REFLECTIONS OF CLOSURES

by

ZACHARY J. SULLIVAN

A DISSERTATION

Presented to the Department of Computer Science
and the Division of Graduate Studies of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December 2023

DISSERTATION APPROVAL PAGE

Student: Zachary J. Sullivan

Title: Reflections of Closures

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer Science by:

Zena M. Ariola	Chair
Boyana Norris	Core Member
Michal Young	Core Member
Benjamin Young	Institutional Representative

and

Krista Chronister	Vice Provost for Graduate Studies

Original approval signatures are on file with the University of Oregon Division of Graduate Studies.

Degree awarded December 2023

DISSERTATION ABSTRACT

Zachary J. Sullivan

Doctor of Philosophy

Department of Computer Science

December 2023

Title: Reflections of Closures

The idea that programs are data forms the bedrock of functional programming languages, but it is also found in object-oriented languages and recent iterations of systems languages. Since passing and returning programs as data is incompatible with the architecture of modern machines, implementations of such a feature gives rise to *closures*, which package code with the environment that it needs to run. The first implementations of these objects are as part of the runtime system of an abstract machine. However, to be able to optimize these structures, compiler writers often choose instead to embed this structure in their code when compiling to lower-level languages in a transformation called closure conversion. While this transformation and closures more generally are well studied with respect to certain types of programming languages, how such a language interacts with different evaluation strategies still remains unstudied in a theoretical setting. Moreover, the current approaches to performing, optimizing, and proving correct this transformation lack the flexibility of other language features.

This thesis develops these ideas by presenting closure conversions for missing evaluation strategies, specifying a new implementation approach that allows for the flexible implementation and optimization of closures, and formalizing them in an intermediate language that captures multiple notions of closures and evaluation strategies in one. Our approach follows from first principles meaning that our closures are a reflection of the environment-based abstract machines that birth them. We develop an approach to reasoning about closures that connects their equational properties with the abstract machines on which they run. Thereby, we can prove not only that closure conversion does not change the output of programs, but that closure conversion removes the need for the runtime system to capture closures.

This dissertation includes previously published, co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Zachary J. Sullivan

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Indiana University, Bloomington, IN, USA

DEGREES AWARDED:

Master of Science, Computer Science, 2018, University of Oregon
Bachelor of Science, Computer Science, 2016, Indiana University

AREAS OF SPECIAL INTEREST:

Compilation
Design of Programming Languages
Verification

PROFESSIONAL EXPERIENCE:

Year Round Intern. Sandia CA. 2019-Present.
Graduate Employee. University of Oregon. 2017-Present.

PUBLICATIONS:

Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. Closure Conversion in Little Pieces. *International Symposium on Principles and Practice of Declarative Programming*. 2023.

Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. Strictly Capturing Non-strict Closures. *Workshop on Partial Evaluation and Program Manipulation.* 2021.

Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. *Making a Faster Curry with Extensional Types.* Haskell Symposium. 2019.

Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. *Codata in Action.* European Symposium on Programming. 2019.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the many people who helped me out along the way. My advisor, Zena, guided me throughout my entire time in Oregon. She, more than anyone, drove me to clearly understand and articulate the necessary parts of my work that I tended to gloss over. Paul Downen helped me not only with the technical knowledge necessary for this work, but also played a huge role along with Zena in influencing my research ideas.

Sandia National Laboratory supported most of my PhD financially. Additionally, the community there played a role in motivating the correctness ideas found in much of my work. My mentors, Johnson-Freyd and Sam Pollard, each took the time to listen to my raw, unfiltered thoughts when I was moving towards a new idea. A special thanks to Sam and Anthony Dario, who provided feedback in polishing and presenting this document.

Lastly, I would like to thank those closest to me. My parents made many sacrifices to put me in a position to succeed in my schooling while also providing unconditional support for anything I might do. My older siblings, being successful in their own endeavours, were role models that motivated me to realize my goals. My friends and especially my girlfriend, Dewi, made my time in graduate school amazing. For all of this, I am grateful.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

CHAPTER I

INTRODUCTION

Higher-order functions are those that may take other functions as arguments. These are *the* essential feature for functional programming languages based on Church's λ-calculus [13]. A canonical example of why this is useful is the map function:

$$\begin{array}{lcl} map\ f\ [] & = & [] \\ map\ f\ (x :: xs) & = & f\ x :: map\ f\ xs \end{array}$$

With such a function, which takes any function f as an argument, we turn one list into another considering while applying f each element. For example, we may now write programs like $map\ (+1)\ [1,2,3]$ and map `print` $[1,2,3]$. In a language like C, we could write a similar function using arrays instead of recursively defined lists and assuming the void type for the elements of the array:

```
void *  map(void (*f) void, void * xs, int len) {
   for (int i = 0; i ++; i < len){
      xs[i] = (*f)(xs[i]);
   }
}
```

However, there is an important difference between function pointers in C and the function that can be passed to higher-order functions like map: the former must be defined at the top level of a program, whereas the latter may be specified by being nested deep within the structure of a program—*i.e.* its lexical scope—or even returned as a result of other programs. For example, we can pass the following function, which is nested within a let-expression, to map as the formal parameter f:

$$\texttt{let}\ x\ \texttt{be}\ 21\ \texttt{in}\ \lambda y.\, x + y$$

In evaluation, some representation of the unevaluated function $\lambda y.\, x + y$ will be bound to f when the body of map is run. Because of its lexical scope, x should be bound to 21 and y should be found as an argument when the code $x + y$ is finally run.

It is not only higher-order functions that involve the passing of lexically-scoped, unevaluated expressions. Indeed, languages with lazy evaluation or non-strictness also wait to evaluate expressions until later. Laziness can be found in languages as old as ALGOL-60, but is found today in more popular languages like Haskell or in the memoizable types of OCaml. For instance, the following Haskell program:

$$\texttt{let}\ x = 21\ \texttt{in}\ [1, 2, x + 2]$$

will immediately return a list of three elements; however, the third element $x + 2$ will remain an unevaluated until some other code forces it to become a value, *e.g.* by printing it. Therefore, we end up with the same need to maintain the lexical structure of the unevaluated code when we return.

1.1 The Emergence of Closures

Though C's code pointers are not as flexible, they can be directly represented by the instruction-set architecture's of modern machines. When passing unevaluated expressions as values in a modern machine, the nested, lexical structure of these languages must be captured in some way as fixed machine code. The language's operational semantics is often specified with a substitution operation, which is an intuitive, high-level description; however, using substitutions runs counter to the goal of generating fixed code since it must create new expressions dynamically. To cope with this, implementations instead keep a local store mapping variables to values; these values are looked up when the variable is needed. The unevaluated expressions of these languages are encoded as *closures*, which are structures containing some code and the environment for which it needs to run. Closures do not appear explicitly in source code so there is a question of whether their usage correctly reflects our source semantics.

Landin's SECD-machine [24] is one of the earliest implementations of the λ-calculus; it made use of closures. However, Plotkin [46] later found that his implementation did not actually capture the the semantics of Church's calculus [13], since it always evaluated the arguments of functions. Plotkin specified a new calculus that *was* reified by the SECD-machine called the *call-by-value* λ-calculus. Later, Krivine [23] would succeed in specifying a machine that captured Church's theory, which a theory we now refer to as *call-by-name*. A notable difference between these two machines is their use of closures: the SECD-machine generated closures only for the functions in the language whereas the Krivine-machine generated closures for any expression in the language when it was a function argument. As optimizations of the call-by-name machines, lazy machines [48] evaluated function arguments at most once by passing closures as a references and updating them with their evaluated form when forced. Indeed, here the source semantics did not properly capture what was occurring in the machine. The source semantics that these machines implemented was described was later specified as *call-by-need* by Ariola *et al.* [9]. We refer to these three methods of computation as *evaluation strategies* and they all treat closures differently in their abstract machines.

Another approach to implementation of functional languages is to compile them to lower-level languages for which we already have a compiler, *e.g.* C. Therein, a transformation called *closure conversion* encodes in the target language a data structure that captures the environment, which the unevaluated code requires, along with a global function, which knows how to re-instantiate that environment before executing. All previous work on such a transformation [34, 36, 3, 38, 39] describes the transformation necessary for compiling *call-by-value* programs. Thus, like the SECD-machine, their transformation only builds closures around functions. The first contribution presented in this thesis is that we describe the missing closure conversions transformations for the call-by-name and call-by-need evaluation strategies [52]. Like their machines, the call-by-name closure conversion transformation generates closure code for function arguments and the call-by-need closure conversion generates extra code that performs the memoization of closure evaluation. We are sure to emphasize the importance of the effect the transformation has on the kind of target language it employs: the target language need not have lexically-scoped passing of delayed computations.

1.2 Reflecting Closures in a Language

Treating closure conversion as a compilation between high- and low-level intermediate languages—as is the case with previous work—is less than ideal for optimizing compilers. An effective approach for optimization is to have a core intermediate language wherein a large amount of optimizations are done by local transformations [44, 51, 5, 39]. Being a global transformation, closure conversion has been excluded from this approach. Another contribution in this thesis is that we propose a new approach to closure conversion which enables it to work within such optimizing compilers [53]. We build on the concept of abstract closures [21, 34, 11], which are reflections of runtime closures as objects within language itself. Specifically, abstract closures are delayed code paired with a delayed substitution. Using them allows us to make closure conversion a local transformation within an intermediate language, instead of between high- and low-level languages.

To formalize the idea, we create a single intermediate language for the optimization of closures that supports the compilation of call-by-name, call-by-value, and call-by-need evaluation strategies. To support the first two, we start with call-by-push-value [27], a language that has a strong enough theory to subsume both call-by-name and call-by-value. We present a new extension to this language that allows it to support the call-by-need evaluation strategy as well; this is why it is called CBPVS or call-by-push-value plus sharing. We then show how to add closures to such a language and specify a closure conversion transformation. To demonstrate that our language is suitable for optimization, aside from the strong equational theory that it gets from being derived from call-by-push-value, we show how closure optimizations that already exist in the literature [50, 19] are local rewrites.

1.3 Reasoning about Closures

Our new approach to closures gives us strong reasoning properties about the relationship between our language and our operational semantics. Not only does closure conversion preserve the evaluation of programs, but its application also allows us to use a simpler operational semantics. This latter property we refer to as the *adequacy of closure conversion*. To prove it, we develop operational semantics for our source and target languages with delayed runtime environments such that we can precisely show that this property along with semantics preservation. We believe that closure conversion should be proved adequate with respect to a particular abstract machine, since closures arise from their implementation in these machines. An inadequate version of the transformation would still require that the machine construct closures within its runtime. We extend previous the logical relation reasoning approaches [3, 34] for closure conversion to work over this new operational semantics.

For our new approach to closure conversion—*i.e.* within the intermediate language—verifying the correctness of our approach to the transformation is a corollary of the soundness of the theory over these delayed runtime environments. That is, all transformations that are expressible within the equational theory, including closure conversion, are correct by construction. This is in contrast to previous work including our extension just mentioned [3, 52, 34] that constructs a bespoke logical relation between the source and target languages for each new closure conversion.

The final contribution of this work is prove that our language CBPVS is sound with respect an abstract machine with delayed substitutions. This verifies our new approach to closure conversion, shows that we correctly added closures to a new language, and as a corollary that the transformation is adequate. In so doing, we created new proof techniques using logical relations for working with delayed substitutions, a sharing heap, and extensional equational theories all at once.

1.4 Outline

Chapter 2 presents important abstract machines and the call-by-name, call-by-value, and call-by-need λ-calculi that reflect those machines. Chapter 3 describes closure conversion which embeds closures in a lower-level language. Here, we present our contribution [52] that extended the transformation to call-by-name and call-by-need languages. Chapter 4 argues for a new approach to reasoning about closures and performing closure conversions within a single compiler intermediate language. We show how this applies to each evaluation strategy separately. Chapter 5 presents a common compiler intermediate language, CBPVS, that unifies the languages of the previous chapter. Chapter 6 shows how closures arise in CBPV and CBPVS by developing environment abstract machines. Chapter 7 proves the correctness of CBPVS's equational theory including closures with respect to this abstract machine. Thereafter, it describes what it means for closure conversions to be adequate, proves how our use of abstract closures satisfies it.

This dissertation is based on the work of two papers—*i.e. Strictly Capturing Non-strict Closures* [52] and *Closure Conversion in Little Pieces* [53]—and their appendices that were authored by Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. In both of the published papers, Zachary J. Sullivan was the primary contributor while being closely advised by Paul Downen and Zena M. Ariola for the technical aspects, design, and framing of work.

CHAPTER II

MACHINES AND THEIR CALCULI

The earliest approaches to mechanizing the λ-calculus were presented as abstract machines; that is, state transitions systems represented at a level suitable for easy mapping to an instruction-set architecture. For various reasons that we will explore, the machines themselves diverged from the original calculus [13] that they were meant to implement. As a reflection of the different machine evaluation behavior, different evaluation strategies, or different source semantics, were created to characterize how the programs would run. Analogously, the work in this dissertation will be a further reflection on these implementations. Specifically, we focus on their *closures*. This chapter presents three forms of semantics for the λ-calculus and their interaction: equational theories, type theories, and abstract machines.

The first kind of semantics are equational theories. Therein, we define a set of axioms relating two expressions of a language. These axioms are combined with the following rules to make up the theory:

$$\frac{}{M = M}\;\textit{Refl.} \qquad \frac{N = M}{M = N}\;\textit{Sym.} \qquad \frac{M = N \quad N = L}{M = L}\;\textit{Trans.} \qquad \frac{M = N}{C[M] = C[N]}\;\textit{Comp.}$$

The first three rules make the theory and equivalence relation. The last of these rules is compatibility; it states that for any context, an expression with a hole, if two expressions are equal, then they are equal when plugged into that context. Such a rule is essential for flexibility when optimizing since it allows us to preform local rewrites anywhere within a program.

$$M, N, L \in \quad \mathit{Expression} \quad ::= \mathsf{b} \mid x \mid \lambda x.M \mid M\ N$$

(a) Syntax

$$\begin{aligned} \lambda x.\, M &=_\alpha \lambda y.\, M[y/x] \\ (\lambda x.\, M)\ N &=_\beta M[N/x] \\ \lambda x.\, M\ x &=_\eta M \end{aligned}$$

(b) Axioms

Figure 2.1. Church's λ-calculus

Let us consider Church's λ-calculus [13] presented Figure 2.1 as our first example of an equational theory. It contains only a few syntactic constructions that make up the expressions of the language. The anonymous function $\lambda x.\, M$ takes an argument and binds it to x in the expression M. The application form $M\ N$ will call the function M with the argument N. Finally, we reference the bound variables with the variable expression. Along with the syntax, Figure 2.1 specifies three axioms. The first, α, states that any two functions are equivalent if they are the same except for the name of their formal parameter. The second, β, describes how we compute with functions: if a function is applied, then it is equal to replacing the occurrence of the variable with the argument within the function's body. The third, η, says that a function is equivalent to a new function that immediately applies it to its argument. This last is equivalent to function extensionality

$$M\ x = N\ x \Longrightarrow M = N$$

stating that if two functions behave the same when applied to the same arguments, then they are the same.

The axioms make use of a meta-syntactic function called *substitution*. Generically, $M[N/x]$ means that we recursively traverse the expressions M and replace free

$$\frac{}{\Gamma \vdash \mathsf{b} : B}B \quad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}var \quad \frac{\Gamma, x{:}\tau \vdash M : \sigma}{\Gamma \vdash \lambda x.\, M : \tau \to \sigma}\to_I \quad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\, N : \tau}\to_E$$

Figure 2.2. Types for λ-expressions.

occurrences of x with N to generate a new expression. The set of expressions capable of being substituted, M in the definition below, are referred to as the *values* of the calculus.

Definition 2.1 (Substitution)**.**

$$\begin{array}{rcl}
\mathsf{b}[M/x] & = & \mathsf{b} \\
x[M/x] & = & M \\
y[M/x] & = & y \\
(\lambda x.\, N)[M/x] & = & \lambda x.\, N \\
(\lambda y.\, N)[M/x] & = & \lambda y.\, N[M/x] \\
(N\, L)[M/x] & = & N[M/x]\; L[M/x]
\end{array}$$

The second notion of semantics are typing rules. Those for the simply-typed λ-calculus are presented in Figure 2.2. These are “static” in the sense that they do not change how we evaluate programs, but we use them to ensure that the program is well formed. The typing rules are important for η axioms, in particular, which only apply when they are applied to programs of the correct type. This is not a problem in calculi with only functions types, since everything can be treated as a function; however, it does not make sense to η expand a pair into a function. We codify this restriction by giving the following definition of syntactic equality:

$$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \tau \quad M = N}{\Gamma \vdash M = N : \tau}$$

$$\begin{array}{rcl} S \in & \textit{Stack} & ::= \varepsilon \mid \mathbb{V} \cdot S \\ E \in & \textit{Machine Environment} & ::= \varepsilon \mid E, \mathbb{V}/x \\ C \in & \textit{Control} & ::= \varepsilon \mid M \cdot C \mid \mathsf{ap} \cdot C \\ D \in & \textit{Dump} & ::= \varepsilon \mid (S, E, C, D) \\ \mathbb{V} \in & \textit{Machine Value} & ::= \mathsf{b} \mid (E, \lambda x. M) \\ & \textit{Configuration} & ::= \langle\!\langle S \parallel E \parallel C \parallel D \rangle\!\rangle \end{array}$$

(a) Machine Syntax

$$\begin{array}{rcl} \langle\!\langle S \parallel E \parallel \mathsf{b} \cdot C \parallel D \rangle\!\rangle & \longmapsto_1 & \langle\!\langle \mathsf{b} \cdot S \parallel E \parallel C \parallel D \rangle\!\rangle \\ \langle\!\langle S \parallel E \parallel x \cdot C \parallel D \rangle\!\rangle & \longmapsto_2 & \langle\!\langle x[E] \cdot S \parallel E \parallel C \parallel D \rangle\!\rangle \\ \langle\!\langle S \parallel E \parallel \lambda x. M \cdot C \parallel D \rangle\!\rangle & \longmapsto_3 & \langle\!\langle (E, \lambda x. M) \cdot S \parallel E \parallel C \parallel D \rangle\!\rangle \\ \langle\!\langle S \parallel E \parallel M\, N \cdot C \parallel D \rangle\!\rangle & \longmapsto_4 & \langle\!\langle S \parallel E \parallel N \cdot M \cdot \mathsf{ap} \cdot C \parallel D \rangle\!\rangle \\ \langle\!\langle (E', \lambda x. M) \cdot \mathbb{V} \cdot S \parallel E \parallel \mathsf{ap} \cdot C \parallel D \rangle\!\rangle & \longmapsto_5 & \langle\!\langle \varepsilon \parallel E', \mathbb{V}/x \parallel M \cdot \varepsilon \parallel (S, E, C, D) \rangle\!\rangle \\ \langle\!\langle \mathbb{V} \cdot \varepsilon \parallel E \parallel \varepsilon \parallel (S', E', C', D') \rangle\!\rangle & \longmapsto_6 & \langle\!\langle \mathbb{V} \cdot S' \parallel E' \parallel C' \parallel D' \rangle\!\rangle \end{array}$$

(b) Transitions

Figure 2.3. The SECD Machine

We will only have one type system for expressions, but we will present several calculi and abstract machines which capture different notions of dynamically evaluating these expressions. While substitution is useful for intuitive explanations of calculi, it is not good for implementation on machines since we must generate fixed code sequences for programs. Thus, our abstract machines will be presented with *delayed* substitutions, wherein "substitution" only occurs if the variable is demanded at runtime.

2.1 The SECD Machine and Call-by-Value

The earliest cited λ-calculus abstract machine was Landin's SECD machine [24]. The machine is named for the four parts of its state: S, an intermediate result stack; E, an environment containing a mapping from variables to values; C, a control stack; and D, a dump of a machine state. Evaluation of the machine is a transition, denoted ($\longmapsto$), from machine state to machine state.

Figure 2.3 gives the syntax and transition rules for the machine. The result stack holds only values and the environment maps to values. The notion of value for the SECD machine, and many other abstract machines, is different from the notion often used for reduction theory, structural operational semantics, and equational theories wherein values are the subset of the whole language which are substitutable. Instead, values refer to objects that can be mapped to from the machine's environment; hence, we sometime refer to them as *machine values* to contrast them with those of the source language. For an object that behaves like an integer, constants like 4 are values whereas arithmetic expressions like 3 + 1 must be evaluated before they can be placed on the result stack or in the environment. In the case of functions, the SECD machine constructs function closures, $(E, \lambda x.\, y)$, which pair a λ-expression with some environment.

The control stack is so named because the next state transition always depends on the value at the top of this stack. When the top of the control stack is an application expression, the application is deconstructed and an application marker, the function, and the argument are placed on the control stack, *in that order*. This deconstruction can be seen as searching for the next expression to evaluate. In this case, we evaluate the function argument then the function itself. When both the argument and the function are evaluated to a value, an application marker will be at the top of the control stack which triggers the application.

The dump of the SECD machine is used to return from function calls. When a function is applied, in transition 5, the state of the machine is saved. The state is re-instantiated when the function returns, in transition 6, and after the machine value computed by the function call is added to the result stack. Note that returning from a function call returns to the environment *before* that call. This means that any unevaluated code that is returned ought to have saved its environment in a closure.

Definition 2.2 (SECD Evaluation). $\mathrm{Eval}_{\mathrm{SECD}}(M) = \mathsf{b}$ *where* $\langle\!\langle \varepsilon \parallel \varepsilon \parallel M \cdot \varepsilon \parallel \varepsilon \rangle\!\rangle \longmapsto^* \langle\!\langle \mathsf{b} \cdot \varepsilon \parallel E \parallel \varepsilon \parallel \varepsilon \rangle\!\rangle$.

As an example, consider the evaluation trace for the program $(\lambda x.\, \lambda y.\, x)\ 4\ 2$ with an arbitrary starting S, E, C, and D (*i.e.* anywhere in a program):

$$\begin{aligned}
&\langle\!\langle S \parallel E \parallel (\lambda x.\, \lambda y.\, x)\ 4\ 2 \cdot C \parallel D \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle S \parallel E \parallel 2 \cdot (\lambda x.\, \lambda y.\, x)\ 4 \cdot \mathsf{ap} \cdot C \parallel D \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle 2 \cdot S \parallel E \parallel (\lambda x.\, \lambda y.\, x)\ 4 \cdot \mathsf{ap} \cdot C \parallel D \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle 2 \cdot S \parallel E \parallel 4 \cdot \lambda x.\, \lambda y.\, x \cdot \mathsf{ap} \cdot \mathsf{ap} \cdot C \parallel D \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle 4 \cdot 2 \cdot S \parallel E \parallel \lambda x.\, \lambda y.\, x \cdot \mathsf{ap} \cdot \mathsf{ap} \cdot C \parallel D \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle (E, \lambda x.\, \lambda y.\, x) \cdot 4 \cdot 2 \cdot S \parallel E \parallel \mathsf{ap} \cdot \mathsf{ap} \cdot C \parallel D \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle \varepsilon \parallel E, 4/x \parallel \lambda y.\, x \cdot \varepsilon \parallel (2 \cdot S, E, \mathsf{ap} \cdot C, D) \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle ((E, 4/x), \lambda y.\, x) \cdot \varepsilon \parallel E, 4/x \parallel \varepsilon \parallel (2 \cdot S, E, \mathsf{ap} \cdot C, D) \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle ((E, 4/x), \lambda y.\, x) \cdot 2 \cdot S \parallel E \parallel \mathsf{ap} \cdot C \parallel D \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle \varepsilon \parallel E, 4/x, 2/y \parallel x \cdot \varepsilon \parallel (S, E, C, D) \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle 4 \cdot \varepsilon \parallel E, 4/x, 2/y \parallel \varepsilon \parallel (S, E, C, D) \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle 4 \cdot S \parallel E \parallel C \parallel D \rangle\!\rangle
\end{aligned}$$

This machine does not match the behavior of the β law in Church's theory. If we consider the program $(\lambda x.\, y)\ \Omega$, where Ω is the infinitely looping expression $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$, then it would reduce to the normal form y in Church's calculus, but would loop forever in the machine. This is because the machine must continue to evaluate the argument of a function until it reaches a machine value. The property of evaluating function arguments before continuing is referred to as being *strictly* evaluated. Noting the difference in the strictness of the machine and Church's calculus, Plotkin [46] specified a new calculus called *call-by-value* in Figure 2.4. There is a sub-syntax

$$V, W \in \mathit{Value} ::= \mathsf{b} \mid x \mid \lambda x.\, M$$

(a) Syntax

$$\begin{aligned}(\lambda x.\, M)\; V &=_\beta M[V/x] \\ \lambda x.\, V\; x &=_\eta V\end{aligned}$$

(b) Axioms

Figure 2.4. The Call-by-Value Calculus

of values that determine when the β law is applicable. Just like the SECD machine, a call-by-value calculus yields no normal form for the program $(\lambda x.\, y)\ \Omega$ because Ω has no normal form and must be evaluated. The η axiom for functions in call-by-value must be restricted to values; otherwise, an η-reduction could turn a value into a non-value breaking its connection with the SECD machine. That connection is captured in the following proposition proved by Plotkin [46].

Proposition 2.1. $\mathrm{Eval}_{\mathrm{SECD}}(M) = \mathsf{b}$ *if and only if* $\vdash M =_{\mathrm{CBV}} \mathsf{b} : B$.

2.2 The Krivine Machine and Call-by-Name

The Krivine machine [23] is the earliest abstract machine that actually captured Church's calculus, despite remaining unpublished folklore until 2007. An essential aspect in this machine is to delay the evaluation of arguments until later. As published, the machine operates on a special version of DeBruijn indices [14], but we present, instead, a machine that operates on variables to make it easier to read. The machine's syntax and transitions are shown in Figure 2.5. Like the SECD machine, the Krivine machine has a notion of call stack; but here it contains only arguments to functions. Whereas SECD has a stack for values and a dump for returning from function calls, the Krivine machine encodes all of this information with its single stack.

$$\begin{array}{rcl} K \in & \textit{Continuation} & ::= \varepsilon \mid \mathbb{V} \cdot K \\ \Sigma \in & \textit{Machine Env.} & ::= \varepsilon \mid \Sigma, \mathbb{V}/x \\ \mathbb{V} \in & \textit{Machine Value} & ::= (\Sigma, M) \\ & \textit{Configuration} & ::= \langle\!\langle \Sigma \parallel M \parallel K \rangle\!\rangle \end{array}$$

(a) Machine Syntax

$$\begin{array}{rcl} \langle\!\langle \Sigma \parallel x \parallel K \rangle\!\rangle & \longmapsto & \langle\!\langle \Sigma' \parallel M \parallel K \rangle\!\rangle \textbf{ where } x[\Sigma] = (\Sigma', M) \\ \langle\!\langle \Sigma \parallel \lambda x.\, M \parallel \mathbb{V} \cdot K \rangle\!\rangle & \longmapsto & \langle\!\langle \Sigma, \mathbb{V}/x \parallel M \parallel K \rangle\!\rangle \\ \langle\!\langle \Sigma \parallel M\, N \parallel K \rangle\!\rangle & \longmapsto & \langle\!\langle \Sigma \parallel M \parallel (\Sigma, N) \cdot K \rangle\!\rangle \end{array}$$

(b) Transitions

Figure 2.5. The Krivine Machine

$$\begin{array}{rcl} (\lambda x.\, M)\, N & =_\beta & M[N/x] \\ \lambda x.\, M\, x & =_\eta & M \end{array}$$

Figure 2.6. The Call-by-Name Calculus

Since the machine does not evaluate the arguments to functions—making it non-strict—and unevaluated arguments may contain free variables, the machine must have closures for everything that is added to its local environment. These closures are constructed *eagerly* when they are pushed on the stack; an observation that will become relevant in later chapters. Unlike the SECD machine, which implements what Plotkin called call-by-value, the Krivine machine implements *call-by-name* by avoiding the evaluation of its argument and proceeding directly to evaluating the left-hand side of the application. The axioms for call-by-name are indeed those given by Church in Figure 2.1.

Definition 2.3 (Krivine Machine Evaluation)**.** $\text{Eval}_{\text{KAM}}(M) = \mathsf{b}$ *where* $\langle\!\langle \varepsilon \parallel M \parallel \varepsilon \rangle\!\rangle \longmapsto^* \langle\!\langle \Sigma \parallel \mathsf{b} \parallel \varepsilon \rangle\!\rangle$.

As an example of execution, consider again the program $(\lambda x.\, \lambda y.\, x)\ 4\ 2$:

$$\begin{aligned}
&\langle\!\langle \Sigma \parallel (\lambda x.\, \lambda y.\, x)\ 4\ 2 \parallel K \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle \Sigma \parallel (\lambda x.\, \lambda y.\, x)\ 4 \parallel (\Sigma, 2) \cdot K \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle \Sigma \parallel \lambda x.\, \lambda y.\, x \parallel (\Sigma, 4) \cdot (\Sigma, 2) \cdot K \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle \Sigma, (\Sigma, 4)/x \parallel \lambda y.\, x \parallel (\Sigma, 2) \cdot K \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle \Sigma, (\Sigma, 4)/x, (\Sigma, 2)/y \parallel x \parallel K \rangle\!\rangle \\
\longmapsto\ &\langle\!\langle \Sigma \parallel 4 \parallel K \rangle\!\rangle
\end{aligned}$$

An important difference between the SECD machine and the Krivine machine, which is evident in their evaluation of the above expression beside how they handle function arguments, is how they handle functions. Notice that the SECD machine first *evaluates* the function and returns it on the stack before *applying* it, whereas the Krivine machine merely *pushes* the argument on the stack and *enters* the left-hand side and will never return a function closure. Thus, we never see the Krivine machine create a closure for the function in this example code, but the SECD machine does even though it is entered immediately.

Proposition 2.2. $\mathrm{Eval}_{\mathrm{KAM}}(M) = \mathsf{b}$ *if and only if* $\vdash M =_{\mathrm{CBN}} \mathsf{b} : B$.

2.3 The Sestoft Machine and Call-by-Need

Since it delays the evaluation of function arguments, the Krivine machine will evaluate those arguments every time that the variable that binds them appears. A more efficient way to perform this non-strict evaluation is by memoization. Memoization is a core idea of computer science. It is found in the study of algorithms as dynamic programming wherein we may make a divide and conquer algorithm significantly faster when sub-parts of the problem are shared. A simple example of this is computing the *n*th

Fibonacci number with a recursive algorithm:

$$\begin{aligned} \text{fib}\ n &= \texttt{if}\ n = 0 \lor n = 1 \\ &\quad\ \texttt{then}\ 1 \\ &\quad\ \texttt{else}\ \text{fib}\ (n-1) + \text{fib}\ (n-2) \end{aligned}$$

Such a program runs in $O(n^2)$ time since we perform two recursive function calls for each n. Using memoization in a programming language like Haskell, on the other hand, we may share the work that is redundant in the recursive calls. That is, fib $(n-2)$ makes use of fib $(n-1)$.

$$\begin{aligned} \text{fibs} &= 1 : 1 : \text{zipWith}\ (+)\ \text{fibs}\ (\text{tail fibs}) \\ \text{fib}\ n &= \text{get}\ n\ \text{fibs} \end{aligned}$$

Now the recursive calls to fib will access the same memory cell that has been memoized and this program runs in $O(n)$ time.

Though there have been other machines that implement this memoized non-strict evaluation, we present the machine of Sestoft [48] in Figure 2.7 because it is the simplest abstract machine that uses closures and closely matches the way such languages are implemented today.[1] First to note about this machine is that it does not operate directly on λ-expressions; instead a preprocessing to administrative normal form (ANF) [18] is done first. Essentially, we give names to all function arguments. The machine configurations are similar to that of the Krivine machine containing a local environment, an expression (in ANF), and a stack. It is extended to a heap that contains closures for any expression. Local environments only point to these heap cells; this restriction is possible because

[1]Another, older approach to these memoized non-strict languages is via graph reduction [54, 22, 12]. Indeed, closures which pair an environment with code is not all that different than considering partial application of super-combinators.

$$\begin{array}{rcl}
\mathrm{ANF}(x) & = & x \\
\mathrm{ANF}(\lambda x.\, M) & = & \lambda x.\, \mathrm{ANF}(M) \\
\mathrm{ANF}(M\, N) & = & \mathtt{let}\ x\ \mathsf{be}\ \mathrm{ANF}(N)\ \mathtt{in}\ \mathrm{ANF}(M)\ x
\end{array}$$

(a) Compilation to A-normal Form

$$\begin{array}{rcl}
\Phi \in & \textit{Heap} & ::= \varepsilon \mid \Phi, l \mapsto (\Sigma, M) \\
\Sigma \in & \textit{Machine Env.} & ::= \varepsilon \mid \Sigma, l/x \\
S \in & \textit{Stack} & ::= l \cdot S \mid \#l \cdot S \\
 & \textit{Machine State} & ::= \langle\!\langle \Phi \parallel \Sigma \parallel M \parallel K \rangle\!\rangle
\end{array}$$

(b) Machine Syntax

$$\begin{array}{rcl}
\langle\!\langle \Phi \parallel \Sigma \parallel \mathsf{b} \parallel \#l \cdot K \rangle\!\rangle & \longmapsto_1 & \langle\!\langle \Phi, l \mapsto (\Sigma, \mathsf{b}) \parallel \Sigma \parallel \mathsf{b} \parallel K \rangle\!\rangle \\
\langle\!\langle (\Phi, x[\Sigma] \mapsto (\Sigma', M))\Phi' \parallel \Sigma \parallel x \parallel K \rangle\!\rangle & \longmapsto_2 & \langle\!\langle \Phi\Phi' \parallel \Sigma' \parallel M \parallel \#l \cdot K \rangle\!\rangle \\
\langle\!\langle \Phi \parallel \Sigma \parallel \lambda x.\, M \parallel \#l \cdot K \rangle\!\rangle & \longmapsto_3 & \langle\!\langle \Phi, l \mapsto (\Sigma, \lambda x.\, M) \parallel \Sigma \parallel \lambda x.\, M \parallel K \rangle\!\rangle \\
\langle\!\langle \Phi \parallel \Sigma \parallel \lambda x.\, M \parallel l \cdot K \rangle\!\rangle & \longmapsto_4 & \langle\!\langle \Phi \parallel \Sigma, l/x \parallel M \parallel K \rangle\!\rangle \\
\langle\!\langle \Phi \parallel \Sigma \parallel M\, x \parallel K \rangle\!\rangle & \longmapsto_5 & \langle\!\langle \Phi \parallel \Sigma \parallel M \parallel x[\Sigma] \cdot K \rangle\!\rangle \\
\langle\!\langle \Phi \parallel \Sigma \parallel \mathtt{let}\ x\ \mathsf{be}\ M\ \mathtt{in}\ N \parallel K \rangle\!\rangle & \longmapsto_6 & \langle\!\langle \Phi, l \mapsto (\Sigma, M) \parallel \Sigma, l/x \parallel N \parallel K \rangle\!\rangle
\end{array}$$

(c) Transitions

Figure 2.7. The Sestoft Machine

$$\begin{array}{rcl}
V, W \in & \textit{Value} & ::= \mathsf{b} \mid x \mid \lambda x.\, M \\
E, F \in & \textit{Evaluation Cxt.} & ::= \square \mid E\, N \mid \mathtt{let}\ x\ \mathsf{be}\ M\ \mathtt{in}\ E \mid \mathtt{let}\ x\ \mathsf{be}\ E\ \mathtt{in}\ F[x]
\end{array}$$

(a) Syntax

$$\begin{array}{rcl}
(\lambda x.\, M)\, N & =_\beta & \mathtt{let}\ x\ \mathsf{be}\ N\ \mathtt{in}\ M \\
\lambda x.\, V\, x & =_\eta & V \\
\mathtt{let}\ x\ \mathsf{be}\ V\ \mathtt{in}\ M & =_x & M[V/x] \\
E[\mathtt{let}\ x\ \mathsf{be}\ M\ \mathtt{in}\ N] & =_\kappa & \mathtt{let}\ x\ \mathsf{be}\ M\ \mathtt{in}\ E[N] \\
\mathtt{let}\ x\ \mathsf{be}\ (\mathtt{let}\ y\ \mathsf{be}\ M\ \mathtt{in}\ N)\ \mathtt{in}\ L & =_\chi & \mathtt{let}\ y\ \mathsf{be}\ M\ \mathtt{in}\ \mathtt{let}\ x\ \mathsf{be}\ N\ \mathtt{in}\ L
\end{array}$$

(b) Axioms

Figure 2.8. A Call-by-Need Calculus

we have already converted expressions to ANF before running them on this machine. Compared to the Krivine machine, there is also another stack frame, denoted $\#l \cdot S$, which is used for memoizing the evaluation of a heap closure.

Like Krivine, when a function is reached, it merely pulls its argument off the stack. The argument is pushed on the stack for an application just like the Krivine machine too; the difference being that we push the heap location of that argument. The argument already has a heap location because of ANF. We see in the last rule that the argument, which was given a name by the ANF transformation, is allocated as a *closure* on the heap. The rules that memoize are 1, 2, and 3. When a variable is demanded (rule 2), its closure in the heap is entered and a memoization frame is pushed. When the evaluation of that closure has reached a memoizable normal form, *e.g.* a λ-expression, the heap label of the memoization frame is added back to the heap with the updated closure.

Definition 2.4 (Sestoft Machine Evaluation)**.** $\mathrm{Eval}_{\mathrm{SM}}(M) = \mathsf{b}$ *where* $\langle\!\langle \varepsilon \parallel \varepsilon \parallel M \parallel \varepsilon \rangle\!\rangle \longmapsto^{*} \langle\!\langle \Gamma \parallel E \parallel \mathsf{b} \parallel \varepsilon \rangle\!\rangle$.

Ariola *et al.* [8, 7] and Maraist *et al.* [28] captured this shared evaluation of function arguments in the call-by-need axiomitization of the λ-calculus. Therein, a computation can be shared by constructing a let-expression that binds it, only forcing the reduction of the bound expression when the evaluation of the variable is required, and substituting its value thereafter.[2] This is apparent in the axioms given in Figure 2.8 wherein the β rule for functions creates a binding and the x rule for let-expressions performs a substitution only when the bound expression is a value. In call-by-need, we can think of values as expressions that are safe to substitute without duplicating work. In addition to those axioms, rules for lifting and reassociating let-expressions are required to expose reducible

[2]Ariola *et al.* [9] show that the let-expression is not necessary since sharing can be captured by not reducing the function application.

expressions. The κ rule is required for lifting expressions out of evaluation contexts and the χ rule is used to reassociate let-expressions. Note that the calculus has the weaker function η laws of call-by-value; otherwise, a value would be changed to a non-value. In call-by-need, blurring this distinction means duplicating work.

Conjecture 2.1. $\mathrm{Eval}_{\mathrm{SM}}(M) = \mathsf{b}$ *if and only if* $\vdash M = \mathsf{b} : B$*.*

The connection between the Sestoft machine and the call-by-need λ-calculus is not found in the literature to our knowledge. In Chapter 7, we will see how to connect a sharing equational theory with a memoizing environment abstract machine.

Remark 2.1 (Push/Enter versus Eval/Apply)**.** *Marlow and Peyton Jones [29] highlight the distinction of two different methods of treating functions in an operational semantics: push/enter and eval/apply. The first will place an argument on the stack and enter the function code; when a function is reached, it merely pull their argument from the stack and place it in the local environment. On the other hand, eval/apply machines will evaluate a function and push it on the stack as a closure value before entering its closure. Marlow and Peyton Jones draw attention to the difference because it matters for fast, curried function calls. We too are interested in the distinction because they differ with respect to closures; push/enter does not need to construct a function closure during application because the function can simply grab its argument from the stack. Of the machines presented above the strict call-by-value machine, i.e. SECD, is eval/apply whereas the two non-strict machines given are push/enter. However, there are examples of strict push/enter machines [26] and non-strict eval/apply machines [29].*

CHAPTER III

CLOSURE CONVERSIONS

This chapter is a revised version of Strictly Capturing Non-strict Closures [52] co-authored with Paul Downen and Zena M. Ariola. Zachary J. Sullivan is the primary author with the guidance of Paul Downen and Zena M. Ariola.

Instead of having a complex runtime system that knows to construct and enter closures at runtime as we see in the machines from the previous chapter, we now look at another approach to handling the nested, lexical structure of languages with passable code: to compile away that structure into a lower-level language. This approach, referred to as *closure conversion*, embeds the delayed code of the source as products and global functions in the target language. A typical target language for such a transformation is C, which has both of these features. Previous work has investigated the efficiency [6, 5, 56, 49] and correctness [34, 3, 36, 40, 38] of closure conversion, and explored its application in more expressive languages with dependent types [11] and mutable references [30]. This line of work, however, mostly applies to just strict languages. Non-strict languages are rarely discussed, if at all. Some work [6, 38] focused on languages in *continuation passing style* (CPS), which subsumes call-by-value and call-by-name semantics, but call-by-need is still left out. A call-by-need CPS exists [37], but it requires a mutable store and is not used in compilers for lazy languages. Rather, these compilers, such as those for Lazy ML [10] and Miranda [41], rely on other methods such as *lambda-lifting*. Haskell's premier optimizing compiler, GHC [43], does use closures, but they are only considered as a small part of low-level code generation.

Extending our understanding of the closure conversion transformation to non-strict languages is not such an easy feat. As we saw in the machines of the previous chapter, non-strict languages create different sorts of closures: every function argument or variable binding is delayed, creating closures that are not needed in a strict language. But just making more closures is not enough: *closures must be strict.* That is to say that in a low-level language without automatic runtime closure support, the compile-time code for creating closures cannot be lazily evaluated because by then it is too late to capture the long-gone static environment. Instead, the environment must be captured *now* when it is available, without inadvertently evaluating anything in the environment. Closure conversion in a non-strict language is a delicate dance between the lazy and the eager.

This chapter examines the canonical approach to this transformation and then describes our work extending it to non-strict evaluation strategies with and without sharing. After reviewing the strict closure conversion (Section 3.1), we show how closure conversion of a non-strict language cannot be embedded into a purely non-strict target language (Section 3.2), but rather strictness is needed in the target language to create closures at the right moment. Similarly, we show thereafter how sharing introduces yet another unintended interaction (Section 3.3): some closures need to be memoized when they are run, but others don't. To eliminate unnecessary details, we present both source and target language's operation semantics as big-step semantics instead of abstract machines. The key difference from the machines is that we do need to consider the continuation or stack. The big-step semantics are described by a judgment of the form $Conf. \Downarrow \mathbb{R}$ where $\mathbb{R}$ is a final result. We still use a delayed substitution in the semantics because it is an essential feature in showing how closure conversion impacts how we may run our program.

3.1 The Canonical Closure Conversion

We first look at the closure conversion that is widely discussed in the literature. Consider the following program:

$$\texttt{let}\ x\ \texttt{be}\ (\texttt{let}\ y\ \texttt{be}\ 2 + 1\ \texttt{in}\ \lambda z.\, y)\ \texttt{in}\ (x\ 3) + (x\ 4) \tag{3.1}$$

Note that when x is called, y is no longer in scope. To remember it when the function is called, we can save its value, *i.e.* 3, in a data structure. In other words, we closure convert the program to:

$$\begin{gathered}\texttt{let}\ x\ \texttt{be}\ (\texttt{let}\ y\ \texttt{be}\ 2 + 1\ \texttt{in}\ \texttt{pack}\ \langle\langle y\rangle, \lambda\langle\langle y\rangle, z\rangle.\, y\rangle)\ \texttt{in}\\ (\texttt{unpack}\ x\ \texttt{as}\ \langle e, f\rangle\ \texttt{in}\ f\ \langle e, 3\rangle) + (\texttt{unpack}\ x\ \texttt{as}\ \langle e, f\rangle\ \texttt{in}\ f\ \langle e, 4\rangle)\end{gathered}$$

Here, the λ-expression $\lambda z.\, y$ in the source is replaced with a data structure containing a representation of the environment and a closed function which accesses that environment. Now, unlike the machine from the previous chapter, the function definition and call site themselves encode in the code the packing and unpacking its local environment to find the binding of y.

In this example, we needed to generate both an unpack expression and then case expression for destructing the generated closure objects. We make use of pattern matching as syntactic sugar for doing a case expression immediately after the binding. So for λ-expressions, we have

$$\lambda\langle x_0, \ldots, x_n\rangle.\, M \overset{\text{def}}{=} \lambda z.\, \texttt{case}\ z\ \texttt{of}\ \{\langle x_0, \ldots, x_n\rangle \longrightarrow M\}.$$

$$\begin{array}{rcl} \Sigma \in & \textit{Machine Env.} & ::= \varepsilon \mid \Sigma, \mathbb{V}/x \\ \mathbb{V}, \mathbb{W} \in & \text{Machine Value} & ::= \mathsf{b} \mid (\Sigma, \lambda x.\, M) \\ \textit{Conf} \in & \textit{Configuration} & ::= \langle\!\langle \Sigma \parallel M \rangle\!\rangle \end{array}$$

(a) Syntax

$$\boxed{\langle\!\langle \Sigma \parallel M \rangle\!\rangle \Downarrow \mathbb{V}}$$

$$\frac{}{\langle\!\langle \Sigma \parallel \mathsf{b} \rangle\!\rangle \Downarrow \mathsf{b}} \qquad \frac{}{\langle\!\langle \Sigma \parallel x \rangle\!\rangle \Downarrow x[\Sigma]} \qquad \frac{}{\langle\!\langle \Sigma \parallel \lambda x.\, M \rangle\!\rangle \Downarrow (\Sigma, \lambda x.\, M)}$$

$$\frac{\langle\!\langle \Sigma \parallel M \rangle\!\rangle \Downarrow (\Sigma', \lambda x.\, L) \quad \langle\!\langle \Sigma \parallel N \rangle\!\rangle \Downarrow \mathbb{W} \quad \langle\!\langle \Sigma', \mathbb{W}/x \parallel L \rangle\!\rangle \Downarrow \mathbb{V}}{\langle\!\langle \Sigma \parallel M\ N \rangle\!\rangle \Downarrow \mathbb{V}}$$

(b) Evaluation Rules

Figure 3.1. Strict Evaluation

And similarly for unpack-expressions. So in our example, `unpack` x `as` $\langle e, f \rangle$ `in` $f\ \langle e, 3 \rangle$ pattern matches on an existential package and then a product. If the pattern match is nested like this, then there will be nested case expressions.

We use existential types in addition to products here to make closure conversion a type-preserving transformation. Without existentially quantifying over the type of the environment (following from Minamide *et al.* [34]), the two following programs, for instance, of type `int` $\to$ `int` would have different types:

$$\lambda x.\, y \qquad\qquad \lambda x.\, x$$

The first would closure convert into a pair with the first component being an empty product and the second would have a unary product in the first component.

3.1.1 Source Language. The syntax of configurations and evaluation rules for strictly evaluating an expression are presented in Figure 3.1. In the syntax, notice that machine values are identical to those of the SECD machine. When evaluating, the λ-expression rule knows how to automatically construct a closure and the application rule

$$\begin{array}{rcl}
\tau, \sigma \in & \mathit{Type} & ::= B \mid \tau \to \sigma \mid \tau_0 \times \cdots \times \tau_n \mid X \mid \exists X.\, \tau \\
\Gamma \in & \mathit{Type\ Env.} & ::= \varepsilon \mid \Gamma, x{:}\tau \\
\Delta \in & \mathit{TypeVar.Env.} & ::= \varepsilon \mid \Delta, X \\
M, N, L \in & \mathit{Expression} & ::= \mathsf{b} \mid x \mid \lambda x.\, M \mid M\ N \\
& & \quad \mid \langle M_0, \ldots, M_n \rangle \mid \mathsf{case}\ M\ \mathsf{of}\ \{\langle x_0, \ldots, x_n \rangle \to N\} \\
& & \quad \mid \mathsf{pack}\ M \mid \mathsf{unpack}\ M\ \mathsf{as}\ x\ \mathsf{in}\ N
\end{array}$$

(a) Syntax

$$\frac{}{\Delta; \Gamma \vdash \mathsf{b} : B}\, B \qquad \frac{x{:}\tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau}\, \mathit{var}$$

$$\frac{\varepsilon, x{:}\tau \vdash M : \sigma}{\Delta; \Gamma \vdash \lambda x.\, M : \tau \to \sigma} \to_I \qquad \frac{\Delta; \Gamma \vdash M : \tau \to \sigma \quad \Delta; \Gamma \vdash N : \tau}{\Delta; \Gamma \vdash M\ N : \sigma} \to_E$$

$$\frac{\Delta; \Gamma \vdash M_0 : \tau_0 \quad \cdots \quad \Delta; \Gamma \vdash M_n : \tau_n}{\Delta; \Gamma \vdash \langle M_0, \ldots, M_n \rangle : \tau_0 \times \cdots \times \tau_n} \times_I$$

$$\frac{\Delta; \Gamma \vdash M : \sigma_0 \times \cdots \times \sigma_n \quad \Delta; \Gamma \vdash, x_0{:}\sigma_0, \ldots, x_n{:}\sigma_n \vdash N : \tau}{\Delta; \Gamma \vdash \mathsf{case}\ M\ \mathsf{of}\ \{\langle x_0, \ldots, x_n \rangle \to N\} : \tau} \times_E$$

$$\frac{\Delta; \Gamma \vdash M : \tau[\sigma/X]}{\Delta; \Gamma \vdash \mathsf{pack}\ M : \exists X.\, \tau} \exists_I \qquad \frac{\Delta; \Gamma \vdash M : \exists X.\, \sigma \quad \Delta, X; \Gamma, x{:}\sigma \vdash N : \tau}{\Delta; \Gamma \vdash \mathsf{unpack}\ M\ \mathsf{as}\ x\ \mathsf{in}\ N : \tau} \exists_E$$

(b) Typing Rules

Figure 3.2. Closure Conversion Target Language

knows how to unpack it, instantiate its local environment, and jump into the body with the value of the actual parameter.

Definition 3.1 (Strict Big-Step Evaluation)**.** $\mathrm{Eval}_{\mathrm{SBS}}(M) = \mathsf{b}$ *where* $\langle\langle \varepsilon \parallel M \rangle\rangle \Downarrow \mathsf{b}$.

This approach to evaluation coincides with the SECD machine when they reach a value. The following is a result by Plotkin [46].

Proposition 3.1. $\mathrm{Eval}_{\mathrm{SECD}}(M) = \mathrm{Eval}_{\mathrm{SBS}}(M)$.

3.1.2 Target Language. Such a transformation requires two features in the target language that we have yet to specify formally: products and existential types. The syntax and typing rules for these are presented in Figure 3.2. We have new types for products and existential types along with the type variables that are introduced by the existential types. Elements of the product type are introduced by $\langle M_0, \ldots, M_n \rangle$ and eliminated by case expressions, which bind their sub-components. Note that single angle brackets $\langle \ldots \rangle$ are used for expressions and double angle brackets $\langle\!\langle \ldots \rangle\!\rangle$ are used for operational semantics. Elements of existential types are introduced by $\mathtt{pack}\ M$ and eliminated by unpack expressions, which bind a pack expression's sub-component. The typing rules have been expanded with Δ which contains the live type variables. There is an important restriction in the $\exists_E$ rule, where we see that the type variable X is available to type check N, but it is not available in the whole unpack expression's type-variable environment; without such a restriction, we could leak the type hidden by the existential.

There is another change in the target language's typing rules that is especially relevant to the *adequacy* of closure conversion: the function type is global in the sense that it only knows about its formal parameter. Before closure conversion, some program of a function type would implicitly carry around its environment in a closure; but after closure conversion, some program of a function type is merely an object that we can jump to at runtime with its formal parameter on the stack. The former requires some runtime support, whereas the latter is easily implementable on a stack machine.

The evaluation rules for the target language are given in Figure 3.3. In the strict source semantics, the set of machine values was not a subset of the surface language because evaluation rules must form and return closures instead of λ-expressions. In contrast, the closed functions of the target language are already values; they can be compiled simply into function pointers. For evaluation, the λ-expression simply returns itself; it does not construct a closure since the function must already be closed except for its formal parameter as we saw in the typing rules. At the call site, the target application

$$
\begin{array}{rcl}
\Sigma \in & \textit{Machine Env.} & ::= \varepsilon \mid \Sigma, \mathbb{V}/x \\
\mathbb{V} \in & \textit{Machine Value} & ::= \mathsf{b} \mid \lambda x.\, M \mid \langle \mathbb{V}_0, \dots, \mathbb{V}_n \rangle \mid \mathsf{pack}\ \mathbb{V} \\
\mathit{Conf} \in & \textit{Configuration} & ::= \langle\!\langle \Sigma \parallel M \rangle\!\rangle
\end{array}
$$

(a) Syntax

$$\boxed{\langle\!\langle \Sigma \parallel M \rangle\!\rangle \Downarrow \mathbb{V}}$$

$$
\frac{}{\langle\!\langle \Sigma \parallel \mathsf{b} \rangle\!\rangle \Downarrow \mathsf{b}}
\qquad
\frac{}{\langle\!\langle \Sigma \parallel x \rangle\!\rangle \Downarrow x[\Sigma]}
\qquad
\frac{}{\langle\!\langle \Sigma \parallel \lambda x.\, M \rangle\!\rangle \Downarrow \lambda x.\, M}
$$

$$
\frac{\langle\!\langle \Sigma \parallel M \rangle\!\rangle \Downarrow \lambda x.\, L \quad \langle\!\langle \Sigma \parallel N \rangle\!\rangle \Downarrow \mathbb{V} \quad \langle\!\langle \varepsilon, \mathbb{V}/x \parallel L \rangle\!\rangle \Downarrow \mathbb{W}}{\langle\!\langle \Sigma \parallel M\ N \rangle\!\rangle \Downarrow \mathbb{W}}
$$

$$
\frac{\langle\!\langle \Sigma \parallel M_0 \rangle\!\rangle \Downarrow \mathbb{V}_0 \quad \cdots \quad \langle\!\langle \Sigma \parallel M_n \rangle\!\rangle \Downarrow \mathbb{V}_n}{\langle\!\langle \Sigma \parallel \langle M_0, \dots, M_n \rangle \rangle\!\rangle \Downarrow \langle \mathbb{V}_0, \dots, \mathbb{V}_n \rangle}
$$

$$
\frac{\langle\!\langle \Sigma \parallel M \rangle\!\rangle \Downarrow \langle \mathbb{V}_0, \dots, \mathbb{V}_n \rangle \quad \langle\!\langle \Sigma, \mathbb{V}_0/x_0, \dots, \mathbb{V}_n/x_n \parallel N \rangle\!\rangle \Downarrow \mathbb{W}}{\langle\!\langle \Sigma \parallel \mathsf{case}\ M\ \mathsf{of}\ \{\langle x_0, \dots, x_n \rangle \to N\} \rangle\!\rangle \Downarrow \mathbb{W}}
$$

$$
\frac{\langle\!\langle \Sigma \parallel M \rangle\!\rangle \Downarrow \mathbb{V}}{\langle\!\langle \Sigma \parallel \mathsf{pack}\ M \rangle\!\rangle \Downarrow \mathsf{pack}\ \mathbb{V}}
\qquad
\frac{\langle\!\langle \Sigma \parallel M \rangle\!\rangle \Downarrow \mathbb{V} \quad \langle\!\langle \Sigma, \mathbb{V}/x \parallel N \rangle\!\rangle \Downarrow \mathbb{W}}{\langle\!\langle \Sigma \parallel \mathsf{unpack}\ M\ \mathsf{as}\ x\ \mathtt{in}\ N \rangle\!\rangle \Downarrow \mathbb{W}}
$$

(b) Evaluation Rules

Figure 3.3. Target Language Evaluation

$$\begin{array}{rcl}
\mathsf{CC}(\mathsf{b}) & = & \mathsf{b} \\
\mathsf{CC}(x) & = & x \\
\mathsf{CC}(\lambda x.\, M) & = & \mathsf{pack}\ \langle\langle y_0, \ldots, y_n\rangle, \lambda\langle\langle y_0, \ldots, y_n\rangle, x\rangle.\mathsf{CC}(M)\rangle \\
\multicolumn{3}{c}{\quad\textbf{where}\ \mathrm{FV}(\lambda x.\, M) = \{y_0, \ldots, y_n\}} \\
\mathsf{CC}(M\ N) & = & \mathsf{unpack}\ \mathsf{CC}(M)\ \mathsf{as}\ \langle e, f\rangle\ \mathsf{in}\ f\ \langle e, \mathsf{CC}(N)\rangle
\end{array}$$

(a) Expression Translation

$$\begin{array}{rcl}
\mathrm{CC}_V(B) & = & B \\
\mathrm{CC}_V(\tau \to \sigma) & = & \exists X.\, X \times (X \times \mathrm{CC}_V(\tau) \to \mathrm{CC}_V(\sigma)) \\
\mathrm{CC}_\Gamma(\varepsilon) & = & \varepsilon \\
\mathrm{CC}_\Gamma(\Gamma, x{:}\tau) & = & \mathrm{CC}_\Gamma(\Gamma), x{:}\mathrm{CC}_V(\tau)
\end{array}$$

(b) Type Translations

Figure 3.4. Canonical Closure Conversion

rule correspondingly expects to find just a λ-expression, and jumps into a function body with only a binding for its parameter in the otherwise empty environment.

Definition 3.2 (Target Big-Step Evaluation)**.** $\mathrm{Eval}_{\mathrm{TBS}}(M) = \mathsf{b}$ *where* $\langle\!\langle \varepsilon \parallel M \rangle\!\rangle \Downarrow \mathsf{b}$.

3.1.3 Transformation. The full transformation from a simply-typed λ-calculus into this target language is shown in Figure 3.4. In the translation of expressions, functions are transformed into packages containing a closed function and a data structure. The generated closed function knows how to access this data structure to re-instantiate the local environment in its body via patter matching. Applications $M\ N$ are transformed—assuming M will evaluate to a closure—into code extracting the environment and function from M, and then calling that function with the environment and argument N. Since functions become data structures, we must translate the type of a program as well. Function types are translated to an existential which hides the type of environment used. Thus, two functions with the same type but different environments will still have the same type after closure conversion.

$$
\begin{array}{rcl}
\mathcal{M}[\![\tau]\!] & = & \{(Conf_s, Conf_t) \mid \forall \mathbb{V}_s.\, Conf_s \Downarrow \mathbb{V}_s \Longrightarrow \exists (\mathbb{V}_s, \mathbb{V}_t) \in \mathcal{V}[\![\tau]\!].\, Conf_t \Downarrow \mathbb{V}_t\} \\
\mathcal{V}[\![B]\!] & = & \{(\mathsf{b}, \mathsf{b}) \mid \mathsf{b} \in B\} \\
\mathcal{V}[\![\tau_0 \rightarrow \tau_1]\!] & = & \{((\Sigma_s, \lambda x.\, M_s), \mathsf{pack}\,\langle\langle \mathbb{W}_0, \ldots, \mathbb{W}_n\rangle, \lambda\langle\langle y_0, \ldots, y_n\rangle, x\rangle.\, M_t\rangle) \\
 & & \mid \forall (\mathbb{V}_s, \mathbb{V}_t) \in \mathcal{V}[\![\tau_0]\!]. \\
 & & \quad (\langle\!\langle \Sigma_s, W_s/x \parallel M_s\rangle\!\rangle, \langle\!\langle \varepsilon, \mathbb{W}_0/y_0, \ldots, \mathbb{W}_n/y_n, \mathbb{V}_t/x \parallel M_t\rangle\!\rangle) \in \mathcal{M}[\![\tau_1]\!]\} \\
\mathcal{E}[\![\Gamma]\!] & = & \{(\Sigma_s, \Sigma_t) \mid \forall (x{:}\tau) \in \Gamma.\, (x[\Sigma_s], x[\Sigma_t]) \in \mathcal{V}[\![\tau]\!]\}
\end{array}
$$

Figure 3.5. Strict Closure Conversion Logical Relations

As a result of using existential types, Minamide *et al.* showed that such a transformation preserves well-typed programs. This can be proved by induction over the typing derivation.

Theorem 3.1 (Type Preservation). *If* $\Gamma \vdash M : \tau$, *then* $\mathrm{CC}_\Gamma(\Gamma) \vdash \mathrm{CC}(M) : \mathrm{CC}_V(\tau)$.

3.1.4 Operational Semantics Preservation. While the theorem above shows that our static checks are preserved, in implementation we care that the dynamic behavior of the program is preserved from the encoding. The common approach to semantic preservation of typed closure conversion, as seen in Minamide *et al.* [34], must step outside of these evaluation theories. They construct a cross-language logical relation over a *substituting* big-step semantics; that is, it relates source expressions to their closure converted form in the target language. Because we are concerned with observing the difference between the closure-constructing runtime system of the source language and the simpler target runtime, we have specified *environment* big-step semantics instead; thus, we must extend their proof approach to such a setting.

We specify a family of relations in Figure 3.5. First, $\mathcal{M}[\![\tau]\!]$ relates closed source and target configurations that behave like source τ expressions. When a source configuration evaluates to a value, then the target must evaluate to a related value. Second, $\mathcal{V}[\![\tau]\!]$ relates machine values that behave like source τ machine values. For base types, this means they are equivalent after conversion. For functions, it must be the case that when given related arguments, that we can construct related configurations that enter the functions with arguments. Third, $\mathcal{E}[\![\Gamma]\!]$ relates environments that behave like source environments Γ. This just means that the environments contain all related values and completely cover Γ.

Lemma 3.1 (Strengthening)**.** *If* $\langle\!\langle \Sigma\, \Sigma' \parallel M \rangle\!\rangle \Downarrow \mathbb{V}$ *and* $\mathrm{FV}(M) \cap \mathrm{Dom}(\Sigma') = \emptyset$*, then* $\langle\!\langle \Sigma \parallel M \rangle\!\rangle \Downarrow \mathbb{V}$ *in the target language.*

Proof. By induction on the derivation of $\langle\!\langle \Sigma\, \Sigma' \parallel M \rangle\!\rangle \Downarrow \mathbb{V}$. □

Lemma 3.2 (Fundamental Lemma)**.** *If* $\Gamma \vdash M_s : \tau$ *and* $(\Sigma_s, \Sigma_t) \in \mathcal{E}[\![\Gamma]\!]$*, then* $(\langle\!\langle \Sigma_s \parallel M_s \rangle\!\rangle, \langle\!\langle \Sigma_t \parallel \mathrm{CC}(M_s) \rangle\!\rangle) \in \mathcal{M}[\![\tau]\!]$.

Proof. By induction on the typing derivation of $\Gamma \vdash M_s : \tau$, for a generic $(\Sigma_s, \Sigma_t) \in \mathcal{E}[\![\Gamma]\!]$:

Case $\Gamma \vdash \mathsf{b} : B$:

So $M_s = \mathsf{b}$, $\mathrm{CC}(\mathsf{b}) = \mathsf{b}$, and we must show that $(\langle\!\langle \Sigma_s \parallel \mathsf{b} \rangle\!\rangle, \langle\!\langle \Sigma_t \parallel \mathsf{b} \rangle\!\rangle) \in \mathcal{M}[\![B]\!]$.

The only evaluations are $\langle\!\langle \Sigma_s \parallel \mathsf{b} \rangle\!\rangle \Downarrow \mathsf{b}$ in the source and $\langle\!\langle \Sigma_t \parallel \mathsf{b} \rangle\!\rangle \Downarrow \mathsf{b}$ in the target.

We have $(\mathsf{b}, \mathsf{b}) \in \mathcal{V}[\![B]\!]$ by definition.

Therefore, $(\langle\!\langle \Sigma_s \parallel \mathsf{b} \rangle\!\rangle, \langle\!\langle \Sigma_t \parallel \mathrm{CC}(\mathsf{b}) \rangle\!\rangle) \in \mathcal{M}[\![B]\!]$.

Case $\Gamma \vdash x : \tau$ because $x{:}\tau \in \Gamma$:

So $M_s = x$, $\mathrm{CC}(x) = x$, and we must show that $(\langle\!\langle \Sigma_s \parallel x \rangle\!\rangle, \langle\!\langle \Sigma_t \parallel x \rangle\!\rangle) \in \mathcal{M}[\![\tau]\!]$.

The only evaluations are $\langle\!\langle \Sigma_s \parallel x \rangle\!\rangle \Downarrow x[\Sigma_s]$ in the source and $\langle\!\langle \Sigma_t \parallel x \rangle\!\rangle \Downarrow x[\Sigma_t]$ in the target.

From the assumptions $(\Sigma_s, \Sigma_t) \in \mathcal{E}[\![\Gamma]\!]$ and $x{:}\tau \in \Gamma$, we know $(x[\Sigma_s], x[\Sigma_t]) \in \mathcal{V}[\![\tau]\!]$ by definition of $\mathcal{E}[\![\Gamma]\!]$.

Therefore, $(\langle\langle\Sigma_s \parallel x\rangle\rangle, \langle\Sigma_t \parallel x\rangle) \in \mathcal{M}[\![\tau]\!]$ by the definition.

Case $\Gamma \vdash \lambda x.N_s : \tau_1 \to \tau_2$ because $\Gamma, x{:}\tau_1 \vdash N_s : \tau_2$:

So $\tau = \tau_1 \to \tau_2$, $M_s = \lambda x.N_s$, and

$$\mathrm{CC}(\lambda x.N_s) = \mathsf{pack}\ \langle\langle y_0, \dots, y_n\rangle, \lambda\langle\langle y_0, \dots, y_n\rangle, x\rangle.\, \mathrm{CC}(N_s)\rangle$$

where $\mathrm{FV}(\lambda x.\, N_s) = \{y_0, \dots, y_n\}$. We must show that $(\langle\langle\Sigma_s \parallel \lambda x.N_s\rangle\rangle, \langle\langle\Sigma_t \parallel \mathrm{CC}(\lambda x.N_s)\rangle\rangle) \in \mathcal{M}[\![\tau_1 \to \tau_2]\!]$.

The unique evaluations are $\langle\langle\Sigma_s \parallel \lambda x.N_s\rangle\rangle \Downarrow (\Sigma_s, \lambda x.N_s)$ in the source and $\langle\langle\Sigma_t \parallel \mathrm{CC}(\lambda x.N_s)\rangle\rangle \Downarrow \mathsf{pack}\ \langle\langle y_0[\Sigma_t], \dots, y_n[\Sigma_t]\rangle, \lambda\langle\langle y_0, \dots, y_n\rangle, x\rangle.\, \mathrm{CC}(N_s)\rangle$ in the target.

Proving $((\Sigma_s, \lambda x.N_s), \mathsf{pack}\ \langle\langle y_0[\Sigma_t], \dots, y_n[\Sigma_t]\rangle, \lambda\langle\langle y_0, \dots, y_n\rangle, x\rangle.\, \mathrm{CC}(N_s)\rangle) \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$ still needs to be shown. Thus, suppose an arbitrary $(\mathbb{W}_s, \mathbb{W}_t) \in \mathcal{V}[\![\tau_1]\!]$:

Note that $((\Sigma_s, \mathbb{W}_s/x), (\Sigma_t, \mathbb{W}_t/x)) \in \mathcal{E}[\![\Gamma, x{:}\tau_1]\!]$ by definition of $\mathcal{E}$ and the assumption $(\Sigma_s, \Sigma_t) \in \mathcal{E}[\![\Gamma]\!]$.

From the inductive hypothesis on $\Gamma, x{:}\tau_1 \vdash N_s : \tau_2$, we know $(\langle\langle\Sigma_s, \mathbb{W}_s/x \parallel N_s\rangle\rangle, \langle\langle\Sigma_t, \mathbb{W}_t/x \parallel \mathrm{CC}(N_s)\rangle\rangle) \in \mathcal{M}[\![\tau_2]\!]$.

Assuming $\langle\langle\Sigma_s, \mathbb{W}_s/x \parallel N_s\rangle\rangle \Downarrow \mathbb{V}_s$ in the source, there must be a $(\mathbb{V}_s, \mathbb{V}_t) \in \mathcal{V}[\![\tau_2]\!]$ such that $\langle\langle\Sigma_t, \mathbb{W}_t/x \parallel \mathrm{CC}(N_s)\rangle\rangle \Downarrow \mathbb{V}_t$ in the target by the definition of $\mathcal{M}[\![\tau_2]\!]$.

Expanding, $\langle\langle\varepsilon, y_0[\Sigma_t]/y_0, \dots, y_n[\Sigma_t]/y_n, \mathbb{W}_t/x \parallel \mathrm{CC}(N_s)\rangle\rangle \Downarrow \mathbb{V}_t$ in the target as well by strengthening (Lemma 3.1) the evaluation $\langle\langle\Sigma_t, \mathbb{W}_t/x \parallel \mathrm{CC}(N_s)\rangle\rangle \Downarrow \mathbb{V}_t$.

Case $\Gamma \vdash N_s\ O_s : \tau$ because $\Gamma \vdash N_s : \tau' \to \tau$ and $\Gamma \vdash O_s : \tau'$:

So $M_s = N_s\ O_s$, $\mathrm{CC}(N_s\ O_s) = \mathsf{unpack}\ \mathrm{CC}(N_s)\ \mathsf{as}\ \langle e, f\rangle\ \mathsf{in}\ f\ \langle e, \mathrm{CC}(O_s)\rangle$. We must show that $(\langle\!\langle \Sigma_s \parallel N_s\ O_s\rangle\!\rangle, \langle\!\langle \Sigma_t \parallel \mathrm{CC}(N_s\ O_s)\rangle\!\rangle) \in \mathcal{M}[\![\tau]\!]$. Thus, we suppose that $\langle\!\langle \Sigma_s \parallel N_s\ O_s\rangle\!\rangle \Downarrow \mathbb{V}_s$:

The conclusion of that derivation must be an instance of the application evaluation from inversion, which gives us the following evaluation derivations in the source:

1. $\langle\!\langle \Sigma_s \parallel N_s\rangle\!\rangle \Downarrow (\Sigma_{1s}, \lambda x.L_s)$,
2. $\langle\!\langle \Sigma_s \parallel O_s\rangle\!\rangle \Downarrow \mathbb{W}_s$, and
3. $\langle\!\langle \Sigma_{1s}, W_s/x \parallel L_s\rangle\!\rangle \Downarrow \mathbb{V}_s$.

From the first inductive hypothesis, we know $(\langle\!\langle \Sigma_s \parallel N_s\rangle\!\rangle, \langle\!\langle \Sigma_t \parallel \mathrm{CC}(N_s)\rangle\!\rangle) \in \mathcal{M}[\![\tau' \to \tau]\!]$. It follows by definition of $\mathcal{M}$ that there is some $((\Sigma_{1s}, \lambda x.L_s), \mathsf{pack}\ \langle\langle \mathbb{W}_0', \ldots, \mathbb{W}_n'\rangle, \lambda\langle\langle y_0, \ldots, y_n\rangle, x\rangle.\, L_t\rangle) \in \mathcal{V}[\![\tau' \to \tau]\!]$ such that $\langle\!\langle \Sigma_t \parallel \mathrm{CC}(N_s)\rangle\!\rangle \Downarrow \mathsf{pack}\ \langle\langle \mathbb{W}_0', \ldots, \mathbb{W}_n'\rangle, \lambda\langle\langle y_0, \ldots, y_n\rangle, x\rangle.\, L_t\rangle$ in the target.

From the second inductive hypothesis, we know $(\langle\!\langle \Sigma_s \parallel O_s\rangle\!\rangle, \langle\!\langle \Sigma_t \parallel \mathrm{CC}(O_s)\rangle\!\rangle) \in \mathcal{M}[\![\tau']\!]$. Likewise, it follows that there is a $(\mathbb{W}_s, \mathbb{W}_t) \in \mathcal{V}[\![\tau']\!]$ such that $\langle\!\langle \Sigma_t \parallel \mathrm{CC}(O_s)\rangle\!\rangle \Downarrow \mathbb{W}_t$ in the target.

We also have $(\langle\!\langle \Sigma_s, \mathbb{W}_t/x \parallel L_s\rangle\!\rangle, \langle\!\langle \varepsilon, \mathbb{W}_0'/y, \ldots, \mathbb{W}_n'/y, \mathbb{W}_t/x \parallel L_t\rangle\!\rangle) \in \mathcal{M}[\![\tau]\!]$, from $((\Sigma_{1s}, \lambda x.L_s), \mathsf{pack}\ \langle\langle \mathbb{W}_0', \ldots, \mathbb{W}_n'\rangle, \lambda\langle\langle y_0, \ldots, y_n\rangle, x\rangle.\, L_t\rangle) \in \mathcal{V}[\![\tau' \to \tau]\!]$. It follows that there is a $(\mathbb{V}_s, \mathbb{V}_t) \in \mathcal{V}[\![\tau]\!]$ such that $\langle\!\langle \varepsilon, \mathbb{W}_0'/y, \ldots, \mathbb{W}_n'/y, \mathbb{W}_t/x \parallel L_t\rangle\!\rangle \Downarrow \mathbb{V}_t$ in the target.

Expanding, we get that $\langle\!\langle \Sigma_t \parallel \mathsf{unpack}\ \mathrm{CC}(N_s)\ \mathsf{as}\ \langle e, f \rangle\ \mathsf{in}\ f\ \langle e, \mathrm{CC}(O_s)\rangle \rangle\!\rangle \Downarrow \mathbb{V}_t$ as well by applications of the evaluation rules for unpack, case, and application.

Therefore, $(\langle\!\langle \Sigma_s \parallel N_s\ O_s \rangle\!\rangle, \langle\!\langle \Sigma_t \parallel \mathrm{CC}(N_s\ O_s) \rangle\!\rangle)$ is in the relation $\mathcal{M}[\![\tau]\!]$ by definition.

□

Corollary 3.1 (Evaluation Preservation). *If* $\Gamma \vdash M : \tau$ *and* $\mathrm{Eval}_{\mathrm{SBS}}(M) = \mathsf{b}$*, then* $\mathrm{Eval}_{\mathrm{TBS}}(\mathrm{CC}(M)) = \mathsf{b}$.

3.2 Non-strict Closure Conversions

Though not emphasized by the work, the closure conversion above is presented for strict languages alone [34, 36, 3, 38, 39]. The functions of a strict source language, wherein values coincide with normal forms, are transformed into in a strict target language. Can we do the same thing for non-strict languages? That is, can we convert a non-strict source language to a non-strict target language that lacks automatic closure management at run-time?

To answer these questions, we first need to know how non-strict data types are evaluated, since closures will be constructed with them when following the common approach. In strict languages, data are evaluated before they are considered values capable of substitution; in contrast, non-strict data are not evaluated until forced by their context, *i.e.* until they are pattern matched by a case expression. For example, a non-strict existential package has the following semantics, based on delayed evaluation rules for data in lazy languages, *e.g.* Launchbury's natural semantics extended with constructors

[25]:

$$\frac{}{\langle \Sigma \parallel \mathsf{pack}\ M \rangle \Downarrow (\Sigma, \mathsf{pack}\ M)}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow (\Sigma', \mathsf{pack}\ L) \quad \langle \Sigma, x \mapsto (\Sigma', L) \parallel N \rangle \Downarrow \mathbb{R}}{\langle \Sigma \parallel \mathsf{unpack}\ M\ \mathsf{as}\ x\ \mathtt{in}\ N \rangle \Downarrow \mathbb{R}}$$

To avoid evaluating inside of the data constructor until pattern matching, a non-strict evaluator must return a closure to capture the environment needed to evaluate it later. But this gets us nowhere! The point of closure conversion is to eliminate the need for automatic closure management at runtime; yet, when trying to eliminate automatic closure management, we introduced a new type ...that requires automatic closure management at runtime.

Our goal is to simulate these non-strict rules above in the text of the program, so the instructions for capturing and restoring the environment are in the compile-time code, not the runtime system. The root of the problem for non-strict closure conversion, then, is that before pack returns, it needs to look up the current definitions of its free variables in scope, so that these bindings can actually be captured in the environment value it contains. In other words, pack must be strict—to some degree—in its argument. But we also must be careful to not introduce too much strictness. In a non-strict evaluation of example (3.1),

$$\mathtt{let}\ x\ \mathsf{be}\ (\mathtt{let}\ y\ \mathsf{be}\ 2 + 1\ \mathtt{in}\ \lambda z.\, y)\ \mathtt{in}\ (x\ 3) + (x\ 4)$$

we must not evaluate the expression $2 + 1$ bound to y when the closure is formed; rather, computation of y itself must still be delayed until its value is forced. Thankfully, this complication, too, is solved by closure conversion. In general, bound, delayed computations, like $\mathtt{let}\ y\ \mathsf{be}\ 2 + 1\ \mathtt{in} \dots$ might also refer to other free variables, so bound expressions must be closure converted like functions were for strict closure conversion. As a consequence, delayed computations bound by let- and λ-expressions will also be

$$\begin{array}{rcl}
\Sigma \in & \textit{Machine Env.} & ::= \varepsilon \mid \Sigma, \mathbb{V}/x \\
\mathbb{V}, \mathbb{W} \in & \textit{Machine Value} & ::= (\Sigma, M) \\
\mathbb{R} \in & \textit{Result} & ::= \mathsf{b} \mid (\Sigma, \lambda x.\, M) \\
\textit{Conf} \in & \textit{Configuration} & ::= \langle\!\langle \Sigma \parallel M \rangle\!\rangle
\end{array}$$

(a) Syntax

$$\boxed{\langle\!\langle \Sigma \parallel M \rangle\!\rangle \Downarrow \mathbb{R}}$$

$$\frac{}{\langle\!\langle \Sigma \parallel \mathsf{b} \rangle\!\rangle \Downarrow \mathsf{b}} \qquad \frac{x[\Sigma] = (\Sigma', M) \quad \langle\!\langle \Sigma' \parallel M \rangle\!\rangle \Downarrow \mathbb{R}}{\langle\!\langle \Sigma \parallel x \rangle\!\rangle \Downarrow \mathbb{R}} \qquad \frac{}{\langle\!\langle \Sigma \parallel \lambda x.\, M \rangle\!\rangle \Downarrow (\Sigma, \lambda x.\, M)}$$

$$\frac{\langle\!\langle \Sigma \parallel M \rangle\!\rangle \Downarrow (\Sigma', \lambda x.\, L) \quad \langle\!\langle \Sigma', (\Sigma, N)/x \parallel L \rangle\!\rangle \Downarrow \mathbb{R}}{\langle\!\langle \Sigma \parallel M\, N \rangle\!\rangle \Downarrow \mathbb{R}}$$

(b) Evaluation Rules

Figure 3.6. Non-strict Evaluation

converted to values—in the sense of strict evaluation—ensuring that they are not evaluated too early.

In brief, a non-strict closure conversion must transform λ-expressions, application arguments, and let-bound expressions into *strictly-constructed* packages of their free variables and a closed function. Applying such a transformation to our example program would produce the following output (to keep the example simple, we did not construct closures for x, 3, and 4):

$$\begin{array}{l}
\mathtt{let}\ x\ \mathsf{be}\ (\mathtt{let}\ y\ \mathsf{be}\ \mathsf{pack}\ \langle\langle\rangle, \lambda\langle\rangle.\, 2 + 1\rangle\ \mathtt{in} \\
\qquad\qquad\qquad \mathsf{pack}\ \langle\langle y\rangle, \lambda\langle\langle y\rangle, z\rangle.\, \mathsf{unpack}\ y\ \mathsf{as}\ \langle e, f\rangle\ \mathtt{in}\ f\, e\rangle)\ \mathtt{in} \\
\quad (\mathsf{unpack}\ x\ \mathsf{as}\ \langle e, f\rangle\ \mathtt{in}\ f\ \langle e, 3\rangle) + (\mathsf{unpack}\ x\ \mathsf{as}\ \langle e, f\rangle\ \mathtt{in}\ f\ \langle e, 4\rangle)
\end{array}$$

In addition to the function closures needed in strict closure conversion, we have added a closure construction for the binding of y.

3.2.1 Source Language. The syntax and evaluation rules for a non-strict evaluation are given in Figure 3.6. In a non-strict evaluator, all the values in the environment are thunk closures. In contrast to the strict setting, the values stored in the environment are different from the results of evaluation; thus, we see a separate notion of *results* to which configurations evaluate. Examining the evaluation rules, the variable rule unpacks and evaluates the thunk that finds in the environment. The application rule must handle two different types of closures: it must unpack the function closure returned from evaluating the left-hand side and it must construct a thunk closure for the formal parameter following closely the closure conversion for this case. This behavior for applications, and the fact that function closures are results, conveys that this machine is more closely related to an eval/apply style abstract machine than the Krivine machine. That being said, such evaluation still has the same termination behavior as the Krivine machine.

Definition 3.3 (Non-Strict Big-Step Evaluation)**.** $\mathrm{Eval}_{\mathrm{NSBS}}(M) = \mathsf{b}$ *where* $\langle\!\langle \varepsilon \parallel M \rangle\!\rangle \Downarrow \mathsf{b}$.

The following proposition is similar to that for the SECD machine that Plotkin [46]. We have not proved it here.

Conjecture 3.1. $\mathrm{Eval}_{\mathrm{KAM}}(M) = \mathrm{Eval}_{\mathrm{NSBS}}(M)$.

3.2.2 Target Language. Since every function and let-expression in the target language will be strict and they are generated from those of the source language, the target language of the non-strict closure conversion is indeed that of strict closure conversion.

3.2.3 Transformation. Non-strict closure conversion is presented in Figure 3.7. Variables are converted into code for unpacking thunk closures. Applications are converted into code that turns arguments into thunk closures while unpacking the function closure and applying it. The non-strict transformation is careful to distinguish

$$
\begin{array}{rcl}
\mathrm{CC}(\mathsf{b}) & = & \mathsf{b} \\
\mathrm{CC}(x) & = & \mathsf{unpack}\ x\ \mathsf{as}\ \langle e, f\rangle\ \mathsf{in}\ f\ e \\
\mathrm{CC}(\lambda x.\, M) & = & \mathsf{pack}\ \langle\langle y_0, \ldots, y_n\rangle, \lambda\langle\langle y_0, \ldots, y_n\rangle, x\rangle.\, \mathrm{CC}(M)\rangle \\
\multicolumn{3}{c}{\textbf{where } \mathrm{FV}(M) - \{x\} = \{y_0, \ldots, y_n\}} \\
\mathrm{CC}(M\, N) & = & \mathsf{unpack}\ \mathrm{CC}(M)\ \mathsf{as}\ \langle e, f\rangle\ \mathsf{in} \\
 & & \quad f\ \langle e, \mathsf{pack}\ \langle\langle y_0, \ldots, y_n\rangle, \lambda\langle y_0, \ldots, y_n\rangle.\, \mathrm{CC}(N)\rangle\rangle \\
\multicolumn{3}{c}{\textbf{where } \mathrm{FV}(N) = \{y_0, \ldots, y_n\}}
\end{array}
$$

(a) Expression Translation

$$
\begin{array}{rcl}
\mathrm{CC}_R(B) & = & B \\
\mathrm{CC}_R(\tau \to \sigma) & = & \exists X.\, X \times (X \times \mathrm{CC}_V(\tau) \to \mathrm{CC}_R(\sigma)) \\
\mathrm{CC}_V(\tau) & = & \exists X.\, X \times (X \to \mathrm{CC}_R(\tau)) \\
\mathrm{CC}_\Gamma(\varepsilon) & = & \varepsilon \\
\mathrm{CC}_\Gamma(\Gamma, x{:}\tau) & = & \mathrm{CC}_\Gamma(\Gamma), x{:}\mathrm{CC}_V(\tau)
\end{array}
$$

(b) Type Translations

Figure 3.7. A Non-strict Closure Conversion

thunk closures from function closures. Whereas the former contains a closed function from some environment, the latter contains a closed function that takes a pair of some environment *and* a formal parameter.

Extending the strict type translation to a non-strict language requires a different translation for values and results. Intuitively, the three type translations can be thought of as a translation of expressions that we intend to evaluate to results (CC_R) versus placing them directly in the environment (CC_V), along with translation of the environment needed for evaluating an expression (CC_Γ). The result type translation of a function has changed from the strict closure conversion to reflect that it now accepts only thunks as arguments.

Like strict closure conversion, the non-strict transformation preserves typing derivations.

Theorem 3.2 (Type Preservation)**.** *If* $\Gamma \vdash M : \tau$, *then* $\mathrm{CC}_\Gamma(\Gamma) \vdash \mathrm{CC}(M) : \mathrm{CC}_R(\tau)$.

$$\begin{array}{rcl}
\mathcal{M}[\![\tau]\!] & = & \{(\mathit{Conf}_s, \mathit{Conf}_t) \mid \forall \mathbb{R}_s.\, \mathit{Conf}_s \Downarrow \mathbb{R}_s \implies \exists (\mathbb{R}_s, \mathbb{V}_t) \in \mathcal{R}[\![\tau]\!].\, \mathit{Conf}_t \Downarrow \mathbb{V}_t\} \\
\mathcal{V}[\![\tau]\!] & = & \{((\Sigma_s, M_s), \mathsf{pack}\,\langle\langle \mathbb{V}_0, \ldots, \mathbb{V}_n\rangle, \lambda\langle x_0, \ldots, x_n\rangle.\, M_t\rangle) \\
 & & \mid (\langle\!\langle \Sigma_s \parallel M_s\rangle\!\rangle, \langle\!\langle \varepsilon, \mathbb{V}_0/x_0, \ldots, \mathbb{V}_n/x_n \parallel M_t\rangle\!\rangle) \in \mathcal{M}[\![\tau]\!]\} \\
\mathcal{R}[\![B]\!] & = & \{(\mathsf{b}, \mathsf{b}) \mid \mathsf{b} \in B\} \\
\mathcal{R}[\![\tau_0 \to \tau_1]\!] & = & \{((\Sigma_s, \lambda x.\, M_s), \mathsf{pack}\,\langle\langle \mathbb{W}_0, \ldots, \mathbb{W}_n\rangle, \lambda\langle\langle y_0, \ldots, y_n\rangle, x\rangle.\, M_t\rangle) \\
 & & \mid \forall (\mathbb{V}_s, \mathbb{V}_t) \in \mathcal{V}[\![\tau_0]\!]. \\
 & & \quad (\langle\!\langle \Sigma_s, W_s/x \parallel M_s\rangle\!\rangle, \langle\!\langle \varepsilon, \mathbb{W}_0/y_0, \ldots, \mathbb{W}_n/y_n, \mathbb{V}_t/x \parallel M_t\rangle\!\rangle) \in \mathcal{M}[\![\tau_1]\!]\} \\
\mathcal{E}[\![\Gamma]\!] & = & \{(\Sigma_s, \Sigma_t) \mid \forall (x{:}\tau) \in \Gamma.\, (\Sigma_s(x), \Sigma_t(x)) \in \mathcal{V}[\![\tau]\!]\}
\end{array}$$

Figure 3.8. Non-strict Closure Conversion Logical Relations

Such a non-strict closure conversion is dependent on the operational semantics of the target language. For our work in PEPM [52], we used a natural semantics, which must evaluate programs to results. Therefore, the case of functions must capture a closure as a final result. However, if we use the Krivine machine, then the evaluation context is maintained while evaluating the function of an application. Therein, we can get away with only constructing closures for function arguments since the function's environment will still be available! We will see in later chapters how to avoid this redundant closure in machines with push/enter style function evaluation.

3.2.4 Operational Semantics Preservation. Using a similar delayed evaluation logical relation technique to the strict case, we may prove that evaluation is preserved by this conversion. Repeating the theme of distinguishing values and results, the family of logical relations from strict closure-conversion can be modified to work for the non-strict transformation with similar modifications to those of the semantics and type translations. Thus, the non-strict family of relations in Figure 3.8 includes a separate relation for values and results.

The $\mathcal{V}$ relation from the strict closure conversion has become the result relation $\mathcal{R}$ for non-strict closure conversion; it relates source results to target machine values. The relation for values, $\mathcal{V}$, is wholly new. A source value, which is a thunk closure, is related to a target package when they form related configurations by unpacking and applying their respective enclosed environments. As before, the fundamental lemma of this logical relation implies correct evaluation.

Lemma 3.3 (Fundamental Lemma). *If* $\Gamma \vdash M_s : \tau$ *and* $(\Sigma_s, \Sigma_t) \in \mathcal{E}[\![\Gamma]\!]$, *then* $(\langle\Sigma_s \parallel M_s\rangle, \langle\Sigma_t \parallel \mathrm{CC}(M_s)\rangle) \in \mathcal{M}[\![\tau]\!]$.

Proof. By induction on the typing derivation of $\Gamma \vdash M_s : \tau$, for a generic $(\Sigma_s, \Sigma_t) \in \mathcal{E}[\![\Gamma]\!]$. The cases for base types and function introduction are analogous to Lemma 3.2. The remaining two cases are:

Case $\Gamma \vdash x : \tau$ because $x{:}\tau \in \Gamma$.

So $M_s = x$, $\mathrm{CC}(x) = \mathtt{unpack}\ x\ \mathtt{as}\ \langle e, f\rangle\ \mathtt{in}\ f\ e$, and we must show that $(\langle\!\langle\Sigma_s \parallel x\rangle\!\rangle, \langle\!\langle\Sigma_t \parallel \mathrm{CC}(x)\rangle\!\rangle) \in \mathcal{M}[\![\tau]\!]$.

From the assumptions $(\Sigma_s, \Sigma_t) \in \mathcal{E}[\![\Gamma]\!]$ and $x{:}\tau \in \Gamma$, we know $(x[\Sigma_s], x[\Sigma_t]) \in \mathcal{V}[\![\tau]\!]$ by definition of $\mathcal{E}[\![\Gamma]\!]$. Furthermore, the definition of $\mathcal{V}$ yields that $x[\Sigma_s] = (\Sigma_s', M_s)$ and $x[\Sigma_t] = \mathtt{pack}\ \langle\!\langle\langle\mathbb{V}_0, \dots, \mathbb{V}_n\rangle, \lambda\langle\!\langle x_0, \dots, x_n\rangle\!\rangle.\, M_t\rangle\!\rangle$ such that $(\langle\!\langle\Sigma_s' \parallel M_s\rangle\!\rangle, \langle\!\langle\varepsilon, \mathbb{V}_0/x_0, \dots, \mathbb{V}_n/x_n \parallel M_t\rangle\!\rangle) \in \mathcal{M}[\![\tau]\!]$.

Assume the source evaluation $\langle\!\langle\Sigma_s \parallel x\rangle\!\rangle \Downarrow \mathbb{R}_s$. By inversion on this derivation, $\langle\!\langle\Sigma_s' \parallel M_s\rangle\!\rangle \Downarrow \mathbb{R}_s$.

We are guaranteed related results by $\mathcal{M}[\![\tau]\!]$. That is, $\langle\!\langle\varepsilon, \mathbb{V}_0/x_0, \dots, \mathbb{V}_n/x_n \parallel M_t\rangle\!\rangle \Downarrow \mathbb{V}_t$ such that $(\mathbb{R}_s, \mathbb{V}_t) \in \mathcal{R}[\![\tau]\!]$.

Expanding, we have $\langle\!\langle\Sigma_t \parallel \mathrm{CC}(x)\rangle\!\rangle \Downarrow \mathbb{V}_t$ from the above evaluation combined with unpack, case, and application rules of the target.

Therefore, $(\langle\!\langle\Sigma_s \parallel x\rangle\!\rangle, \langle\!\langle\Sigma_t \parallel \mathrm{CC}(x)\rangle\!\rangle) \in \mathcal{M}[\![\tau]\!]$ follows by definition.

Case $\Gamma \vdash N_s\ O_s : \tau$ because $\Gamma \vdash N_s : \tau' \to \tau$ and $\Gamma \vdash O_s : \tau'$.

So $M_s = N_s\ O_s$, $\mathrm{FV}(O_s) = \{y_0, \ldots, y_n\}$ and $\mathrm{CC}(N_s\ O_s)$ is

$$\mathsf{unpack}\ \mathrm{CC}(N_s)\ \mathsf{as}\ \langle e, f\rangle\ \mathsf{in}\ f\ \langle e, \mathsf{pack}\ \langle\langle y_0, \ldots, y_n\rangle, \lambda\langle y_0, \ldots, y_n\rangle.\, \mathrm{CC}(O_s)\rangle\rangle.$$

and we must show that $\mathcal{M}[\![\tau]\!]$ contains $(\langle\!\langle \Sigma_s \parallel N_s\ O_s\rangle\!\rangle, \langle\!\langle \Sigma_t \parallel \mathrm{CC}(N_s\ O_s)\rangle\!\rangle)$

Suppose that $\langle\!\langle \Sigma_s \parallel N_s\ O_s\rangle\!\rangle \Downarrow \mathbb{R}_s$ in the source. The conclusion of that derivation must be an instance of the application rule by inversion, which gives us

1. $\langle\!\langle \Sigma_s \parallel N_s\rangle\!\rangle \Downarrow (\Sigma_s', \lambda x.L_s)$ and
2. $\langle\!\langle \Sigma_s', (\Sigma_s, O_s)/x \parallel L_s\rangle\!\rangle \Downarrow \mathbb{R}_s$.

From the first inductive hypothesis, we know $(\langle\!\langle \Sigma_s \parallel N_s\rangle\!\rangle, \langle\!\langle \Sigma_t \parallel \mathrm{CC}(N_s)\rangle\!\rangle) \in \mathcal{M}[\![\tau' \to \tau]\!]$. Thus, $((\Sigma_s', \lambda x.L_s), \mathsf{pack}\ \langle\langle \mathbb{W}_0, \ldots, \mathbb{W}_n\rangle, \lambda\langle\langle z_0, \ldots, z_n\rangle, x\rangle.\, L_t\rangle) \in \mathcal{R}[\![\tau' \to \tau]\!]$ where $\langle\!\langle \Sigma_t \parallel \mathrm{CC}(N_s)\rangle\!\rangle \Downarrow \mathsf{pack}\langle\langle \mathbb{W}_0, \ldots, \mathbb{W}_m\rangle, \lambda\langle\langle z_0, \ldots, z_m\rangle, x\rangle.\, L_t\rangle$ by definition of $\mathcal{M}$.

By the pack, product, and variable evaluation rules, we know that $\langle\!\langle \Sigma_t \parallel \mathsf{pack}\ \langle\langle y_0, \ldots, y_n\rangle, \lambda\langle y_0, \ldots, y_n\rangle.\, \mathrm{CC}(O_s)\rangle\rangle\!\rangle$ evaluates to the target machine value $\mathsf{pack}\ \langle\langle y_0[\Sigma_t], \ldots, y_n[\Sigma_t]\rangle, \lambda\langle y_0, \ldots, y_n\rangle.\, \mathrm{CC}(O_s)\rangle$.

We show $((\Sigma_s, O_s), \mathsf{pack}\ \langle\langle y_0[\Sigma_t], \ldots, y_n[\Sigma_t]\rangle, \lambda\langle y_0, \ldots, y_n\rangle.\, \mathrm{CC}(O_s)\rangle) \in \mathcal{V}[\![\tau']\!]$ by showing that $(\langle\!\langle \Sigma \parallel O_s\rangle\!\rangle, \langle\!\langle \varepsilon, y_0[\Sigma_t]/y_0, \ldots, y_n[\Sigma_t]/y_n \parallel \mathrm{CC}(O_s)\rangle\!\rangle) \in \mathcal{M}[\![\tau']\!]$. Thus, suppose $\langle\!\langle \Sigma_s \parallel O_s\rangle\!\rangle \Downarrow \mathbb{S}_s$:

From the second inductive hypothesis, we know $(\langle\!\langle \Sigma_s \parallel O_s\rangle\!\rangle, \langle\!\langle \Sigma_t \parallel \mathrm{CC}(O_s)\rangle\!\rangle) \in \mathcal{M}[\![\tau']\!]$. And thus, there is some $(\mathbb{S}_s, \mathbb{X}_t) \in \mathcal{R}[\![\tau']\!]$ such that $\langle \Sigma_t \parallel \mathrm{CC}(O_s)\rangle \Downarrow \mathbb{X}_t$.

We can conclude that $\langle\!\langle \varepsilon, y_0[\Sigma]/y_0, \ldots, y_n[\Sigma]/y_n \parallel \mathrm{CC}(O_s)\rangle\!\rangle \Downarrow \mathbb{X}_t$ if $\langle \Sigma_t \parallel \mathrm{CC}(O_s)\rangle \Downarrow \mathbb{X}_t$ by strengthening (Lemma 3.1).

By the property of $\mathcal{R}[\![\tau' \to \tau]\!]$ with the related values above in $\mathcal{V}[\![\tau']\!]$, we know that $(\langle\!\langle \Sigma_s', (\Sigma_s, O_s)/x \parallel L_s \rangle\!\rangle, \langle\!\langle \varepsilon, \mathbb{W}_0/z_0, \ldots, \mathbb{W}_m/z_m, \mathbb{Y}/x \parallel L_t \rangle\!\rangle) \in \mathcal{M}[\![\tau]\!]$ where $\mathbb{Y}$ is the package $\mathsf{pack}\ \langle\langle y_0[\Sigma_t], \ldots, y_n[\Sigma_t]\rangle, \lambda\langle y_0, \ldots, y_n\rangle.\, \mathrm{CC}(O_s)\rangle$. From this property and (2) above, we know that there is some $(\mathbb{R}_s, \mathbb{V}_t) \in \mathcal{R}[\![\tau]\!]$ such that $\langle\!\langle \varepsilon, \mathbb{W}_0/z_0, \ldots, \mathbb{W}_m/z_m, \mathbb{Y}/x \parallel L_t \rangle\!\rangle) \Downarrow \mathbb{V}_t$.

By the unpack, case, and application evaluation rules, we know that $\langle\!\langle \Sigma_t \parallel \mathsf{unpack}\, \mathrm{CC}(N_s)\ \mathsf{as}\ \langle e, f\rangle\ \mathsf{in}\, f\ \langle e, \mathsf{pack}\ \langle\langle y_0, \ldots, y_n\rangle, \lambda\langle y_0, \ldots, y_n\rangle.\, \mathrm{CC}(O_s)\rangle\rangle\rangle\!\rangle \Downarrow \mathbb{V}_t$.

Therefore, $(\langle\!\langle \Sigma_s \parallel N_s\ O_s \rangle\!\rangle, \langle\!\langle \Sigma_t \parallel \mathrm{CC}(N_s\ O_s)\rangle\!\rangle)$ is in $\mathcal{M}[\![\tau]\!]$ by definition.

□

Corollary 3.2 (Evaluation Preservation). *If* $\Gamma \vdash M : \tau$ *and* $\mathrm{Eval}_{\mathrm{NSBS}}(M) = \mathsf{b}$*, then* $\mathrm{Eval}_{\mathrm{TBS}}(\mathrm{CC}(M)) = \mathsf{b}$.

3.3 Sharing Closure Conversion

When we applied strict closure conversion to our non-strict language, we found that closures need to be strict and that we need to close over arguments of functions. This forced us to use a strict target language even when closure converting non-strict programs. Running an analogous experiment, consider the non-strict, sharing evaluation of the resulting program from non-strict closure conversion (again, avoiding the closures necessary for x, 3, and 4) of the program in (3.1).

$$
\begin{array}{l}
\mathsf{let}\ x\ \mathsf{be}\ (\mathsf{let}\ y\ \mathsf{be}\ \mathsf{pack}\ \langle\langle\rangle, \lambda\langle\rangle.\, 2+1\rangle\ \mathsf{in} \\
\qquad\qquad \mathsf{pack}\ \langle\langle y\rangle, \lambda\langle\langle y\rangle, z\rangle.\, \mathsf{unpack}\ y\ \mathsf{as}\ \langle e, f\rangle\ \mathsf{in}\ f e\rangle)\ \mathsf{in} \\
\quad (\mathsf{unpack}\ x\ \mathsf{as}\ \langle e, f\rangle\ \mathsf{in}\ f\ \langle e, 3\rangle) + (\mathsf{unpack}\ x\ \mathsf{as}\ \langle e, f\rangle\ \mathsf{in}\ f\ \langle e, 4\rangle)
\end{array}
$$

Since the transformation replaces every binding with a strict binding, we are left with a program with only strict bindings. Thus, the two evaluations of the thunk bound to y are no longer shared. A proper lazy closure conversion should share computations; that is, thunk closures must be evaluated at most one time.

An obvious solution is to add a restricted form of mutable references to the target language and replace thunks after their evaluation. Instead of closure converting a function argument to a thunk, it will be converted into a pointer to a heap-allocated, tagged thunk. We will use the following shorthand for tagged heap storage:

$$\mathsf{store}\ M \overset{\text{def}}{=} \mathsf{new}\ (\mathsf{inr}\ M)$$

At the thunk's call site, *i.e.* a variable lookup in the source, we will generate code that checks the tag to determine whether to simply return a value or to evaluate the thunk and update the pointer. This, we capture in a memoization macro:

$$\begin{aligned}\mathsf{memo}\ x \overset{\text{def}}{=} \mathsf{case}\ !x\ \mathsf{of}\\ &\mathsf{inl}\ v \longrightarrow v\\ &\mathsf{inr}\ t \longrightarrow \mathsf{unpack}\ t\ \mathsf{as}\ (y, z)\ \mathsf{in}\\ &\qquad \mathsf{let}\ v = z\ y\ \mathsf{in}\\ &\qquad\quad \mathsf{let}\ \_ = (x := \mathsf{inl}\ v)\ \mathsf{in}\ v\end{aligned}$$

In our example, applying these ideas to the thunk created for y yields the following target program:

$$\begin{aligned}&\mathsf{let}\ x\ \mathsf{be}\ (\mathsf{let}\ y\ \mathsf{be}\ \mathsf{store}\ (\mathsf{pack}\ \langle\langle\rangle, \lambda\langle\rangle.\, 2+1\rangle)\ \mathsf{in}\\ &\qquad\qquad \mathsf{pack}\ \langle\langle y\rangle, \lambda\langle\langle y\rangle, z\rangle.\, \mathsf{memo}\ y\rangle)\ \mathsf{in}\\ &\quad (\mathsf{unpack}\ x\ \mathsf{as}\ \langle e, f\rangle\ \mathsf{in}\ f\ \langle e, 3\rangle) + (\mathsf{unpack}\ x\ \mathsf{as}\ \langle e, f\rangle\ \mathsf{in}\ f\ \langle e, 4\rangle)\end{aligned}$$

$$\begin{array}{rcl} \Phi \in & \mathit{Heap} & ::= \varepsilon \mid \Phi, l \mapsto (\Sigma, M) \mid \Phi, l \mapsto \mathbb{A} \\ \Sigma \in & \mathit{Machine\ Env.} & ::= \varepsilon \mid \Sigma, l/x \\ \mathbb{A} \in & \mathit{Answer} & ::= \mathsf{b} \mid (\Sigma, \lambda x.\, M) \\ \mathbb{R} \in & \mathit{Result} & ::= (\Phi, \mathbb{A}) \\ & \mathit{Configuration} & ::= \langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle \end{array}$$

(a) Syntax

$$\frac{l \notin \mathrm{Dom}(\Phi) \quad \Phi'(l) = P \quad \forall l' \in (\mathrm{Dom}(\Phi') - \{l\}).\, \Phi(l') = \Phi'(l')}{\mathtt{alloc}(\Phi, P) = (l, \Phi')}$$

$$\frac{l \in \mathrm{Dom}(\Phi) \quad \Phi'(l) = P \quad \forall l' \in (\mathrm{Dom}(\Phi') - \{l\}).\, \Phi(l') = \Phi'(l')}{\mathtt{update}(\Phi, l, P) = \Phi'}$$

(b) Heap Semantics

$$\boxed{\langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle \Downarrow \mathbb{R}}$$

$$\frac{}{\langle\!\langle \Phi \parallel \Sigma \parallel \mathsf{b} \rangle\!\rangle \Downarrow (\Phi, \mathsf{b})} \qquad \frac{\Phi(x[\Sigma]) = \mathbb{A}}{\langle\!\langle \Phi \parallel \Sigma \parallel x \rangle\!\rangle \Downarrow (\Phi, \mathbb{A})}$$

$$\frac{\Phi(x[\Sigma]) = (\Sigma', M) \quad \langle\!\langle \Phi \parallel \Sigma' \parallel M \rangle\!\rangle \Downarrow (\Phi', \mathbb{A}) \quad \mathtt{update}(\Phi', x[\Sigma], \mathbb{A}) = \Phi''}{\langle\!\langle \Phi \parallel \Sigma \parallel x \rangle\!\rangle \Downarrow (\Phi'', \mathbb{A})}$$

$$\frac{}{\langle\!\langle \Phi \parallel \Sigma \parallel \lambda x.\, M \rangle\!\rangle \Downarrow (\Phi, (\Sigma, \lambda x.\, M))}$$

$$\frac{\langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle \Downarrow (\Phi', (\Sigma', \lambda x.\, L)) \quad \mathtt{alloc}(\Phi', (\Sigma, N)) = (l, \Phi'') \quad \langle\!\langle \Phi'' \parallel \Sigma', l/x \parallel L \rangle\!\rangle \Downarrow \mathbb{R}}{\langle\!\langle \Phi \parallel \Sigma \parallel M\ N \rangle\!\rangle \Downarrow \mathbb{R}}$$

(c) Evaluation Rules

Figure 3.9. Lazy Evaluation

We arrive at a lazy closure conversion in modifying the non-strict transformation by inserting these thunk mutating macros at the locations where source variable bindings are introduced (*i.e.* let-bound expressions and function arguments) and eliminated (*i.e.* variable lookup).

3.3.1 Source Language. As the source operational semantics for our lazy closure conversion, the syntax and evaluation rules for a sharing non-strict evaluator are given in Figure 3.9. The major difference in this semantics versus the ones that we presented for strict and non-strict evaluation is the addition of the heap. As a result, we must now distinguish results, values, and *answers*. Answers are the set of normalized expressions and the result of evaluation now contains both an updated heap and an answer. Configurations include a heap and an environment. Whereas heaps hold thunks and answers at specified locations, environments are only a mapping from variables to locations into the heap.

We model heaps as objects for which we only know how to allocate, update, and lookup. Since our heaps remain abstract, our heap semantics specifies only the properties that allocation and update operations must satisfy. Allocation requires that we are allocating a fresh variable, the new heap correctly returns the expression being allocated, and everything else in the heap remains unchanged. Update requires that the variable is already in the heap, that the new heap correctly returns the value, and that everything else in the heap remains unchanged.

Just like the other two big-step evaluators, the rule for λ-expressions must construct a function closure. Like the non-strict language, the application rule constructs a thunk closure, but here it is added to the heap and a pointer to it is passed in the environment. The differential treatment between closure types is more obvious in a shared non-strict evaluator: function closures are returned from evaluations, whereas thunk closures are passed as pointers to the heap where they can be updated.

As we noticed with the non-strict big-step semantics, such a semantics acts more like an eval/apply style machine since the left-hand side of an application is evaluated to a function closure which is then applied to its argument. For the memoization aspect of sharing to work, there is no way of avoiding function closures if we wanted a more push/enter style big-step semantics. This is because it is precisely this normal form, with its environment, that we must memoize.

$$
\begin{array}{rll}
\tau, \sigma \in & \textit{Type} & ::= \cdots \mid \tau + \sigma \mid \mathsf{ref}\ \tau \\
M, N, L \in & \textit{Expression} & ::= \cdots \mid \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid \mathsf{case}\ M\ \mathsf{of}\ \{\mathsf{inl}\ x \to N; \mathsf{inr}\ y \to L\} \\
 & & \quad \mid \mathsf{new}\ M \mid\ !M \mid M := N
\end{array}
$$

(a) Additional Syntax

$$
\frac{\Delta; \Gamma \vdash M : \tau}{\Delta; \Gamma \vdash \mathsf{inl}\ M : \tau + \sigma}\ +_{I1} \qquad \frac{\Delta; \Gamma \vdash M : \sigma}{\Delta; \Gamma \vdash \mathsf{inr}\ M : \tau + \sigma}\ +_{I2}
$$

$$
\frac{\Delta; \Gamma \vdash M : \sigma_l + \sigma_r \quad \Delta; \Gamma, x{:}\sigma_l \vdash N : \tau \quad \Delta; \Gamma, x{:}\sigma_r \vdash L : \tau}{\Delta; \Gamma \vdash \mathsf{case}\ M\ \mathsf{of}\ \{\mathsf{inl}\ x \to N; \mathsf{inr}\ x \to L\} : \tau}\ +_E
$$

$$
\frac{\Delta; \Gamma \vdash M : \tau}{\Delta; \Gamma \vdash \mathsf{new}\ M : \mathsf{ref}\ \tau}\ \mathsf{ref}_I \quad \frac{\Delta; \Gamma \vdash M : \mathsf{ref}\ \tau}{\Delta; \Gamma \vdash\ !M : \tau}\ \mathsf{ref}_E \quad \frac{\Delta; \Gamma \vdash M : \mathsf{ref}\ \tau \quad \Delta; \Gamma \vdash N : \tau}{\Delta; \Gamma \vdash M := N : 1}\ \textit{Mut}
$$

(b) Additional Typing Rules

Figure 3.10. Target Language extended with Sums and Mutation

Definition 3.4 (Shared Non-Strict Big-Step Evaluation)**.** $\mathrm{Eval}_{\mathrm{SNSBS}}(M) = \mathsf{b}$ *where* $\langle\langle \varepsilon \parallel \varepsilon \parallel M \rangle\rangle \Downarrow \mathsf{b}$.

Like with the other machines and big-step semantics, we believe this evaluation to coincide with the Sestoft machine.

Conjecture 3.2. $\mathrm{Eval}_{\mathrm{SM}}(M) = \mathrm{Eval}_{\mathrm{SNSBS}}(M)$.

3.3.2 Target Language. In order to handle the added problem of updating thunks, the strict target language for lazy closure conversion extends the previous target language with sums and mutable references. The additional typing rules for this target language are given in Figure 3.10. Heap manipulation in this language is via reference types, à la Standard ML [33]. This can be seen in the *Mut* rule wherein a reference to an integer, for instance, is a different type `ref int` that can only be updated with another integer. Assignment expressions will return a value of the empty product type (i.e. 1) which we denote by $\langle\rangle$.

$$\begin{array}{rcl}
\Phi \in & \textit{Heap} & ::= \varepsilon \mid \Phi, l \mapsto \mathbb{V} \\
\Sigma \in & \textit{Machine Env.} & ::= \varepsilon \mid \Sigma, \mathbb{V}/x \\
\mathbb{V} \in & \textit{Value} & ::= \mathsf{b} \mid \lambda x.\, M \mid \langle \mathbb{V}_0, \ldots, \mathbb{V}_n \rangle \mid \mathsf{pack}\ \mathbb{V} \mid \mathtt{inl}\ \mathbb{V} \mid \mathtt{inr}\ \mathbb{V} \mid l \\
\mathbb{R} \in & \textit{Result} & ::= (\Phi, \mathbb{V}) \\
 & \textit{Configuration} & ::= \langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle
\end{array}$$

(a) Syntax

$$\frac{}{\langle\!\langle \Phi \parallel \Sigma \parallel \mathsf{b} \rangle\!\rangle \Downarrow (\Phi, \mathsf{b})} \qquad \frac{}{\langle\!\langle \Phi \parallel \Sigma \parallel x \rangle\!\rangle \Downarrow (\Phi, x[\Sigma])} \qquad \frac{}{\langle\!\langle \Phi \parallel \Sigma \parallel \lambda x.\, M \rangle\!\rangle \Downarrow (\Phi, \lambda x.\, M)}$$

$$\frac{\langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle \Downarrow (\Phi', \lambda x.\, L) \quad \langle\!\langle \Phi' \parallel \Sigma \parallel N \rangle\!\rangle \Downarrow (\Phi'', \mathbb{V}) \quad \langle\!\langle \Phi'' \parallel \varepsilon, \mathbb{V}/x \parallel L \rangle\!\rangle \Downarrow \mathbb{R}}{\langle\!\langle \Phi \parallel \Sigma \parallel M\ N \rangle\!\rangle \Downarrow \mathbb{R}}$$

$$\frac{\langle\!\langle \Phi_0 \parallel \Sigma \parallel M_0 \rangle\!\rangle \Downarrow (\Phi_1, \mathbb{V}_0) \quad \cdots \quad \langle\!\langle \Phi_n \parallel \Sigma \parallel M_n \rangle\!\rangle \Downarrow (\Phi_{n+1}, \mathbb{V}_n)}{\langle\!\langle \Phi_0 \parallel \Sigma \parallel \langle M_0, \ldots, M_n \rangle \rangle\!\rangle \Downarrow (\Phi_{n+1}, \langle \mathbb{V}_0, \ldots, \mathbb{V}_n \rangle)}$$

$$\frac{\langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle \Downarrow (\Phi', \langle \mathbb{V}_0, \ldots, \mathbb{V}_n \rangle) \quad \langle\!\langle \Phi \parallel \Sigma, \mathbb{V}_0/x_0, \ldots, \mathbb{V}_n/x_n \parallel N \rangle\!\rangle \Downarrow \mathbb{R}}{\langle\!\langle \Phi \parallel \Sigma \parallel \mathsf{case}\ M\ \mathsf{of}\ \{\langle x_0, \ldots, x_n \rangle \rightarrow N\} \rangle\!\rangle \Downarrow \mathbb{R}}$$

$$\frac{\langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle \Downarrow (\Phi', \mathbb{V})}{\langle\!\langle \Phi \parallel \Sigma \parallel \mathsf{pack}\ M \rangle\!\rangle \Downarrow (\Phi', \mathsf{pack}\ \mathbb{V})}$$

$$\frac{\langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle \Downarrow (\Phi', \mathsf{pack}\ \mathbb{V}) \quad \langle\!\langle \Phi' \parallel \Sigma, \mathbb{V}/x \parallel N \rangle\!\rangle \Downarrow \mathbb{R}}{\langle\!\langle \Phi \parallel \Sigma \parallel \mathsf{unpack}\ M\ \mathsf{as}\ x\ \mathsf{in}\ N \rangle\!\rangle \Downarrow \mathbb{R}}$$

$$\frac{\langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle \Downarrow (\Phi', \mathbb{V})}{\langle\!\langle \Phi \parallel \Sigma \parallel \mathtt{inl}\ M \rangle\!\rangle \Downarrow (\Phi', \mathtt{inl}\ \mathbb{V})} \qquad \frac{\langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle \Downarrow (\Phi', \mathbb{V})}{\langle\!\langle \Phi \parallel \Sigma \parallel \mathtt{inr}\ M \rangle\!\rangle \Downarrow (\Phi', \mathtt{inr}\ \mathbb{V})}$$

$$\frac{\langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle \Downarrow (\Phi', \mathtt{inl}\ \mathbb{V}) \quad \langle\!\langle \Phi' \parallel \Sigma, \mathbb{V}/x \parallel N \rangle\!\rangle \Downarrow \mathbb{R}}{\langle\!\langle \Phi \parallel \Sigma \parallel \mathsf{case}\ M\ \mathsf{of}\ \{\mathtt{inl}\ x \rightarrow N; \mathtt{inr}\ y \rightarrow L\} \rangle\!\rangle \Downarrow \mathbb{R}}$$

$$\frac{\langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle \Downarrow (\Phi', \mathtt{inr}\ \mathbb{V}) \quad \langle\!\langle \Phi' \parallel \Sigma, \mathbb{V}/x \parallel L \rangle\!\rangle \Downarrow \mathbb{R}}{\langle\!\langle \Phi \parallel \Sigma \parallel \mathsf{case}\ M\ \mathsf{of}\ \{\mathtt{inl}\ x \rightarrow N; \mathtt{inr}\ y \rightarrow L\} \rangle\!\rangle \Downarrow \mathbb{R}}$$

$$\frac{\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow (\Phi', \mathbb{V}) \quad \mathtt{alloc}(\Phi', l, \mathbb{V}) = (l, \Phi'')}{\langle\!\langle \Phi \parallel \Sigma \parallel \mathsf{new}\ M \rangle\!\rangle \Downarrow (\Phi'', l)} \qquad \frac{\langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle \Downarrow (\Phi', l) \quad \Phi'(l) = \mathbb{V}}{\langle\!\langle \Phi \parallel \Sigma \parallel !M \rangle\!\rangle \Downarrow (\Phi', \mathbb{V})}$$

$$\frac{\langle\!\langle \Phi \parallel \Sigma \parallel M \rangle\!\rangle \Downarrow (\Phi', l) \quad \langle\!\langle \Phi' \parallel \Sigma \parallel N \rangle\!\rangle \Downarrow (\Phi'', \mathbb{V}) \quad \mathtt{update}(\Phi'', l, \mathbb{V}) = \Phi'''}{\langle\!\langle \Phi \parallel \Sigma \parallel M := N \rangle\!\rangle \Downarrow (\Phi''', \langle\rangle)}$$

(b) Evaluation Rules

Figure 3.11. Mutable Target Language Evaluation

$$
\begin{array}{rcl}
\mathrm{CC}(\mathtt{b}) & = & \mathtt{b} \\
\mathrm{CC}(x) & = & \mathtt{case}\ !x\ \mathtt{of} \\
 & & \quad \mathtt{inl}\ v \longrightarrow v \\
 & & \quad \mathtt{inr}\ t \longrightarrow \mathtt{unpack}\ t\ \mathtt{as}\ \langle e, f\rangle\ \mathtt{in}\ x := \mathtt{inl}\ (f\ e); !x \\
\mathrm{CC}(\lambda x.\, M) & = & \mathtt{pack}\ \langle\langle y_0, \ldots, y_n\rangle, \lambda\langle\langle y_0, \ldots, y_n\rangle, x\rangle.\, \mathrm{CC}(M)\rangle \\
\multicolumn{3}{l}{\qquad\quad \textbf{where}\ \mathrm{FV}(\lambda x.\, M) = \{y_0, \ldots, y_n\}} \\
\mathrm{CC}(M\ N) & = & \mathtt{let}\ x\ \mathtt{be}\ (\mathtt{new}\ (\mathtt{inr}\ \langle\langle y_0, \ldots, y_n\rangle, \lambda\langle y_0, \ldots, y_n\rangle.\, \mathrm{CC}(N)\rangle))\ \mathtt{in} \\
 & & \quad \mathtt{unpack}\ \mathrm{CC}(M)\ \mathtt{as}\ \langle e, f\rangle\ \mathtt{in}\ f\ \langle e, x\rangle \\
\multicolumn{3}{l}{\qquad\quad \textbf{where}\ \mathrm{FV}(N) = \{y_0, \ldots, y_n\}}
\end{array}
$$

(a) Expression Translation

$$
\begin{array}{rcl}
\mathrm{CC}_A(B) & = & B \\
\mathrm{CC}_A(\tau \longrightarrow \sigma) & = & \exists X.\, X \times (X \times \mathrm{CC}_V(\tau) \longrightarrow \mathrm{CC}_A(\sigma)) \\
\mathrm{CC}_V(\tau) & = & \mathtt{ref}\ (\mathrm{CC}_A(\tau) + (\exists X.\, X \times (X \longrightarrow \mathrm{CC}_A(\tau)))) \\
\mathrm{CC}_\Gamma(\varepsilon) & = & \varepsilon \\
\mathrm{CC}_\Gamma(\Gamma, x{:}\tau) & = & \mathrm{CC}_\Gamma(\Gamma), x{:}\mathrm{CC}_V(\tau)
\end{array}
$$

(b) Type Translations

Figure 3.12. Memoizing Non-strict Closure Conversion

The operational semantics is given in Figure 3.11. Unlike the syntax and typing rules, which were a direct extension of the old target language, all of the evaluation rules differ because they must pass the heap around explicitly. For instance, the product evaluation rule is limited to left-to-right evaluation of its components. In the non-mutable target language, this order was irrelevant.

The new mutable references rules make use of the same heap interface as our lazy semantics. The rule for a `new` expression evaluates its argument to a value, places that value in the heap, and returns its location in memory. The dereference rule evaluates its argument to a location and returns the value at that location. Finally, the mutation rule will evaluate the left-hand side to get the location where the right-hand side's value will go. After the update, a mutation will return the empty product $\langle\rangle$.

3.3.3 Transformation. The lazy closure conversion is found in Figure 3.12. Our lazy source language's different variable lookup rules are encoded in a single case expression: either the heap location contains a thunk or a normal form. The application case is the same as in the non-strict closure conversion case, but the thunk is tagged as a thunk with `inr` and it is placed in the heap with `new` instead of in the local environment. The type translation reflects the heap allocation with a `ref` type and the thunk tagging with a sum type in the $\mathrm{CC}_V(\tau)$ translation. Interestingly, the cases for already normalized expressions (*i.e.* constants of base types and manifest functions) are the same for strict, non-strict, and lazy closure conversion.

Theorem 3.3 (Type Preservation). *If* $\Gamma \vdash M : \tau$, *then* $\mathrm{CC}_\Gamma(\Gamma) \vdash \mathrm{CC}(M) : \mathrm{CC}_R(\tau)$.

3.3.4 Operational Semantics Preservation. As part of the PEPM submission [52] that this chapter is based on, the delayed-substitution logical relation for a sharing non-strict language was left undone. By PPDP [53], we have discovered how to expand on the non-strict relation to make them work here. Chapter 7 presents such an extension.

CHAPTER IV

ABSTRACT CLOSURES

This chapter contains published and unpublished co-authored material. It is a revision of the work Closure Conversion in Little Pieces [53] co-authored with Paul Downen and Zena M. Ariola. Zachary J. Sullivan is the primary author under the guidance of Paul Downen and Zena M. Ariola.

Abstract machines expressed closures as part of the runtime system of the language and closure conversion expressed closures by a transformation between a high-level source language and a different language which does not have the ability to pass nested, unevaluated code. However, it would be really convenient for the transformation to be expressed in the *same* language. Specifically, we would like the following to hold:

Theorem 4.1. *If* $\Gamma \vdash M : \tau$, *then* $\Gamma \vdash M = \mathrm{CC}(M) : \tau$.

That is, we want an expression M to be axiomatically equal, via the typical β and η axioms, to its closure converted form $\mathrm{CC}(M)$. Such a language enables not only the simple definition of the transformation but for optimized versions, *e.g.* those that share environments, to be implemented locally and incrementally. Indeed, it is the compatibility and transitivity of equational theories that allow these closures optimizations to compose with themselves and other optimizations within a compiler. There are also benefits to reasoning about correctness. Whereas in previous work [34, 52] different closure conversion techniques correspond to different cross-language logical relations, closure transformations encoded via the axioms of a compiler intermediate language (IL) are proven correct merely by the soundness of these axioms. That is, for a sound equational theory, axiomatic equality implies contextual equivalence.

Working within a single equational theory as the main focus of optimizations has a history of success in compilers [44, 51, 5, 39]. A key idea therein is that a core IL is modified repeatedly, in a series of passes, by a set of small, local transformations. Some global transformations, *e.g.* strictness analysis, are still necessary but are less modular. Inlining, constant folding, and common sub-expression elimination are all examples of local transformations. Local transformations may be built from smaller ones, *e.g.* common sub-expression elimination is a case of β expansion when we give a name to a repeated sub-program. Such an approach has even been successful for optimization problems that are typically handled in lower-level code, like join points [31] and unboxed types [42], by extending the IL to capture some essential properties of these concepts. Once included, the low-level parts may be optimized with existing optimizations; for instance, redundant unboxing operations can be eliminated via common sub-expression elimination.

To date, closures have been excluded from this local approach because closure conversion [34, 36, 3, 38, 39, 52] as a data representation of code does not make this goal easy. For example, consider using the strict closure conversion from the previous chapter (Figure 3.4):

$$\mathsf{CC}(\lambda x.\, y + z) = \mathsf{pack}\ \langle \langle y, z \rangle, \lambda \langle e, x \rangle.\, \mathsf{case}\ e\ \mathsf{of}\ \{\langle y, z \rangle \to y + z\} \rangle$$

If we want Theorem 4.1 to be true, then we immediately run into a problem since the type has changed from a function to a existential pair. Therefore, to say that something is $\beta\eta$ equal to its closure converted form is false because we cannot apply an existential pair as we can a function. Of course, we could remedy this problem by changing all of the calling contexts of a function, but then we have a global transformation thereby losing the local reasoning that enables optimization in little pieces. Our solution to this problem is *not* to consider closure conversion a data representation for functions, instead we build on the work on *abstract* closures [21, 34, 11]. These are special objects for which we may give bespoke semantics, distinct from a usual function's semantics. An abstract closure object "knows" about the relationship between the environment and code parts of a closure; that is, the environment part of the closure will be substituted at the same time as the formal parameter.

This chapter presents the following contributions: defines closure conversion not as a *global* cross-language transformation but in terms of *little pieces* that correspond to provable equalities. Thereby, closure conversion is correct by construction and we elevate closure conversion to be on par with other optimizations. In other words, it is done within the IL itself; and thus, closure conversion optimizations can be expressed by standard IL transformations.

4.1 Why Abstract Closures

If our goal is to promote reasoning about closures from a low-level runtime or code generation phase of compilation to an equational theory suitable for optimizations, then the canonical closure conversion presents more problems than just being a global transformation.

Instead of a transformation between a different source and target language, in some works [39, 3] the two languages are the same. Alas, this still does not work well with equational theories. If it did, then it should be the case that the transformation preserves equality, by the transitivity of the theory:

$$M = N \qquad \textit{implies} \qquad \mathrm{CC}(M) = \mathrm{CC}(N)$$

For example, let us attempt to preserve the call-by-value β axiom:

$$\mathrm{CC}((\lambda x.\, M)\ V) = \mathrm{CC}(M[V/x])$$

For any closure conversion, preserving this law is hard since the transformation changes the number of free variables in the function body; therefore, it does not commute with substitution:

$$\mathrm{CC}(M)[\mathrm{CC}(V)/x] \neq \mathrm{CC}(M[V/x])$$

For an example of why this is not true, let M be $\lambda z.\, x$ and we will need to prove the following:

$$\mathrm{CC}(\lambda z.\, x)[\mathrm{CC}(V)/x] \overset{?}{=} \mathrm{CC}((\lambda z.\, x)[V/x])$$

The variable x will be part of the closure on the left but not on the right; and therefore, the following equation does not hold (assume V is closed):

$$\begin{array}{rcl}(\mathsf{pack}\ \langle\langle x\rangle, \lambda\langle\langle x\rangle, z\rangle.\, x\rangle)[\mathrm{CC}(V)/x] & = & \mathsf{pack}\ \langle\langle \mathrm{CC}(V)\rangle, \lambda\langle\langle x\rangle, z\rangle.\, x\rangle \\ & \neq & \mathsf{pack}\ \langle\langle\rangle, \lambda\langle\langle\rangle, z\rangle.\, \mathrm{CC}(V)\rangle \\ \langle x\rangle & \neq & \langle\rangle\end{array}$$

A problem also arises in preserving η-laws where we need to show that $\mathrm{CC}(\lambda x.\, V\ x) = \mathrm{CC}(V)$. If V is a λ-expression, then we may prove this with β; but if V is a variable, say z, then we get stuck, as shown below:

$$\mathsf{pack}\ \langle\langle z\rangle, \lambda\langle\langle z\rangle, x\rangle.\, \mathsf{unpack}\ z\ \mathsf{as}\ \langle e, f\rangle\ \mathsf{in} f\ \langle e, x\rangle\rangle \overset{?}{=} z$$

Both of these failures are because closure conversion creates products that have a distinct relation between the first and second components: the second will always expect the first as an argument when applied. However, products and functions do *not* have this property in general. This is why we step outside of the equational theory and use

$$\varsigma \in \quad \mathit{Environment} \quad ::= \varepsilon \mid \varsigma, V/x$$

$$\frac{}{\Gamma \vdash \varepsilon : \varepsilon}\ \Gamma_{I_B} \qquad \frac{\Gamma \vdash \varsigma : \Gamma' \quad \Gamma \vdash V : \tau}{\Gamma \vdash (\varsigma, V/x) : (\Gamma', x{:}\tau)}\ \Gamma_{I_I}$$

Figure 4.1. Syntactic Environments

logical relations, which capture the lost information, to prove the correctness of closure conversion.

Non-strict closure conversions add more troubles for a local transformation. First, the target language of non-strict closures is a strict language, so it does not work by necessity. Moreover, the sharing non-strict closure conversion requires mutation in the target language. If source and target were the same here, then our ability to reason would be greatly reduced since mutable languages provide much weaker guarantees than the λ-calculi that we have seen so far.

Working directly in the syntax, abstract closures [21, 34, 11] solve all of the above problems, though previous work has not given them an equational theory nor specified them for non-strict calculi. They have the same type as the non-closure versions of functions and consume the same applicative contexts. Thus, they solve the global transformation problem and enable type-preserving reductions. Additionally, consuming the same contexts allows their η laws to be preserved. The environment and the formal parameter of a function are substituted at the same time when entering the code part. Thus, they solve the disconnection of environment and code that happens with the product encoding thereby enabling us to equate closures that capture different environments. Finally, we will be able to specify non-strict semantics for them thereby allowing us to remain within a non-strict language while doing closure conversion.

$$
\begin{array}{rcl}
M, N, L \in & \mathit{Expression} & ::= V \mid M\ N \\
V, W \in & \mathit{Value} & ::= \mathsf{b} \mid x \mid \{\varsigma, \lambda x.\, M\}
\end{array}
$$

(a) Syntax

$$\lambda x.\, M = \{\varepsilon, \lambda x.\, M\}$$

(b) Syntactic Sugar

$$\frac{\Gamma \vdash \varsigma : \Gamma' \quad \Gamma\Gamma', x{:}\tau \vdash M : \sigma}{\Gamma \vdash \{\varsigma, \lambda x.\, M\} : \tau \rightarrow \sigma} \rightarrow_I$$

(c) New Introduction Rule

$$
\begin{array}{rcl}
\{\varsigma, \lambda x.\, M\}\ V & =_\beta & M[\varsigma, V/x] \\
\lambda x.\, V\ x & =_\eta & V
\end{array}
$$

(d) Axioms

Figure 4.2. Call-by-Value with Closures

4.2 Closures for Different Evaluation Strategies

Adding abstract closures to our different calculi begins with a notion of delayed substitution, which we call *environments*. Figure 4.1 presents the typing rules and syntax of these. We have not specified what values V are there because the objects in an environment will differ depending on evaluation strategy.

4.2.1 Call-by-Value. A call-by-value calculus with closures is presented in Figure 4.2. Abstract closures $\{\varsigma, \lambda x.\, M\}$ pair a function together with a syntactic environment. Note that the closure replaces the function form of the original calculus, but we can recover that using the syntactic sugar of an abstract closure in an empty environment. We have the same type system as the original simply typed λ-calculus, but have replaced the typing rule for functions with that of closures. Examining that rule, we see that the body of the function extends the current environment Γ with the one from the environment Γ' and the formal parameter x. Since Γ is included in this environment, abstract closures do not necessarily need to capture all of there free variables in their environment; this is essential for closure conversion in little pieces.

The β law for functions has been replaced with one for closures wherein we merely perform the delayed substitution when the function is applied along with the substitution for the formal parameter. Note the similarity to the SECD machine's step number 5

(Figure 2.3) and strict big-step semantics application case (Figure 3.1). The η law remains unchanged and applies *only* to λ-expressions with an empty environment. Indeed, the more general η law, which we call $\eta'_{\to}$, with a non-empty environment:

$$\{\varsigma, \lambda x.\, V\ x\} =_{\eta'_{\to}} V[\varsigma]$$

can be derived as follows:

$$\begin{aligned} V[\varsigma] &=_{\eta_{\to}} && \lambda x.\, V[\varsigma]\ x \\ &=_{\text{subst.}} && \lambda x.\, (V\ x)[\varsigma, x/x] \\ &=_{\beta_{\to}} && \lambda x.\, \{\varsigma, \lambda x.\, V\ x\}\ x \\ &=_{\eta_{\to}} && \{\varsigma, \lambda x.\, V\ x\} \end{aligned}$$

Since we now have a notion of closures that ties the environment and code part together in both the type system and equational theory, we are able to fix the problem of syntactically equating closures which have captured different numbers of variables; this failed before since the canonical closure conversion did not commute with substitution. For example, we can now equate these two abstract closures where one has an extra variable:

$$\begin{aligned} \{(\varepsilon, y/y), \lambda x.\, M\} &=_{\eta_{\to}} && \{(\varepsilon, y/y, z/z), \lambda x.\, \{(\varepsilon, y/y), \lambda x.\, M\}\ x\} \\ &=_{\beta_{\to}} && \{(\varepsilon, y/y, z/z), \lambda x.\, M\} \end{aligned}$$

4.2.2 Call-by-Name. A call-by-name calculus with closures is presented in Figure 4.3. Matching the Krivine machine, which builds closures for function arguments, we place abstract closures around function arguments. Contrary to call-by-value that constructs closures for function introduction, we need them for function elimination; and

$$\begin{array}{rcl} M, N, L \in & \mathit{Expression} & ::= \mathsf{b} \mid x \mid \lambda x.\, M \mid M\ \{\varsigma, N\} \\ V, W \in & \mathit{Value} & = \mathit{Expression} \end{array}$$

(a) Syntax

$$M\ N = M\ \{\varepsilon, N\}$$

(b) Syntactic Sugar

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash \varsigma : \Gamma' \quad \Gamma\Gamma' \vdash N : \sigma}{\Gamma \vdash M\ \{\varsigma, N\} : \tau} \to_E$$

(c) New Elimination Rule

$$\begin{array}{rcl} (\lambda x.\, M)\ V & =_\beta & M[V/x] \\ \lambda x.\, M\ x & =_\eta & M \\ M\ \{\varsigma, N\} & =_{cl} & M\ (N[\varsigma]) \end{array}$$

(d) Axioms

Figure 4.3. Call-by-Name with Closures

thus, that is the typing rule which we replace. The axioms from call-by-name remain the same here, but we add a new axiom *cl* for entering an argument's closure. We changed the look of the β law, but since call-by-name values are any expression, we have lost nothing.

It may seem unsatisfying to see that the *cl* axiom "enters" the closure before function application, whereas the Krivine machine does not enter a function until the variable evaluation is forced. However, this fits with the flexibility of an equational theory and we are primarily focused on stating where in our code that the operational semantics will construct closures. Indeed, a rule more like the Krivine machines evaluation:

$$(\lambda x.\, M)\ \{\varsigma, N\} = M[N[\varsigma]/x]$$

is derivable simply by using *cl* and then β.

4.2.3 Call-by-Need. A call-by-need calculus with closures is presented in Figure 4.4. With the addition of the memoizing expression, there are now more places where unevaluated code with free variables may be substituted to other parts of the program in an environment machine. Although they are not substituted by the axioms of the theory, the call-by-need evaluation context `let` x `be` E `in` $F[x]$ suggests that

$$\begin{array}{rcl} M, N, L \in & \textit{Expression} & ::= V \mid M\ N \mid \texttt{let}\ x\ \texttt{be}\ \{\varsigma, M\}\ \texttt{in}\ N \\ V, W \in & \textit{Value} & ::= \mathsf{b} \mid x \mid \{\varsigma, \lambda x.\, M\} \end{array}$$

(a) Syntax

$$\begin{array}{rcl} \lambda x.\, M & = & \{\varepsilon, \lambda x.\, M\} \\ \texttt{let}\ x\ \texttt{be}\ M\ \texttt{in}\ N & = & \texttt{let}\ x\ \texttt{be}\ \{\varepsilon, M\}\ \texttt{in}\ N \end{array}$$

(b) Syntactic Sugar

$$\frac{\Gamma \vdash \varsigma : \Gamma' \quad \Gamma\Gamma', x{:}\tau \vdash M : \sigma}{\Gamma \vdash \{\varsigma, \lambda x.\, M\} : \tau \to \sigma} \to_I \qquad \frac{\Gamma \vdash \varsigma : \Gamma' \quad \Gamma\Gamma' \vdash M : \sigma \quad \Gamma, x{:}\sigma \vdash N : \tau}{\Gamma \vdash \texttt{let}\ x\ \texttt{be}\ \{\varsigma, M\}\ \texttt{in}\ N : \tau} \textit{let}$$

(c) New Typing Rules

$$\begin{array}{rcl} \{\varsigma, \lambda x.\, M\}\ N & =_\beta & \texttt{let}\ x\ \texttt{be}\ N\ \texttt{in}\ M[\varsigma] \\ \lambda x.\, V\ x & =_\eta & V \\ \texttt{let}\ x\ \texttt{be}\ V\ \texttt{in}\ M & =_x & M[V/x] \\ E[\texttt{let}\ x\ \texttt{be}\ M\ \texttt{in}\ N] & =_\kappa & \texttt{let}\ x\ \texttt{be}\ M\ \texttt{in}\ E[N] \\ \texttt{let}\ x\ \texttt{be}\ (\texttt{let}\ y\ \texttt{be}\ M\ \texttt{in}\ N)\ \texttt{in}\ L & =_\chi & \texttt{let}\ y\ \texttt{be}\ M\ \texttt{in}\ \texttt{let}\ x\ \texttt{be}\ N\ \texttt{in}\ L \\ \texttt{let}\ x\ \texttt{be}\ \{\varsigma, M\}\ \texttt{in}\ N & =_{cl} & \texttt{let}\ x\ \texttt{be}\ M[\varsigma]\ \texttt{in}\ N \end{array}$$

(d) Axioms

Figure 4.4. Call-by-Need with Closures

while we are evaluating $F[x]$ inside of the let-binding we will need to "jump" to the location E to evaluate there. In an environment machine, this amounts to entering a different environment at runtime; therefore, it requires a closure as we see in the lazy abstract machine of Sestoft [48]. Thus, there are two spaces for abstract closures in the calculus: one for functions that works like that of the call-by-value closures and one for let-expressions that can be seen as the memoizing version of the call-by-name closures. Reflecting the machines as before, we see these two kinds of closures constructed in the Sestoft machine's rules 3 and 6 (Figure 2.7).

For the let-expression closure, we add a new law called *cl* which—like the similarly named law from call-by-name—allows us to perform the delayed substitution at any time. For the function closure, we have a new β law that performs the delayed substitution of the function while creating a memoizable binding for the argument. Indeed, we can derive the call-by-value abstract closure law:

$$\begin{aligned}
\{\varsigma, \lambda x.\, M\}\ V &=_\beta && \mathtt{let}\ x\ \mathtt{be}\ V\ \mathtt{in}\ M[\varsigma] \\
&=_x && M[\varsigma][V/x] \\
&=_{\text{subst.}} && M[\varsigma, V/x]
\end{aligned}$$

4.3 Deriving Closure Conversions

As an example of Theorem 4.1, we can now construct a naïve closure conversion transformation syntactically. We do this by deriving a simple rewriting rule that adds one free variable at a time. In the call-by-value with closure calculus, we would have the following rule:

$$\frac{y \in \mathrm{FV}(\lambda x.\, M) - \mathrm{Dom}(\varsigma)}{\{\varsigma, \lambda x.\, M\} \longrightarrow_{\mathrm{CC}} \{(\varsigma, y/y), \lambda x.\, M\}}$$

For each application of the rule, the local environment ς grows by an identity substitution y/y. If we started from an empty environment, then the entire ς after closure conversion is the identity. In the case where ς already had some non-identity part within, *e.g.* $\{(\varepsilon, 3/x, y/y), \lambda z.\, M\}$, the delayed substitution is preserved.

We show next that the rewrite rule is derivable, where we let the environment ς be $V_0/z_0, \ldots, V_n/z_n$ and y be a free variable in $\mathrm{FV}(\lambda x.\, M) - \mathrm{Dom}(\varsigma)$:

$$\begin{aligned}
\{\varsigma, \lambda x.\, M\} &=_{\text{subst.}} \\
\{\varsigma, \lambda x.\, M[z_0/z_0, \ldots, z_n/z_n, y/y, x/x]\} &=_{\beta_{\to}} \\
\{\varsigma, \lambda x.\, \{(z_0/z_0, \ldots, z_n/z_n, y/y), \lambda x.\, M\}\ x\} &=_{\eta'_{\to}} \\
\{(z_0/z_0, \ldots, z_n/z_n, y/y), \lambda x.\, M\}[\varsigma] &=_{\text{subst.}} \\
\{(\varsigma, y/y), \lambda x.\, M\}
\end{aligned}$$

We say that an expression is closure converted when it is a normal form with respect to the CC-rule. Such normal forms are unique up to the reordering of the substitutions. A closure conversion procedure can be derived by applying the transformation until this normal form is reached.

Definition 4.1 (Naïve Closure Conversion)**.**

$\mathrm{NCC}(A) = B$ *iff* $A \longrightarrow^*_{\mathrm{CC}} B$ *and* B *is in* CC-*normal form.*

We can define similar rules and thus a closure conversion for our call-by-name and call-by-need closure calculi.

4.4 Using Abstract Closures

The above closure conversion is a flat closure representation containing all of the free variables in a simple product-like data structure; this is but one approach for choosing a layout for a closure's environment. There is a diverse collection of work on closure analysis and optimizations [39, 19, 50, 34], but they assume a global closure conversion phase. Using a language with abstract closures allows us to do these locally after the naïve transformation has been applied. Here, we focus on two of these examples.

4.4.1 Choosing an Environment Representation.

Minamide *et al.* [34] combine the environments of different closures to save space when allocating a closure, at the cost of possible space leaks [50] and extended lookup times for closure variables. To do this with abstract closures, we need an easy way to combine sub-parts of environments together so they may be shared with other closures. Whereas in the closure laws above, we only substitute a flat environment of values, we now wish to represent environments with nested structures via pattern matching on finite products.

Like empty closures and let-expressions, pattern matching can be considered syntactic sugar. For instance, the pattern-matching closure $\{(\varepsilon, V /\!\!/ \langle x, \langle y, z\rangle\rangle), \lambda w.\, M\}$ desugars into the following:

$$\{(\varepsilon, V/v), \lambda w.\, \mathtt{case}\ v\ \mathtt{of}\ \{\langle x, v'\rangle \to \mathtt{case}\ v'\ \mathtt{of}\ \{\langle y, z\rangle \to M\}\}\}$$

Using this sugar, the environment sharing of the example from Shao and Appel [50] is a derivable equality in our language. Examining the first expression in Figure 4.5, we have already run our naïve closure conversion. The program allocates three closures named h, g, and j, which all contain the variables w, x, y, and z. To save space, we may derive an equality wherein these three closures point to a single sub-environment containing those values. We η expand the program saving the same variables in each closure, but this time we pair the variables that they have in common together. A β reduction in the body allows us to remove the old closure structure. Finally, we β expand to have a pointer e to share the product; and therefore, the value of variables will not be duplicated to allocate these closures in a runtime system that passes products by reference.

This is an example of taking naïve closure conversion as a starting point and transforming our code further to optimize sub-programs. So not only does $M = \mathrm{NCC}(M)$,

$$
\begin{array}{l}
\mathtt{let}\ g\ \mathsf{be}\ \{(\varepsilon, g/g, v/v, w/w, x/x, y/y, z/z), \lambda q.\, A\}\ \mathtt{in}\\
\mathtt{let}\ h\ \mathsf{be}\ \{(\varepsilon, h/h, u/u, w/w, x/x, y/y, z/z), \lambda q.\, B\}\ \mathtt{in}\\
\mathtt{let}\ j\ \mathsf{be}\ \{(\varepsilon, i/i, w/w, x/x, y/y, z/z), \lambda q.\, C\}\ \mathtt{in}\\
\quad D\\
\qquad =^{3}_{\eta_{\rightarrow}}\\
\mathtt{let}\ g\ \mathsf{be}\ \left\{\begin{array}{l}(\varepsilon, g/g, v/v, \langle w, x, y, z\rangle \mathbin{/\!\!/} \langle w, x, y, z\rangle),\\ \quad \lambda q.\, \{(\varepsilon, g/g, v/v, w/w, x/x, y/y, z/z), \lambda q.\, A\}\ q\end{array}\right\}\ \mathtt{in}\\
\mathtt{let}\ h\ \mathsf{be}\ \left\{\begin{array}{l}(\varepsilon, h/h, u/u, \langle w, x, y, z\rangle \mathbin{/\!\!/} \langle w, x, y, z\rangle),\\ \quad \lambda q.\, \{(\varepsilon, h/h, u/u, w/w, x/x, y/y, z/z), \lambda q.\, B\}\ q\end{array}\right\}\ \mathtt{in}\\
\mathtt{let}\ j\ \mathsf{be}\ \left\{\begin{array}{l}(\varepsilon, i/i, \langle w, x, y, z\rangle \mathbin{/\!\!/} \langle w, x, y, z\rangle),\\ \quad \lambda q.\, \{(\varepsilon, i/i, w/w, x/x, y/y, z/z), \lambda q.\, C\}\ q\end{array}\right\}\ \mathtt{in}\\
\quad D\\
\qquad =^{3}_{\beta_{\rightarrow}}\\
\mathtt{let}\ g\ \mathsf{be}\ \{(\varepsilon, g/g, v/v, \langle w, x, y, z\rangle \mathbin{/\!\!/} \langle w, x, y, z\rangle), \lambda q.\, A\}\ \mathtt{in}\\
\mathtt{let}\ h\ \mathsf{be}\ \{(\varepsilon, h/h, u/u, \langle w, x, y, z\rangle \mathbin{/\!\!/} \langle w, x, y, z\rangle), \lambda q.\, B\}\ \mathtt{in}\\
\mathtt{let}\ j\ \mathsf{be}\ \{(\varepsilon, i/i, \langle w, x, y, z\rangle \mathbin{/\!\!/} \langle w, x, y, z\rangle), \lambda q.\, C\}\ \mathtt{in}\\
\quad D\\
\qquad =_{\beta_{\mathtt{let}}}\\
\mathtt{let}\ e\ \mathsf{be}\ \langle w, x, y, z\rangle\ \mathtt{in}\\
\mathtt{let}\ g\ \mathsf{be}\ \{(\varepsilon, g/g, v/v, e \mathbin{/\!\!/} \langle w, x, y, z\rangle), \lambda q.\, A\}\ \mathtt{in}\\
\mathtt{let}\ h\ \mathsf{be}\ \{(\varepsilon, h/h, u/u, e \mathbin{/\!\!/} \langle w, x, y, z\rangle), \lambda q.\, B\}\ \mathtt{in}\\
\mathtt{let}\ j\ \mathsf{be}\ \{(\varepsilon, i/i, e \mathbin{/\!\!/} \langle w, x, y, z\rangle), \lambda q.\, C\}\ \mathtt{in}\\
\quad D
\end{array}
$$

Figure 4.5. Example of Environment Sharing with Abstract Closures

but also $M = (\mathrm{EnvShare} \circ \mathrm{NCC})(M)$. Moreover, this transformation preserves the CC-normal form property of $\mathrm{NCC}(M)$.

4.4.2 Choosing Environment Passing Technique. Another optimization presented for closures is lambda-lifting [19, 39]. In essence, lambda-lifting as an optimization is meant to pass parts of a code's environment on the call stack instead of its closure environment. It is enabled by β expansion on the free variables of functions whose code is visible from the call site, also referred to in the literature as *known functions*. For instance, consider the following example where the closure bound to f is transformed and whose body M has the free variable y:

$$\begin{aligned}
\texttt{let}\ f\ \texttt{be}\ \{(\varsigma, y/y), \lambda x.\, M\}\ \texttt{in}\ \ldots f\ 3 \ldots \quad &=_{\beta_{\to}} \\
\ldots \{(\varsigma, y/y), \lambda x.\, M\}\ 3 \ldots \quad &=_{\beta_{\to}} \\
\ldots M[\varsigma, y/y, 3/x] \ldots \quad &=_{\beta_{\to}} \\
\ldots \{\varsigma, \lambda y, x.\, M\}\ y\ 3 \ldots \quad &=_{\beta_{\texttt{let}}} \\
\texttt{let}\ f\ \texttt{be}\ \{\varsigma, \lambda y, x.\, M\}\ \texttt{in}\ \ldots f\ y\ 3 \ldots \quad &
\end{aligned}$$

Again, we start with a program that is already in CC-normal form. To avoid passing y within the closure's environment, which may require more allocation, the function closure has it added as an extra formal parameter and at the call site it is added as an extra argument. In the special case where the rest of the environment ς is empty, such an optimization may completely avoid allocating space for the environment part of a closure.

Such a transformation only depends on being able to β expand and having multi-arity functions, so many existing ILs can already do this. The advantage of having closures in our IL is that we may encode both environments sharing and lambda-lifting directly in the syntax and have the two optimizations interact with one another. Indeed, the final program here could have been specified incrementally, by first applying the naïve closure conversion followed by environment sharing and lambda-lifting. Additionally, transformations unrelated to closures will need to respect them as closures, in contrast to closure conversions that represent functions as normal products.

CHAPTER V

CBPVS: A COMMON INTERMEDIATE LANGUAGE

This chapter contains published and unpublished co-authored material. It contains revisions of the work Closure Conversion in Little Pieces [53] co-authored with Paul Downen and Zena M. Ariola. Zachary J. Sullivan is the primary author under the guidance of Paul Downen and Zena M. Ariola.

We have seen how closures arise from implementing the λ-calculus on modern machines and we presented a new approach to reasoning about closures as part of the calculus. However, we presented only a sketch of how it may be done for different evaluation strategies. Formalizing the relationship between abstract closures and the abstract machine's on which they run is a large task for just one language as we will see in the coming chapters. Thus, instead of doing the work three times for call-by-value, call-by-name, and call-by-need, we propose a single intermediate language that we formalize. We base our new intermediate language on Levy's call-by-push-value (CBPV) [27], which was originally motivated by a similar duplication of work in the denotational semantics for the call-by-value and call-by-name evaluation strategies. Unfortunately, CBPV is not equipped to handle call-by-need. Thus, we first need to describe how to extend the language to include sharing in a manner that preserves all of the equational theories of call-by-name, call-by-value, and call-by-need.

5.1 CBPV

CBPV (Figure 5.1) achieves its strong equational theory—thereby making a suitable target for call-by-name and call-by-value—by separating the objects that have different β and η laws. There is a syntactic distinction between expressions that *are*, which we call values and will write in green, and expressions that *do*, *i.e.* computations, which we

$$
\begin{array}{rcl}
\tau, \sigma \in & \textit{Type} & ::= \tau \mid \tau \\
\tau, \sigma \in & \textit{Value Type} & ::= B \mid \tau \otimes \sigma \mid U\ \tau \\
\tau, \sigma \in & \textit{Computation Type} & ::= \tau \mathbin{\&} \sigma \mid \tau \to \sigma \mid F\ \tau \\
\\
A, B, C \in & \textit{Expression} & ::= V \mid M \\
V, W \in & \textit{Value} & ::= \mathtt{b} \mid x \mid \langle V, W \rangle \mid \{\mathtt{force} \to M\} \\
M, N \in & \textit{Computation} & ::= \mathtt{case}\ V\ \mathtt{of}\ \{\langle x, y \rangle \to M\} \mid \{\mathtt{fst} \to M; \mathtt{snd} \to N\} \\
 & & \quad \mid M.\mathtt{fst} \mid M.\mathtt{snd} \mid \lambda x.\, M \mid M\ V \\
 & & \quad \mid \mathtt{ret}\ V \mid M\ \mathtt{to}\ x\ \mathtt{in}\ N \mid V.\mathtt{force}
\end{array}
$$

Figure 5.1. Syntax

$$
\frac{}{\Gamma \vdash \mathtt{b} : B}B \qquad
\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\mathit{var} \qquad
\frac{\Gamma, x{:}\tau \vdash M : \sigma}{\Gamma \vdash \lambda x.\, M : \tau \to \sigma}{\to_I} \qquad
\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash V : \sigma}{\Gamma \vdash M\ V : \tau}{\to_E}
$$

$$
\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \rho}{\Gamma \vdash \{\mathtt{fst} \to M; \mathtt{snd} \to N\} : \tau \mathbin{\&} \rho}\&_I \qquad
\frac{\Gamma \vdash M : \tau \mathbin{\&} \rho}{\Gamma \vdash M.\mathtt{fst} : \tau}\&_{E1} \qquad
\frac{\Gamma \vdash M : \tau \mathbin{\&} \rho}{\Gamma \vdash M.\mathtt{snd} : \rho}\&_{E2}
$$

$$
\frac{\Gamma \vdash V : \tau \qquad \Gamma \vdash W : \sigma}{\Gamma \vdash \langle V, W \rangle : \tau \otimes \sigma}\otimes_I \qquad
\frac{\Gamma \vdash V : \sigma \otimes \rho \qquad \Gamma, x{:}\sigma, y{:}\rho \vdash M : \tau}{\Gamma \vdash \mathtt{case}\ V\ \mathtt{of}\ \{\langle x, y \rangle \to M\} : \tau}\otimes_E
$$

$$
\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \{\mathtt{force} \to M\} : U\ \tau}U_I \qquad
\frac{\Gamma \vdash V : U\ \tau}{\Gamma \vdash V.\mathtt{force} : \tau}U_E
$$

$$
\frac{\Gamma \vdash V : \tau}{\Gamma \vdash \mathtt{ret}\ V : F\ \tau}F_I \qquad
\frac{\Gamma \vdash M : F\ \sigma \qquad \Gamma, x{:}\sigma \vdash N : \tau}{\Gamma \vdash M\ \mathtt{to}\ x\ \mathtt{in}\ N : \tau}F_E
$$

Figure 5.2. Typing Rules

write in orange. Indeed, there are two different product types: one is a computation, the $\&$ type, which will be the target of a lazy style of product and the other, the $\otimes$ type, is a value for the strict products. The former is defined by the projections `.fst` and `.snd`, whereas the latter is constructed as pair that is eliminated by a case expression; the products that we have used thus far are the latter. Note that we depart from the syntax of Levy for expressions containing computations that wait for method calls, *e.g.* `thunk` M is written as $\{\mathtt{force} \to M\}$, to emphasize how they behave like objects. This is because we will introduce more expressions of a similar kind later in this chapter.

By the rules of the syntax, arguments to function calls and the interrogated expression of the case expression are *already* values and no reduction will be needed; this—in contrast with call-by-value where we must evaluate expressions to get to a value—conveys the idea that values *are*, and that they can be predicted based on their *type* alone. Moreover, as we see from the syntax and typing rules Figure 5.2, variables can only range over value types, and so substitutions only occur with values. On the other hand, computations are the only expressions that are allowed to do work to find an answer, so a computation that returns values of type τ will have type $F\ \tau$. For example, a computation returning $\langle 4, 2\rangle$ will be written $\mathtt{ret}\ \langle 4, 2\rangle$ in a manner reminiscent of returning from a statement in C. A to-expression, which consumes computations of type $F\ \tau$, may need to evaluate its interrogated computation before being able to match on the pattern and extract a value to bind it to x.

Despite only values being substitutable, the language is still able to pass unevaluated code because we may delay a computation as a value by shifting it. The object-like shift for delaying a computation as a value, written as $\{\mathtt{force} \to M\}$, is eliminated by $V.\mathtt{force}$. Conversely, the ret- and to-expressions shift from values into computations. We will see later that these shifts play a key role in both closure conversion and sharing. For now, note that an important difference between data-like and object-like shifts is that the former creates a binding upon elimination whereas the latter does not.

5.1.1 Equational Theory.

Examining the equational theory in Figure 5.3, we see that it has much of the same β laws as call-by-value. However, the requirement that the argument of a function and the subject of a case expression is a value becomes a *syntactic* restriction, whereas in a call-by-value calculus the requirement is imposed at runtime. Examining the η laws, we see that CBPV has the η law for call-by-name functions in

$$\begin{array}{rcl}
(\lambda x.\, M)\ V & =_{\beta_\to} & M[V/x] \\
\{\mathtt{fst} \to M; \mathtt{snd} \to N\}.\mathtt{fst} & =_{\beta_{\&1}} & M \\
\{\mathtt{fst} \to M; \mathtt{snd} \to N\}.\mathtt{snd} & =_{\beta_{\&2}} & N \\
\mathtt{case}\ \langle V, W\rangle\ \mathtt{of}\ \{\langle x, y\rangle \to M\} & =_{\beta_\otimes} & M[V/x, W/y] \\
\{\mathtt{force} \to M\}.\mathtt{force} & =_{\beta_U} & M \\
(\mathtt{ret}\ V)\ \mathtt{to}\ x\ \mathtt{in}\ M & =_{\beta_F} & M[V/x] \\
\\
\lambda x.\, M\, x & =_{\eta_\to} & M \\
\{\mathtt{fst} \to M.\mathtt{fst}; \mathtt{snd} \to M.\mathtt{snd}\} & =_{\eta_\&} & M \\
\mathtt{case}\ V\ \mathtt{of}\ \{\langle x, y\rangle \to M[\langle x, y\rangle/z]\} & =_{\eta_\otimes} & M[V/z] \\
\{\mathtt{force} \to V.\mathtt{force}\} & =_{\eta_U} & V \\
M\ \mathtt{to}\ x\ \mathtt{in}\ E[\mathtt{ret}\ x] & =_{\eta_F} & E[M]
\end{array}$$

$$E \in \quad \textit{Evaluation Context} \quad ::= \square \mid E\ V \mid E.\mathtt{fst} \mid E.\mathtt{snd} \mid E\ \mathtt{to}\ x\ \mathtt{in}\ M$$

Figure 5.3. CBPV Axioms

contrast to the restrictions that we see in the call-by-value and call-by-need.[1] Additionally, there are laws for the two shift types. The β and η laws for the value shift $U\ \tau$ are analogous to those of the $\&$ type. For $F\ \tau$, the β law is analogous to the pattern matching in a case expression, but its η law is more restricted. That is, it can only be applied when the reconstructed data appears in an evaluation context; note that this is a slightly more general law than originally given by Levy.

5.1.2 Subsuming Call-by-Value and Call-by-Name.

The compilation to CBPV for use as an intermediate language is given in Figure 5.4. Both translations are composed of sub-translations for types, type environments, and expressions. For call-by-name, arguments are compiled into shifts which delay evaluation until they are forced in the variable case. Because of the delayed evaluation of all function arguments, we see that the type environment translation is U of the translated argument type. In call-by-value, we must compile functions, which are treated as values in the source, into delayed computations since functions are computations in CBPV. This way, a function

[1]CBPV's strong η laws for computation types and value types follow closely the observations about call-by-name and call-by-value types in work on the duality of programming languages [55, 15].

$$\begin{aligned}
\underline{\tau \to \rho} &= U\,\underline{\tau} \to \underline{\rho} \\
\underline{B} &= F\,B \\
\underline{\tau \times \sigma} &= \underline{\tau} \mathbin{\&} \underline{\sigma} \\[4pt]
\underline{\varepsilon} &= \varepsilon \\
\underline{\Gamma, x{:}\tau} &= \underline{\Gamma}, x{:}U\,\underline{\tau}
\end{aligned}
\qquad
\begin{aligned}
\underline{x} &= x.\mathsf{force} \\
\underline{b} &= \mathsf{ret}\;b \\
\underline{\lambda x.\,M} &= \lambda x.\,\underline{M} \\
\underline{M\;N} &= \underline{M}\;\{\mathsf{force} \to \underline{N}\} \\
\underline{\langle M, N\rangle} &= \{\mathsf{fst} \to \underline{M}; \mathsf{snd} \to \underline{N}\} \\
\underline{\mathsf{fst}\;M} &= \underline{M}.\mathsf{fst} \\
\underline{\mathsf{snd}\;M} &= \underline{M}.\mathsf{snd}
\end{aligned}$$

(a) Call-by-Name (CBN)

$$\begin{aligned}
\underline{\tau \to \sigma} &= U\,(\underline{\tau} \to F\,\underline{\sigma}) \\
\underline{b} &= B \\
\underline{\tau \times \sigma} &= \underline{\tau} \otimes \underline{\sigma} \\[4pt]
\underline{\varepsilon} &= \varepsilon \\
\underline{\Gamma, x{:}\tau} &= \underline{\Gamma}, x{:}\underline{\tau}
\end{aligned}
\qquad
\begin{aligned}
\underline{x} &= \mathsf{ret}\;x \\
\underline{b} &= \mathsf{ret}\;b \\
\underline{\lambda x.\,M} &= \mathsf{ret}\;\{\mathsf{force} \to \lambda x.\,\underline{M}\} \\
\underline{M\;N} &= \underline{M}\;\mathsf{to}\;x\;\mathsf{in}\;\underline{N}\;\mathsf{to}\;y\;\mathsf{in}\;x.\mathsf{force}\;y \\
\underline{\langle M, N\rangle} &= \underline{M}\;\mathsf{to}\;x\;\mathsf{in}\;\underline{N}\;\mathsf{to}\;y\;\mathsf{in}\;\mathsf{ret}\;\langle x, y\rangle \\
\underline{\mathsf{case}\;M\;\mathsf{of}\;\{\langle x, y\rangle \to N\}} &= \underline{M}\;\mathsf{to}\;z\;\mathsf{in}\;\mathsf{case}\;z\;\mathsf{of}\;\{\langle x, y\rangle \to \underline{N}\}
\end{aligned}$$

(b) Call-by-Value (CBV)

Figure 5.4. Compiling CBN and CBV to CBPV

in the source is turned into F of the translated function value. Both translations also include a notion of product as well, the call-by-name version of a product is treated as the computation product type $\&$ whereas the call-by-value version is treated as the value product $\otimes$.

These transformations preserve not only types, but also the equational theories of the source languages. The following theorems are found in Levy [27] and show that CBPV is suitable as an intermediate language.

Theorem 5.1 (CBN Compilation Preserves Equations)**.** *If* $\Gamma \vdash M =_{\text{CBN}} N : \tau$*, then* $\underline{\Gamma} \vdash \underline{M} =_{\text{CBPV}} \underline{N} : \underline{\tau}$ *using the call-by-name translation.*

Theorem 5.2 (CBV Compilation Preserves Equations)**.** *If* $\Gamma \vdash M =_{\text{CBV}} N : \tau$*, then* $\underline{\Gamma} \vdash \underline{M} =_{\text{CBPV}} \underline{N} : F\,\underline{\tau}$ *using the call-by-value translation.*

$$\begin{array}{rcll}
\tau, \sigma \in & \textit{Value Type} & ::= & B \mid \tau \otimes \sigma \mid U\ \tau \mid \tilde{U}\ \tau \\
\tau, \sigma \in & \textit{Shared Type} & ::= & \check{U}\ \tau \mid \hat{F}\ \tau \\
\tau, \sigma \in & \textit{Comp. Type} & ::= & \tau \mathbin{\&} \sigma \mid \tau \to \sigma \mid F\ \tau \mid \tilde{F}\ \tau \\
 & & & \\
V, W \in & \textit{Value} & ::= & \mathtt{b} \mid x \mid \langle V, W \rangle \mid \{\mathtt{force} \to M\} \mid \mathtt{box}\ V \\
V, W \in & \textit{Shared Value} & ::= & a \mid \mathtt{val}\ V \mid \{\mathtt{enter} \to M\} \\
R, S \in & \textit{Shared Comp.} & ::= & V \mid M.\mathtt{eval} \mid B[R] \\
M, N \in & \textit{Comp.} & ::= & \{\mathtt{fst} \to M; \mathtt{snd} \to N\} \mid M.\mathtt{fst} \mid M.\mathtt{snd} \mid \lambda x.M \mid M\ V \\
 & & & \mid V.\mathtt{force} \mid \mathtt{ret}\ V \mid B[M] \mid R.\mathtt{enter} \mid \{\mathtt{eval} \to R\} \\
 & & & \\
B \in & \textit{Block Ctxt.} & ::= & P\ \mathtt{to}\ x\ \mathtt{in}\ \square \mid R\ \mathtt{memo}\ a\ \mathtt{in}\ \square \\
 & & & \mid \mathtt{case}\ V\ \mathtt{of}\ \{\langle x, y \rangle \to \square\} \mid \mathtt{case}\ V\ \mathtt{of}\ \{\mathtt{box}\ a \to \square\} \\
 & & & \\
P, Q \in & \textit{Comp. Expr.} & ::= & R \mid M \\
 & & & \\
\varsigma \in & \textit{Env.} & ::= & \varepsilon \mid \varsigma, V/x \mid \varsigma, V/a
\end{array}$$

Figure 5.5. CBPVS: CBPV with Sharing

5.2 Adding Sharing

To serve as a suitable intermediate language for a call-by-need source, we need to support sharing within CBPV itself. Our new language, which we refer to as CBPVS for short ("S" for sharing), is shown in Figure 5.5.

The first place to start when extending our intermediate language with sharing is to, like in call-by-need, add a binding construct that gives names to the computation that we wish to share and only evaluates it when needed. Such a binding may look like $M\ \mathtt{memo}\ a\ \mathtt{in}\ N$. Of course, we would like to save computations that return values (those of type $F\ \tau$): imagine having M be the program $1 + 2\ \mathtt{to}\ x\ \mathtt{in}\ \mathtt{ret}\ x$. To maximize sharing, we also need to be able to memoize the evaluation of intermediate computations of all types. For instance, we would only want to perform the β reduction on the argument 42 once, where M is $(\lambda x.\lambda y.\,\mathtt{ret}\ x)\ 42$, if we were to bind it to a variable and apply it in multiple parts of the program. In general, the point at which we may substitute without work duplication is when an introduction form for a computation type is reached, *i.e.* $\mathtt{ret}\ V$,

$\{\texttt{fst} \to M; \texttt{snd} \to N\}$, and $\lambda x.\, M$. We could merely declare these forms "computational values" that are substituted without duplicating work. However, we would then have to sacrifice our strong η law for CBPV function types for the weaker one found in call-by-need, which would mean that the compilation of the other evaluation strategies would no longer be equivalence preserving. Instead, we package computations that will be shared under a third syntactic category which stands apart from values and computations. This way, computations can keep their axioms from CBPV.

The new syntactic category that we introduce, which we write in purple, is for shared computations and its substitutable forms are the subset of shared values. Shared computations will be β reducible similar to computations, but with the addition of variables that refer only to them. There is an overlap of the block structures of the language where both computations and shared computations can pattern match on values, sequence computations, and bind shared computations. This is captured in the idea of block contexts, which contain one of these structures with a hole at the bottom. Where computations and shared computations overlap, we describe them as computable expressions and write them in the color black.

Since we will not be using computation introduction forms for sharing, like in $\lambda x.\, \texttt{ret}\ 42$, we need a shift from computations to shared values, which we write $\{\texttt{enter} \to \lambda x.\, \texttt{ret}\ 42\}$ and give the type $\check{U}\ (\tau \to F\ \mathbb{N})$. Similarly, we want a shift from values, which we write $\texttt{val}\ 42$ and give the type $\hat{F}\ \mathbb{N}$. The introduction forms of these shifts are the memoizable sub-syntax of shared computations: shared values. The opposite direction is true as well. We will want to embed shared computations within a data structure; this we do with the shift $\texttt{box}\ V$ with the type $\tilde{U}\ \tau$. And we want shared computations to be capable of being embedded within the normal computations to make use of computation types within a program: $\{\texttt{eval} \to R\}$ with the type $\tilde{F}\ \tau$. Indeed, computational types like functions and $\&$ can only contain shared sub-computations through such a shift; for example, $\{\mathtt{fst} \rightarrow \{\mathtt{eval} \rightarrow R\}; \mathtt{fst} \rightarrow \{\mathtt{eval} \rightarrow S\}\}$. In summary, the new shifts into shared expressions, $\check{U}\ \tau$ and $\hat{F}\ \tau$, are used to capture the CBPV values and computations that a shared expression reduced to, whereas that new shifts from shared expressions $\tilde{U}\ \tau$ and $\tilde{F}\ \tau$ are there so that we can make use of the existing value and computation types when building shared computations.

5.2.1 Typing Rules. The typing rules for CBPVS are given in Figure 5.6; for the subset that is CBPV, the rule remain the same except that the typing context Γ is now composed of both value and shared variables. For sharing, we must break the convention that CBPV only substitutes values; note that Levy himself breaks the convention with complex values [27] and other work adding memoization to CBPV does as well [32]. The type system reveals the similarities between the shared shifts and the ones that already existed in CBPV. Like the sequencing to-expression consuming values shifted to computations, the shared to-expression will bind a shifted value of type $\hat{F}\ \tau$ in another computation or shared computation.

5.2.2 Equational Theory. The axioms for CBPVS are given in Figure 5.7. The rules are divided into three sets wherein the first two are the usual β and η laws and the last includes rules for lifting and reassociating shared binders as in call-by-need. In general, we see that the β and η laws for shared computations and values operate in a similar manner to the other U and F types already in CBPV.

Concerning η, there is a notable difference between the laws for the $\tilde{U}$ types and those for $\hat{F}$ and F even though they all reconstruct a data-like expression; that is, the former does not have a restriction that the reconstructed data appears within an evaluation context. This lack of a restriction in the axiom, despite containing a shared expression, is possible because of the syntactic restriction to shared values for $\mathsf{box}\ V$. Without the syntactic

$$\frac{}{\Gamma \vdash \mathsf{b} : B}\,B \qquad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\,\mathit{var}$$

$$\frac{\Gamma \vdash V : \tau \quad \Gamma \vdash W : \sigma}{\Gamma \vdash \langle V, W \rangle : \tau \otimes \sigma}\,\otimes_I \qquad \frac{\Gamma \vdash V : \sigma \otimes \rho \quad \Gamma, x{:}\sigma, y{:}\rho \vdash P : \tau}{\Gamma \vdash \mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y \rangle \longrightarrow P\} : \tau}\,\otimes_E$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \{\texttt{force} \longrightarrow M\} : U\ \tau}\,U_I \qquad \frac{\Gamma \vdash R : U\ \tau}{\Gamma \vdash R.\texttt{force} : \tau}\,U_E$$

$$\frac{\Gamma \vdash V : \tau}{\Gamma \vdash \texttt{ret}\ V : F\ \tau}\,F_I \qquad \frac{\Gamma \vdash M : F\ \sigma \quad \Gamma, x{:}\sigma \vdash P : \tau}{\Gamma \vdash M\ \mathsf{to}\ x\ \mathsf{in}\ P : \tau}\,F_E$$

$$\frac{\Gamma \vdash V : \tau}{\Gamma \vdash \texttt{box}\ V : \tilde{U}\ \tau}\,\tilde{U}_I \qquad \frac{\Gamma \vdash V : \tilde{U}\ \sigma \quad \Gamma, a{:}\sigma \vdash P : \tau}{\Gamma \vdash \mathsf{case}\ V\ \mathsf{of}\ \{\texttt{box}\ a \longrightarrow P\} : \tau}\,\tilde{U}_E$$

$$\frac{a{:}\tau \in \Gamma}{\Gamma \vdash a : \tau}\,\mathit{svar} \qquad \frac{\Gamma \vdash R : \sigma \quad \Gamma, a{:}\sigma \vdash P : \tau}{\Gamma \vdash R\ \mathsf{memo}\ a\ \mathsf{in}\ P : \tau}\,H$$

$$\frac{\Gamma \vdash V : \tau}{\Gamma \vdash \texttt{val}\ V : \hat{F}\ \tau}\,\hat{F}_I \qquad \frac{\Gamma \vdash R : \hat{F}\ \sigma \quad \Gamma, x{:}\sigma \vdash P : \tau}{\Gamma \vdash R\ \mathsf{to}\ x\ \mathsf{in}\ P : \tau}\,\hat{F}_E$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \{\texttt{enter} \longrightarrow M\} : \check{U}\ \tau}\,\check{U}_I \qquad \frac{\Gamma \vdash R : \check{U}\ \tau}{\Gamma \vdash R.\texttt{enter} : \tau}\,\check{U}_E$$

$$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash \{\texttt{fst} \longrightarrow M; \texttt{snd} \longrightarrow N\} : \tau \mathbin{\&} \sigma}\,\&_I \qquad \frac{\Gamma \vdash M : \tau \mathbin{\&} \sigma}{\Gamma \vdash M.\texttt{fst} : \tau}\,\&_{E1} \qquad \frac{\Gamma \vdash M : \tau \mathbin{\&} \sigma}{\Gamma \vdash M.\texttt{snd} : \sigma}\,\&_{E2}$$

$$\frac{\Gamma, x{:}\tau \vdash M : \sigma}{\Gamma \vdash \lambda x.\, M : \tau \longrightarrow \sigma}\,\longrightarrow_I \qquad \frac{\Gamma \vdash M : \tau \longrightarrow \sigma \quad \Gamma \vdash V : \tau}{\Gamma \vdash M\ V : \sigma}\,\longrightarrow_E$$

$$\frac{\Gamma \vdash R : \tau}{\Gamma \vdash \{\texttt{eval} \longrightarrow R\} : \tilde{F}\ \tau}\,\tilde{F}_I \qquad \frac{\Gamma \vdash M : \tilde{F}\ \tau}{\Gamma \vdash M.\texttt{eval} : \tau}\,\tilde{F}_E$$

Figure 5.6. CBPVS Typing Rules

$$\begin{array}{rcl}
(\lambda x.\, M)\ V & =_{\to} & M[V/x] \\
\{\texttt{fst} \longrightarrow M; \texttt{snd} \longrightarrow N\}.\texttt{fst} & =_{\&1} & M \\
\{\texttt{fst} \longrightarrow M; \texttt{snd} \longrightarrow N\}.\texttt{snd} & =_{\&2} & N \\
\texttt{case}\ \langle V, W\rangle\ \texttt{of}\ \{\langle x, y\rangle \longrightarrow P\} & =_{\otimes} & P[V/x, W/y] \\
\{\texttt{force} \longrightarrow M\}.\texttt{force} & =_{U} & M \\
(\texttt{ret}\ V)\ \texttt{to}\ x\ \texttt{in}\ P & =_{F} & P[V/x] \\
\texttt{case}\ (\texttt{box}\ V)\ \texttt{of}\ \{\texttt{box}\ a \longrightarrow P\} & =_{\tilde{U}} & P[V/a] \\
(\texttt{val}\ V)\ \texttt{to}\ x\ \texttt{in}\ P & =_{\hat{F}} & P[V/x] \\
\{\texttt{enter} \longrightarrow M\}.\texttt{enter} & =_{\check{U}} & M \\
\{\texttt{eval} \longrightarrow R\}.\texttt{eval} & =_{\tilde{F}} & R
\end{array}$$

(a) β-laws

$$\begin{array}{rcl}
\lambda x.\, M\ x & =_{\to} & M \\
\{\texttt{fst} \longrightarrow M.\texttt{fst}; \texttt{snd} \longrightarrow M.\texttt{snd}\} & =_{\&} & M \\
\texttt{case}\ V\ \texttt{of}\ \{\langle x, y\rangle \longrightarrow P[\langle x, y\rangle/z]\} & =_{\otimes} & P[V/z] \\
\{\texttt{force} \longrightarrow V.\texttt{force}\} & =_{U} & V \\
M\ \texttt{to}\ x\ \texttt{in}\ E[\texttt{ret}\ x] & =_{F} & E[M] \\
\texttt{case}\ V\ \texttt{of}\ \{\texttt{box}\ a \longrightarrow P[\texttt{box}\ a/x]\} & =_{\tilde{U}} & P[V/x] \\
R\ \texttt{to}\ x\ \texttt{in}\ E[\texttt{val}\ x] & =_{\hat{F}} & E[R] \\
\{\texttt{enter} \longrightarrow V.\texttt{enter}\} & =_{\check{U}} & V \\
\{\texttt{eval} \longrightarrow M.\texttt{eval}\} & =_{\tilde{F}} & M
\end{array}$$

(b) η-laws

$$\begin{array}{rcl}
E[R\ \texttt{memo}\ a\ \texttt{in}\ P] & =_{\kappa} & R\ \texttt{memo}\ a\ \texttt{in}\ E[P] \\
(R\ \texttt{memo}\ b\ \texttt{in}\ S)\ \texttt{memo}\ a\ \texttt{in}\ P & =_{\chi} & R\ \texttt{memo}\ b\ \texttt{in}\ (S\ \texttt{memo}\ a\ \texttt{in}\ P) \\
V\ \texttt{memo}\ a\ \texttt{in}\ C[a] & =_{\text{deref}} & V\ \texttt{memo}\ a\ \texttt{in}\ C[V] \\
R\ \texttt{memo}\ a\ \texttt{in}\ P & =_{\text{GC}} & P \\
R & =_{\text{name}} & R\ \texttt{memo}\ a\ \texttt{in}\ a
\end{array}$$

(c) Other laws

$$\begin{array}{lll}
\textbf{where}\quad E, F & ::= & \square \mid E\ V \mid E.\texttt{fst} \mid E.\texttt{snd} \mid E\ \texttt{to}\ x\ \texttt{in}\ P \\
 & \mid & E.\texttt{enter} \mid E.\texttt{eval} \mid R\ \texttt{memo}\ a\ \texttt{in}\ E \mid E\ \texttt{memo}\ a\ \texttt{in}\ F[a]
\end{array}$$

Figure 5.7. CBPVS Axioms

restriction, a program like the following will duplicate work:

$$\begin{aligned}\mathtt{case}\ \langle 42, \mathtt{box}\ R\rangle\ \mathtt{of}\ \{\langle x, y\rangle \to \dots y \dots y \dots\} &\longrightarrow_\beta \\ \dots \mathtt{box}\ R \dots \mathtt{box}\ R \dots &\end{aligned}$$

This duplication will happen whenever a box-shift is nested inside of another value. Note that we can still describe a program like the one above where R is shared, but this time we will need to bind the non-duplicated part to a memo-expression first:

$$\begin{aligned}R\ \mathtt{memo}\ a\ \mathtt{in}\ \mathtt{case}\ \langle 42, \mathtt{box}\ a\rangle\ \mathtt{of}\ \{\langle x, y\rangle \to \dots x \dots x \dots\} &\longrightarrow_\beta \\ R\ \mathtt{memo}\ a\ \mathtt{in}\ \dots \mathtt{box}\ a \dots \mathtt{box}\ a \dots &\end{aligned}$$

Now the shared computation R is shared among the various places where a may occur.

Whereas η for $\tilde{U}$ is flexible because of a syntactic restriction, the law for $\check{U}$ types has a value restriction. This is for the same reason as the call-by-need value restriction for function type η: we must preserve that the expression is a shared value before and after an η reduction.

In CBPV, we were able to derive the sequencing laws of to-expressions with the generalized η law for F; we may do this for $\hat{F}$ as well. This is not true for lifting shared memoization bindings out of an evaluation context since the expression does not force its bound expression and that expression may be of any shared type. Therefore, the equational theory has a κ law specifically for this as in call-by-need.

5.3 Subsuming Call-by-Need

Figure 5.8 shows how a call-by-need source program will be compiled into our intermediate language. The transformation turns both types and expressions into their shared version in CBPVS. Those familiar with the subsumption of call-by-name and call-by-value into CBPV may see the transformation as merging the two: functions must

$$\begin{aligned}
\underline{\tau \to \sigma} &= \check{U}\,(\tilde{U}\,\underline{\tau} \to \tilde{F}\,\underline{\sigma}) \\
\underline{\mathbb{N}} &= \hat{F}\,\mathbb{N} \\
\underline{\tau \times \sigma} &= \hat{F}\,(\tilde{U}\,\underline{\tau} \otimes \tilde{U}\,\underline{\sigma}) \\
\underline{\varepsilon} &= \varepsilon \\
\underline{\Gamma, x{:}\tau} &= \underline{\Gamma}, x{:}\underline{\tau}
\end{aligned}$$

$$\begin{aligned}
\underline{x} &= x \\
\underline{\mathsf{b}} &= \mathtt{val}\ \mathsf{b} \\
\underline{\lambda x.\, M} &= \{\mathtt{enter} \to \lambda y.\,\mathtt{case}\ y\ \mathtt{of} \\
&\qquad \{\mathtt{box}\ x \to \{\mathtt{eval} \to \underline{M}\}\}\} \\
\underline{M\ N} &= \underline{M}\ \mathtt{memo}\ a\ \mathtt{in}\ \underline{N}\ \mathtt{memo}\ b\ \mathtt{in} \\
&\qquad (a.\mathtt{enter}\ (\mathtt{box}\ b)).\mathtt{eval} \\
\underline{\mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ N} &= \underline{M}\ \mathtt{memo}\ x\ \mathtt{in}\ \underline{N} \\
\underline{\langle M, N\rangle} &= \underline{M}\ \mathtt{memo}\ a\ \mathtt{in}\ \underline{N}\ \mathtt{memo}\ b\ \mathtt{in} \\
&\qquad \mathtt{val}\ \langle \mathtt{box}\ a, \mathtt{box}\ b\rangle \\
\underline{\mathtt{case}\ M\ \mathtt{of}\ \{\langle x, y\rangle \to N\}} &= \underline{M}\ \mathtt{to}\ z\ \mathtt{in}\ \mathtt{case}\ z\ \mathtt{of} \\
&\qquad \{\langle \mathtt{box}\ x, \mathtt{box}\ y\rangle \to \underline{N}\}
\end{aligned}$$

Figure 5.8. Compiling CBNeed to CBPVS

delay their argument type with $\tilde{U}$ instead of U, return their result with $\tilde{F}$ instead of F, and the whole computation must be delayed with $\check{U}$ instead of U. For expressions, the transformation has striking similarities to the call-by-value compilation. First, functions are placed in an enter-expression and expect their argument to come in a box-value; this means that arguments of a function must be given a shared binding before entering the function. Second, in an application, we must give names to the parts whose evaluation we want to share; in the call-by-value transformation, it is the parts that we simply want to evaluate. Though it is only the argument part that we wish to share, we must also give a name to the function part in order to preserve the ordering of memoized binders from the source. Similarly, we must give memoized binders to the sub-components of products. In so doing, we have preserved the Haskell-like product property that the sub-components will share their evaluation.

For brevity, the compilation from call-by-need makes use of nested pattern matching in the case expression transform, *i.e.* in unpacking a box-value inside of a product. Like with CBPV, nested pattern matching is equivalent to doing a pattern match one at a time in CBPVS.

$$\begin{array}{rl}
(\lambda x.\, M)\ N =_{\beta_\to} & \mathtt{let}\ x\ \mathtt{be}\ N\ \mathtt{in}\ M \\
\mathtt{case}\ \langle M, N\rangle\ \mathtt{of}\ \{\langle x,y\rangle \to L\} =_{\beta_\otimes} & \mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ \mathtt{let}\ y\ \mathtt{be}\ N\ \mathtt{in}\ L \\
\\
\lambda x.\, V\ x =_{\eta_\to} & V \\
\mathtt{case}\ M\ \mathtt{of}\ \{\langle x,y\rangle \to E[\langle x,y\rangle]\} =_{\eta_\otimes} & E[M] \\
\\
(\mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ N)\ L =_{\mathrm{lift1}} & \mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ (N\ L) \\
M\ (\mathtt{let}\ x\ \mathtt{be}\ N\ \mathtt{in}\ L) =_{\mathrm{lift2}} & \mathtt{let}\ x\ \mathtt{be}\ N\ \mathtt{in}\ (M\ L) \\
\lambda x.\, \mathtt{let}\ y\ \mathtt{be}\ V\ \mathtt{in}\ M =_{\mathrm{lift3}} & \mathtt{let}\ y\ \mathtt{be}\ V\ \mathtt{in}\ \lambda x.\, M \\
\mathtt{case}\ (\mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ N)\ \mathtt{of}\ \{\langle x,y\rangle \to L\} =_{\mathrm{lift4}} & \mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ (\mathtt{case}\ N\ \mathtt{of}\ \{\langle x,y\rangle \to L\}) \\
\\
\mathtt{let}\ x\ \mathtt{be}\ (\mathtt{let}\ y\ \mathtt{be}\ M\ \mathtt{in}\ N)\ \mathtt{in}\ L =_{\mathrm{merge}} & \mathtt{let}\ y\ \mathtt{be}\ M\ \mathtt{in}\ \mathtt{let}\ x\ \mathtt{be}\ N\ \mathtt{in}\ L \\
\mathtt{let}\ x\ \mathtt{be}\ V\ \mathtt{in}\ C[x] =_{\mathrm{deref}} & \mathtt{let}\ x\ \mathtt{be}\ V\ \mathtt{in}\ C[V] \\
\mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ N =_{\mathrm{GC}} & N \\
M =_{\mathrm{name}} & \mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ x
\end{array}$$

$$\begin{array}{lrcl}
\textbf{where} & V, W \in & \mathit{Value} & ::= x \mid \mathsf{b} \mid \lambda x.\, M \mid \langle V, W\rangle \\
& E, F \in & \mathit{EvalCxt} & ::= \square \mid E\ N \mid \mathtt{case}\ E\ \mathtt{of}\ \{\langle x,y\rangle \to N\} \\
& & & \quad \mid \mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ E \mid \mathtt{let}\ x\ \mathtt{be}\ E\ \mathtt{in}\ F[x]
\end{array}$$

Figure 5.9. More Flexible Call-by-Need Axioms

Lemma 5.1 (Compilation Commutes with Substitution). *Syntactically, we have* $\Gamma \vdash \underline{M}[\underline{V}/x] = \underline{M[V/x]} : \tau$.

Proof. Follows by the definitions of substitution for the two languages and induction on the structure of the expression. □

Lemma 5.2 (CBNeed Compilation Preserves Values). *For some source value* V*, there is some target value* W *such that* CBPVS $\vdash \underline{V} = W : \tau$.

Proof. By the induction on the syntax of source values. The cases for x, b, and $\lambda x.\, M$ hold immediately with reflexivity. The case for $\langle V, W\rangle$ follows by its inductive hypotheses followed by deref reductions. □

Note that we prove our subsumption theorem with respect to the call-by-need calculus in Figure 5.9. It presents a more flexible call-by-need than that of Figure 2.8, which includes more lifting rules from Ariola and Felleisen [7] and the garbage collection rules from Maraist *et al.* [28]. We wanted CBPVS to preserve the most laws possible. Since the laws of the original CBPV part of CBPVS were left unchanged, the call-by-name and call-by-value compilations to CBPV still preserve equations for CBPVS.

Theorem 5.3 (CBNeed Compilation Preserves Equations)**.** *If* $\Gamma \vdash M =_{\text{CBNeed}} N : \tau$, *then* $\underline{\Gamma} \vdash \underline{M} =_{\text{CBPVS}} \underline{N} : \underline{\Gamma}$.

Proof. By induction on the equality derivation. The cases for reflexivity, symmetry, transitivity, and compatibility follow by their inductive hypotheses and the respective rule for CBPVS. Thus, we need only show it holds for the axioms; in each case, we show that the translation of left side of the equation is equal to the translation of the right side:

Case $\beta_{\rightarrow}$:

$$(\lambda x. M)\ N = \texttt{let}\ x\ \texttt{be}\ N\ \texttt{in}\ M$$

$$\begin{aligned}
\underline{(\lambda x. M)\ N} &=_{\text{defn.}} \\
\{\texttt{enter} \rightarrow \lambda y.\, \texttt{case}\ y\ \texttt{of}\ \{\texttt{box}\ x \rightarrow \{\texttt{eval} \rightarrow \underline{M}\}\}\}\ \texttt{memo}\ a\ \texttt{in} & \\
\underline{N}\ \texttt{memo}\ b\ \texttt{in}\ (a.\texttt{enter}\ (\texttt{box}\ b)).\texttt{eval} &=_{\text{deref}}=_{\text{GC}} \\
\underline{N}\ \texttt{memo}\ b\ \texttt{in} & \\
(\{\texttt{enter} \rightarrow \lambda y.\, \texttt{case}\ y\ \texttt{of}\ \{\texttt{box}\ x \rightarrow \{\texttt{eval} \rightarrow \underline{M}\}\}\}.\texttt{enter}\ (\texttt{box}\ b)).\texttt{eval} &=_{\beta_{\check{U}}} \\
\underline{N}\ \texttt{memo}\ b\ \texttt{in}\ (\lambda y.\, \texttt{case}\ y\ \texttt{of}\ \{\texttt{box}\ x \rightarrow \{\texttt{eval} \rightarrow \underline{M}\}\}\ (\texttt{box}\ b)).\texttt{eval} &=_{\beta_{\rightarrow}} \\
\underline{N}\ \texttt{memo}\ b\ \texttt{in}\ (\texttt{case}\ (\texttt{box}\ b)\ \texttt{of}\ \{\texttt{box}\ x \rightarrow \{\texttt{eval} \rightarrow \underline{M}\}\}).\texttt{eval} &=_{\alpha} \\
\underline{N}\ \texttt{memo}\ x\ \texttt{in}\ (\texttt{case}\ (\texttt{box}\ x)\ \texttt{of}\ \{\texttt{box}\ x \rightarrow \{\texttt{eval} \rightarrow \underline{M}\}\}).\texttt{eval} &=_{\beta_{\tilde{U}}} \\
\underline{N}\ \texttt{memo}\ x\ \texttt{in}\ \{\texttt{eval} \rightarrow \underline{M}\}.\texttt{eval} &=_{\beta_{\tilde{F}}} \\
\underline{N}\ \texttt{memo}\ x\ \texttt{in}\ \underline{M} &=_{\text{defn.}} \\
\underline{\texttt{let}\ x\ \texttt{be}\ N\ \texttt{in}\ M} &
\end{aligned}$$

Case $\beta_{\otimes}$:

$$\mathsf{case}\ \langle M, N\rangle\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow L\} = \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ N\ \mathsf{in}\ L$$

$$\begin{aligned}
\underline{\mathsf{case}\ \langle M, N\rangle\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow L\}} &=_{\text{defn.}} \\
\underline{M}\ \mathsf{memo}\ a\ \mathsf{in}\ \underline{N}\ \mathsf{memo}\ b\ \mathsf{in}\ \mathsf{val}\ \langle \mathsf{box}\ a, \mathsf{box}\ b\rangle \\
\mathsf{to}\ y\ \mathsf{in}\ \mathsf{case}\ y\ \mathsf{of}\ \{\langle \mathsf{box}\ x, \ldots, \mathsf{box}\ x\rangle \longrightarrow \underline{L}\} &=_{\kappa} \\
\underline{M}\ \mathsf{memo}\ a\ \mathsf{in}\ \underline{N}\ \mathsf{memo}\ b\ \mathsf{in} \\
(\mathsf{val}\ \langle \mathsf{box}\ a, \mathsf{box}\ b\rangle\ \mathsf{to}\ y\ \mathsf{in}\ \mathsf{case}\ y\ \mathsf{of}\ \{\langle \mathsf{box}\ x, \mathsf{box}\ y\rangle \longrightarrow \underline{L}\}) &=_{\beta_{\hat{F}}} \\
\underline{M}\ \mathsf{memo}\ a\ \mathsf{in}\ \underline{N}\ \mathsf{memo}\ b\ \mathsf{in} \\
\mathsf{case}\ \langle \mathsf{box}\ a, \mathsf{box}\ b\rangle\ \mathsf{of}\ \{\langle \mathsf{box}\ x, \mathsf{box}\ y\rangle \longrightarrow \underline{L}\} &=_{\alpha} \\
\underline{M}\ \mathsf{memo}\ x\ \mathsf{in}\ \underline{N}\ \mathsf{memo}\ y\ \mathsf{in} \\
\mathsf{case}\ \langle \mathsf{box}\ x, \mathsf{box}\ y\rangle\ \mathsf{of}\ \{\langle \mathsf{box}\ x, \mathsf{box}\ y\rangle \longrightarrow \underline{L}\} &=_{\beta_{\otimes}} =_{\beta_{\check{U}}} \\
\underline{M}\ \mathsf{memo}\ x\ \mathsf{in}\ \underline{N}\ \mathsf{memo}\ y\ \mathsf{in}\ \underline{L} &=_{\text{defn.}} \\
\underline{\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ N\ \mathsf{in}\ L} &
\end{aligned}$$

Case $\eta_{\rightarrow}$:

$$\lambda x.\, V\ x = V$$

$$
\begin{aligned}
&\underline{\lambda x.\, V\ x} &&=_{\text{defn.}}\\
&\left\{ \mathsf{enter} \to \lambda y.\, \mathsf{case}\ y\ \mathsf{of} \left\{ \mathsf{box}\ x \to \left\{ \mathsf{eval} \to \begin{array}{r} \underline{V}\ \mathsf{memo}\ a\ \mathsf{in}\ x\ \mathsf{memo}\ b\ \mathsf{in} \\ (a.\mathsf{enter}\ (\mathsf{box}\ b)).\mathsf{eval} \end{array} \right\} \right\} \right\} &&=_{\text{deref}}=_{\text{GC}}\\
&\left\{ \mathsf{enter} \to \lambda y.\, \mathsf{case}\ y\ \mathsf{of} \left\{ \mathsf{box}\ x \to \left\{ \mathsf{eval} \to \begin{array}{r} x\ \mathsf{memo}\ b\ \mathsf{in} \\ (\underline{V}.\mathsf{enter}\ (\mathsf{box}\ b)).\mathsf{eval} \end{array} \right\} \right\} \right\} &&=_{\text{deref}}=_{\text{GC}}\\
&\{\mathsf{enter} \to \lambda y.\, \mathsf{case}\ y\ \mathsf{of}\ \{\mathsf{box}\ x \to \{\mathsf{eval} \to (\underline{V}.\mathsf{enter}\ (\mathsf{box}\ x)).\mathsf{eval}\}\}\} &&=_{\eta_{\hat{F}}}\\
&\{\mathsf{enter} \to \lambda y.\, \mathsf{case}\ y\ \mathsf{of}\ \{\mathsf{box}\ x \to \underline{V}.\mathsf{enter}\ (\mathsf{box}\ x)\}\} &&=_{\eta_{\check{U}}}\\
&\{\mathsf{enter} \to \lambda y.\, \underline{V}.\mathsf{enter}\ y\} &&=_{\eta_{\to}}\\
&\{\mathsf{enter} \to \underline{V}.\mathsf{enter}\} &&=_{\eta_{\check{U}}}\\
&\underline{V}
\end{aligned}
$$

Case $\eta_{\otimes}$:

$$\mathsf{case}\ M\ \mathsf{of}\ \{\langle x, y\rangle \to E[\langle x, y\rangle]\} = E[M]$$

$$
\begin{aligned}
&\underline{\mathsf{case}\ M\ \mathsf{of}\ \{\langle x, y\rangle \to E[\langle x, y\rangle]\}} &&=_{\text{defn.}}\\
&\begin{array}{l} \underline{M}\ \mathsf{to}\ z\ \mathsf{in}\ \mathsf{case}\ z\ \mathsf{of} \\ \quad \{\langle \mathsf{box}\ x, \mathsf{box}\ y\rangle \to \underline{E}[x\ \mathsf{memo}\ a\ \mathsf{in}\ y\ \mathsf{memo}\ b\ \mathsf{in}\ \mathsf{val}\ \langle \mathsf{box}\ a, \mathsf{box}\ b\rangle]\} \end{array} &&(=_{\text{deref}}=_{\text{GC}})^2\\
&\underline{M}\ \mathsf{to}\ z\ \mathsf{in}\ \mathsf{case}\ z\ \mathsf{of}\ \{\langle \mathsf{box}\ x, \mathsf{box}\ y\rangle \to \underline{E}[\mathsf{val}\ \langle \mathsf{box}\ x, \mathsf{box}\ y\rangle]\} &&=_{\eta_{\otimes}}=_{\eta_{\check{U}}}\\
&\underline{M}\ \mathsf{to}\ z\ \mathsf{in}\ \underline{E}[\mathsf{val}\ z] &&=_{\eta_{\hat{F}}}\\
&\underline{E[M]}
\end{aligned}
$$

Case lift1:

$$(\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N)\ L = \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ (N\ L)$$

$$
\begin{aligned}
&\underline{(\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N)\ L} &&=_{\text{defn.}}\\
&(\underline{M}\ \mathsf{memo}\ x\ \mathsf{in}\ \underline{N})\ \mathsf{memo}\ a\ \mathsf{in}\ \underline{L}\ \mathsf{memo}\ b\ \mathsf{in}\ (a.\mathsf{enter}\ (\mathsf{box}\ b)).\mathsf{eval} &&=_{\chi}\\
&\underline{M}\ \mathsf{memo}\ x\ \mathsf{in}\ \underline{N}\ \mathsf{memo}\ a\ \mathsf{in}\ \underline{L}\ \mathsf{memo}\ b\ \mathsf{in}\ (a.\mathsf{enter}\ (\mathsf{box}\ b)).\mathsf{eval} &&=_{\text{defn.}}\\
&\underline{\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ (N\ L)}
\end{aligned}
$$

Case lift2:

$$M\ (\mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ L) = \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ (M\ L)$$

$$\begin{aligned}
\underline{M\ (\mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ L)} &=_{\text{defn.}} \\
\underline{M}\ \mathsf{memo}\ a\ \mathsf{in}\ (\underline{N}\ \mathsf{memo}\ x\ \mathsf{in}\ \underline{L})\ \mathsf{memo}\ b\ \mathsf{in}\ (a.\mathsf{enter}\ (\mathsf{box}\ b)).\mathsf{eval} &=_{\kappa} \\
\underline{N}\ \mathsf{memo}\ x\ \mathsf{in}\ \underline{M}\ \mathsf{memo}\ a\ \mathsf{in}\ \underline{L}\ \mathsf{memo}\ b\ \mathsf{in}\ (a.\mathsf{enter}\ (\mathsf{box}\ b)).\mathsf{eval} &=_{\text{defn.}} \\
\underline{\mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ (M\ L)} &
\end{aligned}$$

Case lift3:

$$\lambda x.\,\mathsf{let}\ y\ \mathsf{be}\ V\ \mathsf{in}\ M = \mathsf{let}\ y\ \mathsf{be}\ V\ \mathsf{in}\ \lambda x.\,M$$

$$\begin{aligned}
\underline{\lambda x.\,\mathsf{let}\ y\ \mathsf{be}\ V\ \mathsf{in}\ M} &=_{\text{defn.}} \\
\{\mathsf{enter} \longrightarrow \lambda z.\,\mathsf{case}\ z\ \mathsf{of}\ \{\mathsf{box}\ x \longrightarrow \{\mathsf{eval} \longrightarrow \underline{V}\ \mathsf{memo}\ y\ \mathsf{in}\ \underline{M}\}\}\} &=^{*}_{\text{deref}}=_{\text{GC}} \\
\{\mathsf{enter} \longrightarrow \lambda z.\,\mathsf{case}\ z\ \mathsf{of}\ \{\mathsf{box}\ x \longrightarrow \{\mathsf{eval} \longrightarrow \underline{M}[\underline{V}/x]\}\}\} &=_{\text{GC}}=^{*}_{\text{deref}} \\
\underline{V}\ \mathsf{memo}\ y\ \mathsf{in}\ \{\mathsf{enter} \longrightarrow \lambda z.\,\mathsf{case}\ z\ \mathsf{of}\ \{\mathsf{box}\ x \longrightarrow \{\mathsf{eval} \longrightarrow \underline{M}\}\}\} &=_{\text{defn.}} \\
\underline{\mathsf{let}\ y\ \mathsf{be}\ V\ \mathsf{in}\ \lambda x.\,M} &
\end{aligned}$$

Case lift4:

$$\begin{aligned}
&\mathsf{case}\ (\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N)\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow L\} \quad = \\
&\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ (\mathsf{case}\ N\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow L\})
\end{aligned}$$

$$\begin{aligned}
\underline{\mathsf{case}\ (\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N)\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow L\}} &=_{\text{defn.}} \\
(\underline{M}\ \mathsf{memo}\ x\ \mathsf{in}\ \underline{N})\ \mathsf{to}\ z\ \mathsf{in}\ \mathsf{case}\ z\ \mathsf{of}\ \{\langle \mathsf{box}\ x, \mathsf{box}\ y\rangle \longrightarrow \underline{L}\} &=_{\kappa} \\
\underline{M}\ \mathsf{memo}\ x\ \mathsf{in}\ \underline{N}\ \mathsf{to}\ z\ \mathsf{in}\ \mathsf{case}\ z\ \mathsf{of}\ \{\langle \mathsf{box}\ x, \mathsf{box}\ y\rangle \longrightarrow \underline{L}\} &=_{\text{defn.}} \\
\underline{\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ (\mathsf{case}\ N\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow L\})} &
\end{aligned}$$

Case merge:

$$\mathsf{let}\ x\ \mathsf{be}\ (\mathsf{let}\ y\ \mathsf{be}\ M\ \mathsf{in}\ N)\ \mathsf{in}\ L = \mathsf{let}\ y\ \mathsf{be}\ M\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ L$$

$$\begin{aligned}
\underline{\mathtt{let}\ x\ \mathtt{be}\ (\mathtt{let}\ y\ \mathtt{be}\ M\ \mathtt{in}\ N)\ \mathtt{in}\ L} &=_{\text{defn.}}\\
(\underline{M}\ \mathtt{memo}\ y\ \mathtt{in}\ \underline{N})\ \mathtt{memo}\ x\ \mathtt{in}\ \underline{L} &=_{\chi}\\
\underline{M}\ \mathtt{memo}\ y\ \mathtt{in}\ (\underline{N}\ \mathtt{memo}\ x\ \mathtt{in}\ \underline{L}) &=_{\text{defn.}}\\
\underline{\mathtt{let}\ y\ \mathtt{be}\ M\ \mathtt{in}\ \mathtt{let}\ x\ \mathtt{be}\ N\ \mathtt{in}\ L} &
\end{aligned}$$

Case deref:

$$\mathtt{let}\ x\ \mathtt{be}\ V\ \mathtt{in}\ C[x] = \mathtt{let}\ x\ \mathtt{be}\ V\ \mathtt{in}\ C[V]$$

Note that Lemma 5.2 gives us that $\underline{V} = W$.

$$\begin{aligned}
\underline{\mathtt{let}\ x\ \mathtt{be}\ V\ \mathtt{in}\ C[x]} &=_{\text{defn.}}\\
\underline{V}\ \mathtt{memo}\ x\ \mathtt{in}\ \underline{C}[x] &=_{\text{deref}}\\
\underline{V}\ \mathtt{memo}\ x\ \mathtt{in}\ \underline{C}[\underline{V}] &=_{\text{defn.}}\\
\underline{\mathtt{let}\ x\ \mathtt{be}\ V\ \mathtt{in}\ C[V]} &
\end{aligned}$$

Case GC:

$$\mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ N = N$$

$$\begin{aligned}
\underline{\mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ N} &=_{\text{defn.}}\\
\underline{M}\ \mathtt{memo}\ x\ \mathtt{in}\ \underline{N} &=_{\text{GC}}\\
\underline{N} &
\end{aligned}$$

Case name:

$$M = \mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ x$$

$$\begin{aligned}
\underline{M} &=_{\text{name}} \\
\underline{M}\ \texttt{memo}\ a\ \texttt{in}\ a &=_{\text{defn.}} \\
\underline{\texttt{let}\ a\ \texttt{be}\ M\ \texttt{in}\ a} &
\end{aligned}$$

□

CHAPTER VI

CLOSURES AND MACHINES FOR CBPVS

This chapter contains published and unpublished co-authored material. It contains revisions of the work Closure Conversion in Little Pieces [53] co-authored with Paul Downen and Zena M. Ariola. Zachary J. Sullivan is the primary author under the guidance of Paul Downen and Zena M. Ariola.

While CBPVS captures call-by-name, call-by-value, and call-by-need in a single intermediate language, we have not yet specified an operational semantics in the manner of the abstract machines presented earlier in this dissertation. Moreover, we do not yet know how to extend our intermediate language with closures that are connected to this machine. This chapter does both. To start, we design an environment abstract machine for CBPV to inform where we will need closures in such a language. Thereafter, we are able to describe where abstract closures must be included in CBPVS. We extend the CBPV machine to include sharing and show how abstract closures in the CBPVS are a superset of machine representations of closures. Finally, we begin to formally connect the calculus with the machine by proving a backwards simulation and defining observational equivalence of CBPVS expressions.

6.1 An Environment Machine for CBPV

The first place to start when designing an environment abstract machine for CBPV is discovering where closures are required. In Section 1.4, we saw that call-by-value required closures for functions alone, whereas call-by-name and call-by-need required closures for arguments of functions. In general, closures need to save the values that would otherwise be substituted by the operational semantics and that occur in reducible code. In CBPV, this quality is dictated by the syntactic categories: only values have the

$$\begin{array}{rcl}
\mathit{Conf} \in & \mathit{Configuration} & ::= \langle\!\langle \Sigma \parallel M \parallel K \rangle\!\rangle \\
\Sigma \in & \mathit{Machine\ Environment} & ::= \varepsilon \mid \Sigma, \mathbb{V}/x \\
\mathbb{V}, \mathbb{W} \in & \mathit{Machine\ Value} & ::= \mathtt{b} \mid \langle \mathbb{V}, \mathbb{W} \rangle \mid (\Sigma, \{\mathtt{force} \to M\}) \\
K \in & \mathit{Stack} & ::= \star \mid F \cdot K \\
F \in & \mathit{Frame} & ::= \square\ \mathbb{V} \mid \square.\mathtt{fst} \mid \square.\mathtt{snd} \mid (\Sigma, \square\ \mathtt{to}\ x\ \mathtt{in}\ M)
\end{array}$$

Figure 6.1. CBPV Machine Syntax

potential to be bound to variables and passed to other parts of the program. Indeed, the only location where these expressions contain unevaluated code is in the shift from computations: $\{\mathtt{force} \to M\}$. Thus, this is where we must add closures. Surprisingly, we do not need closures for functions. A function is a computation and therefore is never bound to a variable unless it is shifted to a value. Compiling a call-by-value program will put the function into a thunk and *that* instead is where the free variables need to be captured. Compiling a call-by-name program will put a function argument into a thunk, thus creating a closure in the same location as we would've seen when running the program on the Krivine machine.

Our machine is essentially Levy's CK-machine [27] augmented with an environment that delays substitutions. Its syntax is presented in Figure 6.1. Machine environments Σ are local, *i.e.* they may disappear when an intermediate result is returned, which is why we use closures. The continuation part of the machine is a list of stack frames which for the most part are evaluation contexts. Exceptionally, the to-expression frame $(\Sigma, \square\ \mathtt{to}\ x\ \mathtt{in}\ M)$ also contains a local environment to re-instantiate when we evaluate M after an intermediate result is returned. Such a frame may be implemented using stack pointers in a C-like runtime; that is, returning to one of these saved environments is simply moving the stack frame back to that location.

The machine uses machine values instead of the ones in the full equational theory. Syntactic values can contain variables, but machine ones only refer to objects that can

$$\begin{array}{rcl}
\mathrm{Build}_V & : & \mathit{Machine\ Environment} \times \mathit{Value} \longrightarrow \mathit{Machine\ Value} \\
\mathrm{Build}_V(\Sigma, x) & = & x[\Sigma] \\
\mathrm{Build}_V(\Sigma, \mathtt{b}) & = & \mathtt{b} \\
\mathrm{Build}_V(\Sigma, \langle V, W\rangle) & = & \langle \mathrm{Build}_V(\Sigma, V), \mathrm{Build}_V(\Sigma, W)\rangle \\
\mathrm{Build}_V(\Sigma, \{\mathtt{force} \rightarrow M\}) & = & (\Sigma, \{\mathtt{force} \rightarrow M\})
\end{array}$$

Figure 6.2. Building CBPV Machine Values

$$\begin{array}{rcl}
\langle\!\langle \Sigma \parallel \mathtt{case}\ V\ \mathtt{of}\ \{\langle x, y\rangle \rightarrow M\} \parallel K \rangle\!\rangle & \longmapsto_1 & \langle\!\langle \Sigma, \mathbb{W}/x, \mathbb{W}'/y \parallel M \parallel K \rangle\!\rangle \\
\multicolumn{3}{c}{\textbf{where } \mathrm{Build}_V(\Sigma, V) = \langle \mathbb{W}, \mathbb{W}'\rangle} \\
\langle\!\langle \Sigma \parallel M\ \mathtt{to}\ x\ \mathtt{in}\ N \parallel K \rangle\!\rangle & \longmapsto_2 & \langle\!\langle \Sigma \parallel M \parallel (\Sigma, \square\ \mathtt{to}\ x\ \mathtt{in}\ N) \cdot K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \mathtt{ret}\ V \parallel (\Sigma', \square\ \mathtt{to}\ x\ \mathtt{in}\ M) \cdot K \rangle\!\rangle & \longmapsto_3 & \langle\!\langle \Sigma', \mathrm{Build}_V(\Sigma, V)/x \parallel M \parallel K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel M.\mathtt{fst} \parallel K \rangle\!\rangle & \longmapsto_4 & \langle\!\langle \Sigma \parallel M \parallel \square.\mathtt{fst} \cdot K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel M.\mathtt{snd} \parallel K \rangle\!\rangle & \longmapsto_5 & \langle\!\langle \Sigma \parallel M \parallel \square.\mathtt{snd} \cdot K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \{\mathtt{fst} \rightarrow M; \mathtt{snd} \rightarrow N\} \parallel \square.\mathtt{fst} \cdot K \rangle\!\rangle & \longmapsto_6 & \langle\!\langle \Sigma \parallel M \parallel K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \{\mathtt{fst} \rightarrow M; \mathtt{snd} \rightarrow N\} \parallel \square.\mathtt{snd} \cdot K \rangle\!\rangle & \longmapsto_7 & \langle\!\langle \Sigma \parallel N \parallel K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel M\ V \parallel K \rangle\!\rangle & \longmapsto_8 & \langle\!\langle \Sigma \parallel M \parallel \square\ \mathrm{Build}_V(\Sigma, V) \cdot K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \lambda x.\, M \parallel \square\ \mathbb{V} \cdot K \rangle\!\rangle & \longmapsto_9 & \langle\!\langle \Sigma, \mathbb{V}/x \parallel M \parallel K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel V.\mathtt{force} \parallel K \rangle\!\rangle & \longmapsto_{10} & \langle\!\langle \Sigma' \parallel M \parallel K \rangle\!\rangle \\
\multicolumn{3}{c}{\textbf{where } \mathrm{Build}_V(\Sigma, V) = (\Sigma', \{\mathtt{force} \rightarrow M\})}
\end{array}$$

Figure 6.3. CBPV Machine Transitions

be pattern matched or forced; indeed, values are a superset of machine values and thus environments are a superset of machine environments as well. To transform between syntactic values and machine values, we must apply the delayed substitution manipulated by the machine; this is given by the build rules in Figure 6.2. In most cases, this is a standard substitution application; however, the case for force-expressions is different since they contain unevaluated code. To generate a fixed sequence of code for the body, building a machine value cannot perform a substitution on it. Therefore, we capture the current machine environment Σ in a closure.

The evaluation transitions are given in Figure 6.3. Since in CBPV values *are* and computations *do*, there are only rules for evaluating computations. Values, on the other hand, are built from the local environment when needed. In a manner similar to the Krivine machine, when evaluating a function application, we push the argument on the call stack (capturing a closure if necessary) and jump into the function body; when a λ-expression is encountered, we can add that machine value to the local environment. The to-expression and projection-expressions operate in a similar manner by pushing a stack frame, which will be consumed when an introduction form of the computation is reached. Note that anywhere Build is called is a location where our runtime system may have to create a closure. Closures are entered only when we force a value in the transition 10.

6.2 CBPVS with Closures

In the CBPV environment machine, we must create closures force-expressions since they delay unevaluated code within a substitutable variable. Similarly, the CBPVS enter-expression delays a computation within a shared computation may be substituted and thus will need to be a closure. Additionally, since the memo-expressions of CBPVS will behave in the same manner as call-by-need let-expressions, we will have closures like $\{\varsigma, R\}\ \mathtt{memo}\ a\ \mathtt{in}\ P$. The new syntax for CBPVS with closures is given in Figure 6.4. We merely replace force-, enter-, and memo-expressions with forms that contain an environment. Note that environments now contain a mixture of values and shared values. As with abstract closures from Section 3.3.4, there is syntactic sugar for when the environment of a closure is empty: $\{\mathtt{force} \to M\}$, $\{\mathtt{enter} \to M\}$, and $R\ \mathtt{memo}\ a\ \mathtt{in}\ P$.

The typing rules for the new closure forms Figure 6.4c follow the same pattern as those that we saw in Chapter 4: the local environment $\varsigma : \Gamma'$ extends the current type environment Γ for the body of the closure. All of the other typing rules for CBPVS remain unchanged.

The closure axioms for CBPVS are presented in Figure 6.4d. We have only new β-laws as the η-laws remain unchanged. The β laws for force- and enter-closures work similar to

$$
\begin{array}{rcll}
V, W \in & \textit{Value} & ::= & \mathsf{b} \mid x \mid \langle V, W \rangle \mid \{\varsigma, \mathsf{force} \longrightarrow M\} \mid \mathsf{box}\ V \\
V, W \in & \textit{Shared Value} & ::= & a \mid \mathsf{val}\ V \mid \{\varsigma, \mathsf{enter} \longrightarrow M\} \\
R, S \in & \textit{Shared Comp.} & ::= & V \mid M.\mathsf{eval} \mid B[R] \\
M, N \in & \textit{Comp.} & ::= & \{\mathsf{fst} \longrightarrow M; \mathsf{snd} \longrightarrow N\} \mid M.\mathsf{fst} \mid M.\mathsf{snd} \mid \lambda x.M \mid M\,V \\
 & & & \mid V.\mathsf{force} \mid \mathsf{ret}\ V \mid B[M] \mid R.\mathsf{enter} \mid \{\mathsf{eval} \longrightarrow R\} \\
B \in & \textit{Block Ctxt.} & ::= & P\ \mathsf{to}\ x\ \mathsf{in}\ \square \mid \{\varsigma, R\}\ \mathsf{memo}\ a\ \mathsf{in}\ \square \\
 & & & \mid \mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y \rangle \longrightarrow \square\} \mid \mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{box}\ a \longrightarrow \square\} \\
P, Q \in & \textit{Comp. Expr.} & ::= & R \mid M \\
\varsigma \in & \textit{Env.} & ::= & \varepsilon \mid \varsigma, V/x \mid \varsigma, V/a
\end{array}
$$

(a) Syntax

$$
\begin{array}{rcl}
\{\mathsf{force} \longrightarrow M\} & = & \{\varepsilon, \mathsf{force} \longrightarrow M\} \\
\{\mathsf{enter} \longrightarrow M\} & = & \{\varepsilon, \mathsf{enter} \longrightarrow M\} \\
R\ \mathsf{memo}\ a\ \mathsf{in}\ P & = & \{\varepsilon, R\}\ \mathsf{memo}\ a\ \mathsf{in}\ P
\end{array}
$$

(b) Syntactic Sugar

$$
\frac{\Gamma \vdash \varsigma : \Gamma' \quad \Gamma\Gamma' \vdash M : \tau}{\Gamma \vdash \{\varsigma, \mathsf{force} \longrightarrow M\} : U\ \tau}U_I
\qquad
\frac{\Gamma \vdash \varsigma : \Gamma' \quad \Gamma\Gamma' \vdash M : \tau}{\Gamma \vdash \{\varsigma, \mathsf{enter} \longrightarrow M\} : \check{U}\ \tau}\check{U}_I
$$

$$
\frac{\Gamma \vdash \varsigma : \Gamma' \quad \Gamma\Gamma' \vdash R : \sigma \quad \Gamma, a{:}\sigma \vdash P : \tau}{\Gamma \vdash \{\varsigma, R\}\ \mathsf{memo}\ a\ \mathsf{in}\ P : \tau}H
\qquad
\frac{}{\Gamma \vdash \varepsilon : \varepsilon}\Gamma_{I_B}
$$

$$
\frac{\Gamma \vdash \varsigma : \Gamma' \quad \Gamma \vdash V : \tau}{\Gamma \vdash (\varsigma, V/x) : (\Gamma', x{:}\tau)}\Gamma_{I1}
\qquad
\frac{\Gamma \vdash \varsigma : \Gamma' \quad \Gamma \vdash V : \tau}{\Gamma \vdash (\varsigma, V/a) : (\Gamma', a{:}\tau)}\Gamma_{I2}
$$

(c) Closure Typing Rules

$$
\begin{array}{rcl}
\{\varsigma, \mathsf{force} \longrightarrow M\}.\mathsf{force} & =_{\beta_U} & M[\varsigma] \\
\{\varsigma, \mathsf{enter} \longrightarrow M\}.\mathsf{enter} & =_{\beta_{\check{U}}} & M[\varsigma] \\
\{\varsigma, R\}\ \mathsf{memo}\ a\ \mathsf{in}\ P & =_{cl} & R[\varsigma]\ \mathsf{memo}\ a\ \mathsf{in}\ P
\end{array}
$$

$$
\begin{array}{lll}
\textbf{where} & E, F & ::= \square \mid E\ V \mid E.\mathsf{fst} \mid E.\mathsf{snd} \mid E\ \mathsf{to}\ x\ \mathsf{in}\ P \\
 & & \quad \mid E.\mathsf{enter} \mid E.\mathsf{eval} \mid \{\varsigma, R\}\ \mathsf{memo}\ a\ \mathsf{in}\ E \mid \{\varsigma, E\}\ \mathsf{memo}\ a\ \mathsf{in}\ F[a]
\end{array}
$$

(d) Closure Axioms

Figure 6.4. CBPVS with Closures

the call-by-value function closures from Chapter 4: the delayed environment is substituted as part of the β axiom. On the other hand, the *cl* law for the memo-closures, which is in addition to the memoization laws of CBPVS, allows the closure to be entered at any time like the call-by-name argument closures and the call-by-need memo-closures. Using the syntactic sugar, we see that the memo-expression laws are all restricted to the case where the environment is empty; this is sufficient to subsume call-by-need and perform closure conversion in little pieces.

As in Section 4.3, we can derive a naïve closure conversion from the equational theory. For each of the three closure locations, we have a similar rule for incrementally adding free variables:

$$\frac{x \in \mathrm{FV}(M) - \mathrm{Dom}(\varsigma)}{\{\varsigma, \texttt{force} \to M\} \longrightarrow_{\mathrm{CC}} \{(\varsigma, x/x), \texttt{force} \to M\}}$$

$$\frac{x \in \mathrm{FV}(M) - \mathrm{Dom}(\varsigma)}{\{\varsigma, \texttt{enter} \to M\} \longrightarrow_{\mathrm{CC}} \{(\varsigma, x/x), \texttt{enter} \to M\}}$$

$$\frac{x \in \mathrm{FV}(R) - \mathrm{Dom}(\varsigma)}{\{\varsigma, R\}\ \texttt{memo}\ a\ \texttt{in}\ P \longrightarrow_{\mathrm{CC}} \{(\varsigma, x/x), R\}\ \texttt{memo}\ a\ \texttt{in}\ P}$$

Note that we use black V and x here for values and variables that may be shared or not.

6.3 The CBPVS Machine

Extending the CBPV environment machine to handle shared expressions requires a heap to manage memoization and extra rules for the shared expressions. We also need to support abstract closures within the machine itself. Figure 6.5 presents the syntax for this machine. Since they are a reflection of the runtime closures, *the abstract closures from our equational theory are a superset of the machine closures used by the abstract machines.* Heaps are mappings from labels to closures, which will include both unevaluated and evaluated shared expressions. Machine environments are extended

$$
\begin{array}{rcl}
\mathit{Conf} \in & \textit{Configuration} & ::= \langle\!\langle \Phi \parallel \Sigma \parallel P \parallel K \rangle\!\rangle \\
\Phi \in & \textit{Heap} & ::= \varepsilon \mid \Phi, l \mapsto \{\Sigma, R\} \\
\mathbb{I} \in & \textit{Machine Shared Intro.} & ::= \mathtt{val}\ \mathbb{V} \mid \{\Sigma, \mathtt{enter} \longrightarrow M\} \\
\mathbb{V}, \mathbb{W} \in & \textit{Machine Shared Value} & ::= l \mid \mathbb{I} \\
\Sigma \in & \textit{Machine Environment} & ::= \varepsilon \mid \Sigma, \mathbb{V}/x \mid \Sigma, \mathbb{V}/a \\
\mathbb{V}, \mathbb{W} \in & \textit{Machine Value} & ::= b \mid \langle \mathbb{V}, \mathbb{W} \rangle \mid \{\Sigma, \mathtt{force} \longrightarrow M\} \\
 & & \quad \mid \mathtt{box}\ \mathbb{V} \\
K \in & \textit{Stack} & ::= \star \mid F \cdot K \\
F \in & \textit{Frame} & ::= \square\ \mathbb{V} \mid \square.\mathtt{fst} \mid \square.\mathtt{snd} \\
 & & \quad \mid (\Sigma, \square\ \mathtt{to}\ x\ \mathtt{in}\ P) \mid (\Sigma, \square\ \mathtt{to}\ x\ \mathtt{in}\ P) \\
 & & \quad \mid \square.\mathtt{enter} \mid \square.\mathtt{eval} \mid (\Phi, l)
\end{array}
$$

Figure 6.5. CBPVS Machine Syntax

to include substitutions of shared variables to either machine-shared introductions or pointers to memoizable heap objects. Machine-shared introductions are the shared expressions in the machine that may be safely duplicated. Stack frames are extended to include evaluation contexts for the new shifts and a memoization frame (Φ, l), which corresponds to the evaluation context E `memo` a `in` $F[a]$.

Figure 6.6 gives new building definitions extending the previous definitions to include box-expressions, environments that include shared values, and adding in a definition for building machine-shared values and heap objects. For the closure cases now, we can make use of the abstract closure objects since they are runtime objects. We return the whole machine environment as well as build the syntactic environment. Therefore, it is possible that we have duplicated bindings when capturing the closure. This version of the rule is necessary for our abstract machine to accept programs both before and after a closure conversion. However, after closure conversion we will not need anything from the existing machine environment that is not specified in the closure (Theorem 7.2).

Figure 6.7 specifies the additional machine transitions for the sharing extension while making use of all of the rules from the CBPV machine. We have divided it into the additional rules that do not manipulate the heap and the ones that do. A memo-expression

$$
\begin{aligned}
\mathrm{Build}_{V} &: \mathit{Machine\ Env.} \times \mathit{Value} \longrightarrow \mathit{Machine\ Value} \\
\mathrm{Build}_{V}(\Sigma, \mathsf{b}) &= \mathsf{b} \\
\mathrm{Build}_{V}(\Sigma, x) &= x[\Sigma] \\
\mathrm{Build}_{V}(\Sigma, \langle V, W\rangle) &= \langle \mathrm{Build}_{V}(\Sigma, V), \mathrm{Build}_{V}(\Sigma, W)\rangle \\
\mathrm{Build}_{V}(\Sigma, \{\varsigma, \mathtt{force} \to M\}) &= \{\Sigma\ \mathrm{Build}_{\varsigma}(\Sigma, \varsigma), \mathtt{force} \to M\} \\
\mathrm{Build}_{V}(\Sigma, \mathtt{box}\ V) &= \mathrm{Build}_{V}(\Sigma, V) \\[1em]
\mathrm{Build}_{\varsigma} &: \mathit{Mach.\ Env.} \times \mathit{Env.} \longrightarrow \mathit{Mach.\ Env.} \\
\mathrm{Build}_{\varsigma}(\Sigma, \varepsilon) &= \varepsilon \\
\mathrm{Build}_{\varsigma}(\Sigma, (\varsigma, V/x)) &= \mathrm{Build}_{\varsigma}(\Sigma, \varsigma), \mathrm{Build}_{V}(\Sigma, V)/x \\
\mathrm{Build}_{\varsigma}(\Sigma, (\varsigma, V/a)) &= \mathrm{Build}_{\varsigma}(\Sigma, \varsigma), \mathrm{Build}_{V}(\Sigma, V)/a \\[1em]
\mathrm{Build}_{V} &: \mathit{Mach.\ Env.} \times \mathit{Shared\ Value} \longrightarrow \mathit{Mach.\ Shared\ Value} \\
\mathrm{Build}_{V}(\Sigma, a) &= a[\Sigma] \\
\mathrm{Build}_{V}(\Sigma, \mathtt{val}\ V) &= \mathtt{val}\ \mathrm{Build}_{V}(\Sigma, V) \\
\mathrm{Build}_{V}(\Sigma, \{\varsigma, \mathtt{enter} \to M\}) &= \{\Sigma\ \mathrm{Build}_{\varsigma}(\Sigma, \varsigma), \mathtt{enter} \to M\} \\[1em]
\mathrm{Build}_{a} &: \mathit{Mach.\ Env.} \times \{\{\varsigma, R\}\} \to \{\{\Sigma, R\}\} \\
\mathrm{Build}_{a}(\Sigma, \{\varsigma, R\}) &= \{\Sigma\ \mathrm{Build}_{\varsigma}(\Sigma, \varsigma), R\}
\end{aligned}
$$

Figure 6.6. Building CBPVS Machine Values and Heap Objects

will build a heap object with the Build_{a} rules before evaluating the body. When a shared variable is evaluated *and* it points to a heap object, then a memoization frame is added and the closure it points to is evaluated. Otherwise, the shared variable will point to a machine-shared value that is in the local environment, which is returned. Memoization frames are consumed when evaluating a shared expression that may be built into a machine introduction; in that case, the built object is added to the reconstructed heap.

Definition 6.1 (CBPVS Evaluator)**.** $\mathrm{EvalS}(P) = \mathsf{b}$ *where* $\langle\!\langle \varepsilon \parallel \varepsilon \parallel P \parallel \star \rangle\!\rangle \longmapsto^{*} \langle\!\langle \Phi \parallel \Sigma \parallel \mathtt{ret}\ \mathsf{b} \parallel \star \rangle\!\rangle$.

6.4 Backwards Simulation

Taking steps in our environment machine reflects to equalities in our calculus. This we prove by first defining a decoding of configurations to expressions (Figure 6.8) and then show that every transition in both abstract machines corresponds to a derivable

$$\begin{array}{rcl}
\langle\!\langle \Sigma \parallel \mathtt{case}\ V\ \mathtt{of}\ \{\langle x, y\rangle \to R\} \parallel K\rangle\!\rangle & \longmapsto_{11} & \langle\!\langle \Sigma, \mathbb{W}/x, \mathbb{W}'/y \parallel R \parallel K\rangle\!\rangle \\
& \textbf{where}\ \mathrm{Build}_V(\Sigma, V) = \langle \mathbb{W}, \mathbb{W}'\rangle & \\
\langle\!\langle \Sigma \parallel \mathtt{case}\ V\ \mathtt{of}\ \{\mathtt{box}\ a \to P\} \parallel K\rangle\!\rangle & \longmapsto_{12} & \langle\!\langle \Sigma, \mathbb{V}/a \parallel P \parallel K\rangle\!\rangle \\
& \textbf{where}\ \mathrm{Build}_V(\Sigma, V) = \mathtt{box}\ \mathbb{V} & \\
\langle\!\langle \Sigma \parallel P\ \mathtt{to}\ x\ \mathtt{in}\ Q \parallel K\rangle\!\rangle & \longmapsto_{13} & \langle\!\langle \Sigma \parallel P \parallel (\Sigma, \square\ \mathtt{to}\ x\ \mathtt{in}\ Q) \cdot K\rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \mathtt{ret}\ V \parallel (\Sigma', \square\ \mathtt{to}\ x\ \mathtt{in}\ P) \cdot K\rangle\!\rangle & \longmapsto_{14} & \langle\!\langle \Sigma', \mathrm{Build}_V(\Sigma, V)/x \parallel P \parallel K\rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \mathtt{val}\ V \parallel (\Sigma', \square\ \mathtt{to}\ x\ \mathtt{in}\ P) \cdot K\rangle\!\rangle & \longmapsto_{15} & \langle\!\langle \Sigma', \mathrm{Build}_V(\Sigma, V)/x \parallel P \parallel K\rangle\!\rangle \\
\langle\!\langle \Sigma \parallel R.\mathtt{enter} \parallel K\rangle\!\rangle & \longmapsto_{16} & \langle\!\langle \Sigma \parallel R \parallel \square.\mathtt{enter} \cdot K\rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \{\varsigma, \mathtt{enter} \to M\} \parallel \square.\mathtt{enter} \cdot K\rangle\!\rangle & \longmapsto_{17} & \langle\!\langle \Sigma' \parallel M \parallel K\rangle\!\rangle \\
& \textbf{where}\ \mathrm{Build}_V(\Sigma, V) = \{\Sigma', \mathtt{enter} \to M\} & \\
\langle\!\langle \Sigma \parallel M.\mathtt{eval} \parallel K\rangle\!\rangle & \longmapsto_{18} & \langle\!\langle \Sigma \parallel M \parallel \square.\mathtt{eval} \cdot K\rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \{\mathtt{eval} \to R\} \parallel \square.\mathtt{eval} \cdot K\rangle\!\rangle & \longmapsto_{19} & \langle\!\langle \Sigma \parallel R \parallel K\rangle\!\rangle \\
\langle\!\langle \Sigma \parallel a \parallel K\rangle\!\rangle & \longmapsto_{20} & \langle\!\langle \varepsilon \parallel \mathbb{I} \parallel K\rangle\!\rangle \\
& \textbf{where}\ a[\Sigma] = \mathbb{I} &
\end{array}$$

(a) Additional Stateless Transitions

$$\frac{\langle\!\langle \Sigma \parallel P \parallel K\rangle\!\rangle \longmapsto \langle\!\langle \Sigma' \parallel P' \parallel K'\rangle\!\rangle}{\langle\!\langle \Phi \parallel \Sigma \parallel P \parallel K\rangle\!\rangle \longmapsto_{21} \langle\!\langle \Phi \parallel \Sigma' \parallel P' \parallel K'\rangle\!\rangle}$$

$$\begin{array}{rcl}
\langle\!\langle \Phi \parallel \Sigma \parallel \{\varsigma, R\}\ \mathtt{memo}\ a\ \mathtt{in}\ P \parallel K\rangle\!\rangle & \longmapsto_{22} & \langle\!\langle \Phi, l \mapsto \mathrm{Build}_a(\Sigma, \{\varsigma, R\}) \parallel \Sigma, l/a \parallel P \parallel K\rangle\!\rangle \\
\langle\!\langle (\Phi_0, a[\Sigma] \mapsto \{\Sigma', R\})\Phi_1 \parallel \Sigma \parallel a \parallel K\rangle\!\rangle & \longmapsto_{23} & \langle\!\langle \Phi_0 \parallel \Sigma' \parallel R \parallel (\Phi_1, l) \cdot K\rangle\!\rangle \\
\langle\!\langle \Phi \parallel \Sigma \parallel V \parallel (\Phi', l) \cdot K\rangle\!\rangle & \longmapsto_{24} & \langle\!\langle (\Phi, l \mapsto \{\varepsilon, \mathbb{I}\})\Phi' \parallel \Sigma \parallel V \parallel K\rangle\!\rangle \\
& \textbf{where}\ \mathrm{Build}_V(\Sigma, V) = \mathbb{I} &
\end{array}$$

(b) Stateful Transitions

Figure 6.7. CBPVS Machine Transitions

$$
\begin{array}{rcl}
\underline{\langle\!\langle \Phi \parallel \Sigma \parallel P \parallel K \rangle\!\rangle} & = & \underline{\Phi}[\underline{\langle\!\langle \Sigma \parallel P \parallel K \rangle\!\rangle}] \\
\underline{\langle\!\langle \Sigma \parallel P \parallel K \rangle\!\rangle} & = & \underline{K}[P[\Sigma]] \\
\underline{\star} & = & \square \\
\underline{F \cdot K} & = & \underline{K}[\underline{F}] \\
\underline{\Phi, l \mapsto \{\Sigma, R\}} & = & \underline{\Phi}[\{\Sigma, R\}\ \mathtt{memo}\ l\ \mathtt{in}\ \square] \\
\underline{\varepsilon} & = & \square \\
\underline{\square\ \mathbb{V}} & = & \square\ \mathbb{V} \\
\underline{\square\mathtt{.fst}} & = & \square\mathtt{.fst} \\
\underline{\square\mathtt{.snd}} & = & \square\mathtt{.snd} \\
\underline{(\Sigma, \square\ \mathtt{to}\ x\ \mathtt{in}\ P)} & = & (\square\ \mathtt{to}\ x\ \mathtt{in}\ P)[\Sigma] \\
\underline{\square\mathtt{.enter}} & = & \square\mathtt{.enter} \\
\underline{\square\mathtt{.eval}} & = & \square\mathtt{.eval} \\
\underline{(\Phi, l)} & = & \square\ \mathtt{memo}\ l\ \mathtt{in}\ \underline{\Phi}[l]
\end{array}
$$

Figure 6.8. CBPVS Machine Decoding

equality. Since machine values are included in values and machine environments included in environments, the decoding is a simple unfolding of the stack followed by applying the delayed substitution of the configuration. There are two properties essential to proving backwards simulation. First, the decoding of a machine configuration must be well typed because we must use η in order equate the different closure forms of type $U\ \tau$. Second, the decoding of any stack is an evaluation context; we must be able to commute heaps with stacks via the κ law of memo-expressions.

Lemma 6.1 (Built Values equal Applied Substitutions)**.**

$$\Gamma \vdash \mathrm{Build}_V(\Sigma, V) =_{\mathrm{CBPV}} V[\Sigma] : \tau$$

Proof. Follows by induction on V. For x, b, and $\langle W, W' \rangle$, the definition of substitution and Build_V are identical; thus, built values are equal to substituted values by reflexivity in the equational theory and the inductive hypotheses in the case for products. For the final case, we have

$$\begin{aligned}
\mathrm{Build}_V(\Sigma, \{\varsigma, \mathtt{force} \to M\}) &=_{\text{defn.}} \\
\{\Sigma\,\mathrm{Build}_\varsigma(\Sigma, \varsigma), \mathtt{force} \to M\} &=_{\text{subst.}} \\
\{\Sigma\,\mathrm{Build}_\varsigma(\Sigma, \varsigma), \mathtt{force} \to M[\varsigma_{\text{id.}}]\} &=_{\beta_U} \\
\{\Sigma\,\mathrm{Build}_\varsigma(\Sigma, \varsigma), \mathtt{force} \to \{\varsigma_{\text{id.}}, \mathtt{force} \to M\}.\mathtt{force}\} &=_{\eta_U} \\
\{\mathrm{Build}_\varsigma(\Sigma, \varsigma), \mathtt{force} \to M[\Sigma]\} &=_{\text{I.H.}} \\
\{\varsigma[\Sigma], \mathtt{force} \to M[\Sigma]\} &=_{\text{subst.}} \\
\{\varsigma, \mathtt{force} \to M\}[\Sigma] &
\end{aligned}$$

□

Theorem 6.1 (Backward Simulation). *If $\Gamma \vdash \underline{Conf} : \tau$ and $Conf \longmapsto Conf'$, then $\Gamma \vdash \underline{Conf} =_{\text{CBPV}} \underline{Conf'} : \tau$.*

Proof. Follows by the cases of the machine transitions:

Case:

$$\begin{aligned}
\langle\!\langle \Sigma \parallel \mathtt{case}\ V\ \mathtt{of}\ \{\langle x, y\rangle \to M\} \parallel K \rangle\!\rangle &\longmapsto_1 \\
\langle\!\langle \Sigma, \mathbb{W}/x, \mathbb{W}'/y \parallel M \parallel K \rangle\!\rangle &
\end{aligned}$$

where $\mathrm{Build}_V(\Sigma, V) = \langle \mathbb{W}, \mathbb{W}'\rangle$.

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel \mathtt{case}\ V\ \mathtt{of}\ \{\langle x, y\rangle \to M\} \parallel K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[\mathtt{case}\ V[\Sigma]\ \mathtt{of}\ (\{\langle x, y\rangle \to M\}[\Sigma])] &=_{\text{Lemma 6.1}} \\
\underline{K}[\mathtt{case}\ \langle \mathbb{W}, \mathbb{W}'\rangle\ \mathtt{of}\ (\{\langle x, y\rangle \to M\}[\Sigma])] &=_{\beta_\otimes} \\
\underline{K}[M[\Sigma, \mathbb{W}/x, \mathbb{W}'/y]] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma, \mathbb{W}/x, \mathbb{W}'/y \parallel M \parallel K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\langle\!\langle \Sigma \parallel M\ \mathsf{to}\ x\ \mathsf{in}\ N \parallel K \rangle\!\rangle \longmapsto_2 \langle\!\langle \Sigma \parallel M \parallel (\Sigma, \square\ \mathsf{to}\ x\ \mathsf{in}\ N) \cdot K \rangle\!\rangle$$

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel M\ \mathsf{to}\ x\ \mathsf{in}\ N \parallel K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[(M\ \mathsf{to}\ x\ \mathsf{in}\ N)[\Sigma]] &=_{\text{subst.}} \\
\underline{K}[M[\Sigma]\ (\mathsf{to}\ x\ \mathsf{in}\ N)[\Sigma]] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma \parallel M \parallel (\Sigma, \square\ \mathsf{to}\ x\ \mathsf{in}\ N) \cdot K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\langle\!\langle \Sigma \parallel \mathsf{ret}\ V \parallel (\Sigma', \square\ \mathsf{to}\ x\ \mathsf{in}\ M) \cdot K \rangle\!\rangle \quad \longmapsto_3$$
$$\langle\!\langle \Sigma', \mathrm{Build}_V(\Sigma, V)/x \parallel M \parallel K \rangle\!\rangle$$

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel \mathsf{ret}\ V \parallel (\Sigma', \square\ \mathsf{to}\ x\ \mathsf{in}\ M) \cdot K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[(\mathsf{ret}\ (V[\Sigma]))\ (\mathsf{to}\ x\ \mathsf{in}\ M)[\Sigma']] &=_{\text{Lemma 6.1}} \\
\underline{K}[(\mathsf{ret}\ \mathrm{Build}_V(\Sigma, V))\ (\mathsf{to}\ x\ \mathsf{in}\ M)[\Sigma']] &=_{\beta_F} \\
\underline{K}[M[\Sigma', \mathrm{Build}_V(\Sigma, V)/x]] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma', \mathrm{Build}_V(\Sigma, V)/x \parallel M \parallel K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\langle\!\langle \Sigma \parallel M.\mathsf{fst} \parallel K \rangle\!\rangle \longmapsto_4 \langle\!\langle \Sigma \parallel M \parallel \square.\mathsf{fst} \cdot K \rangle\!\rangle$$

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel M.\mathsf{fst} \parallel K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[(M.\mathsf{fst})[\Sigma]] &=_{\text{subst.}} \\
\underline{K}[(M[\Sigma]).\mathsf{fst}] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma \parallel M \parallel \square.\mathsf{fst} \cdot K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\langle\!\langle \Sigma \parallel M.\mathsf{snd} \parallel K \rangle\!\rangle \longmapsto_5 \langle\!\langle \Sigma \parallel M \parallel \square.\mathsf{snd} \cdot K \rangle\!\rangle$$

Follows in a similar manner to the case above.

Case:

$$\langle\!\langle \Sigma \parallel \{\mathtt{fst} \rightarrow M; \mathsf{snd} \rightarrow N\} \parallel \square.\mathtt{fst} \cdot K \rangle\!\rangle \longmapsto_6 \langle\!\langle \Sigma \parallel M \parallel K \rangle\!\rangle$$

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel \{\mathtt{fst} \rightarrow M; \mathsf{snd} \rightarrow N\} \parallel \square.\mathtt{fst} \cdot K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[(\{\mathtt{fst} \rightarrow M; \mathsf{snd} \rightarrow N\}[\Sigma]).\mathtt{fst}] &=_{\text{subst.}} \\
\underline{K}[\{\mathtt{fst} \rightarrow M[\Sigma]; \mathsf{snd} \rightarrow N[\Sigma]\}.\mathtt{fst}] &=_{\beta_{\&1}} \\
\underline{K}[M[\Sigma]] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma \parallel M \parallel K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\langle\!\langle \Sigma \parallel \{\mathtt{fst} \rightarrow M; \mathsf{snd} \rightarrow N\} \parallel \square.\mathsf{snd} \cdot K \rangle\!\rangle \longmapsto_7 \langle\!\langle \Sigma \parallel N \parallel K \rangle\!\rangle$$

Follows in a similar manner to the case above.

Case:

$$\langle\!\langle \Sigma \parallel M\ V \parallel K \rangle\!\rangle \longmapsto_8 \langle\!\langle \Sigma \parallel M \parallel \square\ \mathrm{Build}_V(\Sigma, V) \cdot K \rangle\!\rangle$$

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel M\ V \parallel K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[(M\ V)[\Sigma]] &=_{\text{subst.}} \\
\underline{K}[M[\Sigma]\ V[\Sigma]] &=_{\text{Lemma 6.1}} \\
\underline{K}[M[\Sigma]\ \mathrm{Build}_V(\Sigma, V)] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma \parallel M \parallel \square\ \mathrm{Build}_V(\Sigma, V) \cdot K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\langle\!\langle \Sigma \parallel \lambda x.\, M \parallel \square\ \mathbb{V} \cdot K \rangle\!\rangle \longmapsto_9 \langle\!\langle \Sigma, \mathbb{V}/x \parallel M \parallel K \rangle\!\rangle$$

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel \lambda x.\, M \parallel \square\ \mathbb{V} \cdot K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[(\lambda x.\, M)[\Sigma]\ \mathbb{V}] &=_{\beta_{\rightarrow}} \\
\underline{K}[M[\Sigma, \mathbb{V}/x]] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma, \mathbb{V}/x \parallel M \parallel K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\langle\!\langle \Sigma \parallel V.\mathtt{force} \parallel K \rangle\!\rangle \longmapsto_{10} \langle\!\langle \Sigma' \parallel M \parallel K \rangle\!\rangle$$

where $\mathrm{Build}_V(\Sigma, V) = \{\Sigma', \mathtt{force} \to M\}$.

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel V.\mathtt{force} \parallel K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[(V.\mathtt{force})[\Sigma]] &=_{\text{subst.}} \\
\underline{K}[(V[\Sigma]).\mathtt{force}] &=_{\text{Lemma 6.1}} \\
\underline{K}[\{\Sigma', \mathtt{force} \to M\}.\mathtt{force}] &=_{\beta_U} \\
\underline{K}[M[\Sigma']] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma' \parallel M \parallel K \rangle\!\rangle}
\end{aligned}$$

Case:

$$\begin{aligned}
\langle\!\langle \Sigma \parallel \mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y\rangle \to R\} \parallel K \rangle\!\rangle &\longmapsto_{11} \\
\langle\!\langle \Sigma, \mathbb{W}/x, \mathbb{W}'/y \parallel R \parallel K \rangle\!\rangle
\end{aligned}$$

where $\mathrm{Build}_V(\Sigma, V) = \langle \mathbb{W}, \mathbb{W}' \rangle$.

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel \mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y\rangle \to R\} \parallel K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[\mathsf{case}\ V[\Sigma]\ \mathsf{of}\ (\{\langle x, y\rangle \to R\}[\Sigma])] &=_{\text{Lemma 6.1}} \\
\underline{K}[\mathsf{case}\ \langle \mathbb{W}, \mathbb{W}' \rangle\ \mathsf{of}\ (\{\langle x, y\rangle \to R\}[\Sigma])] &=_{\beta_\otimes} \\
\underline{K}[R[\Sigma, \mathbb{W}/x, \mathbb{W}'/y]] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma, \mathbb{W}/x, \mathbb{W}'/y \parallel R \parallel K \rangle\!\rangle}
\end{aligned}$$

Case:

$$\langle\!\langle \Sigma \parallel \mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{box}\ a \to P\} \parallel K \rangle\!\rangle \longmapsto_{12} \langle\!\langle \Sigma, \mathbb{V}/a \parallel P \parallel K \rangle\!\rangle$$

where $\mathrm{Build}_V(\Sigma, V) = \mathsf{box}\ \mathbb{V}$.

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel \mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{box}\ a \to P\} \parallel K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[\mathsf{case}\ V[\Sigma]\ \mathsf{of}\ (\{\mathsf{box}\ a \to P\}[\Sigma])] &=_{\text{Lemma 6.1}} \\
\underline{K}[\mathsf{case}\ \mathsf{box}\ \mathbb{V}\ \mathsf{of}\ (\{\mathsf{box}\ a \to P\}[\Sigma])] &=_{\beta_{\tilde{U}}} \\
\underline{K}[P[\Sigma, \mathbb{V}/a]] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma, \mathbb{V}/x \parallel P \parallel K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\langle\!\langle \Sigma \parallel P\ \mathsf{to}\ x\ \mathsf{in}\ Q \parallel K \rangle\!\rangle \longmapsto_{13} \langle\!\langle \Sigma \parallel P \parallel (\Sigma, \square\ \mathsf{to}\ x\ \mathsf{in}\ Q) \cdot K \rangle\!\rangle$$

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel P\ \mathsf{to}\ x\ \mathsf{in}\ Q \parallel K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[(P\ \mathsf{to}\ x\ \mathsf{in}\ Q)[\Sigma]] &=_{\text{subst.}} \\
\underline{K}[P[\Sigma]\ (\mathsf{to}\ x\ \mathsf{in}\ Q)[\Sigma]] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma \parallel P \parallel (\Sigma, \square\ \mathsf{to}\ x\ \mathsf{in}\ Q) \cdot K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\begin{aligned}
\langle\!\langle \Sigma \parallel \mathsf{ret}\ V \parallel (\Sigma', \square\ \mathsf{to}\ x\ \mathsf{in}\ P) \cdot K \rangle\!\rangle &\longmapsto_{14} \\
\langle\!\langle \Sigma', \mathrm{Build}_V(\Sigma, V)/x \parallel P \parallel K \rangle\!\rangle &
\end{aligned}$$

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel \mathsf{ret}\ V \parallel (\Sigma', \square\ \mathsf{to}\ x\ \mathsf{in}\ P) \cdot K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[(\mathsf{ret}\ (V[\Sigma]))\ (\mathsf{to}\ x\ \mathsf{in}\ P)[\Sigma']] &=_{\text{Lemma 6.1}} \\
\underline{K}[(\mathsf{ret}\ \mathrm{Build}_V(\Sigma, V))\ (\mathsf{to}\ x\ \mathsf{in}\ P)[\Sigma']] &=_{\beta_F} \\
\underline{K}[P[\Sigma', \mathrm{Build}_V(\Sigma, V)/x]] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma', \mathrm{Build}(\Sigma, V)/x \parallel P \parallel K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\begin{aligned}
\langle\!\langle \Sigma \parallel \mathsf{val}\ V \parallel (\Sigma', \square\ \mathsf{to}\ x\ \mathsf{in}\ P) \cdot K \rangle\!\rangle &\longmapsto_{15} \\
\langle\!\langle \Sigma', \mathrm{Build}_V(\Sigma, V)/x \parallel P \parallel K \rangle\!\rangle &
\end{aligned}$$

Follows in a similar manner to the case above.

Case:

$$\langle\!\langle \Sigma \parallel R.\mathtt{enter} \parallel K \rangle\!\rangle \longmapsto_{16} \langle\!\langle \Sigma \parallel R \parallel \square.\mathtt{enter} \cdot K \rangle\!\rangle$$

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel R.\mathtt{enter} \parallel K \rangle\!\rangle} \quad &=_{\text{defn.}} \\
\underline{K}[(R.\mathtt{enter})[\Sigma]] \quad &=_{\text{subst.}} \\
\underline{K}[(R[\Sigma]).\mathtt{enter}] \quad &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma \parallel R \parallel \square.\mathtt{enter} \cdot K \rangle\!\rangle} \quad &
\end{aligned}$$

Case:

$$\langle\!\langle \Sigma \parallel \{\mathtt{enter} \to M\} \parallel \square.\mathtt{enter} \cdot K \rangle\!\rangle \longmapsto_{17} \langle\!\langle \Sigma \parallel M \parallel K \rangle\!\rangle$$

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel \{\mathtt{enter} \to M\} \parallel \square.\mathtt{enter} \cdot K \rangle\!\rangle} \quad &=_{\text{defn.}} \\
\underline{K}[(\{\mathtt{enter} \to M\}[\Sigma]).\mathtt{enter}] \quad &=_{\text{subst.}} \\
\underline{K}[(\{\mathtt{enter} \to M\}.\mathtt{enter})[\Sigma]] \quad &=_{\beta_{\check{U}}} \\
\underline{K}[M[\Sigma]] \quad &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma \parallel M \parallel K \rangle\!\rangle} \quad &
\end{aligned}$$

Case:

$$\langle\!\langle \Sigma \parallel M.\mathtt{eval} \parallel K \rangle\!\rangle \longmapsto_{18} \langle\!\langle \Sigma \parallel M \parallel \square.\mathtt{eval} \cdot K \rangle\!\rangle$$

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel M.\mathtt{eval} \parallel K \rangle\!\rangle} \quad &=_{\text{defn.}} \\
\underline{K}[(M.\mathtt{eval})[\Sigma]] \quad &=_{\text{subst.}} \\
\underline{K}[(M[\Sigma]).\mathtt{eval}] \quad &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma \parallel M \parallel \square.\mathtt{eval} \cdot K \rangle\!\rangle} \quad &
\end{aligned}$$

Case:

$$\langle\!\langle \Sigma \parallel \{\texttt{eval} \rightarrow R\} \parallel \square.\texttt{eval} \cdot K \rangle\!\rangle \longmapsto_{19} \langle\!\langle \Sigma \parallel R \parallel K \rangle\!\rangle$$

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel \{\texttt{eval} \rightarrow R\} \parallel \square.\texttt{eval} \cdot K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[(\{\texttt{eval} \rightarrow R\}[\Sigma]).\texttt{eval}] &=_{\text{subst.}} \\
\underline{K}[(\{\texttt{eval} \rightarrow R\}.\texttt{eval})[\Sigma]] &=_{\beta_{\tilde{F}}} \\
\underline{K}[R[\Sigma]] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Sigma \parallel R \parallel K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\langle\!\langle \Sigma \parallel a \parallel K \rangle\!\rangle \longmapsto_{20} \langle\!\langle \varepsilon \parallel \mathbb{I} \parallel K \rangle\!\rangle$$

where $a[\Sigma] = \mathbb{I}$.

$$\begin{aligned}
\underline{\langle\!\langle \Sigma \parallel a \parallel K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{K}[a[\Sigma]] &= \\
\underline{K}[\mathbb{I}] &=_{\text{subst.}} \\
\underline{K}[\mathbb{I}[\varepsilon]] &=_{\text{defn.}} \\
\underline{\langle\!\langle \varepsilon \parallel \mathbb{I} \parallel K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\frac{\langle\!\langle \Sigma \parallel P \parallel K \rangle\!\rangle \longmapsto \langle\!\langle \Sigma' \parallel P' \parallel K' \rangle\!\rangle}{\langle\!\langle \Phi \parallel \Sigma \parallel P \parallel K \rangle\!\rangle \longmapsto_{21} \langle\!\langle \Phi \parallel \Sigma' \parallel P' \parallel K' \rangle\!\rangle}$$

$$\begin{aligned}
\underline{\langle\!\langle \Phi \parallel \Sigma \parallel P \parallel K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{\Phi}[\underline{\langle\!\langle \Sigma \parallel P \parallel K \rangle\!\rangle}] &=_{\text{I.H.}} \\
\underline{\Phi}[\underline{\langle\!\langle \Sigma' \parallel P' \parallel K' \rangle\!\rangle}] &=_{\text{defn.}} \\
\underline{\langle\!\langle \Phi \parallel \Sigma' \parallel P' \parallel K' \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\begin{aligned}\langle\!\langle \Phi \parallel \Sigma \parallel R\ \mathtt{memo}\ a\ \mathtt{in}\ P \parallel K \rangle\!\rangle &\quad \longmapsto_{22} \\ \langle\!\langle \Phi, l \mapsto \mathrm{Build}_{a}(\Sigma, R) \parallel \Sigma, l/a \parallel P \parallel K \rangle\!\rangle &\end{aligned}$$

Note that stacks are decoded into evaluation contexts.

$$\begin{aligned}
\underline{\langle\!\langle \Phi \parallel \Sigma \parallel R\ \mathtt{memo}\ a\ \mathtt{in}\ P \parallel K \rangle\!\rangle} &\quad =_{\text{defn.}} \\
\underline{\Phi}[\underline{K}[(R\ \mathtt{memo}\ a\ \mathtt{in}\ P)[\Sigma]]] &\quad =_{\text{subst.}} \\
\underline{\Phi}[\underline{K}[R[\Sigma]\ \mathtt{memo}\ a\ \mathtt{in}\ P[\Sigma]]] &\quad =_{\kappa} \\
\underline{\Phi}[R[\Sigma]\ \mathtt{memo}\ a\ \mathtt{in}\ \underline{K}[P[\Sigma]]] &\quad = \\
\underline{\Phi, a \mapsto \mathrm{Build}_{a}(\Sigma, R)}[\underline{K}[P[\Sigma]]] &\quad =_{\alpha} \\
\underline{\Phi, l \mapsto \mathrm{Build}_{a}(\Sigma, R)}[\underline{K}[P[\Sigma, l/a]]] &\quad =_{\text{defn.}} \\
\underline{\langle\!\langle \Phi, l \mapsto \mathrm{Build}_{a}(\Sigma, R) \parallel \Sigma, l/a \parallel P \parallel K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\begin{aligned}\langle\!\langle (\Phi_0, a[\Sigma] \mapsto \{\Sigma', R\})\Phi_1 \parallel \Sigma \parallel a \parallel K \rangle\!\rangle &\quad \longmapsto_{23} \\ \langle\!\langle \Phi_0 \parallel \Sigma' \parallel R \parallel (\Phi_1, l) \cdot K \rangle\!\rangle &\end{aligned}$$

Note that stacks are decoded into evaluation contexts.

$$\begin{aligned}
\underline{\langle\!\langle (\Phi_0, a[\Sigma] \mapsto \{\Sigma', R\})\Phi_1 \parallel \Sigma \parallel a \parallel K \rangle\!\rangle} &\quad =_{\text{defn.}} \\
\underline{\Phi_0, a[\Sigma] \mapsto \{\Sigma', R\}}[\underline{\Phi_1}[\underline{K}[a[\Sigma]]]] &\quad = \\
\underline{\Phi_0, l \mapsto \{\Sigma', R\}}[\underline{\Phi_1}[\underline{K}[l]]] &\quad =_{\kappa} \\
\underline{\Phi_0}[\underline{K}[\{\Sigma', R\}\ \mathtt{memo}\ l\ \mathtt{in}\ \underline{\Phi_1}[l]]] &\quad =_{\text{cl.}} \\
\underline{\Phi_0}[\underline{K}[R[\Sigma']\ \mathtt{memo}\ l\ \mathtt{in}\ \underline{\Phi_1}[l]]] &\quad =_{\text{defn.}} \\
\underline{\langle\!\langle \Phi_0 \parallel \Sigma' \parallel R \parallel (\Phi_1, l) \cdot K \rangle\!\rangle} &
\end{aligned}$$

Case:

$$\langle\!\langle \Phi \parallel \Sigma \parallel V \parallel (\Phi', l) \cdot K \rangle\!\rangle \longmapsto_{24} \langle\!\langle (\Phi, l \mapsto \{\varepsilon, \mathbb{I}\})\Phi' \parallel \Sigma \parallel V \parallel K \rangle\!\rangle$$

where $\mathrm{Build}_V(\Sigma, V) = \mathbb{I}$.

$$\begin{aligned}
\underline{\langle\!\langle \Phi \parallel \Sigma \parallel V \parallel (\Phi', l) \cdot K \rangle\!\rangle} &=_{\text{defn.}} \\
\underline{\Phi}[\underline{K}[V[\Sigma]\ \texttt{memo}\ l\ \texttt{in}\ \underline{\Phi'}[l]]] &=_{\text{deref.}} \\
\underline{\Phi}[\underline{K}[V[\Sigma]\ \texttt{memo}\ l\ \texttt{in}\ \underline{\Phi'}[V[\Sigma]]]] &= \\
\underline{\Phi}[\underline{K}[\mathbb{I}\ \texttt{memo}\ l\ \texttt{in}\ \underline{\Phi'}[V[\Sigma]]]] &=_{\kappa} \\
\underline{\Phi}[\mathbb{I}\ \texttt{memo}\ l\ \texttt{in}\ \underline{\Phi'}[\underline{K}[V[\Sigma]]]] &=_{\text{defn.}} \\
\underline{\langle\!\langle (\Phi, l \mapsto \mathbb{I})\Phi' \parallel \Sigma \parallel V \parallel K \rangle\!\rangle} &=_{\text{sugar}} \\
\underline{\langle\!\langle (\Phi, l \mapsto \{\varepsilon, \mathbb{I}\})\Phi' \parallel \Sigma \parallel V \parallel K \rangle\!\rangle} &
\end{aligned}$$

□

Corollary 6.1. *If* $\mathrm{EvalS}(M) = \mathtt{b}$, *then* $\vdash M = \texttt{ret b} : F\ B$.

6.5 Observational Equivalence

Since the machine forms our operational semantics for CBPVS, we define a notion of observational equivalence on top of it. And since this notion of equivalence is the semantics for which we will verify the correctness of our closure theory, it must support the whole language and be fine-grained enough to discuss parts of sharing. As with the evaluators seen previously, we consider only running computable expressions that return base values. We will begin by defining observationally equivalent machine configurations as configurations that mutually imply the same base value computations; intuitively, $Conf_1 \simeq Conf_2$ if and only if their behaviors imply each other. On top of this, we define a notion of observably equivalent expressions, wherein we place expressions in a closing context that returns a base value; intuitively, $A \simeq B$ if and only if the behaviors of $C[A]$ and $C[B]$ in the machine imply each other for any closing computation.

Making this work with sharing means muddying this common notion of observational equivalence. The notion of equivalent expressions is similar in that we place them in closing computation contexts, but we must add more information to the notion of equivalent configurations. If we are evaluating a shared computation within a stack that demands an introduction form—as it will if we are memoizing or pattern matching, then equivalent configurations must evaluate to an intermediate state that includes a machine introduction if and only if the other does; those future configurations must also be observably equivalent. The stacks that will force a shared computation are the elimination forms for each shared type and the memoization frame. We refer to this notion as strict stacks; they consume introduction forms.

Definition 6.2 (Introductions and Strict Stacks)**.**

$$\begin{array}{rrcl} I \in & \mathit{Introduction} & = & \mathit{Shared\ Value} - \{a\} \\ \mathbb{K} \in & \mathit{VStack} & ::= & (\Sigma, \square\ \texttt{to}\ x\ \texttt{in}\ P) \cdot K \mid \square\texttt{.enter} \cdot K \quad \mid (\Phi, l) \cdot \mathbb{K} \end{array}$$

Formally, we define this observational equivalence with two relations. A base relation on configurations $\mathcal{B}$ and a shared introduction configuration $\mathcal{I}_{\mathcal{B}}$.

Definition 6.3 (Observably Equivalent Configurations).

$$\begin{aligned}
\mathit{Conf}_l \simeq \mathit{Conf}_r \quad \overset{\text{def}}{=} \quad & \mathit{Conf}_l \ \mathcal{B}\ \mathit{Conf}_r \\
\wedge \quad & \mathit{Conf}_l = \langle\!\langle \Phi_l \parallel \Sigma_l \parallel R_l \parallel \mathbb{K}_l \rangle\!\rangle \\
& \mathit{Conf}_r = \langle\!\langle \Phi_r \parallel \Sigma_r \parallel R_r \parallel \mathbb{K}_r \rangle\!\rangle \\
& \quad \Longrightarrow \mathit{Conf}_l \ \mathcal{I}_{\mathcal{B}}\ \mathit{Conf}_r
\end{aligned}$$

$$\begin{aligned}
\mathit{Conf}_l \ \mathcal{B}\ \mathit{Conf}_r \quad \overset{\text{def}}{=} \quad & \forall i, j \in \{l, r\}.\, \mathit{Conf}_i \longmapsto^{*} \langle\!\langle \Phi_i \parallel \Sigma_i \parallel \texttt{ret b} \parallel \star \rangle\!\rangle \\
& \quad \Longrightarrow \mathit{Conf}_j \longmapsto^{*} \langle\!\langle \Phi_j \parallel \Sigma_j \parallel \texttt{ret b} \parallel \star \rangle\!\rangle
\end{aligned}$$

$$\begin{aligned}
& \langle\!\langle \Phi_l \parallel \Sigma_l \parallel R_l \parallel \mathbb{K}_l \rangle\!\rangle \ \mathcal{I}_{\mathcal{B}}\ \langle\!\langle \Phi_r \parallel \Sigma_r \parallel R_r \parallel \mathbb{K}_r \rangle\!\rangle \\
& \qquad \overset{\text{def}}{=} \quad \forall i, j \in \{l, r\}. \\
& \qquad \qquad \langle\!\langle \Phi_i \parallel \Sigma_i \parallel R_i \parallel \mathbb{K}_i \rangle\!\rangle \longmapsto^{*} \langle\!\langle \Phi'_i \parallel \Sigma'_i \parallel I_i \parallel \mathbb{K}_i \rangle\!\rangle \\
& \qquad \qquad \quad \Longrightarrow \langle\!\langle \Phi_j \parallel \Sigma_j \parallel R_j \parallel \mathbb{K}_j \rangle\!\rangle \longmapsto^{*} \langle\!\langle \Phi'_j \parallel \Sigma'_j \parallel I_j \parallel \mathbb{K}_j \rangle\!\rangle \\
& \qquad \qquad \qquad \wedge \langle\!\langle \Phi_i \parallel \Sigma_i \parallel I_i \parallel \mathbb{K}_i \rangle\!\rangle \ \mathcal{B}\ \langle\!\langle \Phi_j \parallel \Sigma_j \parallel I_j \parallel \mathbb{K}_j \rangle\!\rangle
\end{aligned}$$

Definition 6.4 (Observably Equivalent Expressions).

$$\begin{aligned}
A \simeq B \quad \overset{\text{def}}{=} \quad & \forall C \in \mathit{Context}.\, \varepsilon \vdash C[A] : F\,B \wedge \varepsilon \vdash C[B] : F\,B \\
& \quad \Longrightarrow \langle\!\langle \varepsilon \parallel \varepsilon \parallel C[A] \parallel \star \rangle\!\rangle \simeq \langle\!\langle \varepsilon \parallel \varepsilon \parallel C[B] \parallel \star \rangle\!\rangle
\end{aligned}$$

What follows are extra properties that follow from our definition of observationally equivalent configurations. First, we have a recursive definition that allows us to build up a call stack given an evaluation context. This is used in observational equivalence property number (4) below.

Definition 6.5 (Building Stacks and Bindings).

$$
\begin{array}{rcl}
\text{Build}_K(\Phi, \Sigma, \square, K) & = & (\Phi, \Sigma, K) \\
\text{Build}_K(\Phi, \Sigma, E\ V, K) & = & \text{Build}_K(\Phi, \Sigma, E, \square\ \text{Build}_V(\Sigma, V) \cdot K) \\
\text{Build}_K(\Phi, \Sigma, E.\texttt{fst}, K) & = & \text{Build}_K(\Phi, \Sigma, E, \square.\texttt{fst} \cdot K) \\
\text{Build}_K(\Phi, \Sigma, E.\texttt{snd}, K) & = & \text{Build}_K(\Phi, \Sigma, E, \square.\texttt{snd} \cdot K) \\
\text{Build}_K(\Phi, \Sigma, E\ \texttt{to}\ x\ \texttt{in}\ P, K) & = & \text{Build}_K(\Phi, \Sigma, E, (\Sigma, \square\ \texttt{to}\ x\ \texttt{in}\ P) \cdot K) \\
\text{Build}_K(\Phi, \Sigma, E.\texttt{enter}, K) & = & \text{Build}_K(\Phi, \Sigma, E, \square.\texttt{enter} \cdot K) \\
\text{Build}_K(\Phi, \Sigma, E.\texttt{eval}, K) & = & \text{Build}_K(\Phi, \Sigma, E, \square.\texttt{eval} \cdot K) \\
\text{Build}_K(\Phi, \Sigma, R\ \texttt{memo}\ a\ \texttt{in}\ E, K) & = & \text{Build}_K((\Phi, l \mapsto \text{Build}_a(\Sigma, R)), (\Sigma, l/a), E, K) \\
\text{Build}_K(\Phi, \Sigma, E\ \texttt{memo}\ a\ \texttt{in}\ F[a], K) & = & \text{Build}_K(\Phi, \Sigma, E, (\Phi', l) \cdot K') \\
 & & \textbf{where}\ \text{Build}_K(\varepsilon, (\Sigma, l/a), F, K) = (\Phi', \Sigma', K')
\end{array}
$$

Proposition 6.1 (Observational Equivalence Properties).

1. *$Conf \longmapsto^* Conf'$ implies $Conf \simeq Conf'$.*

2. *Observational equivalence relations for configurations ($\simeq$), expressions ($\simeq$), base value configurations ($\mathcal{B}$), and shared introduction configurations ($\mathcal{I}_\mathcal{B}$) are reflexive, symmetric, and transitive.*

3. *$(Conf_0 \simeq Conf_1) \wedge (Conf_0 \longmapsto^* Conf_0')$ implies $Conf_0' \simeq Conf_1$.*

4. *$\langle\!\langle \Phi \parallel \Sigma \parallel E[P] \parallel K \rangle\!\rangle \simeq \langle\!\langle \Phi' \parallel \Sigma' \parallel P \parallel K' \rangle\!\rangle$ where $\text{Build}_K(\Sigma, E, K) = (\Phi', \Sigma', K')$.*

5. *$\mathcal{I}_\mathcal{B} \subseteq (\simeq)$.*

6. *$\langle\!\langle \Phi \parallel \Sigma \parallel R \parallel (\Phi', l) \cdot \mathbb{K} \rangle\!\rangle \simeq \langle\!\langle \Phi \parallel \Sigma \parallel R \parallel \mathbb{K} \rangle\!\rangle$ where $\text{Dom}(\Phi') \cap \text{Dom}(\Phi) = \emptyset$.*

Proof.

1. Suppose that $\mathit{Conf} \longmapsto^* \mathit{Conf}'$, we may conclude that $\mathit{Conf} \simeq \mathit{Conf}'$ by the determinism of $(\longmapsto)$.

2. These, we prove for Conf $\mathcal{B}$ Conf' as Conf $\mathcal{I}_{\mathcal{B}}$ Conf' is proved in a similar manner to $\mathcal{B}$. The property for $(\simeq)$ of configurations and expressions is derived therefrom.

   Reflexivity: $\mathit{Conf}\ \mathcal{B}\ \mathit{Conf}$ then follows immediately by its assumptions.

   Symmetry: Suppose $\mathit{Conf}_0\ \mathcal{B}\ \mathit{Conf}_1$. Then $\mathit{Conf}_1\ \mathcal{B}\ \mathit{Conf}_0$ by definition, by swapping the choice of indexes in the underlying implications.

   Transitivity: Suppose $\mathit{Conf}_0\ \mathcal{B}\ \mathit{Conf}_1$ and $\mathit{Conf}_1\ \mathcal{B}\ \mathit{Conf}_2$. Then $\mathit{Conf}_0\ \mathcal{B}\ \mathit{Conf}_2$ follows by composing the underlying implications from the assumptions:

$$\begin{aligned}
&\mathit{Conf}_0 \longmapsto^* \langle\!\langle \Phi_0 \parallel \Sigma_0 \parallel \mathtt{ret}\ \mathtt{b} \parallel \star \rangle\!\rangle \\
&\qquad \Longrightarrow \mathit{Conf}_1 \longmapsto^* \langle\!\langle \Phi_1 \parallel \Sigma_1 \parallel \mathtt{ret}\ \mathtt{b} \parallel \star \rangle\!\rangle \\
&\qquad \Longrightarrow \mathit{Conf}_2 \longmapsto^* \langle\!\langle \Phi_2 \parallel \Sigma_2 \parallel \mathtt{ret}\ \mathtt{b} \parallel \star \rangle\!\rangle \\
&\mathit{Conf}_2 \longmapsto^* \langle\!\langle \Phi_2 \parallel \Sigma_2 \parallel \mathtt{ret}\ \mathtt{b} \parallel \star \rangle\!\rangle \\
&\qquad \Longrightarrow \mathit{Conf}_1 \longmapsto^* \langle\!\langle \Phi_1 \parallel \Sigma_1 \parallel \mathtt{ret}\ \mathtt{b} \parallel \star \rangle\!\rangle \\
&\qquad \Longrightarrow \mathit{Conf}_0 \longmapsto^* \langle\!\langle \Phi_0 \parallel \Sigma_0 \parallel \mathtt{ret}\ \mathtt{b} \parallel \star \rangle\!\rangle
\end{aligned}$$

3. Follows by the first property of $(\simeq)$ and transitivity.

4. Follows by induction on the evaluation context and the first property of $(\simeq)$. Each evaluation context has a rule for constructing a larger stack, which is exactly replicated by the Build function for evaluation contexts.

5. Follows from assumptions and backwards closure.

6. We show $\langle\!\langle \Phi \parallel \Sigma \parallel R \parallel (\Phi', l) \cdot \kappa \rangle\!\rangle \ \mathcal{I}_{\mathcal{B}} \ \langle\!\langle \Phi \parallel \Sigma \parallel R \parallel \kappa \rangle\!\rangle$. The rest follows from property (5).

   Suppose $\langle\!\langle \Phi_0 \parallel \Sigma \parallel R \parallel (\Phi_1, l) \cdot \kappa \rangle\!\rangle \longmapsto^* \langle\!\langle \Phi'_0 \parallel \Sigma' \parallel I \parallel (\Phi_1, l) \cdot \kappa \rangle\!\rangle$. The configuration takes one more step where $\mathrm{Build}_V(\Sigma', I) = \mathbb{I}$:

$$\begin{array}{c}\langle\!\langle \Phi'_0 \parallel \Sigma' \parallel I \parallel (\Phi_1, l) \cdot \kappa \rangle\!\rangle \quad \longmapsto \\ \langle\!\langle (\Phi'_0, l \mapsto \{\varepsilon, \mathbb{I}\})\Phi_1 \parallel \Sigma' \parallel I \parallel \kappa \rangle\!\rangle\end{array}$$

   By the determinism of $(\longmapsto)$ and the fact that value stacks only consume introductions, we know $\langle\!\langle \Phi_0 \parallel \Sigma \parallel R \parallel \kappa \rangle\!\rangle \longmapsto^* \langle\!\langle \Phi'_0 \parallel \Sigma' \parallel I \parallel \kappa \rangle\!\rangle$. By reflexivity and that I and Σ' cannot reference anything in the extended heap, we know $\langle\!\langle (\Phi'_0, l \mapsto \{\{\Sigma', I\}\})\Phi_1 \parallel \Sigma' \parallel I \parallel \kappa \rangle\!\rangle \ \mathcal{I}_{\mathcal{B}} \ \langle\!\langle \Phi'_0 \parallel \Sigma' \parallel I \parallel \kappa \rangle\!\rangle$. Backwards closure of $\mathcal{I}_{\mathcal{B}}$ yields $\langle\!\langle \Phi_0 \parallel \Sigma \parallel R \parallel (\Phi_1, l) \cdot \kappa \rangle\!\rangle \ \mathcal{I}_{\mathcal{B}} \ \langle\!\langle \Phi_0 \parallel \Sigma \parallel R \parallel \kappa \rangle\!\rangle$.

   The other direction follows with a similar determinism reasoning with the fact that value stacks only consume introductions. We conclude by $\mathcal{I}_{\mathcal{B}} \subseteq (\simeq)$.

□

CHAPTER VII

A MODEL OF TYPES OVER THE CBPVS MACHINE

This chapter is a revision the appendix of Closure Conversion in Little Pieces [53] co-authored with Paul Downen and Zena M. Ariola. The proof structure is authored by Zachary J. Sullivan under the guidance of Paul Downen.

We would like to know that our equational theory for CBPVS with closures is indeed reified by our CBPVS machine, especially since our call-by-name, call-by-value, and call-by-need approaches to closures rests on compiling the languages to CBPVS first. Specifically, we would like to show not only that the theory is sound with respect to the CBPVS machine (Theorem 7.1), but also that the abstract closures in the theory have a meaningful impact on the machine (Theorem 7.2) which we call the adequacy of closure conversion. This, we will do by constructing a semantic model over the CBPVS machine, which we will denote $\Gamma \vDash M \approx N : \tau$. Like the syntactic equational and typing rules, semantic equality should be reflexive, transitive, symmetric, and compatible; in other words, it should be a congruence relation. The semantic model should imply that M and N are observationally equivalent in the machine. And since it is defined over the machine, we should be able to discuss the impact of closure constructed equalities on the machine.

Such a model over an abstract machine or any other form of operational semantics is not an uncommon approach to proving correctness of compiler transformations, proving type safety (*e.g.* type-preserving mutable state [2]), normalization of programs (*e.g.* simply-typed call-by-need with control is normalizing [35]), and many other properties about the runtime system. What makes this work unique is that we wish to combine the following:

– our operational semantics is a delayed substitution semantics so that closures are visible sub-components,

– our operational semantics includes a memoizing heap,

– our operational semantics is stack based, and

– we are interested in *equational theories* which contain non-computational axioms like η.

The main structure of our model follows the $\top\top$-closure approach of Pitts [45]. That is, we base our model on two families of relations defined inductively over the types in our language: the first is over computational units and the other is over stacks. Such an approach handles both the stack-based operational semantics and aids in reasoning about η axioms. To handle delayed substitutions as we saw in Chapter 3, computational units are closed pairs of computational expressions and the local environment that they need to run. To handle the memoizing heap, these two families of relations are extended to Kripke relations, where the world is the heap which they depend on and an accessible world is a heap extended with new cells. After defining these two families of relations, we will define exactly the notion of semantic equivalence upon it.

7.1 Logical Relations

We base our relations on the notion of observational equivalent configurations from the previous chapter (Definition 6.3). We wish to construct a relation on the expressions in CBPVS that have the same behavior when placed in a configuration. For instance, two computations M and N should be related in $\mathcal{R}$ when they produce observationally equivalence configurations in an empty machine:

$$(M, N) \in \mathcal{R} \text{ iff } \langle\!\langle \varepsilon \parallel \varepsilon \parallel M \parallel \star \rangle\!\rangle \simeq \langle\!\langle \varepsilon \parallel \varepsilon \parallel N \parallel \star \rangle\!\rangle$$

Since our typing rules and equational theory works over open terms, our logical relation ought to as well. Usually, this is done by developing a notion of related environments for a typing environment Γ and then substituting thereafter to get closed expressions. In our context, however, substitution is delayed in the operational semantics. Ignoring for now the memoizing heap, we have relations over closures of local environments and computations like the following:

$$((\Sigma_l, M), (\Sigma_r, N)) \in \mathcal{R} \text{ iff } \langle\langle \varepsilon \parallel \Sigma_l \parallel M \parallel \star \rangle\rangle \simeq \langle\langle \varepsilon_r \parallel \Sigma_r \parallel N \parallel \star \rangle\rangle$$

Such a relation is now open for expressions M and N, but they are still for top-level programs in the sense that they occur in the empty stack. For our model to be compatible, we must be able to bury these related expressions within any context. Of course, stacks can vary across machine configurations like computations expressions; imagine, for example, that we push a differently closure converted expression on the stack. Thus, our relations should consider an additional relation on stacks:

$$((\Sigma_l, M), (\Sigma_r, N)) \in \mathcal{R} \quad \text{iff} \quad \forall (K_l, K_r) \in \mathcal{K}.\, \langle\langle \varepsilon \parallel \Sigma_l \parallel M \parallel K_l \rangle\rangle \simeq \langle\langle \varepsilon \parallel \Sigma_r \parallel N \parallel K_r \rangle\rangle$$

This approach to constructing relations is referred to as *bi-orthogonality* [35, 17, 16]. For each type, we will have a notion of stack and expression relations; the two different notions of relations will depend on each other. Since values and shared values are built into machine values and shared values before being placed on the stack making them closed, the relation on stacks does not need to be paired with a local environment.

Note that since expressions paired with an environment and stacks may have free heap locations, these relations must be closed by a notion of heap as well. We include them in the relations as well:

$$((\Phi_l, \Sigma_l, M), (\Phi_r, \Sigma_r, N)) \in \mathcal{R} \quad \text{iff} \quad \forall((\Phi_l, K_l), (\Phi_r, K_r)) \in \mathcal{K}. \\ \langle\!\langle \Phi_l \parallel \Sigma_l \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r \parallel \Sigma_r \parallel N \parallel K_r \rangle\!\rangle$$

See that the related stacks are paired with a heap; this is because stacks may have free heap locations like local environments. Thus, we have arrive at the final structure of the relations that we will use to model types. Of course, shared computations are runnable as well, so we generalize from the exposition thus far.

Definition 7.1 (Expression and Stack Relations)**.** *An expression relation is a relation $\mathcal{P} \subseteq$ $(Heap \times Mach.\ Env. \times Computable\ Exp.)^2$ such that the machine environment closes over the free variables of computable expression and the heap closes over the free locations of the machine environment.*

A stack relation is a relation $\mathcal{K} \subseteq (Heap \times Stack)^2$ such that the heap closes over the free locations of the stack.

Our logical relations should be closed whenever we replace the current heap with one in the future. For instance, if $(\Phi_l, \Sigma_l, M), (\Phi_r, \Sigma_r, N)) \in \mathcal{R}$ are related computations that are stored within thunks in the heap, then they should still be related when we perform an update to the heap Φ_l. Heap updates are frequent in shared code. We define precisely the notion of future heap as the following:

Definition 7.2 (Accessible Heap)**.** *$\Phi' \sqsupseteq \Phi$ iff for any division $(\Phi_0, l \mapsto \{\Sigma, R\})\Phi_1$ of Φ, we know Φ' is $(\Phi'_0, l \mapsto \{\Sigma, R\})\Phi'_1$ such that $\Phi'_0 \sqsupseteq \Phi_0$ and $\Phi'_1 \sqsupseteq \Phi_1$.*

Proposition 7.1. *$(Heap, \sqsupseteq)$ is an antisymmetric preorder.*

Closing relations over a preorder captures the idea of a Kripke relation. We take the pair of heaps (Φ_l, Φ_r) to be a world and say that a relation has a Kripke property if it is closed under any accessible world.

Definition 7.3 (Kripke Relation)**.** *A relation $\mathcal{R}$ is a Kripke relation if and only if $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi_l, A), (\Phi_r, B)) \in \mathcal{R}$ implies $((\Phi'_l, A), (\Phi'_r, B)) \in \mathcal{R}$. We have a similar definition for triples.*

Note that, Kripke logical relations for operational semantics with a heap are not new. In previous approaches like that of Ahmed *et al.* [4], we see that the world is modeled with a single heap typing among other things. Here, it is important that we have a part of the world for each side of the relation; this is because laws like χ show that thunks in the heap may be evaluated at different times, but still be related. This property appears to be unique to our shared evaluation setting.

To build up our logical relations, we define orthogonality operations $\mathit{CompRel}^{\perp\!\!\!\perp}$ and $\mathit{StackRel}^{\top\!\!\!\top}$ over relations for computable expressions and stacks. The former operation will give us all of the computations that when combined with the related stacks in *StackRel* will produce observationally equivalent configurations; similarly for the latter definition.

Definition 7.4 (Kripke Orthogonal Operations)**.**

$$\begin{aligned}
\mathcal{P}^{\perp\!\!\!\perp} &= \{((\Phi_l, K_l), (\Phi_r, K_r)) \mid \forall \Phi'_l \sqsupseteq \Phi_l . \forall \Phi'_r \sqsupseteq \Phi_r . \forall ((\Phi'_l, \Sigma_l, P), (\Phi'_r, \Sigma_r, Q)) \in \mathcal{P}. \\
&\qquad \langle\!\langle \Phi'_l \parallel \Sigma_r \parallel P \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel Q \parallel K_r \rangle\!\rangle \} \\
\mathcal{K}^{\top\!\!\!\top} &= \{((\Phi_l, \Sigma_l, P), (\Phi_r, \Sigma_r, Q)) \mid \forall \Phi'_l \sqsupseteq \Phi_l . \forall \Phi'_r \sqsupseteq \Phi_r . \forall ((\Phi'_l, K_l), (\Phi', K_r)) \in \mathcal{K}. \\
&\qquad \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel P \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel Q \parallel K_r \rangle\!\rangle \}
\end{aligned}$$

These orthogonal operations have important qualities necessary for us to use them for our logical relations. First, they construct Kripke relations for arbitrary expression and stack relations. Second, they are both contravariant. Third, their double application is an inclusion. And fourth, their triple application is equal to single application.

Proposition 7.2 (Orthogonal Relations are Kripke Relations)**.** *For any $\mathcal{P}$ and $\mathcal{K}$, $\mathcal{P}^{\perp\!\!\perp}$ and $\mathcal{K}^{\top\!\!\top}$ are Kripke relations.*

Proof. Given $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and some $((\Phi_l, K_l), (\Phi_r, K_r)) \in \mathcal{P}^{\perp\!\!\perp}$. $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathcal{P}^{\perp\!\!\perp}$ is shown by considering some $\Phi''_l \sqsupseteq \Phi'_l$, $\Phi''_r \sqsupseteq \Phi'_r$ and $((\Phi''_l, \Sigma_l, P), (\Phi''_r, \Sigma_r, Q)) \in \mathcal{P}$ and then show $\langle\!\langle \Phi''_l \parallel \Sigma_l \parallel P \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi''_r \parallel \Sigma_r \parallel Q \parallel K_r \rangle\!\rangle$. This follows from the property of $((\Phi_l, K_l), (\Phi_r, K_r)) \in \mathcal{P}^{\perp\!\!\perp}$ since $\Phi''_l \sqsupseteq \Phi_l$ by transitivity, $\Phi''_r \sqsupseteq \Phi_r$ by transitivity also, and $((\Phi''_l, \Sigma_l, P), (\Phi''_r, \Sigma_r, Q)) \in \mathcal{P}$.

The proof follows similarly for the second conjunct. □

Proposition 7.3 (Orthogonal Inclusion)**.** *For Kripke relations $\mathcal{P}$ and $\mathcal{K}$, $\mathcal{P} \subseteq \mathcal{P}^{\perp\!\!\perp\top\!\!\top}$ and $\mathcal{K} \subseteq \mathcal{K}^{\top\!\!\top\perp\!\!\perp}$.*

Proof. Considering an arbitrary pair $((\Phi_l, K_l), (\Phi_r, K_r)) \in \mathcal{K}$, we must show that it is also in $\mathcal{K}^{\top\!\!\top\perp\!\!\perp}$. Thus, consider further some $\Phi'_l \sqsupseteq \Phi_r$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, \Sigma_l, P), (\Phi'_r, \Sigma_r, Q)) \in \mathcal{K}^{\top\!\!\top}$, we must show $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel P \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel Q \parallel K_r \rangle\!\rangle$. This follows from the property of $((\Phi'_l, \Sigma_l, P), (\Phi'_r, \Sigma_r, Q)) \in \mathcal{K}^{\top\!\!\top}$ with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathcal{K}$ (closed under accessible worlds).

The first proposition is proved in a similar manner. □

Proposition 7.4 (Orthogonal Contravariance)**.** *For Kripke relations $\mathcal{P}$, $\mathcal{P}'$, $\mathcal{K}$, and $\mathcal{K}'$,*

- *$\mathcal{P} \subseteq \mathcal{P}'$ implies $\mathcal{P}^{\perp\!\!\perp} \supseteq \mathcal{P}'^{\perp\!\!\perp}$.*
- *$\mathcal{K} \subseteq \mathcal{K}'$ implies $\mathcal{K}^{\top\!\!\top} \supseteq \mathcal{K}'^{\top\!\!\top}$.*

Proof. Given $\mathcal{P} \subseteq \mathcal{P}'$ and $((\Phi_l, K_l), (\Phi_r, K_r)) \in \mathcal{P}'^{\perp\!\!\perp}$, we prove $((\Phi_l, K_l), (\Phi_r, K_r)) \in \mathcal{P}^{\perp\!\!\perp}$, by considering further some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, \Sigma_l, P), (\Phi'_r, \Sigma_r, Q)) \in \mathcal{P}$:

$((\Phi'_l, \Sigma_l, P), (\Phi'_r, \Sigma_r, Q)) \in \mathcal{P}'$, by $\mathcal{P} \subseteq \mathcal{P}'$.

$((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathcal{P}'^{\perp\!\!\!\perp}$, by closure over future worlds.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel P \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel Q \parallel K_r \rangle\!\rangle$, by the property of $\mathcal{P}'^{\perp\!\!\!\perp}$ with worlds $\Phi'_l \sqsupseteq \Phi'_l$ and $\Phi'_r \sqsupseteq \Phi'_r$ and the expressions $((\Phi'_l, \Sigma_l, P), (\Phi'_r, \Sigma_r, Q)) \in \mathcal{P}'$.

□

Proposition 7.5 (Triple Orthogonal Elimination**).**

For Kripke relations $\mathcal{P}$ and $\mathcal{K}$, $\mathcal{P}^{\perp\!\!\!\perp} = \mathcal{P}^{\perp\!\!\!\perp\top\!\!\!\top\perp\!\!\!\perp}$ and $\mathcal{K}^{\top\!\!\!\top} = \mathcal{K}^{\top\!\!\!\top\perp\!\!\!\perp\top\!\!\!\top}$.

Proof. First, considering an arbitrary $((\Phi_l, \Sigma_l, P), (\Phi_r, \Sigma_r, Q)) \in \mathcal{K}^{\top\!\!\!\top}$, we must show that the triple is in $\mathcal{K}^{\top\!\!\!\top\perp\!\!\!\perp\top\!\!\!\top}$. This follows by double orthogonal inclusion on the Kripke relation. Note that $\mathcal{K}^{\top\!\!\!\top}$ is a Kripke relation because $\mathcal{K}$ is.

Second, from $((\Phi_l, \Sigma_l, P), (\Phi_r, \Sigma_r, Q)) \in \mathcal{K}^{\top\!\!\!\top\perp\!\!\!\perp\top\!\!\!\top}$, we must show that the pair is in $\mathcal{K}^{\top\!\!\!\top}$; that is, $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel P \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel Q \parallel K_r \rangle\!\rangle$ for any $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathcal{K}$. By double orthogonal inclusion with the Kripke relation $\mathcal{K}$, we know that $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathcal{K}^{\top\!\!\!\top\perp\!\!\!\perp}$. Thus, we may conclude this case by the property of $((\Phi_l, \Sigma_l, P), (\Phi_r, \Sigma_r, Q)) \in \mathcal{K}^{\top\!\!\!\top\perp\!\!\!\perp\top\!\!\!\top}$ with $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathcal{K}^{\top\!\!\!\top\perp\!\!\!\perp}$.

The second part of this proposition follows in a similar manner. □

Without further ado, our logical relation definitions of types is presented in Figure 7.1. Since value types require no reduction, we do not need to make use of the orthogonal operations in order to specify related elements of a type. The interesting case for this work is that of machine values of type $U\ \tau$, since abstract closures will have this type. Indeed, this relation is simple since abstract closures are—by design—the exact objects that the abstract machine will generate at runtime. Thus, it does not matter here whether the closure was captured via compilation or at runtime; we merely

$$
\begin{aligned}
\mathit{Val}[\![B]\!] &= \{((\Phi_l, \mathsf{b}), (\Phi_r, \mathsf{b}))\} \\
\mathit{Val}[\![U\ \tau]\!] &= \{((\Phi_l, \{\varsigma_l, \mathtt{force} \longrightarrow M\}), (\Phi_r, \{\varsigma_r, \mathtt{force} \longrightarrow N\})) \mid \\
&\qquad ((\Phi_l, \Sigma_l, M), (\Phi_r, \Sigma_r, N)) \in \mathit{Comp}[\![\tau]\!]\} \\
\mathit{Val}[\![\tau \otimes \sigma]\!] &= \{((\Phi_l, \langle \mathbb{V}_l, \mathbb{W}_l\rangle), (\Phi_r, \langle \mathbb{V}_r, \mathbb{W}_r\rangle)) \mid \\
&\qquad ((\Phi_l, \mathbb{V}_l), (\Phi_r, \mathbb{V}_r)) \in \mathit{Val}[\![\tau]\!] \wedge ((\Phi_l, \mathbb{W}_l), (\Phi_r, \mathbb{W}_r)) \in \mathit{Val}[\![\sigma]\!]\} \\
\mathit{Val}[\![\tilde{U}\ \tau]\!] &= \{((\Phi_l, \mathtt{box}\ \mathbb{V}), (\Phi_r, \mathtt{box}\ \mathbb{W})) \mid (\mathrm{Run}(\Phi_l, \mathbb{V}), \mathrm{Run}(\Phi_r, \mathbb{W})) \in \mathit{Shared}[\![\tau]\!]\} \\
&\quad \textbf{where} \begin{array}{l} \mathrm{Run}(\Phi, l) \ = \ (\Phi, (\varepsilon, l/a), a) \\ \mathrm{Run}(\Phi, \mathbb{I}) \ = \ (\Phi, \varepsilon, \mathbb{I}) \end{array}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Elim}[\![\tau \longrightarrow \sigma]\!] &= \{((\Phi_l, \square\ \mathbb{V} \cdot K_l), (\Phi_r, \square\ \mathbb{W} \cdot K_r)) \mid \\
&\qquad ((\Phi_l, \mathbb{V}), (\Phi_r, \mathbb{W})) \in \mathit{Val}[\![\tau]\!] \wedge ((\Phi_l, K_l), (\Phi_r, K_r)) \in \mathit{Stack}[\![\sigma]\!]\} \\
\mathit{Comp}[\![\tau \longrightarrow \sigma]\!] &= \mathit{Elim}[\![\tau \longrightarrow \sigma]\!]^{\top\!\!\top} \\
\mathit{Stack}[\![\tau \longrightarrow \sigma]\!] &= \mathit{Comp}[\![\tau \longrightarrow \sigma]\!]^{\perp\!\!\perp}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Elim}[\![\tau \mathbin{\&} \sigma]\!] &= \{((\Phi_l, \square.\mathtt{fst} \cdot K_l), (\Phi_r, \square.\mathtt{fst} \cdot K_r)) \mid ((\Phi_l, K_l), (\Phi_r, K_r)) \in \mathit{Stack}[\![\tau]\!]\} \\
&\quad \cup \{((\Phi_l, \square.\mathtt{snd} \cdot K_l), (\Phi_r, \square.\mathtt{snd} \cdot K_r)) \mid ((\Phi_l, K_l), (\Phi_r, K_r)) \in \mathit{Stack}[\![\sigma]\!]\} \\
\mathit{Comp}[\![\tau \mathbin{\&} \sigma]\!] &= \mathit{Elim}[\![\tau \mathbin{\&} \sigma]\!]^{\top\!\!\top} \\
\mathit{Stack}[\![\tau \mathbin{\&} \sigma]\!] &= \mathit{Comp}[\![\tau \mathbin{\&} \sigma]\!]^{\perp\!\!\perp}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Intro}[\![F\ \tau]\!] &= \{((\Phi_l, \Sigma_l, \mathtt{ret}\ V), (\Phi_r, \Sigma_r, \mathtt{ret}\ W)) \mid \\
&\qquad ((\Phi_l, \mathrm{Build}_V(\Sigma_l, V)), (\Phi_r, \mathrm{Build}_V(\Sigma_r, W))) \in \mathit{Val}[\![\tau]\!]\} \\
\mathit{Stack}[\![F\ \tau]\!] &= \mathit{Intro}[\![F\ \tau]\!]^{\perp\!\!\perp} \\
\mathit{Comp}[\![F\ \tau]\!] &= \mathit{Stack}[\![F\ \tau]\!]^{\top\!\!\top}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Elim}[\![\tilde{F}\ \tau]\!] &= \{((\Phi_l, \square.\mathtt{eval} \cdot K_l), (\Phi_r, \square.\mathtt{eval} \cdot K_r)) \mid ((\Phi_l, K_l), (\Phi_r, K_r)) \in \mathit{Stack}[\![\tau]\!]\} \\
\mathit{Comp}[\![\tilde{F}\ \tau]\!] &= \mathit{Elim}[\![\tilde{F}\ \tau]\!]^{\top\!\!\top} \\
\mathit{Stack}[\![\tilde{F}\ \tau]\!] &= \mathit{Comp}[\![\tilde{F}\ \tau]\!]^{\perp\!\!\perp}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Intro}[\![\hat{F}\ \tau]\!] &= \{((\Phi_l, \Sigma_l, \mathtt{val}\ V), (\Phi_r, \Sigma_r, \mathtt{val}\ W)) \mid \\
&\qquad ((\Phi_l, \mathrm{Build}_V(\Sigma_l, V)), (\Phi_r, \mathrm{Build}_V(\Sigma_r, W))) \in \mathit{Val}[\![\tau]\!]\} \\
\mathit{VStack}[\![\hat{F}\ \tau]\!] &= \mathit{Intro}[\![\hat{F}\ \tau]\!]^{\perp\!\!\perp} \cap \{((\Phi_l, \mathbb{K}_l), (\Phi_r, \mathbb{K}_r))\} \\
\mathit{Shared}[\![\hat{F}\ \tau]\!] &= \mathit{VStack}[\![\hat{F}\ \tau]\!]^{\top\!\!\top} \\
\mathit{Stack}[\![\hat{F}\ \tau]\!] &= \mathit{Shared}[\![\hat{F}\ \tau]\!]^{\perp\!\!\perp}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Elim}[\![\check{U}\ \tau]\!] &= \{((\Phi_l, \square.\mathtt{enter} \cdot K_l), (\Phi_r, \square.\mathtt{enter} \cdot K_r)) \mid \\
&\qquad ((\Phi_l, K_l), (\Phi_r, K_r)) \in \mathit{Stack}[\![\tau]\!]\} \\
\mathit{Val}[\![\check{U}\ \tau]\!] &= \mathit{Elim}[\![\check{U}\ \tau]\!]^{\top\!\!\top} \cap \{((\Phi_l, \Sigma_l, V), (\Phi_r, \Sigma_r, W))\} \\
\mathit{VStack}[\![\check{U}\ \tau]\!] &= \mathit{Val}[\![\check{U}\ \tau]\!]^{\perp\!\!\perp} \cap \{((\Phi_l, \mathbb{K}_l), (\Phi_r, \mathbb{K}_r))\} \\
\mathit{Shared}[\![\check{U}\ \tau]\!] &= \mathit{VStack}[\![\check{U}\ \tau]\!]^{\top\!\!\top} \\
\mathit{Stack}[\![\check{U}\ \tau]\!] &= \mathit{Shared}[\![\check{U}\ \tau]\!]^{\perp\!\!\perp}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Env}[\![\Gamma]\!] &= \{((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \mid \\
&\qquad (\forall x{:}\tau \in \Gamma.\ ((\Phi_l, x[\Sigma_l]), (\Phi_r, x[\Sigma_r])) \in \mathit{Val}[\![\tau]\!]) \wedge \\
&\qquad (\forall a{:}\tau \in \Gamma.\ ((\Phi_l, \Sigma_l, a), (\Phi_r, \Sigma_r, a)) \in \mathit{Shared}[\![\tau]\!])\}
\end{aligned}
$$

Figure 7.1. CBPVS Logical Relations

check that the environment and computation pairs are in Comp$[\![\tau]\!]$. For each computable type in general, we begin with a base definition of either their introduction or elimination forms and build up the rest of the relation with the orthogonal operations. Whereas the computation types are built from only the double orthogonal of the base definition, the shared computation types are built from more applications of the orthogonal operations. For instance, the $Val[\![\check{U}\ \tau]\!]$ relation is the orthogonal of $Elim[\![\check{U}\ \tau]\!]$ restricted to values. And we take the orthogonal of that for $VStack[\![\check{U}\ \tau]\!]$ but restrict it to value stacks. These extra steps are essential for capturing both memoization and η laws within the same relation. At the end of Figure 7.1, we include a relation on typing environments Γ. It says that related environments will have related variables. For shared variables, we must consider running the variables since they can either point to a value in the local environment, an unevaluated thunk, or an evaluated thunk.

Proposition 7.6 (Logical Relations are Kripke Relations)**.**

- *For any computation type τ, $Comp[\![\tau]\!]$ is a Kripke relation.*
- *For any shared type τ, $Shared[\![\tau]\!]$ is a Kripke relation.*
- *For any value type τ, $Val[\![\tau]\!]$ is a Kripke relation.*
- *For any type environment Γ, $Env[\![\Gamma]\!]$ is a Kripke relation.*

These in addition to their sub-components.

Proof. By mutual induction on the structure of types:

Case B:

Considering $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi_l, \mathsf{b}), (\Phi_r, \mathsf{b})) \in Val[\![B]\!]$, $((\Phi'_l, \mathsf{b}), (\Phi'_r, \mathsf{b})) \in Val[\![B]\!]$ follows trivially by definition.

Case $\tau \otimes \sigma$:

Considering future heaps $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi_l, \langle \mathbb{V}_l, \mathbb{W}_l \rangle), (\Phi_r, \langle \mathbb{V}_r, \mathbb{W}_r \rangle)) \in \mathit{Val}[\![\tau \otimes \sigma]\!]$. By definition, we know that $((\Phi_l, \mathbb{V}_l), (\Phi_r, \mathbb{V}_r)) \in \mathit{Val}[\![\tau]\!]$ and also that $((\Phi_l, \mathbb{W}_l), (\Phi_r, \mathbb{W}_r)) \in \mathit{Val}[\![\tau]\!]$. From the inductive hypotheses, we know that $((\Phi'_l, \mathbb{V}_l), (\Phi'_r, \mathbb{V}_r)) \in \mathit{Val}[\![\tau]\!]$ and $((\Phi'_l, \mathbb{W}_l), (\Phi'_r, \mathbb{W}_r)) \in \mathit{Val}[\![\tau]\!]$. By definition, we may conclude that $((\Phi'_l, \langle \mathbb{V}_l, \mathbb{W}_l \rangle), (\Phi'_r, \langle \mathbb{V}_r, \mathbb{W}_r \rangle)) \in \mathit{Val}[\![\tau \otimes \sigma]\!]$.

Case $U\ \tau$:

Considering the future heaps $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and that

$$\begin{aligned}&((\Phi_l, \{\Sigma_l, \mathtt{force} \longrightarrow M\})\\ &,(\Phi_r, \{\Sigma_l, \mathtt{force} \longrightarrow N\})) \in \mathit{Val}[\![U\ \tau]\!].\end{aligned}$$

By definition, $((\Phi_l, \Sigma_l, M), (\Phi_r, \Sigma_r, N)) \in \mathit{Comp}[\![\tau]\!]$. By the inductive hypothesis, $((\Phi'_l, \Sigma_l, M), (\Phi'_r, \Sigma_r, N)) \in \mathit{Comp}[\![\tau]\!]$. Therefore, we may conclude this case by definition.

Case $\tilde{U}\ \tau$:

Given $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi_l, \mathsf{box}\ \mathbb{V}), (\Phi_r, \mathsf{box}\ \mathbb{W})) \in \mathit{Val}[\![\tilde{U}\ \tau]\!]$. $(\mathrm{Run}(\Phi_l, \mathbb{V}), \mathrm{Run}(\Phi_r, \mathbb{W})) \in \mathit{Shared}[\![\tau]\!]$ follows by definition. By the inductive hypothesis, we know that $(\mathrm{Run}(\Phi'_l, \mathbb{V}), \mathrm{Run}(\Phi'_r, \mathbb{W})) \in \mathit{Shared}[\![\tau]\!]$. Thus, by definition, $((\Phi'_l, \mathsf{box}\ \mathbb{V}), (\Phi'_r, \mathsf{box}\ \mathbb{W})) \in \mathit{Val}[\![\tilde{U}\ \tau]\!]$.

Case τ:

Similar to the cases above, we prove that $\mathit{Intro}[\![\hat{F}\ \sigma]\!]$ and $\mathit{Elim}[\![\check{U}\ \sigma]\!]$ using the inductive hypothesis. The rest of the relations are constructed with orthogonal operations, which are Kripke relations. Note that the restrictions do not pose any problem for proving the Kripke property.

Case τ:

Similar to the cases for shared types.

Case Γ:

Follows because shared and value types are Kripke relations.

$\square$

Proposition 7.7 (Shared Logical Relation Inclusion Properties)**.**

For $\check{U}\ \tau$, $\mathit{Elim}[\![\check{U}\ \tau]\!] \subseteq \mathit{VStack}[\![\check{U}\ \tau]\!] \subseteq \mathit{Stack}[\![\check{U}\ \tau]\!]$ *and* $\mathit{Val}[\![\check{U}\ \tau]\!] \subseteq \mathit{Shared}[\![\check{U}\ \tau]\!]$*.*

For $\hat{F}\ \tau$, $\mathit{Intro}[\![\hat{F}\ \tau]\!] \subseteq \mathit{Shared}[\![\hat{F}\ \tau]\!]$ *and* $\mathit{VStack}[\![\hat{F}\ \tau]\!] \subseteq \mathit{Stack}[\![\hat{F}\ \tau]\!]$

Proof. Given $((\Phi_l, \square.\mathtt{enter} \cdot K_l), (\Phi_r, \square.\mathtt{enter} \cdot K_r)) \in \mathit{Elim}[\![\check{U}\ \tau]\!]$, we must show that the pair is also in $\mathit{VStack}[\![\check{U}\ \tau]\!]$. By definition, this is to show the pair is $\mathit{Val}[\![\check{U}\ \tau]\!]^{\perp\!\!\perp}$ restricted to value stacks. Note that these stacks are syntactically value stacks. Thus, consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, \Sigma_l, V), (\Phi'_r, \Sigma_r, W)) \in \mathit{Val}[\![\check{U}\ \tau]\!]$:

By definition, $((\Phi'_l, \Sigma_l, V), (\Phi'_r, \Sigma_r, W)) \in \mathit{Elim}[\![\check{U}\ \tau]\!]^{\top\!\!\top}$.

$((\Phi'_l, \square.\mathtt{enter} \cdot K_l), (\Phi'_r, \square.\mathtt{enter} \cdot K_r)) \in \mathit{Elim}[\![\check{U}\ \tau]\!]$, by the closure over future heaps.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel V \parallel \square.\mathtt{enter} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel W \parallel \square.\mathtt{enter} \cdot K_r \rangle\!\rangle$, by the property of the related values with the eliminations in the worlds $\Phi'_l \sqsupseteq \Phi'_l$ and $\Phi'_r \sqsupseteq \Phi'_r$.

Considering an arbitrary $((\Phi_l, \mathbb{K}_l), (\Phi_r, \mathbb{K}_r)) \in \mathit{VStack}[\![\check{U}\ \tau]\!]$, we must show that the pair is also in $\mathit{Stack}[\![\check{U}\ \tau]\!]$. By definition, this is to show the pair is $\mathit{Shared}[\![\check{U}\ \tau]\!]^{\perp\!\!\perp}$. Thus, consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, \Sigma_l, R), (\Phi'_r, \Sigma_r, S)) \in \mathit{Shared}[\![\check{U}\ \tau]\!]$:

By definition, $((\Phi'_l, \Sigma_l, R), (\Phi'_r, \Sigma_r, S)) \in \mathit{VStack}[\![\check{U}\ \tau]\!]^{\top\!\!\top}$.

$((\Phi'_l, \mathbb{K}_l), (\Phi'_r, \mathbb{K}_r)) \in \mathit{VStack}[\![\check{U}\ \tau]\!]$, by closure over future heaps.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R \parallel \mathbb{K}_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel S \parallel \mathbb{K}_r \rangle\!\rangle)$, by the property of the related expressions with the related shared stacks in the worlds $\Phi'_l \sqsupseteq \Phi'_l$ and $\Phi'_r \sqsupseteq \Phi'_r$.

Considering an arbitrary $((\Phi_l, \Sigma_l, V), (\Phi_r, \Sigma_r, W)) \in \mathit{Val}[\![\check{U}\ \tau]\!]$, we must show that the pair is also in $\mathit{Shared}[\![\check{U}\ \tau]\!]$. By definition, this is to show the pair is $\mathit{VStack}[\![\check{U}\ \tau]\!]^{\top\!\!\top}$. Thus, consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, \mathbb{K}_l), (\Phi'_r, \mathbb{K}_r) \in \mathit{VStack}[\![\check{U}\ \tau]\!]$:

By definition, $((\Phi'_l, \mathbb{K}_l), (\Phi'_r, \mathbb{K}_r) \in \mathit{Val}[\![\check{U}\ \tau]\!]^{\perp\!\!\perp}$.

$((\Phi'_l, \Sigma_l, V), (\Phi'_r, \Sigma_r, W)) \in \mathit{Val}[\![\check{U}\ \tau]\!]$, by closure over future heaps.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel V \parallel \mathbb{K}_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel W \parallel \mathbb{K}_r \rangle\!\rangle$, by the property of the related value stacks with the related values in the worlds $\Phi'_l \sqsupseteq \Phi'_l$ and $\Phi'_r \sqsupseteq \Phi'_r$.

The proofs for $\hat{F}\ \tau$ relations follow in a similar manner. □

Proposition 7.8 (Orthogonal Completeness of the Logical Relations)**.**

- *For any computable type τ, both $\mathit{Comp}[\![\tau]\!]^{\perp\!\!\perp} = \mathit{Stack}[\![\tau]\!]$ and $\mathit{Stack}[\![\tau]\!]^{\top\!\!\top} = \mathit{Comp}[\![\tau]\!]$.*

- *For any shared type τ, both*

  $\mathit{Shared}[\![\tau]\!]^{\perp\!\!\perp} = \mathit{Stack}[\![\tau]\!]$ and $\mathit{Stack}[\![\tau]\!]^{\top\!\!\top} \supseteq \mathit{Shared}[\![\tau]\!]$.

Proof. By induction on the structure of the computable type. Herein, we use triple orthogonal elimination liberally; this we may do because all semantic types are Kripke relations.

Case $\tau \longrightarrow \sigma$:

$$\begin{aligned} \mathit{Comp}[\![\tau \longrightarrow \sigma]\!] &=_{\text{defn.}} \mathit{Elim}[\![\tau \longrightarrow \sigma]\!]^{\pi} \\ &=_{\text{tri. orth.}} \mathit{Elim}[\![\tau \longrightarrow \sigma]\!]^{\pi\perp\!\!\!\perp\pi} \\ &=_{\text{defn.}} \mathit{Comp}[\![\tau \longrightarrow \sigma]\!]^{\perp\!\!\!\perp\pi} \\ &=_{\text{defn.}} \mathit{Stack}[\![\tau \longrightarrow \sigma]\!]^{\pi} \end{aligned}$$

and

$$\mathit{Stack}[\![\tau \longrightarrow \sigma]\!] =_{\text{defn.}} \mathit{Comp}[\![\tau \longrightarrow \sigma]\!]^{\perp\!\!\!\perp}.$$

Case $\tau \mathbin{\&} \sigma$ follows in a similar manner to the case above since they were both defined via their eliminations.

Case $F\ \tau$:

$$\mathit{Comp}[\![F\ \tau]\!] =_{\text{defn.}} \mathit{Stack}[\![F\ \tau]\!]^{\pi}$$

and

$$\begin{aligned} \mathit{Stack}[\![F\ \tau]\!] &=_{\text{defn.}} \mathit{Intro}[\![F\ \tau]\!]^{\perp\!\!\!\perp} \\ &=_{\text{tri. orth.}} \mathit{Intro}[\![F\ \tau]\!]^{\perp\!\!\!\perp\pi\perp\!\!\!\perp} \\ &=_{\text{defn.}} \mathit{Stack}[\![F\ \tau]\!]^{\pi\perp\!\!\!\perp} \\ &=_{\text{defn.}} \mathit{Comp}[\![F\ \tau]\!]^{\perp\!\!\!\perp}. \end{aligned}$$

Case $\hat{F}\ \tau$:

$$\begin{aligned} \mathit{Shared}[\![\hat{F}\ \tau]\!] &=_{\text{defn.}} \mathit{VStack}[\![\hat{F}\ \tau]\!]^{\pi} \\ &\subseteq_{\text{contra.}} \mathit{Stack}[\![\hat{F}\ \tau]\!]^{\pi} \end{aligned}$$

and

$$\mathit{Stack}[\![\hat{F}\ \tau]\!] =_{\text{defn.}} \mathit{Shared}[\![\hat{F}\ \tau]\!]^{\perp\!\!\!\perp}$$

Case $\check{U}\ \tau$:

$$\begin{aligned} \mathit{Shared}[\![\check{U}\ \tau]\!] &= \mathit{VStack}[\![\check{U}\ \tau]\!]^{\top\!\top} \\ &\subseteq_{\text{contra.}} \mathit{Stack}[\![\check{U}\ \tau]\!]^{\top\!\top} \end{aligned}$$

and

$$\mathit{Stack}[\![\check{U}\ \tau]\!] =_{\text{defn.}} \mathit{Shared}[\![\check{U}\ \tau]\!]^{\perp\!\perp}$$

□

Lemma 7.1 (Related Heap Extension)**.**

$(\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r) \in \mathit{Env}[\![\Gamma]\!]$ and $((\Phi_l, \Sigma_l\,\mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), R), (\Phi_r, \Sigma_r\,\mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r), S)) \in \mathit{Shared}[\![\tau]\!]$ imply

$$\begin{aligned} &(((\Phi_l, l_a \mapsto \{\Sigma_l\,\mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), R\}), (\Sigma_l, l_a/a)) \\ &, ((\Phi_r, l_a \mapsto \{\Sigma_r\,\mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r), S\}), (\Sigma_r, l_a/a))) \in \mathit{Env}[\![\Gamma, a{:}\tau]\!]. \end{aligned}$$

Proof. We only need to show that

$$\begin{aligned} &(((\Phi_l, l_a \mapsto \{\Sigma_l\,\mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), R\}), (\Sigma_l, l_a/a), a) \\ &, ((\Phi_r, l_a \mapsto \{\Sigma_r\,\mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r), S\}), (\Sigma_r, l_a/a), a)) \in \mathit{Shared}[\![\tau]\!] \end{aligned}$$

given our assumption and the definition of $\mathit{Env}[\![\Gamma, a{:}\tau]\!]$. By definition for any shared type, it is enough to show that the pair is in $\mathit{VStack}[\![\tau]\!]^{\top\!\top}$. Thus, suppose $\Phi'_l \sqsupseteq (\Phi_l, l_a \mapsto \{\Sigma_l\,\mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), R\})$, $\Phi'_r \sqsupseteq (\Phi_r, l_a \mapsto \{\Sigma_r\,\mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r), S\})$, and $((\Phi'_l, \mathbb{K}_l), (\Phi'_r, \mathbb{K}_r)) \in \mathit{VStack}[\![\tau]\!]$:

By the definition of heap extension, we know $\Phi'_l = (\Phi'_{l0}, l_a \mapsto \{\Sigma_l\,\mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), R\})\Phi'_{l1}$ and $\Phi'_r = (\Phi'_{r0}, l_a \mapsto \{\Sigma_r\,\mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r), S\})\Phi'_{r1}$. Additionally, we know $\Phi'_{l0} \sqsupseteq \Phi_l$ and $\Phi'_{r0} \sqsupseteq \Phi_r$.

On the left side, there is the transition

$$\begin{aligned} \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel a \parallel \mathbb{K}_l \rangle\!\rangle &\longmapsto \\ \langle\!\langle \Phi'_{l0} \parallel \Sigma_l \, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l) \parallel R \parallel (\Phi'_{l1}, l_a) \cdot \mathbb{K}_l \rangle\!\rangle & \end{aligned}$$

And similarly on the right.

$$\begin{aligned} &((\Phi'_{l0}, \Sigma_l \, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), R) \\ &, (\Phi'_{r0}, \Sigma_r \, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r), S)) \in \mathit{Shared}[\![\tau]\!] \end{aligned}$$

follows from our initial assumption with closure over accessible heaps.

$$\begin{aligned} \langle\!\langle \Phi'_{l0} \parallel \Sigma_l \, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l) \parallel R \parallel \mathbb{K}_l \rangle\!\rangle &\simeq \\ \langle\!\langle \Phi'_{r0} \parallel \Sigma_r \, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r) \parallel S \parallel \mathbb{K}_r \rangle\!\rangle & \end{aligned}$$

by the the property of the related expressions immediately above with $\Phi'_l \sqsupseteq \Phi'_{l0}$, $\Phi'_r \sqsupseteq \Phi'_{r0}$, and $((\Phi'_l, \mathbb{K}_l), (\Phi'_r, \mathbb{K}_r)) \in \mathit{VStack}[\![\tau]\!]$.

$$\begin{aligned} \langle\!\langle \Phi'_{l0} \parallel \Sigma_l \, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l) \parallel R \parallel \mathbb{K}_l \rangle\!\rangle &\simeq \\ \langle\!\langle \Phi'_{l0} \parallel \Sigma_l \, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l) \parallel R \parallel (\Phi'_{l1}, l_a) \cdot \mathbb{K}_l \rangle\!\rangle & \end{aligned}$$

by the memoizing closure property of $(\simeq)$. We have a similar fact on the right side.

$$\begin{aligned} \langle\!\langle \Phi'_l \parallel \Sigma_l \, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l) \parallel a \parallel \mathbb{K}_l \rangle\!\rangle &\simeq \\ \langle\!\langle \Phi'_r \parallel \Sigma_r \, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r) \parallel a \parallel \mathbb{K}_r \rangle\!\rangle & \end{aligned}$$

by the closure properties of $(\simeq)$.

□

Lemma 7.2 (Build then Run). *If* $((\Phi_l, \Sigma_l, V), (\Phi_r, \Sigma_r, W)) \in \mathit{Shared}[\![\tau]\!]$, *then* $(\mathrm{Run}(\Phi_l, \mathrm{Build}_V(\Sigma_l, V)), \mathrm{Run}(\Phi_r, \mathrm{Build}_V(\Sigma_r, W))) \in \mathit{Shared}[\![\tau]\!]$.

Proof. Follows from definitions. Note that the run function places locations in the smallest environment necessary. □

What follows are three lemmas that behave similarly to show that inlining, garbage collection, and substitutions yield logically related expressions. They describe properties of our delayed substitution semantics; that we may eagerly substitute values and not change the observable behavior and that we may remove unused variables. Note that this second property is implied by the first.

Lemma 7.3 (Contextual Inlining). *Given any typed substitutable value* $\Gamma \vdash V : \tau$ *and related environments* $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, *we know the following:*

$$\frac{\Gamma, x{:}\tau \vdash C[x] : \sigma}{((\Phi_l, (\Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x), C[x]), (\Phi_r, (\Sigma_r, \mathrm{Build}_V(\Sigma_r, V)/x), C[V])) \in \mathit{Comp}[\![\sigma]\!]}$$

$$\frac{\Gamma, x{:}\tau \vdash C[x] : \sigma}{((\Phi_l, \mathrm{Build}_V((\Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x), C[x])), (\Phi_r, \mathrm{Build}_V(\Sigma_r, C[V]))) \in \mathit{Val}[\![\sigma]\!]}$$

$$\frac{\Gamma, x{:}\tau \vdash C[x] : \sigma}{((\Phi_l, (\Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x), C[x]), (\Phi_r, \Sigma_r, C[V])) \in \mathit{Shared}[\![\sigma]\!]}$$

$$\frac{\Gamma, x{:}\tau \vdash C[x] : \sigma}{(((\Phi_l, l \mapsto \mathrm{Build}_V(\Sigma_l, V)), (\Sigma_l, l/x), C[x]), (\Phi_r, \Sigma_r, C[V])) \in \mathit{Shared}[\![\sigma]\!]}$$

$$\frac{\Gamma, x{:}\tau \vdash \varsigma[x] : \Gamma'}{((\Phi_l, \mathrm{Build}_\varsigma((\Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x), \varsigma[x])), (\Phi_r, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma[V]))) \in \mathit{Env}[\![\Gamma']\!]}$$

Proof. By mutual induction on the typing derivations. □

Lemma 7.4 (Garbage Collection). *Given related environments* $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$*, we know the following:*

$$\frac{\Gamma \vdash M : \sigma}{((\Phi_l, (\Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x), M), (\Phi_r, \Sigma_r, M)) \in Comp[\![\sigma]\!]}$$

$$\frac{\Gamma \vdash V : \sigma}{((\Phi_l, \mathrm{Build}_V((\Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x), V)), (\Phi_r, \mathrm{Build}_V(\Sigma_r, V))) \in Val[\![\sigma]\!]}$$

$$\frac{\Gamma \vdash R : \sigma}{((\Phi_l, (\Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x), R), (\Phi_r, \Sigma_r, R)) \in Shared[\![\sigma]\!]}$$

$$\frac{\Gamma \vdash R : \sigma}{(((\Phi_l, l \mapsto \mathrm{Build}_V(\Sigma_l, V)), (\Sigma_l, l/x), R), (\Phi_r, \Sigma_r, R)) \in Shared[\![\sigma]\!]}$$

$$\frac{\Gamma \vdash \varsigma : \Gamma'}{((\Phi_l, \mathrm{Build}_\varsigma((\Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x), \varsigma)), (\Phi_r, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma))) \in Env[\![\Gamma']\!]}$$

Proof. By mutual induction on the typing derivations. □

Corollary 7.1 (Substitutive Inlining). *Given any typed substitutable value* $\Gamma \vdash \varsigma : \Gamma'$ *and* $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$*, we know the following:*

$$\frac{\Gamma\Gamma' \vdash M : \sigma}{((\Phi_l, \Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma), M), (\Phi_r, \Sigma_r, M[\varsigma])) \in Comp[\![\sigma]\!]}$$

$$\frac{\Gamma\Gamma' \vdash W : \sigma}{((\Phi_l, \mathrm{Build}_V(\Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma), W)), (\Phi_r, \mathrm{Build}_V(\Sigma_r, W[\varsigma]))) \in Val[\![\sigma]\!]}$$

$$\frac{\Gamma\Gamma' \vdash R : \sigma}{((\Phi_l, \Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma), R), (\Phi_r, \Sigma_r, R[\varsigma])) \in Shared[\![\sigma]\!]}$$

$$\frac{\Gamma\Gamma' \vdash \varsigma' : \Gamma''}{((\Phi_l, \mathrm{Build}_\varsigma(\Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma), \varsigma')), (\Phi_r, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma'[\varsigma]))) \in Env[\![\Gamma'']\!]}$$

Lemma 7.5 (Building Related Stacks from Eval Contexts). *If* $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$, $((\Phi_l, K_l), (\Phi_r, K_r)) \in Stack[\![\tau]\!]$, $\mathrm{Build}_K(\Phi_l, \Sigma_l, E, K_l) = (\Phi_l', \Sigma_l', K_l')$*, and* $\mathrm{Build}_K(\Phi_r, \Sigma_r, E, K_r) = (\Phi_r', \Sigma_r', K_r')$*, then* $((\Phi_l', K_l'), (\Phi_r', K_r')) \in Stack[\![\tau]\!]$.

Proof. By induction on the evaluation context. □

7.2 Semantic Equality

Semantic equivalence is built from the logical relations by considering related instantiations of the type environment and then having related expressions for the type.

Definition 7.5 (Semantic Equality)**.**

$$
\begin{aligned}
\Gamma \vDash M \approx N : \tau &\overset{\text{def}}{=} \forall((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!].\, ((\Phi_l, \Sigma_l, M), (\Phi_r, \Sigma_r, N)) \in \mathit{Comp}[\![\tau]\!] \\
\Gamma \vDash R \approx S : \tau &\overset{\text{def}}{=} \forall((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!].\, ((\Phi_l, \Sigma_l, R), (\Phi_r, \Sigma_r, S)) \in \mathit{Shared}[\![\tau]\!] \\
\Gamma \vDash V \approx W : \tau &\overset{\text{def}}{=} \forall((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]. \\
&\qquad ((\Phi_l, \mathrm{Build}_V(\Sigma_l, V)), (\Phi_r, \mathrm{Build}_V(\Sigma_r, W))) \in \mathit{Val}[\![\tau]\!] \\
\Gamma \vDash \varsigma_l \approx \varsigma_r : \Gamma' &\overset{\text{def}}{=} \forall((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]. \\
&\qquad ((\Phi_l, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l)), (\Phi_r, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r))) \in \mathit{Env}[\![\Gamma']\!]
\end{aligned}
$$

From here, we prove compatibility propositions that align with all of the typing rules for CBPVS in Figure 5.6. These propositions will be enough to show that our semantic equivalence relations are indeed a partial semantic congruence relation; partial in the sense that expressions need to be well-typed. This sections ends with a proof of the fundamental lemma that syntactic equivalence implies semantic equivalence; the proof of which has a case for each axiom of our theory.

Proposition 7.9 (Compatibility *var*)**.** *If* $x{:}\tau \in \Gamma$*, then* $\Gamma \vDash x \approx x : \tau$*.*

Proof. Consider an arbitrary $x{:}\tau \in \Gamma$ and $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$. By the definition of $\mathit{Env}[\![\Gamma]\!]$, $((\Phi_l, x[\Sigma_l]), (\Phi_r, x[\Sigma_r])) \in \mathit{Val}[\![\tau]\!]$. Finally, from the definition of Build_V, we may conclude that $((\Phi_l, \mathrm{Build}_V(\Sigma_l, x)), (\Phi_r, \mathrm{Build}_V(\Sigma_r, x))) \in \mathit{Val}[\![\tau]\!]$. □

Proposition 7.10 (Compatibility *b*)**.** $\Gamma \vDash \mathsf{b} \approx \mathsf{b} : B$.

Proof. If $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, then, immediately from definitions, we know $((\Phi_l, \mathrm{Build}_V(\Sigma_l, \mathsf{b})), (\Phi_r, \mathrm{Build}_V(\Sigma_r, \mathsf{b}))) \in \mathit{Val}[\![B]\!]$. □

Proposition 7.11 (Compatibility $\to_I$). *If* $\Gamma, x{:}\tau \vDash M \approx N : \sigma$, *then* $\Gamma \vDash \lambda x.\, M \approx \lambda x.\, N : \tau \to \sigma$.

Proof. Considering an arbitrary $\Gamma, x{:}\tau \vDash M \approx N : \sigma$, we show $\Gamma \vDash \lambda x.\, M \approx \lambda x.\, N : \tau \to \sigma$ by further supposing arbitrary environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ then showing $((\Phi_l, \Sigma_l, \lambda x.\, M), (\Phi_r, \Sigma_r, \lambda x.\, N)) \in \mathit{Comp}[\![\tau \to \sigma]\!]$. By definition, this is equivalent to showing that the pair is in $\mathit{Elim}[\![\tau \to \sigma]\!]^{\top\!\!\top}$. Thus, let us consider an arbitrary $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, $((\Phi'_l, \square\ \mathbb{V} \cdot K_l), (\Phi'_r, \square\ \mathbb{W} \cdot K_r)) \in \mathit{Elim}[\![\tau \to \sigma]\!]$:

By definition, we have that $((\Phi'_l, \mathbb{V}), (\Phi'_r, \mathbb{W})) \in \mathit{Val}[\![\tau]\!]$ and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\sigma]\!]$.

On the left, we have the transition

$$\begin{aligned}\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \lambda x.\, M \parallel \square\ \mathbb{V} \cdot K_l \rangle\!\rangle \quad &\longmapsto \\ \langle\!\langle \Phi'_l \parallel \Sigma_l, \mathbb{V}/x \parallel M \parallel K_l \rangle\!\rangle&\end{aligned}$$

and similarly on the right side.

$((\Phi'_l, (\Sigma_l, \mathbb{V}/x)), (\Phi'_r, (\Sigma_r, \mathbb{W}/x))) \in \mathit{Env}[\![\Gamma, x{:}\tau]\!]$ by definition with our assumed related environments. Note that $((\Phi'_l, \Sigma_l), (\Phi'_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ because the relation is closed over accessible worlds.

$((\Phi'_l, (\Sigma_l, \mathbb{V}/x), M), (\Phi'_r, (\Sigma_r, \mathbb{W}/x), N)) \in \mathit{Comp}[\![\sigma]\!]$ from our initial assumption about the function bodies with the related extended environments above.

And by Proposition 7.8 with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\sigma]\!]$, we have $\langle\!\langle \Phi'_l \parallel \Sigma_l, \mathbb{V}/x \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r, \mathbb{W}/x \parallel N \parallel K_r \rangle\!\rangle$.

With our closure properties of $(\simeq)$, we have $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \lambda x.\, M \parallel \square\ \mathbb{V} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel \lambda x.\, N \parallel \square\ \mathbb{W} \cdot K_r \rangle\!\rangle$.

$\square$

Proposition 7.12 (Compatibility $\to_E$). *If* $\Gamma \vDash M \approx N : \sigma \to \tau$ *and* $\Gamma \vDash V \approx W : \sigma$, *then* $\Gamma \vDash M\ V \approx N\ W : \tau$.

Proof. Given $\Gamma \vDash M \approx N : \sigma \to \tau$ and $\Gamma \vDash V \approx W : \sigma$, we show $\Gamma \vDash M\ V \approx N\ W : \tau$ by considering $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ and then proving $((\Phi_l, \Sigma_l, M\ V), (\Phi_r, \Sigma_r, N\ W)) \in \mathit{Comp}[\![\tau]\!]$. Since this by Proposition 7.8 is equivalent to showing that the pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$, we further consider an arbitrary $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

By our initial assumptions with the related environments $((\Phi'_l, \Sigma_l), (\Phi'_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ (using the Kripke property to advance to a future world), we may conclude both that $((\Phi'_l, \Sigma_l, M), (\Phi'_r, \Sigma_r, N)) \in \mathit{Comp}[\![\sigma \to \tau]\!]$ and $((\Phi'_l, \mathrm{Build}_V(\Sigma_l, V)), (\Phi'_r, \mathrm{Build}_V(\Sigma_r, W))) \in \mathit{Val}[\![\sigma]\!]$.

On the left side, we have the transition

$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M\ V \parallel K_l \rangle\!\rangle \longmapsto \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \parallel \square\ \mathrm{Build}_V(\Sigma_l, V) \cdot K_l \rangle\!\rangle$$

Similarly on the right side.

$((\Phi'_l, \square\ \mathrm{Build}_V(\Sigma_l, V) \cdot K_l), (\Phi'_r, \square\ \mathrm{Build}_V(\Sigma_l, W) \cdot K_l)) \in \mathit{Elim}[\![\sigma \to \tau]\!]$ by definition.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \parallel \square\ \mathrm{Build}_V(\Sigma_l, V) \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel N \parallel \square\ \mathrm{Build}_V(\Sigma_r, W) \cdot K_r \rangle\!\rangle$ by the definition of $\mathit{Comp}[\![\sigma \to \tau]\!]$ with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and the above related stacks.

By our closure properties of $(\simeq)$, we may conclude that $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M\ V \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel N\ W \parallel K_r \rangle\!\rangle$.

$\square$

Proposition 7.13 (Compatibility $\&_I$). *If* $\Gamma \vDash M_l \approx M_r : \tau$ *and* $\Gamma \vDash N_l \approx N_r : \sigma$, *then* $\Gamma \vDash \{\texttt{fst} \longrightarrow M_l; \texttt{snd} \longrightarrow N_l\} \approx \{\texttt{fst} \longrightarrow M_r; \texttt{snd} \longrightarrow N_r\} : \tau \,\&\, \sigma$.

Proof. Given $\Gamma \vDash M_l \approx M_l : \tau$ and $\Gamma \vDash N_r \approx N_r : \sigma$, we prove $\Gamma \vDash \{\texttt{fst} \longrightarrow M_l; \texttt{snd} \longrightarrow N_l\} \approx \{\texttt{fst} \longrightarrow M_r; \texttt{snd} \longrightarrow N_r\} : \tau \,\&\, \sigma$ by considering $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ and then proving that

$$((\Phi_l, \Sigma_l, \{\texttt{fst} \longrightarrow M_l; \texttt{snd} \longrightarrow N_l\})$$
$$, (\Phi_r, \Sigma_r, \{\texttt{fst} \longrightarrow M_r; \texttt{snd} \longrightarrow N_r\})) \in \mathit{Comp}[\![\tau \,\&\, \sigma]\!].$$

By definition, this is equivalent to showing that the pair is in $\mathit{Elim}[\![\tau \,\&\, \sigma]\!]^{\top\!\top}$. Thus, considering an arbitrary $\Phi'_l \sqsupseteq \Phi_l$,$\Phi'_r \sqsupseteq \Phi_r$, and element of $\mathit{Elim}[\![\tau \,\&\, \sigma]\!]$ there are two cases to consider:

Case $((\Phi'_l, \square.\texttt{fst} \cdot K_l), (\Phi'_r, \square.\texttt{fst} \cdot K_r))$:

$((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$ follows from the definition of $\mathit{Elim}[\![\tau \,\&\, \sigma]\!]$.

By the first of our initial assumptions with $((\Phi'_l, \Sigma_l), (\Phi'_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ (closed with accessible worlds), $((\Phi'_l, \Sigma_l, M_l), (\Phi'_r, \Sigma_r, M_l)) \in \mathit{Comp}[\![\tau]\!]$ follows. Proposition 7.8 with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, the related stacks above, we have that $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M_l \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M_r \parallel K_r \rangle\!\rangle$.

On the left side, we have the transition

$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\texttt{fst} \longrightarrow M_l; \texttt{snd} \longrightarrow N_l\} \parallel \square.\texttt{fst} \cdot K_l \rangle\!\rangle \quad \longmapsto$$
$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M_l \parallel K_l \rangle\!\rangle$$

Similarly on the right.

By the closure properties of $(\simeq)$, we have $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\texttt{fst} \longrightarrow M_l; \texttt{snd} \longrightarrow N_l\} \parallel \square.\texttt{fst} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel \{\texttt{fst} \longrightarrow M_r; \texttt{snd} \longrightarrow N_r\} \parallel \square.\texttt{fst} \cdot K_r \rangle\!\rangle$.

Case $((\Phi'_l, \square.\mathtt{snd} \cdot K_l), (\Phi'_r, \square.\mathtt{snd} \cdot K_r))$:

This follows in a similar manner to the case above by using the second initial assumption.

$\square$

Proposition 7.14 (Compatibility $\&_{E1}$). *If* $\Gamma \vDash M \approx N : \tau \,\&\, \sigma$*, then* $\Gamma \vDash M.\mathtt{fst} \approx N.\mathtt{fst} : \tau$*.*

Proof. Given $\Gamma \vDash M \approx N : \tau \,\&\, \sigma$, we show $\Gamma \vDash M.\mathtt{fst} \approx N.\mathtt{fst} : \tau$ by first considering $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$ and then proving $((\Phi_l, \Sigma_l, M.\mathtt{fst}), (\Phi_r, \Sigma_r, N.\mathtt{fst})) \in Comp[\![\tau]\!]$. Using Proposition 7.8, this is the same as showing that the pair is in $Stack[\![\tau]\!]^{\top\!\top}$. Thus, consider further an arbitrary $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$ $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in Stack[\![\tau]\!]$:

$((\Phi'_l, \Sigma_l, M), (\Phi'_r, \Sigma_r, N)) \in Comp[\![\tau \,\&\, \sigma]\!]$, by our initial assumption with the related environments (note closure over accessible worlds).

On the left side, we have the transition

$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M.\mathtt{fst} \parallel K_l \rangle\!\rangle \quad \longmapsto$$
$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \parallel \square.\mathtt{fst} \cdot K_l \rangle\!\rangle$$

And similarly on the right.

$((\Phi'_l, \square.\mathtt{fst} \cdot K_l), (\Phi'_r, \square.\mathtt{fst} \cdot K_r)) \in Stack[\![\tau \,\&\, \sigma]\!]$ by definition. This and Proposition 7.8 with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and $((\Phi'_l, \Sigma_l, M), (\Phi'_r, \Sigma_r, N)) \in Comp[\![\tau \,\&\, \sigma]\!]$ yields that $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \parallel \square.\mathtt{fst} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel N \parallel \square.\mathtt{fst} \cdot K_r \rangle\!\rangle$.

By the closure properties of $(\simeq)$, we have $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M.\mathtt{fst} \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel N.\mathtt{fst} \parallel K_r \rangle\!\rangle$.

$\square$

Proposition 7.15 (Compatibility $\&_{E2}$). *If* $\Gamma \vDash M \approx N : \tau \,\&\, \sigma$*, then* $\Gamma \vDash M.\mathsf{snd} \approx N.\mathsf{snd} : \sigma$*.*

Proof. Follows in a similar manner to compatibility for $\&_{E1}$. □

Proposition 7.16 (Compatibility $\otimes_I$). *If* $\Gamma \vDash V_l \approx V_r : \tau$ *and* $\Gamma \vDash W_l \approx W_r : \sigma$*, then* $\Gamma \vDash \langle V_l, W_l\rangle \approx \langle V_r, W_r\rangle : \tau \otimes \sigma$*.*

Proof. Given $\Gamma \vDash V_l \approx V_l : \tau$ and $\Gamma \vDash W_r \approx W_r : \sigma$, we show $\Gamma \vDash \langle V_l, W_l\rangle \approx \langle V_r, W_r\rangle : \tau \otimes \sigma$ by considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$ and then showing that

$$\begin{aligned}&((\Phi_l, \mathrm{Build}_V(\Sigma_l, \langle V_l, W_l\rangle))\\ &,(\Phi_r, \mathrm{Build}_V(\Sigma_r, \langle V_r, W_r\rangle))) \in Val[\![\tau \otimes \sigma]\!].\end{aligned}$$

Next, $\mathrm{Build}_V(\Sigma_l, \langle V_l, W_l\rangle) = \langle \mathrm{Build}_V(\Sigma_l, V_l), \mathrm{Build}_V(\Sigma_l, W_l)\rangle$ and similarly for the right side, follow by definition of building values. From our first initial assumption with the assumed related environments, we know $((\Phi_l, \mathrm{Build}_V(\Sigma_l, V_l)), (\Phi_r, \mathrm{Build}_V(\Sigma_r, V_r))) \in Val[\![\tau]\!]$ and similarly for the other sub-component. We can conclude by the definition of $Val[\![\tau \otimes \sigma]\!]$. □

Proposition 7.17 (Compatibility $\otimes_E$). *If* $\Gamma \vDash V \approx W : \sigma \otimes \rho$ *and* $\Gamma, x{:}\sigma, y{:}\rho \vDash P \approx Q : \tau$*, then* $\Gamma \vDash \mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow P\} \approx \mathsf{case}\ W\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow Q\} : \tau$*.*

Proof. Given that $\Gamma \vDash V \approx W : \sigma \otimes \rho$ and $\Gamma, x{:}\sigma, y{:}\rho \vDash P \approx Q : \tau$, $\Gamma \vDash \mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow P\} \approx \mathsf{case}\ W\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow Q\} : \tau$ is true by considering $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$ and then proving

$$\begin{aligned}&((\Phi_l, \Sigma_l, \mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow M\})\\ &,(\Phi_r, \Sigma_r, \mathsf{case}\ W\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow N\})) \in Comp[\![\tau]\!].\end{aligned}$$

We have chosen τ to be a computation type, $P = M$, and $Q = N$, but the proof follows the same for shared types since we have the same transitions. By Proposition 7.8, this is the same as showing that the pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\!\top}$. Thus, we suppose an arbitrary $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

By the first initial assumption with the related environments $((\Phi'_l, \Sigma_l), (\Phi'_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ (using the Kripke property to advance to a future world), we know

$$\begin{aligned}&((\Phi'_l, \mathrm{Build}_V(\Sigma_l, V))\\&, (\Phi'_r, \mathrm{Build}_V(\Sigma_r, W))) \in \mathit{Val}[\![\sigma \otimes \rho]\!].\end{aligned}$$

By this definition, we also know $\mathrm{Build}_V(\Sigma_l, V) = \langle \mathbb{V}, \mathbb{V} \rangle$ and $\mathrm{Build}_V(\Sigma_r, W) = \langle \mathbb{W}, \mathbb{W}' \rangle$ where the sub-parts are related in world Φ'_l and Φ'_r respectively.

On the left side, we have the transition

$$\begin{aligned}&\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y \rangle \to M\} \parallel K_l \rangle\!\rangle \quad \longmapsto\\&\qquad\qquad \langle\!\langle \Phi'_l \parallel \Sigma_l, \mathbb{V}/x, \mathbb{V}'/y \parallel M \parallel K_l \rangle\!\rangle\end{aligned}$$

And similarly on the right side.

$$\begin{aligned}&((\Phi'_l, (\Sigma_l, \mathbb{V}/x, \mathbb{V}'/y))\\&, (\Phi'_r, (\Sigma_r, \mathbb{W}/x, \mathbb{W}'/y))) \in \mathit{Env}[\![\Gamma, x{:}\sigma, y{:}\rho]\!],\end{aligned}$$

by definition. And thus, we may use our second initial assumption with these related environments to conclude that

$$\begin{aligned}&((\Phi'_l, (\Sigma_l, \mathbb{V}/x, \mathbb{V}'/y), M),\\&(\Phi'_r, (\Sigma_r, \mathbb{W}/x, \mathbb{W}'/y), N)) \in \mathit{Comp}[\![\tau]\!].\end{aligned}$$

By Proposition 7.8, the property of our related stacks in combination with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, these related computations, gives us $\langle\!\langle \Phi'_l \parallel \Sigma_l, \mathbb{V}/x, \mathbb{V}'/y \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r, \mathbb{W}/x, \mathbb{W}'/y \parallel N \parallel K_r \rangle\!\rangle$.

We may conclude $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \texttt{case}\ V\ \texttt{of}\ \{\langle x, y\rangle \rightarrow M\} \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel \texttt{case}\ W\ \texttt{of}\ \{\langle x, y\rangle \rightarrow N\} \parallel K_r \rangle\!\rangle$ by the closure rules for $(\simeq)$.

□

Proposition 7.18 (Compatibility U_I). *If* $\Gamma \vDash \varsigma_l \approx \varsigma_r : \Gamma'$ *and* $\Gamma\Gamma' \vDash M \approx N : \tau$, *then* $\Gamma \vDash \{\varsigma_l, \texttt{force} \rightarrow M\} \approx \{\varsigma_r, \texttt{force} \rightarrow N\} : U\ \tau$.

Proof. Given $\Gamma \vDash \varsigma_l \approx \varsigma_r : \Gamma'$ and $\Gamma\Gamma' \vDash M \approx N : \tau$, we show that $\Gamma \vDash \{\varsigma_l, \texttt{force} \rightarrow M\} \approx \{\varsigma_r, \texttt{force} \rightarrow N\} : U\ \tau$ by considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$ then proving

$$\begin{aligned}&((\Phi_l, \mathrm{Build}_V(\Sigma_l, \{\varsigma_l, \texttt{force} \rightarrow M\}))\\ &, (\Phi_r, \mathrm{Build}_V(\Sigma_r, \{\varsigma_r, \texttt{force} \rightarrow N\}))) \in Val[\![U\ \tau]\!].\end{aligned}$$

By the definition of building machine values and $Val[\![U\ \tau]\!]$, this can be proved by showing

$$\begin{aligned}&((\Phi_l, \Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), M)\\ &, (\Phi_r, \Sigma_r\, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r), N)) \in Comp[\![\tau]\!].\end{aligned}$$

Since we already know $((\Phi_l, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l)), (\Phi_r, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r))) \in Env[\![\Gamma']\!]$ from our first initial assumption with the environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$, the definition of $Env[\![\Gamma\Gamma']\!]$ allows us to conclude $((\Phi_l, \Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l)), (\Phi_r, \Sigma_r\, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r))) \in Env[\![\Gamma\Gamma']\!]$. We may then conclude this proof by our second initial assumption with these extended, related environments. □

Proposition 7.19 (Compatibility U_E). *If* $\Gamma \vDash V \approx W : U\ \tau$, *then* $\Gamma \vDash V.\texttt{force} \approx W.\texttt{force} : \tau$.

Proof. Given $\Gamma \vDash V \approx W : U\ \tau$, showing $\Gamma \vDash V.\mathtt{force} \approx W.\mathtt{force} : \tau$ requires considering $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$ and then proving $((\Phi_l, \Sigma_l, V.\mathtt{force}), (\Phi_r, \Sigma_r, W.\mathtt{force})) \in Comp[\![\tau]\!]$. By Proposition 7.8, it suffices to show that the pair is in $Stack[\![\tau]\!]^{\top\!\top}$. Thus, consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in Stack[\![\tau]\!]$:

By our initial assumption with the related environments $((\Phi'_l, \Sigma_l), (\Phi'_r, \Sigma_r)) \in Env[\![\Gamma]\!]$ (using the Kripke property), we have $((\Phi'_l, \mathrm{Build}_V(\Sigma_l, V)), (\Phi'_r, \mathrm{Build}_V(\Sigma_r, W))) \in Val[\![U\ \tau]\!]$. By the definition of building values and $Val[\![U\ \tau]\!]$, $\mathrm{Build}_V(\Sigma_l, V) = \{\Sigma'_l, \mathtt{force} \longrightarrow M\}$ and $\mathrm{Build}_V(\Sigma_r, W) = \{\Sigma'_r, \mathtt{force} \longrightarrow N\}$ where we know the computations are related, *i.e.* $((\Phi'_l, \Sigma'_l, M), (\Phi'_r, \Sigma'_r, N)) \in Comp[\![\tau]\!]$.

On the left side, we have the transition

$$\begin{aligned} \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel V.\mathtt{force} \parallel K_l \rangle\!\rangle \quad &\longmapsto \\ \langle\!\langle \Phi'_l \parallel \Sigma'_l \parallel M \parallel K_l \rangle\!\rangle \end{aligned}$$

And similarly for the right side.

And by Proposition 7.8, the property of the related stacks with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and the related computations above, we can conclude that $\langle\!\langle \Phi'_l \parallel \Sigma'_l \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma'_r \parallel M \parallel K_r \rangle\!\rangle$.

We may conclude that $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel V.\mathtt{force} \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel W.\mathtt{force} \parallel K_r \rangle\!\rangle$ by the closure properties of $(\simeq)$.

□

Proposition 7.20 (Compatibility F_I). *If* $\Gamma \vDash V \approx W : \tau$, *then* $\Gamma \vDash \mathtt{ret}\ V \approx \mathtt{ret}\ W : F\ \tau$.

Proof. Given $\Gamma \vDash V \approx W : \tau$, showing $\Gamma \vDash \mathtt{ret}\ V \approx \mathtt{ret}\ W : F\ \tau$ requires considering $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$ and then proving $((\Phi_l, \Sigma_l, \mathtt{ret}\ V), (\Phi_r, \Sigma_r, \mathtt{ret}\ W)) \in$

$Comp[\![F\ \tau]\!]$. By the definition of $Comp[\![F\ \tau]\!]$ and double orthogonal inclusion, it is enough to show that this triple is in $Intro[\![F\ \tau]\!]$. This follows by combining our initial assumption with the related environments. $\square$

Proposition 7.21 (Compatibility F_E). *If* $\Gamma \vDash M \approx N : F\ \sigma$ *and* $\Gamma, x{:}\sigma \vDash P \approx Q : \tau$, *then* $\Gamma \vDash M\ \mathtt{to}\ x\ \mathtt{in}\ P \approx N\ \mathtt{to}\ x\ \mathtt{in}\ Q : \tau$.

Proof. Given $\Gamma \vDash M \approx N : F\ \sigma$ and $\Gamma, x{:}\sigma \vDash P \approx Q : \tau$, showing $\Gamma \vDash M\ \mathtt{to}\ x\ \mathtt{in}\ P \approx N\ \mathtt{to}\ x\ \mathtt{in}\ Q : \tau$ requires considering some $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$ and then proving that $((\Phi_l, \Sigma_l, M\ \mathtt{to}\ x\ \mathtt{in}\ R), (\Phi_r, \Sigma_r, N\ \mathtt{to}\ x\ \mathtt{in}\ S)) \in Shared[\![\tau]\!]$. We pick τ to be a shared computation, $P = R$, and $Q = S$, but the proof follows the same for computations types since we have the same transitions. By Proposition 7.8, it suffices to show that the pair $Stack[\![\tau]\!]^{\top\!\top}$. Thus, consider an arbitrary $\Phi_l' \sqsupseteq \Phi_l$, $\Phi_r' \sqsupseteq \Phi_r$, and $((\Phi_l', K_l), (\Phi_r', K_r)) \in Stack[\![\tau]\!]$:

On the left side, we have the transition

$$\begin{aligned} \langle\!\langle \Phi_l' \parallel \Sigma_l \parallel M\ \mathtt{to}\ x\ \mathtt{in}\ R \parallel K_l \rangle\!\rangle &\longmapsto \\ \langle\!\langle \Phi_l' \parallel \Sigma_l \parallel M \parallel (\Sigma_l, \square\ \mathtt{to}\ x\ \mathtt{in}\ R) \cdot K_l \rangle\!\rangle & \end{aligned}$$

And similarly on the right.

$((\Phi_l', \Sigma_l, M), (\Phi_r', \Sigma_r, N)) \in Comp[\![F\ \tau]\!]$ by our first initial assumption with the environments $((\Phi_l', \Sigma_l), (\Phi_r', \Sigma_r)) \in Env[\![\Gamma]\!]$ (closure under accessible worlds).

$((\Phi_l', (\Sigma_l, \square\ \mathtt{to}\ x\ \mathtt{in}\ R \cdot K_l)), (\Phi_r', (\Sigma_r, \square\ \mathtt{to}\ x\ \mathtt{in}\ S) \cdot K_r)) \in Stack[\![F\ \tau]\!]$, by considering an arbitrary $\Phi_l'' \sqsupseteq \Phi_l'$, $\Phi_r'' \sqsupseteq \Phi_r'$, and $((\Phi_l'', \Sigma_l', \mathtt{ret}\ V), (\Phi_r'', \Sigma_r', \mathtt{ret}\ W)) \in Intro[\![F\ \tau]\!]$:

$((\Phi_l'', \mathrm{Build}_V(\Sigma_l', V)), (\Phi_r'', \mathrm{Build}_V(\Sigma_r', W))) \in Val[\![\tau]\!]$, by the definition of $Intro[\![F\ \tau]\!]$.

On the left side, there is the transition

$$\langle\!\langle \Phi_l'' \parallel \Sigma_l' \parallel \mathtt{ret}\ V \parallel (\Sigma_l, \square\ \mathtt{to}\ x\ \mathtt{in}\ R) \cdot K_l \rangle\!\rangle \quad \longmapsto$$
$$\langle\!\langle \Phi_l'' \parallel \Sigma_l, \mathrm{Build}_V(\Sigma_l', V)/x \parallel R \parallel K_l \rangle\!\rangle$$

And similarly on the right.

By the definition of related environments:

$$((\Phi_l'', (\Sigma_l, \mathrm{Build}_V(\Sigma_l', V)/x)),$$
$$(\Phi_r'', (\Sigma_r, \mathrm{Build}_V(\Sigma_r', W)/x))) \in \mathit{Env}[\![\Gamma, x{:}\sigma]\!].$$

Note $((\Phi_l'', \Sigma_l), (\Phi_r'', \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ by closure under accessible worlds.

$$((\Phi_l'', (\Sigma_l, \mathrm{Build}_V(\Sigma_l', V)/x), R)$$
$$, (\Phi_r'', (\Sigma_r, \mathrm{Build}_V(\Sigma_r', W)/x), S)) \in \mathit{Shared}[\![\tau]\!],$$

follows by the second initial assumption with the related environments immediately above.

By Proposition 7.8 on $((\Phi_l', K_l), (\Phi_r', K_r)) \in \mathit{Stack}[\![\tau]\!]$ with $\Phi_l'' \sqsupseteq \Phi_l'$, $\Phi_r'' \sqsupseteq \Phi_r'$, and the related shared expressions above, we have $\langle\!\langle \Phi_l'' \parallel \Sigma_l, \mathrm{Build}_V(\Sigma_l', V)/x \parallel R \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r'' \parallel \Sigma_r, \mathrm{Build}_V(\Sigma_r', W)/x \parallel S \parallel K_r \rangle\!\rangle$.

$\langle\!\langle \Phi_l'' \parallel \Sigma_l' \parallel \mathtt{ret}\ V \parallel (\Sigma_l, \square\ \mathtt{to}\ x\ \mathtt{in}\ R) \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r'' \parallel \Sigma_r' \parallel \mathtt{ret}\ W \parallel (\Sigma_r, \square\ \mathtt{to}\ x\ \mathtt{in}\ S) \cdot K_r \rangle\!\rangle$ by the closure properties of $(\simeq)$.

$\langle\!\langle \Phi_l' \parallel \Sigma_l \parallel M \parallel (\Sigma_l, \square\ \mathtt{to}\ x\ \mathtt{in}\ R) \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r' \parallel \Sigma_r \parallel N \parallel (\Sigma_r, \square\ \mathtt{to}\ x\ \mathtt{in}\ S) \cdot K_r \rangle\!\rangle$ by Proposition 7.8 with the related computation and stacks above with $\Phi_l' \sqsupseteq \Phi_l'$ and $\Phi_r' \sqsupseteq \Phi_r'$.

Finally, $\langle\langle \Phi'_l \parallel \Sigma_l \parallel M\ \texttt{to}\ x\ \texttt{in}\ R \parallel K_l \rangle\rangle \simeq \langle\langle \Phi'_r \parallel \Sigma_r \parallel N\ \texttt{to}\ x\ \texttt{in}\ S \parallel K_r \rangle\rangle$ by the closure properties of $(\simeq)$.

□

Proposition 7.22 (Compatibility *svar*). *If* $a{:}\tau \in \Gamma$, *then* $\Gamma \vDash a \approx a : \tau$.

Proof. Consider an arbitrary $a{:}\tau \in \Gamma$ and $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$. We may conclude $((\Phi_l, \Sigma_l, a), (\Phi_r, \Sigma_r, a)) \in \mathit{Shared}[\![\tau]\!]$ by the definition of $\mathit{Env}[\![\Gamma]\!]$. □

Proposition 7.23 (Compatibility *H*). *If* $\Gamma \vdash \varsigma_l \approx \varsigma_r : \Gamma'$, $\Gamma\Gamma' \vDash R \approx S : \tau$ *and* $\Gamma, a{:}\tau \vDash P \approx Q : \sigma$, *then* $\Gamma \vDash \{\varsigma_l, R\}\ \texttt{memo}\ a\ \texttt{in}\ P \approx \{\varsigma_r, S\}\ \texttt{memo}\ a\ \texttt{in}\ Q : \sigma$.

Proof. Considering some arbitrary $\Gamma \vdash \varsigma_l \approx \varsigma_r : \Gamma'$, $\Gamma\Gamma' \vDash R \approx S : \tau$, and $\Gamma, a{:}\tau \vDash P \approx Q : \sigma$, we show $\Gamma \vDash \{\varsigma_l, R\}\ \texttt{memo}\ a\ \texttt{in}\ P \approx \{\varsigma_r, S\}\ \texttt{memo}\ a\ \texttt{in}\ Q : \sigma$ by considering $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ and showing

$$\begin{aligned}&((\Phi_l, \Sigma_l, \{\varsigma_l, R\}\ \texttt{memo}\ a\ \texttt{in}\ M)\\ &, (\Phi_r, \Sigma_r, \{\varsigma_r, S\}\ \texttt{memo}\ a\ \texttt{in}\ N)) \in \mathit{Comp}[\![\sigma]\!].\end{aligned}$$

Note that we have picked σ to be a computation type σ, $P = M$, and $Q = N$; the proof will follow similarly for a shared type since the same transitions exist. By Proposition 7.8, it is sufficient to show that the pair is in the set $\mathit{Stack}[\![\sigma]\!]^{\top\!\top}$. Note that for shared types, we show the pair is in $\mathit{VStack}[\![\sigma]\!]^{\top\!\top}$ by definition. Thus, consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\sigma]\!]$:

On the left side, there is the transition:

$$\begin{aligned}\langle\langle \Phi'_l \parallel \Sigma_l \parallel \{\varsigma_l, R\}\ \texttt{memo}\ a\ \texttt{in}\ M \parallel K_l \rangle\rangle &\longmapsto\\ \langle\langle \Phi'_l, l \mapsto \{\Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), R\} \parallel \Sigma_l, l/a \parallel M \parallel K_l \rangle\rangle&\end{aligned}$$

And similarly on the right.

$$((\Phi_l, \Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), R)$$
$$, (\Phi_r, \Sigma_r\, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r), S)) \in \mathit{Shared}[\![\tau]\!]$$

by the first and second initial assumption together with our assumed related environments.

Using Lemma 7.1, we know

$$(((\Phi'_l, l_a \mapsto \{\Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), R\}), (\Sigma_l, l_a/a))$$
$$, ((\Phi'_r, l_a \mapsto \{\Sigma_r\, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r), S\}), (\Sigma_r, l_a/a))) \in \mathit{Env}[\![\Gamma, a{:}\tau]\!].$$

By the third initial assumption with the environment above,

$$(((\Phi'_l, l_a \mapsto \{\Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), R\}), (\Sigma_l, l_a/a), M)$$
$$, ((\Phi'_r, l_a \mapsto \{\Sigma_r\, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r), S\}), (\Sigma_r, l_a/a), N)) \in \mathit{Comp}[\![\sigma]\!].$$

$\langle\!\langle \Phi'_l, l \mapsto \{\Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), R\} \parallel \Sigma_l, l/a \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r, l \mapsto \{\Sigma_r\, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r), S\} \parallel \Sigma_r, l/a \parallel N \parallel K_r \rangle\!\rangle$, by Proposition 7.8 on $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$ with the future heaps $(\Phi'_l, l \mapsto \{\Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), R\}) \sqsupseteq \Phi'_l$ and $(\Phi'_r, l \mapsto \{\Sigma_r\, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r), S\}) \sqsupseteq \Phi'_r$, and the related computations immediately above. (For shared types, we apply the property of the related shared expressions to the value stacks.)

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\varsigma_l, R\}\ \mathtt{memo}\ a\ \mathtt{in}\ M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel \{\varsigma_r, S\}\ \mathtt{memo}\ a\ \mathtt{in}\ N \parallel K_r \rangle\!\rangle$ by the closure properties of $(\simeq)$.

□

Proposition 7.24 (Compatibility $\tilde{U}_I$). *If* $\Gamma \vDash V \approx W : \tau$, $\Gamma \vDash \mathtt{box}\ V \approx \mathtt{box}\ W : \tilde{U}\ \tau$.

Proof. Given $\Gamma \vDash V \approx W : \tau$, we show $\Gamma \vDash \mathsf{box}\ V \approx \mathsf{box}\ W : \tilde{U}\ \tau$ by considering some $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$ and then proving that $((\Phi_l, \mathrm{Build}_V(\Sigma_l, \mathsf{box}\ V)), (\Phi_r, \mathrm{Build}_V(\Sigma_r, \mathsf{box}\ W))) \in Val[\![\tilde{U}\ \tau]\!]$. By the definition of building, we may conclude that $\mathrm{Build}_V(\Sigma_l, \mathsf{box}\ V) = \mathsf{box}\ \mathrm{Build}_V(\Sigma_l, V)$ and similarly for the right side. Our initial assumption with $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$ allows us to conclude that $((\Phi_l, \Sigma_l, V), (\Phi_r, \Sigma_r, W)) \in Shared[\![\tau]\!]$. We may conclude by the definition of $Val[\![\tilde{U}\ \tau]\!]$ and Lemma 7.2. □

Proposition 7.25 (Compatibility $\tilde{U}_E$). *If* $\Gamma \vDash V \approx W : \tilde{U}\ \sigma$ *and* $\Gamma, a{:}\sigma \vDash P \approx Q : \tau$, *then* $\Gamma \vDash \mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{box}\ a \rightarrow P\} \approx \mathsf{case}\ W\ \mathsf{of}\ \{\mathsf{box}\ a \rightarrow Q\} : \tau$.

Proof. Given that $\Gamma \vDash V \approx W : \tilde{U}\ \sigma$ and $\Gamma, a{:}\sigma \vDash P \approx Q : \tau$, we prove $\Gamma \vDash \mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{box}\ a \rightarrow P\} \approx \mathsf{case}\ W\ \mathsf{of}\ \{\mathsf{box}\ a \rightarrow Q\} : \tau$, by first considering $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$ and then showing

$$\begin{aligned}&((\Phi_l, \Sigma_l, \mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{box}\ a \rightarrow M\})\\ &, (\Phi_r, \Sigma_r, \mathsf{case}\ W\ \mathsf{of}\ \{\mathsf{box}\ a \rightarrow N\})) \in Comp[\![\tau]\!].\end{aligned}$$

We have chosen τ to be a computation type, $P = M$, and $Q = N$, but the proof follows the same for shared types since we have the same transitions. By Proposition 7.8, this is the same as showing that the triple is in $Stack[\![\rho]\!]^{\top\!\top}$. Thus, we suppose an arbitrary $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in Stack[\![\rho]\!]$:

$((\Phi'_l, \mathrm{Build}_V(\Sigma_l, V)), (\Phi'_r, \mathrm{Build}_V(\Sigma_r, W))) \in Val[\![\tilde{U}\ \sigma]\!]$ by the first initial assumption with the related environments $((\Phi'_l, \Sigma_l), (\Phi'_r, \Sigma_r)) \in Env[\![\Gamma]\!]$ (making use of the Kripke property). By this definition, we also know $\mathrm{Build}_V(\Sigma_l, V) = \mathsf{box}\ \mathbb{V}$ and $\mathrm{Build}_V(\Sigma_r, W) = \mathsf{box}\ \mathbb{W}$ with the property that $(\mathrm{Run}(\Phi'_l, \mathbb{V}), \mathrm{Run}(\Phi'_r, \mathbb{W})) \in Shared[\![\tau]\!]$.

On the left side, we have the transition

$$\begin{aligned}\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{box}\ a \longrightarrow M\} \parallel K_l \rangle\!\rangle \quad &\longmapsto \\ \langle\!\langle \Phi'_l \parallel \Sigma_l, \mathbb{V}/a \parallel M \parallel K_l \rangle\!\rangle\end{aligned}$$

And similarly on the right side.

$((\Phi'_l, (\Sigma_l, \mathbb{V}/a)), (\Phi'_r, (\Sigma_r, \mathbb{W}/a))) \in \mathit{Env}[\![\Gamma, a{:}\sigma]\!]$, by definition and that running them gives related shared expressions. And thus, we may use our second initial assumption with these related environments to conclude that

$$\begin{aligned}&((\Phi'_l, (\Sigma_l, \mathbb{V}/a), M) \\ &, (\Phi'_r, (\Sigma_r, \mathbb{W}/a), N)) \in \mathit{Comp}[\![\tau]\!].\end{aligned}$$

By Proposition 7.8, the property of our related stacks in combination with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and these related computations, gives us $\langle\!\langle \Phi'_l \parallel \Sigma_l, \mathbb{V}/a \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r, \mathbb{W}/a \parallel N \parallel K_r \rangle\!\rangle$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{box}\ a \longrightarrow M\} \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel \mathsf{case}\ W\ \mathsf{of}\ \{\mathsf{box}\ a \longrightarrow N\} \parallel K_r \rangle\!\rangle$ by the closure rules for $(\simeq)$.

$\square$

Proposition 7.26 (Compatibility $\check{U}_I$). *If* $\Gamma \vdash \varsigma_l \approx \varsigma_r : \Gamma'$ *and* $\Gamma\Gamma' \vDash M \approx N : \tau$*, then* $\Gamma \vDash \{\varsigma_l, \mathsf{enter} \longrightarrow M\} \approx \{\varsigma_r, \mathsf{enter} \longrightarrow N\} : \check{U}\ \tau$.

Proof. Given $\Gamma \vdash \varsigma_l \approx \varsigma_r : \Gamma'$ and $\Gamma\Gamma' \vDash M \approx N : \tau$, showing $\Gamma \vDash \{\varsigma_l, \mathsf{enter} \longrightarrow M\} \approx \{\varsigma_r, \mathsf{enter} \longrightarrow N\} : \check{U}\ \tau$ requires that we consider related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ and show that

$$\begin{aligned}&((\Phi_l, \Sigma_l, \{\varsigma_l, \texttt{enter} \to M\})\\&, (\Phi_r, \Sigma_r, \{\varsigma_r, \texttt{enter} \to N\})) \in \mathit{Shared}[\![\check{U}\ \tau]\!].\end{aligned}$$

As $\mathit{Val}[\![\check{U}\ \tau]\!] \subseteq \mathit{Shared}[\![\check{U}\ \tau]\!]$, it will suffice to show that the triple is in $\mathit{Val}[\![\check{U}\ \tau]\!]$. Thus, consider an arbitrary $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, \square.\texttt{enter} \cdot K_l), (\Phi'_r, \square.\texttt{enter} \cdot K_r)) \in \mathit{Elim}[\![\check{U}\ \tau]\!]$:

On the left side, there is the transition:

$$\begin{aligned}&\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\varsigma_l, \texttt{enter} \to M\} \parallel \square.\texttt{enter} \cdot K_l \rangle\!\rangle \quad \longmapsto\\&\qquad\langle\!\langle \Phi'_l \parallel \Sigma_l\,\mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l) \parallel M \parallel K_l \rangle\!\rangle\end{aligned}$$

And similarly on the right.

$((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$, by definition.

$((\Phi'_l, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l)), (\Phi'_r, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r))) \in \mathit{Env}[\![\Gamma']\!]$ from our first initial assumption with the related environments $((\Phi'_l, \Sigma_l), (\Phi'_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ (using the Kripke property).

The extended environments are related

$$\begin{aligned}&((\Phi'_l, \Sigma_l\,\mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l))\\&, (\Phi'_r, \Sigma_r\,\mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r))) \in \mathit{Env}[\![\Gamma\Gamma']\!]\end{aligned}$$

by definition.

$$((\Phi'_l, \Sigma_l\ \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l), M)$$
$$, (\Phi'_r, \Sigma_r\ \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r), N)) \in \mathit{Comp}[\![\tau]\!]$$

from these related environments with our second initial assumption.

It follows that $\langle\!\langle \Phi'_l \parallel \Sigma_l\ \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l) \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r\ \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r) \parallel N \parallel K_r \rangle\!\rangle$, by Proposition 7.8 with the related computations above in the worlds $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\varsigma_l, \mathtt{enter} \to M\} \parallel \square.\mathtt{enter} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel \{\varsigma_r, \mathtt{enter} \to N\} \parallel \square.\mathtt{enter} \cdot K_r \rangle\!\rangle$, by the closure properties of $(\simeq)$.

□

Proposition 7.27 (Compatibility $\check{U}_E$). *If* $\Gamma \vDash R \approx S : \check{U}\ \tau$, *then* $\Gamma \vDash R.\mathtt{enter} \approx S.\mathtt{enter} : \tau$.

Proof. Given $\Gamma \vDash R \approx S : \check{U}\ \tau$, showing that $\Gamma \vDash R.\mathtt{enter} \approx S.\mathtt{enter} : \tau$ requires considering further the related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ and then proving $((\Phi_l, \Sigma_l, R.\mathtt{enter}), (\Phi_r, \Sigma_r, S.\mathtt{enter})) \in \mathit{Comp}[\![\tau]\!]$. By Proposition 7.8, it suffices to show that the pair is in the set $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$. Thus, further consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left side, there is the transition:

$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R.\mathtt{enter} \parallel K_l \rangle\!\rangle \quad \longmapsto$$
$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R \parallel \square.\mathtt{enter} \cdot K_l \rangle\!\rangle$$

And similarly on the right.

$((\Phi'_l, \square.\mathtt{enter} \cdot K_l), (\Phi'_r, \square.\mathtt{enter} \cdot K_r)) \in \mathit{Stack}[\![\check{U}\ \tau]\!]$, by definition.

$((\Phi'_l, \Sigma_l, R), (\Phi'_r, \Sigma_r, S)) \in \mathit{Shared}[\![\check{U}\ \tau]\!]$, by the initial assumption with the related environments $((\Phi'_l, \Sigma_l), (\Phi'_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ (closure under accessible worlds).

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R \parallel \square.\mathtt{enter} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel R \parallel \square.\mathtt{enter} \cdot K_r \rangle\!\rangle$ by the property of $((\Phi'_l, \Sigma_l, R), (\Phi'_r, \Sigma_r, S)) \in \mathit{Shared}[\![\check{U}\ \tau]\!]$ with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and $((\Phi'_l, \square.\mathtt{enter} \cdot K_l), (\Phi'_r, \square.\mathtt{enter} \cdot K_r)) \in \mathit{Stack}[\![\check{U}\ \tau]\!]$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R.\mathtt{enter} \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel S.\mathtt{enter} \parallel K_r \rangle\!\rangle$ by the closure properties of $(\simeq)$.

□

Proposition 7.28 (Compatibility $\hat{F}_I$). *If $\Gamma \vDash V \approx W : \tau$, then $\Gamma \vDash \mathtt{val}\ V \approx \mathtt{val}\ W : \hat{F}\ \tau$.*

Proof. Given $\Gamma \vDash V \approx W : \tau$, showing both $\Gamma \vDash \mathtt{val}\ V \approx \mathtt{val}\ W : F\ \tau$ and $\Gamma \vDash \mathtt{val}\ V \approx \mathtt{val}\ W : F\ \tau$ require considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$. By our initial assumption with these related environments, we may conclude that $((\Phi_l, \mathrm{Build}_V(\Sigma_l, V)), (\Phi_r, \mathrm{Build}_V(\Sigma_r, W))) \in \mathit{Val}[\![\tau]\!]$. This is enough to conclude $((\Phi_l, \Sigma_l, \mathtt{val}\ V), (\Phi_r, \Sigma_r, \mathtt{val}\ W)) \in \mathit{Intro}[\![\hat{F}\ \tau]\!]$ and by the inclusion properties of, with the definitions of the relations for this type, $((\Phi_l, \Sigma_l, \mathtt{val}\ V), (\Phi_r, \Sigma_r, \mathtt{val}\ W)) \in \mathit{Shared}[\![\hat{F}\ \tau]\!]$. □

Proposition 7.29 (Compatibility $\hat{F}_E$). *If $\Gamma \vDash R \approx S : \hat{F}\ \sigma$ and $\Gamma, x{:}\sigma \vDash P \approx Q : \tau$, then $\Gamma \vDash R\ \mathtt{to}\ x\ \mathtt{in}\ P \approx S\ \mathtt{to}\ x\ \mathtt{in}\ Q : \tau$.*

Proof. Given $\Gamma \vDash R \approx S : \hat{F}\ \sigma$ and $\Gamma, x{:}\sigma \vDash P \approx Q : \tau$, showing $\Gamma \vDash R\ \mathtt{to}\ x\ \mathtt{in}\ P \approx S\ \mathtt{to}\ x\ \mathtt{in}\ Q : \tau$ requires considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ and then proving that $((\Phi_l, \Sigma_l, R\ \mathtt{to}\ x\ \mathtt{in}\ M), (\Phi_r, \Sigma_r, S\ \mathtt{to}\ x\ \mathtt{in}\ N)) \in \mathit{Comp}[\![\tau]\!]$. We have chosen τ to be a computation, $P = M$, and $Q = N$, but the proof follows the same for shared types since we have the same transitions. By Proposition 7.8, it suffices to show that the pair is in $\mathit{Stack}[\![\tau]\!]^{\top\top}$. Thus, consider an arbitrary $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left side, we have the transition

$$\begin{aligned} &\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R \texttt{ to } x \texttt{ in } M \parallel K_l \rangle\!\rangle \quad \longmapsto \\ &\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R \parallel (\Sigma_l, \square \texttt{ to } x \texttt{ in } M) \cdot K_l \rangle\!\rangle \end{aligned}$$

And similarly on the right.

$((\Phi_l, \Sigma_l, R), (\Phi_r, \Sigma_r, S)) \in \mathit{Shared}[\![\hat{F}\ \tau]\!]$ by our first initial assumption with the related environments $((\Phi'_l, \Sigma_l), (\Phi'_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ (closure under accessible worlds).

$((\Phi'_l, (\Sigma_l, \square \texttt{ to } x \texttt{ in } M\ \cdot\ K_l)), (\Phi'_r, (\Sigma_r, \square \texttt{ to } x \texttt{ in } N)\ \cdot\ K_r)) \in \mathit{VStack}[\![\hat{F}\ \tau]\!]$ and thus $\mathit{Stack}[\![\hat{F}\ \tau]\!]$ with the inclusion properties of the relation, by considering an arbitrary $\Phi''_l \sqsupseteq \Phi'_l$, $\Phi''_r \sqsupseteq \Phi'_r$, and $((\Phi''_l, \Sigma'_l, \texttt{val } V), (\Phi''_r, \Sigma'_r, \texttt{val } W)) \in \mathit{Intro}[\![\hat{F}\ \tau]\!]$:

$((\Phi''_l, \mathrm{Build}_V(\Sigma'_l, V)), (\Phi''_r, \mathrm{Build}_V(\Sigma'_r, W))) \in \mathit{Val}[\![\tau]\!]$ by the definition of $\mathit{Intro}[\![\hat{F}\ \tau]\!]$.

On the left side, there is the transition

$$\begin{aligned} &\langle\!\langle \Phi''_l \parallel \Sigma'_l \parallel \texttt{val } V \parallel (\Sigma_l, \square \texttt{ to } x \texttt{ in } M) \cdot K_l \rangle\!\rangle \quad \longmapsto \\ &\langle\!\langle \Phi''_l \parallel \Sigma_l, \mathrm{Build}_V(\Sigma'_l, V)/x \parallel M \parallel K_l \rangle\!\rangle \end{aligned}$$

And similarly on the right.

$$\begin{aligned} &((\Phi''_l, (\Sigma_l, \mathrm{Build}_V(\Sigma'_l, V)/x)) \\ &, (\Phi''_r, (\Sigma_r, \mathrm{Build}_V(\Sigma'_r, W)/x))) \in \mathit{Env}[\![\Gamma, x{:}\sigma]\!], \end{aligned}$$

by definition; note $((\Phi''_l, \Sigma_l), (\Phi''_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ by closure under accessible worlds.

$$((\Phi''_l, (\Sigma_l, \mathrm{Build}_V(\Sigma'_l, V)/x), M)$$
$$, (\Phi''_r, (\Sigma_r, \mathrm{Build}_V(\Sigma'_r, W)/x), N)) \in \mathit{Comp}[\![\tau]\!],$$

by the second initial assumption with the related environments immediately above.

By Proposition 7.8 on $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$ with future heaps $\Phi''_l \sqsupseteq \Phi'_l$ and $\Phi''_r \sqsupseteq \Phi'_r$, and the related shared expressions immediately above allow us to conclude that $\langle\!\langle \Phi''_l \parallel \Sigma_l, \mathrm{Build}_V(\Sigma'_l, V)/x \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi''_r \parallel \Sigma_r, \mathrm{Build}_V(\Sigma'_r, W)/x \parallel N \parallel K_r \rangle\!\rangle$.

By the closure properties of $(\simeq)$, $\langle\!\langle \Phi''_l \parallel \Sigma'_l \parallel \mathtt{val}\ V \parallel (\Sigma_l, \square\ \mathtt{to}\ x\ \mathtt{in}\ M) \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi''_r \parallel \Sigma'_r \parallel \mathtt{val}\ W \parallel (\Sigma_r, \square\ \mathtt{to}\ x\ \mathtt{in}\ N) \cdot K_r \rangle\!\rangle$.

By Proposition 7.8 with the related shared computation and stacks above in the world $\Phi'_l \sqsupseteq \Phi'_l, \Phi'_r \sqsupseteq \Phi'_r$, we know $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R \parallel (\Sigma_l, \square\ \mathtt{to}\ x\ \mathtt{in}\ M) \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel S \parallel (\Sigma_r, \square\ \mathtt{to}\ x\ \mathtt{in}\ N) \cdot K_r \rangle\!\rangle$.

Finally, $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R\ \mathtt{to}\ x\ \mathtt{in}\ M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel S\ \mathtt{to}\ x\ \mathtt{in}\ N \parallel K_r \rangle\!\rangle$ by the closure properties of $(\simeq)$.

$\square$

Proposition 7.30 (Compatibility $\tilde{F}_I$). *If* $\Gamma \vDash R \approx S : \tau$, *then* $\Gamma \vDash \{\mathtt{eval} \to R\} \approx \{\mathtt{eval} \to S\} : \tilde{F}\ \tau$.

Proof. Given $\Gamma \vDash R \approx S : \tau$, we prove that $\Gamma \vDash \{\mathtt{eval} \to R\} \approx \{\mathtt{eval} \to S\} : \tilde{F}\ \tau$ by considering $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ and proving that

$$((\Phi_l, \Sigma_l, \{\mathtt{eval} \to R\}), (\Phi_r, \Sigma_r, \{\mathtt{eval} \to S\})) \in \mathit{Comp}[\![\tilde{F}\ \tau]\!].$$

By definition, this is equivalent to showing that the pair is in $\mathit{Elim}[\![\tilde{F}\ \tau]\!]^{\top\!\top}$. Thus, considering an arbitrary $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi$, and $((\Phi'_l, \square.\mathtt{eval} \cdot K_l), (\Phi'_r, \square.\mathtt{eval} \cdot K_r)) \in \mathit{Elim}[\![\tilde{F}\ \tau]\!]$:

By definition of $\mathit{Elim}[\![\tilde{F}\ \tau]\!]$, we know $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$.

By the first of our initial assumptions with related environments $((\Phi'_l, \Sigma_l), (\Phi'_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ (closed with accessible worlds), we know $((\Phi'_l, \Sigma_l, R), (\Phi'_r, \Sigma_r, S)) \in \mathit{Shared}[\![\tau]\!]$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel S \parallel K_r \rangle\!\rangle$, by the definition of $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$ in the worlds $\Phi'_l \sqsupseteq \Phi'_l$ and $\Phi'_r \sqsupseteq \Phi'_r$, with the related expressions above.

On the left side, we have the transition

$$\begin{aligned} \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\mathtt{eval} \to R\} \parallel \square.\mathtt{eval} \cdot K_l \rangle\!\rangle \quad &\longmapsto \\ \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R \parallel K_l \rangle\!\rangle & \end{aligned}$$

Similarly on the right.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\mathtt{eval} \to R\} \parallel \square.\mathtt{eval} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel \{\mathtt{eval} \to S\} \parallel \square.\mathtt{eval} \cdot K_r \rangle\!\rangle$, by the closure properties of $(\simeq)$.

$\square$

Proposition 7.31 (Compatibility $\tilde{F}_E$). *If* $\Gamma \vDash M \approx N : \tilde{F}\ \tau$, *then* $\Gamma \vDash M.\mathtt{eval} \approx N.\mathtt{eval} : \tau$.

Proof. Given $\Gamma \vDash M \approx N : \tilde{F}\ \tau$, we show $\Gamma \vDash M.\mathtt{eval} \approx N.\mathtt{eval} : \tau$ by first considering $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ and then proving $((\Phi_l, \Sigma_l, M.\mathtt{eval}), (\Phi_r, \Sigma_r, N.\mathtt{eval})) \in \mathit{Shared}[\![\tau]\!]$. By definition, this is the same as showing that the pair is in $\mathit{VStack}[\![\tau]\!]^{\top\!\top}$. Thus, consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, \mathbb{K}_l), (\Phi'_r, \mathbb{K}_r)) \in \mathit{VStack}[\![\tau]\!]$:

On the left side, we have the transition

$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M.\mathtt{eval} \parallel \mathbb{K}_l \rangle\!\rangle \longmapsto$$
$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \parallel \square.\mathtt{eval} \cdot \mathbb{K}_l \rangle\!\rangle$$

And similarly on the right.

$((\Phi'_l, \Sigma_l, M), (\Phi'_r, \Sigma_r, N)) \in \mathit{Comp}[\![\tilde{F}\ \tau]\!]$, by our initial assumption with the related environments above (note closure over accessible worlds).

$((\Phi'_l, \mathbb{K}_l), (\Phi'_r, \mathbb{K}_r)) \in \mathit{Stack}[\![\tau]\!]$, since $\mathit{VStack}[\![\tau]\!] \subseteq \mathit{Stack}[\![\tau]\!]$.

$((\Phi'_l, \square.\mathtt{eval} \cdot \mathbb{K}_l), (\Phi'_r, \square.\mathtt{eval} \cdot \mathbb{K}_r)) \in \mathit{Stack}[\![\tilde{F}\ \tau]\!]$ by definition. This and Proposition 7.8 with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and the related computations above yields that $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \parallel \square.\mathtt{eval} \cdot \mathbb{K}_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel N \parallel \square.\mathtt{eval} \cdot \mathbb{K}_r \rangle\!\rangle$.

By the closure properties of $(\simeq)$, we have $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M.\mathtt{eval} \parallel \mathbb{K}_l \rangle\!\rangle) \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel N.\mathtt{eval} \parallel \mathbb{K}_r \rangle\!\rangle$.

□

Proposition 7.32 (Compatibility Γ_{I_B}). *$\Gamma \vDash \varepsilon \approx \varepsilon : \varepsilon$.*

Proof. Since the captured environment is empty, this case holds trivially for any related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$. □

Proposition 7.33 (Compatibility $\Gamma_{I_{I1}}$). *If $\Gamma \vDash V \approx W : \tau$ and $\Gamma \vDash \varsigma_l \approx \varsigma_r : \Gamma'$, then $\Gamma \vDash (\varsigma_l, V/x) \approx (\varsigma_r, W/x) : (\Gamma', x{:}\tau)$.*

Proof. Given $\Gamma \vDash V \approx W : \tau$ and $\Gamma \vDash \varsigma_l \approx \varsigma_r : \Gamma'$, we show $\Gamma \vDash (\varsigma_l, V/x) \approx (\varsigma_r, W/x) : (\Gamma', x{:}\tau)$ by considering arbitrary related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ then proving that

$$\begin{aligned}&((\Phi_l, \mathrm{Build}_\varsigma(\Sigma_l, (\varsigma_l, V/x)))\\ &, (\Phi_r, \mathrm{Build}_\varsigma(\Sigma_r, (\varsigma_r, W/x)))) \in \mathit{Env}[\![\Gamma', x{:}\tau]\!].\end{aligned}$$

By our first initial assumption with the related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, we know $((\Phi_l, \mathrm{Build}_V(\Sigma_l, V)), (\Phi_r, \mathrm{Build}_V(\Sigma_r, W))) \in \mathit{Val}[\![\tau]\!]$. And by the second, $((\Phi_l, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma_l)), (\Phi_r, \mathrm{Build}_\varsigma(\Sigma_r, \varsigma_r))) \in \mathit{Env}[\![\Gamma']\!]$. Considering some arbitrary $y{:}\sigma \in \Gamma', x{:}\tau$, there are two cases to prove depending on whether $x = y$. If so, then we have the related values at hand. Otherwise, we find the related shared expressions or values by looking further back in the constructed environment. □

Proposition 7.34 (Compatibility $\Gamma_{I_{l2}}$). *If* $\Gamma \vDash V \approx W : \tau$ *and* $\Gamma \vDash \varsigma_l \approx \varsigma_r : \Gamma'$, *then* $\Gamma \vDash (\varsigma_l, V/a) \approx (\varsigma_r, W/a) : (\Gamma', a{:}\tau)$.

Proof. Follows in a similar manner to compatibility for the $\Gamma_{I_{l1}}$ rule. □

Proposition 7.35 (Semantic Congruence). *Semantic equivalence forms a congruence relation; it is:*

1. Reflexive: *If* $\Gamma \vdash A : \tau$ *then* $\Gamma \vDash A \approx A : \tau$,
2. Symmetric: *If* $\Gamma \vDash A \approx B : \tau$ *then* $\Gamma \vDash B \approx A : \tau$,
3. Transitive: *If* $\Gamma \vDash A \approx B : \tau$ *and* $\Gamma \vDash B \approx C : \tau$ *then* $\Gamma \vDash A \approx C : \tau$, *and*
4. Compatible: *If* $\Gamma \vDash A \approx B : \tau$ *and* $\Gamma' \vdash C[D] : \sigma$ *for all* $\Gamma \vdash D : \tau$, *then* $\Gamma' \vDash C[A] \approx C[B] : \sigma$.

Lemma 7.6 (Fundamental Lemma). *If* $\Gamma \vdash A = B : \tau$, *then* $\Gamma \vDash A \approx B : \tau$.

Proof. By induction on the equality derivation:

Case $\beta_{\to}$:

$$\Gamma \vdash (\lambda x.\, M)\ V = M[V/x] : \tau$$

Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, we show $((\Phi_l, \Sigma_l, (\lambda x.\, M)\ V), (\Phi_r, \Sigma_r, M[V/x])) \in \mathit{Comp}[\![\tau]\!]$. By Proposition 7.8, it is enough to show that the pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$. Thus, further consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left, there are the transitions:

$$\begin{aligned} \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel (\lambda x.\, M)\ V \parallel K_l \rangle\!\rangle &\longmapsto \\ \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \lambda x.\, M \parallel \square\ \mathrm{Build}_V(\Sigma_l, V) \cdot K_l \rangle\!\rangle &\longmapsto \\ \langle\!\langle \Phi'_l \parallel \Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x \parallel M \parallel K_l \rangle\!\rangle & \end{aligned}$$

Note that $\Gamma \vdash M : \sigma \to \tau$ and $\Gamma \vdash V : \sigma$ follows from the equality.

$$\begin{aligned} &((\Phi'_l, (\Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x), M) \\ &, (\Phi'_r, \Sigma_r, M[V/x])) \in \mathit{Comp}[\![\tau]\!], \end{aligned}$$

by Corollary 7.1 with the related environments above (considering closure under accessible heaps). And this pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$ by Proposition 7.8.

$\langle\!\langle \Phi'_l \parallel \Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M[V/x] \parallel K_r \rangle\!\rangle$ by the property of the related computations with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel (\lambda x.\, M)\ V \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M[V/x] \parallel K_r \rangle\!\rangle$ by the closure properties of $(\simeq)$.

Case $\beta_{\&1}$:

$$\Gamma \vdash \{\mathtt{fst} \to M; \mathtt{snd} \to N\}.\mathtt{fst} = M : \tau$$

Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, we must show

$$\begin{aligned}&((\Phi_l, \Sigma_l, \{\mathtt{fst} \to M; \mathtt{snd} \to N\}.\mathtt{fst})\\ &, (\Phi_r, \Sigma_r, M)) \in \mathit{Comp}[\![\tau]\!].\end{aligned}$$

By Proposition 7.8, it is enough to show that the pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\!\top}$. Thus, further consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left, there is the transition sequence:

$$\begin{aligned}\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\mathtt{fst} \to M; \mathtt{snd} \to N\}.\mathtt{fst} \parallel K_l \rangle\!\rangle &\longmapsto\\ \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\mathtt{fst} \to M; \mathtt{snd} \to N\} \parallel \square.\mathtt{fst} \cdot K_l \rangle\!\rangle &\longmapsto\\ \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \parallel K_l \rangle\!\rangle &\end{aligned}$$

Note that $\Gamma \vdash M : \tau$ follows from the equality. And by the reflexivity of $(\approx)$ with the related environments, $((\Phi_l, \Sigma_l, M), (\Phi_r, \Sigma_r, M)) \in \mathit{Comp}[\![\tau]\!]$. Using Proposition 7.8, this pair is in the set $\mathit{Stack}[\![\tau]\!]^{\top\!\!\top}$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M \parallel K_r \rangle\!\rangle$ follows from the property of the related computations above with $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$.

The closure properties of $(\simeq)$ yield that $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\mathtt{fst} \to M; \mathtt{snd} \to N\}.\mathtt{fst} \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M \parallel K_r \rangle\!\rangle$.

Case $\beta_{\&2}$:

$$\Gamma \vdash \{\mathtt{fst} \to M; \mathtt{snd} \to N\}.\mathtt{snd} = N : \tau$$

Follows in the same manner as the case immediately above.

Case $\beta_\otimes$:

$$\Gamma \vdash \mathsf{case}\ \langle V, W\rangle\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow P\} = P[V/x, W/y] : \tau$$

Note that we prove this for the shared computation case where $\tau = \tau$ and $P = R$, but the proof follows in a similar manner for the computation case since the same transitions exist. Considering arbitrary related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma]\!]$, we must show that

$$\begin{aligned}&((\Phi_l, \Sigma_l, \mathsf{case}\ \langle V, W\rangle\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow R\})\\ &, (\Phi_r, \Sigma_r, R[V/x, W/y])) \in \mathit{Shared}[\![\tau]\!].\end{aligned}$$

By Proposition 7.8, it is enough to show that the pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$. Thus, further consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left, there is the transition:

$$\begin{aligned}\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \mathsf{case}\ \langle V, W\rangle\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow R\} \parallel K_l\rangle\!\rangle \quad \longmapsto\\ \langle\!\langle \Phi'_l \parallel \Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x, \mathrm{Build}_V(\Sigma_l, W)/y \parallel R \parallel K_l\rangle\!\rangle\end{aligned}$$

Note that $\Gamma \vdash \langle V, W\rangle : \sigma \otimes \rho$ and $\Gamma, x{:}\sigma, y{:}\rho \vdash R : \tau$ follows from the equality. By inversion on the former, we know $\Gamma \vdash V : \sigma$ and $\Gamma \vdash W : \rho$. Applying Corollary 7.1 with our related environments, we know

$$\begin{aligned}&((\Phi_l, (\Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x, \mathrm{Build}_V(\Sigma_l, W)/y), R)\\ &, (\Phi_r, \Sigma_r, R[V/x, W/y])) \in \mathit{Shared}[\![\tau]\!].\end{aligned}$$

By Proposition 7.8, the pair is in the set $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x, \mathrm{Build}_V(\Sigma_l, W)/y \parallel R \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel R[V/x, W/y] \parallel K_r \rangle\!\rangle$, by the above related expressions with $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \mathtt{case}\ \langle V, W\rangle\ \mathtt{of}\ \{\langle x, y\rangle \to R\} \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel R[V/x, W/y] \parallel K_r \rangle\!\rangle$, by the backwards closure of $(\simeq)$.

Case β_U:

$$\Gamma \vdash \{\varsigma, \mathtt{force} \to M\}.\mathtt{force} = M[\varsigma] : \tau$$

Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, we must be able to show that

$$((\Phi_l, \Sigma_l, \{\varsigma, \mathtt{force} \to M\}.\mathtt{force}), (\Phi_r, \Sigma_r, M)) \in \mathit{Comp}[\![\tau]\!].$$

By Proposition 7.8, it is enough to show that the pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$. Thus, further consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left, there is the transition:

$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\varsigma, \mathtt{force} \to M\}.\mathtt{force} \parallel K_l \rangle\!\rangle \longmapsto$$
$$\langle\!\langle \Phi'_l \parallel \Sigma_l\,\mathrm{Build}_\varsigma(\Sigma_l, \varsigma) \parallel M \parallel K_l \rangle\!\rangle$$

Note that $\Gamma \vdash \{\varsigma, \mathtt{force} \to M\} : U\ \tau$ follows from the equality. By inversion, we know $\Gamma \vdash \varsigma : \Gamma'$ and $\Gamma\Gamma' \vdash M : \tau$. Applying Corollary 7.1 with the assumed related environments, we know

$$((\Phi_l, \Sigma_l\,\mathrm{Build}_\varsigma(\Sigma_l, \varsigma), M), (\Phi_r, \Sigma_r, M[\varsigma])) \in \mathit{Comp}[\![\tau]\!].$$

By Proposition 7.8, this pair is in the set $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l\,\mathrm{Build}_\varsigma(\Sigma_l,\varsigma) \parallel M \parallel K_l\rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M[\varsigma] \parallel K_r\rangle\!\rangle$, by the related computations above with $\Phi'_l \sqsupseteq \Phi$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\varsigma, \mathtt{force} \longrightarrow M\}.\mathtt{force} \parallel K_l\rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M[\varsigma] \parallel K_r\rangle\!\rangle$, by the backwards closure of $(\simeq)$.

Case β_F:

$$\Gamma \vdash (\mathtt{ret}\ V)\ \mathtt{to}\ x\ \mathtt{in}\ P = P[V/x] : \tau$$

Note that we prove this for the computation case where $P = M$ and $\tau = \tau$, but the proof follows in the same manner for shared computations since the same transitions exist. Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, we must show that

$$\begin{aligned}&((\Phi_l, \Sigma_l, (\mathtt{ret}\ V)\ \mathtt{to}\ x\ \mathtt{in}\ M)\\ &, (\Phi_r, \Sigma_r, M[V/x])) \in \mathit{Comp}[\![\tau]\!].\end{aligned}$$

By Proposition 7.8, we may just show that the pair is in the set $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$. Thus, further consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_l)) \in \mathit{Stack}[\![\tau]\!]$:

On the left, there is the transition sequence:

$$\begin{aligned}\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel (\mathtt{ret}\ V)\ \mathtt{to}\ x\ \mathtt{in}\ M \parallel K_l\rangle\!\rangle &\longmapsto\\ \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \mathtt{ret}\ V \parallel (\Sigma_l, \square\ \mathtt{to}\ x\ \mathtt{in}\ M)\cdot K_l\rangle\!\rangle &\longmapsto\\ \langle\!\langle \Phi'_l \parallel \Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x \parallel M \parallel K_l\rangle\!\rangle &\end{aligned}$$

Note that $\Gamma \vdash (\mathtt{ret}\ V)\ \mathtt{to}\ x\ \mathtt{in}\ M : \tau$ follows from the equality. By inversion, we can conclude that $\Gamma \vdash V : \sigma$ and $\Gamma, x{:}\sigma \vdash M : \tau$. By Corollary 7.1 with our related environments, we know

$$((\Phi_l, (\Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x), M)$$
$$, (\Phi_r, \Sigma_r, M[V/x])) \in \mathit{Comp}[\![\tau]\!].$$

And further, Proposition 7.8 says that the pair is in the set $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M[V/x] \parallel K_r \rangle\!\rangle$, by the property of the related computations above with $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel (\texttt{ret}\ V)\ \texttt{to}\ x\ \texttt{in}\ M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M[V/x] \parallel K_r \rangle\!\rangle$, by the closure properties of $(\simeq)$.

Case $\beta_{\tilde{U}}$:

$$\Gamma \vdash \texttt{case}\ (\texttt{box}\ V)\ \texttt{of}\ \{\texttt{box}\ a \longrightarrow P\} = P[V/a] : \tau$$

Note that we prove this for the computation case where $\tau = \tau$ and $P = M$, but the proof follows in a similar manner for the computation case since the same transitions exist. Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, we must show that

$$((\Phi_l, \Sigma_l, \texttt{case}\ (\texttt{box}\ V)\ \texttt{of}\ \{\texttt{box}\ a \longrightarrow M\})$$
$$, (\Phi_r, \Sigma_r, M[V/x])) \in \mathit{Comp}[\![\tau]\!].$$

It is enough to show that the pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$ by Proposition 7.8. Thus, further consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left, there is the transition:

$$\begin{aligned}\langle\!\langle \Phi_l \parallel \Sigma_l \parallel \texttt{case}\ (\texttt{box}\ V)\ \texttt{of}\ \{\texttt{box}\ x \to M\} \parallel K_l \rangle\!\rangle &\longmapsto \\ \langle\!\langle \Phi_l \parallel \Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x \parallel M \parallel K_l \rangle\!\rangle\end{aligned}$$

Note that $\Gamma \vdash \texttt{box}\ V : \tilde{U}\ \sigma$ and $\Gamma, x{:}\sigma \vdash M : \tau$ follows from the equality. By inversion on the former, we know $\Gamma \vdash V : \sigma$. Applying Corollary 7.1 with the assumed related environment, we know

$$\begin{aligned}&((\Phi_l', (\Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x), M) \\ &, (\Phi_r', \Sigma_r, M[V/x])) \in \mathit{Comp}[\![\tau]\!].\end{aligned}$$

By Proposition 7.8, the pair is in the set $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$.

$\langle\!\langle \Phi_l' \parallel \Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r' \parallel \Sigma_r \parallel M[V/x] \parallel K_r \rangle\!\rangle$ by the above related expressions with $\Phi_l' \sqsupseteq \Phi_l$, $\Phi_r' \sqsupseteq \Phi_r$, and $((\Phi_l', K_l), (\Phi_r', K_r)) \in \mathit{Stack}[\![\tau]\!]$.

$\langle\!\langle \Phi_l' \parallel \Sigma_l \parallel \texttt{case}\ (\texttt{box}\ V)\ \texttt{of}\ \{\texttt{box}\ x \to M\} \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r' \parallel \Sigma_r \parallel M[V/x] \parallel K_r \rangle\!\rangle$ by the backwards closure of $(\simeq)$.

Case $\beta_{\hat{F}}$:

$$\Gamma \vdash (\texttt{val}\ V)\ \texttt{to}\ x\ \texttt{in}\ P = P[V/x] : \tau$$

Note that we prove this for the shared computation case where $P = R$ and $\tau = \tau$, but the proof follows in the same manner for shared computations since the same transitions exist. Considering arbitrary related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, we must show that

$$((\Phi_l, \Sigma_l, (\texttt{val}\ V)\ \texttt{to}\ x\ \texttt{in}\ R), (\Phi_r, \Sigma_r, R[V/x])) \in \mathit{Shared}[\![\tau]\!].$$

By Proposition 7.8, it is enough to show that the pair is in the set $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$. Thus, further consider some $\Phi_l' \sqsupseteq \Phi_l$, $\Phi_r' \sqsupseteq \Phi_r$, and $((\Phi_l', K_l), (\Phi_r', K_l)) \in \mathit{Stack}[\![\tau]\!]$:

On the left, there is the transition sequence:

$$\begin{aligned} \langle\!\langle \Phi_l' \parallel \Sigma_l \parallel (\mathtt{val}\ V)\ \mathtt{to}\ x\ \mathtt{in}\ R \parallel K_l \rangle\!\rangle &\longmapsto \\ \langle\!\langle \Phi_l' \parallel \Sigma_l \parallel \mathtt{val}\ V \parallel (\Sigma_l, \square\ \mathtt{to}\ x\ \mathtt{in}\ R) \cdot K_l \rangle\!\rangle &\longmapsto \\ \langle\!\langle \Phi_l' \parallel \Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x \parallel R \parallel K_l \rangle\!\rangle & \end{aligned}$$

Note that $\Gamma \vdash (\mathtt{val}\ V)\ \mathtt{to}\ x\ \mathtt{in}\ R : \tau$ follows from the equality. By inversion, we can conclude that $\Gamma \vdash V : \sigma$ and $\Gamma, x{:}\sigma \vdash R : \tau$. By Corollary 7.1 with our related environments, we know that

$$((\Phi_l, (\Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x), R), (\Phi_r, \Sigma_r, R[V/x])) \in \mathit{Shared}[\![\tau]\!].$$

And further, Proposition 7.8 says that the pair is in the set $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$.

$\langle\!\langle \Phi_l' \parallel \Sigma_l, \mathrm{Build}_V(\Sigma_l, V)/x \parallel R \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r' \parallel \Sigma_r \parallel R[V/x] \parallel K_r \rangle\!\rangle$, by the property of the related computations above with $\Phi_l' \sqsupseteq \Phi_l$, $\Phi_r' \sqsupseteq \Phi_r$, and $((\Phi_l', K_l), (\Phi_r', K_r)) \in \mathit{Stack}[\![\tau]\!]$.

$\langle\!\langle \Phi_l' \parallel \Sigma_l \parallel (\mathtt{val}\ V)\ \mathtt{to}\ x\ \mathtt{in}\ R \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r' \parallel \Sigma_r \parallel R[V/x] \parallel K_r \rangle\!\rangle$, by the closure properties of $(\simeq)$.

Case $\beta_{\check{U}}$:

$$\Gamma \vdash \{\varsigma, \mathtt{enter} \rightarrow M\}.\mathtt{enter} = M[\varsigma] : \tau$$

Follows in a similar manner to the β_U case, except for the fact that there is an evaluation context for $\square.\mathtt{enter}$. Thus, we have an extra step on the left side:

$$\begin{aligned}
\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\varsigma, \mathtt{enter} \to M\}.\mathtt{enter} \parallel K_l \rangle\!\rangle &\longmapsto \\
\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\varsigma, \mathtt{enter} \to M\} \parallel \square.\mathtt{enter} \cdot K_l \rangle\!\rangle &\longmapsto \\
\langle\!\langle \Phi'_l \parallel \Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma) \parallel M \parallel K_l \rangle\!\rangle &
\end{aligned}$$

The transitivity of $(\simeq)$ permits this extra step without issue.

Case $\beta_{\tilde{F}}$:

$$\Gamma \vdash \{\mathtt{eval} \to R\}.\mathtt{eval} = R : \tau$$

Follows in a similar manner to the β_U case and the case above, except that there is no closure to enter as well. The transitions on the left side look like the following:

$$\begin{aligned}
\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\mathtt{eval} \to R\}.\mathtt{eval} \parallel K_l \rangle\!\rangle &\longmapsto \\
\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\mathtt{eval} \to R\} \parallel \square.\mathtt{eval} \cdot K_l \rangle\!\rangle &\longmapsto \\
\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R \parallel K_l \rangle\!\rangle &
\end{aligned}$$

Case $\eta_\to$:

$$\Gamma \vdash \lambda x.\, M\ x = M : \tau \to \sigma$$

Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, we must prove $((\Phi_l, \Sigma_l, \lambda x.\, M\ x), (\Phi_l, \Sigma_r, M)) \in \mathit{Comp}[\![\tau \to \sigma]\!]$. By definition, we must show that the pair is in $\mathit{Elim}[\![\tau \to \sigma]\!]^{\top\!\!\top}$. Thus, further consider an arbitrary $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, \square\ \mathbb{V} \cdot K_l), (\Phi'_r, \square\ \mathbb{W} \cdot K_r)) \in \mathit{Elim}[\![\tau \to \sigma]\!]$:

By definition, we know $((\Phi'_l, \mathbb{V}), (\Phi'_r, \mathbb{W})) \in \mathit{Val}[\![\tau]\!]$ and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\sigma]\!]$.

Double orthogonal inclusion yields that $((\Phi'_l, \square\ \mathbb{V} \cdot K_l), (\Phi'_r, \square\ \mathbb{W} \cdot K_r)) \in$ $\mathit{Stack}[\![\tau \longrightarrow \sigma]\!]$.

The left side has the following transition sequence:

$$\begin{aligned} \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \lambda x.\, M\ x \parallel \square\ \mathbb{V} \cdot K_l \rangle\!\rangle &\longmapsto \\ \langle\!\langle \Phi'_l \parallel \Sigma_l, \mathbb{V}/x \parallel M\ x \parallel K_l \rangle\!\rangle &\longmapsto \\ \langle\!\langle \Phi'_l \parallel \Sigma_l, \mathbb{V}/x \parallel M \parallel \square\ \mathbb{V} \cdot K_l \rangle\!\rangle & \end{aligned}$$

Note that $\Gamma \vdash M : \tau \longrightarrow \sigma$ follows from the equality. By the reflexivity of $(\approx)$, we know that $\Gamma \vDash M \approx M : \tau \longrightarrow \sigma$.

With $((\Phi'_l, (\Sigma_l, \mathbb{V}/x)), (\Phi'_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ (which are related by Lemma 7.4 with our assumed environments and closure under accessible worlds), we know that $((\Phi'_l, (\Sigma_l, \mathbb{V}/x), M), (\Phi'_r, \Sigma_r, M)) \in \mathit{Comp}[\![\tau \longrightarrow \sigma]\!]$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l, \mathbb{V}/x \parallel M \parallel \square\ \mathbb{V} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M \parallel \square\ \mathbb{W} \cdot K_r \rangle\!\rangle$ by the property the stacks $((\Phi'_l, \square\ \mathbb{V} \cdot K_l), (\Phi'_r, \square\ \mathbb{W} \cdot K_r)) \in \mathit{Stack}[\![\tau \longrightarrow \sigma]\!]$ with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_l \sqsupseteq \Phi'_l$, and the related expressions above.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \lambda x.\, M\ x \parallel \square\ \mathbb{V} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M \parallel \square\ \mathbb{W} \cdot K_r \rangle\!\rangle$ by our closure properties of $(\simeq)$.

Case $\eta_{\&}$:

$$\Gamma \vdash \{\mathtt{fst} \longrightarrow M.\mathtt{fst}; \mathtt{snd} \longrightarrow M.\mathtt{snd}\} = M : \tau \mathbin{\&} \sigma$$

Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, and then showing that

$$\begin{aligned} &((\Phi_l, \Sigma_l, \{\mathtt{fst} \longrightarrow M.\mathtt{fst}; \mathtt{snd} \longrightarrow M.\mathtt{snd}\}) \\ &, (\Phi_r, \Sigma_r, M)) \in \mathit{Comp}[\![\tau \mathbin{\&} \sigma]\!] \end{aligned}$$

requires that the pair is in $\mathit{Elim}[\![\tau \,\&\, \sigma]\!]^{\top\!\top}$ by definition. Thus, further consider an arbitrary $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and an element in $\mathit{Elim}[\![\tau \,\&\, \sigma]\!]$, there are two sub-cases to consider:

case the element is $((\Phi'_l, \square.\mathtt{fst} \cdot K_l), (\Phi'_r, \square.\mathtt{fst} \cdot K_r))$ where $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

By double orthogonal inclusion and the definition of $\mathit{Stack}[\![\tau \,\&\, \sigma]\!]$, we know $((\Phi'_l, \square.\mathtt{fst} \cdot K_l), (\Phi'_r, \square.\mathtt{fst} \cdot K_r)) \in \mathit{Stack}[\![\tau \,\&\, \sigma]\!]$ and thus the pair is in the set $\mathit{Comp}[\![\tau \,\&\, \sigma]\!]^{\perp\!\!\perp}$.

On the left side, we have the following transition sequence:

$$\begin{aligned} \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\mathtt{fst} \to M.\mathtt{fst}; \mathtt{snd} \to M.\mathtt{snd}\} \parallel \square.\mathtt{fst} \cdot K_l \rangle\!\rangle &\longmapsto \\ \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M.\mathtt{fst} \parallel K_l \rangle\!\rangle &\longmapsto \\ \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \parallel \square.\mathtt{fst} \cdot K_l \rangle\!\rangle & \end{aligned}$$

Note that $\Gamma \vdash M : \tau \,\&\, \sigma$ follows from the equality. By the reflexivity of $(\approx)$, we have $\Gamma \vdash M \approx M : \tau \,\&\, \sigma$. This in combination with the related environments $((\Phi'_l, \Sigma_l), (\Phi'_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ (closure under accessible worlds), we can now conclude $((\Phi'_l, \Sigma_l, M), (\Phi'_r, \Sigma_r, M)) \in \mathit{Comp}[\![\tau \,\&\, \sigma]\!]$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \parallel \square.\mathtt{fst} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M \parallel \square.\mathtt{fst} \cdot K_r \rangle\!\rangle$ by the definition of the stacks $((\Phi'_l, \square.\mathtt{fst} \cdot K_l), (\Phi'_r, \square.\mathtt{fst} \cdot K_r)) \in \mathit{Stack}[\![\tau \,\&\, \sigma]\!]$ with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and the related computations above.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\mathtt{fst} \to M.\mathtt{fst}; \mathtt{snd} \to M.\mathtt{snd}\} \parallel \square.\mathtt{fst} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M \parallel \square.\mathtt{fst} \cdot K_r \rangle\!\rangle$ by the closure properties of $(\simeq)$.

case the element is $((\Phi'_l, \square.\mathtt{snd} \cdot K_l), (\Phi'_r, \square.\mathtt{snd} \cdot K_r))$ where $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\sigma]\!]$:

This follows in the same manner as the sub-case above.

Case $\eta_\otimes$:

$$\Gamma \vdash \mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y\rangle \to P[\langle x, y\rangle/z]\} = P[V/z] : \tau$$

Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ and then showing that

$$((\Phi_l, \Sigma_l, \mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y\rangle \to M[\langle x, y\rangle/z]\})$$
$$, (\Phi_r, \Sigma_r, M[V/z])) \in \mathit{Comp}[\![\tau]\!],$$

can be done by proving that the pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$ by the Proposition 7.8. We prove this for the case that $P = M$ and $\tau = \tau$, but the proof follows similarly for a shared computation in $\mathit{Shared}[\![\tau]\!]$. Thus, further consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

Note that $\Gamma \vdash V : \sigma \otimes \rho$, and $\Gamma, x{:}\sigma, y{:}\rho \vdash M[\langle x, y\rangle/z] : \tau$ follow from the equality.

By the reflexivity of $(\approx)$, we know that $\Gamma \vDash V \approx V : \sigma \otimes \rho$. With environments $((\Phi'_l, \Sigma_l), (\Phi'_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ (note closure under accessible worlds), we may conclude $((\Phi'_l, \mathrm{Build}_V(\Sigma_l, V)), (\Phi'_r, \mathrm{Build}_V(\Sigma_r, V))) \in \mathrm{Val}[\![\sigma \otimes \rho]\!]$. Furthermore, we know by this definition that $\mathrm{Build}_V(\Sigma_l, V) = \langle \mathbb{V}_l, \mathbb{W}_l\rangle$ and $\mathrm{Build}_V(\Sigma_r, V) = \langle \mathbb{V}_r, \mathbb{W}_r\rangle$.

On the left side, we have the transition:

$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y\rangle \to M[\langle x, y\rangle/z]\} \parallel K_l\rangle\!\rangle \longmapsto$$
$$\langle\!\langle \Phi'_l \parallel \Sigma_l, \mathbb{V}_l/x, \mathbb{W}_l/y \parallel M[\langle x, y\rangle/z] \parallel K_l\rangle\!\rangle$$

$\langle\!\langle \Phi'_l \parallel \Sigma_l, \mathbb{V}_l/x, \mathbb{W}_l/y \parallel M[\langle x, y\rangle/z] \parallel K_l\rangle\!\rangle \simeq \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M[\langle \mathbb{V}_l, \mathbb{W}_l\rangle/z] \parallel K_l\rangle\!\rangle$ follows by Corollary 7.1 with the assumed environments then using the reflexively related stack (Φ'_l, K_l).

We have the following chain of reasoning:

$$\begin{aligned}
\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow M[\langle x, y\rangle/z]\} \parallel K_l\rangle\!\rangle &\simeq \\
\langle\!\langle \Phi'_l \parallel \Sigma_l, \mathbb{V}_l/x, \mathbb{W}_l/y \parallel M[\langle x, y\rangle/z] \parallel K_l\rangle\!\rangle &\simeq \\
\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M[\langle \mathbb{V}_l, \mathbb{W}_l\rangle/z] \parallel K_l\rangle\!\rangle &\simeq \\
\langle\!\langle \Phi'_l \parallel \Sigma_l, \langle \mathbb{V}_l, \mathbb{W}_l\rangle/z \parallel M \parallel K_l\rangle\!\rangle &\simeq \\
\langle\!\langle \Phi'_r \parallel \Sigma_r, \langle \mathbb{V}_r, \mathbb{W}_r\rangle/z \parallel M \parallel K_r\rangle\!\rangle &\simeq \\
\langle\!\langle \Phi'_r \parallel \Sigma_r, \mathrm{Build}_V(\Sigma_r, V)/z \parallel M \parallel K_r\rangle\!\rangle &\simeq \\
\langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M[V/z] \parallel K_r\rangle\!\rangle &
\end{aligned}$$

We may conclude by the closure properties of $(\simeq)$ that $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y\rangle \longrightarrow M[\langle x, y\rangle/z]\} \parallel K_l\rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M[V/z] \parallel K_r\rangle\!\rangle$.

Case η_U:

$$\Gamma \vdash \{\mathsf{force} \longrightarrow V.\mathsf{force}\} = V : \tau$$

Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in Env[\![\Gamma']\!]$, we must show that

$$\begin{aligned}
&((\Phi_l, \mathrm{Build}_V(\Sigma_l, \{\mathsf{force} \longrightarrow V.\mathsf{force}\}))\\
&, (\Phi_r, \mathrm{Build}_V(\Sigma_r, V))) \in Val[\![U\ \sigma]\!].
\end{aligned}$$

By the reflexivity of $(\approx)$, we know that $\Gamma \vDash V \approx V : U\ \sigma$. By that definition with our related environments, we know $((\Phi_l, \mathrm{Build}_V(\Sigma_l, V)), (\Phi_r, \mathrm{Build}_V(\Sigma_r, V))) \in Val[\![U\ \sigma]\!]$. Unfolding the definition further, we have that $\mathrm{Build}_V(\Sigma_l, V) = \{\Sigma'_l, \mathsf{force} \longrightarrow M\}$ and $\mathrm{Build}_V(\Sigma_r, V) = \{\Sigma'_r, \mathsf{force} \longrightarrow N\}$ such that their bodies are related computations $((\Phi_l, \Sigma'_l, M), (\Phi_r, \Sigma'_r, N)) \in Comp[\![\sigma]\!]$.

By the definition of building, we know also that

$$\mathrm{Build}_V(\Sigma_l, \{\mathtt{force} \longrightarrow V.\mathtt{force}\}) = \{\Sigma_l, \mathtt{force} \longrightarrow V.\mathtt{force}\}.$$

We have left to show that $((\Phi_l, \Sigma_l, V.\mathtt{force}), (\Phi_r, \Sigma_r', N)) \in \mathit{Comp}[\![\sigma]\!]$. By Proposition 7.8, we need only show that the pair is in $\mathit{Stack}[\![\sigma]\!]^{\top\!\top}$. Thus, consider some $\Phi_l' \sqsupseteq \Phi_l$, $\Phi_r' \sqsupseteq \Phi_r$, and $((\Phi_l', K_l), (\Phi_r', K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left side, we have the following transition

$$\begin{aligned} \langle\!\langle \Phi_l' \parallel \Sigma_l \parallel V.\mathtt{force} \parallel K_l \rangle\!\rangle \quad &\longmapsto \\ \langle\!\langle \Phi_l' \parallel \Sigma_l' \parallel M \parallel K_l \rangle\!\rangle & \end{aligned}$$

$\langle\!\langle \Phi_l' \parallel \Sigma_l' \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r' \parallel \Sigma_r' \parallel N \parallel K_r \rangle\!\rangle$, by Proposition 7.8 with our related computations $((\Phi_l', \Sigma_l', M), (\Phi_r', \Sigma_r', N)) \in \mathit{Comp}[\![\tau]\!]$ (note closure under accessible world) with $\Phi_l' \sqsupseteq \Phi_l'$, $\Phi_l' \sqsupseteq \Phi_l'$, and $((\Phi_l', K_l), (\Phi_r', K_r)) \in \mathit{Stack}[\![\tau]\!]$.

$\langle\!\langle \Phi_l' \parallel \Sigma_l \parallel V.\mathtt{force} \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r' \parallel \Sigma_r' \parallel N \parallel K' \rangle\!\rangle$, by the closure properties of $(\simeq)$.

Case η_F:

$$\Gamma \vdash M \;\mathtt{to}\; x \;\mathtt{in}\; E[\mathtt{ret}\; x] = E[M] : \tau$$

Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ and then showing that

$$\begin{gathered} ((\Phi_l, \Sigma_l, M \;\mathtt{to}\; x \;\mathtt{in}\; E[\mathtt{ret}\; x]) \\ , (\Phi_r, \Sigma_r, E[M])) \in \mathit{Comp}[\![\tau]\!], \end{gathered}$$

it will suffice to show that the pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$ by Proposition 7.8. Note that we consider only the computation case here, but the shared computation case follows similarly. Thus, consider further an arbitrary $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left side, there is the following transition:

$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \texttt{ to } x \texttt{ in } E[\texttt{ret } x] \parallel K_l \rangle\!\rangle \longmapsto$$
$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \parallel (\Sigma, \square \texttt{ to } x \texttt{ in } E[\texttt{ret } x]) \cdot K_l \rangle\!\rangle$$

On the right side, the properties of $(\simeq)$ give us that $\langle\!\langle \Phi'_r \parallel \Sigma_r \parallel E[M] \parallel K_r \rangle\!\rangle \simeq \langle\!\langle \Phi''_r \parallel \Sigma'_r \parallel M \parallel K'_r \rangle\!\rangle$ where $\mathrm{Build}_K(\Phi'_r, \Sigma_r, E, K_r) = (\Phi''_r, \Sigma'_r, K'_r)$. Note that Σ'_r is Σ_r extended with more bindings.

Note that $\Gamma \vdash M : F\ \sigma$ follows from the typing derivation. By the reflexivity of $(\approx)$, we have $\Gamma \vDash M \approx M : F\ \sigma$. With related environments $((\Phi'_l, \Sigma_l), (\Phi''_r, \Sigma'_r)) \in \mathit{Env}[\![\Gamma]\!]$ (note closure on accessible worlds), we know that $((\Phi'_l, \Sigma_l, M), (\Phi''_r, \Sigma'_r, M)) \in \mathit{Comp}[\![F\ \sigma]\!]$.

Next, we prove that $((\Phi'_l, (\Sigma_l, \square \texttt{ to } x \texttt{ in } E[\texttt{ret } x]) \cdot K_l), (\Phi''_r, K'_r)) \in \mathit{Stack}[\![F\ \sigma]\!]$, by considering $\Phi''_l \sqsupseteq \Phi'_l$, $\Phi'''_r \sqsupseteq \Phi''_r$, and $((\Phi''_l, \Sigma'_l, \texttt{ret } V), (\Phi'''_r, \Sigma''_r, \texttt{ret } W)) \in \mathit{Intro}[\![F\ \sigma]\!]$:

$((\Phi''_l, \mathrm{Build}_V(\Sigma'_l, V)), (\Phi'''_r, \mathrm{Build}_V(\Sigma''_r, W))) \in \mathit{Val}[\![\sigma]\!]$ follows from the assumption that $\mathit{Intro}[\![F\ \sigma]\!]$. From this, we have

$$((\Phi''_l, \mathrm{Build}_V((\Sigma'_l, \mathrm{Build}_V(\Sigma'_l, V)/x), x))$$
$$, (\Phi'''_r, \mathrm{Build}_V(\Sigma''_r, W))) \in \mathit{Val}[\![\sigma]\!].$$

And it follows by definition that

$$((\Phi_l'', (\Sigma_l', \mathrm{Build}_V(\Sigma_l', V)/x), \mathtt{ret}\ x)$$
$$, (\Phi_r''', \Sigma_r'', \mathtt{ret}\ W)) \in \mathit{Intro}[\![F\ \sigma]\!].$$

On the left side, there is the transition:

$$\langle\!\langle \Phi_l'' \parallel \Sigma_l'' \parallel \mathtt{ret}\ V \parallel (\Sigma_l, \square\ \mathtt{to}\ x\ \mathtt{in}\ E[\mathtt{ret}\ x]) \cdot K_l \rangle\!\rangle \longmapsto$$
$$\langle\!\langle \Phi_l'' \parallel \Sigma_l', \mathrm{Build}_V(\Sigma_l, V)/x \parallel E[\mathtt{ret}\ x] \parallel K_l \rangle\!\rangle$$

And this is observationally equivalent to the following by the closure properties of $(\simeq)$: $\langle\!\langle \Phi_l'' \parallel \Sigma_l, \mathrm{Build}(\Sigma_l', V)/x \parallel E[\mathtt{ret}\ x] \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_l''' \parallel \Sigma_l'' \parallel \mathtt{ret}\ x \parallel K_l' \rangle\!\rangle$ where building the stack $\mathrm{Build}_K(\Phi_l'', (\Sigma_l, \mathrm{Build}(\Sigma_l', V)/x), E, K_l)$ is equal to $(\Phi_l''', \Sigma_l'', K_l')$.

We know $((\Phi_l''', K_l), (\Phi_r''', K_r)) \in \mathit{Stack}[\![F\ \sigma]\!]$ by Lemma 7.5. This with $\Phi_l''' \sqsupseteq \Phi_l'''$, $\Phi_r''' \sqsupseteq \Phi_r'''$, and the related introductions (note closure over accessible heaps) yields $\langle\!\langle \Phi_l''' \parallel \Sigma_l'' \parallel \mathtt{ret}\ x \parallel K_l' \rangle\!\rangle \simeq \langle\!\langle \Phi_r''' \parallel \Sigma_r'' \parallel \mathtt{ret}\ W \parallel K_r' \rangle\!\rangle$.

$\langle\!\langle \Phi_l'' \parallel \Sigma_l'' \parallel \mathtt{ret}\ V \parallel (\Sigma_l, \square\ \mathtt{to}\ x\ \mathtt{in}\ E[\mathtt{ret}\ x]) \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r''' \parallel \Sigma_r'' \parallel \mathtt{ret}\ W \parallel K_r' \rangle\!\rangle$ by the closure properties of $(\simeq)$.

By Proposition 7.8 with $\Phi_l' \sqsupseteq \Phi_l'$, $\Phi_r'' \sqsupseteq \Phi_r''$, and the related computations above, we know $\langle\!\langle \Phi_l' \parallel \Sigma_l \parallel M \parallel (\Sigma_l, \square\ \mathtt{to}\ x\ \mathtt{in}\ E[\mathtt{ret}\ x]) \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r'' \parallel \Sigma_r' \parallel M \parallel K_r' \rangle\!\rangle$.

By the closure properties of $(\simeq)$, we have $\langle\!\langle \Phi_l' \parallel \Sigma_l \parallel M\ \mathtt{to}\ x\ \mathtt{in}\ E[\mathtt{ret}\ x] \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r' \parallel \Sigma_r \parallel E[M] \parallel K_r \rangle\!\rangle$.

Case $\eta_{\tilde{U}}$:

$$\Gamma \vdash \mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{box}\ a \longrightarrow P[\mathsf{box}\ a/x]\} = P[V/x] : \tau$$

Proved in a similar manner to $\eta_\otimes$.

Case $\eta_{\hat{F}}$:

$$\Gamma \vdash R \; \texttt{to} \; x \; \texttt{in} \; E[\texttt{val} \; x] = E[R] : \tau$$

Proved in a similar manner to η_F.

Case $\eta_{\check{U}}$:

$$\Gamma \vdash \{\texttt{enter} \to V.\texttt{enter}\} = V : \check{U} \; \tau$$

Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, we must show that

$$((\Phi_l, \Sigma_l, \{\texttt{enter} \to V.\texttt{enter}\}), (\Phi_r, \Sigma_r, V)) \in \mathit{Shared}[\![\check{U} \; \tau]\!].$$

Note that this equality gives $\Gamma \vdash V : \check{U} \; \tau$. By the reflexivity of $(\approx)$, we have thus $\Gamma \vDash V \approx V : \check{U} \; \tau$. We will show $((\Phi_l, \Sigma_l, \{\texttt{enter} \to V.\texttt{enter}\}), (\Phi_r, \Sigma_r, V)) \in \mathit{Val}[\![\check{U} \; \tau]\!]$ and this concludes the proof in this case because $\mathit{Val}[\![\check{U} \; \tau]\!] \subseteq \mathit{Shared}[\![\check{U} \; \tau]\!]$. By definition, this is to show that the pair is in $\mathit{Elim}[\![\check{U} \; \tau]\!]^{\top\!\top}$; we already satisfy the requirement that the expressions be shared values. Consider further some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, \square.\texttt{enter} \cdot K_l), (\Phi_r, \square.\texttt{enter} \cdot K_r)) \in \mathit{Elim}[\![\check{U} \; \tau]\!]$:

On the left, there is the transition sequence:

$$\begin{aligned}
\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\texttt{enter} \to V.\texttt{enter}\} \parallel \square.\texttt{enter} \cdot K_l \rangle\!\rangle &\longmapsto \\
\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel V.\texttt{enter} \parallel K_l \rangle\!\rangle &\longmapsto \\
\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel V \parallel \square.\texttt{enter} \cdot K_l \rangle\!\rangle &
\end{aligned}$$

$((\Phi'_l, \Sigma_l, V), (\Phi'_r, \Sigma_r, V)) \in \mathit{Val}[\![\check{U} \; \tau]\!]$ from our initial assumption with the relate environments (closed under future heaps).

Since $\mathit{Elim}[\![\check{U}\ \tau]\!] \subseteq \mathit{VStack}[\![\check{U}\ \tau]\!]$, we may conclude that $((\Phi'_l, \square.\mathtt{enter} \cdot K_l), (\Phi'_r, \square.\mathtt{enter} \cdot K_r)) \in \mathit{VStack}[\![\check{U}\ \tau]\!]$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel V \parallel \square.\mathtt{enter} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel V \parallel \square.\mathtt{enter} \cdot K_r \rangle\!\rangle$, by the above related value stacks with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and the related environments above.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\mathtt{enter} \to V.\mathtt{enter}\} \parallel \square.\mathtt{enter} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel V \parallel \square.\mathtt{enter} \cdot K_r \rangle\!\rangle$ by the closure properties of $(\simeq)$.

Case $\eta_{\tilde{F}}$:

$$\Gamma \vdash \{\mathtt{eval} \to M.\mathtt{eval}\} = M : \tilde{F}\ \tau$$

Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, and then showing that

$$((\Phi_l, \Sigma_l, \{\mathtt{eval} \to M.\mathtt{eval}\}), (\Phi_r, \Sigma_r, M)) \in \mathit{Comp}[\![\tilde{F}\ \tau]\!]$$

requires that the pair is in $\mathit{Elim}[\![\tilde{F}\ \tau]\!]^{\top\!\!\top}$ by definition. Thus, further considering an arbitrary $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, \square.\mathtt{eval} \cdot K_l), (\Phi'_r, \square.\mathtt{eval} \cdot K_r)) \in \mathit{Elim}[\![\tilde{F}\ \tau]\!]$:

By double orthogonal inclusion and the definition of $\mathit{Stack}[\![\tilde{F}\ \tau]\!]$, we may conclude that $((\Phi'_l, \square.\mathtt{eval} \cdot K_l), (\Phi'_r, \square.\mathtt{eval} \cdot K_r)) \in \mathit{Stack}[\![\tilde{F}\ \tau]\!]$ and therefore the pair is in the set $\mathit{Comp}[\![\tilde{F}\ \tau]\!]^{\perp\!\!\perp}$ by Proposition 7.8.

On the left side, we have the following transition sequence:

$$\begin{aligned}
\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\mathtt{eval} \to M.\mathtt{eval}\} \parallel \square.\mathtt{eval} \cdot K_l \rangle\!\rangle &\longmapsto \\
\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M.\mathtt{eval} \parallel K_l \rangle\!\rangle &\longmapsto \\
\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \parallel \square.\mathtt{eval} \cdot K_l \rangle\!\rangle &
\end{aligned}$$

Note that $\Gamma \vdash M : \tilde{F}\,\tau$ follows from the equality. By the reflexivity of $(\approx)$, we have $\Gamma \vdash M \approx M : \tilde{F}\,\tau$. With the related environments $((\Phi'_l, \Sigma_l), (\Phi'_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ (closure under accessible worlds), we can now conclude that $((\Phi'_l, \Sigma_l, M), (\Phi'_r, \Sigma_r, M)) \in \mathit{Comp}[\![\tilde{F}\,\tau]\!]$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel M \parallel \square.\mathtt{eval} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M \parallel \square.\mathtt{eval} \cdot K_r \rangle\!\rangle$, by the property of our related stacks with $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and the related computations immediately above.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\mathtt{eval} \longrightarrow M.\mathtt{eval}\} \parallel \square.\mathtt{eval} \cdot K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M \parallel \square.\mathtt{eval} \cdot K_r \rangle\!\rangle$, by the closure properties of $(\simeq)$.

Case κ:

$$\Gamma \vdash E[R\ \mathsf{memo}\ a\ \mathtt{in}\ P] = R\ \mathsf{memo}\ a\ \mathtt{in}\ E[P] : \tau$$

Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ and then showing that

$$((\Phi_l, \Sigma_l, E[R\ \mathsf{memo}\ a\ \mathtt{in}\ M])$$
$$, (\Phi_r, \Sigma_r, R\ \mathsf{memo}\ a\ \mathtt{in}\ E[M])) \in \mathit{Comp}[\![\tau]\!]$$

requires that we show the pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$ by Proposition 7.8. Note we pick the computation case; the shared case follows similarly except we make use of the definition $\mathit{Shared}[\![\tau]\!] = \mathit{VStack}[\![\tau]\!]^{\top\!\top}$. Thus, further consider an arbitrary $\Phi'_l \sqsupseteq \Phi_l$ and $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left, $\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel E[R\ \mathsf{memo}\ a\ \mathtt{in}\ M] \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi''_l \parallel \Sigma'_l, \parallel R\ \mathsf{memo}\ a\ \mathtt{in}\ M \parallel K'_l \rangle\!\rangle$ where we know $\mathrm{Build}_K(\Phi'_l, \Sigma, E, K_l) = (\Phi''_l, \Sigma'_l, K'_l)$ by the closure properties of $(\simeq)$. Thereafter, there is the transition:

$$\begin{aligned} &\langle\!\langle \Phi''_l \parallel \Sigma'_l \parallel R\ \texttt{memo}\ a\ \texttt{in}\ M \parallel K'_l \rangle\!\rangle \quad \longmapsto \\ \langle\!\langle \Phi''_l, l_a \mapsto \{\Sigma'_l, R\} \parallel \Sigma'_l, l_a/a \parallel M \parallel K'_l \rangle\!\rangle & \end{aligned}$$

On the right side, there is the transition

$$\begin{aligned} &\langle\!\langle \Phi'_r \parallel \Sigma_r \parallel R\ \texttt{memo}\ a\ \texttt{in}\ E[M] \parallel K_r \rangle\!\rangle) \quad \longmapsto \\ \langle\!\langle \Phi'_r, l_a \mapsto \{\Sigma_r, R\} \parallel \Sigma_r, l_a/a \parallel E[M] \parallel K_r \rangle\!\rangle & \end{aligned}$$

Additionally, the fourth closure property of $(\simeq)$ yields $\langle\!\langle \Phi'_r, l_a \mapsto \{\Sigma_r, R\} \parallel \Sigma_r, l_a/a \parallel E[M] \parallel K_r \rangle\!\rangle \simeq \langle\!\langle \Phi''_r \parallel \Sigma'_r \parallel M \parallel K'_r \rangle\!\rangle$ on the right, where we know that $\mathrm{Build}_K((\Phi'_r, l_a \mapsto \{\Sigma_r, R\}), (\Sigma_r, l_a/a), K_r)$ is equal to $(\Phi''_r, \Sigma'_r, K'_r)$.

Note that $\Gamma \vdash R : \sigma$, $\Gamma, a{:}\sigma \vdash E[M] : \tau$, and $(\Gamma, a{:}\sigma)\Gamma' \vdash M : \rho$ follows from the equality. Γ' are extra shared variables that are added to the environment from other memo-expressions. It follows by the reflexivity of $(\approx)$ that $\Gamma \vdash R \approx R : \sigma$ and $(\Gamma, a{:}\sigma)\Gamma' \vdash M \approx M : \rho$.

$((\Phi'_l, \Sigma_l, R), (\Phi'_r, \Sigma_r, R)) \in \mathit{Shared}[\![\sigma]\!]$ follows from the above fact with our related environments.

$$\begin{aligned} &(((\Phi''_l, l_a \mapsto \{\Sigma'_l, R\}) \\ &, (\Sigma'_l, l_a/a)), (\Phi''_r, \Sigma'_r)) \in \mathit{Env}[\![(\Gamma, a{:}\sigma)\Gamma']\!] \end{aligned}$$

by Lemma 7.1 using empty local environments and that building stacks produces future heaps and environments.

$$\begin{aligned} &(((\Phi''_l, l_a \mapsto \{\Sigma'_l, R\}) \\ &, (\Sigma'_l, l_a/a), M), (\Phi''_r, \Sigma'_r, M)) \in \mathit{Comp}[\![\rho]\!] \end{aligned}$$

follows with the environment above.

$(((\Phi_l'', l_a \mapsto \{\Sigma_l', R\}), K_l'), (\Phi_r'', K_r')) \in \mathit{Stack}[\![\rho]\!]$ by Lemma 7.5.

$\langle\!\langle \Phi_l'', l_a \mapsto \{\Sigma_l', R\} \parallel \Sigma_l', l_a/a \parallel M \parallel K_l' \rangle\!\rangle \simeq \langle\!\langle \Phi_r'' \parallel \Sigma_r' \parallel M \parallel K_r' \rangle\!\rangle$, by our related stacks with the same heaps Proposition 7.8 and the above related expressions. Note that for the shared case, we make use of the definition of shared stacks, *i.e.* $\mathit{Shared}[\![\tau]\!]^{\perp\!\!\perp}$.

$\langle\!\langle \Phi_l' \parallel \Sigma_l \parallel E[R \;\texttt{memo}\; a \;\texttt{in}\; M] \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi_r' \parallel \Sigma_r \parallel R \;\texttt{memo}\; a \;\texttt{in}\; E[M] \parallel K_r \rangle\!\rangle$ by the closure properties of $(\simeq)$.

Case χ:

$$\Gamma \vdash (R \;\texttt{memo}\; b \;\texttt{in}\; S) \;\texttt{memo}\; a \;\texttt{in}\; P = R \;\texttt{memo}\; b \;\texttt{in}\; (S \;\texttt{memo}\; a \;\texttt{in}\; P) : \tau$$

Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$ and then showing that

$$\begin{aligned}&((\Phi_l, \Sigma_l, (R \;\texttt{memo}\; b \;\texttt{in}\; S) \;\texttt{memo}\; a \;\texttt{in}\; M)\\ &, (\Phi_r, \Sigma_r, R \;\texttt{memo}\; b \;\texttt{in}\; (S \;\texttt{memo}\; a \;\texttt{in}\; M))) \in \mathit{Comp}[\![\tau]\!]\end{aligned}$$

requires that we show the pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\!\top}$ by Proposition 7.8. As before, the proof for the shared case follows similarly but we show the pair is in $\mathit{VStack}[\![\tau]\!]^{\top\!\!\top}$ by the definition of $\mathit{Shared}[\![\tau]\!]$. Thus, further consider an arbitrary $\Phi_l' \sqsupseteq \Phi_r'$, $\Phi_r' \sqsupseteq \Phi_r$, and $((\Phi_l', K_l), (\Phi_r', K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left side, there is the transition:

$$\begin{aligned}&\langle\!\langle \Phi_l' \parallel \Sigma_l \parallel (R \;\texttt{memo}\; b \;\texttt{in}\; S) \;\texttt{memo}\; a \;\texttt{in}\; M \parallel K_l \rangle\!\rangle \quad \longmapsto\\ &\langle\!\langle \Phi_l', l_a \mapsto \{\Sigma_l, R \;\texttt{memo}\; b \;\texttt{in}\; S\} \parallel \Sigma_l, l_a/a \parallel M \parallel K_l \rangle\!\rangle\end{aligned}$$

On the right side, there is the transition sequence:

$$\begin{aligned}
\langle\!\langle \Phi'_r \parallel \Sigma_r \parallel R\ \texttt{memo}\ b\ \texttt{in}\ (S\ \texttt{memo}\ a\ \texttt{in}\ M) \parallel K_r \rangle\!\rangle &\longmapsto \\
\langle\!\langle \Phi'_r, l_b \mapsto \{\Sigma_r, R\} \parallel \Sigma_r, l_b/b \parallel S\ \texttt{memo}\ a\ \texttt{in}\ M \parallel K_r \rangle\!\rangle &\longmapsto \\
\langle\!\langle \Phi'_r, l_b \mapsto \{\Sigma_r, R\}, l_a \mapsto \{(\Sigma_r, l_b/b), S\} \parallel \Sigma_r, l_b/b, l_a/a \parallel M \parallel K_r \rangle\!\rangle &
\end{aligned}$$

From the equality, we know that $\Gamma \vdash R : \sigma$, $\Gamma, b{:}\sigma \vdash S : \rho$, and $\Gamma, a{:}\rho \vdash M : \tau$.
Thus, the reflexivity of $(\approx)$ yields $\Gamma \vDash R \approx R : \sigma$, $\Gamma, b{:}\sigma \vDash S \approx S : \rho$, and $\Gamma, a{:}\rho \vDash M \approx M : \tau$.

We next prove

$$\begin{aligned}
&((\Phi'_l, \Sigma_l, R\ \texttt{memo}\ b\ \texttt{in}\ S) \\
&, ((\Phi'_r, l_b \mapsto \{\Sigma_r, R\}), (\Sigma_r, l_b/b), S)) \in \mathit{Shared}[\![\rho]\!]
\end{aligned}$$

by its definition where we show that the pair is in $\mathit{VStack}[\![\sigma]\!]^{\top\!\top}$. Thus, consider future worlds $\Phi''_l \sqsupseteq \Phi'_l$ and $\Phi''_r \sqsupseteq (\Phi'_r, l_b \mapsto \{\Sigma_r, R\})$, and some value stack $((\Phi''_l, \mathbb{K}_l), (\Phi''_r, \mathbb{K}_r)) \in \mathit{VStack}[\![\rho]\!]$:

On the left side, there is the transition:

$$\begin{aligned}
\langle\!\langle \Phi''_l \parallel \Sigma_l \parallel R\ \texttt{memo}\ b\ \texttt{in}\ S \parallel \mathbb{K}_l \rangle\!\rangle \quad &\longmapsto \\
\langle\!\langle \Phi''_l, l_b \mapsto \{\Sigma_l, R\} \parallel \Sigma_l, l_b/b \parallel S \parallel \mathbb{K}_l \rangle\!\rangle &
\end{aligned}$$

$((\Phi''_l, \Sigma_l), (\Phi''_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, by closure over accessible worlds and the transitivity of accessibility.

$((\Phi_l, \Sigma_l, R), (\Phi_l, \Sigma_l, R)) \in \mathit{Shared}[\![\sigma]\!]$ by the typing derivation and the reflexivity of $(\approx)$. And for all future worlds.

$$(((\Phi''_l, l_b \mapsto \{\Sigma_l, R\}), (\Sigma_l, l_b/b))$$
$$, (\Phi''_r, (\Sigma_r, l_b/b))) \in Env[\![\Gamma, b{:}\sigma]\!]$$

follows from Lemma 7.1 and closure under future worlds. Note that $\Phi''_r \sqsupseteq (\Phi'_r, l_b \mapsto \{\Sigma_r, R\})$ and thus Φ''_r will contain the same mapping for l_b.

$$(((\Phi''_l, l_b \mapsto \{\Sigma_l, R\}), (\Sigma_l, l_b/b), S)$$
$$, (\Phi''_r, (\Sigma_r, l_b/b)), S) \in Shared[\![\rho]\!],$$

follows from $\Gamma, b{:}\sigma \vDash S \approx S : \rho$ with the related environments immediately above.

$\langle\!\langle \Phi''_l, l_b \mapsto \{\Sigma_l, R\} \parallel \Sigma_l, l_b/b \parallel S \parallel \mathbb{K}_l \rangle\!\rangle \simeq \langle\!\langle \Phi''_r \parallel \Sigma_r, l_b/b \parallel S \parallel \mathbb{K}_r \rangle\!\rangle$ follows from the above fact with the property of related shared expressions, *i.e.* $Shared[\![\rho]\!] = VStack[\![\rho]\!]^{\top\!\top}$, with the future worlds $(\Phi''_l, l_b \mapsto \{\Sigma_l, R\}) \sqsupseteq \Phi''_l$ and $\Phi''_r \sqsupseteq \Phi''_r$, and the pair of value stacks $(((\Phi''_l, l_b \mapsto \{\Sigma_l, R\}), \mathbb{K}_l), (\Phi''_r, \mathbb{K}_r)) \in VStack[\![\rho]\!]$. Again note the closure over accessible heaps for the value stack.

$\langle\!\langle \Phi''_l \parallel \Sigma_l \parallel R \text{ memo } b \text{ in } S \parallel \mathbb{K}_l \rangle\!\rangle \simeq \langle\!\langle \Phi''_r \parallel \Sigma_r, l_b/b \parallel S \parallel \mathbb{K}_r \rangle\!\rangle$, by the closure properties of $(\simeq)$.

Therefore,

$$(((\Phi'_l, l_a \mapsto \{\Sigma_l, R \text{ memo } b \text{ in } S\}), (\Sigma_l, l_a/a))$$
$$, ((\Phi'_r, l_b \mapsto \{\Sigma_r, R\}, l_a \mapsto \{(\Sigma_r, l_b/b), S\}), (\Sigma_r, l_b/b, l_a/a)))$$
$$\in Env[\![\Gamma, a{:}\rho]\!]$$

follows by definition with the previous fact relating the expressions pointed to by a and Lemma 7.1.

From this,

$$\begin{aligned}&(((\Phi_l', l_a \mapsto \{\Sigma_l, R \;\texttt{memo}\; b \;\texttt{in}\; S\}), (\Sigma_l, l_a/a), M)\\ &, ((\Phi_r', l_b \mapsto \{\Sigma_r, R\}, l_a \mapsto \{(\Sigma_r, l_b/b), S\}), (\Sigma_r, l_b/b, l_a/a), M))\\ &\in \mathit{Comp}[\![\tau]\!]\end{aligned}$$

follows from the property of $\Gamma, b{:}\sigma \vDash M \approx M : \tau$.

Thus, $\langle\!\langle \Phi_l', l_a \mapsto \{\Sigma_l, R \;\texttt{memo}\; b \;\texttt{in}\; S\} \parallel \Sigma_l, l_a/a \parallel M \parallel K_l\rangle\!\rangle \simeq \langle\!\langle \Phi_r', l_b \mapsto \{\Sigma_r, R\}, l_a \mapsto \{(\Sigma_r, l_b/b), S\} \parallel \Sigma_r, l_b/b, l_a/a \parallel M \parallel K_r\rangle\!\rangle$ by our related stacks with Proposition 7.8. Note that our heaps are future heaps of assumed stack heaps. For the case where P and Q are shared expressions, we instead use of the definition of $\mathit{Shared}[\![\tau]\!]$ and that we would have assumed value stacks at the beginning.

Finally, $\langle\!\langle \Phi_l' \parallel \Sigma_l \parallel (R \;\texttt{memo}\; b \;\texttt{in}\; S) \;\texttt{memo}\; a \;\texttt{in}\; M \parallel K_l\rangle\!\rangle \simeq \langle\!\langle \Phi_r' \parallel \Sigma_r \parallel R \;\texttt{memo}\; b \;\texttt{in}\; (S \;\texttt{memo}\; a \;\texttt{in}\; M) \parallel K_r\rangle\!\rangle$ follows by the closure properties of $(\simeq)$.

Case *cl*:

$$\Gamma \vdash \{\varsigma, R\} \;\texttt{memo}\; a \;\texttt{in}\; P = R[\varsigma] \;\texttt{memo}\; a \;\texttt{in}\; P : \tau$$

Considering related environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, we must be able to show

$$\begin{aligned}&((\Phi_l, \Sigma_l, \{\varsigma, R\} \;\texttt{memo}\; a \;\texttt{in}\; P)\\ &, (\Phi_r, \Sigma_r, R[\varsigma] \;\texttt{memo}\; a \;\texttt{in}\; P)) \in \mathit{Shared}[\![\tau]\!].\end{aligned}$$

By Proposition 7.8, it is enough to show that this pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$. Thus, further consider $\Phi_l' \sqsupseteq \Phi_l$, $\Phi_r' \sqsupseteq \Phi_r$, and $((\Phi_l', K_l), (\Phi_r', K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left, there is the transition:

$$\begin{aligned} &\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\varsigma, R\}\ \texttt{memo}\ a\ \texttt{in}\ S \parallel K_l \rangle\!\rangle \quad \longmapsto \\ \langle\!\langle \Phi'_l, l \mapsto \{\Sigma_l\, &\mathrm{Build}_\varsigma(\Sigma_l, \varsigma), R\} \parallel \Sigma_l, l/a \parallel S \parallel K_l \rangle\!\rangle \end{aligned}$$

On the right, there is the transition:

$$\begin{aligned} &\langle\!\langle \Phi'_r \parallel \Sigma_r \parallel R[\varsigma]\ \texttt{memo}\ a\ \texttt{in}\ S \parallel K_r \rangle\!\rangle \quad \longmapsto \\ \langle\!\langle \Phi'_r, l \mapsto &\{\Sigma_r, R[\varsigma]\} \parallel \Sigma_r, l/a \parallel S \parallel K_r \rangle\!\rangle \end{aligned}$$

Note that both $\Gamma \vdash \varsigma : \Gamma'$, $\Gamma\Gamma' \vdash R : \rho$, and $\Gamma, a{:}\rho \vdash S : \tau$ follow from the equality.

By application of Corollary 7.1 with related environments, we are able to conclude that

$$\begin{aligned} &((\Phi'_l, \Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma), R) \\ &, (\Phi'_r, \Sigma_r, R[\varsigma])) \in \mathit{Shared}[\![\tau]\!]. \end{aligned}$$

From $\Gamma, a{:}\rho \vDash S \approx S : \tau$, we know

$$\begin{aligned} &(((\Phi'_l, l \mapsto \{\Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma), R\}), (\Sigma_l, l/a), S) \\ &, ((\Phi'_r, l \mapsto \{\Sigma_r, R[\varsigma]\}), (\Sigma_r, l/a), S)) \in \mathit{Shared}[\![\tau]\!] \end{aligned}$$

by applying Lemma 7.1 with the above related expressions.

$\langle\!\langle \Phi'_l, l \mapsto \{\Sigma_l\, \mathrm{Build}_\varsigma(\Sigma_l, \varsigma), R\} \parallel \Sigma_l, l/a \parallel S \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r, l \mapsto \{\Sigma_r, R[\varsigma]\} \parallel \Sigma_r, l/a \parallel S \parallel K_r \rangle\!\rangle$ by Proposition 7.8 together with the above related expressions and the related stacks in the future worlds $(\Phi'_l, l \mapsto \{\Sigma_l \mathrm{Build}_\varsigma(\Sigma_l, \varsigma), R\}) \sqsupseteq \Phi'_l$ and $(\Phi'_r, l \mapsto \{\Sigma_r, R[\varsigma]\}) \sqsupseteq \Phi'_r$.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel \{\varsigma, R\}\ \texttt{memo}\ a\ \texttt{in}\ S \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel R[\varsigma]\ \texttt{memo}\ a\ \texttt{in}\ S \parallel K_r \rangle\!\rangle$, by the closure properties of $(\simeq)$.

Case deref:

$$\Gamma \vdash V \; \texttt{memo} \; a \; \texttt{in} \; C[a] = V \; \texttt{memo} \; a \; \texttt{in} \; C[V] : \tau$$

Note that we take $C[a] = C[a]$ and $C[V] = C[V]$ to be computations and $\tau = \tau$; the proof will follow similarly for shared computations. From the equality, we know both that $\Gamma \vdash V \; \texttt{memo} \; a \; \texttt{in} \; C[a] : \tau$ and $\Gamma \vdash V \; \texttt{memo} \; a \; \texttt{in} \; C[V] : \tau$. By inversion on these, we may conclude further that $\Gamma, a{:}\sigma \vdash C[a] : \tau$ and $\Gamma, a{:}\sigma \vdash C[V] : \tau$. By further inversion, we know $(\Gamma, a{:}\sigma)\Gamma' \vdash a : \sigma$ and $(\Gamma, a{:}\sigma)\Gamma' \vdash V : \sigma$ where C binds the extended environment Γ'. Considering some arbitrary $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, we must show $((\Phi_l, \Sigma_l, V \; \texttt{memo} \; a \; \texttt{in} \; C[a]), (\Phi_r, \Sigma_r, V \; \texttt{memo} \; a \; \texttt{in} \; C[V])) \in \mathit{Comp}[\![\tau]\!]$. By Proposition 7.8, this is to show that the pair is in $\mathit{Stack}[\![\tau]\!]^{\top\!\top}$. Thus, consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left there is the transition,

$$\begin{aligned} \langle\!\langle \Phi'_l \parallel \Sigma_l \parallel V \; \texttt{memo} \; a \; \texttt{in} \; C[a] \parallel K_l \rangle\!\rangle \quad &\longmapsto \\ \langle\!\langle \Phi'_l, l_a \mapsto \mathrm{Build}_a(\Sigma_l, V) \parallel \Sigma_l, l_a/a \parallel C[a] \parallel K_l \rangle\!\rangle & \end{aligned}$$

And similarly on the right.

$$\begin{aligned} &(((\Phi'_l, l_a \mapsto \mathrm{Build}_a(\Sigma_l, V)), (\Sigma_l, l_a/a), C[a]) \\ &, ((\Phi'_r, l_a \mapsto \mathrm{Build}_a(\Sigma_r, V)), (\Sigma_r, l_a/a)), C[V]) \in \mathit{Comp}[\![\sigma]\!], \end{aligned}$$

by Lemma 7.3.

$$\begin{aligned} &(((\Phi'_l, l_a \mapsto \mathrm{Build}_a(\Sigma_l, V)), K_l) \\ &, ((\Phi'_r, l_a \mapsto \mathrm{Build}_a(\Sigma_r, V)), K_r)) \in \mathit{Stack}[\![\tau]\!], \end{aligned}$$

by closure under future heaps. And by Proposition 7.8, we know this pair is in $\mathit{Comp}[\![\tau]\!]^{\perp\!\!\perp}$.

$\langle\!\langle \Phi'_l, l_a \mapsto \mathrm{Build}_a(\Sigma_l, V) \parallel \Sigma_l, l_a/a \parallel C[a] \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r, l_a \mapsto \mathrm{Build}_a(\Sigma_r, V) \parallel \Sigma_r, l_a/a \parallel C[V] \parallel K_r \rangle\!\rangle$, by the combination of the expressions and stacks above from the property of $\mathit{Comp}[\![\tau]\!]^{\perp\!\!\perp}$.

Case GC:

$$\Gamma \vdash R \,\texttt{memo}\, a \,\texttt{in}\, P = P : \tau$$

Note that we take $P = M$ and $\tau = \tau$; the proof will follow similarly for shared computations. From the equality, we know that $\Gamma \vdash M : \tau$. Considering some arbitrary environments $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$, we must show that $((\Phi_l, \Sigma_l, R \,\texttt{memo}\, a \,\texttt{in}\, M), (\Phi_r, \Sigma_r, M)) \in \mathit{Comp}[\![\tau]\!]$. It is enough to show that the pair is in $\mathit{Stack}[\![\tau]\!]^{\top}$ by Proposition 7.8. Thus, further consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi'_l, K_l), (\Phi'_r, K_r)) \in \mathit{Stack}[\![\tau]\!]$:

On the left side, there is the transition:

$$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R \,\texttt{memo}\, a \,\texttt{in}\, M \parallel K_l \rangle\!\rangle \quad \longmapsto$$
$$\langle\!\langle \Phi'_l, l_a \mapsto \mathrm{Build}_a(\Sigma_l, R) \parallel \Sigma_l, l_a/a \parallel M \parallel K_l \rangle\!\rangle$$

$(((\Phi'_l, l_a \mapsto \mathrm{Build}_a(\Sigma_l, R)), \Sigma_l, l_a/a, M), (\Phi'_r, \Sigma_r, M)) \in \mathit{Comp}[\![\tau]\!]$ by Lemma 7.4. Moreover, it follows by Proposition 7.8, this pair is in $\mathit{Stack}[\![\tau]\!]^{\top}$.

$\langle\!\langle \Phi'_l, l_a \mapsto \mathrm{Build}_a(\Sigma_l, R) \parallel \Sigma_l, l_a/a \parallel M \parallel K_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel M \parallel K_r \rangle\!\rangle$, by the property of above related expression in the heaps $\Phi'_l, l_a \mapsto \mathrm{Build}_a(\Sigma_l, R) \sqsupseteq \Phi'_l$ and $\Phi'_r \sqsupseteq \Phi'_r$ with the assumed related stacks (note closure under future worlds).

Case name:

$$\Gamma \vdash R = R \;\mathtt{memo}\; a \;\mathtt{in}\; a : \tau$$

We must prove that $((\Phi_l, \Sigma_r, R), (\Phi_r, \Sigma_r, R \;\mathtt{memo}\; a \;\mathtt{in}\; a)) \in \mathit{Shared}[\![\tau]\!]$ for an arbitrary $((\Phi_l, \Sigma_l), (\Phi_r, \Sigma_r)) \in \mathit{Env}[\![\Gamma]\!]$. By definition, this is to show that the pair is in the set $\mathit{VStack}[\![\tau]\!]^{\top\!\top}$. Thus, we further consider some $\Phi'_l \sqsupseteq \Phi_l$, $\Phi'_r \sqsupseteq \Phi_r$, and $((\Phi_l, \mathbb{K}_l), (\Phi_r, \mathbb{K}_r)) \in \mathit{VStack}[\![\tau]\!]$:

On the right side, there is the transition sequence:

$$\begin{aligned} \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel R \;\mathtt{memo}\; a \;\mathtt{in}\; a \parallel \mathbb{K}_r \rangle\!\rangle &\longmapsto \\ \langle\!\langle \Phi'_r, l_a \mapsto \{\Sigma_r, R\} \parallel \Sigma_r, l_a/a \parallel a \parallel \mathbb{K}_r \rangle\!\rangle &\longmapsto \\ \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel R \parallel (\varepsilon, l_a) \cdot \mathbb{K}_r \rangle\!\rangle & \end{aligned}$$

$\Gamma \vdash R : \tau$ follows from the equality. And thus, $\Gamma \vDash R \approx R : \tau$, by the reflexivity of semantic equivalence. $((\Phi'_l, \Sigma_l, R), (\Phi'_r, \Sigma_r, R)) \in \mathit{Shared}[\![\tau]\!]$, by the above semantic equality with the related environments.

$((\Phi'_l, \mathbb{K}_l), (\Phi'_r, \mathbb{K}_r)) \in \mathit{Stack}[\![\tau]\!]$, since $\mathit{VStack}[\![\tau]\!]$ is include in $\mathit{Stack}[\![\tau]\!]$ and closure over accessible heaps.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R \parallel \mathbb{K}_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel R \parallel \mathbb{K}_r \rangle\!\rangle$, by the property of the related stacks with the heaps $\Phi'_l \sqsupseteq \Phi'_l$, $\Phi'_r \sqsupseteq \Phi'_r$, and the related expressions above.

$\langle\!\langle \Phi'_r \parallel \Sigma_r \parallel R \parallel \mathbb{K}_r \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel R \parallel (\varepsilon, l_a) \cdot \mathbb{K}_r \rangle\!\rangle$ by the last closure property of $(\simeq)$ for shared expressions.

$\langle\!\langle \Phi'_l \parallel \Sigma_l \parallel R \parallel \mathbb{K}_l \rangle\!\rangle \simeq \langle\!\langle \Phi'_r \parallel \Sigma_r \parallel R \;\mathtt{memo}\; a \;\mathtt{in}\; a \parallel \mathbb{K}_r \rangle\!\rangle$, by the closure properties of $(\simeq)$.

Cases *reflexivity*, *symmetry*, *transitivity*, and *compatibility* follow by their inductive hypotheses and the properties of $(\approx)$.

$\square$

7.3 Soundness of CBPVS

Theorem 7.1 (Soudness). *If* $\Gamma \vdash A = B : \tau$, *then* $A \simeq B$.

Proof. By Lemma 7.6, we know that $\Gamma \vDash A \approx B : \tau$. We know that $(\approx)$ is compatible; and thus with a closing context C, we have $\vDash C[A] \approx C[B] : F\ B$. With the related empty environments, we have that $((\varepsilon, \varepsilon, C[A]), (\varepsilon, \varepsilon, C[B])) \in \mathit{Comp}[\![F\ B]\!]$. By Proposition 7.8, we know that $\langle\!\langle \varepsilon \parallel \varepsilon \parallel C[M] \parallel \star \rangle\!\rangle \simeq \langle\!\langle \varepsilon \parallel \varepsilon \parallel C[N] \parallel \star \rangle\!\rangle$ if we can show that $((\Phi_l, \star), (\Phi_r, \star)) \in \mathit{Stack}[\![F\ B]\!]$ for any $\Phi_l \sqsupseteq \varepsilon$ and $\Phi_r \sqsupseteq \varepsilon$. This is true, since trivial to show termination considering an arbitrary $((\Phi_l, \Sigma_l, \texttt{ret b}), (\Phi_r, \Sigma_r, \texttt{ret b})) \in \mathit{Intro}[\![F\ B]\!]$. $\square$

Corollary 7.2. *If* $\vdash M = \texttt{ret b} : F\ B$, *then* $\mathrm{EvalS}(M) = \texttt{b}$.

7.4 Adequacy of Closure Conversions

A question left to answer, especially in a language for which closure conversion is novel, is whether or not the transformations we have specified have an effect on the machine. More specifically, are closures still left unspecified at runtime if we have done closure conversion?

As mentioned earlier, any rule in the machine that uses the build function to construct machine data may need to build its own closure if it was not specified. For instance:

$$\mathrm{Build}_V(\Sigma, \{\varsigma, \texttt{force} \to M\}) = \{\Sigma\, \mathrm{Build}_\varsigma(\Sigma, \varsigma), \texttt{force} \to M\}$$

The environment Σ is completely captured in the closure; it has only the flat structure that the machine uses for its environment and may contain more variables than needed for evaluating M later. If our closure conversion were adequate, then the building machine values ought to be equal to a restricted form of substitution:

$$\mathrm{Build}^{\mathrm{cl}}_{V}(\Sigma, \{\varsigma, \mathtt{force} \longrightarrow M\}) = \{\mathrm{Build}_{\varsigma}(\Sigma, \varsigma), \mathtt{force} \longrightarrow M\}$$

Now building machine values for closures only looks up the variables in the environment specified in the syntax. For the Build_V and Build_a, we can construct similar rules for their closure cases. Since it does not go into the body of the closure (as substitution would), we may generate a fixed code sequence for it.

To show that a conversion is adequate, we construct a new abstract machine, denoted $(\longmapsto_{\mathrm{CC}})$, that uses $\mathrm{Build}^{\mathrm{cl}}$ instead of Build. If a program has been closure converted, then it should be able to run on this machine and produce the same result as the larger machine that creates closures dynamically.

Definition 7.6 (Closure Converted CBPVS Machine Evaluator)**.**

$\mathrm{EvalS}_{\mathrm{CC}}(P) = \mathtt{b}$ *where* $\langle\!\langle \varepsilon \parallel \varepsilon \parallel P \parallel \star \rangle\!\rangle \longmapsto^{*}_{\mathrm{CC}} \langle\!\langle \Phi \parallel \Sigma \parallel \mathtt{ret\ b} \parallel \star \rangle\!\rangle$.

Theorem 7.2 (Adequacy)**.** *If P is a well-typed expression in* CC*-normal form, then* $\mathrm{EvalS}(P) = \mathrm{EvalS}_{\mathrm{CC}}(P)$.

Proof. Because of garbage collection (Lemma 7.4), for any closure we build in the machine, we can show that it is related to using the closed builder. For instance, consider some subexpression of P where $V = \{\varsigma, \mathtt{force} \longrightarrow M\}$ in CC-normal form. We know $\mathrm{Build}_V(\Sigma, V) = \{\Sigma\, \mathrm{Build}_{\varsigma}(\Sigma, \varsigma), \mathtt{force} \longrightarrow M\}$ by definition for an arbitrary Σ covering the free variables of V. And by definition, we also have $\mathrm{Build}^{\mathrm{cl}}_{V}(\Sigma, V) = \{\mathrm{Build}_{\varsigma}(\Sigma, \varsigma), \mathtt{force} \longrightarrow M\}$ for the closed version. By the definition of CC-normal form, M only depends on $\mathrm{Build}_{\varsigma}(\Sigma, \varsigma)$. Therefore, by repeated application of the garbage collection lemma, we have $((\Phi, \mathrm{Build}_V(\Sigma, V)), (\Phi, \mathrm{Build}^{\mathrm{cl}}_{V}(\Sigma, V))) \in \mathit{Val}[\![U\ \tau]\!]$. By the compatibility lemmas, we can do this for each closure converted sub-part of P. This with the knowledge that our evaluators run P in the empty environment and stack (which are trivial related) allows us to conclude. $\square$

There is still a sense in which abstract closures are less adequate than the canonical closure conversion. Whereas our approach keeps the contexts that consume closures (*i.e.* the application case for call-by-value closure conversion), a full closure conversion in the application case generates code for entering a function. Thus, a machine that accepts our closures need not have special rules for capturing environment—they are built the same as data—but will need special rules for closure entry which will instantiate the environment captured. This instantiation amounts to a pattern match followed by a jump; the canonical closure conversion is fine grained enough to detach these two operations, but at the expense of being a global transformation.

CHAPTER VIII

DISCUSSION

8.1 Related Work

Abstract closures have been used in the past for reasoning and optimization. Hannan [21] used abstract closures in an IL to implement the optimizations of Wand and Steckler [56] which reduce the variables in a closure. Minamide *et al.* [34] used them as an intermediate step in their typed closure conversion. These two works use big-step semantics and global transformations. The work most similar to ours is that of Bowman and Ahmed [11] because they give a language with local rewriting rules instead. For them, abstract closures were necessary for their main goal: proving the correctness of a closure conversion for the Calculus of Construction. Unlike their theory, we did not need to give special η laws for abstract closures, only β laws. Our work can be seen as promoting abstract closures further by considering their use in an optimizing compiler's IL. Specifically, we treat the process of closure conversion itself as a rewriting theory capable of being integrated into the optimization passes.

Explicit substitution calculi have a similar goal to ours: to close the gap between an equational theory and a practical implementation. Indeed, after adding our abstract closures, we arrived at a calculus that contains explicit substitutions like that of Abadi *et al.* [1] and that of the later extension to sharing by Seaman and Iyer [47]. A major difference is that we restrict where environments—for them substitutions—can occur in an expression, whereas they allow environments in any expression. For us, they can only occur for computations being delayed to values or shared values and over shared computations bound in a memo-expression; these are directly informed by where closures are constructed in our abstract machines. Moreover, we still make use of a substitution function over the syntax of our language, instead of embedding the entire system in our equational theory. As a result, we can easily specify what it means for a term to be in a closure converted form. The notion of β reduction in Abadi *et al.* and their Krivine machine always construct closure objects. In our system, we can show that a closure converted term does not do this.

Like us, McDermott and Mycroft [32] extend CBPV with sharing. Their approach is to use computation variables of type $F\ \tau$ for sharing whereas we add another syntactic class for shared objects. In both cases, sharing required the addition of special binders to reference the shared computation as in call-by-need. Their motivation was not specifically focused on using CBPV as a compiler IL and thus their language falls short for us. First, we needed to show the soundness of our theory with respect to an abstract machine because we wanted to use the language for optimization. Second, their approach does not subsume the full equational theory of call-by-need. Since we were interested in these goals, our language takes many ideas from Beyond Polarity (BP) [16] instead. As ANF [18] can be seen as a focalized variant of the λ-calculus, our language can be seen as a focalized variant of BP; that is, we must give names to all intermediate computations. We pursued a focalized language because it eliminates syntactic differences between programs; and thus, it is effective in compilation.

Our approach to proving the soundness of our equational theories follows from $\top\top$-closure of [45]. It follows the standard application of the approach, but is extended for environment machines. The correctness of the sharing portion is more novel. We are aware of three other approaches to logical relations for lazy languages [35, 20, 16]. Miquey and Herbelin [35] is only interested in normalization and thus constructs a logical predicate, but makes use of similar multi-level orthogonality operations in order to build up notions of types; their positive characterization of function types will become a problem if they extend their predicate to relation and attempt to prove the soundness of extensional axioms. The relation of Hackett and Hutton [20] is over expressions in the language which imply evaluation in a machine, in a manner similar to us; though it is not clear whether the approach is strong enough to prove the soundness of the extensionality axioms or lifting rules from our equational theories. Their relation does consider two aspects of lazy evaluation that we do not: polymorphism and computational cost. Finally, Downen and Ariola in their BP language [16] prove the soundness of a sharing equational theory with similar axioms including extensionality; it differs in that our operational semantics is an environment machine whereas theirs is a standard reduction theory with substitutions. Our environment machine semantics is why we developed a Kripke logical relation since heaps are part of the related expressions. There is other work by Ahmed [2] on Kripke logical relations for languages with heaps in the operational semantics. The languages that she considers are those with mutable state. Call-by-need heaps are better behaved than those with type-safe mutable references and thus we have a simpler Kripke relationship; hers require step indexing.

8.2 Future Work

As future work, we wish to extend our equational theory of CBPVS to a reduction theory. We generalized Levy's CBPV axioms in order to make the theory more flexible for optimization and we do not know exactly which changes need to be made to the equations to get a reduction theory. Additionally, we want to implement some version of this approach to closure conversion within GHC's intermediate language, since it is organized around the small, local transformations [44] that inspired this paper. In so doing, we would also need to extend our theory of closures to handle polymorphism and mutual recursion. Both cases have already received special attention with regard to closure conversion by Minamide *et al.* [34] and Appel [5], respectively. Preliminary work by us with respect to recursion shows that fixpoints out to be added to $U\ \tau$ types so that they can form a recursive closure; this differs from Levy who adds fixpoints as computations.

8.3 Conclusion

This thesis started by examining the abstract machines used to implement different evaluation strategies. We saw that the strictness of a language influenced the closures that were necessary in abstract machines. We first filled a hole in the literature with respect to non-strict closure conversion as an approach to compilation. As a reflection of the machines, we described new closure conversions for these different evaluation strategies. To add flexibility and higher-level reasoning to the transformation, we describe a new approach to compiling closures for these different evaluation strategies that allows us to perform the transformation locally in the intermediate language as well as optimizations thereof. We show how to combine all of these into a single intermediate language based on call-by-push-value that has the strongest β and η laws for optimization and preserves the equational theories of call-by-name, call-by-value, and call-by-need. To validate our work, we developed a model of types over abstract machines that show that our new approach is both sound and sufficient to remove runtime constructed closures from the machine.

REFERENCES CITED

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 31–46, 1989.

[2] Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[3] Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 157–168, 2008.

[4] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 340–353. ACM, 2009.

[5] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[6] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 293–302, 1989.

[7] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1997.

[8] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 233–246, 1995.

[9] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 233–246, 1995.

[10] Lennart Augustsson. Compiling pattern matching. In *Proceedings Of a Conference on Functional Programming Languages and Computer Architecture*, pages 368–381, 1985.

[11] William J. Bowman and Amal Ahmed. Typed closure conversion for the calculus of constructions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 797–811, 2018.

[12] G. L. Burn, Simon L. Peyton Jones, and J. D. Robson. The spineless g-machine. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 244–258, 1988.

[13] Alonzo Church. *The calculi of λ-conversion*, volume 6. Princeton University Press, 1941.

[14] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392, 1972.

[15] Paul Downen and Zena M. Ariola. The duality of construction. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 249–269, 2014.

[16] Paul Downen and Zena M. Ariola. Beyond polarity: Towards a multi-discipline intermediate language with sharing. In *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*, pages 21:1–21:23, 2018.

[17] Paul Downen, Philip Johnson-Freyd, and Zena M. Ariola. Abstracting models of strong normalization for classical calculi. *J. Log. Algebraic Methods Program.*, 111:100512, 2020.

[18] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993.

[19] Sebastian Graf and Simon Peyton Jones. Selective lambda lifting. *CoRR*, abs/1910.11717, 2019.

[20] Jennifer Hackett and Graham Hutton. Parametric polymorphism and operational improvement. *Proc. ACM Program. Lang.*, 2(ICFP):68:1–68:24, 2018.

[21] John Hannan. Type systems for closure conversions. *The Workshop on Types for Program Analysis*, pages 64–83, 1995.

[22] Richard B. Kieburtz. The g-machine: A fast, graph-reduction evaluator. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, pages 400–413, 1985.

[23] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[24] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[25] John Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 144–154, 1993.

[26] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[27] Paul Blain Levy. *Call-by-push-value*. PhD thesis, Queen Mary University of London, UK, 2001.

[28] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, 1998.

[29] Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, pages 4–15, 2004.

[30] Phillip Mates, Jamie Perconti, and Amal Ahmed. Under control: Compositionally correct closure conversion with mutable state. In Ekaterina Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. ACM, 2019.

[31] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 482–494, 2017.

[32] Dylan McDermott and Alan Mycroft. Extended call-by-push-value: Reasoning about effectful programs and evaluation order. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, pages 235–262, 2019.

[33] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[34] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 271–283, 1996.

[35] Étienne Miquey and Hugo Herbelin. Realizability interpretation and normalization of typed call-by-need lambda-calculus with control. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10803 of *Lecture Notes in Computer Science*, pages 276–292, 2018.

[36] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 85–97, 1998.

[37] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. *LISP Symb. Comput.*, 7(1):57–82, 1994.

[38] Zoe Paraskevopoulou and Andrew W. Appel. Closure conversion is safe for space. *Proc. ACM Program. Lang.*, 3(ICFP):83:1–83:29, 2019.

[39] Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. Compositional optimizations for certicoq. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021.

[40] James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 128–148, 2014.

[41] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[42] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, pages 636–666, 1991.

[43] Simon L. Peyton Jones and Jon Salkild. The spineless tagless g-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 184–201, 1989.

[44] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for haskell. *Sci. Comput. Program.*, 32(1-3):3–47, 1998.

[45] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Math. Struct. Comput. Sci.*, 10(3):321–359, 2000.

[46] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[47] Jill Seaman and S. Purushothaman Iyer. An operational semantics of sharing in lazy evaluation. *Sci. Comput. Program.*, 27(3):289–322, 1996.

[48] Peter Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3):231–264, 1997.

[49] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, Orlando, Florida, USA, 27-29 June 1994*, pages 150–161, 1994.

[50] Zhong Shao and Andrew W. Appel. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst.*, 22(1):129–161, 2000.

[51] Guy L. Steele. Rabbit: A compiler for scheme. Master's thesis, Massachusetts Institute of Technology, 1978.

[52] Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. Strictly capturing non-strict closures. In *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2021, Virtual Event, Denmark, January 18-19, 2021*, pages 74–89. ACM, 2021.

[53] Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. Closure conversion in little pieces. In Santiago Escobar and Vasco T. Vasconcelos, editors, *International Symposium on Principles and Practice of Declarative Programming, PPDP 2023, Lisboa, Portugal, October 22-23, 2023*, pages 10:1–10:13. ACM, 2023.

[54] D. A. Turner. A new implementation technique for applicative languages. *Softw., Pract. Exper.*, 9(1):31–49, 1979.

[55] Philip Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 189–201, 2003.

[56] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 435–445, 1994.