llbracket c_1 \rrbracket \star_S \lambda \delta_1. \\
& \mathcal{C}\llbracket c_1 \rrbracket \star_S \lambda \varphi_1. \\
& \mathcal{S}\llbracket c_2 \rrbracket \star_S \lambda \delta_2. \\
& \mathcal{C}\llbracket c_2 \rrbracket \star_S \lambda \varphi_2. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \varphi_1 \star_D \lambda\_.\varphi_2, \varphi_1 \star_D \lambda\_.\varphi_2) \star_D \lambda\_.\mathtt{init}_{CS})
\end{aligned}
$$

By the innocence of `rdAddr`, `getLabel` and $\mathcal{S}\llbracket c_i \rrbracket$, and observation introduction:

$$
\begin{array}{rl}
= & \mathcal{S}\llbracket c_1 \rrbracket \star_S \lambda\delta_1. \\
& \mathcal{S}\llbracket c_2 \rrbracket \star_S \lambda\delta_2. \\
& \mathtt{rdAddr} \star_S \lambda a. \\
& \mathtt{getLabel} \star_S \lambda L. \\
& \mathcal{C}\llbracket c_1 \rrbracket \star_S \lambda\varphi_1. \\
& \mathtt{getLabel} \star_S \lambda L'. \\
& \mathcal{C}\llbracket c_2 \rrbracket \star_S \lambda\varphi_2. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \varphi_1 \star_D \lambda\_.\varphi_2, \varphi_1 \star_D \lambda\_.\varphi_2) \star_D \lambda\_.\mathtt{init}_{CS})
\end{array}
$$

$$
\begin{array}{rl}
= & \mathcal{S}\llbracket c_1 \rrbracket \star_S \lambda\delta_1. \\
& \mathcal{S}\llbracket c_2 \rrbracket \star_S \lambda\delta_2. \\
& \mathtt{rdAddr} \star_S \lambda a. \\
& \mathtt{getLabel} \star_S \lambda L. \\
& \mathcal{C}\llbracket c_1 \rrbracket \star_S \lambda\varphi_1. \\
& \mathtt{getLabel} \star_S \lambda L'. \\
& \mathcal{C}\llbracket c_2 \rrbracket \star_S \lambda\varphi_2. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \varphi_1 \star_D \lambda\_.\varphi_2 \star_D \lambda\_.\mathtt{init}_{CS}, \varphi_1 \star_D \lambda\_.\varphi_2 \star_D \lambda\_.\mathtt{init}_{CS}))
\end{array}
$$

$$
\begin{array}{rl}
= & \mathcal{S}\llbracket c_1 \rrbracket \star_S \lambda\delta_1. \\
& \mathcal{S}\llbracket c_2 \rrbracket \star_S \lambda\delta_2. \\
& \mathtt{rdAddr} \star_S \lambda a. \\
& \mathtt{getLabel} \star_S \lambda L. \\
& \mathcal{C}\llbracket c_1 \rrbracket \star_S \lambda\varphi_1. \\
& \mathtt{getLabel} \star_S \lambda L'. \\
& \mathcal{C}\llbracket c_2 \rrbracket \star_S \lambda\varphi_2. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}\left(\begin{array}{l}
\mathsf{Pre}(a, L), \\
\qquad \varphi_1 \star_D \lambda\_.\mathsf{Obs}(\mathsf{Pre}(a, L'), \varphi_2, \varphi_2) \star_D \lambda\_.\mathtt{init}_{CS}, \\
\qquad\qquad \varphi_1 \star_D \lambda\_.\varphi_2 \star_D \lambda\_.\mathtt{init}_{CS}
\end{array}\right))
\end{array}
$$

Inductive hypothesis for c_2:

$$
\begin{aligned}
= \;& \mathcal{S}[\![c_1]\!] \;\star_S\; \lambda\delta_1. \\
& \mathcal{S}[\![c_2]\!] \;\star_S\; \lambda\delta_2. \\
& \texttt{rdAddr} \;\star_S\; \lambda a. \\
& \texttt{getLabel} \;\star_S\; \lambda L. \\
& \mathcal{C}[\![c_1]\!] \;\star_S\; \lambda\varphi_1. \\
& \texttt{getLabel} \;\star_S\; \lambda L'. \\
& \mathcal{C}[\![c_2]\!] \;\star_S\; \lambda\varphi_2. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}\begin{pmatrix} \mathsf{Pre}(a,L), \\ \varphi_1 \;\star_D\; \lambda\_.(\mathsf{Obs}(\mathsf{Pre}(a,L'),\delta_2,\varphi_2) \;\star_D\; \lambda\_.\texttt{init}_{CS}), \\ \varphi_1 \;\star_D\; \lambda\_.\varphi_2 \;\star_D\; \lambda\_.\texttt{init}_{CS} \end{pmatrix})
\end{aligned}
$$

Lemmas 8 and 9:

$$
\begin{aligned}
= \;& \mathcal{S}[\![c_1]\!] \;\star_S\; \lambda\delta_1. \\
& \mathcal{S}[\![c_2]\!] \;\star_S\; \lambda\delta_2. \\
& \texttt{rdAddr} \;\star_S\; \lambda a. \\
& \texttt{getLabel} \;\star_S\; \lambda L. \\
& \mathcal{C}[\![c_1]\!] \;\star_S\; \lambda\varphi_1. \\
& \texttt{getLabel} \;\star_S\; \lambda L'. \\
& \mathcal{C}[\![c_2]\!] \;\star_S\; \lambda\varphi_2. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}\begin{pmatrix} \mathsf{Pre}(a,L), \\ \varphi_1 \;\star_D\; \lambda\_.\delta_2 \;\star_D\; \lambda\_.\texttt{init}_{CS}, \\ \varphi_1 \;\star_D\; \lambda\_.\varphi_2 \;\star_D\; \lambda\_.\texttt{init}_{CS} \end{pmatrix})
\end{aligned}
$$

From Axiom 3.1, Axiom 3.2, Axiom 4, and Axiom 5, and δ_2 contains no `updateCode`/`getCode`:

$$
\begin{aligned}
= \;& \mathcal{S}[\![c_1]\!] \star_S \lambda\delta_1. \\
& \mathcal{S}[\![c_2]\!] \star_S \lambda\delta_2. \\
& \texttt{rdAddr} \star_S \lambda a. \\
& \texttt{getLabel} \star_S \lambda L. \\
& \mathcal{C}[\![c_1]\!] \star_S \lambda\varphi_1. \\
& \texttt{getLabel} \star_S \lambda L'. \\
& \mathcal{C}[\![c_2]\!] \star_S \lambda\varphi_2. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a, L), \\ \qquad (\varphi_1 \star_D \lambda\_.\texttt{init}_{CS}) \star_D \lambda\_.\delta_2, \\ \qquad\qquad \varphi_1 \star_D \lambda\_.\varphi_2 \star_D \lambda\_.\texttt{init}_{CS} \end{array}\right))
\end{aligned}
$$

$\boxed{\text{Inductive hypothesis for } c_1\text{:}}$

$$
\begin{aligned}
= \;& \mathcal{S}[\![c_1]\!] \star_S \lambda\delta_1. \\
& \mathcal{S}[\![c_2]\!] \star_S \lambda\delta_2. \\
& \texttt{rdAddr} \star_S \lambda a. \\
& \texttt{getLabel} \star_S \lambda L. \\
& \mathcal{C}[\![c_1]\!] \star_S \lambda\varphi_1. \\
& \texttt{getLabel} \star_S \lambda L'. \\
& \mathcal{C}[\![c_2]\!] \star_S \lambda\varphi_2. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a, L), \\ \qquad (\delta_1 \star_D \lambda\_.\texttt{init}_{CS}) \star_D \lambda\_.\delta_2, \\ \qquad\qquad \varphi_1 \star_D \lambda\_.\varphi_2 \star_D \lambda\_.\texttt{init}_{CS} \end{array}\right))
\end{aligned}
$$

$\boxed{\text{Lemma 1 on page 1:}}$

$$
\begin{aligned}
= \;& \mathcal{S}[\![c_1]\!] \;\star_S\; \lambda\delta_1. \\
& \mathcal{S}[\![c_2]\!] \;\star_S\; \lambda\delta_2. \\
& \texttt{rdAddr} \;\star_S\; \lambda a. \\
& \texttt{getLabel} \;\star_S\; \lambda L. \\
& \mathcal{C}[\![c_1]\!] \;\star_S\; \lambda\varphi_1. \\
& \texttt{getLabel} \;\star_S\; \lambda L'. \\
& \mathcal{C}[\![c_2]\!] \;\star_S\; \lambda\varphi_2. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a,L), \delta_1 \;\star_D\; \lambda\_.\delta_2, \varphi_1 \;\star_D\; \lambda\_.\varphi_2) \;\star_D\; \lambda\_.\texttt{init}_{CS}) \\
= \;& \texttt{rdAddr} \;\star_S\; \lambda a. \\
& \texttt{getLabel} \;\star_S\; \lambda L. \\
& \mathcal{S}[\![c_1 \,;\, c_2]\!] \;\star_S\; \lambda\delta_c. \\
& \mathcal{C}[\![c_1 \,;\, c_2]\!] \;\star_S\; \lambda\varphi_c. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a,L), \delta_c, \varphi_c) \;\star_D\; \lambda\_.\texttt{init}_{CS})
\end{aligned}
$$

□Proof of Theorem 4

B.12 Proof of Theorem 5

Theorem 5 **Exp-spec** *also holds for the constant-folding expression block:*

$$\begin{aligned} \mathsf{CF}[\![e]\!] = \ & \mathtt{rdAddr} \star_S \lambda a. \\ & \mathcal{S}[\![e]\!] \star_S \lambda \delta_e. \\ & \mathsf{CF}[\![e]\!] \star_S \lambda \varphi_e. \\ & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_e, \varphi_e)) \end{aligned}$$

$\boxed{\text{Proof of Theorem 5.}}$

By induction on terms.

$$\boxed{\text{Case } e \text{ is } n.}$$

$$\begin{aligned} \mathsf{CF}[\![n]\!] = \ & \mathtt{rdAddr} \star_S \lambda a. \\ & \mathcal{S}[\![e]\!] \star_S \lambda \delta_e. \\ & \mathsf{CF}[\![e]\!] \star_S \lambda \varphi_e. \\ & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi_e, \varphi_e)) \end{aligned}$$

$\boxed{\text{Observe that } \mathsf{CF}[\![n]\!] = \mathbf{unit}_S(\mathbf{unit}_D(n)) = \mathcal{S}[\![n]\!]\text{:}}$

$$\begin{aligned} = \ & \mathtt{rdAddr} \star_S \lambda a. \\ & \mathsf{CF}[\![e]\!] \star_S \lambda \delta_e. \\ & \mathsf{CF}[\![e]\!] \star_S \lambda \varphi_e. \\ & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_e, \varphi_e)) \end{aligned}$$

$$\boxed{\text{Case } e \text{ is } -e.}$$

$$
\begin{aligned}
\mathsf{CF}[\![-e]\!] = \ & \texttt{rdAddr} \star_S \ \lambda a. \\
& \mathcal{S}[\![-e]\!] \star_S \ \lambda \delta. \\
& \mathsf{CF}[\![-e]\!] \star_S \ \lambda \varphi. \\
& \qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi, \varphi))
\end{aligned}
$$

Assuming $\mathsf{constexp}(-e)$.

By Lemma 12, $\mathsf{CF}[\![-e]\!] = \mathsf{boost}([\![-e]\!]) = \mathcal{S}[\![-e]\!]$. As for the case of constants:

$$
\begin{aligned}
\mathsf{CF}[\![-e]\!] = \ & \texttt{rdAddr} \star_S \ \lambda a. \\
& \mathcal{S}[\![-e]\!] \star_S \ \lambda \delta. \\
& \mathsf{CF}[\![-e]\!] \star_S \ \lambda \varphi. \\
& \qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta, \varphi))
\end{aligned}
$$

Assuming not $\mathsf{constexp}(-e)$.

$$
\begin{aligned}
\mathsf{CF}[\![-e]\!] = \ & \texttt{rdAddr} \star_S \ \lambda a. \\
& \mathcal{S}[\![-e]\!] \star_S \ \lambda \delta. \\
& \texttt{inAddr}\ (a+1) \\
& \quad \mathsf{CF}[\![e]\!] \star_S \ \lambda \varphi_e. \\
& \qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \mathsf{Negate}(\varphi_e, a), \mathsf{Negate}(\varphi_e, a)))
\end{aligned}
$$

As was shown in the proof of **Exp-spec** for negation:

$$
\begin{aligned}
& \ddots \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \mathsf{Negate}(\varphi_e, a), \mathsf{Negate}(\varphi_e, a))) \\
& \qquad = \ \ddots \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta, \mathsf{Negate}(\varphi_e, a)))
\end{aligned}
$$

B.13 Proof of Lemma 12

Lemma 12 (boost-lemma) *Let* $\mathsf{boost} : \mathsf{Static}(\tau) \rightarrow \mathsf{Static}(\mathsf{Dynam}(\tau))$ *be defined as:*

$$\mathsf{boost}(x : \mathsf{Static}(\tau)) = x \ \star_S \ \lambda v : \tau.\ \mathbf{unit}_S(\mathbf{unit}_D(v))$$

Then, for any constant integer expression e (i.e., an integer constant or the negation of a constant integer expression):

$$\mathsf{boost}\,[\![e]\!] = \mathcal{S}[\![e]\!]$$

$\boxed{\text{Proof of Lemma 12}}$

By induction on terms.

$$\boxed{\text{Case } e \text{ is } n\text{:}}$$

$$\begin{array}{rcll}
\mathsf{boost}[\![n]\!] & = & [\![n]\!] \ \star_S \ \lambda v.\mathbf{unit}_S(\mathbf{unit}_D(v)) & (\mathsf{boost}\ \text{defn}) \\
& = & \mathbf{unit}_S(n) \ \star_S \ \lambda v.\mathbf{unit}_S(\mathbf{unit}_D(v)) & ([\![-]\!]\ \text{defn}) \\
& = & \mathbf{unit}_S(\mathbf{unit}_D(n)) & (\text{left unit}) \\
& = & \mathcal{S}[\![n]\!] &
\end{array}$$

$$\boxed{\text{Case } e \text{ is } -e\text{:}}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{boost}[\![-e]\!] =} \\
= \ [[\![e]\!] \ \star_S \ \lambda i.\mathbf{unit}_S(-i)] \ \star_S \ \lambda v.\mathbf{unit}_S(\mathbf{unit}_D(v)) & \\
= \ [\![e]\!] \ \star_S \ \lambda i.\mathbf{unit}_S(\mathbf{unit}_D(-i)) & (\text{left unit}) \\
= \ [\![e]\!] \ \star_S \ \lambda i.\mathbf{unit}_S(\mathbf{unit}_D(i) \ \star_D \ \lambda v.\mathbf{unit}_D(-v)) & (\text{left unit}) \\
= \ [\![e]\!] \ \star_S \ \lambda i.[\mathbf{unit}_S(\mathbf{unit}_D(i)) \ \star_S \ \lambda \delta_e.\mathbf{unit}_S(\delta_e \ \star_D \ \lambda v.\mathbf{unit}_D(-v))] & (\text{left unit}) \\
= \ [[\![e]\!] \ \star_S \ \lambda i.\mathbf{unit}_S(\mathbf{unit}_D(i))] \ \star_S \ \lambda \delta_e.\mathbf{unit}_S(\delta_e \ \star_D \ \lambda v.\mathbf{unit}_D(-v)) & (\text{assoc}) \\
= \ \mathsf{boost}\,[\![e]\!] \ \star_S \ \lambda \delta_e.\mathbf{unit}_S(\delta_e \ \star_D \ \lambda v.\mathbf{unit}_D(-v)) & \\
= \ \mathcal{S}[\![e]\!] \ \star_S \ \lambda \delta_e.\mathbf{unit}_S(\delta_e \ \star_D \ \lambda v.\mathbf{unit}_D(-v)) & (\text{ind hyp}) \\
= \ \mathcal{S}[\![-e]\!] &
\end{array}$$

B.14 Proof of Theorem 6

Theorem 6 *The correctness of the block structure RCBB:*

$$\begin{aligned}
&\mathcal{C}[\![\textbf{new } x \textbf{ in } c]\!] \star_S \lambda\varphi.\textbf{unit}_S(\varphi \star_D \lambda\_.\texttt{init}_{CS}) \\
&\qquad = \texttt{rdAddr} \star_S \lambda a. \\
&\qquad\quad\ \texttt{getLabel} \star_S \lambda L. \\
&\qquad\quad\ \mathcal{S}[\![\textbf{new } x \textbf{ in } c]\!] \star_S \lambda\delta. \\
&\qquad\quad\ \mathcal{C}[\![\textbf{new } x \textbf{ in } c]\!] \star_S \lambda\varphi. \\
&\qquad\quad\quad \textbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a)\ \mathsf{AND}\ \mathsf{FreshLabel}(L), \delta, \varphi) \star_D \lambda\_.\texttt{init}_{CS})
\end{aligned}$$

Proof of Theorem 6

By the innocence of $\texttt{rdAddr}$, $\texttt{getLabel}$, and $\mathcal{S}[\![\textbf{new } x \textbf{ in } c]\!]$:

$$\begin{aligned}
\mathcal{C}[\![\textbf{new } x \textbf{ in } c]\!] \star_S \lambda\varphi.\textbf{unit}_S(\varphi \star_D \lambda\_.\texttt{init}_{CS}) = {} & \texttt{rdAddr} \star_S \lambda a. \\
& \texttt{getLabel} \star_S \lambda L. \\
& \mathcal{S}[\![\textbf{new } x \textbf{ in } c]\!] \star_S \lambda\delta. \\
& \mathcal{C}[\![\textbf{new } x \textbf{ in } c]\!] \star_S \lambda\varphi. \\
& \quad \textbf{unit}_S(\varphi \star_D \lambda\_.\texttt{init}_{CS})
\end{aligned}$$

$$\begin{aligned}
= {} & \texttt{rdAddr} \star_S \lambda a. \\
& \texttt{getLabel} \star_S \lambda L. \\
& \texttt{rdEnv} \star_S \lambda\rho. \\
& \texttt{inAddr}\ (a+1) \\
& \quad \texttt{inEnv}\ \rho[x \mapsto a] \\
& \qquad \mathcal{S}[\![c]\!] \star_S \lambda\delta_c. \\
& \qquad \mathcal{C}[\![c]\!] \star_S \lambda\varphi_c. \\
& \qquad\quad \textbf{unit}_S(\texttt{Alloc}(a) \star_D \lambda\_.\varphi_c \star_D \lambda\_.\texttt{deAlloc}(a) \star_D \lambda\_.\texttt{init}_{CS})
\end{aligned}$$

By Observation introduction (where $\varphi = \texttt{Alloc}(a) \star_D \lambda\_.\varphi_c \star_D \lambda\_.\texttt{deAlloc}(a)$):

$$
\begin{array}{l}
= \ \texttt{rdAddr} \star_S \ \lambda a. \\
\quad \texttt{getLabel} \star_S \ \lambda L. \\
\quad \texttt{rdEnv} \star_S \ \lambda \rho. \\
\quad \texttt{inAddr}\ (a+1) \\
\qquad \texttt{inEnv}\ \rho[x \mapsto a] \\
\qquad\quad \mathcal{S}[\![c]\!] \star_S \ \lambda \delta_c. \\
\qquad\quad \mathcal{C}[\![c]\!] \star_S \ \lambda \varphi_c. \\
\qquad\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \varphi, \varphi) \star_D \ \lambda\_.\texttt{init}_{CS})
\end{array}
$$

Distributing $\texttt{init}_{CS}$ over the branches of the observation:

$$
\begin{array}{l}
= \ \texttt{rdAddr} \star_S \ \lambda a. \\
\quad \texttt{getLabel} \star_S \ \lambda L. \\
\quad \texttt{rdEnv} \star_S \ \lambda \rho. \\
\quad \texttt{inAddr}\ (a+1) \\
\qquad \texttt{inEnv}\ \rho[x \mapsto a] \\
\qquad\quad \mathcal{S}[\![c]\!] \star_S \ \lambda \delta_c. \\
\qquad\quad \mathcal{C}[\![c]\!] \star_S \ \lambda \varphi_c. \\
\qquad\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \varphi \star_D \ \lambda\_.\texttt{init}_{CS}, \varphi \star_D \ \lambda\_.\texttt{init}_{CS}))
\end{array}
$$

Filling in definition of φ:

$$
\begin{array}{l}
= \ \texttt{rdAddr} \star_S \ \lambda a. \\
\quad \texttt{getLabel} \star_S \ \lambda L. \\
\quad \texttt{rdEnv} \star_S \ \lambda \rho. \\
\quad \texttt{inAddr}\ (a+1) \\
\qquad \texttt{inEnv}\ \rho[x \mapsto \mathbf{unit}_S(a)] \\
\qquad\quad \mathcal{S}[\![c]\!] \star_S \ \lambda \delta_c. \\
\qquad\quad \mathcal{C}[\![c]\!] \star_S \ \lambda \varphi_c. \\
\qquad\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \texttt{Alloc}(a) \star_D \ \lambda\_.\varphi_c \star_D \ \lambda\_.\texttt{deAlloc}(a) \star_D \ \lambda\_.\texttt{init}_{CS}, \ldots))
\end{array}
$$

By Axiom 4.4, $\texttt{init}_{CS}$ commutes with $\texttt{deAlloc}(a)$:

$$
\begin{array}{l}
= \ \mathtt{rdAddr} \star_S \ \lambda a. \\
\quad \mathtt{getLabel} \star_S \ \lambda L. \\
\quad \mathtt{rdEnv} \star_S \ \lambda \rho. \\
\quad \mathtt{inAddr}\ (a+1) \\
\qquad \mathtt{inEnv}\ \rho[x \mapsto \mathbf{unit}_S(a)] \\
\qquad\quad \mathcal{S}[\![c]\!] \ \star_S \ \lambda \delta_c. \\
\qquad\quad \mathcal{C}[\![c]\!] \ \star_S \ \lambda \varphi_c. \\
\qquad\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a,L), \mathtt{Alloc}(a) \ \star_D \ \lambda\_.(\varphi_c \ \star_D \ \lambda\_.\mathtt{init}_{CS}) \ \star_D \ \lambda\_.\mathtt{deAlloc}(a), \ldots))
\end{array}
$$

Innocence of `rdAddr` and `getLabel`:

$$
\begin{array}{l}
= \ \mathtt{rdAddr} \star_S \ \lambda a. \\
\quad \mathtt{getLabel} \star_S \ \lambda L. \\
\quad \mathtt{rdEnv} \star_S \ \lambda \rho. \\
\quad \mathtt{inAddr}\ (a+1) \\
\qquad \mathtt{inEnv}\ \rho[x \mapsto \mathbf{unit}_S(a)] \\
\qquad\quad \mathtt{rdAddr} \star_S \ \lambda a'. \\
\qquad\quad \mathtt{getLabel} \star_S \ \lambda L'. \\
\qquad\quad \mathcal{S}[\![c]\!] \ \star_S \ \lambda \delta_c. \\
\qquad\quad \mathcal{C}[\![c]\!] \ \star_S \ \lambda \varphi_c. \\
\qquad\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a,L), \mathtt{Alloc}(a) \ \star_D \ \lambda\_.(\varphi_c \ \star_D \ \lambda\_.\mathtt{init}_{CS}) \ \star_D \ \lambda\_.\mathtt{deAlloc}(a), \ldots))
\end{array}
$$

Inductive hypothesis for c:

$$
\begin{array}{l}
= \ \texttt{rdAddr} \star_S \ \lambda a. \\
\quad \texttt{getLabel} \star_S \ \lambda L. \\
\quad \texttt{rdEnv} \star_S \ \lambda \rho. \\
\quad \texttt{inAddr}\ (a+1) \\
\qquad \texttt{inEnv}\ \rho[x \mapsto \mathbf{unit}_S(a)] \\
\qquad\quad \texttt{rdAddr} \star_S \ \lambda a'. \\
\qquad\quad \texttt{getLabel} \star_S \ \lambda L'. \\
\qquad\quad \mathcal{S}[\![c]\!] \star_S \ \lambda \delta_c. \\
\qquad\quad \mathcal{C}[\![c]\!] \star_S \ \lambda \varphi_c. \\
\qquad\quad \mathbf{unit}_S(\mathsf{Obs} \left(
\begin{array}{l}
\mathsf{Pre}(a, L), \\
\qquad \texttt{Alloc}(a) \star_D \ \lambda\_. \\
\qquad \left[\begin{array}{l}
\mathsf{Obs}(\mathsf{Pre}(a', L'), \\
\qquad \delta_c \star_D \ \lambda\_.\texttt{init}_{CS}, \\
\qquad\qquad \varphi_c \star_D \ \lambda\_.\texttt{init}_{CS})
\end{array} \right] \star_D \ \lambda\_. \ , \\
\qquad \texttt{deAlloc}(a) \\
\qquad\quad \ldots
\end{array}
\right))
\end{array}
$$

By construction $a' = a + 1$ and $L' = L$:

$$
\begin{array}{l}
= \ \ddots \\
\qquad\quad \mathbf{unit}_S(\mathsf{Obs} \left(
\begin{array}{l}
\mathsf{Pre}(a, L), \\
\qquad \texttt{Alloc}(a) \star_D \ \lambda\_. \\
\qquad \left[\begin{array}{l}
\mathsf{Obs}(\mathsf{Pre}(a+1, L), \\
\qquad \delta_c \star_D \ \lambda\_.\texttt{init}_{CS}, \\
\qquad\qquad \varphi_c \star_D \ \lambda\_.\texttt{init}_{CS})
\end{array} \right] \star_D \ \lambda\_. \ , \\
\qquad \texttt{deAlloc}(a) \\
\qquad\quad \ldots
\end{array}
\right))
\end{array}
$$

By Lemma 4, we can discharge the inner observation:

$$
\begin{array}{ll}
= & \texttt{rdAddr} \star_S \lambda a. \\
& \texttt{getLabel} \star_S \lambda L. \\
& \texttt{rdEnv} \star_S \lambda \rho. \\
& \texttt{inAddr}\ (a+1) \\
& \quad \texttt{inEnv}\ \rho[x \mapsto \mathbf{unit}_S(a)] \\
& \qquad \texttt{rdAddr} \star_S \lambda a'. \\
& \qquad \texttt{getLabel} \star_S \lambda L'. \\
& \qquad \mathcal{S}[\![c]\!] \star_S \lambda \delta_c. \\
& \qquad \mathcal{C}[\![c]\!] \star_S \lambda \varphi_c. \\
& \qquad\qquad \mathbf{unit}_S(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \texttt{Alloc}(a) \star_D \lambda\_.\delta_c \star_D \lambda\_.\texttt{init}_{CS} \star_D \lambda\_.\texttt{deAlloc}(a), \\ \qquad\qquad \texttt{Alloc}(a) \star_D \lambda\_.\varphi_c \star_D \lambda\_.\texttt{init}_{CS} \star_D \lambda\_.\texttt{deAlloc}(a) \end{array}\right)) \\
= & \texttt{rdAddr} \star_S \lambda a. \\
& \texttt{getLabel} \star_S \lambda L. \\
& \texttt{rdEnv} \star_S \lambda \rho. \\
& \texttt{inAddr}\ (a+1) \\
& \quad \texttt{inEnv}\ \rho[x \mapsto \mathbf{unit}_S(a)] \\
& \qquad \texttt{rdAddr} \star_S \lambda a'. \\
& \qquad \texttt{getLabel} \star_S \lambda L'. \\
& \qquad \mathcal{S}[\![c]\!] \star_S \lambda \delta_c. \\
& \qquad \mathcal{C}[\![c]\!] \star_S \lambda \varphi_c. \\
& \qquad\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a,L), \texttt{Alloc}(a) \star_D \lambda\_.\delta_c \star_D \lambda\_.\texttt{deAlloc}(a), \ldots) \star_D \lambda\_.\texttt{init}_{CS}) \\
= & \texttt{rdAddr} \star_S \lambda a. \\
& \texttt{getLabel} \star_S \lambda L. \\
& \mathcal{S}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!] \star_S \lambda \delta. \\
& \mathcal{C}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!] \star_S \lambda \varphi. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a,L), \delta, \varphi) \star_D \lambda\_.\texttt{init}_{CS})
\end{array}
$$

□Theorem 6

B.15 Proof of Theorem 7

Theorem 7 *The correctness of the Boolean RCBB:*

$$
\begin{aligned}
\mathcal{C}[\![b : \mathit{Bool}]\!] = \;& \texttt{rdAddr} \star_S \lambda a. \\
& \texttt{getLabel} \star_S \lambda L. \\
& \mathcal{S}[\![b]\!] \star_S \lambda \delta_b. \\
& \mathcal{C}[\![b]\!] \star_S \lambda \varphi_b. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a) \;\mathsf{AND}\; \mathsf{FreshLabel}(L)), \delta_b, \varphi_b)
\end{aligned}
$$

Proof of Theorem 7.

Case $b = e_1$ leq $e2$.

$$
\begin{aligned}
\mathcal{C}[\![e_1 \;\mathsf{leq}\; e2]\!] = \;& \texttt{rdAddr} \star_S \lambda a. \\
& \texttt{getLabel} \star_S \lambda L. \\
& \texttt{inAddr}\,(a+2) \\
& \quad \mathcal{S}[\![e_1]\!] \star_S \lambda \delta_1.\mathcal{S}[\![e_2]\!] \star_S \lambda \delta_2.\mathcal{C}[\![e_1]\!] \star_S \lambda \varphi_1.\mathcal{C}[\![e_2]\!] \star_S \lambda \varphi_2. \\
& \quad\; \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \mathsf{Lteq}(\varphi_1, \varphi_2, a), \mathsf{Lteq}(\varphi_1, \varphi_2, a)))
\end{aligned}
$$

By an argument analogous to the correctness of $\mathcal{C}[\![-e]\!]$ in the proof of **Exp-spec**, it is clear that, within the above context (and assuming **Exp-spec**):

$$
\mathsf{Lteq}(\varphi_1, \varphi_2, a) =
$$

$$
\ddots
\left(
\begin{array}{l}
\varphi_1 \star_D \lambda i. \\
\varphi_2 \star_D \lambda j. \\
\texttt{Alloc}(a) \star_D \lambda\_. \\
\qquad\quad \vdots \\
\texttt{deAlloc}(a+1) \star_D \lambda\_. \\
\quad \mathbf{unit}_D(\lambda\langle \kappa_T, \kappa_F\rangle.(v_1 \leq v_2 \to \kappa_T, \kappa_F))
\end{array}
\right)
=
\begin{array}{l}
\ddots \\
\left(
\begin{array}{l}
\delta_1 \star_D \lambda i. \\
\delta_2 \star_D \lambda j. \\
\quad \mathbf{unit}_D(\lambda\langle \kappa_T, \kappa_F\rangle.(i \leq j \to \kappa_T, \kappa_F))
\end{array}
\right)
\end{array}
$$

But this is simply the right-hand side of **Bool-spec**. □ Theorem 7.

B.16 Proof of Theorem 9

Theorem 9 *For terms cf of the control-flow block:*

$$\begin{aligned}
&\mathcal{C}\llbracket cf \rrbracket \star_S \lambda\varphi.\mathbf{unit}_S(\varphi \star_D \lambda\_.\mathtt{init}_{CS})\\
&\qquad = \mathtt{rdAddr} \star_S \lambda a.\\
&\qquad\quad \mathtt{getLabel} \star_S \lambda L.\\
&\qquad\quad \mathcal{S}\llbracket cf \rrbracket \star_S \lambda\delta.\\
&\qquad\quad \mathcal{C}\llbracket cf \rrbracket \star_S \lambda\varphi.\\
&\qquad\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{FreshLoc}(a)\ \mathsf{AND}\ \mathsf{FreshLabel}(L), \delta, \varphi) \star_D \lambda\_.\mathtt{init}_{CS})
\end{aligned}$$

Proof of Theorem 9

Let $\mathsf{Pre}(a, L) = \mathsf{FreshLoc}(a)\ \mathsf{AND}\ \mathsf{FreshLabel}(L)$.

$\mathcal{C}\llbracket \textbf{if } b \textbf{ then } c \rrbracket \star_S \lambda\varphi.\mathbf{unit}_S(\varphi \star_D \lambda\_.\mathtt{init}_{CS})$

Innocence of $\mathtt{rdAddr}$, $\mathtt{getLabel}$, and $\mathcal{S}\llbracket \textbf{if } b \textbf{ then } c \rrbracket$:

$$\begin{aligned}
= \ &\mathtt{rdAddr} \star_S \lambda a.\\
&\mathtt{getLabel} \star_S \lambda L.\\
&\mathcal{S}\llbracket \textbf{if } b \textbf{ then } c \rrbracket \star_S \lambda\delta.\\
&\mathcal{C}\llbracket \textbf{if } b \textbf{ then } c \rrbracket \star_S \lambda\varphi.\\
&\quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \varphi, \varphi) \star_D \lambda\_.\mathtt{init}_{CS})
\end{aligned}$$

Unfolding definitions of $\mathcal{S}\llbracket \textbf{if } b \textbf{ then } c \rrbracket$ and $\mathcal{C}\llbracket \textbf{if } b \textbf{ then } c \rrbracket$:

$$
\begin{array}{ll}
= & \mathtt{rdAddr} \star_S \lambda a. \\
 & \mathtt{getLabel} \star_S \lambda L. \\
 & \mathcal{S}[\![b]\!] \star_S \lambda \delta_b. \\
 & \mathcal{S}[\![c]\!] \star_S \lambda \delta_c. \\
 & \mathtt{newlabel} \star_S \lambda L_\kappa. \\
 & \mathtt{newlabel} \star_S \lambda L_c. \\
 & \mathcal{C}[\![b]\!] \star_S \lambda \varphi_b. \\
 & \mathcal{C}[\![c]\!] \star_S \lambda \varphi_c. \\
 & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \mathsf{IfThenPS}(\varphi_b, \varphi_c, L_\kappa, L_c), \ldots) \star_D \lambda\_.\mathtt{init}_{CS}) \\
= & \mathtt{rdAddr} \star_S \lambda a. \\
 & \mathtt{getLabel} \star_S \lambda L. \\
 & \mathcal{S}[\![b]\!] \star_S \lambda \delta_b. \\
 & \mathcal{S}[\![c]\!] \star_S \lambda \delta_c. \\
 & \mathtt{newlabel} \star_S \lambda L_\kappa. \\
 & \mathtt{newlabel} \star_S \lambda L_c. \\
 & \qquad \mathtt{rdAddr} \star_S \lambda a'. \\
 & \qquad \mathtt{getLabel} \star_S \lambda L'. \\
 & \qquad \mathcal{C}[\![b]\!] \star_S \lambda \varphi_b. \\
 & \qquad \mathcal{C}[\![c]\!] \star_S \lambda \varphi_c. \\
 & \qquad \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \mathsf{IfThenPS}(\varphi_b, \varphi_c, L_\kappa, L_c), \ldots) \star_D \lambda\_.\mathtt{init}_{CS})
\end{array}
$$

$$\boxed{\text{By construction, } L_\kappa = L,\ L_c = L + 1,\ L' = L + 2, \text{ and } a = a'\text{:}}$$

$$
\begin{aligned}
= {} & \texttt{rdAddr} \star_S \ \lambda a.\\
& \texttt{getLabel} \star_S \ \lambda L.\\
& \mathcal{S}[\![b]\!] \star_S \ \lambda \delta_b.\\
& \mathcal{S}[\![c]\!] \star_S \ \lambda \delta_c.\\
& \texttt{newlabel} \star_S \ \lambda L_\kappa.\\
& \texttt{newlabel} \star_S \ \lambda L_c.\\
& \qquad \texttt{rdAddr} \star_S \ \lambda a'.\\
& \qquad \texttt{getLabel} \star_S \ \lambda L'.\\
& \qquad \mathcal{C}[\![b]\!] \star_S \ \lambda \varphi_b.\\
& \qquad \mathcal{C}[\![c]\!] \star_S \ \lambda \varphi_c.\\
& \qquad\quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \mathsf{IfThenPS}(\varphi_b, \varphi_c, L, L+1), \ldots) \ \star_D \ \lambda\_.\texttt{init}_{CS})
\end{aligned}
$$

Unfolding definition of $\mathsf{IfThenPS}$:

$$
= \ \ddots(\mathsf{Obs} \left(\begin{array}{l}
\mathsf{Pre}(a, L), \\
\qquad \left[\begin{array}{l}
\varphi_b \ \star_D \ \lambda\beta. \\
\texttt{callcc}\ \lambda\kappa. \\
\texttt{updateCode}[L \mapsto \kappa\bullet] \ \star_D \ \lambda\_. \\
\texttt{updateCode}[L+1 \mapsto \varphi_c \ \star_D \ \lambda\_.(\mathsf{jump}\, L)] \ \star_D \ \lambda\_. \\
\quad \beta\langle \mathsf{jump}\,(L+1), \mathsf{jump}\, L\rangle
\end{array} \right], \\
\qquad\quad \ldots
\end{array} \right) \star_D \ \lambda\_.\texttt{init}_{CS}
$$

Inductive hypothesis for b:

$$
= \ \ddots(\mathsf{Obs} \left(\begin{array}{l}
\mathsf{Pre}(a, L), \\
\qquad \left[\begin{array}{l}
\delta_b \ \star_D \ \lambda\beta. \\
\texttt{callcc}\ \lambda\kappa. \\
\texttt{updateCode}[L \mapsto \kappa\bullet] \ \star_D \ \lambda\_. \\
\texttt{updateCode}[L+1 \mapsto \varphi_c \ \star_D \ \lambda\_.(\mathsf{jump}\, L)] \ \star_D \ \lambda\_. \\
\quad \beta\langle \mathsf{jump}\,(L+1), \mathsf{jump}\, L\rangle
\end{array} \right], \\
\qquad\quad \ldots
\end{array} \right) \star_D \ \lambda\_.\texttt{init}_{CS}
$$

Case 1: $\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle = \texttt{jump}\,(L+1)$:

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \left[\begin{array}{l} \delta_b \;\star_D\; \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \texttt{updateCode}[L \mapsto \kappa\bullet] \;\star_D\; \lambda\_. \\ \texttt{updateCode}[L+1 \mapsto \varphi_c \;\star_D\; \lambda\_.(\texttt{jump}\,L)] \;\star_D\; \lambda\_. \\ \quad \texttt{jump}\,(L+1) \end{array}\right] \quad , \\ \qquad\quad \ldots \end{array}\right)) \;\star_D\; \lambda\_.\texttt{init}_{CS}$$

“Unrolling” $\texttt{jump}\,(L+1)$:

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \left[\begin{array}{l} \delta_b \;\star_D\; \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \texttt{updateCode}[L \mapsto \kappa\bullet] \;\star_D\; \lambda\_. \\ \texttt{updateCode}[L+1 \mapsto \varphi_c \;\star_D\; \lambda\_.(\texttt{jump}\,L)] \;\star_D\; \lambda\_. \\ \quad \varphi_c \;\star_D\; \lambda\_.\kappa\bullet \end{array}\right] \quad , \\ \qquad\quad \ldots \end{array}\right)) \;\star_D\; \lambda\_.\texttt{init}_{CS}$$

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \left[\begin{array}{l} \delta_b \;\star_D\; \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \texttt{updateCode}[L \mapsto \kappa\bullet] \;\star_D\; \lambda\_. \\ \texttt{updateCode}[L+1 \mapsto \varphi_c \;\star_D\; \lambda\_.(\texttt{jump}\,L)] \;\star_D\; \lambda\_. \\ \quad \varphi_c \;\star_D\; \lambda\_.\kappa\bullet \end{array}\right] \;\star_D\; \lambda\_.\texttt{init}_{CS}, \\ \qquad\quad \ldots \end{array}\right))$$

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \left[\begin{array}{l} \delta_b \star_D \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \texttt{updateCode}[L \mapsto \texttt{init}_{CS} \star_D \lambda\_.\kappa\bullet] \star_D \lambda\_. \\ \texttt{updateCode}[L+1 \mapsto \varphi_c \star_D \lambda\_.(\texttt{jump}\,L)] \star_D \lambda\_. \\ \quad \varphi_c \star_D \lambda\_.\texttt{init}_{CS} \star_D \lambda\_.\kappa\bullet \end{array}\right], \\ \qquad\quad \ldots \end{array}\right))$$

Observation introduction:

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \left[\begin{array}{l} \delta_b \star_D \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \texttt{updateCode}[L \mapsto \texttt{init}_{CS} \star_D \lambda\_.\kappa\bullet] \star_D \lambda\_. \\ \texttt{updateCode}[L+1 \mapsto \varphi_c \star_D \lambda\_.(\texttt{jump}\,L)] \star_D \lambda\_. \\ \quad \mathsf{Obs}(\mathsf{Pre}(a,L+2), \varphi_c \star_D \lambda\_.\texttt{init}_{CS}, \varphi_c \star_D \lambda\_.\texttt{init}_{CS}) \star_D \lambda\_.\kappa\bullet \end{array}\right], \\ \qquad\quad \ldots \end{array}\right))$$

By inductive hypothesis for c:

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \left[\begin{array}{l} \delta_b \star_D \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \texttt{updateCode}[L \mapsto \texttt{init}_{CS} \star_D \lambda\_.\kappa\bullet] \star_D \lambda\_. \\ \texttt{updateCode}[L+1 \mapsto \varphi_c \star_D \lambda\_.(\texttt{jump}\,L)] \star_D \lambda\_. \\ \quad \mathsf{Obs}(\mathsf{Pre}(a,L+2), \delta_c \star_D \lambda\_.\texttt{init}_{CS}, \ldots) \star_D \lambda\_.\kappa\bullet \end{array}\right], \\ \qquad\quad \ldots \end{array}\right))$$

By Observation Introduction:

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l}
\mathsf{Pre}(a,L),\\
\qquad \left[\begin{array}{l}
\delta_b \star_D \lambda\beta.\\
\texttt{callcc}\ \lambda\kappa.\\
\mathsf{Obs}(\mathsf{Pre}(a,L),\\
\qquad \texttt{updateCode}[L \mapsto \texttt{init}_{CS} \star_D \lambda\_.\kappa\bullet] \star_D \lambda\_.\\
\qquad \texttt{updateCode}[L+1 \mapsto \varphi_c \star_D \lambda\_.(\texttt{jump}\,L)] \star_D \lambda\_.\\
\qquad\quad \mathsf{Obs}(\mathsf{Pre}(a,L+2), \delta_c \star_D \lambda\_.\texttt{init}_{CS}, \ldots) \star_D \lambda\_.\kappa\bullet,\\
\qquad\qquad \ldots)
\end{array}\right],\\
\qquad\quad \ldots
\end{array}\right))$$

By two applications of Lemma 6.2 and Observation Elimination:

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l}
\mathsf{Pre}(a,L),\\
\qquad \left[\begin{array}{l}
\delta_b \star_D \lambda\beta.\\
\texttt{callcc}\ \lambda\kappa.\\
\mathsf{Obs}(\mathsf{Pre}(a,L),\\
\qquad \texttt{updateCode}[L \mapsto \texttt{init}_{CS} \star_D \lambda\_.\kappa\bullet] \star_D \lambda\_.\\
\qquad \texttt{updateCode}[L+1 \mapsto \varphi_c \star_D \lambda\_.(\texttt{jump}\,L)] \star_D \lambda\_.\\
\qquad\quad \delta_c \star_D \lambda\_.\texttt{init}_{CS} \star_D \lambda\_.\kappa\bullet,\\
\qquad\qquad \ldots)
\end{array}\right],\\
\qquad\quad \ldots
\end{array}\right))$$

From Axiom 3.1, Axiom 3.2, Axiom 4, and Axiom 5, and δ_c contains no `updateCode`/`getCode`:

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L),\\ \qquad\left[\begin{array}{l} \delta_b \ \star_D\ \lambda\beta.\\ \texttt{callcc}\ \lambda\kappa.\\ \mathsf{Obs}(\mathsf{Pre}(a,L),\\ \qquad \texttt{updateCode}[L\mapsto \texttt{init}_{CS}\ \star_D\ \lambda\_.\kappa\bullet]\ \star_D\ \lambda\_.\\ \qquad \texttt{updateCode}[L+1\mapsto \varphi_c\ \star_D\ \lambda\_.(\texttt{jump}\,L)]\ \star_D\ \lambda\_.\\ \qquad\quad \texttt{init}_{CS}\ \star_D\ \lambda\_.\delta_c\ \star_D\ \lambda\_.\kappa\bullet,\\ \qquad\qquad \ldots)\end{array}\right],\\ \qquad\quad \ldots \end{array}\right))$$

$\boxed{\texttt{updateCode}\Delta\ \star_D\ \lambda\_.\texttt{init}_{CS} = \texttt{init}_{CS}\text{:}}$

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L),\\ \qquad\left[\begin{array}{l} \delta_b \ \star_D\ \lambda\beta.\\ \texttt{callcc}\ \lambda\kappa.\\ \mathsf{Obs}(\mathsf{Pre}(a,L),\\ \qquad \texttt{init}_{CS}\ \star_D\ \lambda\_.\delta_c\ \star_D\ \lambda\_.\kappa\bullet,\\ \qquad\qquad \ldots)\end{array}\right],\\ \qquad\quad \ldots \end{array}\right))$$

$\boxed{\text{From Lemma 7:}}$

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L),\\ \qquad\left[\begin{array}{l} \delta_b \ \star_D\ \lambda\beta.\\ \mathsf{Obs}(\mathsf{Pre}(a,L),\\ \quad \texttt{callcc}\ \lambda\kappa.\\ \qquad \texttt{init}_{CS}\ \star_D\ \lambda\_.\delta_c\ \star_D\ \lambda\_.\kappa\bullet,\\ \qquad\qquad \ldots)\end{array}\right],\\ \qquad\quad \ldots \end{array}\right))$$

$\boxed{\text{Because } \delta_b = \mathbf{unit}_D(\ldots)\text{, it is innocent:}}$

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \left[\begin{array}{l} \mathsf{Obs}(\mathsf{Pre}(a,L), \\ \quad \delta_b \ \star_D \ \lambda\beta. \\ \quad \mathtt{callcc}\ \lambda\kappa. \\ \qquad \mathtt{init}_{CS} \ \star_D \ \lambda\_.\delta_c \ \star_D \ \lambda\_.\kappa\bullet, \\ \qquad\qquad \ldots) \end{array}\right], \\ \qquad\quad \ldots \end{array}\right))$$

$\boxed{\text{By Observation Elimination:}}$

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \delta_b \ \star_D \ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.\mathtt{init}_{CS} \ \star_D \ \lambda\_.\delta_c \ \star_D \ \lambda\_.\kappa\bullet, \\ \qquad\qquad \mathsf{IfThenPS}(\varphi_b,\varphi_c,L_\kappa,L_c) \ \star_D \ \lambda\_.\mathtt{init}_{CS} \end{array}\right))$$

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \delta_b \ \star_D \ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.\delta_c \ \star_D \ \lambda\_.\mathtt{init}_{CS} \ \star_D \ \lambda\_.\kappa\bullet, \\ \qquad\qquad \mathsf{IfThenPS}(\varphi_b,\varphi_c,L_\kappa,L_c) \ \star_D \ \lambda\_.\mathtt{init}_{CS} \end{array}\right))$$

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad (\delta_b \ \star_D \ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.\delta_c \ \star_D \ \lambda\_.\kappa\bullet) \ \star_D \ \lambda\_.\mathtt{init}_{CS}, \\ \qquad\qquad \mathsf{IfThenPS}(\varphi_b,\varphi_c,L_\kappa,L_c) \ \star_D \ \lambda\_.\mathtt{init}_{CS} \end{array}\right))$$

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad (\delta_b \ \star_D \ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.\delta_c \ \star_D \ \lambda\_.\kappa\bullet), \\ \qquad\qquad \mathsf{IfThenPS}(\varphi_b,\varphi_c,L_\kappa,L_c) \end{array}\right)) \ \star_D \ \lambda\_.\mathtt{init}_{CS}$$

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad (\delta_b \ \star_D \ \lambda\beta.\mathtt{callcc}\ \lambda\kappa.\beta\langle\delta_c \ \star_D \ \lambda\_.\kappa\bullet, \kappa\bullet\rangle), \\ \qquad\qquad \mathsf{IfThenPS}(\varphi_b,\varphi_c,L_\kappa,L_c) \end{array}\right)) \ \star_D \ \lambda\_.\mathtt{init}_{CS}$$

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \mathsf{IfThen}(\delta_b,\delta_c), \\ \qquad\qquad \mathsf{IfThenPS}(\varphi_b,\varphi_c,L_\kappa,L_c) \end{array}\right)) \ \star_D \ \lambda\_.\mathtt{init}_{CS}$$

$$\begin{array}{ll}
= & \mathtt{rdAddr} \star_S \lambda a. \\
 & \mathtt{getLabel} \star_S \lambda L. \\
 & \mathcal{S}[\![b]\!] \star_S \lambda \delta_b. \\
 & \mathcal{S}[\![c]\!] \star_S \lambda \delta_c. \\
 & \mathtt{newlabel} \star_S \lambda L_\kappa. \\
 & \mathtt{newlabel} \star_S \lambda L_c. \\
 & \qquad \mathtt{rdAddr} \star_S \lambda a'. \\
 & \qquad \mathtt{getLabel} \star_S \lambda L'. \\
 & \qquad \mathcal{C}[\![b]\!] \star_S \lambda \varphi_b. \\
 & \qquad \mathcal{C}[\![c]\!] \star_S \lambda \varphi_c. \\
 & \qquad \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \mathsf{IfThen}(\delta_b, \delta_c), \mathsf{IfThenPS}(\varphi_b, \varphi_c, L_\kappa, L_c)) \star_D \lambda\_.\mathtt{init}_{CS})
\end{array}$$

$$\begin{array}{ll}
= & \mathtt{rdAddr} \star_S \lambda a. \\
 & \mathtt{getLabel} \star_S \lambda L. \\
 & \mathcal{S}[\![\mathbf{if}\ b\ \mathbf{then}\ c]\!] \star_S \lambda \delta. \\
 & \mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c]\!] \star_S \lambda \varphi. \\
 & \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \delta, \varphi) \star_D \lambda\_.\mathtt{init}_{CS})
\end{array}$$

Case 2: $\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle = \texttt{jump}\,L$:

$$= \ddots(\mathsf{Obs} \left(\begin{array}{l} \mathsf{Pre}(a, L), \\ \qquad \left[\begin{array}{l} \delta_b \ \star_D \ \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \texttt{updateCode}[L \mapsto \kappa\bullet] \ \star_D \ \lambda\_. \\ \texttt{updateCode}[L+1 \mapsto \varphi_c \ \star_D \ \lambda\_.(\texttt{jump}\,L)] \ \star_D \ \lambda\_. \\ \quad \texttt{jump}\,L \end{array} \right], \\ \qquad\qquad \ldots \end{array} \right)) \ \star_D \ \lambda\_.\texttt{init}_{CS}$$

$\boxed{\text{“Unrolling” } \texttt{jump}\,L\text{:}}$

$$= \ddots(\mathsf{Obs} \left(\begin{array}{l} \mathsf{Pre}(a, L), \\ \qquad \left[\begin{array}{l} \delta_b \ \star_D \ \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \texttt{updateCode}[L \mapsto \kappa\bullet] \ \star_D \ \lambda\_. \\ \texttt{updateCode}[L+1 \mapsto \varphi_c \ \star_D \ \lambda\_.(\texttt{jump}\,L)] \ \star_D \ \lambda\_. \\ \quad \kappa\bullet \end{array} \right], \\ \qquad\qquad \ldots \end{array} \right)) \ \star_D \ \lambda\_.\texttt{init}_{CS}$$

$\boxed{\text{Lemma 1: } \mathsf{Obs}(\theta, x, y) \ \star \ f = \mathsf{Obs}(\theta, x \ \star \ f, y \ \star \ f)\text{:}}$

$$= \ddots(\mathsf{Obs} \left(\begin{array}{l} \mathsf{Pre}(a, L), \\ \qquad \left[\begin{array}{l} \delta_b \ \star_D \ \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \texttt{updateCode}[L \mapsto \kappa\bullet] \ \star_D \ \lambda\_. \\ \texttt{updateCode}[L+1 \mapsto \varphi_c \ \star_D \ \lambda\_.(\texttt{jump}\,L)] \ \star_D \ \lambda\_. \\ \quad \kappa\bullet \end{array} \right] \ \star_D \ \lambda\_.\texttt{init}_{CS}, \\ \qquad\qquad \ldots \end{array} \right))$$

$\boxed{\text{Axiom 3.2:}}$

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \left[\begin{array}{l} \delta_b \star_D \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \texttt{updateCode}[L \mapsto \texttt{init}_{CS} \star_D \lambda\_.\kappa\bullet] \star_D \lambda\_. \\ \texttt{updateCode}[L+1 \mapsto \varphi_c \star_D \lambda\_.(\texttt{jump}\,L)] \star_D \lambda\_. \\ \quad \texttt{init}_{CS} \star_D \lambda\_.\kappa\bullet \end{array}\right], \\ \qquad\quad \ldots \end{array}\right))$$

Since $\texttt{updateCode}\Delta \star_D \lambda\_.\texttt{init}_{CS} = \texttt{init}_{CS}$ (from Axiom 2.1:

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \left[\begin{array}{l} \delta_b \star_D \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \quad \texttt{init}_{CS} \star_D \lambda\_.\kappa\bullet \end{array}\right], \\ \qquad\quad \ldots \end{array}\right))$$

$$= \ddots(\mathsf{Obs}\left(\begin{array}{l} \mathsf{Pre}(a,L), \\ \qquad \left[\begin{array}{l} \delta_b \star_D \lambda\beta. \\ \texttt{callcc}\ \lambda\kappa. \\ \quad \kappa\bullet \end{array}\right] \star_D \lambda\_.\texttt{init}_{CS}, \\ \qquad\quad \ldots \end{array}\right))$$

Lemma 1: $\mathsf{Obs}(\theta, x \star f, y \star f) = \mathsf{Obs}(\theta, x, y) \star f$:

$$= \ddots\mathsf{Obs}(\mathsf{Pre}(a,L), \delta_b \star_D \lambda\beta.\texttt{callcc}\ \lambda\kappa.\,\beta\langle\delta_c \star_D \kappa, \kappa\bullet\rangle), \mathsf{IfThenPS}(\varphi_b, \varphi_c, L_\kappa, L_c)) \star_D \lambda\_.\texttt{init}_{CS}$$

Filling in the context and folding the definition of IfThen:

$$
\begin{aligned}
= {} & \texttt{rdAddr} \star_S \ \lambda a. \\
& \texttt{getLabel} \star_S \ \lambda L. \\
& \mathcal{S}[\![b]\!] \star_S \ \lambda \delta_b. \\
& \mathcal{S}[\![c]\!] \star_S \ \lambda \delta_c. \\
& \texttt{newlabel} \star_S \ \lambda L_\kappa. \\
& \texttt{newlabel} \star_S \ \lambda L_c. \\
& \qquad \texttt{rdAddr} \star_S \ \lambda a'. \\
& \qquad \texttt{getLabel} \star_S \ \lambda L'. \\
& \qquad \mathcal{C}[\![b]\!] \star_S \ \lambda \varphi_b. \\
& \qquad \mathcal{C}[\![c]\!] \star_S \ \lambda \varphi_c. \\
& \qquad\quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \mathsf{IfThen}(\delta_b, \delta_c), \mathsf{IfThenPS}(\varphi_b, \varphi_c, L_\kappa, L_c)) \ \star_D \ \lambda\_.\texttt{init}_{CS}) \\
= {} & \texttt{rdAddr} \star_S \ \lambda a. \\
& \texttt{getLabel} \star_S \ \lambda L. \\
& \mathcal{S}[\![\textbf{if } b \textbf{ then } c]\!] \star_S \ \lambda \delta. \\
& \mathcal{C}[\![\textbf{if } b \textbf{ then } c]\!] \star_S \ \lambda \varphi. \\
& \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \delta, \varphi) \ \star_D \ \lambda\_.\texttt{init}_{CS})
\end{aligned}
$$

$\boxed{\text{Case } t \text{ is } \mathbf{while}\ b\ \mathbf{do}\ c}$

$$
\begin{aligned}
&\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] \star_S \lambda\varphi.\mathbf{unit}_S(\varphi \star_D \lambda\_.\texttt{init}_{CS})\\
&\qquad = \texttt{rdAddr} \star_S \lambda a.\\
&\qquad\quad \texttt{getLabel} \star_S \lambda L.\\
&\qquad\quad \mathcal{S}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] \star_S \lambda\delta.\\
&\qquad\quad \mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] \star_S \lambda\varphi.\\
&\qquad\qquad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a,L),\delta,\varphi) \star_D \lambda\_.\texttt{init}_{CS})
\end{aligned}
$$

$\boxed{\text{Unfolding definitions of } \mathcal{S}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] \text{ and } \mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]\text{:}}$

$$
\begin{aligned}
= \;& \texttt{rdAddr} \star_S \lambda a.\\
& \texttt{getLabel} \star_S \lambda L.\\
& \mathcal{S}[\![b]\!] \star_S \lambda\delta_b.\\
& \mathcal{S}[\![c]\!] \star_S \lambda\delta_c.\\
& \texttt{newlabel} \star_S \lambda L_\kappa.\\
& \texttt{newlabel} \star_S \lambda L_c.\\
& \texttt{newlabel} \star_S \lambda L_{test}.\\
& \mathcal{C}[\![b]\!] \star_S \lambda\varphi_b.\\
& \mathcal{C}[\![c]\!] \star_S \lambda\varphi_c.\\
& \quad \mathbf{unit}_S((\mathsf{Obs}\left(\begin{array}{l}
\mathsf{Pre}(a,L),\\
\qquad \mathsf{whilePS}(\varphi_b,\varphi_c,L_\kappa,L_c,L_{test}),\\
\qquad\qquad \mathsf{whilePS}(\varphi_b,\varphi_c,L_\kappa,L_c,L_{test})
\end{array}\right)) \star_D \lambda\_.\texttt{init}_{CS})
\end{aligned}
$$

$$
\begin{array}{l}
= \; \texttt{rdAddr} \star_S \lambda a. \\
\quad \texttt{getLabel} \star_S \lambda L. \\
\quad \mathcal{S}[\![b]\!] \star_S \lambda \delta_b. \\
\quad \mathcal{S}[\![c]\!] \star_S \lambda \delta_c. \\
\quad \texttt{newlabel} \star_S \lambda L_\kappa. \\
\quad \texttt{newlabel} \star_S \lambda L_c. \\
\quad \texttt{newlabel} \star_S \lambda L_{test}. \\
\qquad \texttt{rdAddr} \star_S \lambda a'. \\
\qquad \texttt{getLabel} \star_S \lambda L'. \\
\qquad \mathcal{C}[\![b]\!] \star_S \lambda \varphi_b. \\
\qquad \mathcal{C}[\![c]\!] \star_S \lambda \varphi_c. \\
\qquad\;\; \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \mathsf{whilePS}(\varphi_b, \varphi_c, L_\kappa, L_c, L_{test}), \ldots) \star_D \lambda\_.\texttt{init}_{CS})
\end{array}
$$

$\boxed{\text{By construction, } L_\kappa = L,\ L_c = L + 1,\ L_{test} = L + 2,\ L' = L + 3, \text{ and } a = a' \text{ (Axiom 6.2):}}$

$$
\begin{array}{l}
= \; \texttt{rdAddr} \star_S \lambda a. \\
\quad \texttt{getLabel} \star_S \lambda L. \\
\quad \mathcal{S}[\![b]\!] \star_S \lambda \delta_b. \\
\quad \mathcal{S}[\![c]\!] \star_S \lambda \delta_c. \\
\quad \texttt{newlabel} \star_S \lambda L_\kappa. \\
\quad \texttt{newlabel} \star_S \lambda L_c. \\
\quad \texttt{newlabel} \star_S \lambda L_{test}. \\
\qquad \texttt{rdAddr} \star_S \lambda a'. \\
\qquad \texttt{getLabel} \star_S \lambda L'. \\
\qquad \mathcal{C}[\![b]\!] \star_S \lambda \varphi_b. \\
\qquad \mathcal{C}[\![c]\!] \star_S \lambda \varphi_c. \\
\qquad\;\; \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \mathsf{whilePS}(\varphi_b, \varphi_c, L, L + 1, L + 2), \ldots) \star_D \lambda\_.\texttt{init}_{CS})
\end{array}
$$

$\boxed{\text{Lemma 1: } \mathsf{Obs}(\theta, x, y) \star f = \mathsf{Obs}(\theta, x \star f, y \star f)\text{:}}$

$$
\begin{array}{rl}
= & \texttt{rdAddr} \star_S \lambda a. \\
 & \texttt{getLabel} \star_S \lambda L. \\
 & \mathcal{S}[\![b]\!] \star_S \lambda \delta_b. \\
 & \mathcal{S}[\![c]\!] \star_S \lambda \delta_c. \\
 & \texttt{newlabel} \star_S \lambda L_\kappa. \\
 & \texttt{newlabel} \star_S \lambda L_c. \\
 & \texttt{newlabel} \star_S \lambda L_{test}. \\
 & \qquad \texttt{rdAddr} \star_S \lambda a'. \\
 & \qquad \texttt{getLabel} \star_S \lambda L'. \\
 & \qquad \mathcal{C}[\![b]\!] \star_S \lambda \varphi_b. \\
 & \qquad \mathcal{C}[\![c]\!] \star_S \lambda \varphi_c. \\
 & \qquad \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \mathsf{whilePS}(\varphi_b, \varphi_c, L, L+1, L+2) \star_D \lambda\_.\texttt{init}_{CS}, \ldots))
\end{array}
$$

Rather than repeat this term over and over, we will define the following context $\mathcal{W}(-)$:

$$
\begin{array}{rl}
\mathcal{W}(-) = & \texttt{rdAddr} \star_S \lambda a. \\
 & \texttt{getLabel} \star_S \lambda L. \\
 & \mathcal{S}[\![b]\!] \star_S \lambda \delta_b. \\
 & \mathcal{S}[\![c]\!] \star_S \lambda \delta_c. \\
 & \texttt{newlabel} \star_S \lambda L_\kappa. \\
 & \texttt{newlabel} \star_S \lambda L_c. \\
 & \texttt{newlabel} \star_S \lambda L_{test}. \\
 & \qquad \texttt{rdAddr} \star_S \lambda a'. \\
 & \qquad \texttt{getLabel} \star_S \lambda L'. \\
 & \qquad \mathcal{C}[\![b]\!] \star_S \lambda \varphi_b. \\
 & \qquad \mathcal{C}[\![c]\!] \star_S \lambda \varphi_c. \\
 & \qquad \quad \mathbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), -, \mathsf{whilePS}(\varphi_b, \varphi_c, L, L+1, L+2) \star_D \lambda\_.\texttt{init}_{CS}))
\end{array}
$$

Repeating the last term using $\mathcal{W}(-)$:

$$\mathcal{W}(\mathsf{whilePS}(\varphi_b, \varphi_c, L, L+1, L+2) \star_D \lambda\_.\texttt{init}_{CS})$$

Unfolding $\mathsf{whilePS}(\varphi_b, \varphi_c, L, L+1, L+2)$:

$$\mathcal{W}(\left(\begin{array}{l} \texttt{callcc}\ \lambda\kappa. \\ \quad \texttt{updateCode}[L \mapsto \kappa\,\bullet]\ \star_D\ \lambda\_. \\ \quad \texttt{updateCode}[(L+1) \mapsto \varphi_c\ \star_D\ \lambda\_.(\texttt{jump}\,(L+2))]\ \star_D\ \lambda\_. \\ \quad \texttt{updateCode}[(L+2) \mapsto \varphi_b\ \star_D\ \lambda\beta.\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle]\ \star_D\ \lambda\_. \\ \qquad\quad \texttt{jump}\,(L+2) \end{array}\right)\ \star_D\ \lambda\_.\texttt{init}_{CS})$$

For the sake of readability, define the context $\mathcal{I}(-)$ to be this inner context:

$$\mathcal{I}(-) = \left(\begin{array}{l} \texttt{callcc}\ \lambda\kappa. \\ \quad \texttt{updateCode}[L \mapsto \kappa\,\bullet]\ \star_D\ \lambda\_. \\ \quad \texttt{updateCode}[(L+1) \mapsto \varphi_c\ \star_D\ \lambda\_.(\texttt{jump}\,(L+2))]\ \star_D\ \lambda\_. \\ \quad \texttt{updateCode}[(L+2) \mapsto \varphi_b\ \star_D\ \lambda\beta.\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle]\ \star_D\ \lambda\_. \\ \qquad\quad - \end{array}\right)\ \star_D\ \lambda\_.\texttt{init}_{CS}$$

The rest of this proof has the following structure:

1. Prove with fixed point induction that:

$$\mathcal{W}(\mathcal{I}(\texttt{jump}\,(L+2))) = \mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\varphi_b, \varphi_c)))$$

2. Then using fixed point induction and the induction hypothesis, demonstrate that:

$$\mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\varphi_b, \varphi_c))) = \mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\delta_b, \delta_c)))$$

Assertions (1) and (2) demonstrate that, within the context $\mathcal{W}(\mathcal{I}(\texttt{jump}\,(L+2)))$, the jump $\texttt{jump}\,(L+2)$ behaves identically to the fixpoint $\mathsf{dynwhile}(\delta_b, \delta_c)$. Theorem 9 follows directly from this fact.

Part 1. To show: $\mathcal{W}(\mathcal{I}(\texttt{jump}\,(L+2))) = \mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\varphi_b, \varphi_c)))$

$\boxed{\text{Since, } \texttt{updateCode}[L \mapsto x] \ \star \ \lambda\_.\texttt{jump}\,L = \texttt{updateCode}[L \mapsto x] \ \star \ \lambda\_.x \ :}$

$$\begin{aligned}
\mathcal{W}(\mathcal{I}(\texttt{jump}\,(L+2))) &= \mathcal{W}(\mathcal{I}(\varphi_b \star_D \lambda\beta.\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle)) \\
&= \mathcal{W}(\mathcal{I}(\varphi_b \star_D \lambda\beta.\beta\langle \varphi_c \star_D \lambda\_.(\texttt{jump}\,(L+2)), \texttt{jump}\,L\rangle)) \\
&= \mathcal{W}(\mathcal{I}(\varphi_b \star_D \lambda\beta.\beta\langle \varphi_c \star_D \lambda\_.(\texttt{jump}\,(L+2)), \kappa\bullet\rangle))
\end{aligned}$$

$\boxed{\text{This process of “unrolling” } \texttt{jump}\,(L+2) \text{ can easily be continued:}}$

$$\begin{aligned}
\mathcal{W}(\mathcal{I}(\texttt{jump}\,(L+2))) &= \mathcal{W}(\mathcal{I}(\varphi_b \star_D \lambda\beta.\beta\langle \varphi_c \star_D \lambda\_.(\texttt{jump}\,(L+2)), \kappa\bullet\rangle)) \\
&= \mathcal{W}(\mathcal{I}(\varphi_b \star_D \lambda\beta.\beta\langle \varphi_c \star_D \lambda\_.(\varphi_b \star_D \lambda\beta.\beta\langle \varphi_c \star_D \lambda\_.(\texttt{jump}\,(L+2)), \kappa\bullet\rangle), \kappa\bullet\rangle)) \\
&\ \vdots
\end{aligned}$$

$\boxed{\text{We can “unroll” } \mathsf{dynwhile}(\varphi_b, \varphi_c)\text{:}}$

$$\begin{aligned}
&\mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\varphi_b, \varphi_c))) \\
&\qquad = \mathcal{W}(\mathcal{I}(\varphi_b \star_D \lambda\beta.\beta\langle \varphi_c \star_D \lambda_{.}[\mathsf{dynwhile}(\varphi_b, \varphi_c)], \kappa\bullet\rangle)) \\
&\qquad = \mathcal{W}(\mathcal{I}(\varphi_b \star_D \lambda\beta.\beta\langle \varphi_c \star_D \lambda_{.}\varphi_b \star_D \lambda\beta.\beta\langle \varphi_c \star_D \lambda_{.}[\mathsf{dynwhile}(\varphi_b, \varphi_c)], \kappa\bullet\rangle, \kappa\bullet\rangle)) \\
&\qquad\qquad \vdots
\end{aligned}$$

$\boxed{\text{By fixed point induction:}}$

$$\mathcal{W}(\mathcal{I}(\texttt{jump}\,(L+2))) = \mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\varphi_b, \varphi_c)))$$

Part 2. To show: $\mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\varphi_b, \varphi_c))) = \mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\delta_b, \delta_c)))$

$$\mathcal{W}(\mathcal{I}(\texttt{jump}\,(L+2)))$$

$$= \mathcal{W}\left(\begin{array}{l}
\texttt{callcc}\ \lambda\kappa. \\
\quad \texttt{updateCode}[L \mapsto \kappa\,\bullet] \star_D \lambda\_. \\
\quad \texttt{updateCode}[(L+1) \mapsto \varphi_c \star_D \lambda\_.(\texttt{jump}\,(L+2))] \star_D \lambda\_. \\
\quad \texttt{updateCode}[(L+2) \mapsto \varphi_b \star_D \lambda\beta.\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle] \star_D \lambda\_. \\
\qquad\quad \mathsf{dynwhile}(\varphi_b, \varphi_c)
\end{array}\right) \star_D \lambda\_.\texttt{init}_{CS}$$

Observe that one of the following alternatives holds:

(i) $\mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\varphi_b, \varphi_c))) = \mathcal{W}(\mathcal{I}(\varphi_b \star_D \lambda\beta_1. \underbrace{\varphi_c \star_D \lambda\_. \ldots \varphi_b \star_D \lambda\beta_n.\varphi_c}_{\varphi_c\ n\text{-times}}))$

(ii) $\mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\varphi_b, \varphi_c))) = \mathcal{W}(\mathcal{I}(\bot_{\mathsf{Dynam(void)}}))$

To show: in either case (i) or (ii) that $\mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\varphi_b, \varphi_c))) = \mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\delta_b, \delta_c)))$

Case (i):

$$\mathcal{W}(\mathcal{I}(\varphi_b \star_D \lambda\beta_1.\varphi_c \star_D \lambda\_. \ldots \varphi_b \star_D \lambda\beta_n.\varphi_c))$$

$$= \mathcal{W}(\left(\begin{array}{l} \texttt{callcc}\ \lambda\kappa. \\ \quad \texttt{updateCode}[L \mapsto \kappa \bullet] \star_D \lambda\_. \\ \quad \texttt{updateCode}[(L+1) \mapsto \varphi_c \star_D \lambda\_.(\texttt{jump}\,(L+2))] \star_D \lambda\_. \\ \quad \texttt{updateCode}[(L+2) \mapsto \varphi_b \star_D \lambda\beta.\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle] \star_D \lambda\_. \\ \qquad\quad \varphi_b \star_D \lambda\beta_1.\varphi_c \star_D \lambda\_. \ldots \varphi_b \star_D \lambda\beta_n.\varphi_c \end{array} \right) \star_D \lambda\_.\texttt{init}_{CS})$$

Associativity:

$$= \mathcal{W}\left(\begin{array}{l} \texttt{callcc}\ \lambda\kappa. \\ \quad \texttt{updateCode}[L \mapsto \kappa \bullet] \star_D \lambda\_. \\ \quad \texttt{updateCode}[(L+1) \mapsto \varphi_c \star_D \lambda\_.(\texttt{jump}\,(L+2))] \star_D \lambda\_. \\ \quad \texttt{updateCode}[(L+2) \mapsto \varphi_b \star_D \lambda\beta.\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle] \star_D \lambda\_. \\ \qquad\quad \varphi_b \star_D \lambda\beta_1.\varphi_c \star_D \lambda\_. \ldots \varphi_b \star_D \lambda\beta_n.\varphi_c \star_D \lambda\_.\texttt{init}_{CS} \end{array} \right)$$

Observation Introduction:

$$= \mathcal{W}\left(\begin{array}{l} \texttt{callcc}\ \lambda\kappa. \\ \quad \texttt{updateCode}[L \mapsto \kappa \bullet] \star_D \lambda\_. \\ \quad \texttt{updateCode}[(L+1) \mapsto \varphi_c \star_D \lambda\_.(\texttt{jump}\,(L+2))] \star_D \lambda\_. \\ \quad \texttt{updateCode}[(L+2) \mapsto \varphi_b \star_D \lambda\beta.\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle] \star_D \lambda\_. \\ \qquad \varphi_b \star_D \lambda\beta_1.\varphi_c \star_D \lambda\_. \ldots \varphi_b \star_D \lambda\beta_n. \\ \qquad\quad \mathsf{Obs}(\mathsf{Pre}(a, L+3), \varphi_c \star_D \lambda\_.\texttt{init}_{CS}, \varphi_c \star_D \lambda\_.\texttt{init}_{CS}) \end{array} \right)$$

Inductive hypothesis for c:

$$= \mathcal{W}\left(\begin{array}{l}
\mathtt{callcc}\ \lambda\kappa.\\
\quad \mathtt{updateCode}[L \mapsto \kappa\,\bullet]\ \star_D\ \lambda\_.\\
\quad \mathtt{updateCode}[(L+1) \mapsto \varphi_c\ \star_D\ \lambda\_.(\mathtt{jump}\,(L+2))]\ \star_D\ \lambda\_.\\
\quad \mathtt{updateCode}[(L+2) \mapsto \varphi_b\ \star_D\ \lambda\beta.\beta\langle \mathtt{jump}\,(L+1), \mathtt{jump}\,L\rangle]\ \star_D\ \lambda\_.\\
\qquad \varphi_b\ \star_D\ \lambda\beta_1.\varphi_c\ \star_D\ \lambda\_.\ldots\varphi_b\ \star_D\ \lambda\beta_n.\\
\qquad\qquad \mathsf{Obs}(\mathsf{Pre}(a, L+3), \delta_c\ \star_D\ \lambda\_.\mathtt{init}_{CS}, \varphi_c\ \star_D\ \lambda\_.\mathtt{init}_{CS})
\end{array}\right)$$

$\boxed{\text{Observation Introduction:}}$

$$= \mathcal{W}\left(\begin{array}{l}
\mathtt{callcc}\ \lambda\kappa.\\
\quad \mathtt{updateCode}[L \mapsto \kappa\,\bullet]\ \star_D\ \lambda\_.\\
\quad \mathtt{updateCode}[(L+1) \mapsto \varphi_c\ \star_D\ \lambda\_.(\mathtt{jump}\,(L+2))]\ \star_D\ \lambda\_.\\
\quad \mathtt{updateCode}[(L+2) \mapsto \varphi_b\ \star_D\ \lambda\beta.\beta\langle \mathtt{jump}\,(L+1), \mathtt{jump}\,L\rangle]\ \star_D\ \lambda\_.\\
\varphi_b\ \star_D\ \lambda\beta_1.\varphi_c\ \star_D\ \lambda\_.\ldots\\
\qquad \mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi_b, \varphi_b)\ \star_D\ \lambda\beta_n.\\
\qquad\qquad \mathsf{Obs}(\mathsf{Pre}(a, L+3), \delta_c\ \star_D\ \lambda\_.\mathtt{init}_{CS}, \varphi_c\ \star_D\ \lambda\_.\mathtt{init}_{CS})
\end{array}\right)$$

$\boxed{\text{Induction hypothesis for } b\text{:}}$

$$= \mathcal{W}\left(\begin{array}{l}
\mathtt{callcc}\ \lambda\kappa.\\
\quad \mathtt{updateCode}[L \mapsto \kappa\,\bullet]\ \star_D\ \lambda\_.\\
\quad \mathtt{updateCode}[(L+1) \mapsto \varphi_c\ \star_D\ \lambda\_.(\mathtt{jump}\,(L+2))]\ \star_D\ \lambda\_.\\
\quad \mathtt{updateCode}[(L+2) \mapsto \varphi_b\ \star_D\ \lambda\beta.\beta\langle \mathtt{jump}\,(L+1), \mathtt{jump}\,L\rangle]\ \star_D\ \lambda\_.\\
\qquad \varphi_b\ \star_D\ \lambda\beta_1.\varphi_c\ \star_D\ \lambda\_.\ldots\\
\qquad\qquad \mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_b, \varphi_b)\ \star_D\ \lambda\beta_n.\\
\qquad\qquad\qquad \mathsf{Obs}(\mathsf{Pre}(a, L+3), \delta_c\ \star_D\ \lambda\_.\mathtt{init}_{CS}, \varphi_c\ \star_D\ \lambda\_.\mathtt{init}_{CS})
\end{array}\right)$$

$\boxed{\text{Observation Introduction:}}$

$$
= \mathcal{W}\left(\begin{array}{l}
\texttt{callcc}\ \lambda\kappa.\\
\quad \texttt{updateCode}[L \mapsto \kappa \bullet] \star_D\ \lambda\_.\\
\quad \texttt{updateCode}[(L+1) \mapsto \varphi_c \star_D\ \lambda\_.(\texttt{jump}\,(L+2))] \star_D\ \lambda\_.\\
\quad \texttt{updateCode}[(L+2) \mapsto \varphi_b \star_D\ \lambda\beta.\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle] \star_D\ \lambda\_.\\
\qquad \varphi_b \star_D\ \lambda\beta_1.\varphi_c \star_D\ \lambda\_.\ldots\\
\qquad \mathsf{Obs}(\mathsf{Pre}(a, L+3),\\
\qquad\qquad \left(\begin{array}{l}
\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_b, \varphi_b) \star_D\ \lambda\beta_n.\\
\qquad \mathsf{Obs}(\mathsf{Pre}(a, L+3), \delta_c \star_D\ \lambda\_.\texttt{init}_{CS}, \varphi_c \star_D\ \lambda\_.\texttt{init}_{CS})
\end{array}\right),\\
\qquad\qquad\qquad\qquad \ldots)
\end{array}\right)
$$

$\boxed{\mathsf{Pre}(a, L+3)\text{ discharges }\mathsf{FreshLoc}(a)\text{:}}$

$$
= \mathcal{W}\left(\begin{array}{l}
\texttt{callcc}\ \lambda\kappa.\\
\quad \texttt{updateCode}[L \mapsto \kappa \bullet] \star_D\ \lambda\_.\\
\quad \texttt{updateCode}[(L+1) \mapsto \varphi_c \star_D\ \lambda\_.(\texttt{jump}\,(L+2))] \star_D\ \lambda\_.\\
\quad \texttt{updateCode}[(L+2) \mapsto \varphi_b \star_D\ \lambda\beta.\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle] \star_D\ \lambda\_.\\
\qquad \varphi_b \star_D\ \lambda\beta_1.\varphi_c \star_D\ \lambda\_.\ldots\\
\qquad \mathsf{Obs}(\mathsf{Pre}(a, L+3),\\
\qquad\qquad \left(\begin{array}{l}
\delta_b \star_D\ \lambda\beta_n.\\
\qquad \mathsf{Obs}(\mathsf{Pre}(a, L+3), \delta_c \star_D\ \lambda\_.\texttt{init}_{CS}, \varphi_c \star_D\ \lambda\_.\texttt{init}_{CS})
\end{array}\right),\\
\qquad\qquad\qquad\qquad \ldots)
\end{array}\right)
$$

$\boxed{\mathsf{Pre}(a, L+3)\text{ discharges }\mathsf{Pre}(a, L+3)\text{:}}$

$$
= \mathcal{W}\left(\begin{array}{l}
\texttt{callcc}\ \lambda\kappa.\\
\quad \texttt{updateCode}[L \mapsto \kappa \bullet] \star_D\ \lambda\_.\\
\quad \texttt{updateCode}[(L+1) \mapsto \varphi_c \star_D\ \lambda\_.(\texttt{jump}\,(L+2))] \star_D\ \lambda\_.\\
\quad \texttt{updateCode}[(L+2) \mapsto \varphi_b \star_D\ \lambda\beta.\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle] \star_D\ \lambda\_.\\
\qquad \varphi_b \star_D\ \lambda\beta_1.\varphi_c \star_D\ \lambda\_.\ldots\\
\qquad \mathsf{Obs}(\mathsf{Pre}(a, L+3), (\delta_b \star_D\ \lambda\beta_n.\delta_c \star_D\ \lambda\_.\texttt{init}_{CS}), \ldots)
\end{array}\right)
$$

$\boxed{\texttt{init}_{CS}\text{ commutes with }\delta_b\text{ and }\delta_c\text{ (Follows from Axiom 4):}}$

$$= \mathcal{W}\left(\begin{array}{l}
\texttt{callcc}\ \lambda\kappa. \\
\quad \texttt{updateCode}[L \mapsto \kappa\,\bullet]\ \star_D\ \lambda\_. \\
\quad \texttt{updateCode}[(L+1) \mapsto \varphi_c\ \star_D\ \lambda\_.(\texttt{jump}\,(L+2))]\ \star_D\ \lambda\_. \\
\quad \texttt{updateCode}[(L+2) \mapsto \varphi_b\ \star_D\ \lambda\beta.\beta\langle\texttt{jump}\,(L+1), \texttt{jump}\,L\rangle]\ \star_D\ \lambda\_. \\
\qquad \varphi_b\ \star_D\ \lambda\beta_1.\varphi_c\ \star_D\ \lambda\_.\ldots \\
\qquad \mathsf{Obs}(\mathsf{Pre}(a, L+3), (\texttt{init}_{CS}\ \star_D\ \lambda\_.\delta_b\ \star_D\ \lambda\beta_n.\delta_c), \ldots)
\end{array}\right)$$

Observation Introduction:

$$= \mathcal{W}\left(\begin{array}{l}
\texttt{callcc}\ \lambda\kappa. \\
\quad \texttt{updateCode}[L \mapsto \kappa\,\bullet]\ \star_D\ \lambda\_. \\
\quad \texttt{updateCode}[(L+1) \mapsto \varphi_c\ \star_D\ \lambda\_.(\texttt{jump}\,(L+2))]\ \star_D\ \lambda\_. \\
\quad \texttt{updateCode}[(L+2) \mapsto \varphi_b\ \star_D\ \lambda\beta.\beta\langle\texttt{jump}\,(L+1), \texttt{jump}\,L\rangle]\ \star_D\ \lambda\_. \\
\qquad \varphi_b\ \star_D\ \lambda\beta_1.\varphi_c\ \star_D\ \lambda\_.\ldots\ \star_D\ \varphi_b\ \star_D\ \lambda\beta_{n-1}. \\
\qquad \mathsf{Obs}(\mathsf{Pre}(a, L+3), \\
\qquad\qquad \varphi_c\ \star_D\ \lambda\_.\mathsf{Obs}(\mathsf{Pre}(a, L+3), (\texttt{init}_{CS}\ \star_D\ \lambda\_.\delta_b\ \star_D\ \lambda\beta_n.\delta_c), \ldots), \\
\qquad\qquad\quad \ldots)
\end{array}\right)$$

Lemmas 8 and 10 imply that φ_c preserves $\mathsf{Pre}(a, L+3)$; discharging the inner observation:

$$= \mathcal{W}\left(\begin{array}{l}
\texttt{callcc}\ \lambda\kappa. \\
\quad \texttt{updateCode}[L \mapsto \kappa\,\bullet]\ \star_D\ \lambda\_. \\
\quad \texttt{updateCode}[(L+1) \mapsto \varphi_c\ \star_D\ \lambda\_.(\texttt{jump}\,(L+2))]\ \star_D\ \lambda\_. \\
\quad \texttt{updateCode}[(L+2) \mapsto \varphi_b\ \star_D\ \lambda\beta.\beta\langle\texttt{jump}\,(L+1), \texttt{jump}\,L\rangle]\ \star_D\ \lambda\_. \\
\qquad \varphi_b\ \star_D\ \lambda\beta_1.\varphi_c\ \star_D\ \lambda\_.\ldots\ \star_D\ \varphi_b\ \star_D\ \lambda\beta_{n-1}. \\
\qquad \mathsf{Obs}(\mathsf{Pre}(a, L+3), \varphi_c\ \star_D\ \lambda\_.\texttt{init}_{CS}\ \star_D\ \lambda\_.\delta_b\ \star_D\ \lambda\beta_n.\delta_c, \ldots)
\end{array}\right)$$

This process of pushing $\mathsf{Pre}(a, L+3)$ back through "$\varphi_b\ \star_D\ \lambda\beta_1.\varphi_c\ \star_D\ \lambda\_.\ldots$" continues until all of the φ_b and φ_c are transformed to δ_b and δ_c, respectively:

$$= \mathcal{W}\left(\begin{array}{l} \texttt{callcc}\ \lambda\kappa. \\ \quad \texttt{updateCode}[L \mapsto \kappa\,\bullet] \star_D\ \lambda\_. \\ \quad \texttt{updateCode}[(L+1) \mapsto \varphi_c \star_D\ \lambda\_.(\texttt{jump}\,(L+2))] \star_D\ \lambda\_. \\ \quad \texttt{updateCode}[(L+2) \mapsto \varphi_b \star_D\ \lambda\beta.\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle] \star_D\ \lambda\_. \\ \qquad \mathsf{Obs}(\mathsf{Pre}(a, L+3), \texttt{init}_{CS} \star_D\ \lambda\_.\delta_b \star_D\ \lambda\beta_1.\dots \star_D\ \lambda\beta_n.\delta_c, \dots) \end{array}\right)$$

$\boxed{\text{Observation Introduction:}}$

$$= \mathcal{W}\left(\begin{array}{l} \texttt{callcc}\ \lambda\kappa. \\ \mathsf{Obs}(\mathsf{Pre}(a, L), \\ \qquad \left(\begin{array}{l} \quad \texttt{updateCode}[L \mapsto \kappa\,\bullet] \star_D\ \lambda\_. \\ \quad \texttt{updateCode}[(L+1) \mapsto \varphi_c \star_D\ \lambda\_.(\texttt{jump}\,(L+2))] \star_D\ \lambda\_. \\ \quad \texttt{updateCode}[(L+2) \mapsto \varphi_b \star_D\ \lambda\beta.\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle] \star_D\ \lambda\_. \\ \qquad \mathsf{Obs}(\mathsf{Pre}(a, L+3), \texttt{init}_{CS} \star_D\ \lambda\_.\delta_b \star_D\ \lambda\beta_1.\dots \star_D\ \lambda\beta_n.\delta_c, \dots) \end{array}\right), \\ \qquad \dots) \end{array}\right)$$

$\boxed{\text{Three applications of Lemmas 5 and 6.2 to discharge } \mathsf{Pre}(a, L+3)\text{:}}$

$$= \mathcal{W}\left(\begin{array}{l} \texttt{callcc}\ \lambda\kappa. \\ \mathsf{Obs}(\mathsf{Pre}(a, L), \\ \qquad \left(\begin{array}{l} \quad \texttt{updateCode}[L \mapsto \kappa\,\bullet] \star_D\ \lambda\_. \\ \quad \texttt{updateCode}[(L+1) \mapsto \varphi_c \star_D\ \lambda\_.(\texttt{jump}\,(L+2))] \star_D\ \lambda\_. \\ \quad \texttt{updateCode}[(L+2) \mapsto \varphi_b \star_D\ \lambda\beta.\beta\langle \texttt{jump}\,(L+1), \texttt{jump}\,L\rangle] \star_D\ \lambda\_. \\ \qquad \texttt{init}_{CS} \star_D\ \lambda\_.\delta_b \star_D\ \lambda\beta_1.\dots \star_D\ \lambda\beta_n.\delta_c \end{array}\right), \\ \qquad \dots) \end{array}\right)$$

$\boxed{\text{Because } \texttt{updateCode}(f) \star \lambda\_.\texttt{init}_{CS} = \texttt{init}_{CS}\text{:}}$

$$= \mathcal{W}\left(\begin{array}{l} \texttt{callcc}\ \lambda\kappa. \\ \qquad \mathsf{Obs}(\mathsf{Pre}(a, L), \texttt{init}_{CS} \star_D\ \lambda\_.\delta_b \star_D\ \lambda\beta_1.\dots \star_D\ \lambda\beta_n.\delta_c, \dots) \end{array}\right)$$

$\boxed{\text{Observation Introduction:}}$

$$= \mathcal{W}\left(\begin{array}{l}\mathsf{Obs}(\mathsf{Pre}(a,L),\\ \qquad \left(\begin{array}{l}\texttt{callcc}\ \lambda\kappa.\\ \qquad \mathsf{Obs}(\mathsf{Pre}(a,L), \texttt{init}_{CS}\ \star_D\ \lambda\_.\delta_b\ \star_D\ \lambda\beta_1.\ldots\ \star_D\ \lambda\beta_n.\delta_c, \ldots)\end{array}\right),\\ \qquad\qquad \ldots)\end{array}\right)$$

$\boxed{\texttt{callcc}\ \text{ commutes with } \mathsf{Pre}(a,L)\ \text{(Lemma 7):}}$

$$= \mathcal{W}\left(\begin{array}{l}\mathsf{Obs}(\mathsf{Pre}(a,L),\\ \qquad \left(\begin{array}{l}\mathsf{Obs}(\mathsf{Pre}(a,L),\\ \qquad \texttt{callcc}\ \lambda\kappa.\texttt{init}_{CS}\ \star_D\ \lambda\_.\delta_b\ \star_D\ \lambda\beta_1.\ldots\ \star_D\ \lambda\beta_n.\delta_c, \texttt{callcc}\ \lambda\kappa.\ldots)\end{array}\right),\\ \qquad\qquad \ldots)\end{array}\right)$$

$\boxed{\text{Observation Elimination:}}$

$$= \mathcal{W}\left(\begin{array}{l}\mathsf{Obs}(\mathsf{Pre}(a,L),\\ \qquad \texttt{callcc}\ \lambda\kappa.\\ \qquad\qquad \texttt{init}_{CS}\ \star_D\ \lambda\_.\delta_b\ \star_D\ \lambda\beta_1.\ldots\ \star_D\ \lambda\beta_n.\delta_c, \ldots)\end{array}\right)$$

$\boxed{\text{Recall that } \mathcal{W}(-) = \texttt{rdAddr}\ldots\mathsf{Obs}(\mathsf{Pre}(a,L),-,\ldots)\text{; Observation Elimination:}}$

$$= \mathcal{W}\left(\begin{array}{l}\texttt{callcc}\ \lambda\kappa.\\ \qquad \texttt{init}_{CS}\ \star_D\ \lambda\_.\delta_b\ \star_D\ \lambda\beta_1.\ldots\ \star_D\ \lambda\beta_n.\delta_c\end{array}\right)$$

$\boxed{\text{Commuting } \texttt{init}_{CS} \text{ with } \delta_b \text{ and } \delta_c\text{:}}$

$$= \mathcal{W}\left(\begin{array}{l}\texttt{callcc}\ \lambda\kappa.\\ \qquad \delta_b\ \star_D\ \lambda\beta_1.\ldots\ \star_D\ \lambda\beta_n.\delta_c\ \star_D\ \lambda\_.\texttt{init}_{CS}\end{array}\right)$$

$$= \mathcal{W}(\left(\begin{array}{l}\texttt{callcc}\ \lambda\kappa.\\ \qquad \delta_b\ \star_D\ \lambda\beta_1.\ldots\ \star_D\ \lambda\beta_n.\delta_c\end{array}\right)\ \star_D\ \lambda\_.\texttt{init}_{CS})$$

Since φ_b produced $\beta_1, \ldots, \beta_n$, we can assume without loss of generality that $\beta_1, \ldots, \beta_{n-1}$ are all true (i.e., $\lambda\langle a, -\rangle.a$), and that β_n is false (i.e., $\lambda\langle -, b\rangle.b$). So, the booleans produced by δ_b will produce the same booleans, and hence:

$$
\begin{aligned}
= \mathcal{W}(\mathsf{dynwhile}(\delta_b, \delta_c) \; \star_D \; \lambda\_.\texttt{init}_{CS}) = \; & \texttt{rdAddr} \; \star_S \; \lambda a. \\
& \texttt{getLabel} \; \star_S \; \lambda L. \\
& \mathcal{S}[\![\textbf{while } b \textbf{ do } c]\!] \; \star_S \; \lambda\delta. \\
& \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] \; \star_S \; \lambda\varphi. \\
& \quad \textbf{unit}_S(\mathsf{Obs}(\mathsf{Pre}(a, L), \delta, \varphi) \; \star_D \; \lambda\_.\texttt{init}_{CS})
\end{aligned}
$$

Done with Case (i).

Case (ii). To show that inside $\mathcal{W}(\mathcal{I}(-))$:

$$
\mathsf{dynwhile}(\varphi_b, \varphi_c) = \perp_{\mathsf{Dynam}(\texttt{void})} \implies \mathsf{dynwhile}(\delta_b, \delta_c) = \perp_{\mathsf{Dynam}(\texttt{void})}
$$

Once this fact has been demonstrated, then the Specification 7 holds for "**while** b **do** c."

Assume $\mathsf{dynwhile}(\delta_b, \delta_c)) \neq \perp_{\mathsf{Dynam}(\texttt{void})}$, then for some n,

$$
\mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\delta_b, \delta_c))) = \mathcal{W}(\mathcal{I}(\delta_b \; \star_D \; \lambda\gamma_1.\delta_c \; \star_D \; \ldots \delta_b \; \star_D \; \lambda\gamma_n.\delta_c))
$$

where $\gamma_1, \ldots, \gamma_{n-1} = \texttt{true}$ and $\gamma_n = \texttt{false}$.

Consider "unrolling" $(\mathsf{dynwhile}(\varphi_b, \varphi_c))$ n times:

$$
\mathcal{W}(\mathcal{I}(\varphi_b \; \star_D \; \lambda\beta_1.\varphi_c \; \star_D \; \ldots \varphi_b \; \star_D \; \lambda\beta_n.[\varphi_c \; \star_D \; \lambda\_.(\mathsf{dynwhile}(\varphi_b, \varphi_c))]))
$$

It must be the case that $\beta_i = \texttt{true}$ for all $1 \leq i \leq n$, or else the loop would have terminated and $\mathsf{dynwhile}(\varphi_b, \varphi_c) \neq \perp_{\mathsf{Dynam}(\texttt{void})}$.

$\boxed{\text{So, by Observation Introduction:}}$

$$
\begin{aligned}
& \mathcal{W}(\mathcal{I}(\mathsf{dynwhile}(\varphi_b, \varphi_c))) \\
& \qquad = \mathcal{W}(\mathcal{I}(\varphi_b \; \star_D \; \lambda\beta_1.\varphi_c \; \star_D \; \ldots \varphi_b \; \star_D \; \lambda\beta_n.[\varphi_c \; \star_D \; \lambda\_.(\mathsf{dynwhile}(\varphi_b, \varphi_c))]))
\end{aligned}
$$

$$
= \mathcal{W}(\mathcal{I} \left(\begin{array}{l}
\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi_b, \varphi_b) \; \star_D \; \lambda\beta_1.\varphi_c \star_D \\
\qquad\qquad \vdots \\
\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi_b, \varphi_b) \; \star_D \; \lambda\beta_n.[\varphi_c \; \star_D \; \lambda\_.(\mathsf{dynwhile}(\varphi_b, \varphi_c))]
\end{array} \right))
$$

Expanding $\mathcal{I}(-)$ yields:

$$\mathcal{W}(\left(\begin{array}{l}
\mathtt{callcc}\ \lambda\kappa.\\
\quad \mathtt{updateCode}[L \mapsto \kappa\,\bullet]\ \star_D\ \lambda\_.\\
\quad \mathtt{updateCode}[(L+1) \mapsto \varphi_c\ \star_D\ \lambda\_.(\mathsf{jump}\,(L+2))]\ \star_D\ \lambda\_.\\
\quad \mathtt{updateCode}[(L+2) \mapsto \varphi_b\ \star_D\ \lambda\beta.\beta\langle \mathsf{jump}\,(L+1), \mathsf{jump}\,L\rangle]\ \star_D\ \lambda\_.\\
\left(\begin{array}{l}
\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi_b, \varphi_b)\ \star_D\ \lambda\beta_1.\varphi_c \star_D\\
\qquad\qquad\vdots\\
\mathsf{Obs}(\mathsf{FreshLoc}(a), \varphi_b, \varphi_b)\ \star_D\ \lambda\beta_n.[\varphi_c\ \star_D\ \lambda\_.(\mathsf{dynwhile}(\varphi_b, \varphi_c))]
\end{array}\right)
\end{array}\right)\ \star_D\ \lambda\_.\mathtt{init}_{CS}\ )$$

Induction hypothesis for b:

$$\mathcal{W}(\left(\begin{array}{l}
\mathtt{callcc}\ \lambda\kappa.\\
\quad \mathtt{updateCode}[L \mapsto \kappa\,\bullet]\ \star_D\ \lambda\_.\\
\quad \mathtt{updateCode}[(L+1) \mapsto \varphi_c\ \star_D\ \lambda\_.(\mathsf{jump}\,(L+2))]\ \star_D\ \lambda\_.\\
\quad \mathtt{updateCode}[(L+2) \mapsto \varphi_b\ \star_D\ \lambda\beta.\beta\langle \mathsf{jump}\,(L+1), \mathsf{jump}\,L\rangle]\ \star_D\ \lambda\_.\\
\left(\begin{array}{l}
\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_b, \varphi_b)\ \star_D\ \lambda\beta_1.\varphi_c \star_D\\
\qquad\qquad\vdots\\
\mathsf{Obs}(\mathsf{FreshLoc}(a), \delta_b, \varphi_b)\ \star_D\ \lambda\beta_n.[\varphi_c\ \star_D\ \lambda\_.(\mathsf{dynwhile}(\varphi_b, \varphi_c))]
\end{array}\right)
\end{array}\right)\ \star_D\ \lambda\_.\mathtt{init}_{CS}\ )$$

Recall that $\mathcal{W}(\diamond) = \mathtt{rdAddr}\ldots\mathsf{Obs}(\mathsf{Pre}(a, L), \diamond, \ldots)$, and because neither reading the current continuation nor $\mathtt{updateCode}(f)$ affect the value store, the first observation may be discharged. Also, because command compilations do not affect store shape (Lemma 8), the other observations may be discharged as well:

$$\mathcal{W}(\left(\begin{array}{l}
\mathtt{callcc}\ \lambda\kappa.\\
\quad \mathtt{updateCode}[L \mapsto \kappa\,\bullet]\ \star_D\ \lambda\_.\\
\quad \mathtt{updateCode}[(L+1) \mapsto \varphi_c\ \star_D\ \lambda\_.(\mathsf{jump}\,(L+2))]\ \star_D\ \lambda\_.\\
\quad \mathtt{updateCode}[(L+2) \mapsto \varphi_b\ \star_D\ \lambda\beta.\beta\langle \mathsf{jump}\,(L+1), \mathsf{jump}\,L\rangle]\ \star_D\ \lambda\_.\\
\left(\begin{array}{l}
\delta_b\ \star_D\ \lambda\beta_1.\varphi_c \star_D\\
\qquad\qquad\vdots\\
\delta_b\ \star_D\ \lambda\beta_n.[\varphi_c\ \star_D\ \lambda\_.(\mathsf{dynwhile}(\varphi_b, \varphi_c))]
\end{array}\right)
\end{array}\right)\ \star_D\ \lambda\_.\mathtt{init}_{CS}\ )$$

Since $\beta_n = \texttt{true}$ and $\gamma_n = \texttt{false}$, it must be the case that in

$$\mathcal{W}(\mathcal{I}(\underbrace{\varphi_c \star_D \lambda\_.\ldots\varphi_c}_{\varphi_c \text{ repeated } n \text{ times}} \star_D \lambda\_.\delta_b)) \tag{B.1}$$

δ_b produces `true`, whereas in

$$\mathcal{W}(\mathcal{I}(\underbrace{\delta_c \star_D \lambda\_.\ldots\delta_c}_{\delta_c \text{ repeated } n \text{ times}} \star_D \lambda\_.\delta_b))$$

δ_b produces `false`.

Assuming b is "e_1 leq e_2", we may assume further without loss of generality that a different integer is computed for e_1 in δ_c than in φ_c. That is, in:

$$\mathcal{W}(\mathcal{I}(\underbrace{\varphi_c \star_D \lambda\_.\ldots\varphi_c}_{\varphi_c \text{ repeated } n \text{ times}} \star_D \lambda\_.\delta_{e_1}))$$
$$\mathcal{W}(\mathcal{I}(\underbrace{\delta_c \star_D \lambda\_.\ldots\delta_c}_{\delta_c \text{ repeated } n \text{ times}} \star_D \lambda\_.\delta_{e_1}))$$

different integers are produced by δ_{e_1}. There must be assignments in c for which δ_c and φ_c have different effects on the value store. That is, e_1 must depend on a program variable x for which δ_c and φ_c assign different values. This is clearly impossible given the Assignment Lemma 11. So, in (B.1), δ_b produces `false`, and thus: $\mathsf{dynwhile}(\varphi_b, \varphi_c) \neq \bot_{\mathsf{Dynam(void)}}$. This is a contradiction, so by *reductio ad absurdum*, QED.

□Theorem 9

Appendix C

Proofs from Chapter 5

C.1 Proof of Lemma 14

Lemma 14 (Separability)

$$\begin{array}{l}
\texttt{rdAddr} \star_S \lambda a.\\
(\texttt{inAddr}\ (a+1)\ \mathsf{compile}(e)) \star_S \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle.\\
\quad \mathbf{unit}_S(\mathcal{M}[\![\langle \pi_e \ ;\ \mathtt{ALLOC}(a) \ ;\ a{:=}\mathit{rhs}_e \ ;\ \mathsf{pop}(\mathit{tmps}_e), -a, \{a\}\rangle]\!])\\
\qquad\qquad\qquad = \texttt{rdAddr} \star_S \lambda a.\\
\qquad\qquad\qquad\qquad (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(e)) \star_S \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle.\\
\qquad\qquad\qquad\qquad\quad \mathbf{unit}_S(\mathsf{Negate}(\mathcal{M}[\![\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle]\!], a))
\end{array}$$

Proof of Lemma 14

By induction on terms.

Case e is "n"

Immediate.

Case e is "$-e$":

$$\begin{array}{l}
\texttt{rdAddr} \star_S \lambda a.\\
(\texttt{inAddr}\ (a+1)\ \mathsf{compile}(-e)) \star_S \lambda\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle.\\
\quad \mathbf{unit}_S(\mathcal{M}[\![\langle \pi \ ;\ \mathtt{ALLOC}(a) \ ;\ a{:=}\mathit{rhs} \ ;\ (\mathsf{pop}\ \mathit{tmps}), -a, \{a\}\rangle]\!])
\end{array}$$

$$\begin{array}{l} = \mathtt{rdAddr} \star_S \lambda a. \\ \quad (\mathtt{inAddr}\ (a+1)\ \mathsf{compile}(-e)) \star_S \lambda\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle. \\ \quad\quad \mathbf{unit}_S(\mathcal{M}[\![\langle \left(\begin{array}{l} \pi_e \mathbin{;} \mathtt{ALLOC}(a+1) \mathbin{;} a+1{:=}\mathit{rhs}_e \mathbin{;} (\mathsf{pop}\ \mathit{tmps}_e) \mathbin{;} \\ \quad \mathtt{ALLOC}(a) \mathbin{;} a{:=}-[a+1] \mathbin{;} (\mathsf{pop}\ \{a+1\}) \end{array} \right), -a, \{a\}\rangle]\!]) \end{array}$$

since, for $\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle$ resulting from the compilation of e,

$$\begin{array}{rcl} \pi & = & \pi_e \mathbin{;} \mathtt{ALLOC}(a+1) \mathbin{;} a+1{:=}\mathit{rhs}_e \mathbin{;} (\mathsf{pop}\ \mathit{tmps}_e) \\ \mathit{rhs} & = & -[a+1] \\ \mathit{tmps} & = & \{a+1\} \end{array} \tag{C.1}$$

By the definition of $\mathsf{compile}(-e)$:

$$\begin{array}{l} = \mathtt{rdAddr} \star_S \lambda a. \\ \quad (\mathtt{inAddr}\ (a+1)\ \mathsf{compile}(-e)) \star_S \lambda\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle. \\ \quad\quad \mathbf{unit}_S \left(\begin{array}{l} \mathcal{M}[\![\pi_e]\!] \star_D \lambda\_. \\ \mathcal{M}[\![\mathtt{ALLOC}(a+1)]\!] \star_D \lambda\_. \\ \mathcal{M}[\![a+1{:=}\mathit{rhs}_e]\!] \star_D \lambda\_. \\ \mathcal{M}[\![(\mathsf{pop}\ \mathit{tmps}_e)]\!] \star_D \lambda\_. \\ \mathcal{M}[\![\mathtt{ALLOC}(a)]\!] \star_D \lambda\_. \\ \mathcal{M}[\![a{:=}-[a+1]]\!] \star_D \lambda\_. \\ \mathcal{M}[\![(\mathsf{pop}\ \{a+1\})]\!] \star_D \lambda\_. \\ \mathcal{M}[\![-[a]]\!] \star_D \lambda v. \\ \mathtt{deAlloc}(a) \star_D \lambda\_. \\ \quad \mathbf{unit}_D(v) \end{array} \right) \end{array}$$

Unfolding several $\mathcal{M}[\![-]\!]$:

$$
\begin{array}{l}
= \; \mathtt{rdAddr} \star_S \lambda a. \\
\quad (\mathtt{inAddr}\ (a+1)\ \mathsf{compile}(-e)) \star_S \lambda\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle. \\
\qquad \mathbf{unit}_S \left(
\begin{array}{l}
\mathcal{M}[\![\pi_e]\!] \star_D \lambda\_. \\
\mathtt{Alloc}(a+1) \star_D \lambda\_. \\
\mathcal{M}[\![\mathit{rhs}_e]\!] \star_D \lambda v_e. \\
\mathtt{store}(a+1, v_e) \star_D \lambda\_. \\
\mathcal{M}[\![(\mathsf{pop}\ \mathit{tmps}_e)]\!] \star_D \lambda\_. \\
\mathtt{Alloc}(a) \star_D \lambda\_. \\
(\mathtt{rdLoc}(a+1) \star_D \lambda v_1.\mathtt{store}(a, -v_1)) \star_D \lambda\_. \\
\mathtt{deAlloc}(a+1) \star_D \lambda\_. \\
(\mathtt{rdLoc}(a) \star_D \lambda v_2.\mathbf{unit}_D(-v_2)) \star_D \lambda v. \\
\mathtt{deAlloc}(a) \star_D \lambda\_. \\
\quad \mathbf{unit}_D(v)
\end{array}
\right)
\end{array}
$$

$\boxed{\text{Associativity:}}$

$$
\begin{array}{l}
= \; \mathtt{rdAddr} \star_S \lambda a. \\
\quad (\mathtt{inAddr}\ (a+1)\ \mathsf{compile}(-e)) \star_S \lambda\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle. \\
\qquad \mathbf{unit}_S \left(
\begin{array}{l}
\mathcal{M}[\![\pi_e]\!] \star_D \lambda\_. \\
\mathtt{Alloc}(a+1) \star_D \lambda\_. \\
\mathcal{M}[\![\mathit{rhs}_e]\!] \star_D \lambda v_e. \\
\mathtt{store}(a+1, v_e) \star_D \lambda\_. \\
\mathcal{M}[\![(\mathsf{pop}\ \mathit{tmps}_e)]\!] \star_D \lambda\_. \\
\mathtt{Alloc}(a) \star_D \lambda\_. \\
\mathtt{rdLoc}(a+1) \star_D \lambda v_1. \\
\mathtt{store}(a, -v_1) \star_D \lambda\_. \\
\mathtt{deAlloc}(a+1) \star_D \lambda\_. \\
\mathtt{rdLoc}(a) \star_D \lambda v_2. \\
\mathtt{deAlloc}(a) \star_D \lambda\_. \\
\quad \mathbf{unit}_D(-v_2)
\end{array}
\right)
\end{array}
\tag{C.2}
$$

Expanding the right-hand side of the lemma:

$$
\begin{array}{l}
\texttt{rdAddr} \star_S \ \lambda a. \\
(\texttt{inAddr}\ (a+1)\ \mathsf{compile}(-e)) \ \star_S \ \lambda\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle. \\
\quad \mathbf{unit}_S(\mathsf{Negate}(\mathcal{M}[\![\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle]\!], a))
\end{array}
$$

$$
\begin{array}{rl}
= & \texttt{rdAddr} \star_S \ \lambda a. \\
 & (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(-e)) \ \star_S \ \lambda\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle. \\
 & \qquad \mathbf{unit}_S \left(\begin{array}{l}
 \mathcal{M}[\![\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle]\!] \ \star_D \ \lambda i. \\
 \texttt{Alloc}(a) \ \star_D \ \lambda\_. \\
 \texttt{Thread}(i, a) \ \star_D \ \lambda v. \\
 \texttt{deAlloc}(a) \ \star_D \ \lambda\_. \\
 \qquad \mathbf{unit}_D(-v)
 \end{array} \right)
\end{array}
$$

Definition of $\mathcal{M}[\![\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle]\!]$:

$$
\begin{array}{rl}
= & \texttt{rdAddr} \star_S \ \lambda a. \\
 & (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(-e)) \ \star_S \ \lambda\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle. \\
 & \qquad \mathbf{unit}_S \left(\begin{array}{l}
 \left(\begin{array}{l}
 \mathcal{M}[\![\pi]\!] \ \star_D \ \lambda\_. \\
 \mathcal{M}[\![\mathit{rhs}]\!] \ \star_D \ \lambda v. \\
 \mathcal{M}[\![(\mathsf{pop}\,\mathit{tmps})]\!] \ \star_D \ \lambda\_. \\
 \quad \mathbf{unit}_D(v)
 \end{array} \right) \ \star_D \ \lambda i. \\
 \texttt{Alloc}(a) \ \star_D \ \lambda\_. \\
 \texttt{Thread}(i, a) \ \star_D \ \lambda v. \\
 \texttt{deAlloc}(a) \ \star_D \ \lambda\_. \\
 \qquad \mathbf{unit}_D(-v)
 \end{array} \right)
\end{array}
$$

Using Equation C.1 on page 247:

$$
\begin{array}{rl}
= & \texttt{rdAddr} \ \star_S \ \lambda a. \\
& (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(-e)) \ \star_S \ \lambda\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle. \\
& \mathbf{unit}_S \left(\begin{array}{l}
\left(\begin{array}{l}
\mathcal{M}[\![\pi_e]\!] \ \star_D \ \lambda\_. \\
\mathcal{M}[\![\texttt{ALLOC}(a+1)]\!] \ \star_D \ \lambda\_. \\
\mathcal{M}[\![a+1{:=}\mathit{rhs}_e]\!] \ \star_D \ \lambda\_. \\
\mathcal{M}[\![(\mathsf{pop}\ \mathit{tmps}_e)]\!] \ \star_D \ \lambda\_. \\
\mathcal{M}[\![-[a+1]]\!] \ \star_D \ \lambda v. \\
\mathcal{M}[\![(\mathsf{pop}\ \{a+1\})]\!] \ \star_D \ \lambda\_. \\
\quad \mathbf{unit}_D(v)
\end{array} \right) \ \star_D \ \lambda i. \\
\texttt{Alloc}(a) \ \star_D \ \lambda\_. \\
\texttt{Thread}(i,a) \ \star_D \ \lambda v. \\
\texttt{deAlloc}(a) \ \star_D \ \lambda\_. \\
\qquad \mathbf{unit}_D(-v)
\end{array} \right)
\end{array}
$$

$\boxed{\text{Unfolding } \mathcal{M}[\![a+1{:=}rhs_e]\!] \text{ and } \mathcal{M}[\![-[a+1]]\!]\text{:}}$

$$
\begin{array}{ll}
= & \mathtt{rdAddr} \star_S \ \lambda a. \\
& (\mathtt{inAddr}\ (a+1)\ \mathsf{compile}(-e)) \star_S \ \lambda\langle \pi, rhs, tmps\rangle. \\
& \mathbf{unit}_S \left(\begin{array}{l}
\left(\begin{array}{l}
\mathcal{M}[\![\pi_e]\!] \star_D \ \lambda\_. \\
\mathtt{Alloc}(a+1) \star_D \ \lambda\_. \\
\mathcal{M}[\![rhs_e]\!] \star_D \ \lambda v_e. \\
\mathtt{store}(a+1, v_e) \star_D \ \lambda\_. \\
\mathcal{M}[\![(\mathsf{pop}\ tmps_e)]\!] \star_D \ \lambda\_. \\
\mathtt{rdLoc}(a+1) \star_D \ \lambda v_1. \\
\mathbf{unit}_D(-v_1) \star_D \ \lambda v. \\
\mathtt{deAlloc}(a+1) \star_D \ \lambda\_. \\
\quad \mathbf{unit}_D(v)
\end{array} \right) \star_D \ \lambda i. \\
\mathtt{Alloc}(a) \star_D \ \lambda\_. \\
\mathtt{Thread}(i, a) \star_D \ \lambda v. \\
\mathtt{deAlloc}(a) \star_D \ \lambda\_. \\
\qquad \mathbf{unit}_D(-v)
\end{array} \right)
\end{array}
$$

$\boxed{\text{Associativity and left unit (twice):}}$

$$
\begin{array}{rl}
= & \texttt{rdAddr} \star_S \ \lambda a. \\
& (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(-e)) \ \star_S \ \lambda\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle. \\
& \mathbf{unit}_S \left(\begin{array}{l}
\mathcal{M}[\![\pi_e]\!] \ \star_D \ \lambda\_. \\
\texttt{Alloc}(a+1) \ \star_D \ \lambda\_. \\
\mathcal{M}[\![\mathit{rhs}_e]\!] \ \star_D \ \lambda v_e. \\
\texttt{store}(a+1, v_e) \ \star_D \ \lambda\_. \\
\mathcal{M}[\![(\mathsf{pop}\ \mathit{tmps}_e)]\!] \ \star_D \ \lambda\_. \\
\texttt{rdLoc}(a+1) \ \star_D \ \lambda v_1. \\
\texttt{deAlloc}(a+1) \ \star_D \ \lambda\_. \\
\texttt{Alloc}(a) \ \star_D \ \lambda\_. \\
\texttt{Thread}(-v_1, a) \ \star_D \ \lambda v. \\
\texttt{deAlloc}(a) \ \star_D \ \lambda\_. \\
\qquad \mathbf{unit}_D(-v)
\end{array} \right)
\end{array}
$$

Unfolding definition of $\texttt{Thread}(-v_1, a)$ and left unit:

$$\begin{array}{l}
= \ \texttt{rdAddr} \star_S \ \lambda a. \\
\quad (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(-e)) \star_S \ \lambda\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle. \\
\quad \mathbf{unit}_S \left(\begin{array}{l}
\mathcal{M}[\![\pi_e]\!] \star_D \ \lambda\_. \\
\texttt{Alloc}(a+1) \star_D \ \lambda\_. \\
\mathcal{M}[\![\mathit{rhs}_e]\!] \star_D \ \lambda v_e. \\
\texttt{store}(a+1, v_e) \star_D \ \lambda\_. \\
\mathcal{M}[\![(\mathsf{pop}\ \mathit{tmps}_e)]\!] \star_D \ \lambda\_. \\
\texttt{rdLoc}(a+1) \star_D \ \lambda v_1. \\
\texttt{deAlloc}(a+1) \star_D \ \lambda\_. \\
\texttt{Alloc}(a) \star_D \ \lambda\_. \\
\texttt{store}(a, -v_1) \star_D \ \lambda\_. \\
\texttt{rdLoc}(a) \star_D \ \lambda v_2. \\
\texttt{deAlloc}(a) \star_D \ \lambda\_. \\
\qquad \mathbf{unit}_D(-v_2)
\end{array} \right)
\end{array} \tag{C.3}$$

Comparing terms C.2 and C.3 (on pages 248 and 253, respecitvely), it is clear that they will be equal if the following holds:

$$\left(\begin{array}{l}
\texttt{Alloc}(a) \star_D \ \lambda\_. \\
\texttt{rdLoc}(a+1) \star_D \ \lambda v_1. \\
\texttt{store}(a, -v_1) \star_D \ \lambda\_. \\
\quad \texttt{deAlloc}(a+1)
\end{array} \right) = \left(\begin{array}{l}
\texttt{rdLoc}(a+1) \star_D \ \lambda v_1. \\
\texttt{deAlloc}(a+1) \star_D \ \lambda\_. \\
\texttt{Alloc}(a) \star_D \ \lambda\_. \\
\quad \texttt{store}(a, -v_1)
\end{array} \right) \tag{C.4}$$

This is a simple consequence of Lemma 13.

□Lemma 14.

C.2 Proof of Lemma 15

Lemma 15 (IfThen **lemma**) *For all* π_b, π_c : MachLang (*where* π_b *is produced by* compile(b : *Bool*)), *and* L_c, L_{exit} : *Label:*

$$\mathsf{IfThen}(\mathcal{M}[\![B]\!], \mathcal{M}[\![\pi_c]\!], L_c, L_{exit}) =$$

$$\mathcal{M}[\![(\texttt{ENDLABEL}\ L_{\mathrm{exit}}\ \ (\texttt{SEGM}(L_c,\ \pi_c\ ;\ \texttt{JUMP}\ L_{\mathrm{exit}})\ ;\ \pi_b \diamond \langle \texttt{JUMP} L_c, \texttt{JUMP} L_{exit}\rangle)))]\!]$$

Proof of Lemma 15

$$\begin{aligned}
&\mathsf{IfThen}(\mathcal{M}[\![\pi_b]\!], \mathcal{M}[\![\pi_c]\!], L_c, L_{exit})\\
&\quad = \quad \texttt{callcc}\ \lambda\kappa.\\
&\qquad\quad \texttt{updateCode}[L_{exit} \mapsto \kappa\bullet]\ \star_D\ \lambda\_.\\
&\qquad\quad \texttt{updateCode}[L_c \mapsto \mathcal{M}[\![\pi_c]\!]\ \star_D\ \lambda\_.(\texttt{JUMP} L_{\mathrm{exit}})]\ \star_D\ \lambda\_.\\
&\qquad\quad \mathcal{M}[\![\pi_b]\!]\ \star_D\ \lambda\beta : \mathsf{Dynam}(\texttt{void}) \times \mathsf{Dynam}(\texttt{void}) \to \mathsf{Dynam}(\texttt{void}).\\
&\qquad\qquad \beta\langle \texttt{jump}\, L_c, \texttt{jump}\, L_{\mathrm{exit}}\rangle)
\end{aligned}$$

Folding by definition of $\mathcal{M}[\![\texttt{JUMP} L_c]\!]$ and $\mathcal{M}[\![\texttt{JUMP} L_{exit}]\!]$:

$$\begin{aligned}
&= \quad \texttt{callcc}\ \lambda\kappa.\\
&\qquad \texttt{updateCode}[L_{exit} \mapsto \kappa\bullet]\ \star_D\ \lambda\_.\\
&\qquad \texttt{updateCode}[L_c \mapsto \mathcal{M}[\![\pi_c\ ;\ \texttt{JUMP}\ L_{exit}]\!]]\ \star_D\ \lambda\_.\\
&\qquad \mathcal{M}[\![\pi_b]\!]\ \star_D\ \lambda\beta.\\
&\qquad\quad \beta\langle \mathcal{M}[\![\texttt{JUMP} L_c]\!], \mathcal{M}[\![\texttt{JUMP} L_{exit}]\!]\rangle)
\end{aligned}$$

$\boxed{\text{Folding by definition of } \mathcal{M}[\![\pi_b \diamond \langle \texttt{JUMP}L_c, \texttt{JUMP}L_{exit}\rangle]\!]\text{:}}$

$$
\begin{array}{ll}
= & \texttt{callcc}\ \lambda\kappa. \\
& \quad \texttt{updateCode}[L_{exit} \mapsto \kappa\bullet]\ \star_D\ \lambda\_. \\
& \quad \texttt{updateCode}[L_c \mapsto \mathcal{M}[\![\pi_c\ ;\ \texttt{JUMP}\ L_{exit}]\!]]\ \star_D\ \lambda\_. \\
& \quad \mathcal{M}[\![\pi_b \diamond \langle \texttt{JUMP}L_c, \texttt{JUMP}L_{exit}\rangle]\!]
\end{array}
$$

$\boxed{\text{Folding by definition } \mathcal{M}[\![\texttt{SEGM}(\texttt{L}, \pi)]\!] \text{ and } \mathcal{M}[\![\pi_1\ ;\ \ldots\ ;\ \pi_n]\!]\ \text{:}}$

$$
\begin{array}{ll}
= & \texttt{callcc}\ \lambda\kappa. \\
& \quad \texttt{updateCode}[L_{exit} \mapsto \kappa\bullet]\ \star_D\ \lambda\_. \\
& \quad \mathcal{M}[\![(\texttt{SEGM}(L_c, \pi_c\ ;\ \texttt{JUMP}\ L_{exit})\ ;\ (\pi_b \diamond \langle \texttt{JUMP}L_c, \texttt{JUMP}L_{exit}\rangle))]\!]
\end{array}
$$

$\boxed{\text{Folding by definition } \mathcal{M}[\![(\texttt{ENDLABEL}\ L\ \ \pi)]\!]\text{:}}$

$$
= \quad \mathcal{M}[\![(\texttt{ENDLABEL}\ L_{exit}\ \ (\texttt{SEGM}(L_c, \pi_c\ ;\ \texttt{JUMP}\ L_{exit})\ ;\ (\pi_b \diamond \langle \texttt{JUMP}L_c, \texttt{JUMP}L_{exit}\rangle)))]\!]
$$

□Lemma 15

C.3 Proof of Lemma 16

Lemma 16 ($\Re$-lemma) $\forall a^* : \mathit{Addr}, \rho^* : \mathit{env}, t : \mathsf{Src}.\ \Re(a^*, \rho^*, t)$.

where the relation $\Re(a^* : \mathit{Addr}, \rho^* : \mathit{env}, t : \mathsf{Src})$ is defined as:

$$\begin{array}{l} (\mathsf{FreeVars}(t) \cap \rho^*) < a^* \Longrightarrow \\ \qquad \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ \mathcal{C}[\![t]\!]) = \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ (\mathsf{compile}(t)\ \star_S\ \lambda \pi_t.\mathbf{unit}_S(\mathcal{M}[\![\pi_t]\!]))) \end{array}$$

Proof of Lemma 16

By induction on terms.

Case t is constant c

Immediate.

Case t is $x{:=}e$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ \mathcal{C}[\![x{:=}e]\!])} \\ = & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\ & \qquad\qquad \left(\begin{array}{l} \mathtt{rdEnv}\ \star_S\ \lambda\rho. \\ (\rho\, x)\ \star_S\ \lambda a. \\ \mathcal{C}[\![e]\!]\ \star_S\ \lambda\varphi_e. \\ \quad \mathbf{unit}_S(\varphi_e\ \star_D\ \lambda i : \mathit{int}.\mathtt{store}(a, i)) \end{array} \right)) \\ = & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\ & \qquad\qquad \left(\begin{array}{l} \mathtt{rdEnv}\ \star_S\ \lambda\rho. \\ (\rho\, x)\ \star_S\ \lambda a. \\ (\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ \mathcal{C}[\![e]\!]))\ \star_S\ \lambda\varphi_e. \\ \quad \mathbf{unit}_S(\varphi_e\ \star_D\ \lambda i : \mathit{int}.\mathtt{store}(a, i)) \end{array} \right)) \end{array}$$

By the Induction Hypothesis:

$$
\begin{array}{ll}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
& \quad \left(\begin{array}{l}
\mathtt{rdEnv}\ \star_S\ \lambda\rho. \\
(\rho\, x)\ \star_S\ \lambda a. \\
(\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \left(\begin{array}{c}
\mathsf{compile}(e)\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle. \\
\mathbf{unit}_S(\mathcal{M}[\![\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle]\!])
\end{array} \right))\ \star_S\ \lambda\varphi_e. \\
\quad \mathbf{unit}_S(\varphi_e\ \star_D\ \lambda i : \mathit{int}.\mathtt{store}(a, i))
\end{array} \right))
\end{array}
$$

$$
\begin{array}{ll}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
& \quad \left(\begin{array}{l}
\mathtt{rdEnv}\ \star_S\ \lambda\rho. \\
(\rho\, x)\ \star_S\ \lambda a. \\
\left(\begin{array}{l}
\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ \mathsf{compile}(e))\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle. \\
\quad \mathbf{unit}_S(\mathcal{M}[\![\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle]\!])
\end{array} \right)\ \star_S\ \lambda\varphi_e. \\
\quad \mathbf{unit}_S(\varphi_e\ \star_D\ \lambda i : \mathit{int}.\mathtt{store}(a, i))
\end{array} \right))
\end{array}
$$

$$
\begin{array}{ll}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
& \quad \left(\begin{array}{l}
\mathtt{rdEnv}\ \star_S\ \lambda\rho. \\
(\rho\, x)\ \star_S\ \lambda a. \\
(\mathtt{inEnv}\ \rho^*\ \mathtt{inAddr}\ a^*\ \mathsf{compile}(e))\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle. \\
\quad \mathbf{unit}_S(\mathcal{M}[\![\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle]\!]\ \star_D\ \lambda i : \mathit{int}.\mathtt{store}(a, i))
\end{array} \right))
\end{array}
$$

$$
\begin{array}{ll}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
& \quad \left(\begin{array}{l}
\mathtt{rdEnv}\ \star_S\ \lambda\rho. \\
(\rho\, x)\ \star_S\ \lambda a. \\
(\mathtt{inEnv}\ \rho^*\ \mathtt{inAddr}\ a^*\ \mathsf{compile}(e))\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle. \\
\quad \mathbf{unit}_S(\left(\begin{array}{l}
\mathcal{M}[\![\pi_e]\!]\ \star_D\ \lambda\_. \\
\mathcal{M}[\![\mathit{rhs}_e]\!]\ \star_D\ \lambda i : \mathit{int}. \\
\mathcal{M}[\![(\mathsf{pop}\ \mathit{tmps}_e)]\!]\ \star_D\ \lambda\_. \\
\quad \mathbf{unit}_D(i)
\end{array} \right)\ \star_D\ \lambda i : \mathit{int}.\mathtt{store}(a, i))
\end{array} \right))
\end{array}
$$

$$
\begin{array}{ll}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
& \quad \left(\begin{array}{l}
\mathtt{rdEnv}\ \star_S\ \lambda\rho. \\
(\rho\, x)\ \star_S\ \lambda a. \\
(\mathtt{inEnv}\ \rho^*\ \mathtt{inAddr}\ a^*\ \mathsf{compile}(e))\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle. \\
\quad \mathbf{unit}_S \left(\begin{array}{l}
\mathcal{M}\llbracket \pi_e \rrbracket\ \star_D\ \lambda\_. \\
\mathcal{M}\llbracket \mathit{rhs}_e \rrbracket\ \star_D\ \lambda i : \mathit{int}. \\
\mathcal{M}\llbracket (\mathsf{pop}\ \mathit{tmps}_e) \rrbracket\ \star_D\ \lambda\_. \\
\quad \mathtt{store}(a, i)
\end{array} \right)
\end{array} \right))
\end{array}
$$

$$
\boxed{\mathsf{FreeVars}(x{:=}e) \cap \rho^* < a^*,\ a \notin \mathit{tmps}_e, \text{ and so } (\mathsf{pop}\ \mathit{tmps}_e) \text{ and } \mathtt{store}(a, i) \text{ commute:}}
$$

$$
\begin{array}{ll}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
& \quad \left(\begin{array}{l}
\mathtt{rdEnv}\ \star_S\ \lambda\rho. \\
(\rho\, x)\ \star_S\ \lambda a. \\
(\mathtt{inEnv}\ \rho^*\ \mathtt{inAddr}\ a^*\ \mathsf{compile}(e))\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle. \\
\quad \mathbf{unit}_S \left(\begin{array}{l}
\mathcal{M}\llbracket \pi_e \rrbracket\ \star_D\ \lambda\_. \\
\mathcal{M}\llbracket \mathit{rhs}_e \rrbracket\ \star_D\ \lambda i : \mathit{int}. \\
\mathtt{store}(a, i)\ \star_D\ \lambda\_. \\
\quad \mathcal{M}\llbracket (\mathsf{pop}\ \mathit{tmps}_e) \rrbracket
\end{array} \right)
\end{array} \right))
\end{array}
$$

$$
\begin{array}{ll}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
& \left[\begin{array}{l}
\mathtt{rdEnv}\ \star_S\ \lambda\rho. \\
(\rho\, x)\ \star_S\ \lambda a. \\
(\mathtt{inEnv}\ \rho^*\ \mathtt{inAddr}\ a^*\ \mathsf{compile}(e))\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle. \\
\quad \mathbf{unit}_S(\mathcal{M}\llbracket \pi_e \rrbracket\ \star_D\ \lambda\_.\mathcal{M}\llbracket a{:=}\mathit{rhs}_e \rrbracket\ \star_D\ \lambda\_.\mathcal{M}\llbracket (\mathsf{pop}\ \mathit{tmps}_e) \rrbracket)
\end{array} \right])
\end{array}
$$

$$
\begin{array}{ll}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
& \left[\begin{array}{l}
\mathtt{rdEnv}\ \star_S\ \lambda\rho. \\
(\rho\, x)\ \star_S\ \lambda a. \\
\mathsf{compile}(e)\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle. \\
\quad \mathbf{unit}_S(\mathcal{M}\llbracket \pi_e \rrbracket\ \star_D\ \lambda\_.\mathcal{M}\llbracket a{:=}\mathit{rhs}_e \rrbracket\ \star_D\ \lambda\_.\mathcal{M}\llbracket (\mathsf{pop}\ \mathit{tmps}_e) \rrbracket)
\end{array} \right])
\end{array}
$$

$$= \quad \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \left[\begin{array}{l} \texttt{rdEnv}\ \star_S\ \lambda\rho. \\ (\rho\, x)\ \star_S\ \lambda a. \\ \mathsf{compile}(e)\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle. \\ \qquad \mathbf{unit}_S(\mathcal{M}\llbracket \pi_e\ ;\ a{:=}\mathit{rhs}_e\ ;\ (\mathsf{pop}\ \mathit{tmps}_e)\rrbracket) \end{array}\right])$$

$$= \quad \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \quad \left[\begin{array}{l} \texttt{rdEnv}\ \star_S\ \lambda\rho. \\ (\rho\, x)\ \star_S\ \lambda a. \\ \mathsf{compile}(e)\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle. \\ \qquad \mathbf{unit}_S(\pi_e\ ;\ a{:=}\mathit{rhs}_e\ ;\ (\mathsf{pop}\ \mathit{tmps}_e)) \end{array}\right] \quad \star_S\ \lambda\pi.\mathbf{unit}_S(\mathcal{M}\llbracket\pi\rrbracket))$$

$$= \quad \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ \mathsf{compile}(x{:=}e)\ \star_S\ \lambda\pi.\mathbf{unit}_S(\mathcal{M}\llbracket\pi\rrbracket))$$

$$\boxed{\text{Case } t \text{ is } x_{rval}}$$

$\texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ \mathcal{C}\llbracket x_{rval}\rrbracket)$

$$\begin{array}{ll}
= & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ [\texttt{rdEnv}\ \star_S\ \lambda\rho.(\rho\, x)\ \star_S\ \lambda a.\ \mathbf{unit}_S(\texttt{read}(a))]) \\
= & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ [\texttt{rdEnv}\ \star_S\ \lambda\rho.(\rho\, x)\ \star_S\ \lambda a.\ \mathbf{unit}_S(\mathcal{M}\llbracket\langle\texttt{NOP}, a, \{\}\rangle\rrbracket)]) \\
= & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ [\texttt{rdEnv}\ \star_S\ \lambda\rho.(\rho\, x)\ \star_S\ \lambda a.\ \mathbf{unit}_S(\langle\texttt{NOP}, a, \{\}\rangle)]\ \star_S\ \lambda\pi.\mathbf{unit}_S(\mathcal{M}\llbracket\pi\rrbracket)) \\
= & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ [\texttt{rdEnv}\ \star_S\ \lambda\rho.(\rho\, x)\ \star_S\ \lambda a.\ \mathbf{unit}_S(\langle\texttt{NOP}, a, \{\}\rangle)])\ \star_S\ \lambda\pi.\mathbf{unit}_S(\mathcal{M}\llbracket\pi\rrbracket) \\
= & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ \mathsf{compile}(x_{rval}))\ \star_S\ \lambda\pi.\mathbf{unit}_S(\mathcal{M}\llbracket\pi\rrbracket) \\
= & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ \mathsf{compile}(x_{rval})\ \star_S\ \lambda\pi.\mathbf{unit}_S(\mathcal{M}\llbracket\pi\rrbracket))
\end{array}$$

$$\boxed{\text{Case } t \text{ is } c_1\ ;\ c_2}$$

$\texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ \mathcal{C}\llbracket c_1\ ;\ c_2\rrbracket)$

$$= \quad \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*(\mathcal{C}\llbracket c_1\rrbracket\ \star_S\ \lambda\varphi_1.\mathcal{C}\llbracket c_2\rrbracket\ \star_S\ \lambda\varphi_2.\ \ \mathbf{unit}_S(\varphi_1\ \star_D\ \lambda\_.\varphi_2)))$$

$$
\begin{array}{rl}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
 & \left(\begin{array}{l}
(\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ \mathcal{C}\llbracket c_1 \rrbracket))\ \star_S\ \lambda\varphi_1. \\
(\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ \mathcal{C}\llbracket c_2 \rrbracket))\ \star_S\ \lambda\varphi_2. \\
\quad \mathbf{unit}_S(\varphi_1\ \star_D\ \lambda\_.\varphi_2)
\end{array} \right))
\end{array}
$$

$$
\begin{array}{rl}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
 & \left(\begin{array}{l}
(\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ [\mathsf{compile}(c_1)\ \star_S\ \lambda\pi_1.\mathbf{unit}_S(\mathcal{M}\llbracket \pi_1 \rrbracket)]))\ \star_S\ \lambda\varphi_1. \\
(\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ [\mathsf{compile}(c_2)\ \star_S\ \lambda\pi_2.\mathbf{unit}_S(\mathcal{M}\llbracket \pi_2 \rrbracket)]))\ \star_S\ \lambda\varphi_2. \\
\quad \mathbf{unit}_S(\varphi_1\ \star_D\ \lambda\_.\varphi_2)
\end{array} \right))
\end{array}
$$

$$
\begin{array}{rl}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
 & \left(\begin{array}{l}
[\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ \mathsf{compile}(c_1))]\ \star_S\ \lambda\pi_1.\mathbf{unit}_S(\mathcal{M}\llbracket \pi_1 \rrbracket)\ \star_S\ \lambda\varphi_1. \\
[\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ \mathsf{compile}(c_2))]\ \star_S\ \lambda\pi_2.\mathbf{unit}_S(\mathcal{M}\llbracket \pi_2 \rrbracket)\ \star_S\ \lambda\varphi_2. \\
\quad \mathbf{unit}_S(\varphi_1\ \star_D\ \lambda\_.\varphi_2)
\end{array} \right))
\end{array}
$$

$$
\begin{array}{rl}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
 & \left(\begin{array}{l}
[\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ \mathsf{compile}(c_1))]\ \star_S\ \lambda\pi_1. \\
[\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ \mathsf{compile}(c_2))]\ \star_S\ \lambda\pi_2. \\
\quad \mathbf{unit}_S(\mathcal{M}\llbracket \pi_1 \rrbracket\ \star_D\ \lambda\_.\mathcal{M}\llbracket \pi_2 \rrbracket)
\end{array} \right))
\end{array}
$$

$$
\begin{array}{rl}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
 & \left(\begin{array}{l}
[\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ \mathsf{compile}(c_1))]\ \star_S\ \lambda\pi_1. \\
[\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ \mathsf{compile}(c_2))]\ \star_S\ \lambda\pi_2. \\
\quad \mathbf{unit}_S(\mathcal{M}\llbracket \pi_1\ ;\ \pi_2 \rrbracket)
\end{array} \right))
\end{array}
$$

$$
\begin{array}{rl}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
 & \left(\begin{array}{l}
\mathsf{compile}(c_1)\ \star_S\ \lambda\pi_1. \\
\mathsf{compile}(c_2)\ \star_S\ \lambda\pi_2. \\
\quad \mathbf{unit}_S(\mathcal{M}\llbracket \pi_1\ ;\ \pi_2 \rrbracket)
\end{array} \right))
\end{array}
$$

$$= \quad \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \left(\begin{array}{l} \mathsf{compile}(c_1) \ \star_S\ \lambda\pi_1. \\ \mathsf{compile}(c_2) \ \star_S\ \lambda\pi_2. \\ \quad \mathbf{unit}_S(\pi_1 \ ; \pi_2) \end{array} \right) \star_S\ \lambda\pi.\mathbf{unit}_S(\mathcal{M}[\![\pi]\!]))$$

$$= \quad \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ [\mathsf{compile}(c_1 \ ; c_2)\ \star_S\ \lambda\pi.\mathbf{unit}_S(\mathcal{M}[\![\pi]\!])])$$

$\boxed{\text{Case } t \text{ is } \textbf{new } x \textbf{ in } c}$

$\texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ \mathcal{C}[\![\textbf{new } x \textbf{ in } c]\!])$

$$= \quad \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \left(\begin{array}{l} \texttt{rdEnv}\ \star_S\ \lambda\rho. \\ \texttt{rdAddr}\ \star_S\ \lambda a. \\ \quad [\texttt{inEnv}\ \rho[x \mapsto \mathbf{unit}_S(a)]\ (\texttt{inAddr}\ (a+1)\ \mathcal{C}[\![c]\!])]\ \star_S\ \lambda\varphi_c. \\ \qquad \mathbf{unit}_S(\texttt{Alloc}(a)\ \star_D\ \lambda\_.\varphi_c\ \star_D\ \lambda\_.\texttt{deAlloc}(a)) \end{array} \right))$$

Observe that $\rho = \rho^*$ and $a = a^*$, and because $\mathsf{FreeVars}(\textbf{new } x \textbf{ in } c) \cap \rho^* < a^*$, we can conclude that $\mathsf{FreeVars}(c) \cap \rho[x \mapsto \mathbf{unit}_S(a)] < a + 1$.

$\boxed{\text{So, by the induction hypothesis for } c\text{:}}$

$$\begin{array}{l} = \quad \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \\ \qquad \left(\begin{array}{l} \texttt{rdEnv}\ \star_S\ \lambda\rho. \\ \texttt{rdAddr}\ \star_S\ \lambda a. \\ \quad \texttt{inEnv}\ \rho[x \mapsto \mathbf{unit}_S(a)]\ (\texttt{inAddr}\ (a+1) \left[\begin{array}{c} \mathsf{compile}(c)\ \star_S\ \lambda\pi_c. \\ \mathbf{unit}_S(\mathcal{M}[\![\pi_c]\!]) \end{array} \right])\ \star_S\ \lambda\varphi_c. \\ \qquad \mathbf{unit}_S(\texttt{Alloc}(a)\ \star_D\ \lambda\_.\varphi_c\ \star_D\ \lambda\_.\texttt{deAlloc}(a)) \end{array} \right)) \end{array}$$

$$
\begin{array}{ll}
= & \texttt{inEnv}\ \rho^{*}\ (\texttt{inAddr}\ a^{*} \\
& \quad \left(\begin{array}{l}
\texttt{rdEnv}\ \star_S\ \lambda\rho. \\
\texttt{rdAddr}\ \star_S\ \lambda a. \\
\quad (\texttt{inEnv}\ \rho[x \mapsto \mathbf{unit}_S(a)]\ (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(c)))\ \star_S\ \lambda\pi_c. \\
\quad \mathbf{unit}_S(\mathcal{M}\llbracket\pi_c\rrbracket)\ \star_S\ \lambda\varphi_c. \\
\qquad\quad \mathbf{unit}_S(\texttt{Alloc}(a)\ \star_D\ \lambda\_.\varphi_c\ \star_D\ \lambda\_.\texttt{deAlloc}(a))
\end{array} \right))
\end{array}
$$

$$
\begin{array}{ll}
= & \texttt{inEnv}\ \rho^{*}\ (\texttt{inAddr}\ a^{*} \\
& \quad \left(\begin{array}{l}
\texttt{rdEnv}\ \star_S\ \lambda\rho. \\
\texttt{rdAddr}\ \star_S\ \lambda a. \\
\quad (\texttt{inEnv}\ \rho[x \mapsto \mathbf{unit}_S(a)]\ (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(c)))\ \star_S\ \lambda\pi_c. \\
\qquad\quad \mathbf{unit}_S(\texttt{Alloc}(a)\ \star_D\ \lambda\_.\mathcal{M}\llbracket\pi_c\rrbracket\ \star_D\ \lambda\_.\texttt{deAlloc}(a))
\end{array} \right))
\end{array}
$$

$$
\begin{array}{ll}
= & \texttt{inEnv}\ \rho^{*}\ (\texttt{inAddr}\ a^{*} \\
& \quad \left(\begin{array}{l}
\texttt{rdEnv}\ \star_S\ \lambda\rho. \\
\texttt{rdAddr}\ \star_S\ \lambda a. \\
\quad (\texttt{inEnv}\ \rho[x \mapsto \mathbf{unit}_S(a)]\ (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(c)))\ \star_S\ \lambda\pi_c. \\
\qquad\quad \mathbf{unit}_S(\mathcal{M}\llbracket\texttt{ALLOC}(a)\ ;\ \pi_c\ ;\ \texttt{DEALLOC}(a)\rrbracket)
\end{array} \right))
\end{array}
$$

$$
\begin{array}{ll}
= & \texttt{inEnv}\ \rho^{*}\ (\texttt{inAddr}\ a^{*} \\
& \quad \left(\begin{array}{l}
\texttt{rdEnv}\ \star_S\ \lambda\rho. \\
\texttt{rdAddr}\ \star_S\ \lambda a. \\
\quad (\texttt{inEnv}\ \rho[x \mapsto \mathbf{unit}_S(a)] \\
\qquad (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(c)))\ \star_S\ \lambda\pi_c. \\
\qquad\quad \mathbf{unit}_S(\texttt{ALLOC}(a)\ ;\ \pi_c\ ;\ \texttt{DEALLOC}(a))
\end{array} \right)\ \star_S\ \lambda\pi.\mathbf{unit}_S(\mathcal{M}\llbracket\pi\rrbracket))
\end{array}
$$

$$
=\quad \texttt{inEnv}\ \rho^{*}\ (\texttt{inAddr}\ a^{*}\ [\mathsf{compile}(\mathbf{new}\ x\ \mathbf{in}\ c)\ \star_S\ \lambda\pi.\mathbf{unit}_S(\mathcal{M}\llbracket\pi\rrbracket)])
$$

$\boxed{\text{Case } t \text{ is } -e}$

$\texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ \mathcal{C}\llbracket -e \rrbracket)$

$$
\begin{array}{cl}
= & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \left(\begin{array}{l} \texttt{rdAddr}\ \star_S\ \lambda a. \\ (\texttt{inAddr}\ (a+1)\ \mathcal{C}\llbracket e \rrbracket)\ \star_S\ \lambda \varphi_e. \\ \quad \mathbf{unit}_S(\mathsf{Negate}(\varphi_e, a)) \end{array} \right)) \\
= & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \left(\begin{array}{l} \texttt{rdAddr}\ \star_S\ \lambda a. \\ (\texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ (a+1)\ \mathcal{C}\llbracket e \rrbracket))\ \star_S\ \lambda \varphi_e. \\ \quad \mathbf{unit}_S(\mathsf{Negate}(\varphi_e, a)) \end{array} \right))
\end{array}
$$

Observe that $a = a^*$, and because $\mathsf{FreeVars}(-e) \cap \rho^* < a^*$, we can conclude that $\mathsf{FreeVars}(e) \cap \rho^* < a + 1$.

So, by the induction hypothesis for e:

$$
\begin{array}{cl}
= & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \\
& \left(\begin{array}{l} \texttt{rdAddr}\ \star_S\ \lambda a. \\ \left(\begin{array}{l} \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ (a+1) \\ \qquad \mathsf{compile}(e)\ \star_S\ \lambda \langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle. \\ \qquad\qquad \mathbf{unit}_S(\mathcal{M}\llbracket \langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle \rrbracket)) \end{array} \right) \star_S\ \lambda \varphi_e. \\ \qquad \mathbf{unit}_S(\mathsf{Negate}(\varphi_e, a)) \end{array} \right))
\end{array}
$$

$$
\begin{array}{cl}
= & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \\
& \left(\begin{array}{l} \texttt{rdAddr}\ \star_S\ \lambda a. \\ \left(\begin{array}{l} \texttt{inAddr}\ (a+1) \\ \qquad \mathsf{compile}(e)\ \star_S\ \lambda \langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle. \\ \qquad\qquad \mathbf{unit}_S(\mathcal{M}\llbracket \langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e \rangle \rrbracket) \end{array} \right) \star_S\ \lambda \varphi_e. \\ \qquad \mathbf{unit}_S(\mathsf{Negate}(\varphi_e, a)) \end{array} \right))
\end{array}
$$

$$= \quad \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \left(\begin{array}{l} \texttt{rdAddr}\ \star_S\ \lambda a. \\ (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(e))\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle . \\ \mathbf{unit}_S(\mathcal{M}[\![\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle]\!])\ \star_S\ \lambda\varphi_e . \\ \qquad \mathbf{unit}_S(\mathsf{Negate}(\varphi_e, a)) \end{array} \right))$$

$$= \quad \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \left(\begin{array}{l} \texttt{rdAddr}\ \star_S\ \lambda a. \\ (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(e))\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle . \\ \qquad \mathbf{unit}_S(\mathsf{Negate}(\mathcal{M}[\![\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle]\!], a)) \end{array} \right))$$

$\boxed{\text{By the Separability (Lemma 14):}}$

$$= \quad \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \left(\begin{array}{l} \texttt{rdAddr}\ \star_S\ \lambda a. \\ (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(e))\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle . \\ \qquad \mathbf{unit}_S(\mathcal{M}[\![\langle \pi_e\ ;\ \texttt{ALLOC}(a)\ ;\ a{:=}\mathit{rhs}_e\ ;\ (\mathsf{pop}\ \mathit{tmps}_e), -a, \{a\}\rangle]\!]) \end{array} \right))$$

$$= \quad \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \left(\begin{array}{l} \texttt{rdAddr}\ \star_S\ \lambda a. \\ (\texttt{inAddr}\ (a+1)\ \mathsf{compile}(e))\ \star_S\ \lambda\langle \pi_e, \mathit{rhs}_e, \mathit{tmps}_e\rangle . \\ \quad \mathbf{unit}_S(\langle \pi_e\ ;\ \texttt{ALLOC}(a)\ ;\ a{:=}\mathit{rhs}_e\ ;\ (\mathsf{pop}\ \mathit{tmps}_e), -a, \{a\}\rangle) \end{array} \right) \star_S\ \lambda \mathit{ip}.\mathbf{unit}_S(\mathcal{M}[\![\mathit{ip}]\!]))$$

$$= \quad \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ [\mathsf{compile}(-e)\ \star_S\ \lambda \mathit{ip}.\mathbf{unit}_S(\mathcal{M}[\![\mathit{ip}]\!])])$$

$\boxed{\text{Case } t \text{ is } \textbf{if } b \textbf{ then } c}$

$$\texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ \mathcal{C}[\![\textbf{if } b \textbf{ then } c]\!])$$

$$\begin{array}{ll}
= & \texttt{inEnv}\;\rho^*\;(\texttt{inAddr}\;a^* \\
& \quad \texttt{newlabel}\;\star_S\;\lambda L_{exit}. \\
& \quad \texttt{newlabel}\;\star_S\;\lambda L_c. \\
& \quad \mathcal{C}[\![b]\!]\;\star_S\;\lambda\varphi_b. \\
& \quad \mathcal{C}[\![c]\!]\;\star_S\;\lambda\varphi_c. \\
& \qquad \mathbf{unit}_S(\mathsf{IfThen}(\varphi_b,\varphi_c,L_c,L_{exit})))
\end{array}$$

$$\begin{array}{ll}
= & \texttt{inEnv}\;\rho^*\;(\texttt{inAddr}\;a^* \\
& \texttt{newlabel}\;\star_S\;\lambda L_{\mathrm{exit}}. \\
& \texttt{newlabel}\;\star_S\;\lambda L_c. \\
& (\texttt{inEnv}\;\rho^*\;(\texttt{inAddr}\;a^*\;\mathcal{C}[\![b]\!]))\;\star_S\;\lambda\varphi_b. \\
& (\texttt{inEnv}\;\rho^*\;(\texttt{inAddr}\;a^*\;\mathcal{C}[\![c]\!]))\;\star_S\;\lambda\varphi_c. \\
& \quad \mathbf{unit}_S(\mathsf{IfThen}(\varphi_b,\varphi_c,L_c,L_{\mathrm{exit}})))
\end{array}$$

$$\begin{array}{ll}
= & \texttt{inEnv}\;\rho^*\;(\texttt{inAddr}\;a^* \\
& \quad \texttt{newlabel}\;\star_S\;\lambda L_{\mathrm{exit}}. \\
& \quad \texttt{newlabel}\;\star_S\;\lambda L_c. \\
& \quad \texttt{inEnv}\;\rho^*\;(\texttt{inAddr}\;a^*\;(\mathsf{compile}(b)\;\star_S\;\lambda\pi_b.\mathbf{unit}_S(\mathcal{M}[\![\pi_b]\!])))\;\star_S\;\lambda\varphi_b. \\
& \quad \texttt{inEnv}\;\rho^*\;(\texttt{inAddr}\;a^*\;(\mathsf{compile}(c)\;\star_S\;\lambda\pi_c.\mathbf{unit}_S(\mathcal{M}[\![\pi_c]\!])))\;\star_S\;\lambda\varphi_c. \\
& \qquad \mathbf{unit}_S(\mathsf{IfThen}(\varphi_b,\varphi_c,L_c,L_{\mathrm{exit}})))
\end{array}$$

$$\begin{array}{ll}
= & \texttt{inEnv}\;\rho^*\;(\texttt{inAddr}\;a^* \\
& \quad \texttt{newlabel}\;\star_S\;\lambda L_{\mathrm{exit}}. \\
& \quad \texttt{newlabel}\;\star_S\;\lambda L_c. \\
& \quad (\texttt{inEnv}\;\rho^*\;(\texttt{inAddr}\;a^*\;\mathsf{compile}(b)))\;\star_S\;\lambda\pi_b. \\
& \quad \mathbf{unit}_S(\mathcal{M}[\![\pi_b]\!])\;\star_S\;\lambda\varphi_b. \\
& \quad (\texttt{inEnv}\;\rho^*\;(\texttt{inAddr}\;a^*\;\mathsf{compile}(c)))\;\star_S\;\lambda\pi_c. \\
& \quad \mathbf{unit}_S(\mathcal{M}[\![\pi_c]\!])\;\star_S\;\lambda\varphi_c. \\
& \qquad \mathbf{unit}_S(\mathsf{IfThen}(\varphi_b,\varphi_c,L_c,L_{\mathrm{exit}})))
\end{array}$$

$$\begin{aligned}
= \quad & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \\
& \quad \texttt{newlabel}\ \star_S\ \lambda L_{exit}. \\
& \quad \texttt{newlabel}\ \star_S\ \lambda L_c. \\
& \quad (\texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ \mathsf{compile}(b)))\ \star_S\ \lambda \pi_b. \\
& \quad (\texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ \mathsf{compile}(c)))\ \star_S\ \lambda \pi_c. \\
& \qquad \mathbf{unit}_S(\mathsf{IfThen}(\mathcal{M}[\![\pi_b]\!], \mathcal{M}[\![\pi_c]\!], L_c, L_{exit})))
\end{aligned}$$

$$\begin{aligned}
= \quad & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \\
& \quad \texttt{newlabel}\ \star_S\ \lambda L_{exit}. \\
& \quad \texttt{newlabel}\ \star_S\ \lambda L_c. \\
& \quad \mathsf{compile}(b)\ \star_S\ \lambda \pi_b. \\
& \quad \mathsf{compile}(c)\ \star_S\ \lambda \pi_c. \\
& \qquad \mathbf{unit}_S(\mathsf{IfThen}(\mathcal{M}[\![\pi_b]\!], \mathcal{M}[\![\pi_c]\!], L_c, L_{exit})))
\end{aligned}$$

Let $\Pi_{\text{if}} = (\texttt{ENDLABEL}\ L_{exit}\ \ (\texttt{SEGM}(L_c, \pi_c\ ;\ \texttt{JUMP}\ L_{exit})\ ;\ (B\diamond\langle \texttt{JUMP}L_c, \texttt{JUMP}L_{exit}\rangle)))$, then by Lemma 15, this equals:

$$\begin{aligned}
= \quad & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \\
& \quad \texttt{newlabel}\ \star_S\ \lambda L_{exit}. \\
& \quad \texttt{newlabel}\ \star_S\ \lambda L_c. \\
& \quad \mathsf{compile}(b)\ \star_S\ \lambda \pi_b. \\
& \quad \mathsf{compile}(c)\ \star_S\ \lambda \pi_c. \\
& \qquad \mathbf{unit}_S(\mathcal{M}[\![\Pi_{\text{if}}]\!]))
\end{aligned}$$

$$\begin{aligned}
= \quad & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \\
& \quad \left(\begin{array}{l}
\texttt{newlabel}\ \star_S\ \lambda L_{exit}. \\
\texttt{newlabel}\ \star_S\ \lambda L_c. \\
\mathsf{compile}(b)\ \star_S\ \lambda \pi_b. \\
\mathsf{compile}(c)\ \star_S\ \lambda \pi_c. \\
\qquad \mathbf{unit}_S(\Pi_{\text{if}})
\end{array}\right)\ \star_S\ \lambda \pi.\mathbf{unit}_S(\mathcal{M}[\![\pi]\!])) \\
= \quad & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ [\mathsf{compile}(\mathbf{if}\ b\ \mathbf{then}\ c)\ \star_S\ \lambda \pi.\mathbf{unit}_S(\mathcal{M}[\![\pi]\!])])
\end{aligned}$$

$$\boxed{\text{Case } t \text{ is } e_1 \mathsf{\ leq\ } e_2}$$

Assume $\mathsf{FreeVars}(t) \cap \rho^*) < a^*$.

$\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^*\ \mathcal{C}[\![e_1 \mathsf{\ leq\ } e_2]\!])$

$$= \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \left(\begin{array}{l} \mathtt{rdAddr} \star_S\ \lambda a. \\ \mathtt{inAddr}\ (a+2) \\ \quad \mathcal{C}[\![e_1]\!] \star_S\ \lambda\varphi_1. \\ \quad \mathcal{C}[\![e_2]\!] \star_S\ \lambda\varphi_2. \\ \qquad \mathbf{unit}_S(\mathsf{Lteq}(\varphi_1, \varphi_2, a)) \end{array} \right)$$

By Axiom 2 on page 90, the above occurrences of $\mathcal{C}[\![e_1]\!]$ and $\mathcal{C}[\![e_2]\!]$ can be replaced, respectively, by

$$\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ (a^*+2)\ \mathcal{C}[\![e_1]\!])$$

and

$$\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ (a^*+2)\ \mathcal{C}[\![e_2]\!])$$

.

$$= \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \left(\begin{array}{l} \mathtt{rdAddr} \star_S\ \lambda a. \\ \mathtt{inAddr}\ (a+2) \\ \quad (\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ (a^*+2)\ \mathcal{C}[\![e_1]\!]) \star_S\ \lambda\varphi_1. \\ \quad (\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ (a^*+2)\ \mathcal{C}[\![e_2]\!]) \star_S\ \lambda\varphi_2. \\ \qquad \mathbf{unit}_S(\mathsf{Lteq}(\varphi_1, \varphi_2, a)) \end{array} \right)$$

By induction, the lemma holds for both e_1 and $e2$. Since $\mathsf{FreeVars}(t) \cap \rho^* < a^*$, it holds that $\mathsf{FreeVars}(t) \cap \rho^* < a^* + 2$, and thus the following (dropping the $\mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ (a^*+2)\ -))$:

$$
\begin{array}{l}
= \ \texttt{inEnv}\ \rho^{*}\ (\texttt{inAddr}\ a^{*} \\
\qquad \left(\begin{array}{l}
\texttt{rdAddr}\ \star_S\ \lambda a. \\
\texttt{inAddr}\ (a+2) \\
\quad (\mathsf{compile}(e_1)\ \star_S\ \lambda\langle \pi_1, \mathit{rhs}_1, \mathit{tmps}_1\rangle.\mathcal{M}[\![\langle \pi_1, \mathit{rhs}_1, \mathit{tmps}_1\rangle]\!]).\ \star_S\ \lambda\varphi_1. \\
\quad (\mathsf{compile}(e_2)\ \star_S\ \lambda\langle \pi_2, \mathit{rhs}_2, \mathit{tmps}_2\rangle.\mathcal{M}[\![\langle \pi_2, \mathit{rhs}_2, \mathit{tmps}_2\rangle]\!])\ \star_S\ \lambda\varphi_2. \\
\qquad \mathbf{unit}_S(\mathsf{Lteq}(\varphi_1, \varphi_2, a))
\end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
= \ \texttt{inEnv}\ \rho^{*}\ (\texttt{inAddr}\ a^{*} \\
\qquad \left(\begin{array}{l}
\texttt{rdAddr}\ \star_S\ \lambda a. \\
\texttt{inAddr}\ (a+2) \\
\quad \mathsf{compile}(e_1)\ \star_S\ \lambda\langle \pi_1, \mathit{rhs}_1, \mathit{tmps}_1\rangle. \\
\quad \mathsf{compile}(e_2)\ \star_S\ \lambda\langle \pi_2, \mathit{rhs}_2, \mathit{tmps}_2\rangle. \\
\qquad \mathbf{unit}_S(\mathsf{Lteq}(\mathcal{M}[\![\langle \pi_1, \mathit{rhs}_1, \mathit{tmps}_1\rangle]\!], \mathcal{M}[\![\langle \pi_2, \mathit{rhs}_2, \mathit{tmps}_2\rangle]\!], a))
\end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
= \; \texttt{inEnv}\,\rho^* \,(\texttt{inAddr}\,a^* \\
\qquad \left(\begin{array}{l}
\texttt{rdAddr} \;\star_S\; \lambda a. \\
\texttt{inAddr}\;(a+2) \\
\quad \mathsf{compile}(e_1) \;\star_S\; \lambda\langle \pi_1, \mathit{rhs}_1, \mathit{tmps}_1\rangle. \\
\quad \mathsf{compile}(e_2) \;\star_S\; \lambda\langle \pi_2, \mathit{rhs}_2, \mathit{tmps}_2\rangle. \\
\qquad \mathbf{unit}_S \left(\begin{array}{l}
\mathcal{M}[\![\langle \pi_1, \mathit{rhs}_1, \mathit{tmps}_1\rangle]\!] \;\star_D\; \lambda i. \\
\mathcal{M}[\![\langle \pi_2, \mathit{rhs}_2, \mathit{tmps}_2\rangle]\!] \;\star_D\; \lambda j. \\
\texttt{Alloc}(a) \;\star_D\; \lambda\_. \\
\texttt{Alloc}(a+1) \;\star_D\; \lambda\_. \\
\texttt{Thread}(i, a) \;\star_D\; \lambda v_1. \\
\texttt{Thread}(i, a+1) \;\star_D\; \lambda v_2. \\
\texttt{deAlloc}(a) \;\star_D\; \lambda\_. \\
\texttt{deAlloc}(a+1) \;\star_D\; \lambda\_. \\
\;\mathbf{unit}_D(\lambda\langle \kappa_T, \kappa_F\rangle.(v_1 \leq v_2 \;\rightarrow\; \kappa_T, \kappa_F))
\end{array}\right)
\end{array}\right)
\end{array}
$$

From the definition of `Thread` and `store`$(a+1, j)$ commuting with `read`(a):

$$
\begin{array}{rl}
= & \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \\
& \left(\begin{array}{l}
\texttt{rdAddr}\ \star_S\ \lambda a. \\
\texttt{inAddr}\ (a+2) \\
\quad \mathsf{compile}(e_1)\ \star_S\ \lambda\langle \pi_1, \mathit{rhs}_1, \mathit{tmps}_1\rangle. \\
\quad \mathsf{compile}(e_2)\ \star_S\ \lambda\langle \pi_2, \mathit{rhs}_2, \mathit{tmps}_2\rangle. \\
\qquad \mathbf{unit}_S \left(\begin{array}{l}
\mathcal{M}[\![\langle \pi_1, \mathit{rhs}_1, \mathit{tmps}_1\rangle]\!]\ \star_D\ \lambda i. \\
\mathcal{M}[\![\langle \pi_2, \mathit{rhs}_2, \mathit{tmps}_2\rangle]\!]\ \star_D\ \lambda j. \\
\texttt{Alloc}(a)\ \star_D\ \lambda\_. \\
\texttt{Alloc}(a+1)\ \star_D\ \lambda\_. \\
\texttt{store}(a, i)\ \star_D\ \lambda\_. \\
\texttt{store}(a+1, j)\ \star_D\ \lambda\_. \\
\texttt{read}(a)\ \star_D\ \lambda v_1. \\
\texttt{read}(a+1)\ \star_D\ \lambda v_2. \\
\texttt{deAlloc}(a)\ \star_D\ \lambda\_. \\
\texttt{deAlloc}(a+1)\ \star_D\ \lambda\_. \\
\ \mathbf{unit}_D(\lambda\langle \kappa_T, \kappa_F\rangle.(v_1 \leq v_2\ \to\ \kappa_T, \kappa_F))
\end{array}\right)
\end{array}\right)
\end{array}
$$

$\boxed{\text{Folding the definition of } \mathcal{M}[\![\texttt{BRLEQ}\,a\,(a{+}1)]\!]\text{:}}$

$$
\begin{array}{rl}
= & \texttt{inEnv}\,\rho^*\;(\texttt{inAddr}\,a^* \\
& \left(\begin{array}{l}
\texttt{rdAddr}\;\star_S\;\lambda a. \\
\texttt{inAddr}\;(a+2) \\
\quad \mathsf{compile}(e_1)\;\star_S\;\lambda\langle \pi_1, rhs_1, tmps_1\rangle. \\
\quad \mathsf{compile}(e_2)\;\star_S\;\lambda\langle \pi_2, rhs_2, tmps_2\rangle. \\
\qquad\quad \mathbf{unit}_S \left(\begin{array}{l}
\mathcal{M}[\![\langle \pi_1, rhs_1, tmps_1\rangle]\!]\;\star_D\;\lambda i. \\
\mathcal{M}[\![\langle \pi_2, rhs_2, tmps_2\rangle]\!]\;\star_D\;\lambda j. \\
\texttt{Alloc}(a)\;\star_D\;\lambda\_. \\
\texttt{Alloc}(a+1)\;\star_D\;\lambda\_. \\
\texttt{store}(a,i)\;\star_D\;\lambda\_. \\
\texttt{store}(a+1,j)\;\star_D\;\lambda\_. \\
\;\mathcal{M}[\![\texttt{BRLEQ}\,a\,(a{+}1)]\!]
\end{array}\right)
\end{array}\right)
\end{array}
$$

From the definition of $\mathcal{M}[\![\langle \pi, \mathit{rhs}, \mathit{tmps}\rangle]\!]$ and simplifying twice by left unit:

$$
\begin{array}{rl}
= & \mathtt{inEnv}\ \rho^*\ (\mathtt{inAddr}\ a^* \\
& \left(\begin{array}{l}
\mathtt{rdAddr}\ \star_S\ \lambda a. \\
\mathtt{inAddr}\ (a+2) \\
\quad \mathsf{compile}(e_1)\ \star_S\ \lambda\langle \pi_1, \mathit{rhs}_1, \mathit{tmps}_1\rangle. \\
\quad \mathsf{compile}(e_2)\ \star_S\ \lambda\langle \pi_2, \mathit{rhs}_2, \mathit{tmps}_2\rangle. \\
\qquad \mathbf{unit}_S \left(\begin{array}{l}
\mathcal{M}[\![\pi_1]\!]\ \star_D\ \lambda\_. \\
\mathcal{M}[\![\mathit{rhs}_1]\!]\ \star_D\ \lambda i. \\
\mathcal{M}[\![\mathsf{pop}\ \mathit{tmps}_1]\!]\ \star_D\ \lambda\_. \\
\mathcal{M}[\![\pi_2]\!]\ \star_D\ \lambda\_. \\
\mathcal{M}[\![\mathit{rhs}_2]\!]\ \star_D\ \lambda j. \\
\mathcal{M}[\![\mathsf{pop}\ \mathit{tmps}_2]\!]\ \star_D\ \lambda\_. \\
\mathtt{Alloc}(a)\ \star_D\ \lambda\_. \\
\mathtt{Alloc}(a+1)\ \star_D\ \lambda\_. \\
\mathtt{store}(a, i)\ \star_D\ \lambda\_. \\
\mathtt{store}(a+1, j)\ \star_D\ \lambda\_. \\
\ \mathcal{M}[\![\mathtt{BRLEQ}\ a\ (a{+}1)]\!]
\end{array}\right)
\end{array}\right)
\end{array}
$$

Three observations:

1 By $\mathsf{FreeVars}(t) \cap \rho^* < a^*$ and the induction hypothesis, any free variable in e_1 or e_2 is bound to some $\mathbf{unit}_S(\widehat{a})$ where $\widehat{a} < a^* = a$.

2 By construction (i.e., e_1 and e_2 are compiled for store shape $a + 2$, and allocated addresses strictly increase), $a, a+1 \notin \mathit{tmps}_1 \cup \mathit{tmps}_2$.

3 Thus, neither a nor $a + 1$ occur in the code π_1 and π_2.

These observations allow the above to be refactored as:

$$
\begin{aligned}
= \ & \texttt{inEnv}\ \rho^* \ (\texttt{inAddr}\ a^* \\
& \left(\begin{array}{l}
\texttt{rdAddr} \ \star_S \ \lambda a. \\
\texttt{inAddr}\ (a+2) \\
\quad \mathsf{compile}(e_1) \ \star_S \ \lambda\langle \pi_1, \mathit{rhs}_1, \mathit{tmps}_1\rangle. \\
\quad \mathsf{compile}(e_2) \ \star_S \ \lambda\langle \pi_2, \mathit{rhs}_2, \mathit{tmps}_2\rangle. \\
\qquad \mathbf{unit}_S \left(\begin{array}{l}
\mathcal{M}[\![\pi_1]\!] \ \star_D \ \lambda\_. \\
\texttt{Alloc}(a) \ \star_D \ \lambda\_. \\
(\mathcal{M}[\![\mathit{rhs}_1]\!] \ \star_D \ \lambda i.\texttt{store}(a,i)) \ \star_D \ \lambda\_. \\
\mathcal{M}[\![\pi_2]\!] \ \star_D \ \lambda\_. \\
(\mathcal{M}[\![\mathit{rhs}_2]\!] \ \star_D \ \lambda j.\texttt{store}(a+1,j)) \ \star_D \ \lambda\_. \\
\texttt{Alloc}(a+1) \ \star_D \ \lambda\_. \\
\mathcal{M}[\![\mathsf{pop}\ \mathit{tmps}_1]\!] \ \star_D \ \lambda\_. \\
\mathcal{M}[\![\mathsf{pop}\ \mathit{tmps}_2]\!] \ \star_D \ \lambda\_. \\
\ \mathcal{M}[\![\texttt{BRLEQ}\ a\ (a{+}1)]\!]
\end{array}\right)
\end{array}\right)
\end{aligned}
$$

$$
\begin{aligned}
= \ & \texttt{inEnv}\ \rho^* \ (\texttt{inAddr}\ a^* \\
& \left(\begin{array}{l}
\texttt{rdAddr} \ \star_S \ \lambda a. \\
\texttt{inAddr}\ (a+2) \\
\quad \mathsf{compile}(e_1) \ \star_S \ \lambda\langle \pi_1, \mathit{rhs}_1, \mathit{tmps}_1\rangle. \\
\quad \mathsf{compile}(e_2) \ \star_S \ \lambda\langle \pi_2, \mathit{rhs}_2, \mathit{tmps}_2\rangle. \\
\qquad \mathbf{unit}_S \mathcal{M}[\![\left(\begin{array}{l}
\pi_1 \ ; \texttt{ALLOC}(a) \ ; a{:=}\mathit{rhs}_1 \ ; \pi_2 \ ; a+1{:=}\mathit{rhs}_2 \\
\quad ; \texttt{ALLOC}(a+1) \ ; \mathsf{pop}\ \mathit{tmps}_1 \ ; \mathsf{pop}\ \mathit{tmps}_2 \ ; \texttt{BRLEQ}\ a\ (a{+}1)
\end{array}\right)]\!]
\end{array}\right)
\end{aligned}
$$

$$
= \texttt{inEnv}\ \rho^* \ (\texttt{inAddr}\ a^* \ (\mathsf{compile}(e_1 \ \mathsf{leq} \ e_2) \ \star_S \ \lambda\pi.\mathbf{unit}_S(\mathcal{M}[\![\pi]\!])))
$$

$\boxed{\text{Case } t \text{ is } \mathbf{while}\ b\ \mathbf{do}\ c}$

$\boxed{\text{First, unfolding the meaning of the code returned by:}}$ $\mathsf{compile}(\mathbf{while}\ b\ \mathbf{do}\ c)$:

$$\mathcal{M}[\![\ \texttt{ENDLABEL}\ L_\kappa \begin{pmatrix} \texttt{SEGM}[L_c, \pi_c\ ;\ \texttt{JUMP}\ L_{test}]\ ; \\ \texttt{SEGM}[L_{test}, \pi_b \diamond \langle \texttt{JUMP}L_c, \texttt{JUMP}L_\kappa\rangle]\ ; \\ \texttt{JUMP}L_{test} \end{pmatrix}]\!]$$

$$\begin{aligned}
= \ & \texttt{callcc}\ \lambda\kappa. \\
& \quad \texttt{updateCode}[L_\kappa \mapsto \kappa\bullet]\ \star_D\ \lambda\_. \\
& \quad \mathcal{M}[\![\texttt{SEGM}[L_c, \pi_c\ ;\ \texttt{JUMP}\ L_{test}]]\!]\ \star_D\ \lambda\_. \\
& \quad \mathcal{M}[\![\texttt{SEGM}[L_{test}, \pi_b \diamond \langle \texttt{JUMP}L_c, \texttt{JUMP}L_\kappa\rangle]]\!]\ \star_D\ \lambda\_. \\
& \quad\ \ \mathcal{M}[\![\texttt{JUMP}L_{test}]\!] \\
= \ & \texttt{callcc}\ \lambda\kappa. \\
& \quad \texttt{updateCode}[L_\kappa \mapsto \kappa\bullet]\ \star_D\ \lambda\_. \\
& \quad \texttt{updateCode}[L_c \mapsto \mathcal{M}[\![\pi_c\ ;\ \texttt{JUMP}\ L_{test}]\!]]\ \star_D\ \lambda\_. \\
& \quad \texttt{updateCode}[L_{test} \mapsto \mathcal{M}[\![\pi_b \diamond \langle \texttt{JUMP}L_c, \texttt{JUMP}L_\kappa\rangle]\!]]\ \star_D\ \lambda\_. \\
& \quad\ \ \texttt{jump}\ L_{test} \\
= \ & \texttt{callcc}\ \lambda\kappa. \\
& \quad \texttt{updateCode}[L_\kappa \mapsto \kappa\bullet]\ \star_D\ \lambda\_. \\
& \quad \texttt{updateCode}[L_c \mapsto \mathcal{M}[\![\pi_c]\!]\ \star_D\ \lambda\_.\texttt{jump}\ L_{test}]\ \star_D\ \lambda\_. \\
& \quad \texttt{updateCode}[L_{test} \mapsto \mathcal{M}[\![\pi_b]\!]\ \star_D\ \lambda\beta.\beta\langle \texttt{jump}\ L_c, \texttt{jump}\ L_\kappa\rangle]\ \star_D\ \lambda\_. \\
& \quad\ \ \texttt{jump}\ L_{test} \\
= \ & \mathsf{WhilePS}(\mathcal{M}[\![\pi_b]\!], \mathcal{M}[\![\pi_c]\!], L_{test}, L_c, L_\kappa)
\end{aligned}$$

$\boxed{\text{Assume } \mathsf{FreeVars}(\mathbf{while}\ b\ \mathbf{do}\ c) \cap \rho^* < a^*}$

$$\texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*\ \mathcal{C}[\![\textbf{while}\ b\ \textbf{do}\ c]\!])$$

$$= \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \left(\begin{array}{l} \texttt{newlabel} \star_S\ \lambda L_{test}. \\ \texttt{newlabel} \star_S\ \lambda L_c. \\ \texttt{newlabel} \star_S\ \lambda L_\kappa. \\ \mathcal{C}[\![b]\!]\ \star_S\ \lambda\varphi_b. \\ \mathcal{C}[\![c]\!]\ \star_S\ \lambda\varphi_c. \\ \quad \mathbf{unit}_S(\mathsf{WhilePS}(\varphi_b, \varphi_c, L_{test}, L_c, L_\kappa)) \end{array}\right))$$

As in previous cases, Axiom 2 can be used to show that, within the above context, the induction hypothesis can be applied to $\mathcal{C}[\![b]\!]$ and $\mathcal{C}[\![c]\!]$. That is, $\mathcal{C}[\![b]\!]$ and $\mathcal{C}[\![c]\!]$ can be replaced by $\mathsf{compile}(b)\ \star_S\ \lambda\pi_b.\mathbf{unit}_S(\mathcal{M}[\![\pi_b]\!])$ and $\mathsf{compile}(c)\ \star_S\ \lambda\pi_c.\mathbf{unit}_S(\mathcal{M}[\![\pi_c]\!])$, respectively. Thus:

$$= \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \left(\begin{array}{l} \texttt{newlabel} \star_S\ \lambda L_{test}. \\ \texttt{newlabel} \star_S\ \lambda L_c. \\ \texttt{newlabel} \star_S\ \lambda L_\kappa. \\ (\mathsf{compile}(b)\ \star_S\ \lambda\pi_b.\mathbf{unit}_S(\mathcal{M}[\![\pi_b]\!]))\ \star_S\ \lambda\varphi_b. \\ (\mathsf{compile}(c)\ \star_S\ \lambda\pi_c.\mathbf{unit}_S(\mathcal{M}[\![\pi_c]\!]))\ \star_S\ \lambda\varphi_c. \\ \quad \mathbf{unit}_S(\mathsf{WhilePS}(\varphi_b, \varphi_c, L_{test}, L_c, L_\kappa)) \end{array}\right))$$

$$= \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^* \left(\begin{array}{l} \texttt{newlabel} \star_S\ \lambda L_{test}. \\ \texttt{newlabel} \star_S\ \lambda L_c. \\ \texttt{newlabel} \star_S\ \lambda L_\kappa. \\ \mathsf{compile}(b)\ \star_S\ \lambda\pi_b. \\ \mathsf{compile}(c)\ \star_S\ \lambda\pi_c. \\ \quad \mathbf{unit}_S(\mathsf{WhilePS}(\mathcal{M}[\![\pi_b]\!], \mathcal{M}[\![\pi_c]\!], L_{test}, L_c, L_\kappa)) \end{array}\right))$$

Now, given the initial remarks and the definition of compile(**while** *b* **do** *c*):

$$= \texttt{inEnv}\ \rho^*\ (\texttt{inAddr}\ a^*(\mathsf{compile}(\mathbf{while}\ b\ \mathbf{do}\ c)\ \star_S\ \lambda\pi.\mathbf{unit}_S(\mathcal{M}[\![\pi]\!])))$$

□Lemma 16

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, 1986.

[2] A. Appel, *Modern Compiler Implementation in ML,* Cambridge University Press, New York, 1998.

[3] A. Appel, *Compiling with Continuations,* Cambridge University Press, New York, 1992.

[4] M. Barr and C. Wells, *Category Theory for Computing Science,* Prentice Hall, 1990.

[5] H. Christiansen and Neil Jones. Control Flow Treatment in a Simple Semantics-directed Compiler Generator. *Formal Description of Programming Concepts II*, North-Holland Publishing Company, 1983.

[6] C. Consel and O. Danvy. Static and Dynamic Semantics Processing. *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1991.

[7] O. Danvy, "Type-Directed Partial Evaluation," *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1996.

[8] O. Danvy and R. Vestergaard, "Semantics-Based Compiling: A Case Study in Type-Directed Partial Evaluation," *Eighth International Symposium on Programming Language Implementation and Logic Programming*, 1996, pages 182–497.

[9] R. Davies and F. Pfenning, "A Modal Analysis of Staged Computation," *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1996.

[10] D. Espinosa, "Semantic Lego," Doctoral Dissertation, Columbia University, 1995.

[11] D. P. Friedman, M. Wand, C. T. Haynes. *Essentials of Programming Languages*, The MIT Press, 1992.

[12] W. Harrison and S. Kamin. Compilation as Partial Evaluation of Functor Category Semantics. Unpublished manuscript.

[13] W. Harrison and S. Kamin, "Modular Compilers Based on Monad Transformers," *Proceedings of the IEEE International Conference on Programming Languages*, 1998, pages 122–131.

[14] W. Harrison and S. Kamin, "Metacomputation-based Compiler Architecture," *Proceedings of the Fifth International Conference on the Mathematics of Program Construction*, LNCS1837, 2000, pages 213–229.

[15] G. Hutton and E. Meijer, "Monadic Parser Combinators," *University of Nottingham Department of Computer Science Technical Report NOTTCS-TR-96-4*, 1996.

[16] N. D. Jones and D. A. Schmidt. *Compiler Generation from Denotational Semantics.* In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, LNCS94, 1980, pages 70–93.

[17] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, 1993.

[18] N. D. Jones, P. Sestoft, and H. Sondergaard. An Experiment in Partial Evaluation: The Generation of a Compiler Generator. *Proceedings of the First International Conference on Rewriting Techniques and Applications*, May 1985.

[19] N. D. Jones, P. Sestoft, and H. Sondergaard. MIX: A Self-applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp and Symbolic Computation*, 2, 1989, pages 9–50.

[20] U. Jorring and W. Scherlis, "Compilers and Staging Transformations," *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1986.

[21] S. N. Kamin. *Programming Languages: An Interpreter Based Approach*, Addison-Wesley, 1990.

[22] R. Kelsey and P. Hudak. Realistic Compilation by Program Transformation. *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1989.

[23] P. Lee, *Realistic Compiler Generation,* MIT Press, 1989.

[24] S. Liang, "A Modular Semantics for Compiler Generation," *Yale University Department of Computer Science Technical Report TR-1067*, February 1995.

[25] S. Liang, P. Hudak, and M. Jones, Monad Transformers and Modular Interpreters. *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1995.

[26] S. Liang, "Modular Monadic Semantics and Compilation," Doctoral Thesis, Yale University, 1997.

[27] J. Loeckx and K. Sieber, *Logical Foundations of Program Verification,* Wiley-Teubner Series in Computer Science, 1987.

[28] S. MacLane, *Categories for the Working Mathematician,* Springer-Verlag, 1971.

[29] T. Mogensen. "Separating Binding Times in Language Specifications," *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pages 12–25, 1989.

[30] E. Moggi. An Abstract View of Programming Languages. *Technical Report ECS-LFCS-90-113*, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1990.

[31] E. Moggi, "Notions of Computation and Monads," *Information and Computation 93(1),* pages 55–92, 1991.

[32] F. L. Morris Advice on structuring compilers and proving them correct. *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1973.

[33] P. Mosses. Abstract semantic algebras! *Formal description of programming concepts II*, IFIP IC-2 Working Conference, D. Bjorner, Ed., North-Holland, Amsterdam, 1982, pages 63–88.

[34] P. Mosses, *Action Semantics,* Cambridge University Press, 1992.

[35] S. Muchnick, *Advanced Compiler Design and Implementation,* Morgan Kaufmann, 1997.

[36] H. Nielson and F. Nielson, "Code Generation from two-level denotational metalanguages," in *Programs as Data Objects*, Lecture Notes in Computer Science **217**, Springer Verlag, 1986.

[37] H. Nielson and F. Nielson, "Automatic Binding Time Analysis for a Typed λ-calculus," *Science of Computer Programming* 10, 2, April 1988, pages 139–176.

[38] L. C. Paulson. Compiler Generation from Denotational Semantics. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, Cambridge University Press, pages 219–250, 1984.

[39] S. L. Peyton-Jones and Philip Wadler. "Imperative Functional Programming," *Twentieth ACM Symposium on Principles of Programming Languages*, 1993.

[40] U. Reddy. "Global State Considered Unnecessary: Semantics of Interference-free Imperative Programming," *ACM SIGPLAN Workshop on State in Programming Languages*, pages 120–135, 1993.

[41] J. Reynolds. "The Essence of Algol," *Algorithmic Languages, Proceedings of the International Symposium on Algorithmic Languages*, pages 345–372, 1981.

[42] J. Reynolds. *The Craft of Programming,* Prentice-Hall, 1981.

[43] J. Reynolds. "Using Functor Categories to Generate Intermediate Code," *Proceedings of the ACM Conference on the Principles of Programming Languages*, pages 25–36, 1995.

[44] J. E. Stoy, *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory,* MIT Press, 1977.

[45] W. Taha, "Multi-Stage Programming: Its Theory and Applications," Doctoral Thesis, Oregon Graduate Institute, 1999.

[46] J. Thatcher, E. Wagner, and J. Wright. More On Advice On Structuring Compilers and Proving Them Correct. *Research Report, Mathematical Sciences Department, IBM Thomas J. Watson Research Center*, 1979.

[47] P. Wadler, "The essence of functional programming," *Proceedings of the ACM Conference on the Principles of Programming Languages*, pages 1–14, 1992.

[48] M. Wand. Different Advice on Structuring Compilers and Proving Them Correct *Technical Report No. 95, Indiana University*, September 1980.

[49] M. Wand. Semantics-Directed Machine Architecture. *Proceedings of the ACM Conference on the Principles of Programming Languages*, pages 234–241, 1982.

[50] M. Wand, "Deriving Target Code as a Representation of Continuation Semantics," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pages 496–517, 1982.

[51] M. Wand. Loops in Combinator-based Compilers. *Information and Control*, 57, pages 148–164, 1983.

Curriculum Vitae

William Lawrence Harrison

Professional Preparation:

June 1986	B.A., Mathematics, University of California, Berkeley, CA.
June 1992	M.S., Computer Science, University of California, Davis, CA.
June 2000–present	Post doctoral position, Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, Beaverton, Oregon.

Appointments:

August 1999–May 2000	Visiting Lecturer, Department of Computer Science, Indiana University, Bloomington, IN 47401.
Spring 1999	Visiting Lecturer, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL.
1991-1992	Graduate Research Assistant, University of California at Davis.
1989-1990	Graduate Teaching Assistant, University of California at Davis.

Closely Related Publications:

- William Harrison and Samuel Kamin. Metacomputation-based Compiler Architecture. *Fifth International Conference on the Mathematics of Program Construction*, Ponte de Lima, Portugal. July 4-7, 2000.
- William Harrison and Samuel Kamin. Modular Compilers Based on Monad Transformers. *IEEE International Conference on Computer Languages.* Chicago, Illinois. May 14-16, 1998.

Other Significant Publications:

- William Harrison, Karl Levitt, and Myla Archer. A HOL Mechanization of the Axiomatic Semantics of a Simple Distributed Programming Language. *Higher-Order Logic Theorem Proving and Its Applications*, North-Holland, Netherlands, 1993.
- William Harrison, Karl Levitt, and Myla Archer. Towards a Verified Code Basis for a Secure Distributed Operating System. *University of California at Davis Technical Report CSE-92-19*, October 1992.
- William Harrison. Mechanizing the Axiomatic Semantics for a Programming Language with Asynchronous Send and Receive in HOL. Master's Thesis. University of California at Davis Technical Report CSE-92-20, September 1992.
- William Harrison and Karl Levitt. Mechanizing Security in HOL. *Proceedings of the IEEE HOL Users Group Meeting*, 1991.