Strictly Capturing Non-strict Closures

Zachary J. Sullivan
University of Oregon
Computer and Information Science
Eugene, Oregon, United States
zsulliva@cs.uoregon.edu

Paul Downen
University of Oregon
Computer and Information Science
Eugene, Oregon, United States
pdownen@cs.uoregon.edu

Zena M. Ariola
University of Oregon
Computer and Information Science
Eugene, Oregon, United States
ariola@cs.uoregon.edu

Abstract

All functional languages need closures. Closure-conversion is a compiler transformation that embeds static code into the program for creating and manipulating closures, avoiding the need for special run-time closure support. For call-by-value languages, closure-conversion has been the focus of extensive studies concerning correctness, such as type preservation and contextual equivalence, and performance, such as space usage. Unfortunately, non-strict languages have been neglected in these studies. This paper aims to fill this gap.

We begin with both a call-by-name and a call-by-need source language whose semantics automatically generates closures at run-time. Next, we give type-preserving closure-conversions for these two non-strict languages into a lower-level target language *without* automatic closure generation at run-time. Despite the fact that our source languages are non-strict, we show that closures must be created eagerly, which requires a strict notion of product in the target language. We extend logical relation techniques used to prove compiler correctness for call-by-value languages, to apply to non-strict languages too. In doing so, we identify some important properties for reasoning about memoization with a heap.

CCS Concepts: • **Software and its engineering** → *Source code generation.*

Keywords: closure-conversion, call-by-name, call-by-need

ACM Reference Format:
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 '21), January 18–19, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3441296.3441398

1 Introduction

Correctly passing code requires *closures*: a data structure combining code with its environment. This is pervasive in functional languages, where higher-order function arguments need to close over their free variables. The early abstract machines for the λ-calculus, like the SECD [15] and Krivine [14] machines, automatically create closures at runtime. Instead, *closure-conversion* embeds the instructions for creating and manipulating these closures statically at compile-time, writing them into the syntax of the program. The transformed program is closer to real machine code, which only has pointers to top-level functions absent of any local scope. Previous work has investigated the efficiency [3, 28, 29] and correctness [2, 19, 23] of closure-conversion, and explored its application in more expressive languages with dependent types [6] and mutable references [17].

This line of work, however, mostly applies to just strict languages. Non-strict languages are rarely discussed, if at all. Some work [3, 23] 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 [22], but it requires a mutable store and is not used in compilers for lazy languages. Rather, these compilers, such as those for Lazy ML [5] and Miranda [24], rely on other methods such as *lambda-lifting*. Haskell's premier optimizing compiler, GHC [26], does use closures, but they are only considered as a small part of low-level code generation.

Delaying low-level details, like closures, can have a serious cost to an optimizing compiler. Most optimizations are done in the middle of the compiler pipeline, usually expressed as transformations in the compiler's *intermediate language*. If closures are not introduced in this phase, they cannot participate in the majority of the optimizations being done. In contrast, work on intermediate languages that can express low-level details like unboxed values [25], the arities and representations of types [8], and join points [18] allows compilers to generate more efficient code, by having different optimizations iteratively improve one another's output.

But doing this for closures—in the context of an intermediate language for a non-strict compiler—is not such an easy feat. Of course non-strict languages create more 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! In a low-level language without automatic run-time 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 paper investigates the effect of evaluation strategy on closure-conversion, with the interest of promoting closure-conversion as a transformation in a non-strict intermediate language suitable for optimization. After reviewing the well-known strict closure-conversion (Section 2) and its correctness, we show how closure-conversion of a non-strict, call-by-name language cannot be embedded into a purely call-by-name target language (Section 3), but rather strictness is needed in the target language to create closures at the right moment. Similarly, we show how call-by-need languages introduce yet another unintended interaction (Section 4): some closures need to be memoized when they are run, but others don't. The contributions of this paper are:

- We specify call-by-name closure-conversion (Section 3.2) to a call-by-value target and prove its correctness (Theorems 3.1 and 3.3) with a logical relation between programs before and after conversion (Lemma 3.2).
- We specify call-by-need closure conversion (Section 4.3) to a call-by-value target with mutable state to perform memoization. We conjecture this conversion is correct, providing a heap-indexed logical relation (Conjecture 4.2), and identify several important properties for reasoning about memoization with an explicit heap.
- We introduce the concept of *partial closure-conversion* (Section 5). In a manner similar to the worker/wrapper transformation for unboxed types, only some closures are written statically into the program with the rest generated automatically at run-time. We illustrate that the same partial closure-conversion works for both call-by-name and call-by-need languages, and can capture memoization implicitly, without explicit mutable state. By its nature, partial closure-conversion requires both strict and non-strict bindings in the same intermediate language.

2 Strict Closure-Conversion

High-level, higher-order languages—like typical functional languages—make closures implicit. In these languages, the run-time system will automatically generate closures as they are needed while the program executes. In contrast, lower-level languages—like C—do not; they may have raw function pointers, but they do not close over their environment. Instead, C programmers must manually insert the instructions into their programs to capture and access a local environment with a function pointer.

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n : \mathtt{int}}\,\mathit{Num} \qquad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\,\mathit{Var} \qquad \frac{\Gamma, x{:}\tau \vdash M : \tau'}{\Gamma \vdash \lambda x.\, M : \tau \to \tau'}\,\mathit{Lam}$$

$$\frac{\Gamma \vdash M : \tau' \to \tau \quad \Gamma \vdash N : \tau'}{\Gamma \vdash M\, N : \tau}\,\mathit{App} \qquad \frac{\Gamma \vdash M : \tau \quad \Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash \mathtt{let}\; x{=}M\; \mathtt{in}\; N : \tau'}\,\mathit{Let}$$

Figure 1. Source typing rules

$$\frac{}{\langle \Sigma \parallel n \rangle \Downarrow_{\mathcal{V}} n}\,\mathit{Num} \qquad \frac{\Sigma(x) = V}{\langle \Sigma \parallel x \rangle \Downarrow_{\mathcal{V}} V}\,\mathit{Var}$$

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{V}} (\Sigma, \lambda x.\, M)}\,\mathit{Lam}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}} (\Sigma', \lambda x.\, L) \quad \langle \Sigma \parallel N \rangle \Downarrow_{\mathcal{V}} W \quad \langle \Sigma', x \mapsto W \parallel L \rangle \Downarrow_{\mathcal{V}} V}{\langle \Sigma \parallel M\, N \rangle \Downarrow_{\mathcal{V}} V}\,\mathit{App}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}} W \quad \langle \Sigma, x \mapsto W \parallel N \rangle \Downarrow_{\mathcal{V}} V}{\langle \Sigma \parallel \mathtt{let}\; x{=}M\; \mathtt{in}\; N \rangle \Downarrow_{\mathcal{V}} V}\,\mathit{Let}$$

Figure 2. Strict source semantics

Closure-conversion is tasked with translating from this high- to low-level; to remove the need for automatic run-time closure generation, by instead embedding the introduction and elimination of closures into the syntax of a given program. As review, let's look at closure-conversion in strict languages. Consider the following program:

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

When x is called, $y = 3$ is no longer in scope. So the interpreter must package this binding along with $\lambda z.\, y$, to remember it when the function is called. This program is closure-converted to:

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

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, instead of the interpreter, the function definition and call site themselves include logic for packing and unpacking its local environment to find the binding of y.

2.1 Strict Source Language: $\mathcal{V}$

To specify closure-conversion in general, we need to define the source language being translated. For illustration, we will use a small expression language with constants, λ-expressions, and let-expressions. Let-expressions are included as real syntax, instead of syntactic sugar for applied functions, since they can be given a more direct translation.

$$L, M, N \in \mathit{Exp} ::= n \mid x \mid \lambda x.\, M \mid M\, N \mid \mathtt{let}\; x{=}M\; \mathtt{in}\; N$$

$$\frac{}{\Delta;\Gamma \vdash n : \mathtt{int}}\;\textit{Num} \qquad \frac{x{:}\tau \in \Gamma}{\Delta;\Gamma \vdash x : \tau}\;\textit{Var}$$

$$\frac{\epsilon; x{:}\tau \vdash M : \sigma}{\Delta;\Gamma \vdash \lambda x.\, M : \tau \to \sigma}\;\textit{Closed Lam}$$

$$\frac{\Delta;\Gamma \vdash M : \tau \to \sigma \qquad \Delta;\Gamma \vdash N : \tau}{\Delta;\Gamma \vdash M\, N : \sigma}\;\textit{App}$$

$$\frac{\Delta;\Gamma \vdash M : \tau \qquad \Delta;\Gamma, x : \tau \vdash M : \tau'}{\Delta;\Gamma \vdash \mathtt{let}\; x = M \;\mathtt{in}\; N : \tau'}\;\textit{Let}$$

$$\frac{\Delta;\Gamma \vdash M_0 : \tau_0 \quad \ldots \quad \Delta;\Gamma \vdash M_n : \tau_n}{\Delta;\Gamma \vdash (M_0, \ldots, M_n) : \tau_0 \times \cdots \times \tau_n}\;\textit{Product}$$

$$\frac{\Delta;\Gamma \vdash M : \sigma_0 \times \cdots \times \sigma_n \qquad \Delta;\Gamma, x_0{:}\sigma_0 \ldots x_n{:}\sigma_n \vdash N : \tau}{\Delta;\Gamma \vdash \mathtt{case}\; M \;\mathtt{of}\; (x_0, \ldots, x_n) \to N : \tau}\;\textit{Case}$$

$$\frac{\Delta;\Gamma \vdash M : \tau[\sigma/X]}{\Delta;\Gamma \vdash \mathtt{pack}\; M : \exists X.\, \tau}\;\textit{Pack}$$

$$\frac{\Gamma \vdash M : \exists X.\, \sigma \qquad \Delta, X; \Gamma, x{:}\sigma \vdash N : \tau}{\Delta;\Gamma \vdash \mathtt{unpack}\; M \;\mathtt{as}\; \mathtt{x} \;\mathtt{in}\; N : \tau}\;\textit{Unpack}$$

Figure 3. Strict target language typing rules

$$\frac{}{\langle \Sigma \parallel n \rangle \Downarrow_{\mathcal{V}'} n}\;\textit{Num} \qquad \frac{\Sigma(x) = V}{\langle \Sigma \parallel x \rangle \Downarrow_{\mathcal{V}'} V}\;\textit{Var}$$

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{V}'} \lambda x.\, M}\;\textit{Lam}$$

$$\frac{\begin{array}{c}\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'} \lambda x.\, L \\ \langle \Sigma \parallel N \rangle \Downarrow_{\mathcal{V}'} V \qquad \langle \varepsilon, x \mapsto V \parallel L \rangle \Downarrow_{\mathcal{V}'} R\end{array}}{\langle \Sigma \parallel M\, N \rangle \Downarrow_{\mathcal{V}'} R}\;\textit{App}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'} W \quad \langle \Sigma, x \mapsto W \parallel N \rangle \Downarrow_{\mathcal{V}'} V}{\langle \Sigma \parallel \mathtt{let}\; x = M \;\mathtt{in}\; N \rangle \Downarrow_{\mathcal{V}'} V}\;\textit{Let}$$

$$\frac{\langle \Sigma \parallel M_0 \rangle \Downarrow_{\mathcal{V}'} V_0 \quad \ldots \quad \langle \Sigma \parallel M_n \rangle \Downarrow_{\mathcal{V}'} V_n}{\langle \Sigma \parallel (M_0, \ldots, M_n) \rangle \Downarrow_{\mathcal{V}'} (V_0, \ldots, V_n)}\;\textit{Product}$$

$$\frac{\begin{array}{c}\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'} (V_0, \ldots, V_n) \\ \langle \Sigma, x_0 \mapsto V_0, \ldots, x_n \mapsto V_n \parallel N \rangle \Downarrow_{\mathcal{V}'} R\end{array}}{\langle \Sigma \parallel \mathtt{case}\; M \;\mathtt{of}\; (x_0, \ldots, x_n) \to N \rangle \Downarrow_{\mathcal{V}'} R}\;\textit{Case}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'} V}{\langle \Sigma \parallel \mathtt{pack}\; M \rangle \Downarrow_{\mathcal{V}'} \mathtt{pack}\; V}\;\textit{Pack}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'} V \quad \langle \Sigma, x \mapsto V \parallel N \rangle \Downarrow_{\mathcal{V}'} R}{\langle \Sigma \parallel \mathtt{unpack}\; M \;\mathtt{as}\; x \;\mathtt{in}\; N \rangle \Downarrow_{\mathcal{V}'} R}\;\textit{Unpack}$$

Figure 4. Strict target language semantics

Type System. The source type system is a typical extension of the simply typed λ-calculus shown in Figure 1. This type system serves to show how closure-conversion preserves types, and how it compares with types in the target.

$$\begin{array}{rcl} \tau, \sigma \in & \textit{Type} & ::= \mathtt{int} \mid \tau \to \sigma \\ \Gamma \in & \textit{Type Environment} & ::= \varepsilon \mid \Gamma, x{:}\tau \end{array}$$

Operational Semantics. To emphasize the creation and destruction of closures, we present a big-step operational semantics with a local environment wherein the construction of closures is explicit. It evaluates a pair of an environment and an expression, *i.e.* a *configuration*, to a value.

$$\begin{array}{rcl} C \in & \textit{Configuration} & ::= \langle \Sigma \parallel M \rangle \\ \Sigma \in & \textit{Environment} & = \textit{Variable} \rightharpoonup \textit{Value} \\ V, W \in & \text{Value} & ::= n \mid (\Sigma, \lambda x.\, M) \end{array}$$

The rules for strictly evaluating an expression are presented in Figure 2. In our source semantics, the *Lam* rule knows how to automatically construct a closure and the *App* rule knows how to unpack it, instantiate its local environment, and jump into the body with the value of the actual parameter. Closure-conversion aims to remove any knowledge of closures from these run-time rules.

2.2 Strict Target Language: $\mathcal{V}'$

The target language of closure-conversion can sometimes be the same as, or a sub-language of, the source. However, in our case, the target is a different language, extending the source with products and existentials needed for closure-conversion. Also note that in the target, all λ-expressions will be closed.

$$\begin{array}{rcl} L, M, N \in & \textit{Expression} & ::= n \mid x \mid \lambda x.\, M \mid M\, N \\ & & \mid \mathtt{let}\; x = M \;\mathtt{in}\; N \mid (M_0, \ldots, M_n) \\ & & \mid \mathtt{case}\; M \;\mathtt{of}\; (x_0, \ldots, x_n) \to N \\ & & \mid \mathtt{pack}\; M \mid \mathtt{unpack}\; M \;\mathtt{as}\; x \;\mathtt{in}\; N \end{array}$$

Type System. New types are added to the target language for products and existentials. The latter requires the addition of type variables as well. To keep track of the type variables introduced by existential types, we add a typing context for live type variables Δ, which guarantees their freshness.

$$\begin{array}{rcl} \tau, \sigma \in & \textit{Type} & ::= \mathtt{int} \mid \tau \to \sigma \\ & & \mid \tau_0 \times \cdots \times \tau_n \mid X \mid \exists X.\, \tau \\ \Gamma \in & \textit{TypeEnv} & ::= \varepsilon \mid \Gamma, x{:}\tau \\ \Delta \in & \textit{TypeVars} & ::= \varepsilon \mid \Delta, X \end{array}$$

The full set of typing rules is given in Figure 3. The rule essential to closure-conversion is the closed function rule which types the body of the function with nothing in the context except the formal parameter. This ensures that the run-time semantics need not generate closures for λ-expressions.

Operational Semantics. In the source language, the set of 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.

$$\begin{array}{rcl} C \in & \textit{Config} & ::= \langle \Sigma \parallel M \rangle \\ \Sigma \in & \textit{Env} & = \textit{Variable} \rightharpoonup \textit{Value} \\ V, W \in & \textit{Value} & ::= n \mid \lambda x.\, M \mid (V_0, \ldots, V_n) \mid \mathtt{pack}\; V \end{array}$$

The source (Figure 2) and target (Figure 4) language semantics both contain a *Lam* and an *App* rules, but they behave differently. In the target, *Lam* simply returns the λ-expression; it

Expression Translation

$$\begin{aligned}
CC_{\mathcal{V}}[\![n]\!] &= n\\
CC_{\mathcal{V}}[\![x]\!] &= x\\
CC_{\mathcal{V}}[\![\mathtt{let}\ x = M\ \mathtt{in}\ N]\!] &= \mathtt{let}\ x = CC_{\mathcal{V}}[\![M]\!]\ \mathtt{in}\ CC_{\mathcal{V}}[\![N]\!]\\
CC_{\mathcal{V}}[\![\lambda x.\, M]\!] &= \mathtt{pack}\,((\vec{y}), \lambda((\vec{y}), x).CC_{\mathcal{V}}[\![M]\!])\\
&\quad \textbf{where}\ \mathrm{FV}(\lambda x.\, M) = \vec{y} = y_0, \dots, y_n\\
CC_{\mathcal{V}}[\![M\ N]\!] &= \mathtt{call}\,(CC_{\mathcal{V}}[\![M]\!], CC_{\mathcal{V}}[\![N]\!])\\
\mathtt{call}(M, N) &\overset{\text{def}}{=} \mathtt{unpack}\ M\ \mathtt{as}\ (y, f)\ \mathtt{in}\ f\ (y, N)
\end{aligned}$$

Type Translations

$$\begin{aligned}
Val_{\mathcal{V}}[\![\mathtt{int}]\!] &= \mathtt{int}\\
Val_{\mathcal{V}}[\![\tau \to \sigma]\!] &= \exists X.\, X \times (X \times Val_{\mathcal{V}}[\![\tau]\!] \to Val_{\mathcal{V}}[\![\sigma]\!])\\
Env_{\mathcal{V}}[\![\varepsilon]\!] &= \varepsilon\\
Env_{\mathcal{V}}[\![\Gamma, x{:}\tau]\!] &= Env_{\mathcal{V}}[\![\Gamma]\!], x{:}Val_{\mathcal{V}}[\![\tau]\!]
\end{aligned}$$

Figure 5. Strict closure-conversion: $\mathcal{V} \to \mathcal{V}'$

does not construct a closure since the function must already be closed. At the call-site, the target *App* 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.

2.3 Transformation

Strict closure-conversion is shown in Figure 5. 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. (We use pattern-matching λ-expressions as syntactic sugar for case expressions.) 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, as defined in the shorthand $\mathtt{call}(M, N)$.

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. For instance, $\lambda x.\, x$ and $\lambda x.\, y$, which are both functions of type $\mathtt{int} \to \mathtt{int}$, will be converted into programs of type $\exists X.X \times (X \times \mathtt{int} \to \mathtt{int})$.

2.4 Properties

Type Preservation. Closure-conversion is defined so that all well-typed expressions in the source are translated to well-typed expressions in the target. This can be proved by induction over the typing derivation.

Theorem 2.1 (Type Preservation). *If* $\Gamma \vdash M : \tau$, *then* $Env_{\mathcal{V}}[\![\Gamma]\!] \vdash CC_{\mathcal{V}}[\![M]\!] : Val_{\mathcal{V}}[\![\tau]\!]$.

Semantics Preservation. The correctness theorem that we will focus on is that a program and its translation evaluate

$$\begin{aligned}
\mathcal{M}_{\mathcal{V}}[\![\tau]\!] &\overset{\text{def}}{=} \{(C, C') \mid \forall V.\, (C \Downarrow_{\mathcal{V}} V) \implies\\
&\qquad \exists (V, V') \in \mathcal{V}_{\mathcal{V}}[\![\tau]\!].\, (C' \Downarrow_{\mathcal{V}'} V')\}\\
\mathcal{E}_{\mathcal{V}}[\![\Gamma]\!] &\overset{\text{def}}{=} \{(\Sigma, \Sigma') \mid \forall (x{:}\tau) \in \Gamma.\\
&\qquad (\Sigma(x), \Sigma'(x)) \in \mathcal{V}_{\mathcal{V}}[\![\tau]\!]\}\\
\mathcal{V}_{\mathcal{V}}[\![\mathtt{int}]\!] &\overset{\text{def}}{=} \{(n, n) \mid n \in \mathbb{Z}\}\\
\mathcal{V}_{\mathcal{V}}[\![\tau \to \tau']\!] &\overset{\text{def}}{=} \{((\Sigma, \lambda x.\, M), \mathtt{pack}\,(V'_e, V'_f))\\
&\qquad \mid \forall (W, W') \in \mathcal{V}_{\mathcal{V}}[\![\tau]\!].\\
&\qquad\quad (\langle \Sigma, x \mapsto W \parallel M\rangle\\
&\qquad\quad , \langle \varepsilon \parallel V'_f\ (V'_e, W')\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau']\!]\}
\end{aligned}$$

Figure 6. Strict closure-conversion logical relations

to the same integer value.The proof depends on a family of logical relations, given in Figure 6, between source and target syntactic categories indexed by the type of the term[1]. For example, elements of the relation $\mathcal{V}_{\mathcal{V}}[\![\mathtt{int} \to \mathtt{int}]\!]$ are pairs of source and target values that behave like functions from integers to integers. The relations have the signatures:

$$\begin{aligned}
\mathcal{M}_{\mathcal{V}} &\in \mathit{Type} \to \mathcal{P}(\mathit{Config} \times \mathit{Config})\\
\mathcal{E}_{\mathcal{V}} &\in \mathit{TypeEnv} \to \mathcal{P}(\mathit{Env} \times \mathit{Env})\\
\mathcal{V}_{\mathcal{V}} &\in \mathit{Type} \to \mathcal{P}(\mathit{Value} \times \mathit{Value})
\end{aligned}$$

The "top level" relation in the family is for source and target configurations $\mathcal{M}_{\mathcal{V}}$. If the source configuration evaluates to a value, then the target configuration must evaluate to a related value[2]. The relation $\mathcal{E}_{\mathcal{V}}$ states that a source and target environment are related when they map variables to values related at their assigned types. Finally, the relation $\mathcal{V}_{\mathcal{V}}$ between source and target values varies depending on the type. Numbers are related only if they are identical. For function types, a source closure is related to a packed target value if they behave the same for any pair of related source-target arguments. That is, the source configuration performing the function call must be related to the target configuration performing the function call.

The adequacy lemma shows that well-typed terms in the source translate to terms that, when both are lifted to configurations with related environments, produce related configurations. It can be proved by induction on the typing derivation (as shown in Appendix A).

Lemma 2.2 (Adequacy). *If* $\Gamma \vdash M : \tau$ *and* $(\Sigma, \Sigma') \in \mathcal{E}_{\mathcal{V}}[\![\Gamma]\!]$, *then* $(\langle \Sigma \parallel M\rangle, \langle \Sigma' \parallel CC_{\mathcal{V}}[\![M]\!]\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau]\!]$.

The correctness of whole programs closure-conversion is a consequence of adequacy applied to an empty environment.

[1]For readability, we use different colors for source and target syntax. For syntax that is the same in the source and target, the font is black.

[2]Unfortunately, this also means that if a source configuration does not evaluate to a value, then any target configuration is related to it trivially. When considering well-typed source programs, an evaluation is always derivable. Thus, this is not a problem in our restricted setting.

Theorem 2.3 (Closure-Conversion Preserves Evaluation)**.** *If* $\varepsilon \vdash M : \texttt{int}$, $\langle \varepsilon \parallel M \rangle \Downarrow_{\mathcal{V}} n$ *implies* $\langle \varepsilon \parallel \mathsf{CC}_{\mathcal{V}'}[\![M]\!] \rangle \Downarrow_{\mathcal{V}'} n$.

3 Non-strict Closure-Conversion

Previously, closure-conversion translated functions of a strict source language into a closure—data structures containing a closed function and a representation of its environment—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. In strict languages, data are evaluated before they are considered a value; in contrast, non-strict data are not evaluated until forced by their context, *i.e.* until they are pattern matched[3]. 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 [16]:

$$\frac{}{\langle \Sigma \parallel \mathsf{pack}\ M \rangle \Downarrow_{\mathcal{N}} (\Sigma, \mathsf{pack}\ M)}\ \textit{Pack}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{N}} (\Sigma', \mathsf{pack}\ L) \quad \langle \Sigma, x \mapsto (\Sigma', L) \parallel N \rangle \Downarrow_{\mathcal{N}} R}{\langle \Sigma \parallel \mathsf{unpack}\ M\ \mathsf{as}\ x\ \mathsf{in}\ N \rangle \Downarrow_{\mathcal{N}} R}\ \textit{Unpack}$$

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 run-time (*i.e.* in the semantics). Yet, when trying to eliminate automatic closure management, we introduced a new type…that requires automatic closure management at run-time.

Our goal is to simulate the non-strict *Pack* and *Unpack* 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 run-time 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 (1),

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

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 $\texttt{let}\ y = 2{+}1\ \texttt{in}\ldots$ might also refer to other free variables, so the right-hand-sides must be closure-converted like functions. As a consequence, delayed computations bound by let- and λ-expressions will also be converted to values—in the sense of call-by-value—ensuring that they are not evaluated too early.

[3]Non-strict data is closely related to codata types [10], which are defined entirely by their forcing contexts.

$$\frac{}{\langle \Sigma \parallel n \rangle \Downarrow_{\mathcal{N}} n}\ \textit{Num} \qquad \frac{\Sigma(x) = (\Sigma', M) \quad \langle \Sigma' \parallel M \rangle \Downarrow_{\mathcal{N}} R}{\langle \Sigma \parallel x \rangle \Downarrow_{\mathcal{N}} R}\ \textit{Var}$$

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{N}} (\Sigma, \lambda x.\, M)}\ \textit{Lam}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{N}} (\Sigma', \lambda x.\, L) \quad \langle \Sigma', x \mapsto (\Sigma, N) \parallel L \rangle \Downarrow_{\mathcal{N}} R}{\langle \Sigma \parallel M\ N \rangle \Downarrow_{\mathcal{N}} R}\ \textit{App}$$

$$\frac{\langle \Sigma, x \mapsto (\Sigma, M) \parallel N \rangle \Downarrow_{\mathcal{N}} R}{\langle \Sigma \parallel \texttt{let}\ x = M\ \texttt{in}\ N \rangle \Downarrow_{\mathcal{N}} R}\ \textit{Let}$$

Figure 7. Non-strict source semantics

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):

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

In addition to the function closure needed in strict closure-conversion, we have added a closure construction for the binding of y. This solution means that every function and let-expression in the target language will be strict. Thus, the target language of the non-strict closure-conversion is indeed that of strict closure-conversion: $\mathcal{V}'$.

3.1 Non-strict Source Language: $\mathcal{N}$

The non-strict source language makes use of the same expression syntax and type system as the strict source language.

Operational Semantics. In a non-strict language, all values are thunk closures which contain the suspended computation code and the environment it needs to execute. Unlike our strict source, values stored in the environment are different from the results of evaluation in the non-strict source.

$$\begin{array}{rcll} C \in & \textit{Configuration} & ::= & \langle \Sigma \parallel M \rangle \\ \Sigma \in & \textit{Environment} & = & \textit{Variable} \rightharpoonup \textit{Value} \\ V, W \in & \textit{Value} & ::= & (\Sigma, M) \\ R \in & \textit{Result} & ::= & n \mid (\Sigma, \lambda x.\, M) \end{array}$$

The evaluation rules are given in Figure 7. The variable rule unpacks and evaluates the object it looks up 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. The *Let* rule instead only constructs the closure for the bound expression.

Expression Translation

$$
\begin{array}{rcl}
\mathrm{CC}_{\mathcal{N}}[\![n]\!] & = & n \\
\mathrm{CC}_{\mathcal{N}}[\![x]\!] & = & \mathtt{eval}\ x \\
\mathrm{CC}_{\mathcal{N}}[\![\mathtt{let}\ x{=}M\ \mathtt{in}\ N]\!] & = & \mathtt{let}\ x{=}\mathtt{pack}\ ((\vec{y}), \lambda(\vec{y}).\mathrm{CC}_{\mathcal{N}}[\![M]\!]) \\
& & \mathtt{in}\ \mathrm{CC}_{\mathcal{N}}[\![N]\!] \\
& & \textbf{where}\ \mathrm{FV}(M) = \vec{y} = y_0, \dots, y_n \\
\mathrm{CC}_{\mathcal{N}}[\![\lambda x.\, M]\!] & = & \mathtt{pack}\ ((\vec{y}), \lambda((\vec{y}), x).\mathrm{CC}_{\mathcal{N}}[\![M]\!]) \\
& & \textbf{where}\ \mathrm{FV}(\lambda x.\, M) = \vec{y} = y_0, \dots, y_n \\
\mathrm{CC}_{\mathcal{N}}[\![M\ N]\!] & = & \mathtt{let}\ z = \mathtt{pack}\ ((\vec{y}), \lambda(\vec{y}).\mathrm{CC}_{\mathcal{N}}[\![N]\!])) \\
& & \mathtt{in}\ \mathtt{call}\,(\mathrm{CC}_{\mathcal{N}}[\![M]\!], z) \\
& & \textbf{where}\ \mathrm{FV}(N) = \vec{y} = y_0, \dots, y_n \\
\mathtt{eval}\ M & \overset{\text{def}}{=} & \mathtt{unpack}\ M\ \mathtt{as}\ (y, f)\ \mathtt{in}\ f\ y \\
\mathtt{call}(M, N) & \overset{\text{def}}{=} & \mathtt{unpack}\ M\ \mathtt{as}\ (y, f)\ \mathtt{in}\ f(y, N)
\end{array}
$$

Type Translations

$$
\begin{array}{rcl}
Res_{\mathcal{N}}[\![\mathtt{int}]\!] & = & \mathtt{int} \\
Res_{\mathcal{N}}[\![\tau \to \sigma]\!] & = & \exists X.\, X \times (X \times Val_{\mathcal{N}}[\![\tau]\!] \to Res_{\mathcal{N}}[\![\sigma]\!]) \\
Val_{\mathcal{N}}[\![\tau]\!] & = & \exists X.\, X \times (X \to Res_{\mathcal{N}}[\![\tau]\!]) \\
Env_{\mathcal{N}}[\![\varepsilon]\!] & = & \varepsilon \\
Env_{\mathcal{N}}[\![\Gamma, x{:}\tau]\!] & = & Env_{\mathcal{N}}[\![\Gamma]\!], x{:}Val_{\mathcal{N}}[\![\tau]\!]
\end{array}
$$

Figure 8. Non-strict closure-conversion: $\mathcal{N} \to \mathcal{V}'$

3.2 Transformation

Non-strict closure-conversion, transforming $\mathcal{N}$ into $\mathcal{V}'$, is presented in Figure 8. Echoes of the source semantics are seen in the transformation. Variables are converted into code for unpacking thunk closures as we see in the *Var* rule. Applications are converted into code that turns arguments into thunk closures like the *App* rule. The non-strict transformation is careful to distinguish 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 ($Res_{\mathcal{N}}$) versus placing them in the environment ($Val_{\mathcal{N}}$), along with translation of the environment needed for evaluating an expression ($Env_{\mathcal{N}}$). The result type translation of a function has changed from the strict closure-conversion to reflect that it now accepts only thunks as arguments.

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

Theorem 3.1 (Type Preservation). *If* $\Gamma \vdash M : \tau$, *then* $Env_{\mathcal{N}}[\![\Gamma]\!] \vdash \mathrm{CC}_{\mathcal{N}}[\![M]\!] : Res_{\mathcal{N}}[\![\tau]\!]$.

Semantic Preservation. 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

$$
\begin{array}{rcl}
\mathcal{M}_{\mathcal{N}}[\![\tau]\!] & \overset{\text{def}}{=} & \{(C, C') \mid \forall R.\, (C \Downarrow_{\mathcal{N}} R) \implies \\
& & \quad \exists (R, R') \in \mathcal{R}_{\mathcal{N}}[\![\tau]\!].\, (C' \Downarrow_{\mathcal{N}'} R')\} \\
\mathcal{E}_{\mathcal{N}}[\![\Gamma]\!] & \overset{\text{def}}{=} & \{(\Sigma, \Sigma') \mid \forall (x{:}\tau) \in \Gamma. \\
& & \quad (\Sigma(x), \Sigma'(x)) \in \mathcal{V}_{\mathcal{N}}[\![\tau]\!]\} \\
\mathcal{V}_{\mathcal{N}}[\![\tau]\!] & \overset{\text{def}}{=} & \{((\Sigma, M), \mathtt{pack}\ (V'_e, V'_f)) \\
& & \mid (\langle \Sigma \parallel M \rangle, \langle \varepsilon \parallel V'_f\ V'_e \rangle) \in \mathcal{M}_{\mathcal{N}}[\![\tau]\!]\} \\
\mathcal{R}_{\mathcal{N}}[\![\mathtt{int}]\!] & \overset{\text{def}}{=} & \{(n, n) \mid n \in \mathbb{Z}\} \\
\mathcal{R}_{\mathcal{N}}[\![\tau \to \tau']\!] & \overset{\text{def}}{=} & \{((\Sigma, \lambda x.\, M), \mathtt{pack}\,(V'_e, V'_f)) \\
& & \mid \forall (W, W') \in \mathcal{V}_{\mathcal{N}}[\![\tau]\!]. \\
& & \quad (\langle \Sigma, x \mapsto W \parallel M \rangle \\
& & \quad , \langle \varepsilon \parallel V'_f\ (V'_e, W') \rangle) \in \mathcal{M}_{\mathcal{N}}[\![\tau']\!]\}
\end{array}
$$

Figure 9. Non-strict closure-conversion logical relations

those of the semantics and type translations. Thus, the non-strict family of relations in Figure 9 includes a separate relation for values and results:

$$
\begin{array}{rcccl}
\mathcal{M}_{\mathcal{N}} & \in & \mathit{Type} & \to & \mathcal{P}(\mathit{Config} \times \mathit{Config}) \\
\mathcal{E}_{\mathcal{N}} & \in & \mathit{TypeEnv} & \to & \mathcal{P}(\mathit{Env} \times \mathit{Env}) \\
\mathcal{V}_{\mathcal{N}} & \in & \mathit{Type} & \to & \mathcal{P}(\mathit{Value} \times \mathit{Value}) \\
\mathcal{R}_{\mathcal{N}} & \in & \mathit{Type} & \to & \mathcal{P}(\mathit{Result} \times \mathit{Result})
\end{array}
$$

The $\mathcal{V}_{\mathcal{V}}$ relation has become the result relation $\mathcal{R}_{\mathcal{N}}$ for non-strict closure-conversion. The relation for values, $\mathcal{V}_{\mathcal{N}}$, is 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, adequacy of this logical relation (whose proof is shown in Appendix B) implies correct evaluation.

Lemma 3.2 (Adequacy). *If* $\Gamma \vdash M : \tau$ *and* $(\Sigma, \Sigma') \in \mathcal{E}_{\mathcal{N}}[\![\Gamma]\!]$, *then* $(\langle \Sigma \parallel M \rangle, \langle \Sigma' \parallel \mathrm{CC}_{\mathcal{N}}[\![M]\!] \rangle) \in \mathcal{M}_{\mathcal{N}}[\![\tau]\!]$.

Theorem 3.3 (Closure-Conversion Preserves Evaluation). *If* $\varepsilon \vdash M : \mathtt{int}$, $\langle \varepsilon \parallel M \rangle \Downarrow_{\mathcal{N}} n$ *implies* $\langle \varepsilon \parallel \mathrm{CC}_{\mathcal{N}}[\![M]\!] \rangle \Downarrow_{\mathcal{N}'} n$.

4 Lazy 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 call-by-need evaluation of the resulting program from non-strict closure-conversion (again, avoiding the closures necessary for x, 3, and 4) of the program in (1).

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

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 closure-converted into a pointer to a heap-allocated tagged thunk. We will use the following shorthand for tagged heap storage:

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

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:

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

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

$$\begin{array}{l} \texttt{let}\ x = \texttt{let}\ y = \texttt{store}\,(\texttt{pack}\,((), \lambda().\,2+1))) \\ \qquad\qquad \texttt{in pack}\,((y), \lambda((y), z).\texttt{memo}\ y) \\ \texttt{in}\,(\texttt{unpack}\ x\ \texttt{as}\,(e,f)\ \texttt{in}\ f\,(e,3)) + (\texttt{unpack}\ x\ \texttt{as}\,(e,f)\ \texttt{in}\ f\,(e,4)) \end{array}$$

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).

4.1 Lazy Source Language: $\mathcal{L}$

Operational Semantics. The major difference in the lazy semantics versus strict and non-strict is the addition of the heap. This means we must now distinguish results, values, and *answers.* Answers are the set of normalized expressions and results now contain 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.

$C \in$	*Configuration*	$::= \langle \Phi \parallel \Sigma \parallel M \rangle$
$\Phi \in$	*Heap*	$= \mathit{Location} \rightharpoonup \mathit{Heap\ Object}$
$l \in$	*Location*	
$O \in$	*Heap Object*	$::= (\Sigma, M) \mid A$
$\Sigma \in$	*Environment*	$= \mathit{Variable} \rightharpoonup \mathit{Value}$
$V \in$	*Value*	$= \mathit{Location}$
$A \in$	*Answer*	$::= n \mid (\Sigma, \lambda x.\, M)$
$R \in$	*Result*	$= \mathit{Heap} \times \mathit{Answer}$

The big-step evaluation rules are specified in Figure 10. Just like the other two source languages, the *Lam* rule must construct a function closure. Like the non-strict language,

$$\frac{}{\langle \Phi \parallel \Sigma \parallel n \rangle \Downarrow_{\mathcal{L}} (\Phi, n)}\ \mathit{Num} \qquad \frac{\Phi(\Sigma(x)) = A}{\langle \Phi \parallel \Sigma \parallel x \rangle \Downarrow_{\mathcal{L}} (\Phi, A)}\ \mathit{VarAns}$$

$$\frac{\Phi(\Sigma(x)) = (\Sigma', M) \quad \langle \Phi \parallel \Sigma' \parallel M \rangle \Downarrow_{\mathcal{L}} (\Phi', A) \quad \texttt{update}(\Phi', \Sigma(x), A) = \Phi''}{\langle \Phi \parallel \Sigma \parallel x \rangle \Downarrow_{\mathcal{L}} (\Phi'', A)}\ \mathit{VarMemo}$$

$$\frac{}{\langle \Phi \parallel \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{L}} (\Phi, (\Sigma, \lambda x.\, M))}\ \mathit{Lam}$$

$$\frac{\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{L}} (\Phi', (\Sigma', \lambda x.\, L)) \quad \texttt{alloc}(\Phi', (\Sigma, N)) = (l, \Phi'') \quad \langle \Phi'' \parallel \Sigma', x \mapsto l \parallel L \rangle \Downarrow_{\mathcal{L}} R}{\langle \Phi \parallel \Sigma \parallel M\ N \rangle \Downarrow_{\mathcal{L}} R}\ \mathit{App}$$

$$\frac{\texttt{alloc}(\Phi, (\Sigma, M)) = (l, \Phi') \quad \langle \Phi' \parallel \Sigma, x \mapsto l \parallel N \rangle \Downarrow_{\mathcal{N}} R}{\langle \Phi \parallel \Sigma \parallel \texttt{let}\ x{=}M\ \texttt{in}\ N \rangle \Downarrow_{\mathcal{L}} R}\ \mathit{Let}$$

Heap Semantics

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

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

Figure 10. Lazy source semantics

$$\frac{\Delta;\Gamma \vdash M : \tau}{\Delta;\Gamma \vdash \texttt{inl}\ M : \tau + \sigma}\ \mathit{Inl} \qquad \frac{\Delta;\Gamma \vdash M : \sigma}{\Delta;\Gamma \vdash \texttt{inr}\ M : \tau + \sigma}\ \mathit{Inr}$$

$$\frac{\Delta;\Gamma \vdash M : \sigma_l + \sigma_r \quad \Delta;\Gamma, x{:}\sigma_l \vdash N : \tau \quad \Delta;\Gamma, x{:}\sigma_r \vdash L : \tau}{\Delta;\Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \{\texttt{inl}\ x \longrightarrow N; \texttt{inr}\ x \longrightarrow L\} : \tau}\ \mathit{CaseSum}$$

$$\frac{\Delta;\Gamma \vdash M : \tau}{\Delta;\Gamma \vdash \texttt{new}\ M : \texttt{ref}\ \tau}\ \mathit{New} \qquad \frac{\Delta;\Gamma \vdash M : \texttt{ref}\ \tau}{\Delta;\Gamma \vdash !M : \tau}\ \mathit{Deref}$$

$$\frac{\Delta;\Gamma \vdash M : \texttt{ref}\ \tau \quad \Delta;\Gamma \vdash N : \tau}{\Delta;\Gamma \vdash M := N : 1}\ \mathit{Mutate}$$

Figure 11. Typing rules for $\mathcal{V}'_!$ extending $\mathcal{V}'$

the *App* 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 lazy language: function closures are returned from evaluations, whereas thunk closures are passed as pointers to the heap where they can be updated.

We model heaps as objects in 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.

4.2 Strict Target Language with Mutation: $\mathcal{V}'_!$

In order to handle the added problem of updating thunks, the strict target language for lazy closure-conversion extends $\mathcal{V}'$ with sums and mutable references.

$$\begin{array}{lll} L, M, N \in & \mathit{Exp} & ::= n \mid x \mid \lambda x.\, M \mid M\ N \\ & & \mid \mathtt{let}\ x = M\ \mathtt{in}\ N \\ & & \mid (M_0, \ldots, M_n) \\ & & \mid \mathtt{case}\ M\ \mathtt{of}\ (x_0, \ldots, x_n) \to N \\ & & \mid \mathtt{pack}\ M \mid \mathtt{unpack}\ M\ \mathtt{as}\ x\ \mathtt{in}\ N \\ & & \mid \mathtt{inl}\ M \mid \mathtt{inr}\ M \\ & & \mid \mathtt{case}\ L\ \mathtt{of}\ \{\mathtt{inl}\ x \to M; \mathtt{inr}\ x \to N\} \end{array}$$

Typing Rules. The typing rules for this target language are given in Figure 11. They are a direct extension of the rules for the strict target language $\mathcal{V}'$. Heap manipulation in $\mathcal{V}'_!$ is through reference types, à la SML. This can be seen in the *Mutate* 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 ().

Operational Semantics. Our strict, mutable target language can do away with the distinctions between heap objects, answers and values that were present in $\mathcal{L}$. Now, both heaps and environments may contain any value. Results are pairs of a heap and value.

$$\begin{array}{lll} C \in & \mathit{Configuration} & ::= \langle \Phi \parallel \Sigma \parallel M \rangle \\ \Phi \in & \mathit{Heap} & = \mathit{Location} \rightharpoonup \mathit{Value} \\ l \in & \mathit{Location} & \\ \Sigma \in & \mathit{Environment} & = \mathit{Variable} \rightharpoonup \mathit{Value} \\ V \in & \mathit{Value} & ::= n \mid \lambda x.\, M \mid (V_0, \ldots, V_n) \\ & & \quad \mid \mathtt{pack}\ V \mid \mathtt{inl}\ V \mid \mathtt{inr}\ V \mid l \\ R \in & \mathit{Result} & = \mathit{Heap} \times \mathit{Value} \end{array}$$

The semantics is given in Figure 12. Unlike the syntax and typing rules for $\mathcal{V}'_!$, which were a direct extension of the target language $\mathcal{V}'$, 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 *New* rule 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 ().

4.3 Transformation

The lazy closure-conversion transformation is found in Figure 13. Our lazy source language's different variable lookup rules are encoded in a single case expression (defined by `memo`): 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 $\mathit{Val}_{\mathcal{L}}[\![\tau]\!]$ translation.

Interestingly, the cases for already normalized expressions (*i.e.* numbers and manifest functions) are the same for strict, non-strict, and lazy closure-conversion.

Theorem 4.1 (Type Preservation)**.** *If* $\Gamma \vdash M : \tau$*, then* $\mathit{Env}_{\mathcal{L}}[\![\Gamma]\!] \vdash \mathrm{CC}_{\mathcal{L}}[\![M]\!] : \mathit{Res}_{\mathcal{L}}[\![\tau]\!]$.

4.3.1 Semantics Preservation.

The technique to reasoning about semantics that we used in the strict and non-strict setting does not easily apply to the lazy one because answers can depend on mutable objects in the heap. In effect, the use of a heap dislocates bindings from their static scope. Closures capture only pointers into the heap, and evaluating an expression must use the dynamically newest heap, rather than the purely static bindings that were possible for strict and non-strict evaluation. For example, consider evaluating

$$\begin{array}{l} \mathtt{let}\ x = 1 + 1 \\ \mathtt{in}\ \mathtt{let}\ y = x + 2 \\ \quad \mathtt{in}\ x + y \end{array}$$

First, thunks for $x = 1+1$ and $y = x+2$ get added to the heap, Φ_1. Then, the addition $x + y$ forces x, returning the result 2 in an updated heap Φ_2 where x has been evaluated. Next, addition forces y so that $x+2$ is evaluated in the newest heap Φ_2, *not* the heap Φ_1 that was available at its binding site.

In general, function application places a thunk value representing the delayed argument in the current heap found at the time of the call. Later, that thunk may be retrieved when that argument is forced, and it must be evaluated with the state of the heap at the time of forcing, not at the time when it was bound. Therefore, we need to be able to reason about evaluating arguments not just *now*—when the binding is formed—but also *later*—when the argument is needed.

In order to specify the behavior of evaluation at different points in time as the dynamic heap changes, we need to enhance our logical relations to be indexed by the current state of the heap. A heap-indexed family of logical relations for lazy evaluation is given in Figure 14, with these signatures:

$$\begin{array}{lll} \mathcal{M}_{\mathcal{L}} & \in & \mathit{Type} \to \mathcal{P}(\mathit{Config} \times \mathit{Config}) \\ \mathcal{R}_{\mathcal{L}} & \in & \mathit{Type} \to \mathit{Heap} \times \mathit{Heap} \to \mathcal{P}(\mathit{Result} \times \mathit{Result}) \\ \mathcal{E}_{\mathcal{L}} & \in & \mathit{TypeEnv} \to \mathit{Heap} \times \mathit{Heap} \to \mathcal{P}(\mathit{Env} \times \mathit{Env}) \\ \mathcal{V}_{\mathcal{L}} & \in & \mathit{Type} \to \mathit{Heap} \times \mathit{Heap} \to \mathcal{P}(\mathit{Value} \times \mathit{Value}) \\ \mathcal{A}_{\mathcal{L}} & \in & \mathit{Type} \to \mathit{Heap} \times \mathit{Heap} \to \mathcal{P}(\mathit{Ans.} \times \mathit{Ans.}) \end{array}$$

Environments, values, and answers all depend on a heap by containing variables which map to pointers. Thus, we augment their relations to depend on a pair of current source and target heaps to specify their meaning. $\mathcal{E}_{\mathcal{L}}$ must look up

$$\frac{}{\langle \Phi \parallel \Sigma \parallel n \rangle \Downarrow_{\mathcal{V}'_!} (\Phi, n)}\,\mathit{Num} \qquad \frac{\Sigma(x) = V}{\langle \Phi \parallel \Sigma \parallel x \rangle \Downarrow_{\mathcal{V}'_!} (\Phi, V)}\,\mathit{Var} \qquad \frac{}{\langle \Phi \parallel \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi, \lambda x.\, M)}\,\mathit{Lam}$$

$$\frac{\begin{array}{c}\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', \lambda x.\, L) \\ \langle \Phi' \parallel \Sigma \parallel N \rangle \Downarrow_{\mathcal{V}'_!} (\Phi'', V) \quad \langle \Phi'' \parallel \varepsilon, x \mapsto V \parallel L \rangle \Downarrow_{\mathcal{V}'_!} R\end{array}}{\langle \Phi \parallel \Sigma \parallel M\ N \rangle \Downarrow_{\mathcal{V}'_!} R}\,\mathit{App} \qquad \frac{\begin{array}{c}\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', V) \\ \langle \Phi \parallel \Sigma, x \mapsto V \parallel N \rangle \Downarrow_{\mathcal{V}'_!} R\end{array}}{\langle \Phi \parallel \Sigma \parallel \mathtt{let}\ x = M\ \mathtt{in}\ N \rangle \Downarrow_{\mathcal{V}'_!} R}\,\mathit{Let}$$

$$\frac{\langle \Phi_0 \parallel \Sigma \parallel M_0 \rangle \Downarrow_{\mathcal{V}'_!} (\Phi_1, V_0) \cdots \langle \Phi_n \parallel \Sigma \parallel M_n \rangle \Downarrow_{\mathcal{V}'_!} (\Phi_{n+1}, V_n)}{\langle \Phi_0 \parallel \Sigma \parallel (M_0, \ldots, M_n) \rangle \Downarrow_{\mathcal{V}'_!} (\Phi_{n+1}, (V_0, \ldots, V_n))}\,\mathit{Prod} \qquad \frac{\begin{array}{c}\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', (V_0, \ldots, V_n)) \\ \langle \Phi \parallel \Sigma, x_0 \mapsto V_0, \ldots, x_n \mapsto V_n \parallel N \rangle \Downarrow_{\mathcal{V}'_!} R\end{array}}{\langle \Phi \parallel \Sigma \parallel \mathtt{case}\ M\ \mathtt{of}\ (x_0, \ldots, x_n) \to N \rangle \Downarrow_{\mathcal{V}'_!} R}\,\mathit{CaseProd}$$

$$\frac{\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', V)}{\langle \Phi \parallel \Sigma \parallel \mathtt{pack}\ M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', \mathtt{pack}\ V)}\,\mathit{Pack} \qquad \frac{\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', V) \quad \langle \Phi' \parallel \Sigma, x \mapsto V \parallel N \rangle \Downarrow_{\mathcal{V}'_!} R}{\langle \Phi \parallel \Sigma \parallel \mathtt{unpack}\ M\ \mathtt{as}\ x\ \mathtt{in}\ N \rangle \Downarrow_{\mathcal{V}'_!} R}\,\mathit{Unpack}$$

$$\frac{\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', V)}{\langle \Phi \parallel \Sigma \parallel \mathtt{inl}\ M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', \mathtt{inl}\ V)}\,\mathit{Inl} \qquad \frac{\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', V)}{\langle \Phi \parallel \Sigma \parallel \mathtt{inr}\ M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', \mathtt{inr}\ V)}\,\mathit{Inr}$$

$$\frac{\begin{array}{c}\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', \mathtt{inl}\ V) \\ \langle \Phi' \parallel \Sigma, x \mapsto V \parallel N \rangle \Downarrow_{\mathcal{V}'_!} R\end{array}}{\langle \Phi \parallel \Sigma \parallel \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{inl}\ x \to N; \ldots\} \rangle \Downarrow_{\mathcal{V}'_!} R}\,\mathit{CaseInl} \qquad \frac{\begin{array}{c}\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', \mathtt{inr}\ V) \\ \langle \Phi' \parallel \Sigma, x \mapsto V \parallel N \rangle \Downarrow_{\mathcal{V}'_!} R\end{array}}{\langle \Phi \parallel \Sigma \parallel \mathtt{case}\ M\ \mathtt{of}\ \{\ldots; \mathtt{inr}\ x \to N\} \rangle \Downarrow_{\mathcal{V}'_!} R}\,\mathit{CaseInr}$$

$$\frac{\begin{array}{c}\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', V) \\ \mathtt{alloc}(\Phi', l, V) = (l, \Phi'')\end{array}}{\langle \Phi \parallel \Sigma \parallel \mathtt{new}\ M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi'', l)}\,\mathit{New} \qquad \frac{\begin{array}{c}\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', l) \\ \Phi'(l) = V\end{array}}{\langle \Phi \parallel \Sigma \parallel\ !M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', V)}\,\mathit{Deref} \qquad \frac{\begin{array}{c}\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}'_!} (\Phi', l) \\ \langle \Phi' \parallel \Sigma \parallel N \rangle \Downarrow_{\mathcal{V}'_!} (\Phi'', V) \quad \mathtt{update}(\Phi'', l, V) = \Phi'''\end{array}}{\langle \Phi \parallel \Sigma \parallel M := N \rangle \Downarrow_{\mathcal{V}'_!} (\Phi''', ())}\,\mathit{Mutate}$$

Figure 12. $\mathcal{V}'_!$: Strict mutable target language semantics

bindings in both the static environment (to get a location) and then the dynamic heap (to get a value). $\mathcal{V}_\mathcal{L}$ relates heap objects (closures and answers) that yield the same result in the current heap—the conversion from heap objects to computable configurations is given by the operations

$$\begin{array}{rcl} \mathrm{config}_\mathcal{L} & \in & \mathit{Heap} \times \mathit{Heap\ Object} \to \mathit{Config} \\ \mathrm{config}_{\mathcal{V}'_!} & \in & \mathit{Heap} \times \mathit{Heap\ Object} \to \mathit{Config} \end{array}$$

$\mathcal{A}_\mathcal{L}$ is almost the same as before, except that the function arguments W and W' get passed via references to the heap. (For clarity, we avoided using our black-box heap allocation in this definition.)

In contrast, configurations and results contain the heaps that they depend on, and they specify the impact of their heap as it evolves over time. The logical relation $\mathcal{M}_\mathcal{L}$ ensures that a source-target pair of configurations, $\langle \Phi_1 \parallel \Sigma \parallel M \rangle$ and $\langle \Phi'_1 \parallel \Sigma' \parallel M' \rangle$, gives related results when evaluated now in their current heaps Φ_1 and Φ'_1. But crucially, $\mathcal{M}_\mathcal{L}$ should hold in the future as well: evaluation still gives back related results when Φ_1 and Φ'_1 are replaced by any *future heaps* Φ_2 and Φ'_2. This extra generality is expressed by quantification over all $(\Phi_1, \Phi'_1) \sqsubseteq (\Phi_2, \Phi'_2)$, where the relation $\sqsubseteq$ is meant to denote that (Φ_2, Φ'_2) is a pair of related source-target future heaps that might come from evaluation with (Φ_1, Φ'_1). The logical relation $\mathcal{R}_\mathcal{L}$ ensures that a source-target pair of results, (Φ_2, A) and (Φ'_2, A'), contain related answers in their respective current heaps Φ_2 and Φ'_2. But in addition, $\mathcal{R}_\mathcal{L}[\![\tau]\!](\Phi_1, \Phi_2)$ also ensures that the contained heaps Φ_2 and Φ'_2 are indeed possible future heaps of an older Φ_1 and Φ_2 used to evaluate the result, as expressed by $(\Phi_1, \Phi'_1) \sqsubseteq (\Phi_2, \Phi'_2)$.

4.3.2 Future Heaps.

Note that while Figure 14 makes use of the notion of future heaps, written as $(\Phi_1, \Phi'_1) \sqsubseteq (\Phi_2, \Phi'_2)$, it does not define the $\sqsubseteq$ relation. We conjecture that this last missing piece will make it possible to reason about the adequacy of the logical relation in Figure 14, which implies correctness of lazy closure-conversion:

Conjecture 4.2 (Adequacy). *There is a definition of* $\sqsubseteq$ *such that the following holds: If* $\Gamma \vdash M : \tau$ *and* $(\Sigma, \Sigma') \in \mathcal{E}_\mathcal{L}[\![\Gamma]\!](\Phi, \Phi')$, *then* $(\langle \Phi \parallel \Sigma \parallel M \rangle, \langle \Phi' \parallel \Sigma' \parallel \mathrm{CC}_\mathcal{L}[\![M]\!] \rangle) \in \mathcal{M}_\mathcal{L}[\![\tau]\!]$.

While we leave the definition of $\sqsubseteq$ as an open problem, we can still specify the essential properties of future heaps required to prove Conjecture 4.2. Of course, this relation should be reflexive and transitive, which follows the usual intuition of "future" states. The most important property is that the $\mathcal{E}_\mathcal{L}$, $\mathcal{V}_\mathcal{L}$, and $\mathcal{A}_\mathcal{L}$ relations are *immortal*: all relations under a particular source-target pair of heaps continue for all future heaps, and never die.

Property 1 (Immortality). *For all* $(\Phi_1, \Phi'_1) \sqsubseteq (\Phi_2, \Phi'_2)$, *the following hold:*

1. *If* $(A, A') \in \mathcal{A}_\mathcal{L}[\![\tau]\!](\Phi_1, \Phi'_1)$ *then* $(A, A') \in \mathcal{A}_\mathcal{L}[\![\tau]\!](\Phi_2, \Phi'_2)$.
2. *If* $(O, O') \in \mathcal{V}_\mathcal{L}[\![\tau]\!](\Phi_1, \Phi'_1)$ *then* $(O, O') \in \mathcal{V}_\mathcal{L}[\![\tau]\!](\Phi_2, \Phi'_2)$.
3. *If* $(\Sigma, \Sigma') \in \mathcal{E}_\mathcal{L}[\![\Gamma]\!](\Phi_1, \Phi'_1)$ *then* $(\Sigma, \Sigma') \in \mathcal{E}_\mathcal{L}[\![\Gamma]\!](\Phi_2, \Phi'_2)$.

Expression Translation

$$
\begin{array}{rcl}
\mathrm{CC}_{\mathcal{L}}\llbracket n \rrbracket & = & n \\
\mathrm{CC}_{\mathcal{L}}\llbracket x \rrbracket & = & \mathtt{memo}\; x \\
\mathrm{CC}_{\mathcal{L}}\llbracket \mathtt{let}\; x{=}M\; \mathtt{in}\; N \rrbracket & = & \mathtt{let}\; x = \\
 & & \quad \mathtt{store}(\mathtt{pack}((\vec{y}), \lambda(\vec{y}).\mathrm{CC}_{\mathcal{L}}\llbracket M \rrbracket)) \\
 & & \mathtt{in}\; \mathrm{CC}_{\mathcal{L}}\llbracket N \rrbracket \\
 & & \textbf{where } \mathrm{FV}(M) = \vec{y} = y_0, \ldots, y_n \\
\mathrm{CC}_{\mathcal{L}}\llbracket \lambda x.\, M \rrbracket & = & \mathtt{pack}\; ((\vec{y}), \lambda((\vec{y}), x).\, \mathrm{CC}_{\mathcal{L}}\llbracket M \rrbracket) \\
 & & \textbf{where } \mathrm{FV}(\lambda x.\, M) = \vec{y} = y_0, \ldots, y_n \\
\mathrm{CC}_{\mathcal{L}}\llbracket M\; N \rrbracket & = & \mathtt{let}\; x = \\
 & & \quad \mathtt{store}(\mathtt{pack}((\vec{y}), \lambda(\vec{y}).\mathrm{CC}_{\mathcal{L}}\llbracket N \rrbracket)) \\
 & & \mathtt{in}\; \mathtt{call}\; (\mathrm{CC}_{\mathcal{L}}\llbracket M \rrbracket, x) \\
 & & \textbf{where } \mathrm{FV}(N) = \vec{y} = y_0, \ldots, y_n \\
\mathtt{memo}\; x & \overset{\mathrm{def}}{=} & \mathtt{case}\; !x\; \mathtt{of} \\
 & & \quad \mathtt{inl}\; v \longrightarrow v \\
 & & \quad \mathtt{inr}\; t \longrightarrow \\
 & & \qquad \mathtt{unpack}\; t\; \mathtt{as}\; (y, z)\; \mathtt{in} \\
 & & \qquad\quad \mathtt{let}\; v = z\; y\; \mathtt{in} \\
 & & \qquad\qquad \mathtt{let}\; \_ \; = (x := \mathtt{inl}\; v)\; \mathtt{in}\; v \\
\mathtt{store}\; M & \overset{\mathrm{def}}{=} & \mathtt{new}\; (\mathtt{inr}\; M) \\
\mathtt{call}\; (M, N) & \overset{\mathrm{def}}{=} & \mathtt{unpack}\; M\; \mathtt{as}\; (y, f)\; \mathtt{in}\; f\; (y, N)
\end{array}
$$

Type Translations

$$
\begin{array}{rcl}
\mathit{Res}_{\mathcal{L}}\llbracket \mathtt{int} \rrbracket & = & \mathtt{int} \\
\mathit{Res}_{\mathcal{L}}\llbracket \tau \to \sigma \rrbracket & = & \exists X.\, X \times (X \times \mathit{Val}_{\mathcal{L}}\llbracket \tau \rrbracket \to \mathit{Res}_{\mathcal{L}}\llbracket \sigma \rrbracket) \\
\mathit{Val}_{\mathcal{L}}\llbracket \tau \rrbracket & = & \mathtt{ref}\; (\mathit{Res}_{\mathcal{L}}\llbracket \tau \rrbracket + (\exists X.\, X \times (X \to \mathit{Res}_{\mathcal{L}}\llbracket \tau \rrbracket))) \\
\mathit{Env}_{\mathcal{L}}\llbracket \varepsilon \rrbracket & = & \varepsilon \\
\mathit{Env}_{\mathcal{L}}\llbracket \Gamma, x{:}\tau \rrbracket & = & \mathit{Env}_{\mathcal{L}}\llbracket \Gamma \rrbracket, x{:}\mathit{Val}_{\mathcal{L}}\llbracket \tau \rrbracket
\end{array}
$$

Figure 13. Lazy closure-conversion

$$
\begin{array}{rcl}
\mathcal{M}_{\mathcal{L}}\llbracket \tau \rrbracket & \overset{\mathrm{def}}{=} & \{(\langle \Phi_1 \parallel \Sigma \parallel M \rangle, \langle \Phi'_1 \parallel \Sigma' \parallel M' \rangle) \\
 & & \mid \forall (\Phi_1, \Phi'_1) \sqsubseteq (\Phi_2, \Phi'_2). \\
 & & \quad \forall R.\, (\langle \Phi_2 \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{L}} R) \implies \\
 & & \qquad \exists (R, R') \in \mathcal{R}_{\mathcal{L}}\llbracket \tau \rrbracket(\Phi_2, \Phi'_2). \\
 & & \qquad \langle \Phi'_2 \parallel \Sigma' \parallel M' \rangle \Downarrow_{\mathcal{V}'_!} R' \} \\
\mathcal{R}_{\mathcal{L}}\llbracket \tau \rrbracket(\Phi, \Phi') & \overset{\mathrm{def}}{=} & \{((\Phi_1, A), (\Phi'_1, A')) \\
 & & \mid (\Phi, \Phi') \sqsubseteq (\Phi_1, \Phi'_1) \\
 & & \quad \wedge (A, A') \in \mathcal{A}_{\mathcal{L}}\llbracket \tau \rrbracket(\Phi_1, \Phi'_1)\} \\
\mathcal{E}_{\mathcal{L}}\llbracket \Gamma \rrbracket(\Phi, \Phi') & \overset{\mathrm{def}}{=} & \{(\Sigma, \Sigma') \mid \forall (x{:}\tau) \in \Gamma. \\
 & & \quad (\Phi(\Sigma(x)), \Phi'(\Sigma'(x))) \\
 & & \qquad \in \mathcal{V}_{\mathcal{L}}\llbracket \tau \rrbracket(\Phi, \Phi')\} \\
\mathcal{V}_{\mathcal{L}}\llbracket \tau \rrbracket(\Phi, \Phi') & \overset{\mathrm{def}}{=} & \{(O, O') \\
 & & \mid (\mathrm{config}_{\mathcal{L}}(\Phi, O), \mathrm{config}_{\mathcal{V}'_!}(\Phi', O')) \\
 & & \quad \in \mathcal{M}_{\mathcal{L}}\llbracket \tau \rrbracket\} \\
\mathcal{A}_{\mathcal{L}}\llbracket \mathtt{int} \rrbracket(\Phi, \Phi') & \overset{\mathrm{def}}{=} & \{(n, n) \mid n \in \mathbb{Z}\} \\
\mathcal{A}_{\mathcal{L}}\llbracket \tau \to \tau' \rrbracket(\Phi, \Phi') & \overset{\mathrm{def}}{=} & \{((\Sigma, \lambda x.M), \mathtt{pack}(V'_e, V'_f)) \\
 & & \mid \forall (W, W') \in \mathcal{V}_{\mathcal{L}}\llbracket \tau \rrbracket(\Phi, \Phi'). \\
 & & \quad (\langle \Phi, l \mapsto W \parallel \Sigma, x \mapsto l \parallel M \rangle \\
 & & \quad , \langle \Phi', l' \mapsto W' \parallel x \mapsto l' \parallel V'_f\; (V'_e, x) \rangle) \\
 & & \qquad \in \mathcal{M}_{\mathcal{L}}\llbracket \tau' \rrbracket\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{config}_{\mathcal{L}}(\Phi, (\Sigma, M)) & \overset{\mathrm{def}}{=} & \langle \Phi \parallel \Sigma \parallel M \rangle \\
\mathrm{config}_{\mathcal{L}}(\Phi, (\Sigma, \lambda x.M)) & \overset{\mathrm{def}}{=} & \langle \Phi \parallel \Sigma \parallel \lambda x.M \rangle \\
\mathrm{config}_{\mathcal{L}}(\Phi, n) & \overset{\mathrm{def}}{=} & \langle \Phi \parallel \varepsilon \parallel n \rangle \\
\mathrm{config}_{\mathcal{V}'_!}(\Phi, \mathtt{inl}\; V) & \overset{\mathrm{def}}{=} & \langle \Phi \parallel \varepsilon \parallel V \rangle \\
\mathrm{config}_{\mathcal{V}'_!}(\Phi, \mathtt{inr}(\mathtt{pack}(V_e, V_f))) & \overset{\mathrm{def}}{=} & \langle \Phi \parallel \varepsilon \parallel V_f\; V_e \rangle
\end{array}
$$

Figure 14. Lazy closure-conversion logical relation

Immortality is a form of monotonicity property, the temporal relationship between heaps (as $(\Phi_1, \Phi'_1) \sqsubseteq (\Phi_2, \Phi'_2)$) is preserved as a logical relationship between typed correctness properties (*e.g.* $\mathcal{A}_{\mathcal{L}}\llbracket \tau \rrbracket(\Phi_1, \Phi'_1) \subseteq \mathcal{A}_{\mathcal{L}}\llbracket \tau \rrbracket(\Phi_2, \Phi'_2)$). Notice that the immortality property for $\mathcal{E}_{\mathcal{L}}$ implies any heap object accessible through a variable x will remain accessible so long as x is in scope. The immortality property for $\mathcal{V}_{\mathcal{L}}$ holds by virtue of its definition in terms of $\mathcal{M}_{\mathcal{L}}$, which already abstracts over future heaps, so long as $\sqsubseteq$ is transitive. Immortality of $\mathcal{A}_{\mathcal{L}}$ requires something more. Because $\mathcal{A}_{\mathcal{L}}\llbracket \tau \to \tau' \rrbracket$ stores functions arguments on the heap, future heaps must be preserved under extension.

Property 2 (Extension). *If* $(\Phi_1, \Phi'_1) \sqsubseteq (\Phi_2, \Phi'_2)$,

$$
\begin{array}{l}
\mathrm{dom}(\Phi_1) \cap \mathrm{dom}(\Phi_3) = \emptyset = \mathrm{dom}(\Phi_2) \cap \mathrm{dom}(\Phi_3), \textit{ and} \\
\mathrm{dom}(\Phi'_1) \cap \mathrm{dom}(\Phi'_3) = \emptyset = \mathrm{dom}(\Phi'_2) \cap \mathrm{dom}(\Phi'_3),
\end{array}
$$

then $((\Phi_1, \Phi_3), (\Phi'_1, \Phi'_3)) \sqsubseteq ((\Phi_2, \Phi_3), (\Phi'_2, \Phi'_3))$.

Additionally, when evaluation forces a variable, the result of that variable gets memoized, which causes an update to the heap. This, too, must be a possible future heap.

Property 3 (Memoization). *If* $(\Sigma, \Sigma') \in \mathcal{E}_{\mathcal{L}}\llbracket \Gamma, x{:}\tau \rrbracket(\Phi_1, \Phi'_1)$,

$$
\begin{array}{c}
\mathrm{config}_{\mathcal{L}}(\Phi_1, \Phi_1(\Sigma(x))) \Downarrow_{\mathcal{L}} (\Phi_2, A) \\
\mathrm{config}_{\mathcal{V}'_!}(\Phi'_1, \Phi'_1(\Sigma'(x))) \Downarrow_{\mathcal{V}'_!} (\Phi'_2, A')
\end{array}
$$

$\mathtt{update}(\Phi_2, \Sigma(x), A) {=} \Phi_3$, *and* $\mathtt{update}(\Phi'_2, \Sigma'(x), A') {=} \Phi'_3$, *then*

$$(\Phi_2, \Phi'_2) \sqsubseteq (\Phi_3, {\Phi_3}')$$

5 Partial Closure Transformation

Since both strict and non-strict closure-conversion translates to the same strict target language, non-strict closure-conversion is essentially equivalent to a thunking transformation followed by strict closure-conversion. The switch in evaluation strategy effectively forces closure-conversion to be delayed until code generation, at the end of the compilation pipeline. Indeed, we see that closure-constructing is introduced during code generation phase of the Glasgow Haskell Compiler (GHC) [26]. If we wish to optimize the code introduced by lazy closure-conversion, then it would be better if it could be done in GHC's lazy core language wherein we already know how to do many kinds of optimizations.

Fortunately, introducing the strict pairs and closed functions needed for closure-conversion into a non-strict or lazy intermediate language is similar to the problem of adding unboxed types to Haskell, which must also be strictly evaluated. Peyton Jones and Launchbury [25] have already shown how to perform unboxing selectively during compilation through the worker/wrapper transformation. Therein, a worker computation on unboxed values is wrapped in a lazy interface after some strictness analyses which determines when unboxing is safe to do preemptively. This technique can apply to closure-conversion. For example, suppose that we have the following code that we wish to closure-convert *locally*:

$$\texttt{let}\ x = y + 1\ \texttt{in}\ x + x$$

We can introduce a worker $\$x$ that is the strict closure-converted form and replace x with a wrapper that knows how to evaluate the worker:

$$\begin{array}{l} \texttt{let}\ [\$x] = \texttt{pack}\ ((y), \lambda[(y)].\, y + 1)\ \texttt{in} \\ \quad \texttt{let}\ x = (\texttt{unpack}\ \$x\ \texttt{as}\ (e, f)\ \texttt{in}\ f[e])\ \texttt{in} \\ \quad\quad x + x \end{array}$$

The square brackets in $\texttt{let}\ [\$x] = \mathit{rhs}\ \texttt{in}\ \mathit{body}$ indicate that a strict binding, where *rhs* will be evaluated before binding the value to $\$x$ and evaluating *body*. In this instance, strictness is essential to capture the definition of y in the package before proceeding. The second binding instead creates a non-strict binding that points to it. Indeed, this technique works for both non-strict and lazy languages. Under call-by-name evaluation, this expression will place a closure for the unevaluated x in the environment, which gets recomputed every time x is forced. Under call-by-need evaluation, the non-strict `let` will allocate a memoized closure for x in the heap, which is evaluated at most once.

5.1 An IL for Partial Closure-Conversion

Syntax. In order to perform partial closure-conversion *within* our non-strict source $\mathcal{N}$, it must be extended with strict let-expressions, strict closed functions, and strict products and existentials. The different functions have different syntax for λ-expressions and applications. Like our let-expressions in the example above, strict functions and applications are marked with square brackets. The syntax of the extended language, $\mathcal{N}'$, is defined as:

$$\begin{array}{rcl} L, M, N \in \mathit{Exp} & ::= & n \mid x \mid \lambda x.\, M \mid M\ N \mid \texttt{let}\ x{=}M\ \texttt{in}\ N \\ & \mid & \lambda[x].\, M \mid M[N] \mid \texttt{let}\ [x]{=}M\ \texttt{in}\ N \\ & \mid & (M_0, \ldots, M_n) \\ & \mid & \texttt{case}\ x\ \texttt{of}\ (x_0, \ldots, x_n) \to N \\ & \mid & \texttt{pack}\ M \mid \texttt{unpack}\ M\ \texttt{as}\ x\ \texttt{in}\ N \end{array}$$

Type System. The main difference with the type system of Figure 1 is the presence of two different function types: The normal arrow ($\to$) denotes a lexically scoped non-strict function and the arrow ($\nrightarrow$) denotes a closed strict function. Figure 15 just shows the rules for these functions and their applications. Application typing requires that the correct kind of function is being applied.

$$\frac{\epsilon; x{:}\tau \vdash M : \sigma}{\Delta; \Gamma \vdash \lambda[x].\, M : \tau \nrightarrow \sigma}\ \textit{Closed SLam}$$

$$\frac{\Delta; \Gamma \vdash M : \tau \nrightarrow \sigma \quad \Delta; \Gamma \vdash N : \tau}{\Delta; \Gamma \vdash M[N] : \sigma}\ \textit{Closed SApp}$$

Figure 15. Strict and non-strict function types in $\mathcal{N}'$

$$\frac{\Sigma(x) = S}{\langle \Sigma \parallel x \rangle \Downarrow_{\mathcal{N}'} S}\ \textit{SVar} \qquad \frac{}{\langle \Sigma \parallel \lambda[x].\, M \rangle \Downarrow_{\mathcal{N}'} \lambda[x].\, M}\ \textit{SLam}$$

$$\frac{\begin{array}{c}\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{N}'} \lambda[x].\, L \\ \langle \Sigma \parallel N \rangle \Downarrow_{\mathcal{N}'} S \quad \langle \varepsilon, x \mapsto S \parallel L \rangle \Downarrow_{\mathcal{N}'} R\end{array}}{\langle \Sigma \parallel M[N] \rangle \Downarrow_{\mathcal{N}'} R}\ \textit{SApp}$$

Figure 16. Evaluation rules for strict functions in $\mathcal{N}'$

$$\begin{array}{llcl} (\textit{WWfun}) & \lambda x.\, M & \longrightarrow & \texttt{let}\ [\$f] = \texttt{pack}((y_0, \ldots, y_n), \\ & & & \qquad\qquad \lambda[((y_0, \ldots, y_n), x)].\, M) \\ & & & \texttt{in}\quad \lambda x.\, \texttt{unpack}\ \$f\ \texttt{as}\ (e, f)\ \texttt{in} \\ & & & \qquad\qquad f[(e, x)] \\ & & & \textbf{where}\ \mathrm{FV}(\lambda x.\, M) = y_0, \ldots, y_n \\ (\textit{WWarg}) & M\ N & \longrightarrow & \texttt{let}\ [\$x] = \texttt{pack}((y_0, \ldots, y_n), \\ & & & \qquad\qquad \lambda[(y_0, \ldots, y_n)].\, N) \\ & & & \texttt{in}\ M\ (\texttt{unpack}\ \$x\ \texttt{as}\ (x, y)\ \texttt{in}\ y[x]) \\ & & & \textbf{where}\ \mathrm{FV}(N) = y_0, \ldots, y_n \end{array}$$

Figure 17. Worker/Wrapper non-strict closure-conversion

Operational Semantics. Like the types and syntax, the sets for values and results for partial closure-conversion are the combination of the non-strict source and the strict target sets for values and results.

$$\begin{array}{rccl} C \in & \mathit{Config} & ::= & \langle \Sigma \parallel M \rangle \\ \Sigma \in & \mathit{Environment} & = & \mathit{Variable} \rightharpoonup \mathit{Value} \\ V, W \in & \mathit{Value} & ::= & S \mid P \\ S \in & \mathit{Strict\ Val} & ::= & n \mid \lambda[x].\, M \mid (S_0, \ldots, S_n) \\ & & & \mid \texttt{pack}\ S \\ P \in & \mathit{NonStrict\ Val} & ::= & (\Sigma, M) \\ R \in & \mathit{Result} & ::= & S \mid (\Sigma, \lambda x.\, M) \end{array}$$

The operational semantics for $\mathcal{N}'$ is a combination of strict and non-strict evaluation. We extend the semantics given in Figure 7 with the strict operational rules for pairs, packages and let-expressions of the form $\texttt{let}\ [x]{=}M\ \texttt{in}\ N$ (as given by the operational rules in Figure 4). Also strict functions and their applications follow a strict evaluation order. For clarity, we give the rules in Figure 16.

5.2 Partial Transformation

Since the transformation is local, we can specify it with two rewriting rules (Figure 17): one to add function closures and one to add thunk closures.

Type Preservation. By inspecting which rule is applied, we can show that the worker/wrapper transformation preserves types. And unlike the total transformation, we do not need to specify a type translation. The *WWfun* rule introduces a strict binding of type $Res_{\mathcal{N}}[\![\tau \to \sigma]\!]$ as a worker and then eliminates it in the wrapper, yielding the same output type as the input. The situation is similar for *WWarg*. If we let WW stand for both rules, we have:

Theorem 5.1 (Type Preservation)**.** *If* $\Gamma \vdash M : \tau$ *and* $\mathrm{WW} \vdash M \longrightarrow M'$*, then* $\Gamma \vdash M' : \tau$.

6 Related Work

Lambda-Lifting. The approach to handling free variables found in early compilers for lazy languages is *lambda-lifting* [4, 7, 12, 24]. Instead of capturing environments with products as in lazy closure-conversion, lambda-lifting β-expands out all of the free variables of functions leaving a partially-applied closed function in its place. Once in this form, the program can be executed on the G-machine [13]. Closure-conversion can be seen as taking lambda-lifting a step further by choosing a more concrete representation, *i.e.* products, for partially-applied closed functions.

Correctness of Closure-Conversion. Our work follows closely the approach of Minamide *et al.* [19] who show the use of existential types for strict closure-conversion type preservation and prove correctness with a family of type indexed logical relations. Our presentation differs in that our semantics is defined for pairs of environments and expressions, which must be carried through to our definitions of logical relations. Since the work of Minamide *et al.*, many notable theorems have been added to our understanding of call-by-value closure-conversion. These include the preservation of observational equivalence [2] and space safety [23].

Reasoning about Heaps. A common solution to reasoning about heaps is to use Kripke-esque or possible worlds approaches to logical relations [1, 11, 27], wherein the worlds are some notion of heap or store. Notably, Ahmed's dissertation [1] develops a logical relation specifically for reasoning about call-by-value programs with a mutable store. Her model fits the property of call-by-need that a value stored in the heap at one time is not necessarily the same as the one accessed later. However, her language differs from call-by-need in non-trivial ways. Firstly, a language with mutable references allows one to create cycles in the heap (the expression stored at r contains a reference to r):

$$\begin{gathered}\texttt{let}\ r = \texttt{ref}\ (\lambda x.\, x)\ \texttt{in} \\ \texttt{let}\ f = \lambda x.\, !r\ x\ \texttt{in} \\ r := f; !r\ 42\end{gathered}$$

Though cycles in the heap are possible in any language with general recursion, we study the simpler simply-typed call-by-need language which cannot create them. The cycles forced Ahmed to use a powerful technique known as *step-indexing* to give well-founded logical relations for her language. A second difference with this work is that the call-by-need mutable store is fundamentally different from that of a call-by-value language with mutable references. In the latter, the program itself manages the allocation and mutation of the values in the store, whereas in call-by-need all updates to the store are governed by the semantics of the language.

Reasoning about Call-by-Need. Recent work by Downen *et al.* [9] presents models of call-by-need that may be suitable for our purposes. However, their language semantics differs from ours in several ways: they reason about a reduction theory, they do not consider an explicit, separate heap and bound variables are maintained by the structure of the coterm or context. An explicit heap model exists for call-by-need classical realizability [20], but it cannot reason about updates to the heap. Both of these approaches are analogous to an ordered heap model wherein looking up variables in the heap allows us to split it into two. As we are focused on lazy languages for compilation and such a model does not match that of C or common garbage collectors, it is not entirely applicable. Fortunately, Mizuno and Sumii [21] define a notion of accessibility for explicit, unordered heaps for Launchbury's lazy semantics. Though their relation and semantics do not take the exact same form as ours, it appears as a promising fit for Conjecture 4.2.

7 Conclusion

Here we have shown how closure-conversion is not just for strict languages; it applies to non-strict ones, too. Our main insight is that the creation of closures must be done eagerly, to correctly capture the static environment, even for non-strict languages. Yet, a mixture of strict and non-strict functions allows us to locally closure-convert only parts of a program, instead of requiring a global transformation. We proved correctness of closure-conversion for call-by-name languages, and illustrated a heap-indexed logical relation for reasoning about correctness of call-by-need closure conversion. As future work, we intend to further develop the use of partial closure-conversion for optimization in the intermediate language of compilers for call-by-need languages. We also plan to elaborate the heap-based techniques for reasoning about effectful memoization of purely functional programs, which could be applicable to fully proving correctness of call-by-need closure conversion, as well as other program transformations that interact with memoization.

Acknowledgments

We thank Simon Peyton Jones and Philip Johnson-Freyd for their discussions and comments on this topic. Additionally, we would like to thank Eijiro Sumii and the anonymous reviewers for their comments on our early drafts. This work is supported by the National Science Foundation under Grant No. 1719158.

A Adequacy of Strict Closure-Conversion

Lemma A.1 (Weakening).
If $\langle\Sigma \| M\rangle \Downarrow_{\mathcal{V}'} R$, then $\langle\Sigma', \Sigma \| M\rangle \Downarrow_{\mathcal{V}'} R$.

Proof. By induction on the derivation of $\langle\Sigma \parallel M\rangle \Downarrow_{\mathcal{V}'} R$. □

Lemma A.2 (Strengthening). *If $\langle\Sigma, \Sigma' \parallel M\rangle \Downarrow_{\mathcal{V}'} R$ and $\mathrm{FV}(M) \cap \mathrm{dom}(\Sigma') = \emptyset$, then $\langle\Sigma \parallel M\rangle \Downarrow_{\mathcal{V}'} R$.*

Proof. By induction on the derivation of $\langle\Sigma, \Sigma' \parallel M\rangle \Downarrow_{\mathcal{V}'} R$. □

Lemma 2.2 (Adequacy). *If $\Gamma \vdash M : \tau$ and $(\Sigma, \Sigma') \in \mathcal{E}_{\mathcal{V}}[\![\Gamma]\!]$, then $(\langle\Sigma \parallel M\rangle, \langle\Sigma' \parallel \mathrm{CC}_{\mathcal{V}}[\![M]\!]\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau]\!]$.*

Proof. By induction on the typing derivation of $\Gamma \vdash M : \tau$, for a generic $(\Sigma, \Sigma') \in \mathcal{E}_{\mathcal{V}}[\![\Gamma]\!]$:

Case (*Num*) $\Gamma \vdash n : \mathtt{int}$.
So $M = n$, $\mathrm{CC}_{\mathcal{V}}[\![n]\!] = n$, and we must show that $(\langle\Sigma \parallel n\rangle, \langle\Sigma' \parallel n\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\mathtt{int}]\!]$.
The only evaluations are $\langle\Sigma \parallel n\rangle \Downarrow_{\mathcal{V}} n$ and $\langle\Sigma' \parallel n\rangle \Downarrow_{\mathcal{V}'} n$.
We have $(n, n) \in \mathcal{V}_{\mathcal{V}}[\![\mathtt{int}]\!]$ by definition of $\mathcal{V}_{\mathcal{V}}[\![\mathtt{int}]\!]$.
Therefore, $(\langle\Sigma \parallel n\rangle, \langle\Sigma' \parallel n\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\mathtt{int}]\!]$.

Case (*Var*) $\Gamma \vdash x : \tau$ because $(x : \tau) \in \Gamma$.
So $M = x$, $\mathrm{CC}_{\mathcal{V}}[\![x]\!] = x$, and we must show that $(\langle\Sigma \parallel x\rangle, \langle\Sigma' \parallel x\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau]\!]$.
The only evaluations are $\langle\Sigma \parallel x\rangle \Downarrow_{\mathcal{V}} \Sigma(x)$ and $\langle\Sigma' \parallel x\rangle \Downarrow_{\mathcal{V}'} \Sigma'(x)$.
From the assumptions $(\Sigma, \Sigma') \in \mathcal{E}_{\mathcal{V}}[\![\Gamma]\!]$ and $(x : \tau) \in \Gamma$, we know $(\Sigma(x), \Sigma'(x)) \in \mathcal{V}_{\mathcal{V}}[\![\tau]\!]$ by definition of $\mathcal{E}_{\mathcal{V}}[\![\Gamma]\!]$.
Therefore, $(\langle\Sigma \parallel x\rangle, \langle\Sigma' \parallel x\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau]\!]$ by the definition of $\mathcal{M}_{\mathcal{V}}$.

Case (Lam) $\Gamma \vdash \lambda x.N : \tau_1 \to \tau_2$ because $\Gamma, x : \tau_1 \vdash N : \tau_2$.
So $\tau = \tau_1 \to \tau_2$, $M = \lambda x.N$, and

$$\begin{aligned}\mathrm{CC}_{\mathcal{V}}[\![\lambda x.N]\!] &= \mathtt{pack}(V_e', V_f') \\ V_e' &= (y_0, \ldots, y_n) \\ V_f' &= \lambda((y_0 \ldots y_n), x).\, \mathrm{CC}_{\mathcal{V}}[\![N]\!])\end{aligned}$$

where $\mathrm{FV}(\lambda x.\, N) = y_0, \ldots, y_n$. We must show that

$$(\langle\Sigma \parallel \lambda x.N\rangle, \langle\Sigma' \parallel \mathrm{CC}_{\mathcal{V}}[\![\lambda x.N]\!]\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau_1 \to \tau_2]\!]$$

The unique evaluations are $\langle\Sigma \parallel \lambda x.N\rangle \Downarrow_{\mathcal{V}} (\Sigma, \lambda x.N)$ and $\langle\Sigma' \parallel \mathrm{CC}_{\mathcal{V}}[\![\lambda x.N]\!]\rangle \Downarrow_{\mathcal{V}'} \mathrm{CC}_{\mathcal{V}}[\![\lambda x.N]\!][\Sigma']$.
It suffices to show that $((\Sigma, \lambda x.N), \mathrm{CC}_{\mathcal{V}}[\![\lambda x.N]\!][\Sigma']) \in \mathcal{V}_{\mathcal{V}}[\![\tau_1 \to \tau_2]\!]$.
Suppose an arbitrary $(W, W') \in \mathcal{V}_{\mathcal{V}}[\![\tau]\!]$, and note that $((\Sigma, x \mapsto W), (\Sigma', x \mapsto W')) \in \mathcal{E}_{\mathcal{V}}[\![\Gamma, x : \tau_1]\!]$ by definition of $\mathcal{E}_{\mathcal{V}}$ and the assumption $(\Sigma, \Sigma') \in \mathcal{E}_{\mathcal{V}}[\![\Gamma]\!]$.
From the inductive hypothesis on $\Gamma, x : \tau_1 \vdash N : \tau_2$, we know $(\langle\Sigma, x \mapsto W \parallel N\rangle, \langle\Sigma', x \mapsto W' \parallel \mathrm{CC}_{\mathcal{V}}[\![N]\!]\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau_2]\!]$.
Assuming $\langle\Sigma, x \mapsto W \parallel N\rangle \Downarrow_{\mathcal{V}} R$, there must be a $(R, R') \in \mathcal{V}_{\mathcal{V}}[\![\tau_2]\!]$ such that $\langle\Sigma', x \mapsto W' \parallel \mathrm{CC}_{\mathcal{V}}[\![N]\!]\rangle \Downarrow_{\mathcal{V}'} R'$ by the definition of $\mathcal{M}_{\mathcal{V}}$.
Expanding, $\langle\varepsilon \parallel V_f'\,((\Sigma'(y_0), \ldots, \Sigma'(y_n)), W')\rangle \Downarrow_{\mathcal{V}'} R'$ as well by strengthening (Lemma A.2) the evaluation $\langle\Sigma', x \mapsto W' \parallel \mathrm{CC}_{\mathcal{V}}[\![N]\!]\rangle \Downarrow_{\mathcal{V}'} R'$.
Therefore, $((\Sigma, \lambda x.N), \mathrm{CC}_{\mathcal{V}}[\![\lambda x.N]\!][\Sigma'])$ is in the relation $\mathcal{V}_{\mathcal{V}}[\![\tau_1 \to \tau_2]\!]$ by definition of $\mathcal{V}_{\mathcal{V}}$, and thus $(\langle\Sigma \parallel \lambda x.N\rangle, \langle\Sigma' \parallel \mathrm{CC}_{\mathcal{V}}[\![\lambda x.N]\!]\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau_1 \to \tau_2]\!]$.

Case (App) $\Gamma \vdash N\ O : \tau$ because $\Gamma \vdash N : \tau' \to \tau$ and $\Gamma \vdash O : \tau'$.
So $M = N\ O$, $\mathrm{CC}_{\mathcal{V}}[\![N\ O]\!] = \mathtt{call}(\mathrm{CC}_{\mathcal{V}}[\![N]\!], \mathrm{CC}_{\mathcal{V}}[\![O]\!])$, and we must show that $(\langle\Sigma \parallel N\ O\rangle, \langle\Sigma' \parallel \mathrm{CC}_{\mathcal{V}}[\![N\ O]\!]\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau]\!]$.
Suppose that $\langle\Sigma \parallel N\ O\rangle \Downarrow_{\mathcal{V}} R$. The conclusion of that derivation must be an instance of *App* by inversion, which gives us

1. $\langle\Sigma \parallel N\rangle \Downarrow_{\mathcal{V}} (\Sigma_1, \lambda x.L)$
2. $\langle\Sigma \parallel O\rangle \Downarrow_{\mathcal{V}} W$
3. $\langle\Sigma_1, x \mapsto W \parallel L\rangle \Downarrow_{\mathcal{V}} R$

From the first inductive hypothesis, we know $(\langle\Sigma \parallel N\rangle, \langle\Sigma' \parallel \mathrm{CC}_{\mathcal{V}}[\![N]\!]\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau' \to \tau]\!]$.
It follows that there is a $((\Sigma_1, \lambda x.L), \mathtt{pack}(V_e', V_f')) \in \mathcal{V}_{\mathcal{V}}[\![\tau' \to \tau]\!]$ such that $\langle\Sigma' \| \mathrm{CC}_{\mathcal{V}}[\![N]\!]\rangle \Downarrow_{\mathcal{V}'} \mathtt{pack}(V_e', V_f')$ by definition of $\mathcal{M}_{\mathcal{V}}$.
From the second inductive hypothesis, we know $(\langle\Sigma \parallel O\rangle, \langle\Sigma' \parallel \mathrm{CC}_{\mathcal{V}}[\![O]\!]\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau']\!]$. Likewise, it follows that there is a $\langle\Sigma' \parallel \mathrm{CC}_{\mathcal{V}}[\![O]\!]\rangle \Downarrow_{\mathcal{V}'} W'$ such that $(W, W') \in \mathcal{V}_{\mathcal{V}}[\![\tau']\!]$.
We also have $(\langle\Sigma, x \mapsto W \parallel L\rangle, \langle\varepsilon \parallel V_f'(V_e', W')\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau]\!]$, from the definition of $((\Sigma_1, \lambda x.L), \mathtt{pack}(V_e', V_f')) \in \mathcal{V}_{\mathcal{V}}[\![\tau' \to \tau]\!]$. It follows that there is a $(R, R') \in \mathcal{V}_{\mathcal{V}}[\![\tau]\!]$ such that $\langle\varepsilon \parallel V_f'(V_e', W')\rangle \Downarrow_{\mathcal{V}'} R'$.
Expanding, we get that $\langle\Sigma' \parallel \mathtt{call}(\mathrm{CC}_{\mathcal{V}}[\![N]\!], \mathrm{CC}_{\mathcal{V}}[\![O]\!])\rangle \Downarrow_{\mathcal{V}'} R'$ as well by weakening (Lemma A.1) $\langle\varepsilon \parallel V_f'(V_e', W')\rangle \Downarrow_{\mathcal{V}'} R'$ to $\langle\Sigma' \parallel V_f'(V_e', W')\rangle \Downarrow_{\mathcal{V}'} R'$.
Therefore, $(\langle\Sigma \parallel N\ O\rangle, \langle\Sigma' \parallel \mathrm{CC}_{\mathcal{V}}[\![N\ O]\!]\rangle)$ is in the relation $\mathcal{M}_{\mathcal{V}}[\![\tau]\!]$ by definition of $\mathcal{M}_{\mathcal{V}}$.

Case (Let) $\Gamma \vdash \mathtt{let}\ x = N\ \mathtt{in}\ O : \tau$ because $\Gamma \vdash N : \tau'$ and $\Gamma, x : \tau' \vdash O : \tau$.
So $M = \mathtt{let}\ x = N\ \mathtt{in}\ O$ and

$$\begin{aligned}\mathrm{CC}_{\mathcal{V}}[\![\mathtt{let}\ x = N\ \mathtt{in}\ O]\!] = {} & \mathtt{let}\ x = \mathrm{CC}_{\mathcal{V}}[\![N]\!] \\ & \mathtt{in}\ \mathrm{CC}_{\mathcal{V}}[\![O]\!]\end{aligned}$$

We must show that

$$(\langle \Sigma \parallel \mathtt{let}\ x = N\ \mathtt{in}\ O\rangle \\ , \langle \Sigma' \parallel CC_{\mathcal{V}}[\![\mathtt{let}\ x = N\ \mathtt{in}\ O]\!]\rangle) \\ \in \mathcal{M}_{\mathcal{V}}[\![\tau]\!]$$

Suppose that $\langle \Sigma \parallel \mathtt{let}\ x = N\ \mathtt{in}\ O\rangle \Downarrow_{\mathcal{V}} R$. The conclusion of that derivation must be an instance of *Let* by inversion, which gives us

1. $\langle \Sigma \parallel N\rangle \Downarrow_{\mathcal{V}} W$
2. $\langle \Sigma, x \mapsto W \parallel O\rangle \Downarrow_{\mathcal{V}} R$

From the first inductive hypothesis, we know $(\langle \Sigma \parallel N\rangle, \langle \Sigma' \parallel CC_{\mathcal{V}}[\![N]\!]\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau']\!]$. It follows that there is a $\langle \Sigma' \parallel CC_{\mathcal{V}}[\![N]\!]\rangle \Downarrow_{\mathcal{V}} W'$ such that $(W, W') \in \mathcal{V}_{\mathcal{V}}[\![\tau']\!]$. We also know that $(\Sigma, x \mapsto W, \Sigma, x \mapsto W') \in \mathcal{E}_{\mathcal{V}}[\![\Gamma, x{:}\tau']\!]$ by the definition of $\mathcal{E}_{\mathcal{V}}$ and the assumption $(\Sigma, \Sigma) \in \mathcal{E}_{\mathcal{V}}[\![\Gamma]\!]$.
From the second inductive hypothesis, we know that $(\langle \Sigma, x \mapsto W \parallel O\rangle, \langle \Sigma', x \mapsto W' \parallel CC_{\mathcal{V}}[\![O]\!]\rangle)$ is in $\mathcal{M}_{\mathcal{V}}[\![\tau']\!]$. It follows that there is a $\langle \Sigma', x \mapsto W' \parallel CC_{\mathcal{V}}[\![O]\!]\rangle \Downarrow_{\mathcal{V}'} R'$ such that $(R, R') \in \mathcal{V}_{\mathcal{V}}[\![\tau]\!]$.
Therefore,

$$(\langle \Sigma \parallel \mathtt{let}\ x = N\ \mathtt{in}\ O\rangle \\ , \langle \Sigma' \parallel CC_{\mathcal{V}}[\![\mathtt{let}\ x = N\ \mathtt{in}\ O]\!]\rangle) \\ \in \mathcal{M}_{\mathcal{V}}[\![\tau]\!] \qquad \square$$

B Adequacy of Non-strict Closure-Conversion

Lemma 3.2 (Adequacy). *If* $\Gamma \vdash M : \tau$ *and* $(\Sigma, \Sigma') \in \mathcal{E}_{\mathcal{N}}[\![\Gamma]\!]$, *then* $(\langle \Sigma \parallel M\rangle, \langle \Sigma' \parallel CC_{\mathcal{N}}[\![M]\!]\rangle) \in \mathcal{M}_{\mathcal{N}}[\![\tau]\!]$.

Proof. By induction on the typing derivation of $\Gamma \vdash M : \tau$, for a generic $(\Sigma, \Sigma') \in \mathcal{E}_{\mathcal{N}}[\![\Gamma]\!]$. The cases for *Num* and *Lam* are analogous to Lemma 2.2. The remaining two cases are:

Case (*Var*) $\Gamma \vdash x : \tau$ because $(x : \tau) \in \Gamma$.
So $M = x$, $CC_{\mathcal{N}}[\![x]\!] = \mathtt{eval}\ x$, and we must show that $(\langle \Sigma \parallel x\rangle, \langle \Sigma' \parallel CC_{\mathcal{N}}[\![x]\!]\rangle) \in \mathcal{M}_{\mathcal{N}}[\![\tau]\!]$.
From the assumptions $(\Sigma, \Sigma') \in \mathcal{E}_{\mathcal{N}}[\![\Gamma]\!]$ and $(x : \tau) \in \Gamma$, we know $(\Sigma(x), \Sigma'(x)) \in \mathcal{V}_{\mathcal{N}}[\![\tau]\!]$ by definition of $\mathcal{E}_{\mathcal{N}}[\![\Gamma]\!]$. Furthermore, the definition of $\mathcal{V}_{\mathcal{N}}$ forces $\Sigma(x) = (\Sigma_1, M)$ and $\Sigma'(x) = \mathtt{pack}(V'_e, V'_f)$, such that $(\langle \Sigma_1 \parallel M\rangle, \langle \varepsilon \parallel V'_e\ V'_f\rangle) \in \mathcal{M}_{\mathcal{N}}[\![\tau]\!]$.
Assume the source evaluation $\langle \Sigma \parallel x\rangle \Downarrow_{\mathcal{N}} R$. By inversion on this derivation, $\langle \Sigma_1 \parallel M\rangle \Downarrow_{\mathcal{N}} R$.
We are guaranteed related results $(R, R') \in \mathcal{R}_{\mathcal{N}}[\![\tau]\!]$ such that $\langle \varepsilon \parallel V'_f\ V'_e\rangle \Downarrow_{\mathcal{V}'} R'$ from the definition of $(\langle \Sigma_1 \parallel M\rangle, \langle \varepsilon \parallel V'_f\ V'_e\rangle) \in \mathcal{M}_{\mathcal{N}}[\![\tau]\!]$.
Expanding, we have $\langle \Sigma' \parallel CC_{\mathcal{N}}[\![x]\!]\rangle \Downarrow_{\mathcal{V}'} R'$ from the above evaluation.
Therefore, $(\langle \Sigma \parallel x\rangle, \langle \Sigma' \parallel CC_{\mathcal{N}}[\![x]\!]\rangle) \in \mathcal{M}_{\mathcal{N}}[\![\tau]\!]$ by the definition of $\mathcal{M}_{\mathcal{N}}$.

Case (App) $\Gamma \vdash N\ O : \tau$ because $\Gamma \vdash N : \tau' \to \tau$ and $\Gamma \vdash O : \tau'$.
So $M = N\ O$, $FV(O) = \{y_0, \ldots, y_n\}$,

$$\begin{aligned} CC_{\mathcal{N}}[\![N\ O]\!] &= \mathtt{let}\ x = W' \\ &\quad \mathtt{in}\ \mathtt{call}(CC_{\mathcal{N}}[\![N]\!], x) \\ W' &= \mathtt{pack}(W'_e, W'_f) \\ W'_f &= \lambda(y_0, \ldots, y_n).CC_{\mathcal{N}}[\![O]\!] \\ W'_e &= (\Sigma'(y_0), \ldots, \Sigma'(y_n)) \end{aligned}$$

and we must show that $\mathcal{M}_{\mathcal{N}}[\![\tau]\!]$ contains $(\langle \Sigma \parallel N\ O\rangle, \langle \Sigma' \parallel CC_{\mathcal{N}}[\![N\ O]\!]\rangle)$.
Suppose that $\langle \Sigma \parallel N\ O\rangle \Downarrow_{\mathcal{N}} R$. The conclusion of that derivation must be an instance of *App* by inversion, which gives us

1. $\langle \Sigma \parallel N\rangle \Downarrow_{\mathcal{N}} (\Sigma_1, \lambda x.L)$
2. $\langle \Sigma_1, x \mapsto (\Sigma, O) \parallel L\rangle \Downarrow_{\mathcal{N}} R$

From the first inductive hypothesis, we know $(\langle \Sigma \parallel N\rangle, \langle \Sigma' \parallel CC_{\mathcal{N}}[\![N]\!]\rangle) \in \mathcal{M}_{\mathcal{N}}[\![\tau' \to \tau]\!]$. It follows that there is a $((\Sigma_1, \lambda x.L), \mathtt{pack}(V'_e, V'_f))$ in the relation $\mathcal{R}_{\mathcal{N}}[\![\tau' \to \tau]\!]$ such that $\langle \Sigma' \parallel CC_{\mathcal{N}}[\![N]\!]\rangle \Downarrow_{\mathcal{V}'} \mathtt{pack}(V'_e, V'_f)$ by definition of $\mathcal{M}_{\mathcal{N}}$.
From the second inductive hypothesis, we know $(\langle \Sigma \parallel O\rangle, \langle \Sigma' \parallel CC_{\mathcal{N}}[\![O]\!]\rangle) \in \mathcal{M}_{\mathcal{N}}[\![\tau']\!]$.
Note that $\langle \varepsilon \parallel W'_f\ W'_e\rangle \Downarrow_{\mathcal{V}'} R'_1$ if $\langle \Sigma' \parallel CC_{\mathcal{N}}[\![O]\!]\rangle \Downarrow_{\mathcal{V}'} R'_1$ by strengthening (Lemma A.2).
It follows that $(\langle \Sigma \parallel N\rangle, \langle \varepsilon \parallel W'_f\ W'_e\rangle) \in \mathcal{M}_{\mathcal{N}}[\![\tau]\!]$ and so $((\Sigma, O), W') \in \mathcal{V}_{\mathcal{N}}[\![\tau']\!]$ by definitions of $\mathcal{M}_{\mathcal{N}}$ and $\mathcal{V}_{\mathcal{N}}$.
From $((\Sigma_1, \lambda x.L), \mathtt{pack}(V'_e, V'_f)) \in \mathcal{R}_{\mathcal{N}}[\![\tau' \to \tau]\!]$ and $((\Sigma, O), W') \in \mathcal{V}_{\mathcal{N}}[\![\tau']\!]$, we know

$$(\langle \Sigma_1, x \mapsto (\Sigma, O) \parallel L\rangle, \langle \varepsilon \parallel V'_f\ (V'_e, W')\rangle) \in \mathcal{M}_{\mathcal{N}}[\![\tau]\!]$$

This gives $(R, R') \in \mathcal{R}_{\mathcal{N}}[\![\tau]\!]$ such that $\langle \varepsilon \parallel V'_f\ (V'_e, W')\rangle) \Downarrow_{\mathcal{V}'} R'$; and likewise, we know from the weakening lemma (Lemma A.1) that $\langle \Sigma' \parallel V'_f\ (V'_e, W')\rangle) \Downarrow_{\mathcal{V}'} R'$.
Expanding $\langle \Sigma' \parallel V'_f\ (V'_e, W')\rangle) \Downarrow_{\mathcal{V}'} R'$, we have $\langle \Sigma' \parallel CC_{\mathcal{N}}[\![N\ O]\!]\rangle) \Downarrow_{\mathcal{V}'} R'$.
Therefore, $(\langle \Sigma \parallel N\ O\rangle, \langle \Sigma' \parallel CC_{\mathcal{N}}[\![N\ O]\!]\rangle)$ is in $\mathcal{M}_{\mathcal{N}}[\![\tau]\!]$ by definition of $\mathcal{M}_{\mathcal{N}}$.

Case (Let) $\Gamma \vdash \mathtt{let}\ x = N\ \mathtt{in}\ O : \tau$ because $\Gamma \vdash N : \tau'$ and $\Gamma, x : \tau' \vdash O : \tau$.
So $M = \mathtt{let}\ x = N\ \mathtt{in}\ O$, $FV(N) = \{y_0, \ldots, y_n\}$,and

$$\begin{aligned} CC_{\mathcal{V}}[\![\mathtt{let}\ x = N\ \mathtt{in}\ O]\!] &= \mathtt{let}\ x = W' \\ &\quad \mathtt{in}\ CC_{\mathcal{N}}[\![O]\!] \\ W' &= \mathtt{pack}(W'_e, W'_f) \\ W'_f &= \lambda(y_0, \ldots, y_n).CC_{\mathcal{N}}[\![N]\!] \\ W'_e &= (\Sigma'(y_0), \ldots, \Sigma'(y_n)) \end{aligned}$$

We must show that

$$\begin{aligned}&(\langle \Sigma \parallel \texttt{let}\ x = N\ \texttt{in}\ O\rangle\\&,\langle \Sigma' \parallel CC_{\mathcal{N}}[\![\texttt{let}\ x = N\ \texttt{in}\ O]\!]\rangle)\\&\qquad \in \mathcal{M}_{\mathcal{N}}[\![\tau]\!]\end{aligned}$$

Suppose that $\langle \Sigma \parallel \texttt{let}\ x = N\ \texttt{in}\ O\rangle \Downarrow_{\mathcal{N}} R$. The conclusion of that derivation must be an instance of *Let* by inversion, which gives us $\langle \Sigma, x \mapsto (\Sigma, N) \parallel O\rangle \Downarrow_{\mathcal{N}} R$
From the first inductive hypothesis, we know $(\langle \Sigma \parallel N\rangle, \langle \Sigma' \parallel CC_{\mathcal{N}}[\![N]\!]\rangle) \in \mathcal{M}_{\mathcal{N}}[\![\tau']\!]$. From the definition of $\mathcal{M}_{\mathcal{N}}$, it follows that $\langle \Sigma' \parallel CC_{\mathcal{N}}[\![N]\!]\rangle \Downarrow_{\mathcal{N}} R'$ such that $(R, R') \in \mathcal{R}_{\mathcal{N}}[\![\tau]\!]$.
Note that $\langle \varepsilon \parallel W_f'\ W_e'\rangle \Downarrow_{\mathcal{N}} R'$ as well by strengthening (Lemma A.2) $\langle \Sigma' \parallel CC_{\mathcal{N}}[\![N]\!]\rangle \Downarrow_{\mathcal{N}} R'$. From this and the definition of $\mathcal{V}_{\mathcal{N}}$, it follows that $((\Sigma, N), W') \in \mathcal{V}_{\mathcal{N}}[\![\tau']\!]$. Thus, $(\Sigma, x \mapsto (\Sigma, N), \Sigma, x \mapsto W')$ is in the relation $\mathcal{E}_{\mathcal{N}}[\![\Gamma, x{:}\tau']\!]$ by the definition of $\mathcal{E}_{\mathcal{N}}$ and the assumption $(\Sigma, \Sigma) \in \mathcal{E}_{\mathcal{N}}[\![\Gamma]\!]$.
From the second inductive hypothesis, we know that $(\langle \Sigma, x \mapsto (\Sigma, N) \parallel O\rangle, \langle \Sigma', x \mapsto W' \parallel CC_{\mathcal{N}}[\![O]\!]\rangle) \in \mathcal{M}_{\mathcal{V}}[\![\tau']\!]$. It follows that there is a $\langle \Sigma', x \mapsto W' \parallel CC_{\mathcal{N}}[\![O]\!]\rangle \Downarrow_{\mathcal{N}} R'$ such that $(R, R') \in \mathcal{R}_{\mathcal{N}}[\![\tau]\!]$.
Therefore, using *Let* reduction in the target

$$\begin{aligned}&(\langle \Sigma \parallel \texttt{let}\ x = N\ \texttt{in}\ O\rangle\\&,\langle \Sigma' \parallel CC_{\mathcal{N}}[\![\texttt{let}\ x = N\ \texttt{in}\ O]\!]\rangle)\\&\qquad \in \mathcal{M}_{\mathcal{N}}[\![\tau]\!] \qquad \square\end{aligned}$$

References

[1] Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[2] 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.

[3] 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.

[4] Lennart Augustsson. A compiler for lazy ml. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, 1984.

[5] Lennart Augustsson. Compiling pattern matching. In *Proceedings Of a Conference on Functional Programming Languages and Computer Architecture*, pages 368–381, 1985.

[6] 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.

[7] Olivier Danvy. An extensional characterization of lambda-lifting and lambda-dropping. In *Functional and Logic Programming, 4th Fuji International Symposium, FLOPS'99, Tsukuba, Japan, November 11-13, 1999, Proceedings*, pages 241–250, 1999.

[8] Paul Downen, Zena M. Ariola, Simon Peyton Jones, and Richard A. Eisenberg. Kinds are calling conventions. *Proc. ACM Program. Lang.*, 4(ICFP):104:1–104:29, 2020.

[9] 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.

[10] Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. Codata in action. 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 119–146, 2019.

[11] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.*, 22(4-5):477–528, 2012.

[12] John Hughes. *The Design and Implementation of Programming languages*. PhD thesis, University of Oxford, 1983.

[13] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, Montreal, Canada, June 17-22, 1984*, pages 58–69, 1984.

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

[15] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Masayuki Mizuno and Eijiro Sumii. Formal verifications of call-by-need and call-by-name evaluations with mutual recursion. In Anthony Widjaja Lin, editor, *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*, volume 11893 of *Lecture Notes in Computer Science*, pages 181–201, 2019.

[22] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. *LISP Symb. Comput.*, 7(1):57–82, 1994.

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

[24] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[25] 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.

[26] 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.

[27] Andrew M. Pitts. Reasoning about local variables with operationally-based logical relations. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 152–163. IEEE Computer Society, 1996.

[28] 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.

[29] 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.