Closure Conversion in Little Pieces

Zachary J. Sullivan
University of Oregon
zsulliva@cs.uoregon.edu

Paul Downen
University of Massachesetts, Lowell
paul@pauldownen.com

Zena M. Ariola
University of Oregon
ariola@cs.uoregon.edu

ABSTRACT

Closure conversion, an essential step in compiling functional programs, is traditionally presented as a global transformation from a language with higher-order functions to one without. Optimizing this transformation can be done at any of its three stages with various tradeoffs. After conversion, optimizations will be in the target language at the cost of a weaker equational theory. During conversion, optimizations may be embedded into the transformation itself at the cost of increasing its complexity and correctness. And before conversion, optimizations may be planned and anticipated in a specialized intermediate language (IL) where all the properties of the source program are still known, with the hope that they will survive the rest of the compilation process.

By building on the notion of abstract closures, we blur the distinctions between these three approaches to closure conversion and optimizations thereof, by combining all of their strengths and avoiding their weaknesses. Specifically, we develop an IL that includes closures alongside unclosed higher-order code, even inhabiting the same type. The IL comes equipped with an equational theory that is shown sound and complete with respect to an environment abstract machine. Thereby, a baseline closure conversion and common optimizations become provable equalities and thus are correct by construction. Moreover, the transformation and its correctness proof are broken down into little steps—as instances of the β and η axioms—instead of being expressed in terms of a recursive procedure.

Our proposed IL is based on call-by-push-value which we extend with sharing in order to capture closure conversion for both strict and lazy languages.

CCS CONCEPTS

• **Software and its engineering** → *Language features*; **Compilers**.

KEYWORDS

closures, closure conversion, compiler intermediate languages

ACM Reference Format:
Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. 2023. Closure Conversion in Little Pieces. In *International Symposium on Principles and Practice of Declarative Programming (PPDP 2023), October 22–23, 2023, Lisboa, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3610612.3610622

PPDP 2023, October 22–23, 2023, Lisboa, Portugal

ACM ISBN 979-8-4007-0812-1/23/10…$15.00
https://doi.org/10.1145/3610612.3610622

1 INTRODUCTION

In the compilation from a higher-order language to a first-order, low-level language like C, functions are converted into data structures containing an environment and a code pointer. Often, the source and target languages are different. However, it would be really convenient for closure conversion to be expressed in the *same* language. Specifically, we would like the following to hold:

Theorem 1.1. *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 [16] 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 [3, 19, 21, 26]. 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 [14] and unboxed types [20], 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 the canonical closure conversion [2, 16–19] does not make this goal easy. For example, we have that:

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

If we want Theorem 1.1 to be true, then we immediately run into a problem since the type has changed from a function to a pair. Therefore, to say that something is $\beta\eta$ equal to its closure converted form is false because we cannot apply a 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 [5, 10, 16]. 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.

To capture several closure conversions in a single language, we choose call-by-push-value (CBPV) [12] as our starting point, since it already subsumes the theories of call-by-name and call-by-value source languages. Moreover, it provides the strongest possible β and η laws for functions and data making it a good candidate for an IL. However, in practice, it is *call-by-need* that our IL needs to support since the call-by-name repetitions of identical computations make its algorithmic complexity unacceptable in practice. Previous work [27] has shown that closure conversion of shared computations requires special consideration; thus, we extend CBPV with both closures and sharing to study their interaction. To our knowledge, CBPV is also a novel language for closure conversion. So to justify where we place abstract closures in such a language, we develop environment abstract machines, like the SECD [7] and Krivine [11] machines, which must capture closures as part of its runtime. It is with respect to these machines that the soundness of our abstract closures is proved.

This paper presents the following contributions:

- We extend CBPV with the notion of *abstract closures* and define an equational theory that embraces both eager and lazy languages.
- We prove soundness and completeness of the extended CBPV (including sharing) with respect to an abstract machine, using a logical relation argument.
- We define closure conversion not as a *global* cross-language transformation but in terms of *little pieces* that correspond to provable equalities. Thus, closure conversion is correct by construction.
- We elevate closure conversion to be on par with other optimizations. In other words, it is done within the IL itself. Thereby, closure conversion optimizations can be expressed by standard IL transformations.
- We show how a closed converted program can be executed on an abstract machine that does not construct closures at runtime. Thus providing evidence that indeed all the necessary machinery for higher-order functions can be handled at compile time.

Section 2 describes how we arrive at using and strengthening abstract closures to solve problems that arise in the canonical closure conversion. Section 3 presents our syntactic theory of closures as an extension to CBPV and defines naïve notion of closure conversion. Section 4 shows how common closure optimizations are derivable in our IL. Section 5 extends our work to the closures required for sharing languages; along the way, we present an approach to sharing in CBPV. Section 6 presents environment abstract machines that serve as the operational semantics of our language. Section 7 establishes the correctness of the equational theory with respect to the machines. Section 8 proves that performing closure conversion removes the need to capture closures in the machines.

2 WHY ABSTRACT CLOSURES

If our goal is to promote reasoning about closures from a low-level or code generation phase of compilation to the IL's optimization passes, then the canonical closure conversion presents more problems than just being a global transformation. The transformation is defined for call-by-value languages by the following recursive function over expressions:

$$\begin{aligned} \mathrm{CC}(x) &= x \\ \mathrm{CC}(\lambda x.\, M) &= \langle \langle y_0, \ldots, y_n \rangle, \lambda \langle \langle y_0, \ldots, y_n \rangle, x \rangle .\, \mathrm{CC}(M) \rangle \\ &\quad \textbf{where } \{y_0, \ldots, y_n\} = \mathrm{FV}(M) - \{x\} \\ \mathrm{CC}(M\ N) &= \mathtt{case}\ \mathrm{CC}(M)\ \mathtt{of}\ \{\langle e, f \rangle \to f\ \langle e, \mathrm{CC}(N) \rangle\} \end{aligned}$$

A function is turned into a pair where the first component is some representation of the free variables of the function body and the second component is a version of that function which also knows how to re-instantiate that environment. An application is turned into a pattern match on a pair, thereafter applying the second component to a pair of the first component and the original argument.

Instead of a transformation between a different source and target language, in some works [2, 19] 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:

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

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 example, if M is $\lambda z.\, x$, we will need to prove the following:

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

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):

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

A problem also arises in preserving η-laws, 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:

$$\langle \langle z \rangle, \lambda \langle \langle z \rangle, x \rangle .\, \mathtt{case}\ z\ \mathtt{of}\ \{\langle e, f \rangle \to f\ \langle e, x \rangle\} \rangle \stackrel{?}{=} z$$

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 logical relations, which capture the lost information,

to prove the correctness of closure conversion. Working directly in the syntax, abstract closures [5, 10, 16] solve all of the above problems. 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. Finally, 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.

3 CLOSURES IN CALL-BY-PUSH-VALUE

Being a new language for the transformation, our first challenge is discovering where CBPV should have closures. CBPV achieves its strong equational theory by separating the objects that have different β and η laws. There is a syntactic distinction between expressions that *are*, called values, and expressions that *do*, called computations. Only values are substitutable and only computations are β reducible. Previous work [5, 10, 16] constructs abstract closures only for functions. This is essential in strict languages since their functions are the only values containing delayed code—which is a β reducible expression—that may be bound to variables and passed to separate parts of the program, which may have a different local environment. In non-strict languages, the arguments of functions also require this treatment. In CBPV, only values have the potential to be bound to variables and passed to other parts of the program. The only time these expressions can contain unevaluated code is when a computation is delayed within a value. Thus, this is where we must add closures. Surprisingly and in contrast with much of the previous work, we do not need closures for functions. A function is a computation in CBPV, and therefore, is never bound to a variable.

3.1 Syntax and Typing Rules

The syntax, some syntactic sugar, and typing rules for CBPV with closures is shown in Figure 1. We write values in green and computations in orange. In the typing rules and elsewhere in this paper, we use juxtaposition, *e.g.* $\Gamma\Gamma'$, to denote combining the two together. Additionally, this paper makes use of program contexts, written as C, which are any expression with a single hole and evaluation contexts, written as E or F, that are a subset of program contexts.

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 conveys the idea that values *are*, and that they can be predicted based on their *type*. Moreover, we see that the syntax and typing rules restrict variables to only range over value types so that 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 to have a computation that returns a value of type τ, it will have to be shifted to the type $F\ \tau$. For example, a computation returning $\langle 4, 2\rangle$ will be written as $\mathtt{ret}\ \langle 4, 2\rangle$ 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.

$$
\begin{array}{rrll}
\tau, \sigma \in & \textit{Type} & ::= & \tau \mid \tau \\
\tau, \sigma \in & \textit{Value Type} & ::= & B \mid \tau \otimes \sigma \mid U\ \tau \\
\tau, \sigma \in & \textit{Comp. Type} & ::= & \tau\ \&\ \sigma \mid \tau \to \sigma \mid F\ \tau \\
\\
A, B, C \in & \textit{Expr.} & ::= & V \mid M \\
\varsigma \in & \textit{Env.} & ::= & \varepsilon \mid \varsigma, V/x \\
V, W \in & \textit{Value} & ::= & x \mid \mathsf{b} \mid \langle V, W\rangle \mid \{\varsigma, \mathtt{force} \to M\} \\
M, N \in & \textit{Comp.} & ::= & \mathtt{case}\ V\ \mathtt{of}\ \{\langle x, y\rangle \to M\} \\
 & & & \mid \{\mathtt{fst} \to M; \mathtt{snd} \to N\} \\
 & & & \mid M.\mathtt{fst} \mid M.\mathtt{snd} \mid \lambda x.\, M \mid M\ V \\
 & & & \mid \mathtt{ret}\ V \mid M\ \mathtt{to}\ x\ \mathtt{in}\ N \mid V.\mathtt{force}
\end{array}
$$

(a) Syntax

$$
\begin{array}{rcl}
\{\mathtt{force} \to M\} & = & \{\varepsilon, \mathtt{force} \to M\} \\
\mathtt{let}\ x\ \mathtt{be}\ V\ \mathtt{in}\ M & = & (\lambda x.\, M)\ V
\end{array}
$$

(b) Syntactic Sugar

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

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

$$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \rho}{\Gamma \vdash \{\mathtt{fst} \to M; \mathtt{snd} \to N\} : \tau\ \&\ \rho}\ \&_I$$

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

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

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

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

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

(c) Typing Rules

Figure 1: CBPV - Call-by-Push-Value with Closures

We use an unconventional syntax for values containing computations that wait for method calls, *i.e.* $\mathtt{thunk}\ M$ is written as the closure $\{\varepsilon, \mathtt{force} \to M\}$, to emphasize how delayed computations behave like objects; we will introduce more expressions of a similar kind later in this paper. The value now includes a local environment ς, which is in essence a delayed substitution. When that environment is empty, ε, we use the syntactic sugar $\{\mathtt{force} \to M\}$. Note that we refer to these objects as closures following the usage from previous work [5, 10, 16], but we have generalized them so that they do not need to close over the entire environment of M in rule U_I.

The environments are typed in such a way that they may freely depend on Γ, but not on previous bindings within. In this way, the substitutions in our IL are simultaneous, meaning each element is

$$
\begin{array}{rll}
(\lambda x.\, M)\ V & =_{\to} & M[V/x] \\
\{\mathtt{fst} \to M; \mathtt{snd} \to N\}.\mathtt{fst} & =_{\&1} & M \\
\{\mathtt{fst} \to M; \mathtt{snd} \to N\}.\mathtt{snd} & =_{\&2} & N \\
\mathtt{case}\ \langle V, W\rangle\ \mathtt{of}\ \{\langle x, y\rangle \to M\} & =_{\otimes} & M[V/x, W/y] \\
\{\varsigma, \mathtt{force} \to M\}.\mathtt{force} & =_{U} & M[\varsigma] \\
(\mathtt{ret}\ V)\ \mathtt{to}\ x\ \mathtt{in}\ M & =_{F} & M[V/x]
\end{array}
$$

(a) β-laws

$$
\begin{array}{rll}
\lambda x.\, M\ x & =_{\to} & M \\
\{\mathtt{fst} \to M.\mathtt{fst}; \mathtt{snd} \to M.\mathtt{snd}\} & =_{\&} & M \\
\mathtt{case}\ V\ \mathtt{of}\ \{\langle x, y\rangle \to M[\langle x, y\rangle/z]\} & =_{\otimes} & M[V/z] \\
\{\mathtt{force} \to V.\mathtt{force}\} & =_{U} & V \\
M\ \mathtt{to}\ x\ \mathtt{in}\ E[\mathtt{ret}\ x] & =_{F} & E[M]
\end{array}
$$

(b) η-laws

where $E \in \mathit{Eval.\ Cont.} ::= \square \mid E\ V \mid E.\mathtt{fst} \mid E.\mathtt{snd} \mid E\ \mathtt{to}\ x\ \mathtt{in}\ M$

Figure 2: CBPV Axioms

independent. When acting on a closure, the substitution is applied to both the closure's local environment and the free variables of the body:

$$\{\varsigma', \mathtt{force} \to M\}[\varsigma] \quad = \quad \{\varsigma'[\varsigma], \mathtt{force} \to M[\varsigma]\}$$

Applying one environment to another is defined as substituting on the values therein:

$$\varepsilon[\varsigma] = \varepsilon \qquad (\varsigma', V/x)[\varsigma] = \varsigma'[\varsigma], V[\varsigma]/x$$

3.2 Equational Theory

The equational theory is shown in Figure 2. For an expression to be equal to another, we require that both sides have the same type. For the most part, the axioms are identical to Levy [12] except for $F\ \tau$ and $U\ \tau$ types. For type $F\ \tau$, we have strengthened its η law to apply in any evaluation context. When that evaluation context is empty, this law coincides with Levy.

The β law for $U\ \tau$ has been replaced with one for closures wherein we merely perform the delayed substitution when the force method is called. The η law remains unchanged and applies *only* to force-expressions with an empty environment. Indeed, the more general η law, which we call $\eta_{U'}$, with a non-empty environment:

$$\{\varsigma, \mathtt{force} \to V.\mathtt{force}\} =_{\eta_{U'}} V[\varsigma]$$

can be derived as follows:

$$
\begin{array}{rll}
V[\varsigma] & =_{\eta_U} & \{\mathtt{force} \to V[\varsigma].\mathtt{force}\} \\
& =_{\text{subst.}} & \{\mathtt{force} \to (V.\mathtt{force})[\varsigma]\} \\
& =_{\beta_U} & \{\mathtt{force} \to \{\varsigma, \mathtt{force} \to V.\mathtt{force}\}.\mathtt{force}\} \\
& =_{\eta_U} & \{\varsigma, \mathtt{force} \to V.\mathtt{force}\}
\end{array}
$$

Remark 1. A set of laws that appear to be missing is the sequencing laws of Levy [12], which lift to-expressions out of computations of various types. We excluded these laws because they are derivable from the strengthened η law for $F\ \tau$ types. First, a useful law for lifting to-expression out of evaluation contexts:

$$E[M\ \mathtt{to}\ x\ \mathtt{in}\ N] =_{\text{lift}} M\ \mathtt{to}\ x\ \mathtt{in}\ E[N]$$

is derived as follows:

$$
\begin{array}{rll}
E[M\ \mathtt{to}\ x\ \mathtt{in}\ N] & =_{\eta_F} & M\ \mathtt{to}\ y\ \mathtt{in}\ E[\mathtt{ret}\ y\ \mathtt{to}\ x\ \mathtt{in}\ N] \\
& =_{\beta_F} & M\ \mathtt{to}\ y\ \mathtt{in}\ E[N[y/x]] \\
& =_{\alpha} & M\ \mathtt{to}\ x\ \mathtt{in}\ E[N].
\end{array}
$$

Thereafter, the three sequencing laws for $F\ \tau$, $(\to)$, and $(\&)$ types are easily derivable. For instance, the sequencing law for functions:

$$M\ \mathtt{to}\ x\ \mathtt{in}\ \lambda y.\, N = \lambda y.\, M\ \mathtt{to}\ x\ \mathtt{in}\ N$$

is proved by the following:

$$
\begin{array}{rll}
M\ \mathtt{to}\ x\ \mathtt{in}\ \lambda y.\, N & =_{\eta_{\to}} & \lambda y.\, (M\ \mathtt{to}\ x\ \mathtt{in}\ \lambda y.\, N)\ y \\
& =_{\text{lift}} & \lambda y.\, M\ \mathtt{to}\ x\ \mathtt{in}\ (\lambda y.\, N)\ y \\
& =_{\beta_{\to}} & \lambda y.\, M\ \mathtt{to}\ x\ \mathtt{in}\ N.
\end{array}
$$

3.3 Deriving a Closure Conversion Transformation

As an example of Theorem 1.1, we can now construct a naïve flat closure conversion transformation syntactically. We do this by deriving a simple rewriting rule that adds one free variable at a time.

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

For each application of the rule, the local environment ς grows by an identity substitution x/x. 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.* $\{(3/x, y/y), \mathtt{force} \to M\}$, the delayed substitution is preserved.

We show next that the rewrite rule is derivable, where we let the environment ς be $V_0/x_0, \ldots, V_n/x_n$ and y be a free variable in $\mathrm{FV}(M) - \mathrm{Dom}(\varsigma)$:

$$
\begin{array}{r}
\{\varsigma, \mathtt{force} \to M\} =_{\text{subst.}} \\
\{\varsigma, \mathtt{force} \to M[x_0/x_0, \ldots, x_n/x_n, y/y]\} =_{\beta_U} \\
\{\varsigma, \mathtt{force} \to \{(x_0/x_0, \ldots, x_n/x_n, y/y), \mathtt{force} \to M\}.\mathtt{force}\} =_{\eta_{U'}} \\
\{(x_0/x_0, \ldots, x_n/x_n, y/y), \mathtt{force} \to M\}[\varsigma] =_{\text{subst.}} \\
\{(\varsigma, y/y), \mathtt{force} \to M\}
\end{array}
$$

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. This step-by-step approach to closures is why our abstract closures can be partial, in constrast with previous work.

Definition 3.1 (Naïve Closure Conversion).
$\mathrm{NCC}(A) = B$ iff $A \longrightarrow_{\mathrm{CC}}^{*} B$ and B is in CC-normal form.

4 INCREMENTAL OPTIMIZATION OF CLOSURES

The above closure conversion is a flat closure representation; 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 [9, 16, 19, 25], 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.

4.1 Choosing an Environment Representation

Minamide *et al.* [16] combine the environments of different closures to save space when allocating a closure, at the cost of possible space-leaks [25] and extended lookup times for closure variables. To do this with abstract closures, we need an easy way to combine subparts 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 $\{(\varepsilon, V \mathbin{/\!\!/} \langle x, \langle y, z\rangle\rangle), \mathtt{force} \to M\}$ desugars into the following:

$$\begin{array}{l}\{(\varepsilon, V/v), \mathtt{force} \to \\ \quad \mathtt{case}\ v\ \mathtt{of}\ \{\langle x, v'\rangle \to \\ \quad\quad \mathtt{case}\ v'\ \mathtt{of}\ \{\langle y, z\rangle \to M\}\}\}\end{array}$$

Using this sugar, the environment sharing of the example from Shao and Appel [25] is a derivable equality in our language. In the following program, we have already run our naïve closure conversion:

$$\begin{array}{l}\mathtt{let}\ g\ \mathtt{be}\ \{(\varepsilon, g/g, v/v, w/w, x/x, y/y, z/z), \mathtt{force} \to A\}\ \mathtt{in} \\ \mathtt{let}\ h\ \mathtt{be}\ \{(\varepsilon, h/h, u/u, w/w, x/x, y/y, z/z), \mathtt{force} \to B\}\ \mathtt{in} \\ \mathtt{let}\ j\ \mathtt{be}\ \{(\varepsilon, i/i, w/w, x/x, y/y, z/z), \mathtt{force} \to C\}\ \mathtt{in} \\ \quad D \\ \quad = \\ \mathtt{let}\ e\ \mathtt{be}\ \langle w, x, y, z\rangle\ \mathtt{in} \\ \mathtt{let}\ g\ \mathtt{be}\ \{(\varepsilon, g/g, v/v, e \mathbin{/\!\!/} \langle w, x, y, z\rangle), \mathtt{force} \to A\}\ \mathtt{in} \\ \mathtt{let}\ h\ \mathtt{be}\ \{(\varepsilon, h/h, u/u, e \mathbin{/\!\!/} \langle w, x, y, z\rangle), \mathtt{force} \to B\}\ \mathtt{in} \\ \mathtt{let}\ j\ \mathtt{be}\ \{(\varepsilon, i/i, e \mathbin{/\!\!/} \langle w, x, y, z\rangle), \mathtt{force} \to C\}\ \mathtt{in} \\ \quad D\end{array}$$

The first instance of 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. In a runtime system where products are passed by reference, the resulting program will be more space efficient.

This is an example of taking naïve closure conversion as a starting point and transforming our code further to optimize subprograms. So not only does $M = \mathrm{NCC}(M)$, but also $M = (\mathrm{EnvShare} \circ \mathrm{NCC})(M)$. Moreover, this transformation preserves the CC-normal form property of $\mathrm{NCC}(M)$.

4.2 Choosing Environment Passing Technique

Another optimization presented for closures is lambda-lifting [9, 19]. 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, in other words "known functions". For instance, consider the following example where the closure bound to x is transformed and whose body M has the free variable n:

$$\begin{array}{rl}\mathtt{let}\ x\ \mathtt{be}\ \{(\varsigma, n/n), \mathtt{force} \to M\}\ \mathtt{in} \ldots x.\mathtt{force} \ldots & =_{\beta_\to} \\ \ldots \{(\varsigma, n/n), \mathtt{force} \to M\}.\mathtt{force} \ldots & =_{\beta_U} \\ \ldots M[\varsigma, n/n] \ldots & =_{\text{subst.}} \\ \ldots M[\varsigma][n/n] \ldots & =_{\beta_\to} \\ \ldots (\lambda n.\, M)[\varsigma]\ n \ldots & =_{\beta_U} \\ \ldots \{\varsigma, \mathtt{force} \to \lambda n.\, M\}.\mathtt{force}\ n \ldots & =_{\beta_\to} \\ \mathtt{let}\ x\ \mathtt{be}\ \{\varsigma, \mathtt{force} \to \lambda n.\, M\}\ \mathtt{in} \ldots x.\mathtt{force}\ n \ldots & \end{array}$$

To avoid passing n within the closure's environment, which may require more allocation, the body of the closure is converted to a function and n is placed as an argument where the closure is entered. 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, 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.

5 CLOSURES FOR SHARING

Since CBPV subsumes call-by-name and call-by-value, this closure conversion in CBPV is now enough to support the transformation of those source languages, but it is not enough to support languages with memoization like Haskell. For that, we need a CBPV that supports sharing before we can consider closures. This section first specifies a call-by-need calculus, extends CBPV with the necessary features for sharing in a manner that preserves the equational theory of call-by-need, and then specifies its closures. Our new language, which we refer to as CBPVS for short ("S" for sharing), is shown in Figure 4.

In a language with sharing, the problems of the canonical closure conversion are exacerbated from those mentioned in Section 2. For function arguments, closures are necessary because the arguments are delayed until their variable is demanded within the function body. And since sharing requires that these closures are evaluated at most once, their evaluation must be followed by an update. Therefore, the target of a sharing closure conversion ends up as a strict language with mutation [27], a language that is much harder to reason about than call-by-need. Using abstract closures instead, in the same manner as we just did for CBPV, means that we are able to remain in an equational theory more similar to call-by-need.

5.1 A Call-by-Need Calculus

In work by Ariola *et al.* [4] and Maraist *et al.* [13], 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.[1]. This is

[1] Ariola *et al.* [4] show that the let-expression is not necessary since sharing can be captured by not reducing the function application.

apparent in the axioms given in Figure 3 wherein the β rule for functions creates a binding and the deref-rule for let-expressions performs substitutions incrementally only when the bound expression is a value. 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 expressions. Note that unlike CBPV which has a syntactic distinction between values and computation that is reflected in the typing rules, this calculus (CBNeed) has no such distinction and thus uses the standard typing rules of the λ-calculus.

The η axioms for CBNeed functions and product types must be more restrictive than CBPV in order to preserve sharing. The law for functions requires that the function be a value; otherwise, η would change it from a value to a non-value thereby modifying its sharing property. The law for products only applies when the pair is reconstructed within an evaluation context. Without that restriction, M will be forced on the left-hand side, but not necessarily on the right.

5.2 CBPV with Sharing and Closures

The first place to start when extending our intermediate language with sharing is to, like in CBNeed, add a binding construct that gives names to the computation that we wish to share and only evaluates it when needed. Such a binding looks like M `memo` a `in` N. Of course, we would like to save computations that return values: imagine having M be the program $1 + 2$ `to` x `in` `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 $(\lambda x.\lambda y.\,\texttt{ret}\ x)\ 42$ 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.* `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, but then we would have to sacrifice our strong η law for CBPV function types for the weaker one found in call-by-need. We instead 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 (Figure 4). 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 `val` 42 and give the type $\hat{F}\ \mathbb{N}$. The opposite direction is true as well. We will want to embed shared computations within a data structure; this we do with the shift `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, $\{\texttt{fst} \to \{\texttt{eval} \to R\}; \texttt{fst} \to \{\texttt{eval} \to S\}\}$. 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.

Regarding closures, there are now more places where unevaluated code with free variables may be substituted to other parts of the program in an environment machine. This is a result of adding new kinds of expressions to the language that can be bound to variables. Although they are not substituted with unevaluated code, the CBNeed evaluation context `let` x `be` E `in` $F[x]$ suggests that 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 [24]. So in CBPVS, we must have a closure for the memo-expression $\{\varsigma, R\}$ `memo` a `in` P. Like with force-expressions, the enter-expression, which delays a computation within a shared computation, may be substituted and thus will need to be a closure. As with CBPV with closures, there is syntactic sugar for when the environment of a closure is empty: R `memo` a `in` P and $\{\texttt{enter} \to M\}$.

The additional typing rules for CBPVS are given in Figure 5; the rest are the same as those for CBPV, with the difference that the typing context Γ is now composed of both value and shared variables. For sharing, we had to break the convention that CBPV only substitutes values. 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. The typing rules for the new closure forms follow the same pattern as those that we saw before.

5.3 Equational Theory

The axioms for CBPVS are given in Figure 6. 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 CBNeed. 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. 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 CBNeed.

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.

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

where $V, W \in$ *Value* $::= x \mid \mathsf{b} \mid \lambda x.\, M \mid \langle V, W\rangle$

$E, F \in$ *EvalCxt* $::= \square \mid E\ N \mid \mathtt{case}\ E\ \mathtt{of}\ \{\langle x, y\rangle \to N\} \mid \mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ E \mid \mathtt{let}\ x\ \mathtt{be}\ E\ \mathtt{in}\ F[x]$

Figure 3: CBNeed - Call-by-Need Axioms

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

(a) Syntax

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

(b) Syntactic Sugar

Figure 4: CBPVS - CBPV with Sharing and Closures

This lack of a restriction in the axiom, despite containing a shared expression, is possible because of the syntactic restriction to shared values for $\mathtt{box}\ V$. Without the syntactic restriction, a program like the following will duplicate work:

$$\mathtt{case}\ \langle 42, \mathtt{box}\ R\rangle\ \mathtt{of}\ \{\langle x, y\rangle \to \ldots y \ldots y \ldots\} \longrightarrow_\beta \ldots \mathtt{box}\ R \ldots \mathtt{box}\ R \ldots$$

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:

$$R\ \mathtt{memo}\ a\ \mathtt{in}\ \mathtt{case}\ \langle 42, \mathtt{box}\ a\rangle\ \mathtt{of}\ \{\langle x, y\rangle \to \ldots x \ldots x \ldots\} \longrightarrow_\beta R\ \mathtt{memo}\ a\ \mathtt{in}\ \ldots \mathtt{box}\ a \ldots \mathtt{box}\ a \ldots$$

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 CBNeed 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

$$\frac{\Gamma \vdash V : \sigma \otimes \rho \quad \Gamma, x{:}\sigma, y{:}\rho \vdash P : \tau}{\Gamma \vdash \mathtt{case}\ V\ \mathtt{of}\ \{\langle x, y\rangle \to P\} : \tau}\otimes_E$$

$$\frac{\Gamma \vdash M : F\ \sigma \quad \Gamma, x{:}\sigma \vdash P : \tau}{\Gamma \vdash M\ \mathtt{to}\ x\ \mathtt{in}\ P : \tau} F_E$$

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

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

$$\frac{\Gamma \vdash \varsigma : \Gamma' \quad \Gamma\Gamma' \vdash M : \tau}{\Gamma \vdash \{\varsigma, \mathtt{enter} \to M\} : \check{U}\ \tau}\check{U}_I \qquad \frac{\Gamma \vdash R : \check{U}\ \tau}{\Gamma \vdash R.\mathtt{enter} : \tau}\check{U}_E$$

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

$$\frac{\Gamma \vdash R : \tau}{\Gamma \vdash \{\mathtt{eval} \to R\} : \tilde{F}\ \tau}\tilde{F}_I$$

$$\frac{\Gamma \vdash M : \tilde{F}\ \tau}{\Gamma \vdash M.\mathtt{eval} : \tau}\tilde{F}_E \qquad \frac{\Gamma \vdash \varsigma : \Gamma' \quad \Gamma \vdash V : \tau}{\Gamma \vdash (\varsigma, V/a) : (\Gamma', a{:}\tau)}\Gamma_{I}2$$

Figure 5: Additional and Extended CBPVS Typing Rules

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

(a) β-laws

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

(b) η-laws

$$
\begin{aligned}
E[R\ \mathtt{memo}\ a\ \mathtt{in}\ P] &=_{\kappa} R\ \mathtt{memo}\ a\ \mathtt{in}\ E[P] \\
(R\ \mathtt{memo}\ b\ \mathtt{in}\ S)\ \mathtt{memo}\ a\ \mathtt{in}\ P &=_{\chi} R\ \mathtt{memo}\ b\ \mathtt{in}\ (S\ \mathtt{memo}\ a\ \mathtt{in}\ P) \\
\{\varsigma, R\}\ \mathtt{memo}\ a\ \mathtt{in}\ P &=_{\mathrm{cl}} R[\varsigma]\ \mathtt{memo}\ a\ \mathtt{in}\ P \\
V\ \mathtt{memo}\ a\ \mathtt{in}\ C[a] &=_{\mathrm{deref}} V\ \mathtt{memo}\ a\ \mathtt{in}\ C[V] \\
R\ \mathtt{memo}\ a\ \mathtt{in}\ P &=_{\mathrm{GC}} P \\
R &=_{\mathrm{name}} R\ \mathtt{memo}\ a\ \mathtt{in}\ a
\end{aligned}
$$

(c) Other laws

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

Figure 6: CBPVS Axioms

$\hat{F}$ 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.

Considering the new closures, the β and η laws for enter-closures work like those of force-closures: the delayed environment is substituted when their method is called. On the other hand, the cl-law for the memo-closures allows the closure to be entered at any time. In the abstract machine, these are only entered when their bound variables occur in an evaluation context.

5.4 Compiling CBNeed to CBPVS

Figure 7 shows how a CBNeed source program will be compiled into our IL. 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

$$
\begin{aligned}
\underline{x_0{:}\sigma_0, \ldots, x_n{:}\sigma_n \vdash M : \tau} &= x_0{:}\underline{\sigma_0}, \ldots, x_n{:}\underline{\sigma_n} \vdash \underline{M} : \underline{\tau} \\
\underline{\tau \to \sigma} &= \check{U}\ (\tilde{U}\ \underline{\tau} \to \tilde{F}\ \underline{\sigma}) \\
\underline{\mathbb{N}} &= \hat{F}\ \mathbb{N} \\
\underline{\tau \times \sigma} &= \hat{F}\ (\tilde{U}\ \underline{\tau} \otimes \tilde{U}\ \underline{\sigma}) \\
\\
\underline{x} &= x \\
\underline{\mathtt{b}} &= \mathtt{val}\ \mathtt{b} \\
\underline{\lambda x.\, M} &= \{\mathtt{enter} \to \lambda y.\, \mathtt{case}\ y\ \mathtt{of} \\
&\qquad \{\mathtt{box}\ x \to \{\mathtt{eval} \to \underline{M}\}\}\} \\
\underline{M\ N} &= \underline{M}\ \mathtt{memo}\ a\ \mathtt{in}\ \underline{N}\ \mathtt{memo}\ b\ \mathtt{in} \\
&\qquad (a.\mathtt{enter}\ (\mathtt{box}\ b)).\mathtt{eval} \\
\underline{\mathtt{let}\ x\ \mathtt{be}\ M\ \mathtt{in}\ N} &= \underline{M}\ \mathtt{memo}\ x\ \mathtt{in}\ \underline{N} \\
\underline{\langle M, N\rangle} &= \underline{M}\ \mathtt{memo}\ a\ \mathtt{in}\ \underline{N}\ \mathtt{memo}\ b\ \mathtt{in} \\
&\qquad \mathtt{val}\ \langle \mathtt{box}\ a, \mathtt{box}\ b\rangle \\
\underline{\mathtt{case}\ M\ \mathtt{of}\ \{\langle x, y\rangle \to N\}} &= \underline{M}\ \mathtt{to}\ z\ \mathtt{in}\ \mathtt{case}\ z\ \mathtt{of} \\
&\qquad \{\langle \mathtt{box}\ x, \mathtt{box}\ y\rangle \to \underline{N}\}
\end{aligned}
$$

Figure 7: Compiling CBNeed to CBPVS

transformation as merging the two: functions must delay their argument type with $\check{U}$ 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.

Theorem 5.1 (CBNeed Compilation Preserves Equations). *If* $\Gamma \vdash M =_{\mathrm{CBNeed}} N : \tau$, *then* $\underline{\Gamma} \vdash \underline{M} =_{\mathrm{CBPVS}} \underline{N} : \underline{\tau}$.

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.

5.5 Deriving a Closure Conversion Transformation

As in Section 3.3, we can derive a naïve closure conversion from the equational theory. In addition to the rules for $U\ \tau$ types from the naïve CBPV closure conversion, there are two other places where a closure must be built: enter-expression and introductions and the expressions bound by memoization blocks. For each, we have a

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

Figure 8: CBPV Machine Syntax

$$
\begin{array}{rl}
\mathrm{Build}_V & : \mathit{Mach.\ Env.} \times \mathit{Value} \to \mathit{Mach.\ Value} \\
\mathrm{Build}_V(\Sigma, x) & = x[\Sigma] \\
\mathrm{Build}_V(\Sigma, \mathtt{b}) & = \mathtt{b} \\
\mathrm{Build}_V(\Sigma, \langle V, W \rangle) & = \langle \mathrm{Build}_V(\Sigma, V), \mathrm{Build}_V(\Sigma, W) \rangle \\
\mathrm{Build}_V(\Sigma, \{\varsigma, \mathtt{force} \to M\}) & = \{\Sigma\, \mathrm{Build}_\varsigma(\Sigma, \varsigma), \mathtt{force} \to M\} \\
\\
\mathrm{Build}_\varsigma & : \mathit{Mach.\ Env.} \times \mathit{Env.} \to \mathit{Mach.\ Env.} \\
\mathrm{Build}_\varsigma(\Sigma, \varepsilon) & = \varepsilon \\
\mathrm{Build}_\varsigma(\Sigma, (\varsigma, V/x)) & = \mathrm{Build}_\varsigma(\Sigma, \varsigma), \mathrm{Build}_V(\Sigma, V)/x
\end{array}
$$

Figure 9: Building CBPV Machine Values

similar rule for incrementally adding free variables:

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

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

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

6 ENVIRONMENT MACHINES

To elucidate how abstract closures interact with the construction of closures at runtime in a machine, we specify our ILs' operational semantics as environment machines. We present two machines: a simple one for just CBPV and one that builds on it to add sharing.

6.1 CBPV Environment Machine

The machine is essentially Levy's CK-machine [12] augmented with an environment that delays substitutions. Its syntax is presented in Figure 8. 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 $(\Sigma, \Box\ \mathtt{to}\ x\ \mathtt{in}\ M)$ 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 of Section 3. Syntactic values can contain variables, but machine ones only refer to objects that can 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 9. 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. Additionally, we must construct a totally closed form of the expression since it will be evaluated in another environment; thus, we must capture the current machine environment Σ *and* substitute values for variables within the closure's specified environment ς. If we could guarantee that the closure was completely closed, then we would only need to do the latter. Thus, these partial closures impose an overhead and it would be better to fully close them before running programs as we will see in Theorem 8.2.

The evaluation transitions are given in Figure 10. Note that 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.

6.2 CBPV Environment Machine with Sharing

Extending the machine to handle shared expressions requires a heap to manage memoization and extra rules for the shared expressions. Figure 11 presents the syntax for this machine. Heaps are mappings from labels to closures, which will include both unevaluated and evaluated shared expressions. Machine environments are extended 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 $E\ \mathtt{memo}\ a\ \mathtt{in}\ F[a]$.

Figure 12 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. The handling of closures here operates in the same way as with the simpler machine.

Figure 13 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 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.

7 CORRECTNESS

From our abstract machines, we may build evaluators that run a reducible expression in an empty environment and stack. The evaluators pull out the value of base type at the end.

$$
\begin{array}{rcl}
\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 \\
\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, \Box\ \mathtt{to}\ x\ \mathtt{in}\ N) \cdot K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \mathtt{ret}\ V \parallel (\Sigma', \Box\ \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 \Box.\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 \Box.\mathtt{snd} \cdot K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \{\mathtt{fst} \to M; \mathtt{snd} \to N\} \parallel \Box.\mathtt{fst} \cdot K \rangle\!\rangle & \longmapsto_6 & \langle\!\langle \Sigma \parallel M \parallel K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \{\mathtt{fst} \to M; \mathtt{snd} \to N\} \parallel \Box.\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 \Box\ \mathrm{Build}_V(\Sigma, V) \cdot K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \lambda x.\, M \parallel \Box\ \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} \to M\}}
\end{array}
$$

Figure 10: CBPVS Machine Transitions

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

Figure 11: CBPVS Machine Syntax

$$
\begin{array}{rcl}
 & \vdots & \\
\mathrm{Build}_V(\Sigma, \mathtt{box}\ V) & = & \mathtt{box}\ \mathrm{Build}_V(\Sigma, V) \\
 & \vdots & \\
\mathrm{Build}_\varsigma(\Sigma, (\varsigma, V/a)) & = & \mathrm{Build}_\varsigma(\Sigma, \varsigma), \mathrm{Build}_V(\Sigma, V)/a \\
\mathrm{Build}_V & : & \mathit{Mach.\ Env.} \times \mathit{Shared\ Value} \\
 & & \quad \to \mathit{Mach.\ Shared\ Value} \\
\mathrm{Build}_V(\Sigma, a) & = & a[\Sigma] \\
\mathrm{Build}_V(\Sigma, \mathtt{val}\ V) & = & \mathtt{val}\ \mathrm{Build}_V(\Sigma, V) \\
\mathrm{Build}_V(\Sigma, \{\varsigma, \mathtt{enter} \to M\}) & = & \{\Sigma\, \mathrm{Build}_\varsigma(\Sigma, \varsigma), \mathtt{enter} \to M\} \\
\mathrm{Build}_a & : & \mathit{Mach.\ Env.} \times \{\{\varsigma, R\}\} \to \{\{\Sigma, R\}\} \\
\mathrm{Build}_a(\Sigma, \{\varsigma, R\}) & = & \{\Sigma\, \mathrm{Build}_\varsigma(\Sigma, \varsigma), R\}
\end{array}
$$

Figure 12: Building CBPVS Machine Values and Heap Objects

Definition 7.1 (Machine Evaluator). $\mathrm{Eval}(M) = \mathtt{b}$ where $\langle\!\langle \varepsilon \parallel M \parallel \star \rangle\!\rangle \longmapsto^* \langle\!\langle \Sigma \parallel \mathtt{ret}\ \mathtt{b} \parallel \star \rangle\!\rangle$.

Definition 7.2 (Sharing Machine Evaluator). $\mathrm{EvalS}(P) = \mathtt{b}$ where $\langle\!\langle \varepsilon \parallel \varepsilon \parallel P \parallel \star \rangle\!\rangle \longmapsto^* \langle\!\langle \Phi \parallel \Sigma \parallel \mathtt{ret}\ \mathtt{b} \parallel \star \rangle\!\rangle$.

The sharing machine contains as a subset the entire CBPV machine and CBPVS contains CBPV; therefore, the sharing machine is enough to run programs from both languages. We know that if a program computes a base value in the sharing machine and it contains no shared expressions, then the CBPV evaluator will produce the same value. Thus, the theorems in this paper only contain the CBPVS evaluator.

For our equational theories to be correct, the following comparison between closed programs of type $F\ \tau$ and our shared evaluator needs to be true:

Theorem 7.3. $\vdash M = \mathtt{ret}\ \mathtt{b} : F\ B$ *if and only if* $\mathrm{EvalS}(M) = \mathtt{b}$.

This is enough for the whole equational theory because M could be $C[V]$, $C[N]$, or $C[R]$. The theorem above is correct with respect to both of these evaluators.

For the forward direction of the theorem, we use the $\top\top$-closure logical relation technique from Pitts [22]. Such an approach is especially helpful in proving the soundness of our η laws. Unique to our setting is that our operational semantics is defined with delayed substitutions. Thus, related computations are defined as relations between pairs of computations and their delayed substitution. For the shared parts of the proof, we had to extend Pitts' approach to a Kripke logical relation since the shared abstract machine has a persistent heap. Therein related computable expressions are relations between triples of heaps, delayed substitutions, and expressions such that they are closed with respect to heap extension.

For the backward direction, we define a decoding of configurations to expressions. We show that every transition in both abstract machines corresponds to a derivable 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.

8 ADEQUACY OF CLOSURE CONVERSION

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:

$$\mathrm{Build}_V(\Sigma, \{\varsigma, \mathtt{force} \to M\}) = \{\Sigma\, \mathrm{Build}_\varsigma(\Sigma, \varsigma), \mathtt{force} \to M\}$$

$$\begin{aligned}
\langle\!\langle \Sigma \parallel \texttt{case}\ V\ \texttt{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 \texttt{case}\ V\ \texttt{of}\ \{\texttt{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) = \texttt{box}\ \mathbb{V} \\
\langle\!\langle \Sigma \parallel P\ \texttt{to}\ x\ \texttt{in}\ Q \parallel K \rangle\!\rangle &\longmapsto_{13} \langle\!\langle \Sigma \parallel P \parallel (\Sigma, \square\ \texttt{to}\ x\ \texttt{in}\ Q) \cdot K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \texttt{ret}\ V \parallel (\Sigma', \square\ \texttt{to}\ x\ \texttt{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 \texttt{val}\ V \parallel (\Sigma', \square\ \texttt{to}\ x\ \texttt{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.\texttt{enter} \parallel K \rangle\!\rangle &\longmapsto_{16} \langle\!\langle \Sigma \parallel R \parallel \square.\texttt{enter} \cdot K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \{\varsigma, \texttt{enter} \to M\} \parallel \square.\texttt{enter} \cdot K \rangle\!\rangle &\longmapsto_{17} \langle\!\langle \Sigma' \parallel M \parallel K \rangle\!\rangle \\
&\textbf{where } \mathrm{Build}_V(\Sigma, V) = \{\Sigma', \texttt{enter} \to M\} \\
\langle\!\langle \Sigma \parallel M.\texttt{eval} \parallel K \rangle\!\rangle &\longmapsto_{18} \langle\!\langle \Sigma \parallel M \parallel \square.\texttt{eval} \cdot K \rangle\!\rangle \\
\langle\!\langle \Sigma \parallel \{\texttt{eval} \to R\} \parallel \square.\texttt{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{aligned}$$

(a) Additional Stateless Transitions

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

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

(b) Stateful Transitions

Figure 13: CBPVS Machine Transitions

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:

$$\mathrm{Build}^{\mathrm{cl}}_V(\Sigma, \{\varsigma, \texttt{force} \to M\}) = \{\mathrm{Build}_\varsigma(\Sigma, \varsigma), \texttt{force} \to M\}$$

Now building machine values for closures only looks up the variables in the environment specified in the syntax. 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 8.1 (Closure Converted CBPVS Machine Evaluator). $\mathrm{Eval}_{\mathrm{CC}}(P) = \mathtt{b}$ where $\langle\!\langle \varepsilon \parallel \varepsilon \parallel P \parallel \star \rangle\!\rangle \longmapsto^*_{\mathrm{CC}} \langle\!\langle \Phi \parallel \Sigma \parallel \texttt{ret}\ \mathtt{b} \parallel \star \rangle\!\rangle$.

Theorem 8.2 (Adequacy). *If P is a well-typed expression in* CC*-normal form, then* $\mathrm{Eval}(P) = \mathrm{Eval}_{\mathrm{CC}}(P)$.

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.

9 RELATED WORK

Abstract closures have been used in the past for reasoning and optimization. Hannan [10] used abstract closures in an IL to implement the optimizations of Wand and Steckler [28] which reduce the variables in a closure. Minamide *et al.* [16] used them as an intermediate step in the typed closures conversion. These two works use big-step semantics and global transformations. The work most similar to ours is that of Bowman and Ahmed [5] 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 [23]. 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 [15] 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) [6] instead. As ANF [8] 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.

Ahmed and Blume [2] prove that closure conversion is fully abstract; that is, sound and complete with respect to the source and target language. Indeed, their approach also makes the source and target language the same. Therein, they describe closure conversion as a wrapper that changes a function into an abstract type; this also includes the type translation. In so doing, they can prove a local equality of closure conversion lemma; similar to our Theorem 1.1. This proof is specialized to a given closure conversion; they choose the type preserving one of Minamide *et al.* [16]. We, on the other hand, provide a methodology for proving correctness of closure conversion, wherein all transformations that are expressible within the equational theory are correct by construction. There is also a sense in which their approach proves that the "form" of closure conversion is fully abstract, but not that closure conversion has an impact on the target language's semantics. Since their wrapper transformation is based on functions, the question remains about whether or not these wrappers need to be closure converted as well. We specify explicitly with our notion of closure conversion adequacy what is required from the runtime system by a closure converted term.

10 CONCLUSION

Our goal has been to lift closure conversion into our optimization language such that $M = \mathrm{CC}(M)$. We could not use the canonical closure conversion because it is a global transformation that does not play well with equational reasoning. Using abstract closures was the way to get both a local transformation and a strong theory for optimizations. In the case of sharing, not using abstract closures would have the additional cost of mutation in the target language. Our approach to closure conversion is a divergence from how the transformation is implemented in optimizing compilers and how its correctness is proved today. Common optimizations for closures can be implemented directly without compiling to a lower-level language with a weaker equational theory and correctness is given merely by β and η conversion.

As future work, we want to implement these ideas within GHC's intermediate language, since it is organized around the small, local transformations [21] that inspired this paper. In so doing, we would also 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.* [16] and Appel [3], respectively.

REFERENCES

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. 1989. Explicit Substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. 31–46.

[2] Amal Ahmed and Matthias Blume. 2008. 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*. 157–168.

[3] Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press.

[4] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. 1995. A Call-by-Need Lambda Calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. 233–246.

[5] William J. Bowman and Amal Ahmed. 2018. 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*. 797–811.

[6] Paul Downen and Zena M. Ariola. 2018. 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*. 21:1–21:23.

[7] Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*. 193–222.

[8] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. 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*. 237–247.

[9] Sebastian Graf and Simon Peyton Jones. 2019. Selective Lambda Lifting. *CoRR* abs/1910.11717 (2019).

[10] John Hannan. 1995. Type systems for closure conversions. *Types for Program Analysis* (1995), 64.

[11] Jean-Louis Krivine. 2007. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation* 20, 3 (2007), 199–207.

[12] Paul Blain Levy. 2001. *Call-by-push-value*. Ph. D. Dissertation. Queen Mary University of London, UK.

[13] John Maraist, Martin Odersky, and Philip Wadler. 1998. The Call-by-Need Lambda Calculus. *J. Funct. Program.* 8, 3 (1998), 275–317.

[14] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. 2017. 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*. 482–494.

[15] Dylan McDermott and Alan Mycroft. 2019. 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. 235–262.

[16] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. 1996. 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*. 271–283.

[17] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. 1998. 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*. 85–97.

[18] Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure conversion is safe for space. *Proc. ACM Program. Lang.* 3, ICFP (2019), 83:1–83:29.

[19] Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. 2021. Compositional optimizations for CertiCoq. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30.

[20] Simon L. Peyton Jones and John Launchbury. 1991. 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*. 636–666.

[21] Simon L. Peyton Jones and André L. M. Santos. 1998. A Transformation-Based Optimiser for Haskell. *Sci. Comput. Program.* 32, 1-3 (1998), 3–47.

[22] Andrew M. Pitts. 2000. Parametric polymorphism and operational equivalence. *Math. Struct. Comput. Sci.* 10, 3 (2000), 321–359.

[23] Jill Seaman and S. Purushothaman Iyer. 1996. An Operational Semantics of Sharing in Lazy Evaluation. *Sci. Comput. Program.* 27, 3 (1996), 289–322.

[24] Peter Sestoft. 1997. Deriving a Lazy Abstract Machine. *J. Funct. Program.* 7, 3 (1997), 231–264.

[25] Zhong Shao and Andrew W. Appel. 2000. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 129–161.

[26] Guy L. Steele. 1978. *Rabbit: A Compiler for Scheme*. Master's thesis. Massachusetts Institute of Technology.

[27] Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. 2021. 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*. ACM, 74–89.

[28] Mitchell Wand and Paul Steckler. 1994. Selective and Lightweight Closure Conversion. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. 435–445.