# Deriving Practical Implementations of First-Class Functions

ZACHARY J. SULLIVAN

This document explores the ways in which a practical implementation is derived from Church's calculus of $\lambda$-conversion, *i.e.* the core of functional programming languages today like Haskell, Ocaml, Racket, and even Javascript. Despite the implementations being based on the calculus, the languages and their semantics that are used in compilers today vary greatly from their original inspiration. I show how incremental changes to Church's semantics yield the intermediate theories of explicit substitutions, operational semantics, abstract machines, and compilation through intermediate languages, *e.g.* CPS, on the way to what is seen in modern functional language implementations. Throughout, a particular focus is given to the effect of evaluation strategy which separates Haskell and its implementation from the other languages listed above.

## 1 INTRODUCTION

The gap between theory and practice can be a large one, both in terms of the gap in time between the creation of a theory and its implementation and the appearance of ideas in one versus the other. The difference in time between the presentation of Church's $\lambda$-calculus—*i.e.* the core of functional programming languages today like Haskell, Ocaml, Racket—and its mechanization was a period of 20 years. In the following 60 years till now, there has been further improvements in both the efficiency of the implementation, but also in reasoning about how implementations are related to the original calculus. To compute in the calculus is flexible (even non-deterministic), but real machines are both more complex and require restrictions on the allowable computation to be tractable. Specifically in the $\lambda$-calculus, computation can be done anywhere at the whim of the theorist whereas compilers often generate code for a single execution sequence on a modern machine executing x86 instructions. This document explores some ways in which a practical implementation can be derived from the $\lambda$-calculus. That is, a series of small changes are made to the semantics of the language until a practical implementation is arrived at.

An aspect of the development of the implementation of function programming languages that I pay special attention to is evaluation strategy, which decides the order and manner in which programs are executed. It has had a large impact on implementations as functional compilers have indeed been devoted to one strategy or another; the most dominant being call-by-value and call-by-need. Throughout this document, I will describe these approaches side-by-side to tease out their similarities and differences. In the past, it has been common to see a researcher spend their time working entirely on compilers and theory of a single strategy resulting in duplicated work. Today, however, the work of the two dominant practical evaluation strategies are converging.

This document does not necessarily follow the historical development of the modern implementation, rather it is focused on how the relationship of the theories themselves. Indeed, I begin with Church's calculus which is historically a starting point for studying the theory and implementation of first class functions. However, there are several points herein where a later development is projected backwards in history to reflect our, *i.e.* those who study language implementation, current understanding of these programming languages.

To begin, Section 2 introduces original presentation of first-class functions given as a calculus of $\lambda$-conversion. This is a reduction theory for converting one $\Lambda$-term into another. I introduce the three different evaluation strategies most discussed in the literature. As a bridge to later theories, I show modifications of the base calculi which ease the handling of free variables (De Bruijn notation) and which perform a more efficient substitution procedure (explicit substitutions). These reduction theories are good for reasoning, but their non-determinism is undesirable in practice. Therefore, the following section, Section 3, presents a restriction of the reduction theory that makes evaluation

deterministic; this is known as operational semantics. I present multiple ways of representing this sort of evaluation mechanism; some act as stateful manipulation of a term while others present evaluation to a result as a single recursive procedure. Next, I present combinators (Section 4) and their abstract machines. Whereas the previous two sections are focused on specification of a theory based on $\Lambda$-terms alone, this section shows how a combinatory algebra can be used to build machines for evaluating programs. The following section, Section 5, examines abstract machines not built for combinators. These were historically some of the first implementations of the $\lambda$-calculus, and already, there are large differences in the machines for different evaluation strategies. Many of today's functional language compilers use neither of these styles of abstract machines, instead they compile the source language through a series of intermediate languages. Therefore, I present, in Section 6, a series of intermediate languages that address different aspects of functional languages that need to be handled before execution on real hardware. For instance, continuation-passing-style is used in call-by-value languages in order to offer more ways to optimize these programs. This document, in the end, presents many ways to represent computation with first-class functions, but this is not obvious. Thus, Section 7 caps off the discussion of these implementation techniques by exploring the reasoning frameworks that have developed alongside them. Such reasoning can show what properties are preserved and/or reflected from the original theory.

In the end, this document does not claim one style of semantics better than another, rather it supplies some understanding about which style to use depending on a specific implementation context: whether that context needs the flexibility of a calculus, the optimization pipelines offered by compilation through intermediate languages, or something in between. Additionally, the document describes how a source language's evaluation strategy influences such a choice of semantics; however, the ultimate hope is that future compiler writers begin to combine different evaluation strategies into a single compiler.

## 2 REDUCTION THEORY

Church's $\lambda$-calculus [14, 17]—which like a Turing machine is capable of specifying computable functions—serves as the foundation for functional programming languages.[1] Its core syntax, which I will refer to as $\Lambda$-terms, is given by the following grammar:

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

where $x$ is a variable, $\lambda x.\, M$ is a function (also commonly referred to as an abstraction) that binds a variable $x$ in the expression $M$, and $M\ N$ a function application. For example, the function $f(x) = x + 1$, which is written in a common mathematical notation, is therefore written in the $\lambda$-calculus as the anonymous function $\lambda x.\, x + 1$.

There are several ways to give a semantics, or meaning, to this small language of expressions including reduction theories, natural semantics, type theories, mathematical semantics (*i.e.* denotational semantics), and machine semantics. Since Church's seminal work specifies a reduction theory, I treat that style of semantics as foundational and will begin by describing them. However, I will discuss some of the other forms of semantics and their relationship to each other throughout this document. Generally, a *reduction theory* specifies what it means for one expression to reduce to another. When discussing numeric expressions, for instance, it is common to say that $1 + 2$ reduces to 3 by the rules of addition. This can be written as the following reduction rule (the right-hand-side of the "**where**" is a meta-language operation):

$$(+) \qquad n + m \quad \longrightarrow \quad c \quad \textbf{where } n + m = c$$

[1] Scala is an exception here, *i.e.* it is considered a functional programming language without $\lambda$-calculus as its core.

For expressions in the $\lambda$-calculus, defining a reduction theory would require stating the rules for function application which is the only $\lambda$-calculus operation. There is not only one way to reduce a function application; and therefore, this section presents the three theories that are used most: *call-by-name*, *call-by-value*, and *call-by-need* as presented by Church [17], Plotkin [69], and Ariola [11], respectively. I discuss how these reduction theories differ in their observable output and usage of meta-language. And after presenting the theories themselves, I discuss how the manner of their specification, *i.e.* the meta-language, has implications for practical implementations of functional programming languages.

Before presenting the reduction theories, I must define two auxiliary functions (to be used therein) over expressions of $\Lambda$: one to provide the set of free variables of an expression and another to say what it means to substitute some expression for a free variable. The free variables of an expression are a set of variables defined inductively over the expression in the following way:

$$\begin{array}{rcl} \mathrm{FV}(x) & = & \{x\} \\ \mathrm{FV}(\lambda x.\, M) & = & \mathrm{FV}(M) - \{x\} \\ \mathrm{FV}(M\ N) & = & \mathrm{FV}(M) \cup \mathrm{FV}(N) \end{array}$$

For example, the free variables of the expression $\lambda x.\, x\ y\ z$ are $y$ and $z$ since they are not bound by any $\lambda$-expression, whereas $x$ is not free since it is bound by the $\lambda x. \ldots$ part of the expression. The second auxiliary function, substitution, relies on this definition of free variables. Denoted $M[N/x]$, a substitution says "for every *free* occurrence of the variable $x$ in the expression $M$, the expression $N$ is substituted". It is defined inductively over the expression $M$:

$$\begin{array}{rcll} x[N/x] & = & N & \\ y[N/x] & = & y & \\ (\lambda x.\, M)[N/x] & = & \lambda x.\, M & \\ (\lambda y.\, M)[N/x] & = & \lambda y.\, M[N/x] & \textbf{where } y \notin \mathrm{FV}(N) \\ (M\ N)[L/x] & = & M[L/x]\ N[L/x] & \end{array}$$

Note the abstraction rules. The first stops a substitution for the variable $x$ upon encountering a $\lambda$-expression binding $x$, thereby any variable is treated as if it is bound to the closest $\lambda x. \ldots$ that binds it. The second abstraction rule is what makes this definition of substitution *capture-free*. If the expression $N$ contains a free occurrence of the variable bound by the $\lambda$-expression and it is substituted in the body of the expression, then it would no longer be free after substitution. For example, the variables $x$ is incorrectly captured in the following substitution $(\lambda x.\, y)[x/y] = \lambda x.\, x$.

### 2.1 Call-by-Name

In Church's theory, which became known as call-by-name reduction, a $\Lambda$-term *reduces* to another with the following reduction rule:

$$\begin{array}{lrcll} (\alpha) & \lambda x.\, M & \longrightarrow & \lambda y.\, M[y/x] & \textbf{where } y \notin \mathrm{FV}(M) \\ (\beta) & (\lambda x.\, M)\ N & \longrightarrow & M[N/x] & \\ (\eta) & \lambda x.\, M\ x & \longrightarrow & M & \textbf{where } x \notin \mathrm{FV}(M) \end{array}$$

The $\alpha$-reduction changes one variable to another for any $\lambda$-expression. Unless stated otherwise, this document treats programs as equal if they are related by only $\alpha$-reduction. The second rule, $\beta$, describes how function application works. And finally, the $\eta$-reduction is the so called *extensionality* property that states that everything can be treated as a function[2]. Though Church presents all three rules, $\beta$ is the rule of computation for which this document is primarily considers.

[2]This does not always hold in my examples, however, since I add constants to the source language as well.

Expressions like 2 and $\lambda x.\, 2$ are considered to be *normal forms*, denoted by the predicate $\mathrm{NF}(M)$, meaning there are no more reductions that can be applied. A term like $\lambda x.\, y\ x$ is considered to be in $\beta$-normal-form since there are no $\beta$-reductions available. For reduction theories, running a program to completion often means that a term is evaluated until it reaches a normal form.

An important aspect of a reduction theory, which separates it from methods that I will examine later, is that reductions can apply to any sub-term of a term by *compatibility*. Therefore, rules can be applied in many different orders to get to the same result. As an example of just how drastic of an effect this has, consider the evaluation of $(\lambda x.\, 2)\ ((\lambda y.\, y\ y)\ (\lambda z.\, z\ z))$. First, let us perform the top-most left-most $\beta$-reduction:

$$(\lambda x.\, 2)\ ((\lambda y.\, y\ y)\ (\lambda z.\, z\ z)) \longrightarrow_\beta 2$$

Another sequence of reduction that I could apply is to reduce the argument of the top most application:

$$\begin{array}{lll} (\lambda x.\, 2)\ ((\lambda y.\, y\ y)\ (\lambda z.\, z\ z)) & \longrightarrow & (\lambda x.\, 2)\ ((\lambda z.\, z\ z)\ (\lambda z.\, z\ z)) \\ & \longrightarrow & 2 \end{array}$$

With this program, in fact, there are an infinite number of reduction sequences wherein each is determined by how many times the argument of the function $\lambda x.\, 2$ is reduced. (This argument term is a common example of an infinitely looping $\Lambda$-term that is called $\Omega$.) Regardless of the order in which reductions are applied, Church's call-by-name calculus will reach the same normal form if one exists; this is a result of the property called the Church-Rosser property [14], or *confluence* when generalized to other rewriting systems.

### 2.2 Call-by-Value

In contrast with Church's presentation the calculus of $\lambda$-conversion, early evaluators for the $\lambda$-calculus, such as the SECD machine [43], chose to fully reduce the argument of a function before performing the substitution. I will explore later, in Section 5, how this simplifies implementations, but for now I will consider its effect on a reduction theory. Always reducing an argument before applying $\beta$ is consistent with Church's calculus, but is more restrictive. Plotkin [69] codified evaluating the argument in the *call-by-value* $\lambda$-calculus. First, a set of *values* is defined as the following:

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

Though at first blush they appear to be similar, values should be distinguished from normal forms. The former determine when a reduction can occur, whereas the latter describes terms that are irreducible. Indeed, the set of values is different since $\lambda x.\, M$ can contain reducible expressions. After the definition of values, $\beta$-reduction is redefined accordingly:

$$(\beta_V) \qquad (\lambda x.\, M)\ V \longrightarrow M[V/x]$$

Unlike the $\beta$-reduction from the call-by-name calculus, this $\beta_V$-reduction does not apply everywhere.

With the restricted $\beta_V$-reduction, the example from the previous subsection would never reach a normal form! Instead, it will attempt to reduce the term $\Omega$, *i.e.* $(\lambda y.\, y\ y)\ (\lambda z.\, z\ z)$, forever because it is the argument of a function. However, this reduction rule does have an advantage over that of the call-by-name $\beta$: it will not duplicate *some* reducible expressions. In practical implementations, this means it can be more efficient. Consider the evaluation of the expression $(\lambda x.\, x + x)\ (1 + 2)$:

$$\begin{array}{lll} (\lambda x.\, x + x)\ (1 + 2) & \longrightarrow_+ & (\lambda x.\, x + x)\ 3 \\ & \longrightarrow_{\beta_V} & 3 + 3 \\ & \longrightarrow_+ & 6 \end{array}$$

In call-by-name, the sub-term $1 + 2$ may be substituted into both occurrences of $x$, thereby adding extra work for the program to reach a normal form. On the other hand, call-by-value guarantees that the most amount of work possible is performed before substitution *at the top-level of the argument.* Work may be duplicated underneath $\lambda$-expressions, however. In the program $(\lambda f.\, f\ (f\ 1))\ (\lambda x.\, 1 + 2 + x)$ the computation $1 + 2$ will be evaluated multiple times even with the call-by-value $\beta$-rule.

The choice of how to evaluate the argument of the function has become known as the language's *evaluation strategy.* Indeed, focusing on variables allows the strategies to be specified with the same $\beta$-reduction but with different definitions of values wherein call-by-name has all terms as values and call-by-value has those just described. Work in the sequent calculus [20, 82], which is a classical variant of the $\lambda$-calculus, often presents these two evaluation strategies in the same formulation as dual to one another.

### 2.3 Call-by-Need

Wadsworth, in his 1971 dissertation [83], notes the inefficiency of call-by-name and proposed a way to avoid the duplication of work caused by substituting unevaluated expressions without switching to call-by-value. His solution treats an expression as a graph wherein variables will be pointers to their binding site. I will describe his approach later in Section 4, but this section is all about calculi. The ability to reason *syntactically* about sharing in non-strict computations came in 1995 when Ariola *et al.* [12] introduced the *call-by-need* evaluation strategy.

Their calculus introduced a new syntactic category for *Answers*:

$$\begin{array}{rrl} V, W \in & \mathit{Value} & ::= \lambda x.\, M \\ A \in & \mathit{Answer} & ::= V \mid (\lambda x.\, A)\ M \end{array}$$

An answer describes the output of a computation which may contain a value or a chain of applied $\lambda$-expressions. First note that any applied $\lambda$-expressions can be thought of as let-expressions, *e.g.* $(\lambda x.\, x)\ (1 + 2)$ is similar to $\texttt{let}\ x = 1 + 2\ \texttt{in}\ x$. Intuitively, an answer can be thought of as a call-by-value value inside of a heap constructed of the applied $\lambda$-expressions.

The reduction rules for the call-by-need $\lambda$-calculus, as presented by the Ariola *et al.* journal paper [11], rely on a notion of *evaluation context.* These specify a *unique* reducible expression within a term. The meta-language search operation $E[M]$ means that "the expression $M$ is in the evaluation context $E$". The shape of an evaluation context $E$ is defined by the following grammar:

$$E \in \mathit{Eval\ Context} ::= \Box \mid E\ N \mid (\lambda x.\, E)\ N \mid (\lambda x.\, E[x])\ E$$

As an example, the term $(\lambda x.\, (\lambda z.\, z)\ 3)\ (1 + 2)$ would be decomposed into the evaluation context $(\lambda x.\, (\lambda z.\, \Box)\ 3)\ (1 + 2)$ and the expression $z$.

There are three reduction rules:

$$\begin{array}{lrcl} (V) & (\lambda x.\, E[x])\ V & \longrightarrow & (\lambda x.\, E[V])\ V \\ (C) & (\lambda x.\, L)\ M\ N & \longrightarrow & (\lambda x.\, L\ N)\ M \\ (A) & (\lambda x.\, L)\ ((\lambda y.\, M)\ N) & \longrightarrow & (\lambda y.\, (\lambda x.\, L)\ M)\ N \end{array}$$

Taking the place of substitution, $V$-reduction[3] states that whenever there is a variable occurrence inside of an evaluation context whose binding is to a value replace the variable with its value. In the heap metaphor, it says to lookup the variable in the heap. $C$-reduction ($C$ for Commute) pushes the binding for $x$ in $L$ into the application $L\ N$. That is, bind $M$ to $x$ in the heap. The final rule, $A$-reduction, re-associates bindings by moving the binding for $y$ outside the binding for $x$. The binding of $y$ to $N$ must be preserved to preserve sharing. With all three of these rules which

[3]The Ariola *et al.* journal paper [11] differs from the conference paper Ariola *et al.* [12] and an other journal paper by Maraist *et al.* [50] in the $V$-reduction. Whereas the former uses evaluation contexts to search for the reducible term, the latter uses any context which is not unique.

move terms under bindings, there is a risk of free variable capture; therefore, they only apply when variable capture will not occur.

Another rule is added by Maraist *et al.* [50] to connect the calculus closer to what is found in practical language implementations at the time.

$$(G) \qquad (\lambda x.\, M)\ N \longrightarrow M \quad \textbf{where } x \notin \mathrm{FV}(M)$$

This rule is similar to garbage collection, hence $(G)$, because it removes unnecessary bindings from an answer.

To demonstrate the calculus, consider an evaluation trace of the same program from the previous sub-section:

$$\begin{array}{rll} (\lambda x.\, x + x)\ (1 + 2) & \longrightarrow_{+} & (\lambda x.\, x + x)\ 3 \\ & \longrightarrow_{V} & (\lambda x.\, 3 + x)\ 3 \\ & \longrightarrow_{V} & (\lambda x.\, 3 + 3)\ 3 \\ & \longrightarrow_{+} & (\lambda x.\, 6)\ 3 \\ & \longrightarrow_{G} & 6 \end{array}$$

Unlike call-by-name, there is no way to duplicate the evaluation of $1 + 2$; similar to call-by-value, this addition must be done before any occurrence $x$ can be replaced with its value. In this trace, garbage collection is done in the last step, but it could have been done before the addition of $3 + 3$ as well.

### 2.4 Substitutions: Meta Language vs. Object Language

The three calculi that I just presented use meta language to perform the task of substituting variables. That is, call-by-name and -value make use of a substitution function for replacing variables with terms, whereas call-by-need makes use of evaluation contexts to search for variables to replace with values. In both cases, it is necessary to be careful—as an implementer or reader—with the naming of variables when replacing terms within other terms. To reduce errors that come from substitutions and to make reduction easier to reason about, two common solutions have evolved for minimizing meta language: De Bruijn notation and explicit substitutions.

*2.4.1 De Bruijn Notation.* As mentioned earlier, the literature often considers $\Lambda$-terms to be quotiented by $\alpha$-equivalence. In proofs and implementations this is not so simple because one expression is $\alpha$-equivalent to a countably infinite set of expressions. To remedy this in practice, the Glasgow Haskell Compiler (GHC), for instance, has a large amount of machinery to handle checking $\alpha$-equivalence of terms. Another remedy for this problem is to remove variables entirely using De Bruijn notation [23] thereby removing the need for the "**where**" provisos in our rules. Variables are replaced with natural numbers $\underline{n}$ that refer to the number of $\lambda$-expressions that must be skipped to find its binding site (these are underlined to distinguish them from the numeric constants found in my example programs).

$$M, N, L \in \mathit{Expression}_{DB} ::= \underline{n} \mid \lambda\, M \mid M\ N$$

For example, the identity function $\lambda x.\, x$ would be represented as $\lambda\, \underline{0}$. Unlike with variables where there are many ways to write this function, *e.g.* $\lambda y.\, y$ and $\lambda z.\, z$, there is only one way to write the identity. In that respect, De Bruijn notation *is* the $\lambda$-calculus quotiented by $\alpha$-equivalence.

The definition of substitution and $\beta$-reduction must be redefined for this new style of writing programs. Substitution is redefined to consider countable sequences of expressions to be substituted.

$$\begin{array}{rcl} \underline{n}[M_0, \dots] & = & M_n \\ (\lambda\, M)[N_0, \dots] & = & \lambda\, M[\underline{0}, N_0, \dots] \\ (M\ N)[L_0, \dots] & = & M[L_0, \dots]\ N[L_0, \dots] \end{array}$$

Lookups, denoted by natural numbers, simply index into the sequence of substitutions. When applying a substitution to a $\lambda$-expression, a new substitution, where the $\underline{0}$ reference is substituted for itself and the rest of the substitution is shifted by one index, is applied to the function body. Applying a substitution to an application applies the substitution to the left- and right-hand-side. When compared to the definition of substitution that was used for specifying call-by-name and -value, De Bruijn eliminates the provisos.

Call-by-value $\beta$-reduction is redefined to build a substitution sequence wherein the formal parameter is at index $\underline{0}$ and every $\underline{i+1}$ index is substituted for $\underline{i}$, *i.e.* shifted down.

$$(\beta_{DB}) \qquad (\lambda\, M)\ V \longrightarrow M[V, \underline{0}, \underline{1}, \underline{2}, \ldots]$$

The reason that every variable is decremented by 1 is because the number of $\lambda$-expressions for variables in $M$ to jump over is also decremented by 1 after an application. As a demonstration of the new substitution and $\beta_{DB}$ rule, consider the following, extra verbose, execution trace of $(\lambda x.\lambda y.\, x)$ 42 9 in De Bruijn notation:

$$\begin{array}{rcl}
(\lambda\, \underline{1})\ 42\ 9 & \longrightarrow_{\beta_{DB}} & (\lambda\, \underline{1})[42, \underline{0}, \underline{1}, \underline{2}, \ldots]\ 9 \\
& = & (\lambda\, \underline{1}[\underline{0}, 42, \underline{0}, \underline{1}, \underline{2}, \ldots])\ 9 \\
& = & (\lambda\, 42)\ 9 \\
& \longrightarrow_{\beta_{DB}} & 42[9, \underline{0}, \underline{1}, \underline{2}, \ldots] \\
& = & 42
\end{array}$$

*2.4.2 Explicit Substitutions.* In language implementations of these calculi, it is impractical to use any of the definitions of substitutions seen above because every function application triggers a substitution and thereby the complete traversal of the function body. Implementations, instead, delay substitutions until the variable is accessed, thereby making function application a single step. Capturing single-step function application as a calculus, Curien [19] introduced the $\lambda\rho$-calculus or the calculus of closures, which Abadi *et al.* [1] later extended to an equational theory in the $\lambda\sigma$-calculus or the calculus of explicit substitutions. I will present the latter because they give a reduction theory. The two papers state plainly that they aim to bridge the gap between language implementations and the $\lambda$-calculi. They do this, like De Bruijn, by minimizing the meta-language further.

The main idea of these calculi is that substitution is no longer a part of the meta-language, instead it is included in the syntax and semantics of the object language. The expression syntax is similar to De Bruijn's extended with an additional syntactic category for substitutions and a new expression applying those substitutions.

$$\begin{array}{rrcl}
L, M, N \in & \textit{Expression} & ::= & \underline{0} \mid \lambda\, M \mid M\ N \mid M\{S\} \\
S, T \in & \textit{Substitution} & ::= & \mathtt{id} \mid \uparrow \mid M \cdot S \mid S \circ T
\end{array}$$

I differentiate the meta-syntactic substitution, seen heretofore, from the syntactic substitution with square- ($M[\ldots]$) and curly- ($M\{\ldots\}$) brackets, respectively. Substitutions are either the identity, a shift, an extension, or a composition. Unlike De Bruijn, the $\lambda\sigma$-calculus only contains one lookup index: $\underline{0}$; all other numbers must be accessed via the shift substitution, *e.g.* $\underline{0}\{\uparrow \circ \uparrow\}$ is similar 2.

There are three sets of reduction rules: one for computation, substitution application, and substitution simplification:

$$
\begin{array}{lrcl}
(\beta) & (\lambda\, M)\ N & \longrightarrow & M\{N \cdot \mathtt{id}\} \\
\\
(\sigma) & \underline{0}\{\mathtt{id}\} & \longrightarrow & \underline{0} \\
& \underline{0}\{M \cdot S\} & \longrightarrow & M \\
& (M\ N)\{S\} & \longrightarrow & M\{S\}\ N\{S\} \\
& (\lambda\, M)\{S\} & \longrightarrow & \lambda\, M\{\underline{0} \cdot (S \circ \uparrow)\} \\
& M\{S\}\{T\} & \longrightarrow & M\{S \circ T\} \\
\\
& \mathtt{id} \circ S & \longrightarrow & S \\
& \uparrow \circ\, \mathtt{id} & \longrightarrow & \uparrow \\
& \uparrow \circ (M \cdot S) & \longrightarrow & S \\
& (M \cdot S) \circ T & \longrightarrow & M\{T\} \cdot (S \circ T) \\
& (S_1 \circ S_2) \circ S_3 & \longrightarrow & S_1 \circ (S_2 \circ S_3)
\end{array}
$$

The $\beta$-reduction is the call-by-name reduction; Abadi *et al.* [1] do not give a call-by-value calculus, but Curien [19] presents both strict and non-strict operational semantics in his paper which I will discuss in the following section. All rules other than $\beta$ are broadly described as $\sigma$-reductions.

De Bruijn notation allowed reasoning about programs without worrying about renaming at the expense of some readability. Calculi with explicit substitutions removes the meta-language operation of substitution entirely and prevents the recursive traversal of function bodies during application at the expense of complex manipulation of explicit substitutions. The example that I gave in the De Bruijn section explodes with explicit substitutions:

$$
\begin{array}{l}
(\lambda\, \lambda\, \underline{0}\{\uparrow\})\ 42\ 9 \\
\longrightarrow_\beta\ (\lambda\, \underline{0}\{\uparrow\})\{42 \cdot \mathtt{id}\}\ 9 \\
\longrightarrow_\sigma\ (\lambda\, \underline{0}\{\uparrow\}\{\underline{0} \cdot ((42 \cdot \mathtt{id}) \circ \uparrow)\})\ 9 \\
\longrightarrow_\beta\ \underline{0}\{\uparrow\}\{\underline{0} \cdot ((42 \cdot \mathtt{id}) \circ \uparrow)\}\{9 \cdot \mathtt{id}\} \\
\longrightarrow_\sigma\ \underline{0}\{\uparrow \circ (\underline{0} \cdot ((42 \cdot \mathtt{id}) \circ \uparrow))\}\{9 \cdot \mathtt{id}\} \\
\longrightarrow_\sigma\ \underline{0}\{(\uparrow \circ (\underline{0} \cdot ((42 \cdot \mathtt{id}) \circ \uparrow))) \circ (9 \cdot \mathtt{id})\} \\
\longrightarrow_\sigma\ \underline{0}\{\uparrow \circ ((\underline{0} \cdot ((42 \cdot \mathtt{id}) \circ \uparrow)) \circ (9 \cdot \mathtt{id}))\} \\
\longrightarrow_\sigma\ \underline{0}\{\uparrow \circ ((\underline{0}\{(9 \cdot \mathtt{id})\} \cdot (((42 \cdot \mathtt{id}) \circ \uparrow) \circ (9 \cdot \mathtt{id}))\} \\
\longrightarrow_\sigma\ \underline{0}\{((42 \cdot \mathtt{id}) \circ \uparrow) \circ (9 \cdot \mathtt{id})\} \\
\longrightarrow_\sigma\ \underline{0}\{(42 \cdot \mathtt{id}) \circ (\uparrow \circ (9 \cdot \mathtt{id}))\} \\
\longrightarrow_\sigma\ \underline{0}\{(42\{\uparrow \circ (9 \cdot \mathtt{id})\} \cdot (\mathtt{id} \circ (\uparrow \circ (9 \cdot \mathtt{id})))\} \\
\longrightarrow_\sigma\ 42\{\uparrow \circ (9 \cdot \mathtt{id})\} \\
\longrightarrow_\sigma\ 42\{\mathtt{id}\} \\
\longrightarrow_\sigma\ 42
\end{array}
$$

## 3 OPERATIONAL SEMANTICS

Reduction theories specify rules for transforming *any* sub-term into a smaller one. Thus, running a program from start to finish, *i.e.* to a normal form, can result in several evaluation paths. For instance, these two reduction sequences are valid in the call-by-name reduction theory:

$$
\begin{array}{l}
(\lambda x.\, x + x)\ (1 + 2) \\
\longrightarrow_{+} (\lambda x.\, x + x)\ 3 \\
\longrightarrow_{\beta} 3 + 3 \\
\longrightarrow_{+} 6
\end{array}
\qquad
\begin{array}{l}
(\lambda x.\, x + x)\ (1 + 2) \\
\longrightarrow_{\beta} (1 + 2) + (1 + 2) \\
\longrightarrow_{+} 3 + (1 + 2) \\
\longrightarrow_{+} 3 + 3 \\
\longrightarrow_{+} 6
\end{array}
$$

Indeed, the left reduction sequence is a valid call-by-value reduction in addition to call-by-name (the reverse is not true). Such ambiguity is especially undesirable in practical programming languages. A program with IO effects like writing output, for instance, may write out characters in an arbitrary order if it is specified as a reduction theory. By selecting one reduction ordering, operational semantics gives a deterministic way to evaluate a term to a normal form. In this section, I present a number of ways to specify an operational semantics: a small-step semantics [70], using evaluation contexts [33], natural semantics [40], and big-step environment semantics [79]. Each of these approaches has a special relationship to how functional languages are specified and implemented. Indeed, the small-step semantics and semantics with evaluation contexts were introduced by elaborating the difference between the SECD-machine (a practical implementation) with the call-by-value $\lambda$-calculus (a reduction theory).

### 3.1 Small-Step Semantics

Plotkin's structural operational semantics [70] describes reduction of a term to a normal form by, as the name suggests, explicitly examining the syntactic structure of the term. If the top-level of a term is not in a position for one of the reduction rules to apply, *e.g.* $((\lambda x.\, x)\ (\lambda z.\, z))\ 42$ since the left-hand-side is an application; then another rule must be provided so that the term can take a step in the right direction. Because of the step-by-step nature of Plotkin's approach, it has been called small-step semantics. To demonstrate, consider the rules for reducing an arithmetic expression to a normal form:

$$\frac{M \longmapsto M'}{M + N \longmapsto M' + N} \qquad \frac{N \longmapsto N'}{m + N \longmapsto m + N'} \qquad \frac{m + n = c}{m + n \longmapsto c}$$

The first two rules are described as structural rules; that is, they perform a search through the structure of the syntax for a unique, reducible sub-expression. An operational semantics is then a single execution path defined by following evaluation function:

$$\text{eval}(M) = M' \quad \textbf{where } M \longmapsto^{*} M' \wedge \text{NF}(M')$$

Note that the definition of normal form is unchanged from the reduction theory in Section 2. Additionally, the arrow ($\longrightarrow$) has been exchanged for ($\longmapsto$). The former will refer to compatible arrows—those that apply anywhere in a term—from the reduction theory, whereas the latter refers to operational arrows for which there is only one.

For evaluating $\Lambda$-terms, a call-by-name operational semantics can be described with just two rules:

$$\frac{M \longmapsto M'}{M\ N \longmapsto M'\ N} \qquad \frac{}{(\lambda x.\, M)\ N \longmapsto M[N/x]}$$

The first rule is a structural rule evaluates the left-hand-side of an applicative expression one step thereby enabling the second rule after repeated application. The rule will continue to apply until the left-hand-side is a function. Thereafter, the second rule for $\beta$-reduction is applied. It is the same as that seen in the reduction theory. For a call-by-value style reduction, there must be another rule for evaluating the right-hand-side of a function application:

$$\frac{M \longmapsto M'}{M\ N \longmapsto M'\ N} \qquad \frac{N \longmapsto N'}{(\lambda x.\, M)\ N \longmapsto (\lambda x.\, M)\ N'} \qquad \frac{}{(\lambda x.\, M)\ V \longmapsto M[V/x]}$$

Notice that the new rule for evaluating the right-hand-side requires the left-hand-side to be the first evaluated to a value. Thus, these rules have made the decision of left-to-right evaluation ordering. Rules could be given for right-to-left (and I shall present some right-to-left implementations later in Section 5), but this is the more common operational semantics.

Re-examining the evaluation of $(\lambda x.\, x + x)\ (1 + 2)$ at the start of the section reveals that the left reduction sequence refers *only* to the call-by-value operational semantics and the right reduction sequence refers *only* to that of call-by-name.

### 3.2 Reduction with Evaluation Contexts

The structural rules above all have the same form: selecting a sub-expression, and when that an expression steps to a new expression, the outer expression takes a step with the updated sub-expression. This can lead to an explosion of rules when new data are added to the language; rules that all essentially do the same thing. Though not the focus of their work, Felleisen and Friedman [33] reveal that using evaluation contexts can reduce the number of structural rules to one, thereby making the specification operational semantics more succinct. Earlier, I presented the call-by-need $\lambda$-calculus reduction theory with evaluation contexts because they were necessary for its specification. Now, I show how evaluation contexts can be used for operational semantics (their original purpose).

After defining a set of evaluation contexts, a semantics with a unique evaluation trace can be built on top of a reduction theory with a single rule:

$$\frac{M \longrightarrow M'}{E[M] \longmapsto E[M']}$$

That is to say, a term which decomposes into the evaluation context $E$ with the sub-term $M$ deterministically steps to the term composed of the evaluation context $E$ with the sub-term $M'$ when $M$ reduces to $M'$.

The following are the set of evaluation contexts for left-to-right evaluation for call-by-name, call-by-value, and call-by-need, respectively:

$$\begin{array}{ll} \mathit{Eval\ Context}_{+} & ::= \Box \mid E + N \mid n + E \\ \mathit{Eval\ Context}_{\mathcal{N}} & ::= \Box \mid E\ N \\ \mathit{Eval\ Context}_{\mathcal{V}} & ::= \Box \mid E\ N \mid (\lambda x.\, M)\ E \\ \mathit{Eval\ Context}_{\mathcal{L}} & ::= \Box \mid E\ N \mid (\lambda x.\, E)\ N \mid (\lambda x.\, E[x])\ E \end{array}$$

The set of contexts differ depending on the requirements for their reduction rules to apply. Herein, the reduction arrow ($\longrightarrow$) is not compatible, *viz.* it only applies at the top level. In call-by-name, there are two evaluation contexts: the empty context for evaluating the current term and the context $E\ N$ which searches for a reducible expression on the left-hand-side of an application. As soon as the left-hand-side of the application is reduced to a value (which will be a $\lambda$-expression), the call-by-name $\beta$-reduction applies. For call-by-value $\beta$-reduction, the argument of a function must be evaluated as well; therefore, there is an evaluation context $(\lambda x.\, M)\ E$ stating that when the left-hand-side of an application is done, then search for a reducible expression in the right-hand-side. As before, this means call-by-value evaluates applications from left-to-right. For call-by-need, there are contexts for evaluating inside of a "heap", *i.e.* an applied $\lambda$-expression; additionally, there is a context for evaluating the bound expression when its variable occurs in an evaluation context in the body of the "heap". The latter can be thought of as forcing an element of the "heap". Call-by-need evaluations proceed in the manner: the application and "inside-of-heap" contexts are used until a variable of the binding occurs in an evaluation context inside of an applied $\lambda$-expression, then the bound term is evaluated (matching the $(\lambda x.\, E[x])\ E$ context), and finally the $V$-reduction applies.

The execution traces at the beginning of the section would not change when using evaluation contexts since they only reduce the number of structural rules needed to specify an operational semantics.

### 3.3 Big-Step Semantics

Another way to specify an operational semantics is with natural semantics as described by Kahn [40] which is also known as big-step semantics. Instead of describing the semantics of our language with a rewriting system—as with both reduction theories and small-step reductions—natural semantics presents the evaluation of a program as a *single* derivation tree. This is in contrast with small-step semantics wherein each reduction step contains a search tree. A big-step semantics must specify, for each syntactic form, a rule for how to get from a program of this form to a fully normalized term. For arithmetic expressions, there are only two rules:

$$\frac{M \Downarrow m \quad N \Downarrow n}{M + N \Downarrow m + n} \qquad \frac{}{n \Downarrow n}$$

In English, the first rule states that $M + N$ evaluates to $m + n$ given proofs that $M$ evaluates to $m$ and $N$ evaluates to $n$.

For call-by-name, the judgement has the form $M \Downarrow_{\mathcal{N}} R$ where $R$ stands for the *results*, *i.e.* the normal forms $x$, $c$, and $\lambda x.\, M$. There are three rules, one for each syntactic form:

$$\frac{}{x \Downarrow_{\mathcal{N}} x} \quad \frac{}{\lambda x.\, M \Downarrow_{\mathcal{N}} \lambda x.\, M} \quad \frac{M \Downarrow_{\mathcal{N}} \lambda x.\, L \quad L[N/x] \Downarrow_{\mathcal{N}} R}{M\ N \Downarrow_{\mathcal{N}} R}$$

The application rule contains all of the information needed to evaluate its left-hand-side to a function *and* perform the application. Therefore, it can be seen as doing more than the $\beta$-reduction from the reduction theory and small-step operational semantics. Moreover, the theory is naturally more efficient. A small step semantics must perform searches for the next reducible expression after every single reduction with an evaluation context decomposition; whereas the big-step semantics, in a sense, keeps track of the sub-term that is being evaluated. As an example of this evaluation mechanism in action, consider again the example from the beginning of the section $(\lambda x.\, x{+}x)\ (1{+}2)$ which yields the following evaluation derivation:

$$\frac{\dfrac{}{\lambda x.\, x + x \Downarrow_{\mathcal{N}} \lambda x.\, x + x} \qquad \dfrac{\dfrac{\dfrac{}{1 \Downarrow_{\mathcal{N}} 1} \quad \dfrac{}{2 \Downarrow_{\mathcal{N}} 2}}{1 + 2 \Downarrow_{\mathcal{N}} 3} \qquad \dfrac{\dfrac{}{1 \Downarrow_{\mathcal{N}} 1} \quad \dfrac{}{2 \Downarrow_{\mathcal{N}} 2}}{1 + 2 \Downarrow_{\mathcal{N}} 3}}{(1 + 2) + (1 + 2) \Downarrow_{\mathcal{N}} 6}}{(\lambda x.\, x + x)\ (1 + 2) \Downarrow_{\mathcal{N}} 6}$$

The call-by-value rules are written $M \Downarrow_{\mathcal{V}} V$ with the same set of values/results (they are the same here) as that of call-by-name. The only difference in the two semantics is the extra evaluation in the call-by-value application case:

$$\frac{}{x \Downarrow_{\mathcal{V}} x} \quad \frac{}{\lambda x.\, M \Downarrow_{\mathcal{V}} \lambda x.\, M} \quad \frac{M \Downarrow_{\mathcal{V}} \lambda x.\, L \quad N \Downarrow_{\mathcal{V}} W \quad L[W/x] \Downarrow_{\mathcal{V}} V}{M\ N \Downarrow_{\mathcal{V}} V}$$

The evaluation of the example program is the following:

$$\frac{\dfrac{}{\lambda x.\, x + x \Downarrow_{\mathcal{V}} \lambda x.\, x + x} \qquad \dfrac{\dfrac{}{1 \Downarrow_{\mathcal{V}} 1} \quad \dfrac{}{2 \Downarrow_{\mathcal{V}} 2}}{1 + 2 \Downarrow_{\mathcal{V}} 3} \qquad \dfrac{\dfrac{}{3 \Downarrow_{\mathcal{V}} 3} \quad \dfrac{}{3 \Downarrow_{\mathcal{V}} 3}}{3 + 3 \Downarrow_{\mathcal{V}} 6}}{(\lambda x.\, x + x)\ (1 + 2) \Downarrow_{\mathcal{V}} 6}$$

As with the small-step semantics, the call-by-value approach avoids the redundant evaluation of $1 + 2$ seen in call-by-name. In specifying an operational semantics for call-by-value with structural

rules, I had to make a choice between left-to-right or right-to-left evaluation of applications. This would be made even more complex when considering data containing more than two reducible sub-terms. However, using the big-step semantics this distinction of ordering is unobservable. To make such a thing important would require the addiction of an effect system.

Call-by-need natural semantics, like its other semantics, stands apart from that of call-by-name and -value. Here, I present the lazy semantics given by Launchbury [44] which, since it shares repeated evaluation of variables, must use a heap. I refer to a pairing of a heap with a term to be evaluated as a *configuration*; thus, the syntax require for evaluation is the following:

$$\begin{array}{lcll} V \in & \mathit{Value} & ::= & c \mid \lambda x.\, M \\ R \in & \mathit{Result} & = & \mathit{Heap} \times \mathit{Value} \\ \Phi \in & \mathit{Heap} & ::= & \varepsilon \mid \Phi, x \mapsto M \\ C \in & \mathit{Configuration} & ::= & \langle \Phi \parallel M \rangle \end{array}$$

Since the heap must be threaded through the sub-derivations of an evaluation, the syntax must also distinguish values from results. Values, herein, are the normalize subsets of terms. Results are returned from evaluation and must pair a normalized term with a heap, *viz.* they are like answers from the call-by-need reduction theory.

The evaluation judgement has the form $\langle \Phi \parallel M \rangle \Downarrow_{\mathcal{L}} (\Phi', V)$ and relates configurations with results. Its rules vary greatly from those of call-by-name and -value:

$$\frac{\langle \Phi_0 \parallel M \rangle \Downarrow_{\mathcal{L}} (\Phi_0', V)}{\langle \Phi_0, x \mapsto M, \Phi_1 \parallel x \rangle \Downarrow_{\mathcal{L}} ((\Phi_0', x \mapsto V, \Phi_1), V)}$$

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

$$\frac{\langle \Phi \parallel M \rangle \Downarrow_{\mathcal{L}} (\Phi', \lambda x.\, L) \quad \langle \Phi', x' \mapsto N \parallel L[x'/x] \rangle \Downarrow_{\mathcal{L}} R}{\langle \Phi \parallel M\ N \rangle \Downarrow_{\mathcal{L}} R}$$

Firstly, if a variable $x$ is not in the heap, then evaluation is stuck unlike the call-by-name and -value big-step semantics. Secondly, the lazy natural semantics does *not* treat variables as normal forms; it must evaluate them and update the heap. Additionally, that rule splits the heap into two parts: one for variables bound before $x$ and one for variables bound after. Intuitively, this follows the call-by-need calculus closely. For example, considering the call-by-need expression $(\lambda x.\, (\lambda y.\, (\lambda z.\, y)\ L)\ N)\ M$. If $y$ is evaluated then it will evaluate $N$; thereby changing the objects referred to by the free variables of $L$, but not $M$ since $y$ is not in scope there. $M$ could be "forced" to a value by the evaluation of $M$, whereas $L$ will only contain pointers to $M$ and $N$ as the variables $x$ and $y$ respectively. Evaluating this example in the lazy natural semantics, the expression $N$, which $y$ is bound to, would be evaluated with the heap $x \mapsto M$. A second difference from the call-by-name and call-by-value semantics is the application case wherein a renaming is performed instead of a normal substitution. Obviously, the expression needs to be added to the heap if its evaluation is to be memoized. Less obviously, the renaming is necessary for avoiding variable capture when a function is applied multiple times. For instance, consider evaluating $(\lambda f.\, f\ (f 1))\ (\lambda x.\, x)$: upon the second application of $f$, $x$ would already be in the heap, and thus the evaluation could not proceed correctly without renaming.

To demonstrate the updating mechanism of the lazy natural semantics, consider the following example from the previous section about call-by-need (which I evaluate in an empty heap):

$$\dfrac{\dfrac{}{\langle \| \ (\lambda x.\, x + x) \rangle \Downarrow_{\mathcal{L}} (, \lambda x.\, x + x)} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\langle \| \ 1 \rangle \Downarrow_{\mathcal{L}} (, 1)} \quad \dfrac{}{\langle \| \ 2 \rangle \Downarrow_{\mathcal{L}} (, 2)}}{\langle \| \ 1 + 2 \rangle \Downarrow_{\mathcal{L}} (, 3)}}{\langle x \mapsto 1 + 2 \ \| \ x \rangle \Downarrow_{\mathcal{L}} (x \mapsto 3, 3)} \quad \dfrac{\dfrac{}{\langle \| \ 3 \rangle \Downarrow_{\mathcal{L}} (, 3)}}{\langle x \mapsto 3 \ \| \ x \rangle \Downarrow_{\mathcal{L}} (x \mapsto 3, 3)}}{\langle x \mapsto 1 + 2 \ \| \ x + x \rangle \Downarrow_{\mathcal{L}} (x \mapsto 3, 6)}}{\langle \| \ (\lambda x.\, x + x) \ (1 + 2) \rangle \Downarrow_{\mathcal{L}} (x \mapsto 3, 6)}$$

### 3.4 Big-Step Environment Semantics

In Launchbury's semantics the heap can be said to *close* the term; that is, all of the free variables of the term are present in the heap. Extending this idea to the other evaluation strategies yields an *environment semantics*. Essentially, this is a style of natural semantics in which configurations of environments and terms, not terms themselves, are evaluated to results. Such semantics can also be found in [56, 79].

An aspect that these semantics have in common—and that they also share with the calculus of explicit substitutions—is that they do not contain eager substitutions, *i.e.* those performed by applying a substitution operation on syntax before continuing. Rather, variables are looked up *lazily* in the environment when they are encountered. This is important when considering practical implementation because substitutions, which environments replace, require code-generation; something not available in our modern computer architectures. Moreover, say that code-generation were possible by having a data representation of the term, substitution would be less efficient since it performs a recursive traversal of the term. Yet another reason for using an environment is presented in Kahn's paper on natural semantics [40]: using environments therein appears to be motivated by its similarity to the type environment found in type-theoretic static semantics (which I will explore later in Section 7).

A non-strict non-memoizing, or call-by-name-like[4], environment semantics depends on the following syntax for configurations and evaluation derivations:

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

Like with the semantics that have been presented thus far, values refer to objects that are associated with variables. However, values herein are objects composed of an unevaluated term together with an environment that it needs for evaluation. As in the substituting big-step semantics, a result is either a constant or a function waiting for an argument; like values which require an environment, a function result requires an environment to give meaning to its free variables. I refer to the value environment-term pairs as *thunk closures* and the result environment-function pairs as *function closures*.

[4]I guard this statement with call-by-name-*like* because I have not proved, nor seen a proof, that shows this semantics to be equivalent to the call-by-name calculus.

The rules for non-strict non-memoizing evaluation reveal how these new closure objects are used:

$$\frac{\Sigma(x) = (\Sigma', M) \quad \langle \Sigma' \parallel M \rangle \Downarrow_{\mathcal{N}} R}{\langle \Sigma \parallel x \rangle \Downarrow_{\mathcal{N}} R} \qquad \frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{N}} (\Sigma, \lambda x.\, M)}$$

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

Whereas nothing happened in the variable case of the call-by-name big-step semantics since variables would be substituted[5], the environment big-step semantics must lookup a closure in the environment and evaluate it. The function evaluation rule must construct a function closure. Finally, the application rule needs to both unpack a function closure and construct a thunk closure. Therein, the body of the function is evaluated with the environment in its closure extended with the thunk closure of its argument.

A strict, or call-by-value-like, environment semantics differs from that of a non-strict non-memoizing semantics in its values and results, *viz.* it keeps the same configuration and environment definitions. Since in call-by-value the argument of a function must be evaluated to a normal form before $\beta$-reduction, the environment semantics must also produce a result before adding it to the environment. It follows that values and results coincide.

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

Despite the strict strategy, the handling of variables is still somewhat lazy. That is, there is still a variable lookup rule, whereas the earlier big-step semantics would have already replaced the variable with some other value. The set of rules is the following:

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

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

When I introduced environment semantics, I described them as a style of semantics wherein configurations are evaluated to results and substitutions are delayed until variables occurrences. Launchbury's semantics already satisfies the former, but not the latter because it contains a renaming substitution in its application rule. As with any other substitution, this is a roadblock for practical implementation because it would require code generation. The problem is solved, like with the non-strict non-memoizing and strict semantics, by using a local environment which contains pointers to heap objects. This approach, seen in Sullivan *et al.* [79], also appears in Sestoft [75]. Therein, the latter performs a similar transformation to his abstract machine semantics to remove renaming.

The syntax for such an evaluation mechanism is much more complicated than those seen earlier. Of course, configurations must now contain heaps, local environments, and terms. Values herein

[5]Such a case would not occur in a closed program.

are only pointers to heap objects[6]. Objects in the heap are either normal forms or thunk closures.

$$\begin{array}{lcll} C \in & \textit{Configuration} & ::= & \langle \Phi \parallel \Sigma \parallel M \rangle \\ \Phi \in & \textit{Heap} & ::= & \varepsilon \mid \Phi, l \mapsto O \\ \Sigma \in & \textit{Environment} & ::= & \textit{Variable} \rightharpoonup \textit{Value} \\ O \in & \textit{Heap Object} & ::= & (\Sigma, M) \mid R \\ V \in & \textit{Value} & ::= & l \\ R \in & \textit{Result} & ::= & c \mid (\Sigma, \lambda x.\, M) \\ A \in & \textit{Answer} & ::= & (\Phi, R) \end{array}$$

As our goal, there is no renaming. However, there is also a change in variable lookup. In the earlier lazy semantics there was only a single rule since unevaluated terms and normal forms were treated in the same way. In the closure semantics, unevaluated terms are thunk closures $(\Sigma, M)$ and normal forms are not. Therefore, the environment semantics requires two evaluation rules for either case.

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

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

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

A property of this semantics—one that would become more obvious if data were added—is that it is order dependent because it must thread the heap through the sub-derivations in the correct order.

This semantics is taken from Sullivan *et al.* [79] and thus it contains a different heap model from that used in Launchbury. Launchbury's heap was split into two parts for each lookup; such a rule is not common in practical implementations of a heap. In this semantics, the heap is treated as a black box with lookup, update, and allocation operations. This is not necessary for an environment semantics, but it is inline with considering the implementation of these different semantics in practice.

## 4 COMBINATORS AND THEIR MACHINES

De Bruijn notation, explicit substitutions, and big-step environment semantics all aim to reduce the problems caused by variables in giving a semantics to Λ-terms. Another approach to handling variables (and removing their importance) comes from combinatory logic [22]. *Combinators* are expressions with no free variables. In combinatory logic, all "user-defined" functions can be replaced by only two primitive combinators known as S and K. Using this as inspiration for programming language implementation yields abstract machines that run combinators. This section explores a few of these combinator-based implementations chronologically including the seminal SK-machine, the Categorical Abstract Machine, and finally the G-machine. The order in which these machines are presented shows a progression in the complexity of the set of combinators accepted by the machines.

[6]The naming scheme does not map exactly to Sullivan *et al.* [79] wherein the notion of results and answers are switched. This is done to better match the terminology of the call-by-need calculus literature.

### 4.1 Fixed Combinator Machines

Turner introduced the idea of combinator machines with his SK-Machine [81]. It is named SK because it *requires* only the S and K combinators in order to run programs. In practice, however, adding more combinators reduces the number of reductions. The definitions of some of these combinators are as follows:

$$\begin{aligned} \mathtt{S}\ f\ g\ x &= f\ x\ (g\ x) \\ \mathtt{K}\ x\ y &= x \\ \mathtt{I}\ x &= x \\ \mathtt{plus}\ x\ y &= x + y \end{aligned}$$

The first combinator, suggestively named S, can be seen as similar to the application rule for substitution. That is, $f$ is applied to $g$; because $f$ and $g$ depend on $x$, the combinator passes $x$ to both. The K and I combinators are more straight forward. The former is the constant function that returns its first argument and the latter is simply the identity function. Finally, I included an addition combinator for computing on constants. The machines input language is composed only of these combinators, constants $n$, and applicative expressions $M\ N$.

Obviously, our syntax of $\Lambda$-terms does not match up with that of the SK-machines input language and must be compiled into it. Unfortunately for my exposition, Turner does not consider the $\lambda$-calculus as its source; rather, it makes use of the SASL programming language which is similar to Haskell without anonymous functions.[7] In SASL, every function gets a name; for instance, the following is an example of a SASL function definition:

$$\mathtt{def}\ \mathit{double}\ x = x + x$$

To turn the SASL definition above into a program consisting of only these combinators, Turner gives the following translation:

$$\begin{aligned} [\![x]\!]\ x &= \mathrm{I} \\ [\![x]\!]\ y &= \mathrm{K}\ y \\ [\![x]\!]\ (M\ N) &= \mathrm{S}\ ([\![x]\!]\ M)\ ([\![x]\!]\ N) \end{aligned}$$

Applying this translation to the function *double* above yields the following:

$$\mathtt{def}\ \mathit{double} = \mathtt{S}\ (\mathtt{S}\ (\mathtt{K}\ \mathtt{plus})\ \mathtt{I})\ \mathtt{I}$$

For function definitions with multiple arguments, the translation is applied to the variables from left to right over the former parameters until they are all eliminated. Turner also presents, as an optimization, an additional translation from this program of combinators into a larger set of combinators.

The essential aspect of Turner's machine, and that of all combinator based systems, is that $\beta$-reduction is not used. Rather, the only rewrites applied are given by the combinators. Turner's machine follows a normal-order reduction (*i.e.* top-most left-most order) on a graph. Using a graph enables sharing thereby replicating a call-by-need operational semantics. For instance, the following is an execution trace of *double* 42 with combinator reductions:

$$\begin{aligned} &(\mathtt{S}\ (\mathtt{S}\ (\mathtt{K}\ \mathtt{plus})\ \mathtt{I})\ \mathtt{I})\ 21 \\ \longrightarrow_{\mathtt{S}}\ &(\mathtt{S}\ (\mathtt{K}\ \mathtt{plus})\ \mathtt{I})\ 21\ (\mathtt{I}\ 21) \\ \longrightarrow_{\mathtt{S}}\ &(\mathtt{K}\ \mathtt{plus})\ 21\ (\mathtt{I}\ 21)\ (\mathtt{I}\ 21) \\ \longrightarrow_{\mathtt{K}}\ &\mathtt{plus}\ (\mathtt{I}\ 21)\ (\mathtt{I}\ 21) \\ \longrightarrow_{\mathtt{I}}^{2}\ &\mathtt{plus}\ 21\ 21 \\ \longrightarrow_{+}\ &42 \end{aligned}$$

[7] Barendregt [14] gives a translation from $\Lambda$ into a combinator calculus, but I am focused on Turner's presentation.

Unfortunately, the machine itself is not presented in the text. From what Turner describes it operates on a representation of the program similar to the trace above.

Expanding on the ideas of Turner, Cousineau *et al.* [18] introduced the Categorical Abstract Machine (CAM). Instead of taking their combinator set from combinatory logic, Cousineau *et al.* elect to use a set of primitive combinators with a relation to $\lambda$-calculus models in category theory, *i.e.* Cartesian closed categories. In contrast to Turner, they provide abstract machines for evaluating strict and non-strict languages.

### 4.2 Super-combinator Machines

With Turner's machine, a graph reduction operated on a program composed of as little as two combinators: S and K. With the CAM, a machine operates on programs composed of a larger set of combinators. Expanding the number of combinators to infinity, super-combinator machines evaluate programs composed of combinators derived directly from the source program. Instead of the one or two argument combinators that have been shown thus far, *super-combinators* are closed, arbitrarily long chains of $\lambda$-expression. Implementations convert source programs into ones composed only of super-combinators and applications through a transformation called *lambda-lifting*. Like Turner's machine, these programs are evaluated through graph reduction. Additionally, the insights of lambda-lifting and source-derived combinator reduction have been adapted for a CAM-style machine in the Categorical Multi-combinator Machine [49].

*4.2.1 Lambda-lifting.* Lambda-lifting arose in the context of lazy language compilers [13, 36, 38, 63]. The transformation works by $\beta$-expanding on the free variables of substitutable expressions—be it a function or an argument to a function—until it is left as a closed $\lambda$-expression that is partially applied. Next, the closed functions are then given a name, removed from the program, and placed in a set of global variables for the G-machine to use.

To demonstrate this, consider the following program:

$$(\lambda x.\,(\lambda f.\,f\ 3 + f\ 4)\ (\lambda y.\,x + y))\ 2$$

There are three functions in this code which lambda-lifting must create super-combinators for and *lift* to the top level. Lambda-lifting will convert this program to the following:

```
super combs
  s0 is λx. s2 (s1 x)
  s1 is λx.λy. x + y
  s2 is λf. f 3 + f 4
main
  s0 2
```

Since the source program only contained one function with a free variable, only one $\beta$-expansion was performed. That is, the function $\lambda y.\,x + y$ depends on the external variable $x$, so it is replaced by the partial application $(\lambda x.\lambda y.\,x + y)\ x$. Later, its super-combinator is given the name $s_0$ lifted, leaving $s_0\ x$ in its place.

As presented by Hughes' dissertation [36], this transformation can be done in two steps. The first being a $\beta$-expansion of all of the closing is specified by the following expansion transformation:

$$\begin{array}{rcl} \mathrm{LL}[\![x]\!] & = & x \\ \mathrm{LL}[\![\lambda x.\,M]\!] & = & (\lambda y_0 \ldots \lambda y_n.\lambda x.\,\mathrm{LL}[\![M]\!])\ y_0 \cdots y_n \quad \textbf{where } y_0 \ldots y_n = \mathrm{FV}(\lambda x.\,M) \\ \mathrm{LL}[\![M\ N]\!] & = & \mathrm{LL}[\![M]\!]\ ((\lambda y_0 \ldots \lambda y_n.\,\mathrm{LL}[\![N]\!])\ y_0 \cdots y_n) \quad \textbf{where } y_0 \ldots y_n = \mathrm{FV}(N) \end{array}$$

Following this expansion, a simple recursive function can traverse the program giving names to the super-combinators and pulling them out.

$$\begin{array}{rcl} \text{Lift}(x) & = & (\varepsilon, x) \\ \text{Lift}(\lambda x_0 \ldots \lambda x_n.\, M) & = & (sup \cup \{\mathsf{f} \mapsto \lambda x_0 \ldots \lambda x_n.\, M'\}, \mathsf{f}) \\ & & \textbf{where } M \neq \lambda y.\, N \textbf{ and } \text{Lift}(M) = (sup, M') \\ \text{Lift}(M\ N) & = & (sup_0 \cup sup_1, M'\ N') \\ & & \textbf{where } \text{Lift}(M) = (sup_0, M') \textbf{ and } \text{Lift}(N) = (sup_1, N') \end{array}$$

A notable difference between my presentation and much of the literature, however, is that they do not consider $\beta$-expansion of function arguments. This is because they work with a source language which has been normalized so that all function arguments are variables that were bound to let-expressions. As extensions to simple lambda-lifting, Johnsson [38] adds mutually recursive blocks to the language which greatly increases the complexity of the transformation and Hughes [36] considers several optimizations. Among Hughes' optimizations are considering the effect of the order in which variables are expanded out and expanding out maximal free expressions instead of only variables themselves. Since the goal of these optimizations is to increase sharing, Hughes appropriately names his transformation: full laziness.

*4.2.2 G-Machines.* A lambda-lifted program serves as the source language for Johnsson's G-machine [37]. As the name implies, the machine makes use of graph representation of programs. The graph is represented as a mutable heap wherein each stored data block refers to a node in the graph. The machine syntax, compilation rules, and transitions are given in Figure 1. A machine state is composed of a code pointer, a graph node stack, a value stack, a heap containing the graph information, and a dump for returning from functions. Unlike the full machine given by Johnsson has 36 transition rules, I have greatly simplified the machine since I am primarily interested in functions. Specifically, I removed the output register, the global environment of super-combinators since it does not change throughout execution, and instructions to handle algebraic data types. I keep numbers and addition because they are particularly interesting in call-by-need evaluation needing to be evaluated strictly as unboxed values.

The compilation of a lambda-lifted source program is made up of four sub-translations. The first $\mathrm{F}[\![-]\!]$ translates super-combinators into G-code by translating the body of the combinator then adding update and return instructions. The latter three translations are all meant to translate expressions and are indexed by a mapping from variables to stack locations and a stack depth. $\mathrm{E}[\![-]\!]$ is meant to generate code for evaluating the expression on the top of the stack. $\mathrm{B}[\![-]\!]$ translates arithmetic expressions. And finally, $\mathrm{C}[\![-]\!]$ translates graph constructing expressions. For demonstration, compiling the lambda-lifted program from the previous subsection yields the following G-machine code wherein I use $\underline{n}$ to differentiate stack accessing natural numbers from numeric constants as in De Bruijn:

```
super combs
  s₀ is PushFun s₂; PushFun s₁; Push 2; MkApp; MkApp; Update 2; Ret 1
  s₁ is Push 0; Eval; Unbox; Push 2; Eval; Unbox; Add; MkNum; Update 3; Ret 2
  s₂ is
    Push 0; PushNum 3; MkApp; Eval; Unbox;
    Push 1; PushNum 4; MkApp; Eval; Unbox;
    Add; MkNum; Update 2; Ret 1
main
  PushFun s₀; PushNum 2; MkApp; Eval
```

**Syntax**

$$
\begin{array}{rlll}
C \in & \textit{Code} & ::= & I \mid I; C \\
I \in & \textit{Instruction} & ::= & \texttt{Eval} \mid \texttt{Unwind} \mid \texttt{Update}\ \underline{n} \mid \texttt{Ret}\ \underline{n} \\
 & & & \mid \texttt{PushFun s} \mid \texttt{PushNum}\ n \mid \texttt{PushVar}\ \underline{n} \\
 & & & \mid \texttt{MkNum} \mid \texttt{MkApp} \mid \texttt{Unbox} \mid \texttt{PushUnboxNum}\ n \mid \texttt{Add} \\
S \in & \textit{Stack} & ::= & \varepsilon \mid l \cdot S \\
V \in & \textit{Primitive Stack} & ::= & \varepsilon \mid c \cdot S \\
G \in & \textit{Graph} & ::= & \textit{Location} \rightharpoonup \textit{Node} \\
N \in & \textit{Node} & ::= & \texttt{Num}\ c \mid \texttt{Fun}\ f \mid \texttt{Ap}\ l\ l \\
D \in & \textit{Dump} & ::= & \varepsilon \mid (C, S) \cdot D \\
 & \textit{Machine State} & ::= & \langle C \parallel S \parallel V \parallel G \parallel D \rangle
\end{array}
$$

**Compilation**

$$
\begin{array}{rcl}
\mathrm{F}[\![\lambda x_0, \ldots, x_n . M]\!] & = & \mathrm{E}[\![M]\!]\ r\ (n+2); \texttt{Update}\ \underline{n+2}; \texttt{Ret}\ \underline{n+1} \\
 & & \textbf{where } r = x_0 \mapsto n+2, \ldots, x_n \mapsto 2 \\
 & & \\
\mathrm{E}[\![c]\!]\ r\ n & = & \texttt{PushNum}\ c \\
\mathrm{E}[\![M+N]\!]\ r\ n & = & \mathrm{B}[\![M+N]\!]\ r\ n; \texttt{MkNum} \\
\mathrm{E}[\![x]\!]\ r\ n & = & \texttt{PushVar}\ \underline{n - r(x)}; \texttt{Eval} \\
\mathrm{E}[\![M]\!]\ r\ n & = & \mathrm{C}[\![M]\!]\ r\ n; \texttt{Eval} \\
 & & \textbf{otherwise} \\
 & & \\
\mathrm{B}[\![c]\!]\ r\ n & = & \texttt{PushUnboxNum}\ c \\
\mathrm{B}[\![M+N]\!]\ r\ n & = & \mathrm{B}[\![M]\!]\ r\ n; \mathrm{B}[\![N]\!]\ r\ (n+1); \texttt{Add} \\
\mathrm{B}[\![M]\!]\ r\ n & = & \mathrm{E}[\![M]\!]\ r\ n; \texttt{Unbox} \\
 & & \textbf{otherwise} \\
 & & \\
\mathrm{C}[\![c]\!]\ r\ n & = & \texttt{PushNum}\ c \\
\mathrm{C}[\![x]\!]\ r\ n & = & \texttt{PushVar}\ \underline{n - r(x)} \\
\mathrm{C}[\![\mathrm{f}]\!]\ r\ n & = & \texttt{PushFun f} \\
\mathrm{C}[\![M\ N]\!]\ r\ n & = & \mathrm{C}[\![M]\!]\ r\ n; \mathrm{C}[\![N]\!]\ r\ (n+1); \texttt{MkApp}
\end{array}
$$

**Transitions**

$$
\begin{array}{rcl}
\langle \texttt{Eval}; C \parallel l \cdot S \parallel V \parallel G[l \mapsto \texttt{Ap}\ l_x\ l_y] \parallel D \rangle & \longmapsto & \langle \texttt{Unwind} \parallel l \cdot \varepsilon \parallel V \parallel G[l \mapsto \texttt{Ap}\ l_x\ l_y] \parallel (C, S) \cdot D \rangle \\
\langle \texttt{Eval}; C \parallel l \cdot S \parallel V \parallel G \parallel D \rangle & \longmapsto & \langle C \parallel S \parallel V \parallel G \parallel D \rangle \\
 & & \textbf{where } G(l) = \texttt{Num}\ c \textbf{ or } G(l) = \texttt{Fun s} \\
\langle \texttt{Unwind} \parallel l \cdot S \parallel V \parallel G[l \mapsto \texttt{Ap}\ l_x\ l_y] \parallel D \rangle & \longmapsto & \langle \texttt{Unwind} \parallel l_x \cdot l \cdot S \parallel V \parallel G[l \mapsto \texttt{Ap}\ l_x\ l_y] \parallel D \rangle \\
\langle \texttt{Unwind} \parallel l \cdot l_1 \cdots l_k \cdot S \parallel V \parallel G[l \mapsto \texttt{Fun}\ f] \parallel D \rangle & \longmapsto & \langle C_\mathrm{s} \parallel l'_1 \cdots l'_k \cdot S \parallel V \parallel G[l \mapsto \texttt{Fun s}] \parallel D \rangle \\
 & & \textbf{where } \mathrm{s} = (k, C_\mathrm{s}) \textbf{ and } \forall i.\, G(l_i) = \texttt{Ap}\ l'_i\ l''_i \\
\langle \texttt{Unwind} \parallel l \cdot l_1 \cdots l_k \cdot \varepsilon \parallel V \parallel G[l \mapsto \texttt{Fun s}] \parallel (C', S') \cdot D \rangle & \longmapsto & \langle C' \parallel l_k \cdot S' \parallel V \parallel G \parallel D \rangle \\
 & & \textbf{where } \mathrm{s} = (j, C_\mathrm{s}) \text{ and } k < j \\
\langle \texttt{Update}\ \underline{n}; C \parallel l_0 \cdot l_1 \cdots l_n \cdot S \parallel V \parallel G[l_0 \mapsto N, l_n \mapsto \_] \parallel D \rangle & \longmapsto & \langle C \parallel l_1 \cdots l_n \cdot S \parallel V \parallel G[l_0 \mapsto N, l_n \mapsto N] \parallel D \rangle \\
\langle \texttt{Ret}\ \underline{n} \parallel l_0 \cdots l_n \cdot l \cdot \varepsilon \parallel V \parallel G \parallel (C, S) \cdot D \rangle & \longmapsto & \langle C' \parallel l \cdot S \parallel V \parallel G \parallel D \rangle \\
 & & \textbf{where } G(l) = \texttt{Num}\ c \\
\langle \texttt{Ret}\ \underline{n} \parallel l_0 \cdots l_n \cdot l \cdot S \parallel V \parallel G \parallel D \rangle & \longmapsto & \langle \texttt{Unwind} \parallel l \cdot S \parallel V \parallel G \parallel D \rangle \\
 & & \textbf{where } G(l) = \texttt{Ap}\ l_x\ l_y \textbf{ or } G(l) = \texttt{Fun s} \\
\langle \texttt{PushFun s}; C \parallel S \parallel V \parallel G \parallel D \rangle & \longmapsto & \langle C \parallel l \cdot S \parallel V \parallel G[l \mapsto \texttt{Fun s}] \parallel D \rangle \\
\langle \texttt{PushNum}\ n; C \parallel S \parallel V \parallel G \parallel D \rangle & \longmapsto & \langle C \parallel l \cdot S \parallel V \parallel G[l \mapsto \texttt{Num}\ n] \parallel D \rangle \\
\langle \texttt{PushVar}\ \underline{n}; C \parallel l_0 \cdots l_n \cdot S \parallel V \parallel G \parallel D \rangle & \longmapsto & \langle C \parallel l_n \cdot l_0 \cdots l_n \cdot S \parallel V \parallel G \parallel D \rangle \\
\langle \texttt{MkNum}; C \parallel S \parallel n \cdot V \parallel G \parallel D \rangle & \longmapsto & \langle C \parallel l \cdot S \parallel V \parallel G[l \mapsto \texttt{Num}\ n] \parallel D \rangle \\
\langle \texttt{MkApp}; C \parallel l_y \cdot l_x \cdot S \parallel V \parallel G \parallel D \rangle & \longmapsto & \langle C \parallel l \cdot S \parallel V \parallel G[l \mapsto \texttt{App}\ l_x\ l_y] \parallel D \rangle \\
\langle \texttt{Unbox}; C \parallel l \cdot S \parallel V \parallel G[l \mapsto \texttt{Num}\ n] \parallel D \rangle & \longmapsto & \langle C \parallel S \parallel n \cdot V \parallel G[l \mapsto \texttt{Num}\ n] \parallel D \rangle \\
\langle \texttt{PushUnboxNum}\ n; C \parallel S \parallel V \parallel G \parallel D \rangle & \longmapsto & \langle C \parallel S \parallel n \cdot V \parallel G \parallel D \rangle \\
\langle \texttt{Add}; C \parallel S \parallel n \cdot m \cdot V \parallel G \parallel D \rangle & \longmapsto & \langle C \parallel S \parallel (n+m) \cdot V \parallel G \parallel D \rangle
\end{array}
$$

Fig. 1. G-machine

After compilation to G-machine code, the evaluation process begins with the `Eval` instruction. If the current node on the top of the stack is an application, then the machine must start to *unwind* the application, *i.e.* to search for the next reducible expression down the left-hand-side of the application. When unwinding transitions finally land on a label which maps to function data, one of two transitions can happen depending on whether there are enough arguments on the stack; note that arguments are actually labels the point to application nodes containing their arguments. If there are enough arguments, then the program replaces the application nodes on the stack with their argument part and enters the code in the body of the function. Otherwise, there are not enough arguments on the stack, and therefore, the function returns prematurely with only the top application on the stack; note that this means the under-applied application will need to be unwinded again when there are enough arguments.

As seen in the G-machine code generation, exiting a super-combinator will perform an update and return. The former selects a specific node in the stack and updates it with the node at the top of the stack. The latter has two cases. Returning a number simply places the last element of the stack on the stack in the dump and start executing the dump code. Otherwise, the node being returned is a function or application; thus, an unwind state is entered and the dump is not restored yet.

The instruction for evaluating numbers—*i.e.* `MkNum`, `PushNum`, `PushUnboxNum`, `Unbox`, and `Add`—are interesting in that they must destruct the graph and make use of the value stack. In the graph machine, unboxed or primitive numbers cannot be passed through variables instead they are hidden inside of boxes. This can lead to inefficient unboxing and reboxing for chains of arithmetic expressions [65].

Following the development of the machine, there have been a number of incremental improvements on the it. Since it is inefficient in that it spends several transitions constructing and updating the graph representation of program, Burn's Spinless G-machine [15] made improvements by reducing the updates to the graph when unwinding an expression. The G-machine above was specifically inefficient in rebuilding the spine, *i.e.* unwinding applications, for under-applied function. Another lazy machine built to execute with source-derived combinators is Fairbairn's [32] which reduced the number of instructions needed to execute lambda-lifted programs to three in the Three Instruction Machine (TIM). Drawing a great deal of inspiration from TIM, the Glasgow Haskell Compiler today uses the Spineless Tagless G-machine, which discards using combinators as an approach to compilation [66].

## 5 ABSTRACT MACHINES

Whereas combinator based evaluation strategies where developed to implement programs in a calculus of combinators, the abstract machines I present in this section are designed to run Λ-terms themselves (or simple instructions from trivial code generation). However, there are some striking similarities with those machines presented before because combinator and abstract machines share the fact that they store a notion of evaluation context as state. This makes them more efficient in practice than a small-step semantics which must search again for the next redex after each step. I will discuss four machines: the SECD, Krivine, ZINC, and STG machines. The first two are early examples of abstract machine executing a strict and non-strict evaluation strategy, respectively. The last two machines are more recent versions that are optimized for fast curried function calls, a common occurrence in function programs.

Although I have heretofore focused on classifying semantics according to their evaluation strategy, the level of abstract machines presents another kind of classification to be aware of: the difference between *push/enter* and *eval/apply* style applications. The difference is focused on how application is performed. Considering the former, for which the Krivine machine is a part, arguments are pushed on the stack and the left-hand-side of the application is entered. In

**Syntax**

$$
\begin{array}{rcll}
S \in & \textit{Stack} & ::= \varepsilon \mid V \cdot S \\
E \in & \textit{Environment} & ::= \varepsilon \mid E, x \mapsto V \\
C \in & \textit{Control} & ::= \varepsilon \mid M \cdot C \mid \mathsf{ap} \cdot C \\
D \in & \textit{Dump} & ::= \varepsilon \mid (S, E, C, D) \\
V \in & \textit{Value} & ::= c \mid (E, \lambda x.\, M) \\
 & \textit{Machine State} & ::= \langle S \parallel E \parallel C \parallel D \rangle
\end{array}
$$

**Transitions**

$$
\begin{array}{rcl}
\langle V \cdot \varepsilon \parallel E \parallel \varepsilon \parallel (S', E', C', D') \rangle & \longmapsto & \langle V \cdot S' \parallel E' \parallel C' \parallel D' \rangle \\
\langle S \parallel E \parallel x \cdot C \parallel D \rangle & \longmapsto & \langle E(x) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle S \parallel E \parallel \lambda x.\, M \cdot C \parallel D \rangle & \longmapsto & \langle (E, \lambda x.\, M) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle (E', \lambda x.\, M) \cdot V \cdot S \parallel E \parallel \mathsf{ap} \cdot C \parallel D \rangle & \longmapsto & \langle \varepsilon \parallel E', x \mapsto V \parallel M \cdot \varepsilon \parallel (S, E, C, D) \rangle \\
\langle S \parallel E \parallel M\, N \cdot C \parallel D \rangle & \longmapsto & \langle S \parallel E \parallel N \cdot M \cdot \mathsf{ap} \cdot C \parallel D \rangle
\end{array}
$$

Fig. 2. SECD machine

contrast, eval/apply machines like SECD evaluate both sides of an application first; afterwards, the application site knows how to pass arguments and enter the function body. At first glance, push/enter seems to align with a call-by-name evaluation strategy because it is easy to push an argument on the stack and continue with the function. On the other hand, eval/apply seems suited for call-by-value evaluation since the argument must be evaluated before entering the function. However, this turns out not to be the case as seen by Marlow and Peyton Jones [51] wherein both of these approaches are shown to work in call-by-need (and eval/apply is faster). And the ZINC machine is indeed a push/enter call-by-value machine.

### 5.1 The SECD Machine

The earliest cited $\lambda$-calculus abstract machine was Landin's SECD machine [43]. The call-by-value machine is named for the four parts of its state: $S$, an intermediate result stack; $E$, an environment containing mappings from variables to values; $C$, a control stack; and $D$, a dump of a machine state.

Figure 2 gives the syntax and transition rules for the machine. The result stack holds only values and the environment maps to values. The notion of value for the SECD machine, and many other abstract machines (at least all of those that I will present), is different from the notion used for the reduction theory and structural operational semantics wherein values are a subset of $\Lambda$-terms. Instead, values refer to objects that can be mapped to from the environment as seen in the big-step environment semantics. For an object that behaves like an integer, constants like 4 are values whereas arithmetic expressions like 3 + 1 must be evaluated before they can be placed on the result stack or in the environment. In the case of functions, the SECD machine constructs function closures, *e.g.* $\lambda x.\, y$ must be evaluated to $(E, \lambda x.\, y)$ for some environment $E$.

The control stack is so named because the next state transition always depends on the value at the top of this stack. When the top of the control stack is an application expression, the application is deconstructed and an application marker, the function, and the argument are placed on the control stack, *in that order*. This deconstruction does the same work as evaluation contexts from the operational semantics: searching for the next expression to evaluate. When both the argument and the function are evaluated to a value, an application marker will be at the top of the control stack which triggers the application.

**Syntax**

$$
\begin{array}{rcll}
\kappa \in & \textit{Continuation} & ::= & \varepsilon \mid V \cdot \kappa \\
\Sigma \in & \textit{Environment} & ::= & \varepsilon \mid \Sigma, x \mapsto V \\
V, W \in & \textit{Value} & ::= & (\Sigma, M) \\
& \textit{Machine State} & ::= & \langle M \parallel \Sigma \parallel \kappa \rangle
\end{array}
$$

**Transitions**

$$
\begin{array}{rcl}
\langle M\ N \parallel \Sigma \parallel \kappa \rangle & \longmapsto & \langle M \parallel \Sigma \parallel (\Sigma, N) \cdot \kappa \rangle \\
\langle \lambda x.\, M \parallel \Sigma \parallel (\Sigma', N) \cdot \kappa \rangle & \longmapsto & \langle M \parallel \Sigma, x \mapsto (\Sigma', N) \parallel \kappa \rangle \\
\langle x \parallel \Sigma, x \mapsto (\Sigma', M) \parallel \kappa \rangle & \longmapsto & \langle M \parallel \Sigma' \parallel \kappa \rangle
\end{array}
$$

Fig. 3. Krivine machine

The dump of the SECD machine is used to return from function calls. When a function is applied, the state of the machine is saved. The state is re-instantiated when the function returns and the value computed by the function call is added to the result stack.

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

$$
\begin{array}{l}
\langle S \parallel E \parallel (\lambda x.\, \lambda y.\, x)\ 4\ 2 \cdot C \parallel D \rangle \\
\longmapsto \langle S \parallel E \parallel 2 \cdot (\lambda x.\, \lambda y.\, x)\ 4 \cdot \mathsf{ap} \cdot C \parallel D \rangle \\
\longmapsto \langle 2 \cdot S \parallel E \parallel (\lambda x.\, \lambda y.\, x)\ 4 \cdot \mathsf{ap} \cdot C \parallel D \rangle \\
\longmapsto \langle 2 \cdot S \parallel E \parallel 4 \cdot \lambda x.\, \lambda y.\, x \cdot \mathsf{ap} \cdot \mathsf{ap} \cdot C \parallel D \rangle \\
\longmapsto \langle 4 \cdot 2 \cdot S \parallel E \parallel \lambda x.\, \lambda y.\, x \cdot \mathsf{ap} \cdot \mathsf{ap} \cdot C \parallel D \rangle \\
\longmapsto \langle (E, \lambda x.\, \lambda y.\, x) \cdot 4 \cdot 2 \cdot S \parallel E \parallel \mathsf{ap} \cdot \mathsf{ap} \cdot C \parallel D \rangle \\
\longmapsto \langle \varepsilon \parallel E, x \mapsto 4 \parallel \lambda y.\, x \cdot \varepsilon \parallel (2 \cdot S, E, \mathsf{ap} \cdot C, D) \rangle \\
\longmapsto \langle (E, x \mapsto 4, \lambda y.\, x) \cdot \varepsilon \parallel E, x \mapsto 4 \parallel \varepsilon \parallel (2 \cdot S, E, \mathsf{ap} \cdot C, D) \rangle \\
\longmapsto \langle (E, x \mapsto 4, \lambda y.\, x) \cdot 2 \cdot S \parallel E \parallel \mathsf{ap} \cdot C \parallel D \rangle \\
\longmapsto \langle \varepsilon \parallel E, x \mapsto 4, y \mapsto 2 \parallel x \cdot \varepsilon \parallel (S, E, C, D) \rangle \\
\longmapsto \langle 4 \cdot \varepsilon \parallel E, x \mapsto 4, y \mapsto 2 \parallel \varepsilon \parallel (S, E, C, D) \rangle \\
\longmapsto \langle 4 \cdot S \parallel E \parallel C \parallel D \rangle
\end{array}
$$

An important detail to note about evaluation is that it is right-to-left, *i.e.* it evaluates the argument of a function to a value before the function itself. This is not equivalent to call-by-value operational semantics when considering some computational effects. For non-termination, it does not matter whether the argument $M$ or $N$ loops forever in the program $L\ M\ N$ since the observable effect would be that the program loops forever. However, if the program was `callcc` $(\lambda k.\, L\ (k\ 1)\ (k\ 2))$, then the program would not produce the same output as a call-by-value execution.

### 5.2 The Krivine Machine

For call-by-name languages, the Krivine machine is the earliest abstract machine despite remaining unpublished folklore until 2007 [21, 28, 42]. As published, the Krivine machine operates on a special version of DeBruijn indices, but I present, instead, a machine that operates on variables because to ease readability. The machine's syntax and transitions are shown in Figure 3. Like the SECD machine, the Krivine machine has a notion of call stack which contains only arguments to functions. Whereas SECD has a stack for values and a dump for returning from function calls, the Krivine machine encodes all of this information with its single stack.

Since the non-strict machine does not evaluate the arguments to functions and substituted unevaluated expressions may contain free variables, the machine creates closures for everything that is added to the environment. These closures are constructed *eagerly* when they are pushed

on the stack. Unlike the SECD machine which did not properly implement call-by-value, the Krivine machine indeed implements call-by-name by avoiding the evaluation of its argument and proceeding directly to evaluating the left-hand-side of the application.

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

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

### 5.3 Fast Curried Function Calls

Noting that the most important operation in functional languages is function application, Cardelli [16] strove to improve on these machines with the Functional Abstract Machine (FAM). The SECD and Krivine machines shown thus far only contained single function application; a fact which can be improved upon. In particular, it is important to fuse multiple applications of curried function calls. To demonstrate the importance, the program $(\lambda x.\, \lambda y.\, M)\ 1\ 2$ could perform the top-left application, return $(\lambda y.M[1/x])\ 2$, and then perform the second application. In the big-step environment semantics and the SECD machine, such an execution would generate two closures; one for each time a function is returned. Fusing these calls would prevent the creation of these extra data structures. I will show how this problem is solved in a call-by-value and call-by-need language with the ZINC and STG machines.

*5.3.1 ZINC Abstract Machine.* Leroy's ZINC abstract machine (ZAM) [45] was designed to be a strict abstract machine optimized for multi-arity function calls. The machine can be seen as a modification of the Krivine machine to support strict evaluation and multi-arity function calls. As presented above, the Krivine machine cannot support these two features by simply evaluating function arguments. Since the argument may be evaluated to a $\lambda x.\, M$, the machine needs to know to stop evaluating and return a function closure instead of entering this function; this needs to be done in the presence of other elements on the stack, and therefore the solution to wait till the stack is empty, as Krivine does, does not work. The ZAM solves this problem by pushing marks on the stack at the beginning of a multiple-application; if a function occurs and the mark is not present, then the machine knows it is evaluating an argument and returns a function closure. The ZAM's combination of being strict and being push/enter means that no intermediate closures are constructed during curried function calls, unless the value of an argument is a function.

The syntax, compilation, and transition rules for the ZAM are given in Figure 4. Unlike the machines seen so far, the machine relies on a simple instruction language which the De Bruijn $\lambda$-calculus source language is compiled into. A machine state is a 5-tuple containing a list of instructions, an accumulator register, an environment, and an argument and return stack. The argument stack is where function arguments are placed in preparation for entering a function, obviously. It holds not only values but also marks, which indicate the end of the argument list. The return stack is used for reinstating the environment after exiting a function call—like the SECD's dump register—and thus, it requires code and an environment to return to.

Compilation of a De Bruijn expression contains two sub-translations: Comp$[\![-]\!]$ is the top-level translation and Ret$[\![-]\!]$ is a special translation to be used for the bodies of functions. If the program has nested $\lambda$-expressions like $\lambda\lambda\ \underline{1}$, then Comp$[\![-]\!]$ would generate `Cur(Grab;Access(`$\underline{1}$`);Return)` which would involve shuffling a closure to the accumulator register before immediately entering it. The Ret$[\![-]\!]$ will avoid this shuffle and just grab an argument from the stack.

**Syntax**

$$
\begin{array}{rcll}
C \in & \textit{Code} & ::= & I \mid I; C \\
I \in & \textit{Instruction} & ::= & \mathtt{Access}(\underline{n}) \mid \mathtt{Pushmark} \mid \mathtt{Push} \mid \mathtt{Apply} \mid \mathtt{AppTerm} \\
 & & & \quad \mid \mathtt{Grab} \mid \mathtt{Cur}(C) \mid \mathtt{Return} \\
V \in & \textit{Value} & ::= & (C, E) \mid c \\
A \in & \textit{Accumulator} & ::= & \varepsilon \mid V \\
E \in & \textit{Environment} & ::= & \varepsilon \mid V \cdot E \\
S \in & \textit{Arg Stack} & ::= & \varepsilon \mid V \cdot S \mid \bullet \cdot S \\
R \in & \textit{Return Stack} & ::= & \varepsilon \mid (C, E) \cdot S \\
 & \textit{Machine State} & ::= & \langle C \parallel A \parallel E \parallel S \parallel R \rangle
\end{array}
$$

**Compilation**

$$
\begin{array}{rcl}
\mathrm{Comp}[\![n]\!] & = & \mathtt{Access}(n) \\
\mathrm{Comp}[\![M\ N_0 \cdots N_n]\!] & = & \mathtt{Pushmark}; \mathrm{Comp}[\![N_n]\!]; \mathtt{Push}; \ldots; \mathrm{Comp}[\![N_0]\!]; \mathtt{Push}; \mathrm{Comp}[\![M]\!]; \mathtt{Apply} \\
\mathrm{Comp}[\![\lambda\, M]\!] & = & \mathtt{Cur}(\mathrm{Ret}[\![M]\!]; \mathtt{Return}) \\
 & & \\
\mathrm{Ret}[\![n]\!] & = & \mathtt{Access}(n) \\
\mathrm{Ret}[\![M\ N_0 \cdots N_n]\!] & = & \mathrm{Comp}[\![N_n]\!]; \mathtt{Push}; \ldots; \mathrm{Comp}[\![N_0]\!]; \mathtt{Push}; \mathrm{Comp}[\![M]\!]; \mathtt{AppTerm} \\
\mathrm{Ret}[\![\lambda\, M]\!] & = & \mathtt{Grab}; \mathrm{Ret}[\![M]\!]
\end{array}
$$

**Transitions**

$$
\begin{array}{rcl}
\langle \mathtt{Access}(n); C \parallel A \parallel \ldots V_n \ldots \parallel S \parallel R \rangle & \longmapsto & \langle C \parallel V_n \parallel \ldots V_n \ldots \parallel S \parallel R \rangle \\
\langle \mathtt{Appterm}; C \parallel (C', E') \parallel E \parallel V \cdot S \parallel R \rangle & \longmapsto & \langle C' \parallel (C', E') \parallel V \cdot E' \parallel S \parallel R \rangle \\
\langle \mathtt{Apply}; C \parallel (C', E') \parallel E \parallel V \cdot S \parallel R \rangle & \longmapsto & \langle C' \parallel (C', E') \parallel V \cdot E' \parallel S \parallel (C, E) \cdot R \rangle \\
\langle \mathtt{Push}; C \parallel A \parallel E \parallel S \parallel R \rangle & \longmapsto & \langle C \parallel A \parallel E \parallel A \cdot S \parallel R \rangle \\
\langle \mathtt{Pushmark}; C \parallel A \parallel E \parallel S \parallel R \rangle & \longmapsto & \langle C \parallel A \parallel E \parallel \bullet \cdot S \parallel R \rangle \\
\langle \mathtt{Cur}(C); C' \parallel A \parallel E \parallel S \parallel R \rangle & \longmapsto & \langle C' \parallel (C, E) \parallel E \parallel S \parallel R \rangle \\
\langle \mathtt{Grab}; C \parallel A \parallel E \parallel \bullet \cdot S \parallel (C', E') \cdot R \rangle & \longmapsto & \langle C' \parallel (C, E) \parallel E' \parallel S \parallel R \rangle \\
\langle \mathtt{Grab}; C \parallel A \parallel E \parallel V \cdot S \parallel R \rangle & \longmapsto & \langle C \parallel A \parallel V \cdot E \parallel S \parallel R \rangle \\
\langle \mathtt{Return}; C \parallel A \parallel E \parallel \bullet \cdot S \parallel (C', E') \cdot R \rangle & \longmapsto & \langle C' \parallel A \parallel V \cdot E' \parallel S \parallel R \rangle \\
\langle \mathtt{Return}; C \parallel (C', E') \parallel E \parallel V \cdot S \parallel R \rangle & \longmapsto & \langle C' \parallel A \parallel E' \parallel S \parallel R \rangle
\end{array}
$$

Fig. 4. ZINC abstract machine

As an example, the program $(\lambda\, \lambda\, \underline{1})$ 42 9 compiles to the ZAM code:

```
Pushmark;
  Num(9); Push; Num(42); Push;
  Cur(Grab; Access(1); Return);
Apply; Halt
```

I have added two instructions for the example: $\texttt{Num}(n)$ returns a constant to the accumulator register and `Halt` ends the program with the accumulator register as the result.

$$
\begin{aligned}
&\langle \texttt{Pushmark}; \cdots \parallel \varepsilon \parallel \varepsilon \parallel \varepsilon \rangle \\
&\longmapsto \langle \texttt{Num}(9); \cdots \parallel \varepsilon \parallel \varepsilon \parallel \bullet \cdot \varepsilon \parallel \varepsilon \rangle \\
&\longmapsto \langle \texttt{Push}; \cdots \parallel 9 \parallel \varepsilon \parallel \bullet \cdot \varepsilon \parallel \varepsilon \rangle \\
&\longmapsto \langle \texttt{Num}(42); \cdots \parallel 9 \parallel \varepsilon \parallel 9 \cdot \bullet \cdot \varepsilon \parallel \varepsilon \rangle \\
&\longmapsto \langle \texttt{Push}; \cdots \parallel 42 \parallel \varepsilon \parallel 9 \cdot \bullet \cdot \varepsilon \parallel \varepsilon \rangle \\
&\longmapsto \langle \texttt{Cur}(\texttt{Grab}; \texttt{Access}(\underline{1}); \texttt{Return}); \cdots \parallel 42 \parallel \varepsilon \parallel 42 \cdot 9 \cdot \bullet \cdot \varepsilon \parallel \varepsilon \rangle \\
&\longmapsto \langle \texttt{Apply}; \texttt{Halt} \parallel ((\texttt{Grab}; \texttt{Access}(\underline{1}); \texttt{Return}), \varepsilon) \parallel \varepsilon \parallel 42 \cdot 9 \cdot \bullet \cdot \varepsilon \parallel \varepsilon \rangle \\
&\longmapsto \langle \texttt{Grab}; \texttt{Access}(\underline{1}); \texttt{Return} \parallel ((\texttt{Grab}; \texttt{Access}(\underline{1}); \texttt{Return}), \varepsilon) \parallel 42 \cdot \varepsilon \parallel 9 \cdot \bullet \cdot \varepsilon \parallel (\texttt{Halt}, \varepsilon) \cdot \varepsilon \rangle \\
&\longmapsto \langle \texttt{Access}(\underline{1}); \texttt{Return} \parallel ((\texttt{Grab}; \texttt{Access}(\underline{1}); \texttt{Return}), \varepsilon) \parallel 9 \cdot 42 \cdot \varepsilon \parallel \bullet \cdot \varepsilon \parallel (\texttt{Halt}, \varepsilon) \cdot \varepsilon \rangle \\
&\longmapsto \langle \texttt{Return} \parallel 42 \parallel 9 \cdot 42 \cdot \varepsilon \parallel \bullet \cdot \varepsilon \parallel (\texttt{Halt}, \varepsilon) \cdot \varepsilon \rangle \\
&\longmapsto \langle \texttt{Halt} \parallel 42 \parallel \varepsilon \parallel \varepsilon \parallel \varepsilon \rangle
\end{aligned}
$$

The only closures that appear in this program are on the return stack and the accumulator register. Since they may be returned from functions and then used as function arguments later, only closures in the environment must be heap-allocated. This means that our example program performs *no* heap allocation. Only if the example had some non-strict applications or one of the arguments to the function was a function, would heap allocation be required.

Like SECD, this machine evaluates in a right-to-left (observe the compilation of applications) order which is not actually compatible with the call-by-value $\lambda$-calculus with effects other than non-termination.

*5.3.2 STG Machine.* As mentioned in Section 4, many lazy language implementations made use of super-combinator graph-reduction machines. The Spineless Tagless G-machine [64, 66], which is still used in GHC today, does away with graph-reduction. That is to say, it no longer opts to keep an explicit representation of the source code's structure, rather it uses a global heap that hold closures representing partial applications and holding free variables. And instead of lambda-lifting, which turned every free variable into a function argument later captured in the graph, the STG-machine simply constructs closures at runtime for the free variables. Thus, the unique evaluation method of call-by-need compilers was ousted in favor of the approaches already found in call-by-value abstract machines like SECD and ZAM. Also like the ZAM, it is a push/enter machine that is optimized to handle multiple applications.

Following the trend of lazy language discussion in the literature [38, 44], its source language is in a special form of $\Lambda$-terms wherein all arguments to functions are atomic (*i.e.* variables or primitive values), the free variables of functions are explicitly stated, and all functions occur bound to let-expressions. The syntax is the following:

$$
\begin{aligned}
M, N, L \in \quad & \textit{Expression} && ::= \texttt{let}\ x = \lambda[\overline{x}]\overline{y}.\, M\ \texttt{in}\ N \mid x\ \overline{\alpha} \mid n \\
\alpha \in \quad & \textit{Atom} && ::= x \mid n
\end{aligned}
$$

As notation, I use $\overline{x}$ to denote a list and, in the case of $\lambda$-expressions, the variable list enclosed in brackets $[\overline{x}]$ refers to the free variables of the function. Given this syntax, unevaluated expressions, *i.e.* thunks, would be written $\lambda[\overline{x}].\, M$ since they have no formal parameters. Note, I have removed algebraic data types and mutual recursion from the STG language to focus on high-order functions.

The machine's syntax and transitions are given in Figure 5. In addition to my language not support everything in Peyton Jones' work, I have removed global bindings. An unfortunate side-effect is that any thunk that computes a number will not be memoized. Adding call-by-need data which contain primitive numbers is the way that their computations are memoized in the full STG-machine [65]. Herein, the only computations that are memoized is that of partial function

**Syntax**

$$
\begin{array}{rcll}
C \in & \textit{Code} & ::= & \texttt{Eval}\ M\ E \mid \texttt{Enter}\ l \mid \texttt{Int}\ n \\
A \in & \textit{Argument Stack} & ::= & \varepsilon \mid V \cdot A \\
U \in & \textit{Update Stack} & ::= & \varepsilon \mid (A, l) \cdot U \\
H \in & \textit{Heap} & = & \textit{Location} \rightharpoonup \textit{Value} \\
V, W \in & \textit{Value} & ::= & n \mid l \\
 & \textit{Machine State} & ::= & \langle C \parallel A \parallel U \parallel H \rangle
\end{array}
$$

**Transitions**

$$
\begin{array}{rcl}
\langle \texttt{Eval}\ (\texttt{let}\ x = \lambda[\overline{x}]\overline{y} \to M\ \texttt{in}\ N)\ E \parallel A \parallel U \parallel H \rangle & \longmapsto & \langle \texttt{Eval}\ N\ E[x \mapsto l] \parallel A \parallel U \parallel H' \rangle \\
 & & \quad \textbf{where}\ H' = H[l \mapsto (\overline{\mathrm{val}(E, x)}, \lambda[\overline{x}]\overline{y} \to M)] \\
\langle \texttt{Eval}\ (f\ \overline{\alpha})\ E \parallel A \parallel U \parallel H \rangle & \longmapsto & \langle \texttt{Enter}\ E(f) \parallel \overline{\mathrm{val}(E, x)} \cdot A \parallel U \parallel H \rangle \\
\langle \texttt{Eval}\ f\ E[f \mapsto n] \parallel A \parallel U \parallel H \rangle & \longmapsto & \langle \texttt{Int}\ n \parallel A \parallel U \parallel H \rangle \\
\langle \texttt{Eval}\ n\ E \parallel A \parallel U \parallel H \rangle & \longmapsto & \langle \texttt{Int}\ n \parallel A \parallel U \parallel H \rangle \\
 & & \\
\langle \texttt{Enter}\ l \parallel \overline{V} \cdot A \parallel U \parallel H \rangle & \longmapsto & \langle \texttt{Eval}\ M\ (\overline{w \mapsto W}, \overline{x \mapsto V}) \parallel A \parallel U \parallel H \rangle \\
 & & \quad \textbf{where}\ H(l) = (\overline{W}, \lambda[\overline{w}]\overline{x} \to M) \\
 & & \qquad |\overline{V}| = |\overline{x}| > 0 \\
\langle \texttt{Enter}\ l \parallel \overline{V} \parallel (A', l') \cdot U \parallel H \rangle & \longmapsto & \langle \texttt{Enter}\ l \parallel \overline{V} \cdot A' \parallel U \parallel H \rangle \\
 & & \quad \textbf{where}\ H(l) = (\overline{W}, \lambda[\overline{w}]\overline{x} \to M) \\
 & & \qquad |\overline{V}| < |\overline{x}| > 0 \\
 & & \qquad H' = H[l' \mapsto ((l, \overline{V}), \lambda[f, \overline{x}].\, f\ \overline{x})] \\
\langle \texttt{Enter}\ l \parallel A \parallel U \parallel H \rangle & \longmapsto & \langle \texttt{Eval}\ M\ \overline{w \mapsto V} \parallel \varepsilon \parallel (A, l) \cdot U \parallel H \rangle \\
 & & \quad \textbf{where}\ H(l) = (\overline{V}, \lambda[\overline{w}] \to M)
\end{array}
$$

$$
\mathrm{val}(E, \alpha) = \begin{cases} E(x) & \alpha = x \\ n & \alpha = n \end{cases}
$$

Fig. 5. STG machine

applications. As a result of this omission, the return stack is unnecessary since it was used to hold the continuations for returning from case expressions.

So the machine I present is made of four parts: a code element, an argument stack, an update stack, and a heap. The code element is either an evaluation of an expression with an environment, a thunk entry, or halting the computation on an integer. Values can either be labels or numbers whereas in the source language atomic arguments could either be variables or numbers. An update stack contains an argument stack and the label to be updated. For transitions, the evaluation transitions follow closely those of other push/enter machines; that is, let-expressions extend the heap with a closure and local environment with its point, and function applications push arguments on the stack then proceed to a closure entry state. In the restricted STG language setting, eval-statements stepping into an `Int` $n$ state will halt the machine. For the closure entering transitions, there is a rule for entering a function with enough arguments on the stack, entering a functions with too few arguments on the stack, and entering a thunk closure. Only the last of those pushes an update frame; as seen with big-step environment semantics: functions are not memoized. For the under-applied case, the update frame on the stack is consumed so that its closure is mutated to a partial application, and the argument stack that it contains is added underneath the current one. Thereby, more arguments are made available to the under-applied function and the work made to get to this state is saved.

As an example demonstrating how multiple application works in the STG machine, consider the following program wherein $g$ is a function with arity two and $h$ appears as $g$ applied to only one argument:

$$\begin{array}{l}\texttt{let } g = \lambda[]x, y.\, x \texttt{ in} \\ \quad \texttt{let } h = \lambda[g].\, g\ 42 \texttt{ in} \\ \quad\quad h\ 0\end{array}$$

Running it on the machine would yield the following trace:

$$\begin{array}{l}\langle \texttt{Eval (let } g = \lambda[]x, y.\, x \texttt{ in } \ldots\texttt{)} \ \varepsilon \parallel \varepsilon \parallel \varepsilon \rangle \\ \longmapsto \langle \texttt{Eval (let } h = \lambda[g].\, g\ 42 \texttt{ in } \ldots\texttt{)}\ [g \mapsto l_g] \parallel \varepsilon \parallel \varepsilon \parallel [l_g \mapsto (\varepsilon, \lambda[]x, y.\, x)] \rangle \\ \longmapsto \langle \texttt{Eval } (h\ 0)\ [g \mapsto l_g, h \mapsto l_h] \parallel \varepsilon \parallel \varepsilon \parallel [l_g \mapsto (\varepsilon, \lambda[]x, y.\, x), l_h \mapsto (l_g, \lambda[g].\, g\ 42)] \rangle \\ \longmapsto \langle \texttt{Enter } l_h \parallel 0 \cdot \varepsilon \parallel \varepsilon \parallel [l_g \mapsto (\varepsilon, \lambda[]x, y.\, x), l_h \mapsto (l_g, \lambda[g].\, g\ 42)] \rangle \\ \longmapsto \langle \texttt{Eval } (g\ 42)\ [g \mapsto l_g] \parallel \varepsilon \parallel (0 \cdot \varepsilon, l_h) \cdot \varepsilon \parallel [l_g \mapsto (\varepsilon, \lambda[]x, y.\, x), l_h \mapsto (l_g, \lambda[g].\, g\ 42)] \rangle \\ \longmapsto \langle \texttt{Enter } l_g \parallel 42 \cdot \varepsilon \parallel (0 \cdot \varepsilon, l_h) \cdot \varepsilon \parallel [l_g \mapsto (\varepsilon, \lambda[]x, y.\, x), l_h \mapsto (l_g, \lambda[g].\, g\ 42)] \rangle \\ \longmapsto \langle \texttt{Enter } l_g \parallel 42 \cdot 0 \cdot \varepsilon \parallel \varepsilon \parallel [l_g \mapsto (\varepsilon, \lambda[]x, y.\, x), l_h \mapsto (l_g, \lambda[g].\, g\ 42)] \rangle \\ \longmapsto \langle \texttt{Eval } x\ [x \mapsto 42, y \mapsto 0] \parallel \varepsilon \parallel \varepsilon \parallel [l_g \mapsto (\varepsilon, \lambda[]x, y.\, x), l_h \mapsto (l_g, \lambda[g].\, g\ 42)] \rangle \\ \longmapsto \langle \texttt{Int } 42 \parallel \varepsilon \parallel \varepsilon \parallel [l_g \mapsto (\varepsilon, \lambda[]x, y.\, x), l_h \mapsto (l_g, \lambda[g].\, g\ 42)] \rangle\end{array}$$

This example clearly demonstrates how applications are delayed in the machine until all of the arguments are available. Note in the fourth step of the trace wherein an update frame is pushed on the stack. Unfortunately, it does not result an interesting memoization because the partial application $g\ 42$ is memoized to be itself. If, instead, there was some work to be done before arriving at such a memoization, then that work would be memoized.

## 6 COMPILATION THROUGH INTERMEDIATE LANGUAGES

Abstract machines can be improved through the addition of more intelligent runtime systems. There exists another school of compiler design, beginning with Steele's Rabbit compiler for Scheme [78], wherein the compiler is constructed with a series of intermediate languages that represent getting a step closer to real hardware. These intermediate languages are languages in their own right; that is to say, they have a semantics that allows the compiler to perform optimizations. Whereas abstract machines are optimized through ever more complex runtime systems, these compilers optimize by transformations on their increasingly lower-level intermediate representations.

A notable call-by-value example of this style of compiler is the SML New Jersey compiler [9, 10] which has the following compiler pipeline:

$$\texttt{SML} \xrightarrow{\textit{TC/Desugar}} \text{System F} \xrightarrow{\textit{CPS}} \Lambda_{\text{CPS}} \xrightarrow{\textit{Clos. Conv}} \Lambda_{\text{clos}} \xrightarrow{\textit{Explicit Alloc.}} \Lambda_{\text{alloc}} \xrightarrow{\textit{Code Gen.}} \texttt{ASM}$$

A desugaring phase removes the syntactic sugar that occurs in the source language; for instance a let-expression that also pattern-matches $\texttt{let } (x, y) = \ldots \texttt{ in } \ldots$ is transformed into a case-expression. $\Lambda_{\text{CPS}}$ is a key optimization language known as continuation-passing-style for the compiler wherein inlining, which is not sound in call-by-value, is possible and all function calls are tail calls. $\Lambda_{\text{clos}}$ is a closure-converted language, meaning that its semantics does not rely on a runtime system that knows how to construct closures. Finally, the language $\Lambda_{\text{alloc}}$ makes manipulation of memory explicit.

In the realm of lazy language compilers, the premier compiler is the Glasgow Haskell Compiler (GHC) which compiles Haskell through the following intermediate languages:

$$\texttt{Haskell} \xrightarrow{\textit{TC/Desugar}} \text{System } \text{F}_{\text{CJ}} \xrightarrow{\textit{Norm./Erasure}} \texttt{STG} \xrightarrow{\textit{Code Gen.}} \texttt{Cmm} \xrightarrow{\textit{Instr. Select}} \texttt{ASM}$$

Following a similar desugar transformation to SML New Jersey, GHC's enters its core optimization language: System $\text{F}_{\text{CJ}}$. Therein, GHC performs a number of optimizations including inlining (though

not entirely sound), constant folding, and fusion of unboxed operations. Following the core language, GHC enters the STG language which I have discussed before; the language has the advantage of being untyped which allows some optimizations that do not preserve types. The final intermediate language is `Cmm` wherein low-level details like interactions with the garbage collector, *e.g.* allocation, are exposed.

This section discusses these intermediate languages and the transformations between them. First, I discuss continuation-passing-style and its successor administrative-normal-form which were initially created for call-by-value compilers. Next, I expand on a problem with the two approaches above: that there are two different compiler pipelines for different interpretations of a similar source language. I discuss a couple of remedies for this, thunking and call-by-push-value which could merge these two compilers. Finally, I discuss a necessary transformation for any functional language doing C-like code generation: closure-conversion. The desugared languages that are variants of System F will not be addressed since they are, from the point of view of this document, extensions of the source language $\Lambda$.

### 6.1 Continuation-Passing Style

A problem using call-by-value in our intermediate language is that its restricted $\beta$-reduction makes it harder to optimize than a call-by-name language. A fundamental code manipulation transformation, inlining, is not universally applicable. In the program $(\lambda x.\, M)\ (N\ L)$, for instance, the expression $N\ L$ cannot be inlined at compile time because $\beta$-reduction does not allow it. If $N\ L$ were to diverge, then inlining would change the meaning of the program. This is the main motivation for continuation-passing-style CPS to be used as a compiler intermediate representation in SML New Jersey [9, 10]. The transformation or something akin to it is also found in the early Scheme compilers [3, 78]. Additionally, Sabry with his coauthors [71–73] have shown the correctness of these transformations as well as the equations preserved by source and target.

Intuitively, the transformation will convert an entire program into one that manipulate a *continuation* which is a function containing the work left to do in the program. When a source program is a normal form or value, the continuation, or left-over work function, can be applied to the program. When a source program is a computation, *e.g.* an application, the function part of the application must be evaluated and given a new continuation that knows to evaluate the right-hand-side as well. The transformation itself is the following wherein all cases become functions on a continuation $k$:

$$
\begin{aligned}
\mathrm{K}_{\mathcal{V}}[\![c]\!] &= \lambda k.\, k\ c \\
\mathrm{K}_{\mathcal{V}}[\![x]\!] &= \lambda k.\, k\ x \\
\mathrm{K}_{\mathcal{V}}[\![\lambda x.\, M]\!] &= \lambda k.\, k\ (\lambda x.\, \mathrm{K}_{\mathcal{V}}[\![M]\!]) \\
\mathrm{K}_{\mathcal{V}}[\![M\ N]\!] &= \lambda k.\, \mathrm{K}_{\mathcal{V}}[\![M]\!]\ (\lambda m.\, \mathrm{K}_{\mathcal{V}}[\![N]\!]\ (\lambda n.\, m\ n\ k))
\end{aligned}
$$

Examining the function case reveals that the order of evaluation has been enshrined in the syntax of the program, because the left-hand-side of the function is evaluated and given a continuation to evaluate the right-hand-side. This matches the left-to-right order of evaluation presented in the call-by-value operational semantics (Section 3). Another consequence of this transformation is that function calls never return. The left-hand-side of the application $\mathrm{K}_{\mathcal{V}}[\![M]\!]$ is given the continuation that knows how to evaluate the entire rest of the program; therefore, there is no need to return back to this point after evaluating $\mathrm{K}_{\mathcal{V}}[\![M]\!]$. This means that $\mathrm{K}_{\mathcal{V}}[\![M]\!]\ (\lambda m.\, \ldots)$ is a tail call which allows the runtime system to avoid creating a new activation record for called tail call elimination[8].

[8]That this is a tail call is contested by some because applying the continuation $k$ is similar to returning from a function.

Has the transformation achieved its goal? Yes. The call-by-value $\beta$-reduction is always applicable, and therefore so is inlining, since the right-hand-side of an application in the image of $\mathrm{K}_{\mathcal{V}}[\![-]\!]$ is always a function value.

A major theme of this document thus far has been the sundering of implementations because of evaluation strategy; the CPS transformation above is no different. Since call-by-name has an unrestricted $\beta$-reduction, a CPS'ed intermediate language is not necessary. Of course, call-by-need is not call-by-name and does not have a $\beta$-rule; however, inlining can be done at the risk of losing sharing. Despite this lack of necessity, CPS transformations do exist for these non-strict languages and studying them can reveal insights about implementation. For instance, Plotkin [69] showed that a program in continuation-passing style will return the same result regardless of whether or not the target language is evaluated in a call-by-value or call-by-name manner.

The essential difference in call-by-name CPS transformation is that variables will be bound to computations awaiting a continuation. The full translation is the following:

$$\begin{array}{rcl}
\mathrm{K}_{\mathcal{N}}[\![c]\!] & = & \lambda k.\, k\ c \\
\mathrm{K}_{\mathcal{N}}[\![x]\!] & = & x \\
\mathrm{K}_{\mathcal{N}}[\![\lambda x.\, M]\!] & = & \lambda k.\, k\ (\lambda x.\, \mathrm{K}_{\mathcal{N}}[\![M]\!]) \\
\mathrm{K}_{\mathcal{N}}[\![M\ N]\!] & = & \lambda k.\, \mathrm{K}_{\mathcal{N}}[\![M]\!]\ (\lambda m.\, m\ \mathrm{K}_{\mathcal{N}}[\![N]\!]\ k)
\end{array}$$

Since variables will be bound to continuation accepting computations, variable case simply compiles to itself. The application case appropriately continues with $m\ \mathrm{K}_{\mathcal{N}}[\![N]\!]\ k$ stating that evaluating the call $m\ \mathrm{K}_{\mathcal{N}}[\![N]\!]$ will *return* before consuming the continuation $k$.

Okasaki *et al.* [60] present a call-by-need variant of the non-strict CPS transformation. For call-by-name, accessing a CPS'ed variable will return the same CPS'ed delayed expression regardless of how many times this is done. Call-by-need requires that variable lookups are shared, and therefore, the CPS'ed expression must be replaced after its first call. In Okasaki's presentation, this is accomplished by adding mutable references to the target language of CPS and presenting the following transformation (`newk` and `derefk` are CPS variants of creating a new mutable cell and accessing that cell):

$$\begin{array}{rcl}
\mathrm{K}_{\mathcal{N}}[\![c]\!] & = & \lambda k.\, k\ c \\
\mathrm{K}_{\mathcal{L}}[\![x]\!] & = & x \\
\mathrm{K}_{\mathcal{L}}[\![\lambda x.\, M]\!] & = & \lambda k.\, k\ (\lambda x.\, \mathrm{K}_{\mathcal{L}}[\![M]\!]) \\
\mathrm{K}_{\mathcal{L}}[\![M\ N]\!] & = & \lambda k.\, \mathrm{K}_{\mathcal{L}}[\![M]\!]\ (\lambda m.\, \mathtt{newk}\ (\lambda l.\, l := (\lambda k'.\, \mathrm{K}_{\mathcal{L}}[\![N]\!]\ (\lambda n.\, l := (\lambda k''.\, k''\ n); k'\ n)); \\
 & & \qquad\qquad m\ (\lambda k'.\, \mathtt{derefk}\ l\ (\lambda k''.\, k''\ k'))\ k))
\end{array}$$

All the changes from call-by-name are found in the application case. Therein, the continuation handed to the translation of the function first allocates a mutable cell for the argument. That mutable cell is assigned to hold a CPS'ed version of the function argument that will update itself upon access.

Unfortunately, the inclusion of memory side-effects in code makes the target of call-by-need CPS harder to reason about. This begs the question of what is gained by such a translation since lazy compilers make use of $\beta$ for optimization already; though they may lose sharing. Okasaki's study does provide evidence that once a call-by-need language is compiled down to a language that makes explicit control flow, it will require mutable references.

The CPS transformations above make inlining with $\beta$-reduction a valid optimization transformation at the expense of mangling the input program entirely. Flannagan *et al.* [34] propose administrative normal-form achieving the goal of CPS, inlining, but with a much lighter touch. That is, it does not completely mangle our program making everything a tail call; in fact, its output

language is quite legible:

$$M, N \in \quad \textit{Expression} \quad ::= V \mid \texttt{let}\ x = V\ \texttt{in}\ M \mid \texttt{let}\ x = V\ W\ \texttt{in}\ M \mid V\ W$$

A special quality of A-normal form is that it explicitly differentiates tail calls $V\ W$ from those that need to return to a binding $\texttt{let}\ x = V\ W\ \texttt{in}\ M$.

ANF is not strictly better than a CPS language. Kennedy [41] points out that, although $\beta$-reduction is valid in the ANF language, reductions in the language are not closed. Therefore, a compiler making use of an ANF intermediate language would require a renormalization pass after every optimization transformation. He suggests a refinement of CPS as the intermediate language based on a graph. More recently, CPS-like intermediate languages, based on the sequent calculus, have even found their way into lazy compilers because of their ability to represent join points [30, 52]; in contrast with CPS however, these languages have been shown to help optimize the use of data types.

### 6.2 A Unified Compiler Pipeline

Upon introducing his language call-by-push-value, Levy [48] notes the amount of duplicated work that programming language researchers engaged in for the purpose of working with either strict or non-strict languages. Theorems must be proved for one strategy and then the other. Completely separate compilers were constructed for each. Plotkin's work [69] shows that CPS could remedy this work duplication since one target language can be transformed to run on the other. Indeed, part of the motivation for call-by-need CPS from Okasaki *et al.* [60] was to make use of the CPS compiler backends already implemented. Herein, I examine two more approaches to removing this problem by compiling to another intermediate language: thunking and call-by-push-value.

*6.2.1 Thunking.* As seen in the big-step environment semantics, Krivine and STG machines, non-strict language runtimes require the creation of thunk closures. The thunking transformation [35, 60] embeds a non-strict language in a call-by-value one, by adding the construction of thunk closures as a language feature of the target. The thunking transformation is then specified as follows:

$$\begin{aligned} \mathrm{T}[\![x]\!] &= \texttt{force}\ x \\ \mathrm{T}[\![\lambda x.\, M]\!] &= \lambda x.\, \mathrm{T}[\![M]\!] \\ \mathrm{T}[\![M\ N]\!] &= \mathrm{T}[\![M]\!]\ (\texttt{delay}\ \mathrm{T}[\![N]\!]) \end{aligned}$$

Since both call-by-name and call-by-need must create these thunk structures for arguments only, this *one* transformation can be used for both source languages.

Figure 6 presents the target language for thunking and along with rules for many of the types of the semantics seen heretofore. The target language is simply $\Lambda$-terms extended with delay and force expressions. In the syntax, I also specify evaluation contexts for specifying a structural operational semantics. The first semantics given is a reduction rule stating that a delayed expression is eliminated when it interacts with a force expression. In the operational semantics, such a state is arrived at by evaluating the right-hand-side of a force expression until the reduction rule fires. The second semantics given is a non-memoizing environment semantics, which is the semantics that must be used for a call-by-name thunking transformation. Therein, closures are constructed for delay expressions since they may contain free variables. The final semantics is a memoizing environment semantics for a call-by-need target language. It will store delay expressions in the heap and pass around their pointers as values.

The gap between the source and target languages appears large because their is a change in evaluation strategy. However, examining the environment semantics shows that all that has changed is that the encoding of thunk closures, which was done in the source's semantics implicitly, is now

**Syntax**

$$\begin{array}{rrll} M, N, L \in & \textit{Expression} & ::= & x \mid \lambda x.\, M \mid M\ N \mid \texttt{delay}\ M \mid \texttt{force}\ M \\ E \in & \textit{Eval. Ctx.} & ::= & \square \mid E\ N \mid (\lambda x.\, M)\ E \mid \texttt{force}\ E \end{array}$$

**Non-Memoizing Reduction**

$$\texttt{force}\ (\texttt{delay}\ M) \longrightarrow M$$

**Non-Memoizing Environment Semantics**

$$\frac{}{\langle \Sigma \parallel \texttt{delay}\ M \rangle \Downarrow_{\mathcal{V}} (\Sigma, \texttt{delay}\ M)} \qquad \frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}} (\Sigma', \texttt{delay}\ N) \quad \langle \Sigma' \parallel N \rangle \Downarrow_{\mathcal{V}} V}{\langle \Sigma \parallel \texttt{force}\ M \rangle \Downarrow_{\mathcal{V}} V}$$

**Memoizing Environment Semantics**

$$\frac{\texttt{alloc}(\Phi, (\Sigma, \texttt{delay}\ M)) = (\Phi', l)}{\langle \Phi \parallel \Sigma \parallel \texttt{delay}\ M \rangle \Downarrow_{\mathcal{V}!} (\Phi', l)}$$

$$\frac{\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}!} (\Phi', l) \quad \Phi'(l) = (\Sigma', N) \quad \langle \Phi' \parallel \Sigma' \parallel N \rangle \Downarrow_{\mathcal{V}!} (\Phi'', V) \quad \texttt{update}(\Phi'', l, V) = \Phi'''}{\langle \Phi \parallel \Sigma \parallel \texttt{force}\ M \rangle \Downarrow_{\mathcal{V}!} (\Phi''', V)}$$

$$\frac{\langle \Phi \parallel \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}!} (\Phi', l) \quad \Phi'(l) = V}{\langle \Phi \parallel \Sigma \parallel \texttt{force}\ M \rangle \Downarrow_{\mathcal{V}!} (\Phi', V)}$$

Fig. 6. Thunk semantics

marked by delay and force expressions in the target. Additionally, the call-by-name $\beta$-reduction of the source is in some sense preserved in the target language with only call-by-value $\beta$-reduction. This is because `delay` $M$ is a value in the reduction theory and the translation $\mathrm{T}[\![-]\!]$ guarantees that the right-hand-side of every application is a thunk.

*6.2.2 Call-by-Push-Value.* Another approach to unifying the evaluation strategy schism is to use Levy's call-by-push-value [48] which aims to subsume call-by-value and call-by-name instead of converting one into the other. The language combines the best of both worlds (in regards to evaluation strategy) for a compiler intermediate language because it has always applicable $\beta$ and $\eta$-laws for optimization. His work is greatly influenced by Moggi's computational $\lambda$-calculus [57] and therefore resembles monadic programs in Haskell today. Though such a language is not used as an intermediate form for today's languages, a closely related language with similar benefit, the mixed-strategy sequent calculus [30, 52], is used into today's GHC.

The language, seen in Figure 7, is built from two distinct syntactic categories: values and computations. The former contains variables, suspended computations which Levy calls thunks, and constants. The latter is anything that can be evaluated including let-expressions, function applications, and thunk forcing. There are two kinds of let-expressions: those that give a name to a value, written `let` $x = V$ `in` $M$, and those that are a monadic binding operation which sequence two computations and are written $M$ `to` $x$ `in` $N$. The semantics presented in the figure are the big-step presented by Levy [48]. They bear resemblance to both call-by-name and call-by-value. For instance, function application evaluates the left-hand-side and then enters the function body while substituting the formal parameter. This is similar to the call-by-name application rule except that the argument of the function is already restricted to be a value. That restriction is similar to the call-by-value application rule.

Call-by-push-value subsumes call-by-name and -value but requires a transformation from our respective source language. $\mathrm{P}_{\mathcal{N}}[\![-]\!]$ is a transformation from $\Lambda$ into the set of computations from

**Syntax**

$$
\begin{array}{rrll}
V, W \in & \textit{Value} & ::= & x \mid \texttt{thunk}\ M \mid c \\
M, N \in & \textit{Computation} & ::= & \texttt{return}\ V \mid \lambda x.\, M \mid M\ V \mid \texttt{force}\ V \\
 & & & \mid \texttt{let}\ x = V\ \texttt{in}\ M \\
 & & & \mid M\ \texttt{to}\ x\ \texttt{in}\ N \\
T \in & \textit{Terminal Comp.} & ::= & \texttt{return}\ V \mid \lambda x.\, M
\end{array}
$$

**Big-step Semantics**

$$\frac{}{\texttt{return}\ V \Downarrow \texttt{return}\ V} \qquad \frac{}{\lambda x.\, M \Downarrow \lambda x.\, M}$$

$$\frac{M \Downarrow \lambda x.\, N \quad N[V/x] \Downarrow T}{M\ V \Downarrow T} \qquad \frac{V = \texttt{thunk}\ M \quad M \Downarrow T}{\texttt{force}\ V \Downarrow T}$$

$$\frac{M[V/x] \Downarrow T}{\texttt{let}\ x = V\ \texttt{in}\ M \Downarrow T} \qquad \frac{M \Downarrow \texttt{return}\ V \quad N[V/x] \Downarrow T}{M\ \texttt{to}\ x\ \texttt{in}\ N \Downarrow T}$$

Fig. 7. Call-by-push-value

CBPV:

$$
\begin{array}{rcl}
\mathrm{P}_{\mathcal{N}}[\![x]\!] & = & \texttt{force}\ x \\
\mathrm{P}_{\mathcal{N}}[\![c]\!] & = & \texttt{return}\ c \\
\mathrm{P}_{\mathcal{N}}[\![\lambda x.\, M]\!] & = & \lambda x.\, \mathrm{P}_{\mathcal{N}}[\![M]\!] \\
\mathrm{P}_{\mathcal{N}}[\![M\ N]\!] & = & \mathrm{P}_{\mathcal{N}}[\![M]\!]\ (\texttt{thunk}\ \mathrm{P}_{\mathcal{N}}[\![N]\!])
\end{array}
$$

There are striking similarities with the thunking transformation from before. That is, the variable case forces a thunk and the right-hand-side of application is wrapped in a delay (called thunk in CBPV). The constant case is different, however, because the transformation must be a computation. Therefore, the case is the computation which returns a constant.

For call-by-value, the transformation needs to evaluate its argument before calling the function. Thus, the application case uses the monadic let-expression to force the left- then right-hand-side of an application:

$$
\begin{array}{rcl}
\mathrm{P}_{\mathcal{V}}[\![x]\!] & = & \texttt{return}\ x \\
\mathrm{P}_{\mathcal{V}}[\![c]\!] & = & \texttt{return}\ c \\
\mathrm{P}_{\mathcal{V}}[\![\lambda x.\, M]\!] & = & \texttt{return}\ (\texttt{thunk}\ (\lambda x.\, \mathrm{P}_{\mathcal{V}}[\![M]\!])) \\
\mathrm{P}_{\mathcal{V}}[\![M\ N]\!] & = & \mathrm{P}_{\mathcal{V}}[\![M]\!]\ \texttt{to}\ x\ \texttt{in}\ \mathrm{P}_{\mathcal{V}}[\![N]\!]\ \texttt{to}\ y\ \texttt{in}\ (\texttt{force}\ x)\ y
\end{array}
$$

The function case appears a bit odd as well. Why is it wrapped in a thunked return statement while the call-by-name transformation simply returned a function? Indeed, this has to do with preserving the semantics of call-by-value in the presence of effects like printing output. In call-by-value, the argument needs to be evaluated before entering the function; therefore, if the argument prints 1 then the program should print 1. However, in the call-by-value structural operational semantics the left-hand-side of the application is performed before the right-hand-side. So if the left-hand-side printed 2 before returning a function, then that effect should be registered before the effects of evaluating the right-hand-side. The strange function and application cases in this call-by-value to call-by-push-value transformation is to get this behavior correct.

Of course, no one actually implements call-by-name languages in practice so a sharing version of the language is necessary. Fortunately, McDermott and Mycroft extend call-by-push-value to consider call-by-need evaluation [53].

### 6.3 Closure-Conversion

Of the intermediate language seen so far, one focused on enabling more optimizations for the intermediate language and the others focused on avoiding duplication of work from different evaluation strategies. Now, I will focus on a transformation that is necessary if our compiler uses C or LLVM IR as its backend: *closure-conversion*. Since those target languages do not have closure constructing higher-order functions, this transformation converts those kinds of functions in the $\Lambda$ source language into global C-like functions that know nothing of closures. Since this transformation essentially eliminates the notion of higher-order function, it is usually performed at the end of the compilation pipeline. Such a transformation was first seen in Steele's Rabbit compiler for Scheme [78] and later SML New Jersey [9]. It did not make it into the lazy language compiler GHC until they stopped using the G-machine which manipulated graphs [66].

The transformation accomplishes its goal by generating code for constructing closures of an environment and a code pointer for functions. Whereas the big-step environment semantics and abstract machines seen earlier require that the semantics or runtime take on the responsibility of constructing and unpacking closures, in a closure-converted program the construction of closures is embedded in the syntax of the program thereby requiring a simpler runtime system. In fact, function applications in a closure-converted programs can be implemented as simple jumps with an argument on the stack.

To get an intuition for the transformation, consider as an example, the following program:

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

which is closure-converted into the target language as:

$$\begin{array}{l}\texttt{let } x = (\texttt{let } y = 1 + 2 \texttt{ in } ((y), \lambda((y), z).\, y + z)) \texttt{ in} \\ \quad (\pi_1\ x)\ (\pi_0\ x, 3) + (\pi_1\ x)\ (\pi_0\ x, 4)\end{array}$$

First, let me point out, this is a *call-by-value* closure-conversion similar to that found in Minamide *et al.* [54]. The $\lambda$-expression from the source program has been replaced by a pair containing a tuple of free variables—in this case, only $y$—and a function which knows the structure of that tuple. The function, like the super-combinators shown earlier, is a scope-independent function. The transformation itself is defined recursively:

$$\begin{array}{rcl}\mathrm{CC}[\![x]\!] & = & x \\ \mathrm{CC}[\![c]\!] & = & c \\ \mathrm{CC}[\![\lambda x.\, M]\!] & = & ((y_0, \ldots, y_n), \lambda z.\, \texttt{case } z \texttt{ of } ((y_0, \ldots, y_n), x) \to \mathrm{CC}[\![M]\!]) \\ & & \textbf{where } y_0, \ldots, y_n = \mathrm{FV}(\lambda x.\, M) \\ \mathrm{CC}[\![M\ N]\!] & = & \texttt{case } \mathrm{CC}[\![M]\!] \texttt{ of } (y, z) \to z\ (y, \mathrm{CC}[\![N]\!])\end{array}$$

The translation is more verbose than the example above which takes let-expressions and projections $\pi_x$ as syntactic sugar.

Closure-conversion's target language and semantics are presented in Figure 8. As the translation implied, the target language is simply the $\Lambda$-terms extended with products; though, in a typed program existential types are required to preserve types when closure-converting. For the semantics of the target language, I use a big-step environment semantics because it reveals that closures are not constructed by the runtime system. Whereas the big-step environment semantics for call-by-value given in Section 3 would construct closures for functions and unpack them in applications, the target language of closure-conversion already considers functions to be results and simply enters the body of the function with the formal parameter in applications, which is like C function calls.

**Syntax**

$$M, N, L \in \quad \textit{Expression} \quad ::= x \mid \lambda x.\, M \mid M\ N \mid (M_0, \ldots, M_n) \mid \texttt{case}\ M\ \texttt{of}\ (x_0, \ldots, x_n) \to N$$

**Big-step Semantics**

$$\frac{\Sigma(x) = V}{\langle \Sigma \parallel x \rangle \Downarrow V} \qquad \frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow \lambda x.\, M} \qquad \frac{\langle \Sigma \parallel M \rangle \Downarrow \lambda x.\, L \quad \langle \Sigma \parallel N \rangle \Downarrow W \quad \langle x \mapsto W \parallel L \rangle \Downarrow V}{\langle \Sigma \parallel M\ N \rangle \Downarrow V}$$

$$\frac{\langle \Sigma \parallel M_0 \rangle \Downarrow V_0 \quad \ldots}{\langle \Sigma \parallel (M_0, \ldots) \rangle \Downarrow (V_0, \ldots)} \qquad \frac{\langle \Sigma \parallel M \rangle \Downarrow (W_0, \ldots) \quad \langle \Sigma, x_0 \mapsto W_0, \ldots \parallel N \rangle \Downarrow V}{\langle \Sigma \parallel \texttt{case}\ M\ \texttt{of}\ (x_0, \ldots, x_n) \to N \rangle \Downarrow V}$$

Fig. 8. Strict target language for closure-conversion

Since our closure-converted program can be run correctly in this closure-free/closure-ignorant runtime, the goal of embedding closures in the syntax is achieved.

*6.3.1 Closure Representation.* As closure-conversion is a well studied and popular implementation technique, there exists a lot of work on how to optimize the closures created [10, 54, 76, 78]. These optimizations manipulate the structure of the closure and what is to be included therein. For instance, in the translation above the free variables of a function are captured in data structure which is a simple array of values. This means that the code for constructing a closure runs in a time proportional to the number of free variables in its body and looking up a variable in this data structure in constant time. Instead, a more complicated environment representations can be created by linked lists of closures providing more efficient construction of closures [76, 77]. Therein, the construction of the closure is much less since it has a link to the previous environments closure and it only needs to add the newly introduced free variables. There is a trade off, however, because now looking up a variable stored in the closure's environment is slower since it involves traversing a linked list.

*6.3.2 Non-strict Closure-Conversions by Thunking.* Non-strict closure-conversions [79] are made more complicated because unevaluated expressions are considered values in these languages, *i.e.* thunk closures. As result of maintaining the environment for these expressions, there are many more closures for both the source and target languages for non-strict evaluation when compared to a strict one. In fact, the new thunk closures are different in many ways to those required by the strict transformation.

Looking at non-strict closure-conversion, consider again the program:

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

This is closure-converted in a non-strict manner into the following program (I avoid the necessary closure-conversion for $x$, 3, and 4 for clarity):

$$\begin{array}{l}\texttt{let}\ x = \texttt{let}\ y = ((), \lambda().\, 1 + 2)\ \texttt{in} \\ \qquad\qquad ((y), \lambda((y), z).\, (\pi_1\ y)\ (\pi_0\ y) + z) \\ \texttt{in}\ (\pi_1\ x)\ (\pi_0\ x, 3) + (\pi_1\ x)\ (\pi_0\ x, 4)\end{array}$$

In addition to the closures constructed for functions like in call-by-value closure-conversion, a thunk closure is also necessary for let-bound expressions and function arguments. A less obvious part of this transformation is that the target language is strict! If the let-expression and the pair $((y), \lambda((y), z).\, (\pi_1\ y)\ (\pi_0\ y) + z)$ were not evaluated strictly, then the runtime system would have

no idea what the variable $y$ is. To understand why, consider the big-step environment evaluation rule for a non-strict pair:

$$\frac{}{\langle \Sigma \parallel (M_0, \ldots, M_n) \rangle \Downarrow_{\mathcal{N}} (\Sigma, (M_0, \ldots, M_n))}$$

Since non-strict data are not evaluated until forced by their context (like functions which are also codata), a closure is needed to provide the correct environment later. This is why the target language for closure conversion is forced to used strict functions and data instead of their non-strict counterparts; that is, strict data will lookup the values of the free variables in the generated closure structure. Additionally, it follows that since there is nothing in the non-strict target language other than strict functions and products, it is a strict language. The semantics still preserves the non-strict evaluation of the source language because the expressions, like $1 + 2$ in the example, are hidden behind $\lambda$-expressions and will not be evaluated until applied to their environment.

Having a strict target language complicates memoizing non-strict closure-conversion. In the example above, the second access to the variable $y$ would have to recompute $2 + 1$ since there is no mechanism to share this computation. In order to share, the thunk bound to $y$ must be replaced, or mutated, after its first evaluation. Obviously, this can be accomplished by adding mutable heap objects wherein $y$ would not be a reference to a thunk in the environment rather a heap cell which initially contains a thunk. With this in mind, the handling of the thunk $y$ in the example above would be transformed for lazy closure-conversion as the following:

$$\begin{array}{l}
\texttt{let } x = \texttt{let } y = \texttt{new (inr } ((), \lambda().\, 1 + 2)) \texttt{ in} \\
\qquad ((y), \lambda((y), z).\ \texttt{(case !}y \texttt{ of} \\
\qquad\qquad \texttt{inl } v \to v \\
\qquad\qquad \texttt{inr } (e, f) \to \texttt{let } y' = f\ e \texttt{ in } y := \texttt{inl } y'; y') + z) \\
\texttt{in } (\pi_1\ x)\ (\pi_0\ x, 3) + (\pi_1\ x)\ (\pi_0\ x, 4)
\end{array}$$

At the definition site, $y$ is allocated as a tagged thunk on the heap; that is, `inr` tags the thunk as unevaluated. At the call site, the value of the heap cell is accessed; if it contains a value (`inl` $v$), then the value is returned; otherwise, it is a thunk (`inr` $(e, f)$) which is then evaluated, the heap cell is updated with the resulting value, and then that value is returned. Function closures receive no different treatment than they did in call-by-name or call-by-value. This is because no sharing can occur for functions because they depend on their argument which may be different for each call.

Using the thunking transformation I presented in Section 6.2, closure-conversion from all three source evaluation strategies can be defined in terms of only the call-by-value closure extended with conversions for delay and force expressions:

$$\begin{array}{rcl}
\mathrm{CC}_{\mathcal{V}} & = & \mathrm{CC} \\
\mathrm{CC}_{\mathcal{N}} & = & \mathrm{CC}_{\mathrm{T}} \circ \mathrm{T} \\
\mathrm{CC}_{\mathcal{L}} & = & \mathrm{CC}_{\mathrm{Tmemo}} \circ \mathrm{T}
\end{array}$$

These extended translations are given in Figure 9. Performing a non-strict closure-conversion through thunking differs from the non-strict conversions originally given by Sullivan *et al.* [79], but appears as a synthesis of Hatcliff and Danvy's work which showed that call-by-name CPS can be decomposed into thunking then call-by value CPS [35] and Okasaki *et al.*'s work on call-by-need CPS [60] with the goal of closure-conversion.

## 7 REASONING ABOUT IMPLEMENTATIONS

Thus far, I have focused on the mechanisms for evaluation of $\lambda$-calculus programs and how they change given a different evaluation strategy. Do these different semantics respect that of the

**Non-memoizing**

$$\begin{array}{rcl}
\mathrm{CC_T}[\![\texttt{delay}\ M]\!] & = & ((y_0,\ldots,y_n), \lambda(y_0,\ldots,y_n).\,\mathrm{CC_T}[\![M]\!]) \\
 & & \textbf{where}\ y_0,\ldots,y_n = \mathrm{FV}(M) \\
\mathrm{CC_T}[\![\texttt{force}\ M]\!] & = & \texttt{case}\ \mathrm{CC_T}[\![M]\!]\ \texttt{of}\ (e,f) \to f\ e
\end{array}$$

**Memoizing**

$$\begin{array}{rcl}
\mathrm{CC_{Tmemo}}[\![\texttt{delay}\ M]\!] & = & \texttt{new}\ (\texttt{inr}\ ((y_0,\ldots,y_n), \lambda(y_0,\ldots,y_n).\,\mathrm{CC_{Tmemo}}[\![M]\!])) \\
 & & \textbf{where}\ y_0,\ldots,y_n = \mathrm{FV}(M) \\
\mathrm{CC_{Tmemo}}[\![\texttt{force}\ M]\!] & = & \texttt{let}\ l = \mathrm{CC_{Tmemo}}[\![M]\!]\ \texttt{in} \\
 & & \quad \texttt{case}\ !l\ \texttt{of} \\
 & & \quad\quad \texttt{inl}\ v \to v \\
 & & \quad\quad \texttt{inr}\ (e,f) \to \texttt{let}\ v = (f\ e)\ \texttt{in}\ (l := \texttt{inl}\ v; v)
\end{array}$$

Fig. 9. Thunked closure-conversion

original calculus? Especially with machines like SECD wherein evaluation requires manipulation of a machine state with multiple stacks, it is not obvious that they will evaluate a program to the same result. Closure-conversion is another non-obvious case of correctness since a function is transformed into a data structure that is later manipulated by more generated code. In this section, I show some of the properties of these implementation techniques. I focus on abstract machines and compilation with program transformations because these are the approaches found in today's language implementations. I answer the following: how abstract machines reflect the semantics of the source language; how a static semantics of the sub-terms of programs, *i.e.* typing, is preserved by program transformations; and, how a stronger technique known as logical relations is used to prove various properties relating to the semantics of the source language.

Interestingly, incorporating call-by-need is not as straight forward as other strategies; therefore, I discuss separately how these techniques must have extra reasoning for heaps. That is not to say that correctness of call-by-need implementations do not exist [4–6, 67, 75]. Indeed, their reasoning covers all of the evaluation strategies without call-by-need languages without trouble. Piróg *et al.* is a special case because they consider the full STG machine. Extending this work further is the work of Encina *et al.* [24–26]. Whereas, the previous approaches defined their abstract machine over the source syntax, they give some form of translation of the source language to a machine.

Though I will not explore them here, two other well used approaches to reasoning about implementation are denotational methods and bisimulation. Denotational methods rely on the mathematical models of the $\lambda$-calculus proposed by Scott [74]. Therein, $\Lambda$-terms are mapped to complete partial orders. Proving a transformation, *e.g.* closure-conversion, correct requires, first, denotation for both the source and target languages, then it must be shown that before and after the transformation the denotation is the same. Launchbury often proves correctness of call-by-need transformations by showing how image of a transformation is related to the denotational semantics of the language [44, 65]. Bisimulation based approaches are another case of reasoning that I omit here [2, 39, 46, 47]. These methods describe the observable behavior of programs and correctness is defined as having no observable difference.

### 7.1 Machine Reflection

One way to show some correctness, which I refer to as machine reflection, is demonstrated by Leroy in his technical report for the ZAM [45]. Therein, he shows that every transition of an abstract machine corresponds to an equality derivable from an equational theory. He shows that this is

the case for the Krivine machine and the calculus of explicit substitutions. A similar approach to correctness is given by Krivine himself [42] and Sestoft [75].

For example, to show that the Krivine machine in Figure 3 is correct, I must provide a translation from a machine state *back* into a term from the calculus, *i.e.* the source language. This is the *reflection*. Intuitively, the environment $\Sigma$ is a substitution applied to the term $M$ and the call stack $K$ are an ordered set of expressions to which $M$ is applied. Thus, the translation of a machine state is the following (I assume $M$ is a De Bruijn expression):

$$[\![\langle \Sigma \parallel M \parallel (\Sigma_0, N_0) \cdots (\Sigma_n, N_n) \cdot \varepsilon \rangle]\!] = M\{[\![\Sigma]\!]\}\ N_0\{[\![\Sigma_0]\!]\} \cdots N_n\{[\![\Sigma_n]\!]\}$$

Translating an environment into a substitution is not all that different from a call stack since it is also a sequence of closures:

$$[\![(\Sigma_0, N_0), \ldots, (\Sigma_n, N_n)]\!] = N_0\{[\![\Sigma_0]\!]\} \cdots N_n\{[\![\Sigma_n]\!]\} \cdot \mathtt{id}$$

The theorem makes use of the an equational theory denoted by $\beta\sigma \vdash M = M'$. This is just an equality derived from the reduction theory (recall that De Bruijn has a $\beta$ rule and a set of $\sigma$ rules) with the inclusion of a symmetry rule: $M \longrightarrow M' \Longrightarrow M = M'$ and $M = M' \Longrightarrow M' = M$.

Theorem 7.1 (Machine Reflection). *If* $\langle \Sigma \parallel M \parallel K \rangle \longmapsto \langle \Sigma' \parallel M' \parallel K' \rangle$, *then* $\beta\sigma \vdash [\![\langle \Sigma \parallel M \parallel K \rangle]\!] = [\![\langle \Sigma' \parallel M' \parallel K' \rangle]\!]$.

For the proof, there are only 3 cases to consider: one for each transition of the Krivine machine. In addition to the call-by-name Krivine, Leroy also provides a Krivine machine that evaluates call-by-value from right-to-left like the SECD machine. That machine also reflects the source semantics with a similar argument.

### 7.2 Type Preservation

Another property that can be proved about our semantics and transformations is that of type preservation. Before I can describe the assurances that this gives us of correctness, I must describe what types are. *Types*, which I have avoided discussing thus far, are static properties of $\Lambda$-terms closely connected to logic that govern how various syntax can be used. For instance, a $\lambda$-expression behaves like a function type which consumes an argument type and uses the argument to produce an output type. In natural deduction, this corresponds to an implication connective $A \supset B$ which states that $B$ can be proved given a proof of $A$. Herein, I make use of the simply typed $\lambda$-calculus which only contains simple function types and integers:

$$\tau, \rho \in \mathit{Type} ::= \mathtt{int} \mid \tau \to \rho$$

$\Lambda$-terms are described as well-typed if a typing derivation can be constructed from their syntax using the following rules (where $\Gamma$ is a mapping from variables to types):

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash c : \mathtt{int}} \qquad \frac{\Gamma, x{:}\tau \vdash M : \rho}{\Gamma \vdash \lambda x.\, M : \tau \to \rho} \qquad \frac{\Gamma \vdash M : \tau \to \rho \quad \Gamma \vdash N : \rho}{\Gamma \vdash M\ N : \rho}$$

Now, type preservation arguments say that the type of a $\Lambda$-term is identical before and after a transformation, be it a computation or compilation step. This is a desirable property to have. For instance, in the expression $M{+}5$, $M$ is expected to behave like an integer. If $M$ steps to some $M'$, then our program ought to be able to treat that as an integer as well. For compilation, type preservation is an important property because the types encode how a term is expected to behave in the target language. Additionally, types can even be used to speed up runtime operations. Morrisett *et al.* [58] show that an SML/NJ style compiler can preserve types all the way from its intermediate language, System F, to an assembly language with types. Finally, there are a number of properties, like strong

normalization, that can be proved given a well-typed program; and therefore, a transformation preserving types may also preserve these properties.

As an important case study, consider the type preservation of closure-conversion. Recall that the transformation converts a source $\lambda$-expression into a pair of a code pointer and an environment. Firstly, that means that showing that types are preserved requires a translation of types in addition to the translation of terms; this is because a function type will become some data type. For example, $\lambda x.\, x + y : \texttt{int} \to \texttt{int}$ may become a program $(y, \lambda(y, x).\, x + y) : \texttt{int} \times (\texttt{int} \times \texttt{int} \to \texttt{int})$ where pairs have the following typing rules:

$$\frac{\Gamma \vdash M_0 : \tau_0 \quad \dots}{\Gamma \vdash (M_0, \dots) : \tau_0 \times \cdots} \qquad \frac{\Gamma \vdash M : (\rho_0, \dots) \quad \Gamma, x{:}\rho_0, \cdots \vdash N : \tau}{\Gamma \vdash \mathsf{case}\ M\ \mathsf{of}\ (x_0, \dots) \to N : \tau}$$

However, this naïve approach does not work for type preservation! Two functions with the same type but different free variables will not have the same type after the transformation, *e.g.* $\lambda x.\, y$ and $\lambda x.\, x$. Minamide *et al.* [54] remedy this problem by extending the type system of the target language with existential types. Therein, the type of the free variables are hidden as existential; the examples from before will have the type $\exists X.\, X \times (X \times \texttt{int} \to \texttt{int})$. The typing rules for existential types are the following:

$$\frac{\Gamma \vdash M : \tau[\rho/X]}{\Gamma \vdash \mathsf{pack}\ M : \exists X.\, \tau} \qquad \frac{\Gamma \vdash M : \exists X.\, \rho \quad \Gamma, X, x{:}\rho \vdash N : \tau}{\Gamma \vdash \mathsf{unpack}\ M\ \mathsf{as}\ x\ \mathtt{in}\ N : \tau}$$

In regards to the operational semantics, pack-expressions behave like any call-by-value data type (*e.g.* pairs) and unpack-expressions behave like case-expressions evaluating their argument $M$ and binding it in the body $N$.

To make use of the existential type solution, I must augment the transformation given earlier so that pack-expressions and unpack-expressions are inserted at the correct location; that is, in the function case and application case, respectively.

$$\begin{aligned}
\mathrm{CC}[\![\lambda x.\, M]\!] &= \mathsf{pack}\ ((y_0, \dots, y_n), \lambda z.\, \mathsf{case}\ z\ \mathsf{of}\ ((y_0, \dots, y_n), x) \to \mathrm{CC}[\![M]\!]) \\
&\quad \textbf{where}\ y_0, \dots, y_n = \mathrm{FV}(\lambda x.\, M) \\
\mathrm{CC}[\![M\ N]\!] &= \mathsf{unpack}\ \mathrm{CC}[\![M]\!]\ \mathsf{as}\ x\ \mathtt{in} \\
&\quad\ \ \mathsf{case}\ x\ \mathsf{of}\ (y, z) \to z\ (y, \mathrm{CC}[\![N]\!])
\end{aligned}$$

And types are translated from source to target as the following:

$$\begin{aligned}
\mathrm{CC}[\![\tau \to \rho]\!] &= \exists X.\, X \times (X \times \mathrm{CC}[\![\tau]\!] \to \mathrm{CC}[\![\rho]\!]) \\
\mathrm{CC}[\![\texttt{int}]\!] &= \texttt{int}
\end{aligned}$$

The type preservation theorem contains extra translations for all of the variables in the type environment $\Gamma$. It can be proved by a simple induction on the source environment.

Theorem 7.2 (Type Preservation of Call-by-Value Closure-conversion). *If* $x_0{:}\tau_0, \dots, x_n{:}\tau_n \vdash M : \tau$, *then* $x_0{:}\mathrm{CC}[\![\tau_0]\!], \dots, x_n{:}\mathrm{CC}[\![\tau_n]\!] \vdash \mathrm{CC}[\![M]\!] : \mathrm{CC}[\![\tau]\!]$.

Of course, the above method only applied to call-by-value closure-conversion. In both call-by-name and call-by-need closure-conversion (or their thunked version), values are thunk closures not normal forms. Therefore, a new type transformation must be given for each. Even before that, abstract thunking itself must be shown to be type preserving. A type translation for thunking is interesting in that there is only a transformation of environment values. First, there are new typing rules for the target language of thunking:

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathsf{delay}\ M : \mathsf{thunk}\ \tau} \qquad \frac{\Gamma \vdash M : \mathsf{thunk}\ \tau}{\Gamma \vdash \mathsf{force}\ M : \tau}$$

The type preservation theorem for thunking only states that the environment is filled with thunked computations instead of their normal forms.

Theorem 7.3 (Type Preservation of Thunking). *If* $x_0{:}\tau_0, \ldots, x_n{:}\tau_n \vdash M : \tau$*, then it follows that* $x_0{:}\texttt{thunk}\ \tau_0, \ldots, x_n{:}\texttt{thunk}\ \tau_n \vdash \mathrm{T}[\![M]\!] : \tau$.

Thunked closure-conversion then requires a new transformation on the new `thunk` types wherein they become existential packages of closed functions from their environment to the normal form transformation.

$$\mathrm{CC_T}[\![\texttt{thunk}\ \tau]\!] = \exists X.\, X \times (X \to \mathrm{CC_T}[\![\tau]\!])$$

Of course, the transformation of terms must also be adjusted to insert and remove packing. As with the transformation itself, Sullivan *et al.* [79] presents this transformation in a single step.

$$\begin{array}{lcl} \mathrm{CC_T}[\![\texttt{delay}\ M]\!] & = & \texttt{pack}\ ((y_0, \ldots, y_n), \lambda(y_0, \ldots, y_n).\, \mathrm{CC_T}[\![M]\!]) \\ & & \textbf{where}\ y_0, \ldots, y_n = \mathrm{FV}(M) \\ \mathrm{CC_T}[\![\texttt{force}\ M]\!] & = & \texttt{unpack}\ \mathrm{CC_T}[\![M]\!]\ \texttt{as}\ x\ \texttt{in}\ \texttt{case}\ x\ \texttt{of}\ (e, f) \to f\ e \end{array}$$

For the memoized closure-conversion used in implementing memoization, the transformation is different still. Firstly, I must extend the typing rules even further to include type preserving mutable references:

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \texttt{new}\ M : \texttt{ref}\ \tau} \qquad \frac{\Gamma \vdash M : \texttt{ref}\ \tau}{\Gamma \vdash\ !M : \tau} \qquad \frac{\Gamma \vdash M : \texttt{ref}\ \tau \quad \Gamma \vdash M : \tau}{\Gamma \vdash M := N : 1}$$

and the standard sum types:

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \texttt{inl}\ M : \tau + \rho} \qquad \frac{\Gamma \vdash M : \rho}{\Gamma \vdash \texttt{inr}\ M : \tau + \rho} \qquad \frac{\Gamma \vdash L : \rho_0 + \rho_1 \quad \Gamma, x{:}\rho_0 \vdash M : \tau \quad \Gamma, y{:}\rho_1 \vdash N : \tau}{\Gamma \vdash \texttt{case}\ L\ \texttt{of}\ \{\texttt{inl}\ x \to M; \texttt{inr}\ y \to N\} : \tau}$$

The type translation turns a thunk type into a reference to either a normal form type or an existential package to compute a normal form type:

$$\mathrm{CC_{Tmemo}}[\![\texttt{thunk}\ \tau]\!] = \texttt{ref}\ (\mathrm{CC_{Tmemo}}[\![\tau]\!] + \exists X.\, X \times (X \to \mathrm{CC_{Tmemo}}[\![\tau]\!]))$$

And finally, packing and unpacking have to be inserted into the term translation at the correct place:

$$\begin{array}{lcl} \mathrm{CC_{Tmemo}}[\![\texttt{delay}\ M]\!] & = & \texttt{new}\ (\texttt{inr}\ (\texttt{pack}\ ((y_0, \ldots, y_n), \lambda(y_0, \ldots, y_n).\, \mathrm{CC_{Tmemo}}[\![M]\!]))) \\ & & \textbf{where}\ y_0, \ldots, y_n = \mathrm{FV}(M) \\ \mathrm{CC_{Tmemo}}[\![\texttt{force}\ M]\!] & = & \texttt{let}\ l = \mathrm{CC_{Tmemo}}[\![M]\!]\ \texttt{in} \\ & & \quad \texttt{case}\ !l\ \texttt{of} \\ & & \quad\quad \texttt{inl}\ v \to v \\ & & \quad\quad \texttt{inr}\ x \to \texttt{unpack}\ x\ \texttt{as}\ (e, f)\ \texttt{in}\ \texttt{let}\ v = (f\ e)\ \texttt{in}\ (l := \texttt{inl}\ v; v) \end{array}$$

The theorem for type preservation of thunked closure-conversion can be stated the same for both memoizing and non-memoizing implementations. The proof follows by induction on the typing derivation.

Theorem 7.4 (Type Preservation of Thunked Closure-conversion). *If* $x_0{:}\tau_0, \ldots, x_n{:}\tau_n \vdash M : \tau$*, then* $x_0{:}\mathrm{CC_{Tmemo}}[\![\tau_0]\!], \ldots, x_n{:}\mathrm{CC_{Tmemo}}[\![\tau_n]\!] \vdash \mathrm{CC_{Tmemo}}[\![M]\!] : \mathrm{CC_{Tmemo}}[\![\tau]\!]$.

**Call-by-Value Relations**

$$
\begin{array}{rcl}
\mathcal{C}[\![\tau]\!] & = & \{(C, C') \mid \forall V.\, C \Downarrow V \Longrightarrow \exists V'.\, C' \Downarrow V' \wedge (V, V') \in \mathcal{V}[\![\tau]\!]\} \\
\\
\mathcal{V}[\![\texttt{int}]\!] & = & \{(c, c) \mid c \in \mathbb{Z}\} \\
\mathcal{V}[\![\tau \to \sigma]\!] & = & \{((\Sigma, V), V) \mid \forall (W, W') \in \mathcal{V}[\![\tau]\!]. \\
& & \quad (\langle \Sigma \parallel V\ W \rangle, \langle \varepsilon \parallel (\pi_1\ V)\ (\pi_0\ V, W) \rangle) \in \mathcal{C}[\![\sigma]\!]\} \\
\\
\mathcal{E}[\![\Gamma]\!] & = & \{(\Sigma, \Sigma') \mid \forall x{:}\tau \in \Gamma.\, (\Sigma(x), \Sigma'(x)) \in \mathcal{V}[\![\tau]\!]\}
\end{array}
$$

**Call-by-Name Relations**

$$
\begin{array}{rcl}
\mathcal{C}[\![\tau]\!] & = & \{(C, C') \mid \forall R.\, C \Downarrow R \Longrightarrow \exists R'.\, C' \Downarrow R' \wedge (R, R') \in \mathcal{R}[\![\tau]\!]\} \\
\\
\mathcal{R}[\![\texttt{int}]\!] & = & \{(c, c) \mid c \in \mathbb{Z}\} \\
\mathcal{R}[\![\tau \to \sigma]\!] & = & \{((\Sigma, V), V') \mid \forall (W, W') \in \mathcal{V}[\![\tau]\!]. \\
& & \quad (\langle \Sigma \parallel V\ W \rangle, \langle \varepsilon \parallel (\pi_1\ V')\ (\pi_0\ V', W') \rangle) \in \mathcal{C}[\![\sigma]\!]\} \\
\\
\mathcal{V}[\![\tau]\!] & = & \{((\Sigma, M), V') \mid (\langle \Sigma \parallel M \rangle, \langle \varepsilon \parallel (\pi_1\ V')\ (\pi_0\ V') \rangle) \in \mathcal{C}[\![\tau]\!]\} \\
\\
\mathcal{C}[\![\Gamma]\!] & = & \{(\Sigma, \Sigma') \mid \forall x{:}\tau \in \Gamma.\, (\Sigma(x), \Sigma'(x)) \in \mathcal{V}[\![\tau]\!]\}
\end{array}
$$

Fig. 10. Logical relations for closure-conversion

### 7.3 Logical Predicates and Relations

Another technique for proving properties of programs is that of logical predicates and relations known as Tait's method [80] or logical relations. The method works by first specifying a number of predicates over $\Lambda$-terms indexed by types; if a term is in the predicate, then it will have some important property. For proving strong normalization [29, 55]), for instance, terms that are in the predicate are terms with the property that they evaluate to a normal form. These predicates can be expanded to binary relations to prove contextual equivalence [7, 31, 68] and compilation correctness [8, 54, 62] wherein the first component of an element of the relation is the source term and the second is the compiled target term.

I will demonstrate the use of logical relations for proving that closure-conversion preserves the source semantics. Establishing correctness for this transformation is not straight forward because a function in the source becomes a pair in the target. (Logical relations are sufficient for such a proof, but I am unaware of discussion that shows they are necessary.) Because of the difference in structure, correctness depends on showing that the pair in the target *behaves* like the function in the source. In the Minamide *et al.* [54] of call-by-value closure-conversion correctness, they specify a type-indexed logical relation for values, terms, and substitutions of the source and target languages. However, like Sullivan *et al.* [79], I elect to construct relations between configurations for the big-step environment semantics since such a relation between source and target better demonstrates the essence of closure-conversion.

Figure 10 presents a family of logical relations for call-by-name and call-by-value closure-conversion. Fitting a theme in this document, call-by-name is more complicated than that of call-by-value since it must consider separately values and the output of computation *i.e.* results. The "top level" relation, $\mathcal{C}$, states that a source configuration is related to a target configuration if the source evaluating to a result implies that the target will evaluate to a related result. A known limitation of this approach is that diverging source computations are related to any target

configurations; however, this is not a problem in a simply typed setting where divergence is not possible. The relation $\mathcal{V}$ is different depending on evaluation strategy. For call-by-name, values are all thunks; and therefore, a source thunk is related to a target thunk unpacking if the two produce related configurations. On the other hand, the call-by-value value relation must relate values. A numeric constant is related to a target constant when they are identical and a function closure is related to a target package if they produce related configurations when applied to related values. Call-by-name normalized terms are described as results; and thus, the result relation $\mathcal{R}$ captures the same ideas as the call-by-value value relation. Finally, the $\mathcal{E}$ relation states that a source environment is related to a target environment for some type environment $\Gamma$ when all of the variables in the type environment have related values. It does not matter that either the source or target environment can contain extra bindings; the type environment dictates where the two must agree.

Using these relations, the adequacy theorem states that any typed term will produce related configurations for closure-conversion. From adequacy, semantic preservation follows. It states that a program computing an integer computes the same integer after closure-conversion.

Theorem 7.5 (Adequacy). *If* $\Gamma \vdash M : \tau$ *and* $(\Sigma, \Sigma') \in \mathcal{E}[\![\Gamma]\!]$, *then* $(\langle \Sigma \parallel M \rangle, \langle \Sigma' \parallel \mathrm{CC}[\![M]\!] \rangle) \in C[\![\tau]\!]$.

Theorem 7.6 (Semantic Preservation). *If* $\Gamma \vdash M : \mathtt{int}$ *and* $\langle \varepsilon \parallel M \rangle \Downarrow n$, *then* $\langle \varepsilon \parallel \mathrm{CC}[\![M]\!] \rangle \Downarrow n$.

### 7.4 Stores and Heaps

The reasoning above was missing call-by-need evaluation. Downen *et al.* [29] have incorporated similar reasoning about the call-by-need sequent calculus, but such calculi—as this document has been exploring—are not well suited for use in practical implementations. Instead, to reason about call-by-need in practical implementations requires reasoning about stores and heaps as seen in environment semantics and abstract machines.

Logical relations for reasoning about heaps do exist in call-by-value languages [7, 31, 68]. These papers discuss contextual equivalence of the strict $\lambda$-calculus with type preserving mutable references; coincidentally, this is the target language for call-by-need closure-conversion. However, when considering the properties of the heap in the call-by-need source language, this call-by-value target language, as presented in Ahmed's dissertation [7], differs in non-trivial ways. First and foremost, a language with mutable references allows one to create cycles in the heap. For instance, the following program stores in $r$ an expression with a reference to $r$:

$$\begin{array}{l} \mathtt{let}\ r = \mathtt{ref}\ (\lambda x.\, x)\ \mathtt{in} \\ \quad \mathtt{let}\ f = \lambda x.\, !r\ x\ \mathtt{in} \\ \quad\quad r := f; !r\ 42 \end{array}$$

Though cycles in the heap are possible in any language with general recursion, this document studies 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 predicates for her language. Therein, an index $i \in \mathbb{N}$ is used to guard the lookup depth of the heap. Her approach is an instance of Kripke-esque or possible worlds logical relations wherein the worlds are some notion of heap with a step-index; that is, a world $W$ is element of $\mathbb{N} \times (\mathit{Location} \rightharpoonup \mathit{Type})$. Each relation/predicate is extended with a notion of world. For instance, the following logical predicate for values used is for type safety in Ahmed [7]:

$$\mathcal{R}[\![\tau \to \sigma]\!] \stackrel{\text{def}}{=} \{(W, \Phi, (\Sigma, \lambda x.\, M)) \mid \forall V \in \mathcal{V}[\![\tau]\!], W'.\, W \sqsubseteq W' \Longrightarrow \langle \Phi' \parallel \Sigma, x \mapsto V \parallel \lambda x.\, M \rangle \in C[\![\sigma]\!]\}$$

A function must now work on any world/heap that is *accessible* from the current world/heap, denoted by $W \sqsubseteq W'$. In her definition of accessibility, a future heap must contain values of the same type as cells of the current heap.

Though this presents a flavor of reasoning about heaps, 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. Miquey and Herbelin [55] work directly with call-by-need languages, and unlike Downen *et al.* [29], they do have a separate heap semantics. They refer to their model as a *realizability*, but it shares many aspects with logical relations. Their heap model rests on a single binary relation on heaps which they call *compatibility* (this is different from the notion of compatibility in a reduction theory):

$$\Phi \diamond \Phi' \stackrel{\text{def}}{=} \forall x \in (\text{dom}(\Phi) \cap \text{dom}(\Phi')).\, \Phi(x) = \Phi'(x)$$

This says that two heaps are compatible if they map to the same terms or coterms (their language is based on the sequent calculus, not the $\lambda$-calculus) for every single variable that they have in common. In their proof of strong normalization, they extend their realizers (logical predicates) with the ability to work with any compatible heap to the one in the realizer. For instance, the realizer for function "strong" values is the following:

$$\mathcal{R}[\![\tau \to \sigma]\!] \stackrel{\text{def}}{=} \{(\Phi, \lambda x.\, M) \mid \forall V \in \mathcal{V}[\![\tau]\!], \Phi'.\, \Phi \diamond \Phi' \Longrightarrow \langle \Phi\Phi', x \mapsto V \parallel M \rangle \in \mathcal{C}[\![\sigma]\!]\}$$

There is an issue lurking here, unfortunately, and it is because of the interaction between their definition of compatibility and the semantics of call-by-need evaluation. Therein evaluation of a variable will replace its thunk in the heap with value which *is not equal*! Indeed, the error in this method's proof is in the case of updating the heap. Moreover, their notion of heap compatibility is odd in that it does not satisfy properties that the call-by-value heap models from Ahmed satisfy. Notably, their notion of heap compatibility is not transitive, which is a property that a heap that monotonically grows during evaluation should enjoy (even with garbage collection it should be observably monotonic).

Noting the problems with Miquey and Herbelin's heap model, Sullivan *et al.* [79] describe the properties necessary for a heap model that may allow the closure-conversion relations to extend to call-by-need evaluation. And like the two previous approaches, their reasoning rests on how to define a binary relation on heaps. They state a few properties that this relations ($\sqsubseteq$) must have. First, if an element of any of the relations is related with a particular heap, then it must be related for any future related heaps. Second, future related heaps are still related when combined with another pair of distinct related heaps. And third, updating any thunk in the heap with the result of its evaluation will produce a valid future heap. Unfortunately, they are unable to specify such a relation and conjecture that its existence would allow them to prove adequacy. They state that the work of Mizuno and Sumii [56] is a promising future direction for searching for such a relation. Therein, they give a relation between call-by-need heaps and call-by-name heaps that has some of the necessary properties.

Given such a relation, Sullivan *et al.* [79] give the logical relations for call-by-need evaluation seen in Figure 11. Every relation, save that of configurations, is augmented with a set of related heaps as input; this is because environment depend on heap objects. The configuration relation $\mathcal{C}$ has the added proviso requiring that configurations must work with any future heap. The result relation, which is indexed by related heaps from the start of the computation that produced them, states that the heaps within are future heaps of the starting heaps and that the answers are related given the final heaps. The answer relation $\mathcal{A}$ is not all that different from the $\mathcal{V}$ relation from call-by-value: only the function-type case has changed to reflect that all of the call-by-need values

$$
\begin{array}{rcl}
C[\![\tau]\!] & \stackrel{\text{def}}{=} & \{(\langle \Phi_s \parallel \Sigma_s \parallel M_s\rangle, \langle \Phi_t \parallel \Sigma_t \parallel M_t\rangle) \\
& & \mid \forall \Phi_s', \Phi_t', R_s.\, (\Phi_s, \Phi_t) \sqsubseteq (\Phi_s', \Phi_t') \wedge \langle \Phi_s' \parallel \Sigma_s \parallel M_s\rangle \Downarrow R_s \\
& & \quad \Longrightarrow \exists R_t.\, \langle \Phi_t' \parallel \Sigma_t \parallel M_t\rangle \Downarrow R_t \wedge (R_s, R_t) \in \mathcal{R}[\![\tau]\!](\Phi_s, \Phi_t)\} \\
\\
\mathcal{R}[\![\tau]\!](\Phi_s, \Phi_t) & \stackrel{\text{def}}{=} & \{((\Phi_s', A_s), (\Phi_t', A_t)) \mid (\Phi_s, \Phi_t) \sqsubseteq (\Phi_s', \Phi_t') \wedge (A_s, A_t) \in \mathcal{A}[\![\tau]\!](\Phi_s', \Phi_t')\} \\
\\
\mathcal{A}[\![\texttt{int}]\!](\Phi_s, \Phi_t) & \stackrel{\text{def}}{=} & \{(n, n) \mid n \in \mathbb{Z}\} \\
\\
\mathcal{A}[\![\tau \to \sigma]\!](\Phi_s, \Phi_t) & \stackrel{\text{def}}{=} & \{((\Sigma, \lambda x.\, M), \mathsf{pack}\ (V, V')) \\
& & \mid \forall (W_s, W_t) \in \mathcal{V}[\![\tau]\!](\Phi_s, \Phi_t). \\
& & \quad (\langle \Phi_s, l \mapsto W_s \parallel \Sigma, x \mapsto l \parallel M\rangle \\
& & \quad , \langle \Phi_t, l \mapsto W_t \parallel \Sigma, x \mapsto l \parallel V'\ (V, x)\rangle) \in C[\![\sigma]\!]\} \\
\\
\mathcal{V}[\![\tau]\!](\Phi_s, \Phi_t) & \stackrel{\text{def}}{=} & \{((\Sigma, M), \mathsf{pack}\ (V, V')) \mid (\langle \Phi_s \parallel \Sigma \parallel M\rangle, \langle \Phi_t \parallel \varepsilon \parallel V\ V'\rangle) \in C[\![\tau]\!]\} \\
& & \cup \\
& & \{(V_s, V_t) \mid ((\Phi_s, V_s), (\Phi_t, V_t)) \in \mathcal{R}[\![\tau]\!](\Phi_s, \Phi_t)\} \\
\\
\mathcal{E}[\![\Gamma]\!](\Phi_s, \Phi_t) & \stackrel{\text{def}}{=} & \{(\Sigma_s, \Sigma_t) \mid \forall x{:}\tau \in \Gamma.\, (\Phi_s(\Sigma_s(x)), \Phi_t(\Sigma_t(x))) \in \mathcal{V}[\![\tau]\!](\Phi_s, \Phi_t)\}
\end{array}
$$

Fig. 11. Call-by-need closure-conversion logical relations

will be placed in the heap. The value relation $\mathcal{V}$ is a union of the value relations for call-by-name and call-by-value depending on if the value is a thunk closure or a normal form, respectively. Note that the call-by-need closure-conversion relations given in Sullivan *et al.* [79] do not include the memoization in their value relation. As their proof is left as a conjecture, it is unclear whether this is an issue.

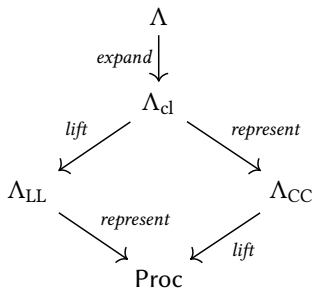
## 8 DISCUSSION

This document has explored the $\lambda$-calculus from its roots in Church's theory to the abstract machines and compiler transformations used to implement it efficiently. Therein, environments improved upon substitutions; machines with continuations as state improved upon using evaluation contexts; and intermediate languages could be used for optimizations and out of necessity. Additionally, I have explored these semantics with a special eye on the effect that different evaluation strategies play in implementation. The differences in strategies have resulted in a rift in the literature which may be filled with the following work in the future.

*Deconstructing Lambda-lifting and Closure-conversion.* Two transformations presented heretofore, *lambda-lifting* (Section 4.2.1) and *closure-conversion* (Section 6.3), appear to solve the same problem: remove, in some way, the need to consider free variables created from nested functions. In other words, they seek to remove lexical scope. Whereas the former prepares code to be run on the G-machine, the latter compiles a source program into a target language with a more flexible semantics. The development of the transformations has been mostly independent in the literature and especially tied to evaluation strategy making it unclear which should be used for a new compiler today. *If I am building a new compiler today, might I use one or the other? Does it matter the evaluation strategy of my source language?* Indeed, I do not need to choose one over the other since the two

are composed of the same sub-transformations. Moreover, *both* work with any of the evaluation strategies presented in this document.

Lambda-lifting and closure-conversion coincide if we break them into smaller sub-translations: expansion, lifting, and representation. The expansion sub-transformation is a strategy-dependent one that *expands* functions with free variables to functions with no free variables. The target language is denoted $\Lambda_{\text{cl}}$ and it contains only super-combinators, partial application values, and function applications. The lifting transformation is a strategy-independent transformation that *lifts* super-combinators to the top-level of a program. Its target language is a set of named super-combinators and a "`main`" expression to evaluate. Finally, representation is a transformation that represents closed functions as closures. The transformations are composed as the following diagram:




The reason a representation transformation for a lambda-lifted language is absent from the literature is because the languages are interpreted by a graph machine instead of compiled to a language like C. Unfortunately, it has been the closure-conversion camp that has been interested in correctness, so lambda-lifters like [36] opt for informal approaches to correctness.

*Reasoning about Memoization.* The compiler correctness community has done great work proving properties about their compilers including, most recently, even cost preservation [61]. However, the style of heap used in these language does not behave like that of call-by-need implementations. Additionally, even the reasoning found in the call-by-need calculi cannot easily be mapped to that of lazy language implementation. Therefore, future work is needed for reasoning about lazy heaps in order to produce a verified compiler for call-by-need. As a corollary, this will also verify the implementation of memoizing thunks found in call-by-value languages like Ocaml, as demonstrated in this document with closure-conversion implemented with thunks.

*Mixed Strategy Implementation.* I have been careful heretofore to discuss call-by-name, then call-by-value, then call-by-need. Call-by-name is never implemented in practical compilers because of its work duplication; rather, call-by-need is implemented in its place. Therefore, the literature regarding compilers often shows a separated development in call-by-value and call-by-need implementation. Practically, the STG machine [64] begins to bridge this gap in implementation; though, the CAM [18] and ZAM [45] were instances of strict implementations borrowing ideas from the non-strict. In reduction and equational theories, the work on polarized languages [27, 48, 53, 59, 84] aims to bridge this gap in reasoning by constructing languages that contain call-by-value and call-by-need components. It appears the future of sophisticated implementations will contain a story for both strategies.

Indeed, these languages have never been totally separate. A call-by-need compiler is not useful at all without call-by-value data types like strings. A call-by-value language is not even a functional language without the computation type of functions.

## REFERENCES


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

[2] Samson Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[3] Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. Orbit: An optimizing compiler for scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, 1986.

[4] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 8–19. ACM, 2003.

[5] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Inf. Process. Lett.*, 90(5):223–232, 2004.

[6] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theor. Comput. Sci.*, 342(1):149–172, 2005.

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

[8] 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.

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

[10] 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.

[11] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 233–246, 1995.

[12] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 233–246, 1995.

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

[14] H. P. Barendregt. The lambda calculus: Its syntax and semantics. 1984.

[15] G. L. Burn, Simon L. Peyton Jones, and J. D. Robson. The spineless g-machine. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 244–258, 1988.

[16] Luca Cardelli. The functional abstract machine. Technical report, Bell Labs, 1983.

[17] Alonzo Church. *The calculi of λ-conversion*, volume 6. Princeton University Press, 1941.

[18] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, pages 50–64, 1985.

[19] Pierre-Louis Curien. Categorical combinators. *Information and Control*, 69(1-3):188–254, 1986.

[20] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 233–243, 2000.

[21] Pierre-Louis Curien and Guillaume Munch-Maccagnoni. The duality of computation under focus. In Cristian S. Calude and Vladimiro Sassone, editors, *Theoretical Computer Science - 6th IFIP TC 1/WG 2.2 International Conference, TCS 2010, Held as Part of WCC 2010, Brisbane, Australia, September 20-23, 2010. Proceedings*, volume 323 of *IFIP Advances in Information and Communication Technology*, pages 165–181, 2010.

[22] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I. North-Holland Publishing Company, 1958.

[23] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392, 1972.

[24] Alberto de la Encina and Ricardo Pena. Proving the correctness of the STG machine. In Thomas Arts and Markus Mohnen, editors, *Implementation of Functional Languages, 13th International Workshop, IFL 2002 Stockholm, Sweden, September 24-26, 2001, Selected Papers*, volume 2312 of *Lecture Notes in Computer Science*, pages 88–104. Springer, 2001.

[25] Alberto de la Encina and Ricardo Pena. Formally deriving an STG machine. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 102–112. ACM, 2003.

[26] Alberto de la Encina and Ricardo Peña-Marí. From natural semantics to C: A formal derivation of two STG machines. *J. Funct. Program.*, 19(1):47–94, 2009.

[27] Paul Downen and Zena M. Ariola. Beyond polarity: Towards a multi-discipline intermediate language with sharing. In *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*, pages 21:1–21:23, 2018.

[28] Paul Downen and Zena M. Ariola. A tutorial on computational classical logic and the sequent calculus. *J. Funct. Program.*, 28, 2018.

[29] 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.

[30] Paul Downen, Luke Maurer, Zena M. Ariola, and Simon Peyton Jones. Sequent calculus as a compiler intermediate language. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 74–88, 2016.

[31] 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.

[32] Jon Fairbairn and Stuart Wray. TIM: A simple, lazy abstract machine to execute supercombinatorics. In *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, pages 34–45, 1987.

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

[34] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993.

[35] John Hatcliff and Olivier Danvy. Thunks and the lambda-calculus. *J. Funct. Program.*, 7(3):303–319, 1997.

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

[37] 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.

[38] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, pages 190–203, 1985.

[39] James H. Morris Jr. *Lambda Calculus Models of Programming Languages*. PhD thesis, Massachusets Institute of Technology, 1968.

[40] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[41] Andrew Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 177–190, 2007.

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

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

[44] 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.

[45] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[46] Xavier Leroy. Coinductive big-step operational semantics. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2006.

[47] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.

[48] Paul Blain Levy. *Call-by-push-value*. PhD thesis, Queen Mary University of London, UK, 2001.

[49] Rafael Dueire Lins. Categorical multi-combinators. In *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, pages 60–79, 1987.

[50] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, 1998.

[51] Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, pages 4–15, 2004.

[52] 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.

[53] Dylan McDermott and Alan Mycroft. Extended call-by-push-value: Reasoning about effectful programs and evaluation order. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, pages 235–262, 2019.

[54] 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.

[55] 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.

[56] 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.

[57] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, 1989.

[58] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 85–97, 1998.

[59] Guillaume Munch-Maccagnoni. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. PhD thesis, Université Paris Diderot, 2013.

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

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

[62] James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 128–148, 2014.

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

[64] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[65] 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.

[66] 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.

[67] Maciej Piróg and Dariusz Biernacki. A systematic derivation of the STG machine verified in coq. In Jeremy Gibbons, editor, *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, pages 25–36. ACM, 2010.

[68] 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.

[69] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[70] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:17–139, 2004.

[71] Amr Sabry. *The formal relationship between direct and continuation-passing style optimizing compilers - a synthesis of two paradigms*. PhD thesis, Rice University, 1994.

[72] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the Conference on Lisp and Functional Programming, LFP 1992, San Francisco, California, USA, 22-24 June 1992*, pages 288–298. ACM, 1992.

[73] Amr Sabry and Philip Wadler. A reflection on call-by-value. In Robert Harper and Richard L. Wexelblat, editors, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, pages 13–24. ACM, 1996.
[74] Dana Scott. Outline of a mathematical theory of computation. Technical report, Oxford University Computing Lab, 1970.
[75] Peter Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3):231–264, 1997.
[76] 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.
[77] Zhong Shao and Andrew W. Appel. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst.*, 22(1):129–161, 2000.
[78] Guy L. Steele. Rabbit: A compiler for scheme. Master's thesis, Massachusetts Institute of Technology, 1978.
[79] Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. Strictly capturing non-strict closures. In *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2021, Virtual Event, Denmark, January 18-19, 2021*, pages 74–89. ACM, 2021.
[80] W. W. Tait. Intensional interpretation of func-tionals of finite type i. *Journal of Symbolic Logic*, 32:198–212, 1967.
[81] D. A. Turner. A new implementation technique for applicative languages. *Softw., Pract. Exper.*, 9(1):31–49, 1979.
[82] Philip Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 189–201, 2003.
[83] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus.* PhD thesis, University of Oxford, 1971.
[84] Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching.* PhD thesis, Carnegie Mellon University, 2009.