Lazy Functions as Processes

Luke Maurer
University of Oregon
maurerl@cs.uoregon.edu

Abstract

CPS transforms have long been important tools in the study of programming languages, especially those related to the λ-calculus. Recently, it has been shown that encodings into process calculi, such as the π-calculus, can also serve as semantics, in the same way as CPS transforms. It is known that common encodings of the call-by-value and call-by-name λ-calculi into the π-calculus can be seen as CPS transforms composed with a naming transform that expresses sharing of values.

We review this analysis and extend it to call-by-need. The new call-by-need CPS transform requires extending the target λ-calculus with an effect, which we call *constructive update*. We present a proof of the correctness of the call-by-need CPS transform, which is hence a new proof of the correctness of the call-by-need π-calculus encoding. Also, we derive abstract machines from the CPS transforms discussed.

1. Introduction

Continuations and continuation-passing style (CPS) provide powerful and versatile tools for understanding programming languages [24]. By representing the "future of the program" as a first-class entity, a *CPS transform* gives a denotational semantics for a programming language in terms of a simple, well-understood low-level language. In a particularly influential use of continuations, Plotkin [23] demonstrated how a CPS transform can weave an implementation strategy for a program into the syntax of the program itself. This methodology gave rise to the call-by-value λ-calculus and was instrumental in closing the gap between the theory and practice of functional languages.

Since that time, CPS transforms have continued to further our understanding of programming languages. The call-by-value CPS transform was more descriptive than Plotkin's original call-by-value λ-calculus, motivating a more thorough study of strict functional languages; in turn, this lead to more advanced techniques for reasoning about programs in continuation-passing style and to a more complete development of the call-by-value λ-calculus [25, 26]. CPS transforms and related techniques have also provided a formal method for reasoning about effects, such as mutable references and non-local jumps, that lie outside of the pure model of the λ-calculus. Of particular note is delimited control, especially the **shift** and **reset** operators [11], which were originally developed by defining them in continuation-passing style.

As an alternative to CPS transforms, we can instead encode a language in a process calculus such as the π-calculus. A π-encoding represents computation as a form of interaction between independent processes, thus providing another view of run-time behavior. Since the π-calculus naturally expresses certain phenomena such as concurrency, nondeterminism, and distributed computation, a π-encoding is particularly suited to studying languages with these features. Processes which are deterministic and sequential, however, correspond to an environment-based presentation of a CPS language, as observed by Sangiorgi [27] and Amadio [3]. This correspondence has been studied largely in the context of the call-by-name and call-by-value λ-calculi. The aim of this paper is to extend the study to call-by-need evaluation [5, 7].

We start, in Section 2, with an introduction of the call-by-name and call-by-value lambda-calculi. We present their operational semantics and a uniform CPS transform, from which we derive CBN and CBV π-encodings, along with abstract machines. In Section 3, we consider call-by-need evaluation; we present a novel call-by-

$$\text{Evaluation Contexts:} \qquad E ::= [\,] \mid EM$$

$$(\lambda x.\, M)N \longrightarrow M\{N/x\} \qquad \beta_n$$

Figure 1. The call-by-name λ-calculus, λ_n

$$\begin{aligned} \text{Values:} \qquad & V ::= x \mid \lambda x.\, M \\ \text{Evaluation Contexts:} \qquad & E ::= [\,] \mid EM \mid VE \end{aligned}$$

$$(\lambda x.\, M)V \longrightarrow M\{V/x\} \qquad \beta_v$$

Figure 2. The call-by-value λ-calculus, λ_v

need CPS transform, which leads to an interesting concept in its own right: the notion of *constructive update*. From the new CPS transform, we derive the call-by-need π-encoding and an abstract machine in much the same way as we did for CBN and CBV.

2. The Call-by-Name and Call-by-Value λ-Calculi

The λ-calculus, defined by Church [10] in the 1930s, is a simple yet powerful model of computation. It consists of only three parts: *functions* from inputs to outputs, *variables* that stand for the inputs, and *applications* that invoke the functions:

$$\text{Terms:} \qquad M, N ::= \lambda x.\, M \mid x \mid MN$$

As the λ-calculus can be a foundation of both strict and lazy languages, a λ-term M can be evaluated according to different *evaluation strategies,* which dictate the operation to be performed first. The two most studied evaluation strategies are *call-by-name* (CBN) and *call-by-value* (CBV). In CBN evaluation, the argument to a function is kept unevaluated as long as possible, then evaluated each time its value is required for computation to continue. In CBV evaluation, the argument is always evaluated before the function receives it. It is usually better to precompute arguments, as CBV does, since then an argument will not be evaluated more than once; however, if the function does not actually use the value, computing it is wasteful. In the extreme case, if the argument *diverges* (that is, loops forever) but is never used by the function, CBV diverges when CBN does not.

The distinct CBN and CBV reduction strategies are captured by the two reduction rules, β_n and β_v (see Figs. 1 and 2). A β_n-reduction starts from a term $(\lambda x.\, M)N$, then proceeds by substituting the unevaluated argument N into the body wherever x appears. Thus the function call precedes the evaluation of the argument. In contrast, a β_v-reduction evaluates the argument first: Only a *value* may be substituted into the body. A value is either a variable or a function literal (also called a λ-*abstraction*).

A term that pattern-matches the left-hand side of a reduction rule is called a *redex*, short for *reducible expression*. For instance, $(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z))$ is a call-by-name redex. However, it is not a call-by-value redex, because the argument $(\lambda y.\, y)(\lambda z.\, z)$ is not a value. If a redex somewhere in M is reduced, and the resulting term is N, we write $M \longrightarrow N$ and say that M *reduces to* N; we also write $\longrightarrow\!\!\!\!\rightarrow$ for zero or more reductions and $\longrightarrow^+$ for one or more reductions.

To complete the semantics, one has to specify where a reduction should take place. Felleisen and Friedman [15] introduced a concise way to do so, using *evaluation contexts*. A context is a "term with a hole": It is the outer portion of some term, surrounding a single occurrence of the symbol $[\,]$. A language's evaluation

contexts delineate the places in a term where evaluation may take place. For instance, consider the grammar for evaluation contexts in Fig. 1: An evaluation context can be either just a hole (the *trivial* or *top-level* context, $[\,]$) or EM, a subcontext applied to an argument. Thus CBN evaluation can take place either at the top level or within the operator in a function application, but not within the argument (since arguments are left unevaluated). The CBV calculus also has contexts VE, so once the function in an application has become a value, CBV evaluation proceeds within the argument.

Given an evaluation context E, we write $E[M]$ for E with the term M in place of the hole, and we say M is *plugged into* E. (This notation applies to general contexts as well.) The inverse of the "plugging in" operation is *decomposition*, and it plays a critical role in evaluation by finding where to perform the next reduction. For example, we can decompose $(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z))$ in CBN as $E[M]$ where E is the top-level context $[\,]$ and M is the entire term. In CBV, we can decompose it with E being $(\lambda x.\, xx)[\,]$ and M being $(\lambda y.\, y)(\lambda z.\, z)$, since the next step of CBV evaluation is to reduce the argument.

If $M \equiv E[M']$ and M' is a redex reducing to N', then we write $M \longmapsto N$ where $N \triangleq E[N']$. In other words, $\longmapsto$ denotes reduction only within an evaluation context, which we call *standard reduction* or *evaluation*. Often we will say that M *steps* or *takes a step* in this case. As before, we write $\longmapsto\!\!\!\!\!\rightarrow$ for the reflexive and transitive closure and $\longmapsto^+$ for the transitive closure.

Finally, we introduce notations for the possible *observations* one can make about a term. These are the potential outcomes of computation, without regard to the particular steps taken. If M is a λ-abstraction, which we also call an *answer*, we write $M \downarrow$. If M *evaluates* to an answer (perhaps because it is one), we write $M \Downarrow$. A term M with no possible evaluation step, but which is *not* an answer, is called *stuck*, written $M \not\downarrow$; a term M that evaluates to a stuck term *gets stuck*, written $M \not\Downarrow$. If M *never* finishes evaluating—that is, it takes infinitely many evaluation steps—it is said to *diverge*, written $M \Uparrow$.

Example 1. Consider the term $(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z))$. CBN and CBV evaluate the term differently (the redex at each step is shaded):

$$
\begin{aligned}
&\underline{(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z))}\\
&\quad\longmapsto \underline{((\lambda y.\, y)(\lambda z.\, z))}((\lambda y.\, y)(\lambda z.\, z))\\
&\quad\longmapsto \underline{(\lambda z.\, z)((\lambda y.\, y)(\lambda z.\, z))}\\
&\quad\longmapsto \underline{(\lambda y.\, y)(\lambda z.\, z)}\\
&\quad\longmapsto (\lambda z.\, z)
\end{aligned}
\qquad\qquad
\begin{aligned}
&\underline{(\lambda x.\, xx)}((\lambda y.\, y)(\lambda z.\, z))\\
&\quad\longmapsto \underline{(\lambda x.\, xx)(\lambda z.\, z)}\\
&\quad\longmapsto \underline{(\lambda z.\, z)(\lambda z.\, z)}\\
&\quad\longmapsto (\lambda z.\, z)
\end{aligned}
$$

CBN reduces the outer β-redex immediately, substituting the argument as is. This duplicates work, since the β_n-redex $(\lambda y.\, y)(\lambda z.\, z)$ now appears for each x in the body of $\lambda x.\, xx$. Instead, CBV evaluates the argument first, reducing it to a value before substituting, thus saving one reduction.

2.1 A Uniform Continuation-Passing-Style Transform

As an alternative to specifying the semantics of a language in terms of rewrite rules for programs, one can specify a function that "compiles," or transforms, programs into some lower-level form. The advantage is that analyzing a lower-level form is easier, since the syntax itself prescribes how a program should be executed, just as assembly code specifies not only calculations but which registers, including the program counter, to use to perform them. A transform into *continuation-passing style*, called a *CPS transform*, is an example of such a compilation function. It produces λ-terms whose evaluation order is predetermined: The same calculations will be performed, in the same order, by call-by-name or call-by-value evaluation. The trick is to pass only precomputed values as arguments to functions, making the question of when to evaluate arguments moot. Then, rather than returning its result in the usual way, a CPS function passes the result to one of its arguments, the so-called *continuation*. A continuation represents the evaluation context in which a function was invoked; hence it plays a similar role to the *call stack* used in most computer architectures. Since their evaluation contexts differ,

$$
\begin{aligned}
\mathcal{C}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, k(\lambda(x, k').\, \mathcal{C}[\![M]\!]k') \\
\mathcal{C}[\![MN]\!] &\triangleq \begin{cases} \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, v(\lambda k'.\, \mathcal{C}[\![N]\!]k', k)) & \text{CBN} \\ \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, \mathcal{C}[\![N]\!](\lambda w.\, v(\lambda k'.\, k'w, k))) & \text{CBV} \end{cases}
\end{aligned}
$$

Figure 3. A uniform CPS transform for call-by-name and call-by-value

we can elucidate the difference between CBN and CBV evaluation by translating each to continuation-passing style.

We focus on a *uniform* CPS transform $\mathcal{C}$, given in Fig. 3, so called because the translations for variables and abstractions are the same between CBN and CBV. This uniformity highlights the differences in evaluation order by varying only the translation of applications. Specifically, once M has evaluated to a function v, the continuation in the CBN transform invokes v immediately, passing it the unevaluated CPS term $\lambda k'.\, \mathcal{C}[\![N]\!]k'$ as x. Evaluating a variable is done by invoking it with a continuation, so each invocation of x within the body of v will evaluate the argument. The CBV transform evaluates M the same way, but its continuation does not use v immediately; instead, it evaluates N to a function w, and only *its* continuation invokes v. This time, the x argument is a function that immediately passes w to the continuation; therefore each invocation of x within the body of v immediately returns the precomputed argument value w.

Example 2. Consider the term $(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z))$ from above, calling it M for now. In CBN (the redex is always the whole CPS term, so we omit the shading):

$$
\begin{aligned}
\mathcal{C}[\![M]\!]k &\triangleq \mathcal{C}[\![\lambda x.\, xx]\!](\lambda v.\, v(\lambda k'.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!]k', k)) \\
&\triangleq (\lambda k.\, k(\lambda(x, k').\, \mathcal{C}[\![xx]\!]k'))(\lambda v.\, v(\lambda k'.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!]k', k)) \\
&\longmapsto (\lambda v.\, v(\lambda k'.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!]k', k)(\lambda(x, k').\, \mathcal{C}[\![xx]\!]k')
\end{aligned}
$$

The function has evaluated to v; we invoke it immediately:

$$
\begin{aligned}
&\longmapsto (\lambda(x, k').\, \mathcal{C}[\![xx]\!]k')(\lambda k'.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!]k', k) \\
&\triangleq (\lambda(x, k').\, (\lambda k.\, \mathcal{C}[\![x]\!](\lambda v.\, v(\lambda k'.\, \mathcal{C}[\![x]\!]k', k)))k')(\lambda k'.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!]k', k) \\
&\triangleq (\lambda(x, k').\, (\lambda k.\, (\lambda k.\, xk)(\lambda v.\, v(\lambda k'.\, (\lambda k.\, xk)k', k)))k')(\lambda k'.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!]k', k) \\
&\longmapsto (\lambda k.\, (\lambda k.\, (\lambda k'.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!]k')k))(\lambda v.\, \\
&\qquad v(\lambda k.\, (\lambda k'.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!]k', k)))k))k
\end{aligned}
$$

This last reduction step duplicates work, as the evaluation of $\mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!]$ must now occur twice.

Now for CBV:

$$
\begin{aligned}
\mathcal{C}[\![M]\!]k &\triangleq \mathcal{C}[\![\lambda x.\, xx]\!](\lambda v.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!](\lambda w.\, v(\lambda k.\, kw, k))) \\
&\triangleq (\lambda k.\, k(\lambda(x, k').\, \mathcal{C}[\![xx]\!]k'))(\lambda v.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!](\lambda w.\, v(\lambda k.\, kw, k))) \\
&\longmapsto (\lambda v.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!](\lambda w.\, v(\lambda k.\, kw, k)))(\lambda(x, k').\, \mathcal{C}[\![xx]\!]k')
\end{aligned}
$$

The function has evaluated to v, but this time we evaluate the argument next:

$$
\longmapsto \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!](\lambda w.\, (\lambda(x, k').\, \mathcal{C}[\![xx]\!]k')(\lambda k.\, kw, k))
$$

Once the argument is computed as w, *then* the function will be invoked, but this time with x being a function that immediately passes along the precomputed value w.

$$\begin{array}{rl} \text{Values:} & V ::= \lambda x.\, M \\ \text{Evaluation Contexts:} & E ::= [\,] \mid EM \mid VE \end{array}$$

$$(\lambda x.\, M)V \longrightarrow M\{V/x\} \qquad \beta_v$$

Figure 4. A revised call-by-value λ-calculus

The uniform CPS transform reflects the behavior of common language implementations: Evaluation always stops at a λ, and a variable always causes a lookup (hence a free variable halts execution). However, these behaviors don't faithfully represent the full theory of the λ-calculus. For instance, the calculus is often considered with the η rule in addition to β. An η-reduction takes $\lambda x.\, Mx$ to just M whenever x does not appear in M. For a free variable y, then, the term $\lambda x.\, yx$ would become stuck after one reduction, whereas the uniform CPS transform gives a term that immediately returns the value $\lambda x.\, yx$.

The η rule is often considered unimportant for language implementations: Nearly all compilers and interpreters for functional languages stop evaluating when they find a λ. Many well-studied CBN CPS transforms, such as that of Plotkin [23], do not validate η; in fact, the CBN fragment of the uniform CPS transform is very similar to Plotkin's transform. The CBV fragment, however, differs more fundamentally: It doesn't follow the conventional β_v rule, either. In the CBV λ-calculus, a variable is considered a value, yet few compilers or interpreters operate this way: The term $(\lambda x.\, \lambda y.\, y)z$ should reduce to $\lambda y.\, y$, but a typical implementation would raise an error on seeing the free variable z. Accordingly, the CBV portion of the uniform CPS transform produces a term that becomes stuck on z rather than reducing to a value.

Therefore the CBV language truly implemented by the uniform CPS transform is not the one given in Fig. 2. Rather, it implements a calculus that further restricts the β_v rule to apply only to a λ-abstraction as an argument. Equivalently, this revised calculus consideres only a λ-abstraction to be a value. From now on, then, when we speak of the call-by-value calculus, we will refer to the version in Fig. 4. In particular, $\longmapsto$ will refer to the restricted notion of evaluation context.

In most cases, our departure from orthodoxy will make no difference. In standard reductions of closed terms, it never happens that a free variable appears as an argument, and thus it does not matter whether we consider it a value or not.

2.2 The CPS Language λ_{cps}

The terms produced by the uniform CPS transform comprise a restricted λ-calculus. The grammar is given in Fig. 5. In an application, the function must be a value, and it can take one or two arguments, which must also be values; we denote this $V(W^+)$ (we will omit parentheses when there is one argument). A value is a variable, a λ-abstraction, or the constant $\mathbf{ret}$. Note that the body of an abstraction must again be a CPS term—that is, an application. In CPS, a function never returns to its caller; it only performs more function calls. Accordingly, the only evaluation context is the trivial context $[\,]$, as the redex is always at the top level.

There are three kinds of value that appear in a CPS term:

Thunks A *thunk* is a suspended computation. In CPS terms, this is a function $\lambda k.\, M$ that takes a continuation, calculates a result, then passes the result to the continuation. Each term of the form $\mathcal{C}[\![M]\!]$ is a thunk. In the uniform CPS transform, variables are represented by thunks.

Continuations A *continuation* is a handler for a result; it has the form $\lambda v.\, M$. It takes a computed source value and performs the next step of evaluation. We can see it as a reification of a term's evaluation context from the source language.

$$
\begin{array}{rl}
\text{Terms:} & M, N ::= V(W^+) \\
\text{Values:} & V, W ::= x \mid \mathbf{ret} \mid \lambda(x^+).\, M \\
\text{Evaluation Contexts:} & E ::= [\,]
\end{array}
$$

$$
(\lambda(x^+).\, M)(V^+) \longrightarrow M\{V^+/x^+\} \qquad \beta
$$

Figure 5. The syntax and semantics of the CPS λ-calculus, λ_{cps}

Source Values Each value from the source calculus has a CPS encoding. As we are translating from calculi having only functions as values, we need only consider how to encode a function. Namely, a source function becomes a binary function $\lambda(x, k).\, M$ that takes a thunk x for computing the argument and a continuation k to invoke with the result.

Before we consider observations, we should consider what it means for a CPS program to be evaluated. A term $\mathcal{C}[\![M]\!]$ is an inert λ-abstraction; it must be given a continuation as its argument for evaluation to occur. This argument represents the context in which to evaluate M. If we consider M to be the whole program, we need an *initial continuation* to represent the top-level context. Thus we introduce the constant $\mathbf{ret}$; to evaluate M as a CPS program, then, one writes $\mathcal{C}[\![M]\!]\,\mathbf{ret}$. If one thinks of a term $\mathcal{C}[\![M]\!]$ as meaning "evaluate M and then," $\mathcal{C}[\![M]\!]\,\mathbf{ret}$ then reads "evaluate M and then return." Thus $\mathbf{ret}$ is analogous to the C function `exit`, which terminates execution and yields its argument as the result of the program.

Since an answer is the result of successful computation, then, a CPS answer is a term of the form $\mathbf{ret}\,V$, for a λ-abstraction V.[1] Thus $M \downarrow$ means that M has the form $\mathbf{ret}(\lambda(x^+).\, N)$, and $M \Downarrow$ means that M evaluates to such a term. As before, $M \not\downarrow$ means that M is stuck (hence not an answer); $M \not\Downarrow$ means that M gets stuck; and $M \Uparrow$ means that M diverges.

2.3 Environment-Based CPS Transform

So far, we have expressed argument passing by *substitution*: Each β-reduction substitutes the arguments for the free occurences of the corresponding variables. Effectively, a copy is made of each argument for each occurrence. Interpreters typically operate differently: Each argument is put into an *environment*, indexed by the variable it is bound to. Then, when a variable appears as a function being invoked, its value is retrieved from the environment.

We can simulate this mechanism by giving a *name* to each abstraction in argument position, substituting only names during β-reduction, and copying the value only as necessary. This is analogous to graph rewriting and can be captured by extending the syntax with a $\mathbf{let}$ construct [6]: A bound name identifies a node in a graph. However, we prefer an alternative syntax which expresses the dynamic allocation of names. We write $\nu x.\, x := \lambda(x^+).\, M\ \mathbf{in}\ N$ to indicate that a new name x is generated and a λ-abstraction is bound to it. Note that we will always bind an abstraction to a name immediately after allocation and only then. The value-named CPS λ-calculus, $\lambda_{cps,vn}$, is given in Fig. 6. Each term is now an application inside some number of bindings, which effectively serve as the environment. Each argument to an application must be a variable.

Note that we now have nontrivial evaluation contexts, unlike with λ_{cps}, whose only evaluation context was $[\,]$. However, the contexts in $\lambda_{cps,vn}$ do not specify work to be done but simply bindings for variables. To emphasize this, we call a context providing only bindings a *binding context*, and say that a CPS calculus has only binding contexts as evaluation contexts.

[1] In principle, we could avoid adding a constant by simply using some free variable k for the initial continuation. We would then have to have a predicate $\Downarrow_k$ for each name k, and correctness theorems would be quantified over k. Thus $\mathbf{ret}$ is merely a convenience.

$$
\begin{array}{rl}
\text{Terms:} & M, N ::= V(x^+) \mid \nu x.\, x \coloneqq \lambda(x^+).\, M \,\mathbf{in}\, N \\
\text{Values:} & V ::= x \mid \mathbf{ret} \mid \lambda(x^+).\, M \\
\text{Binding Contexts:} & B ::= [\,] \mid \nu x.\, x \coloneqq \lambda(x^+).\, M \,\mathbf{in}\, B \\
\text{Eval. Contexts:} & E ::= B
\end{array}
$$

$$
\begin{array}{rcll}
(\lambda(x^+).\, M)(y^+) & \longrightarrow & M\{y^+/x^+\} & \beta \\
\nu f.\, f \coloneqq \lambda(x^+).\, M & & \nu f.\, f \coloneqq \lambda(x^+).\, M & \\
\quad \mathbf{in}\, E[f(y^+)] & \longrightarrow & \quad \mathbf{in}\, E[(\lambda(x^+).\, M)(y^+)] & \mathit{deref}
\end{array}
$$

Figure 6. The value-named CPS λ-calculus, $\lambda_{cps,vn}$

$$
\begin{aligned}
\mathcal{C}_{vn}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}_{vn}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, \nu f.\, f \coloneqq \lambda(x, k').\, \mathcal{C}_{vn}[\![M]\!]k' \,\mathbf{in}\, kf \\
\mathcal{C}_{vn}[\![MN]\!] &\triangleq \begin{cases}
\lambda k.\, \nu k'.\, k' \coloneqq \big(\lambda v. & \\
\quad \nu x.\, x \coloneqq \lambda k''.\, \mathcal{C}_{vn}[\![N]\!]k'' & \text{CBN} \\
\quad \mathbf{in}\, v(x, k)\big) \,\mathbf{in}\, \mathcal{C}_{vn}[\![M]\!]k' & \\
\lambda k.\, \nu k'.\, k' \coloneqq \big(\lambda v.\, \nu k''.\, k'' \coloneqq (\lambda w. & \\
\quad \nu x.\, x \coloneqq \lambda k'.\, k'w \,\mathbf{in}\, v(x, k)) & \text{CBV} \\
\quad \mathbf{in}\, \mathcal{C}_{vn}[\![N]\!]k''\big) \,\mathbf{in}\, \mathcal{C}_{vn}[\![M]\!]k' &
\end{cases}
\end{aligned}
$$

Figure 7. Uniform CPS transform in named form

To convert an unnamed term to a named term, we introduce a *naming transform*, $\mathcal{N}$. The naming transform goes through all arguments appearing in a term, moving each λ-abstraction into a new variable.

$$
\begin{aligned}
\mathcal{N}[\![V(\lambda(x^+).\, M)]\!] &\triangleq \nu y.\, y \coloneqq \lambda(x^+).\mathcal{N}[\![M]\!] \,\mathbf{in}\, \mathcal{N}[\![V(y)]\!] \\
\mathcal{N}[\![V(\lambda(x^+).\, M, W)]\!] &\triangleq \nu y.\, y \coloneqq \lambda(x^+).\mathcal{N}[\![M]\!] \,\mathbf{in}\, \mathcal{N}[\![V(y, W)]\!] \\
\mathcal{N}[\![V(y, \lambda(x^+).\, M)]\!] &\triangleq \nu z.\, z \coloneqq \lambda(x^+).\mathcal{N}[\![M]\!] \,\mathbf{in}\, \mathcal{N}[\![V(y, z)]\!] \\
\mathcal{N}[\![(\lambda(x^+).\, M)(y^+)]\!] &\triangleq (\lambda(x^+).\mathcal{N}[\![M]\!])(y^+) \\
\mathcal{N}[\![f(y^+)]\!] &\triangleq f(y^+)
\end{aligned}
$$

For clarity, here we assume that each function has at most two arguments, as is true for our CPS terms; $\mathcal{N}$ generalizes straightforwardly by iteration.

The uniform CPS transform under the naming transform is given in Fig. 7.

Proposition 3. $\mathcal{C}_{vn}[\![M]\!] \equiv \mathcal{N}[\![\mathcal{C}[\![M]\!]]\!]$.

Proof. Straightforward induction on M. □

2.4 Preservation of Observations

We show correctness of the value-named uniform CPS transform in two steps. We start with the correctness of the unnamed transform, then prove the correctness of the naming step.

2.4.1 Proof Methodology

For a CPS transform to be considered correct, we would want it to preserve termination [19]:

Criterion 4. $M \Downarrow$ *iff* $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Downarrow$.

In order to prove this criterion, generally one proceeds by induction on the evaluation steps. In order for the induction to go through, we need to establish an *invariant*: Something that is true at the beginning of evaluation and remains true after each step. For $\mathcal{C}$, the simplest invariant one can imagine would be this:

$$\begin{array}{ccc} M & \longmapsto & N \\ \downarrow & & \vdots \\ \mathcal{C}[\![M]\!]K & \mapsto & \mathcal{C}[\![N]\!]K \end{array} \tag{1}$$

In words, for any continuation K, whenever M reduces to N, $\mathcal{C}[\![M]\!]K$ reduces to $\mathcal{C}[\![N]\!]K$.[2] However, this invariant does not hold. One reason is that the CPS transform introduces many *administrative redexes* into the term. These are intermediate computations that do not correspond to actual β-reductions in the source language. (Non-administrative redexes are called *proper*.) Hence one step for M may correspond to many in $\mathcal{C}[\![M]\!]K$. Thus consider:

$$\begin{array}{ccc} M & \longmapsto & N \\ \downarrow & & \vdots \\ \mathcal{C}[\![M]\!]K & \mapsto\!\!\!\!\rightarrow & \mathcal{C}[\![N]\!]K \end{array} \tag{2}$$

Unfortunately, there is a more serious issue with (2). As noted by Plotkin [23], administrative reductions do not line up with the CPS transform in the way that the diagram suggests. Because $\mathcal{C}[\![N]\!]K$ introduces administrative redexes of its own, we cannot say that $\mathcal{C}[\![M]\!]K \longmapsto\!\!\!\!\rightarrow \mathcal{C}[\![N]\!]K$; we can only say that $\mathcal{C}[\![M]\!]K \longmapsto\!\!\!\!\rightarrow P$ for some P that $\mathcal{C}[\![N]\!]K$ can reduce to, perhaps through non-standard reductions. In fact, since $\mathcal{C}[\![N]\!]$ is an abstraction, $\mathcal{C}[\![N]\!]K$ is itself an administrative redex, so there is always at least one such step. The true situation is this:

$$\begin{array}{ccccc} M & \longmapsto & & & N \\ \downarrow & & & & \vdots \\ \mathcal{C}[\![M]\!]K & \mapsto\!\!\!\!\rightarrow & P & \leftarrow\!\!\!\!\!\!\cdots & \mathcal{C}[\![N]\!]K \end{array} \tag{3}$$

Plotkin's solution was to derive a new transform that eliminated these initial administrative redexes, thus regaining (2). The problem with this and similar solutions [12, 13] is that the resulting transforms are more complex and difficult to reason about than the original CPS transform. For instance, usually such administration-free CPS transforms are non-compositional [13].

Instead of changing the transform, we can further loosen the invariant using the *bisimulation* technique. Bisimulation is an alternative approach to soundness and completeness that requires only that we find *some* suitable relation to act as the invariant. Given a relation $\sim$, we can use it to prove Criterion 4 so long as the

[2] Ultimately, of course, we observe what happens when K is $\mathbf{ret}$. But there is nothing special about $\mathbf{ret}$; we expect our diagrams to hold for any K.

following hold:[3]

$$\begin{array}{ccccc} M & & M \longmapsto N & & M \vdash\!\!\dashrightarrow\!\!\twoheadrightarrow N \\ \sim & & \sim & & \sim \\ \mathcal{C}[\![M]\!]\,\mathbf{ret} & & P \vdash\!\!\dashrightarrow\!\!\twoheadrightarrow Q & & P \longmapsto Q \\ M\downarrow & & & & M \vdash\!\!\dashrightarrow\!\!\twoheadrightarrow N\downarrow \\ \sim & & & & \sim \\ P \vdash\!\!\dashrightarrow\!\!\twoheadrightarrow Q\downarrow & & Q\downarrow & & \end{array} \tag{4}$$

So, M is related to its image under the transform; when either related term takes a step, the other can take some number of steps to remain in the relation; and if either is an answer, the other evaluates to an answer.

We have that evaluation to an answer is preserved; however, what can we say about a stuck term? It could happen that M is stuck but $\mathcal{C}[\![M]\!]\,\mathbf{ret}$ loops forever, or vice versa. Thus we consider an additional criterion:

Criterion 5. *$M \not\Downarrow$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} \not\Downarrow$.*

To prove Criterion 5, we require two more properties of the simulation $\sim$, in addition to those in (4):

$$\begin{array}{ccc} M\not\downarrow & & M \vdash\!\!\dashrightarrow\!\!\twoheadrightarrow N\not\downarrow \\ \sim & & \sim \\ P \vdash\!\!\dashrightarrow\!\!\twoheadrightarrow Q\not\downarrow & & P\not\downarrow \end{array} \tag{5}$$

In words, if either M or P is stuck, then the other must get stuck.

The final observation that we want to preserve is divergence:

Criterion 6. *$M \Uparrow$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Uparrow$.*

However, because evaluation is deterministic in our calculi, we can get Criterion 6 "for free" from Criteria 4 and 5: If one term diverges, it can neither reduce to an answer nor get stuck, and hence the other term can only diverge.

We can further simplify the proof methodology thanks to Leroy's observation [18] that if the source language is deterministic, the forward simulation may be sufficient. However, we need to strengthen the invariant by requiring that each source evaluation step maps to at least one step in the CPS term; if we allow the CPS term to "spin in place" indefinitely, then a diverging source term could translate as an answer. In short, it will suffice to show:

$$\begin{array}{ccc} M & & M \longmapsto N \\ \sim & & \sim \\ \mathcal{C}[\![M]\!]\,\mathbf{ret} & & P \vdash\!\!\dashrightarrow^{+} Q \\ M\downarrow & & M\not\downarrow \\ \sim & & \sim \\ P \vdash\!\!\dashrightarrow\!\!\twoheadrightarrow Q\downarrow & & P \vdash\!\!\dashrightarrow\!\!\twoheadrightarrow Q\not\downarrow \end{array} \tag{6}$$

From these properties and determinacy, we can prove both directions of Criteria 4 to 6. The forward direction follows directly by induction. For the backward direction of Criterion 4, we can argue by contraposition: If M *does not* reduce to an answer, then it must either diverge or get stuck. If it diverges, then by the second diagram

[3] Technically, it is the second and third diagrams that characterize a bisimulation. The others are additional properties that we need in order to finish the proof. The fourth and fifth are very similar to the requirements on a *barbed bisimulation*[21].

in (6), it must hold that $\mathcal{C}[\![M]\!]\ \mathbf{ret}$ diverges, and hence by determinacy it cannot reduce to an answer. Similarly, if M gets stuck, $\mathcal{C}[\![M]\!]\ \mathbf{ret}$ must get stuck, and hence cannot reduce to an answer. The reasoning for the backward direction of Criteria 5 and 6 is similar.

2.4.2 Correctness of the CPS Transform

We define the simulation $\sim$ by comparing terms in a way that ignores all administrative reductions. We consider a λ-abstraction administrative when it always forms an administrative redex. We mark these administrative λ-abstractions by placing a line over the λ, as in $\bar{\lambda}k.\, M$. The explicitly marked uniform CPS transform is then:

$$\begin{aligned}
\mathcal{C}[\![x]\!] &\triangleq \bar{\lambda}k.\, xk \\
\mathcal{C}[\![\lambda x.\, M]\!] &\triangleq \bar{\lambda}k.\, k(\lambda(x, k').\, \mathcal{C}[\![M]\!]k') \\
\mathcal{C}[\![MN]\!] &\triangleq \begin{cases} \bar{\lambda}k.\, \mathcal{C}[\![M]\!](\bar{\lambda}v.\, v(\bar{\lambda}k'.\, \mathcal{C}[\![N]\!]k', k)) & \text{CBN} \\ \bar{\lambda}k.\, \mathcal{C}[\![M]\!](\bar{\lambda}v.\, \mathcal{C}[\![N]\!](\bar{\lambda}w.\, v(\bar{\lambda}k'.\, k'w, k))) & \text{CBV} \end{cases}
\end{aligned}$$

Notice that the only proper λ-abstractions are the ones that correspond to a λ-abstraction from the original term, since these are the abstractions whose reductions correspond to the actual β-reductions in the source language. To distinguish administrative computation, we introduce the reduction relation $\longrightarrow_{ad}$, defined by the *administrative β-rule*:

$$(\bar{\lambda}(x^+).\, M)(V^+) \longrightarrow_{ad} M\{V^+/x^+\} \qquad \beta_{ad}$$

Keeping to our notational conventions, the reflexive and transitive closure of $\longrightarrow_{ad}$ is $\twoheadrightarrow_{ad}$ and its transitive closure is $\longrightarrow^+_{ad}$. Also, its reflexive, *symmetric*, and transitive closure is $=_{ad}$. Furthermore, $\longmapsto_{ad}$ stands for a standard administrative reduction, which is to say an administrative reduction in the empty context (at top level), and $\longmapsto\!\!\!\!\twoheadrightarrow_{ad}$ are $\longmapsto^+_{ad}$ are the usual closures. We will also use the subscript *pr* in place of *ad* to denote a proper reduction. Finally, $\longmapsto^+_{pr^1}$ is short for $\longmapsto\!\!\!\!\twoheadrightarrow_{ad} \longmapsto_{pr} \longmapsto\!\!\!\!\twoheadrightarrow_{ad}$, which is to say, some number of standard reductions, exactly one of which is proper.

If a term cannot take an administrative standard reduction, then for the moment, the administrative work in that term is finished. Hence, if we consider the administrative subcalculus of λ_{cps}, such a term is the result, or *answer*, of administrative computation. Therefore let a term with no administrative standard reduction be called an *administrative answer*. In the following, we rely on some known properties of the λ-calculus, which also apply to the administrative subset of the CPS λ-calculus.

Proposition 7. *Administrative reduction in λ_{cps}:*

1. *is* confluent*, so that if $M =_{ad} M'$, then there is some N such that $M \twoheadrightarrow_{ad} N$ and $M' \twoheadrightarrow_{ad} N$; and*
2. *has the* standardization *property, so that if $M \twoheadrightarrow_{ad} N$ and N is an administrative answer, then there is an administrative answer M' such that $M \longmapsto\!\!\!\!\twoheadrightarrow_{ad} M' \twoheadrightarrow_{ad} N$.*

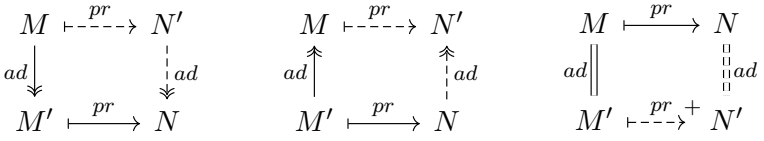
Next we need to know how non-standard, or *internal*, administrative reductions interact with proper standard reductions. In short, they don't—administrative reductions commute with proper standard reductions (see Fig. 8).

Proposition 8.

1. *If $M \twoheadrightarrow_{ad} M' \longmapsto_{pr} N$ and M is an administrative answer, then there is N' with $M \longmapsto_{pr} N' \twoheadrightarrow_{ad} N$.*
2. *If $M \twoheadleftarrow_{ad} M' \longmapsto_{pr} N$, then there is N' with $M \longmapsto_{pr} N' \twoheadleftarrow_{ad} N$.*

Proof.

1. If M is an administrative answer, then it is a proper β-redex; let $M \triangleq (\lambda(x^+).\, P)(V^+)$. Any administrative reductions in M must take place either in P or in V^+; in general, they could take P to some P' and V^+ to some V'^+. Hence $M' \equiv (\lambda(x^+).\, P')(V'^+)$. Since $M' \longmapsto_{pr} N$, this means $N \equiv P'\{V'^+/x^+\}$, and we take $N' \triangleq P\{V^+/x^+\}$.



(a) Proposition 8.1 **(b)** Proposition 8.2 **(c)** Lemma 9

Figure 8. Diagrams of Proposition 8 and Lemma 9.

2. Similar. □

As a consequence, we have that $=_{ad}$, which can involve arbitrary administrative reductions in either direction, commutes with proper standard reduction.

Lemma 9. *If $M =_{ad} M' \longmapsto_{pr} N$, then there is N' such that $M \longmapsto^{+}_{pr^1} N' =_{ad} N$.*

This is a crucial lemma; we will prove it later.

Now that we know what administrative reductions *don't* do, we should see what they *can* do: They serve to bring the standard redex in the source term to the top of the CPS term. This entails reifying the evaluation context as a continuation, so that we begin with $\mathcal{C}[\![E[M]]\!]\ \mathbf{ret}$ and build toward $\mathcal{C}[\![M]\!]K$, where K is a continuation that "represents" E somehow. We can formalize this intuition:

Proposition 10. *For each evaluation context E in λ_n or λ_v and each continuation K, there is a continuation K' such that for every term M we have $\mathcal{C}[\![E[M]]\!]K \longmapsto\!\!\!\!\rightarrow_{ad} \mathcal{C}[\![M]\!]K'$.*

Proof. By induction on the structure of E in each calculus. For call-by-name, we have two cases:

- If $E \equiv [\,]$, take $K' \triangleq K$.
- For $E \equiv E'N$, we have:

$$\begin{aligned}\mathcal{C}[\![E'[M]N]\!]K &\longmapsto_{ad} \mathcal{C}[\![E'[M]]\!](\bar{\lambda}v.\, v(\mathcal{C}[\![N]\!], K)) \\ &\longmapsto\!\!\!\!\rightarrow_{ad} \mathcal{C}[\![M]\!]K' \qquad \text{(by I.H.)}\end{aligned}$$

For call-by-value, we have three cases:

- If $E \equiv [\,]$, take $K' \triangleq K$.
- For $E \equiv E'N$, we have:

$$\begin{aligned}\mathcal{C}[\![E'[M]N]\!]K &\longmapsto_{ad} \mathcal{C}[\![E'[M]]\!](\bar{\lambda}v.\, \mathcal{C}[\![N]\!](\bar{\lambda}w.\, v((\bar{\lambda}k'.\, k'w), K))) \\ &\longmapsto\!\!\!\!\rightarrow_{ad} \mathcal{C}[\![M]\!]K' \qquad \text{(by I.H.)}\end{aligned}$$

- Finally, suppose $E \equiv VE'$. In our modified CBV calculus, V must be a λ-abstraction and not a variable, so we have:

$$\begin{aligned}&\mathcal{C}[\![(\lambda x.\, N)(E'[M])]\!]K \\ &\quad\longmapsto_{ad} (\bar{\lambda}k.\, k(\lambda(x, k').\, \mathcal{C}[\![N]\!]k'))(\bar{\lambda}v.\, \mathcal{C}[\![E'[M]]\!](\bar{\lambda}w.\, v((\bar{\lambda}k.\, {}''k''w), K))) \\ &\quad\longmapsto\!\!\!\!\rightarrow_{ad} \mathcal{C}[\![E'[M]]\!](\bar{\lambda}w.\, (\bar{\lambda}k.\, k(\lambda(x, k').\, \mathcal{C}[\![N]\!]k'))((\bar{\lambda}k''.\, k''w), K))\end{aligned}$$

Note that if V could be a variable, then the CPS transformation would get stuck after the first step, and we would not be able to bring $E'[M]$ to the top of the transformed term. □

We now want to show that observations in the λ_n and λ_v calculi line up with observations of the CPS-transformed terms. In other words, we prove that $\mathcal{C}$ meets Criteria 4 to 6.

We define our forward simulation $\sim$ as follows:

Definition 11. *For a λ-term M (either CBN or CBV) and CPS term P, let $M \sim P$ when $\mathcal{C}[\![M]\!]\,\mathbf{ret} =_{ad} P$.*

Our task is to prove that $\sim$ satisfies the diagrams in (6), making it a forward simulation. To begin, we first prove that "answerness" and "stuckness" are preserved by administrative operations:

Proposition 12. *If $P =_{ad} P'$ and $P \downarrow$, then $P' \Downarrow$.*

Proof. By confluence, there must be a term Q such that $P \longrightarrow\!\!\!\!\!\rightarrow_{ad} Q$ and $P' \longrightarrow\!\!\!\!\!\rightarrow_{ad} Q$. Since P is an answer it must have the form $\mathbf{ret}\,V$, therefore the reductions in $\longrightarrow\!\!\!\!\!\rightarrow_{ad}$ must have been within V, so Q must have the form $\mathbf{ret}\,V'$. Finally, by standardization, since $P' \longrightarrow\!\!\!\!\!\rightarrow_{ad} \mathbf{ret}\,V'$, there must be R with $P' \longmapsto\!\!\!\!\!\rightarrow_{ad} R \longrightarrow\!\!\!\!\!\rightarrow_{ad} \mathbf{ret}\,V'$; since non-standard reductions cannot disturb the top redex, we must have $R \equiv \mathbf{ret}\,V''$, so $P' \Downarrow$. □

Proposition 13. *If $P =_{ad} P'$ and $P \not\downarrow$, then $P' \not\Downarrow$.*

Proof. Similar to Proposition 12, again invoking confluence and standardization. □

We are now ready to prove the third and forth commuting diagrams of Eq. (6), showing that $\sim$ relates answers to answers and stuck terms to stuck terms (up to some remaining steps in the CPS term):

Lemma 14. *If $M \sim P$ and $M \downarrow$, then $P \Downarrow$.*

Proof. Let $M \triangleq \lambda x.\,N$ for some N. Since $(\lambda x.\,N) \sim P$, we know that $P =_{ad} \mathcal{C}[\![\lambda x.\,N]\!]\,\mathbf{ret} =_{ad} \mathbf{ret}(\lambda(x, k').\,\mathcal{C}[\![N]\!]k')$, so the result is immediate by Proposition 12. □

Lemma 15. *If $M \sim P$ and $M \not\downarrow$, then $P \not\Downarrow$.*

Proof. A stuck λ-term, in either CBN or CBV, is one of the form $E[x]$ for some free x. By Proposition 10, $\mathcal{C}[\![E[x]]\!]\,\mathbf{ret} \longmapsto\!\!\!\!\!\rightarrow_{ad} \mathcal{C}[\![x]\!]K \longmapsto_{ad} xK \not\downarrow$. So $\mathcal{C}[\![E[x]]\!]\,\mathbf{ret} \not\Downarrow$, and hence by Proposition 13, $P \not\Downarrow$. □

We also have the first commuting diagram, showing that $\sim$ relates a source term to its translation:

Lemma 16. $M \sim \mathcal{C}[\![M]\!]\,\mathbf{ret}$.

Proof. Unfolding definitions, we need that $\mathcal{C}[\![M]\!]\,\mathbf{ret} =_{ad} \mathcal{C}[\![M]\!]\,\mathbf{ret}$, which is immediate since $=_{ad}$ is reflexive. □

To show correctness of $\mathcal{C}$ it remains to satisfy the second commuting diagram, showing that our invariant $\sim$ is preserved under reduction. We prove this in three steps:

1. Show that, if $M \longmapsto N$ by a β-reduction in the empty context, then we have $\mathcal{C}[\![M]\!]K \longmapsto\!\!\!\!\!\rightarrow\!\!=_{ad} \mathcal{C}[\![N]\!]K$ (Proposition 18).
2. Allow the reduction $M \longmapsto N$ to occur in any evaluation context, not only at the top of the term (Proposition 19).
3. Let the CPS term be *any* $P =_{ad} \mathcal{C}[\![M]\!]\,\mathbf{ret}$ (Lemma 20).

We can get the first step using a simple proposition concerning substitution:

Proposition 17.

1. $\mathcal{C}[\![M]\!]\{\mathcal{C}[\![N]\!]/x\} \longrightarrow\!\!\!\!\!\rightarrow_{ad} \mathcal{C}[\![M\{N/x\}]\!]$
2. $\mathcal{C}[\![M]\!]\{(\bar{\lambda}k.\,\mathcal{C}[\![N]\!]k)/x\} \longrightarrow\!\!\!\!\!\rightarrow_{ad} \mathcal{C}[\![M\{N/x\}]\!]$

Proof.

1. By induction on the structure of M. The most interesting case is when $M \equiv x$, and thus we actually substitute N for x:

$$\mathcal{C}[\![x]\!]\{\mathcal{C}[\![N]\!]/x\} \triangleq \bar{\lambda}k.\,\mathcal{C}[\![N]\!]k$$

By inspection of $\mathcal{C}$, $\mathcal{C}[\![N]\!]$ must be some administrative λ-abstraction of the form $(\bar{\lambda}k'.\, P)$:

$$\begin{aligned}
&\triangleq \bar{\lambda}k.\, (\bar{\lambda}k'.\, P)k \\
&\longrightarrow_{ad} \bar{\lambda}k.\, (P\{k/k'\}) \\
&\equiv \bar{\lambda}k'.\, P \\
&\equiv \mathcal{C}[\![N]\!]
\end{aligned}$$

2. Similar, only with one extra reduction at the beginning:

$$\begin{aligned}
\mathcal{C}[\![x]\!]\{\bar{\lambda}k.\, \mathcal{C}[\![N]\!]k/x\} &\triangleq (\bar{\lambda}k.\, xk)\{(\bar{\lambda}k.\, \mathcal{C}[\![N]\!]k)/x\} \\
&\triangleq (\bar{\lambda}k.\, (\bar{\lambda}k.\, \mathcal{C}[\![N]\!]k)k) \\
&\longrightarrow_{ad} \bar{\lambda}k.\, \mathcal{C}[\![N]\!]k \\
&\longrightarrow_{ad} \mathcal{C}[\![N]\!] \text{ (as before)}
\end{aligned}$$

□

And now:

Proposition 18. *If $M \longmapsto N$ by a reduction in the empty context, then $\mathcal{C}[\![M]\!]K \longmapsto^{+}_{pr^1} =_{ad} \mathcal{C}[\![N]\!]K$.*

Proof. Since the reduction takes place at the top level, we must have that M is a redex. From here, we must consider CBN and CBV separately:

- In CBN, M must have the form $(\lambda x.\, M')N'$ with $N \equiv M'\{N'/x\}$, and:

$$\begin{aligned}
\mathcal{C}[\![M]\!]K &\equiv \mathcal{C}[\![(\lambda x.\, M')N']\!]K \\
&\longmapsto\!\!\!\!\rightarrow_{ad} (\lambda(x, k').\, \mathcal{C}[\![M]\!]k')(\lambda k.\, \mathcal{C}[\![N]\!]k, K) \\
&\longmapsto_{pr} (\mathcal{C}[\![M]\!]k)\{\lambda k.\, \mathcal{C}[\![N]\!]k/x\} \\
&\longrightarrow\!\!\!\!\rightarrow_{ad} \mathcal{C}[\![M'\{N'/x\}]\!]K \qquad \text{(by Proposition 17)} \\
&\equiv \mathcal{C}[\![N]\!]K
\end{aligned}$$

- In CBV, M must have the more specific form $(\lambda x.\, M')V$ with $N \equiv M'\{V/x\}$. Let $V \triangleq \lambda y.\, N'$.

$$\begin{aligned}
\mathcal{C}[\![M]\!]K &\equiv \mathcal{C}[\![(\lambda x.\, M')(\lambda y.\, N')]\!]K \\
&\longmapsto\!\!\!\!\rightarrow_{ad} (\lambda(x, k').\, \mathcal{C}[\![M']\!]k')((\bar{\lambda}h.\, h(\lambda(y, h').\, \mathcal{C}[\![N']\!]h')), K) \\
&\triangleq (\lambda(x, k').\, \mathcal{C}[\![M']\!]k')(\mathcal{C}[\![\lambda y.\, N']\!], K) \\
&\triangleq (\lambda(x, k').\, \mathcal{C}[\![M']\!]k')(\mathcal{C}[\![V]\!], K) \\
&\longmapsto_{pr} \mathcal{C}[\![M']\!]\{\mathcal{C}[\![V]\!]/x\}K \\
&\longrightarrow\!\!\!\!\rightarrow_{ad} \mathcal{C}[\![M'\{V/x\}]\!]K \qquad \text{(by Proposition 17)} \\
&\equiv \mathcal{C}[\![N]\!]K
\end{aligned}$$

□

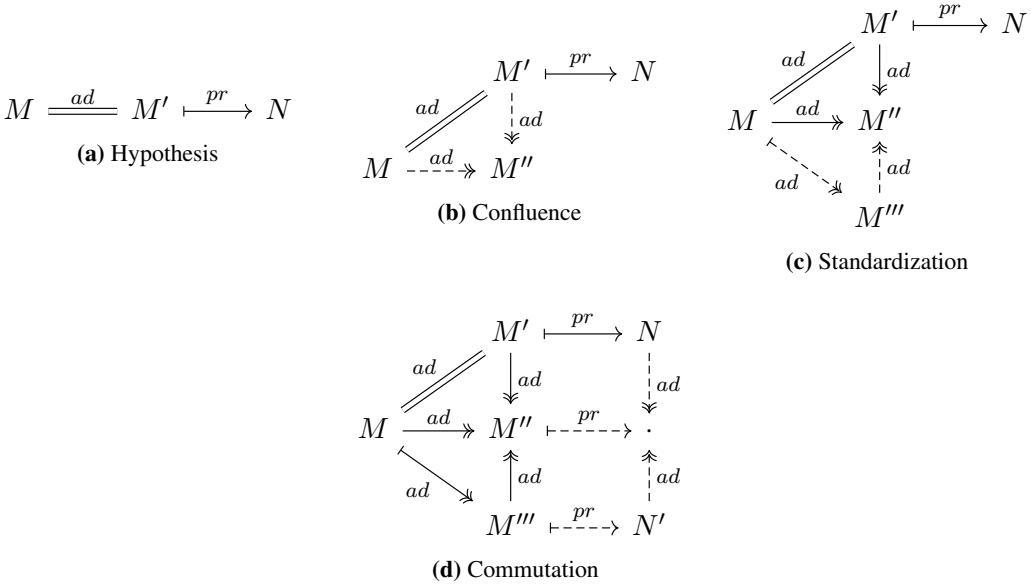
The second step is to show that a reduction in an evaluation context is performed faithfully by the CPS-transformed term.

Proposition 19. *If $M \longmapsto N$, then $\mathcal{C}[\![M]\!]K \longmapsto^{+}_{pr^1} =_{ad} \mathcal{C}[\![N]\!]K$.*

Proof. By definition of $\longmapsto$, we have that $M \equiv E[M']$ and $N \equiv E[N']$, where $M' \longmapsto N'$ at top level.

$$\begin{aligned}
\mathcal{C}[\![M]\!]\,\mathbf{ret} &\equiv \mathcal{C}[\![E[M']]\!]K \\
&\longmapsto\!\!\!\!\rightarrow_{ad} \mathcal{C}[\![M']\!]K' \text{ (by Proposition 10)} \\
&\longmapsto^{+}_{pr^1} =_{ad} \mathcal{C}[\![N']\!]K' \text{ (by Proposition 18)} \\
&\leftarrow\!\!\!\!\mapsto_{ad} \mathcal{C}[\![E[N']]\!]\,\mathbf{ret} \text{ (property of } K' \text{ from Proposition 10)} \\
&\equiv \mathcal{C}[\![N]\!]K
\end{aligned}$$

□



(a) Hypothesis

(b) Confluence

(c) Standardization

(d) Commutation

Figure 9. A summary of the proof of Lemma 9.

The third step of the proof is the most difficult: We must generalize the hypothesis of Proposition 19 so that it may be chained through multiple reduction steps. This hinges on the commutation lemma, which we now prove:

Proof (of Lemma 9). By Proposition 7.1, we know there must be some M'' such that $M \longrightarrow\!\!\!\!\!\rightarrow_{ad} M'' \longleftarrow\!\!\!\!\!\leftarrow_{ad} M'$. By Proposition 8, M'' can take a proper standard reduction, so Proposition 7.2 applies, giving M''' with $M \longmapsto\!\!\!\!\!\rightarrow_{ad} M''' \longrightarrow\!\!\!\!\!\rightarrow_{ad} M''$. From there, we use Proposition 8 to "fill in the arrows" (see Fig. 9). □

From there, we have that reduction preserves the invariant $\sim$:

Lemma 20. *If $M \longmapsto N$ and $M \sim P$, then there is Q such that $P \longmapsto^{+} Q$ and $N \sim Q$.*

Proof. Since $M \sim P$, we have $P =_{ad} \mathcal{C}[\![M]\!]\,\mathbf{ret}$. By Proposition 19, then, $\mathcal{C}[\![M]\!]\,\mathbf{ret} \longmapsto^{+} =_{ad} \mathcal{C}[\![N]\!]\,\mathbf{ret}$, so $P =_{ad} \longmapsto^{+}_{pr^1} =_{ad} \mathcal{C}[\![N]\!]\,\mathbf{ret}$. Since $\longmapsto^{+}_{pr^1}$ is short for $\longmapsto\!\!\!\!\!\rightarrow_{ad} \longmapsto_{pr} \longmapsto\!\!\!\!\!\rightarrow_{ad}$, we have $P =_{ad} \longmapsto_{pr} =_{ad} \mathcal{C}[\![N]\!]\,\mathbf{ret}$. By Lemma 9, we have Q such that $P \longmapsto^{+} Q =_{ad} \mathcal{C}[\![N]\!]\,\mathbf{ret}$, so $P \longmapsto^{+} Q$ and $N \sim Q$. □

And finally, we can show that the uniform CPS transform preserves observations forward:

Theorem 21. *For any λ-term M and variable k:*

1. *If $M \Downarrow$ then $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Downarrow$.*
2. *If $M \Uparrow$ then $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Uparrow$.*
3. *If $M \not\Downarrow$ then $\mathcal{C}[\![M]\!]\,\mathbf{ret} \not\Downarrow$.*

Proof. By Lemma 16, we can generalize $\mathcal{C}[\![M]\!]\,\mathbf{ret}$ to any P with $M \sim P$. From there, 1 and 3 follow by induction on the reduction sequence and Lemmas 14, 15, and 20. 2 is immediate from Lemma 20, since P must take at least as many steps as M. □

Corollary 22. *For any λ-term M and variable k:*

1. *$M \Downarrow$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Downarrow$.*
2. *$M \Uparrow$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Uparrow$.*
3. *$M \not\Downarrow$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} \not\Downarrow$.*

Proof. The forward directions are Theorem 21. For the backward direction of 1, assume M does not reduce to an answer. Then it must either diverge or become stuck. By Theorem 21, in either case, $\mathcal{C}[\![M]\!]\,\mathbf{ret}$ must do the same, and thus (by determinism) it cannot reduce to an answer; by contraposition, $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Downarrow$ implies $M \Downarrow$. The other clauses are similar. □

2.4.3 Correctness of the Naming Transform

Passing names instead of values has a subtle effect on the execution of a program: The CPS terms now express *sharing*. Since values aren't copied but are shared among subterms, relating reductions of unnamed terms to those of named terms requires care. For instance, consider this CPS term:

$$M \triangleq (\lambda x.\, f(x,x))(\lambda x.\, N)$$

It β-reduces and duplicates $(\lambda x.\, N)$:

$$M \longmapsto f(\lambda x.\, N, \lambda x.\, N)$$

Now consider M under the naming transform :

$$\mathcal{N}[\![M]\!] \triangleq \nu y.\, y \coloneqq \lambda x.\, \mathcal{N}[\![N]\!]\ \mathbf{in}\ (\lambda x.\, f(x,x))y$$

It only duplicates the name y:

$$\mathcal{N}[\![M]\!] \longmapsto \nu y.\, y \coloneqq \lambda x.\, \mathcal{N}[\![N]\!]\ \mathbf{in}\ f(y,y)$$

Notice, however, that if we reduce M and *then* translate, we get something different:

$$\mathcal{N}[\![f(\lambda z.\, N, \lambda z.\, N)]\!] \triangleq \nu x.\, x \coloneqq \lambda z.\, \mathcal{N}[\![N]\!]\ \mathbf{in}\ \nu y.\, y \coloneqq \lambda z.\, \mathcal{N}[\![N]\!]\ \mathbf{in}\ f(x,y)$$

Now there is no sharing of the value $\lambda z.\, N$.

In short, reduction does not *commute* with naming: Reducing the named term can produce shared references that do not appear when naming the reduced term. However, differences in sharing do not affect the outcome of the computation. Therefore we seek a way to reason *up to sharing*—that is, we want to consider a term with the same computational content, but more sharing, as "close enough." A straightforward way to remove sharing from the picture is to consider terms under a *readback* function that "flattens" a term's bound variables, returning it to the unnamed form:

$$\mathcal{N}^{-1}\langle\!\langle f(x^+)\rangle\!\rangle \triangleq f(x^+)$$
$$\mathcal{N}^{-1}\langle\!\langle \nu x.\, x \coloneqq \lambda(x^+).\, N\ \mathbf{in}\ M\rangle\!\rangle \triangleq \mathcal{N}^{-1}\langle\!\langle M\rangle\!\rangle\{\lambda(x^+).\mathcal{N}^{-1}\langle\!\langle N\rangle\!\rangle/x\}$$

We can now define our invariant as follows:

Definition 23. *For an unnamed CPS term M and a named CPS term P, let $M \sim P$ when $M \equiv \mathcal{N}^{-1}\langle\!\langle P\rangle\!\rangle$.*

To show correctness of the naming step we will show that $\sim$ is a *backward* simulation: If $M \sim P$ and $P \longmapsto P'$ then there is M' with $M \longmapsto\!\!\!\!\!\rightarrow\ M'$ and $M' \sim P'$. Note that, because *deref* reductions disappear under $\mathcal{N}^{-1}$, the correspondence between reductions is not one-to-at-least-one, as it was in (6). We relied on this property in proving Theorem 21.2, so we will have to adjust our reasoning this time.

First, we prove that reductions are preserved:

Proposition 24. *Given a $\lambda_{cps,vn}$ term P:*

1. *If $P \longmapsto_\beta P'$, then $\mathcal{N}^{-1}\langle\!\langle P\rangle\!\rangle \longmapsto \mathcal{N}^{-1}\langle\!\langle P'\rangle\!\rangle$.*
2. *If $P \longmapsto_{deref} P'$, then $\mathcal{N}^{-1}\langle\!\langle P\rangle\!\rangle \equiv \mathcal{N}^{-1}\langle\!\langle P'\rangle\!\rangle$.*

Proof. Given any context E in the named CPS language, let σ_E be the substitution built up by the unnaming function as it traverses E. (In other words, σ_E is the substitution such that $\mathcal{N}^{-1}\langle\!\langle E[P] \rangle\!\rangle \equiv \mathcal{N}^{-1}\langle\!\langle P \rangle\!\rangle \sigma_E$.)

1. If $P \longmapsto P'$ by the β rule, we have that:

$$\begin{aligned} P &\equiv E[(\lambda(x^+).\, Q)(y^+)] \\ P' &\equiv E[Q\{y^+/x^+\}] \\ \mathcal{N}^{-1}\langle\!\langle P \rangle\!\rangle &\triangleq (\lambda(x^+).\, \mathcal{N}^{-1}\langle\!\langle Q \rangle\!\rangle)(y^+)\sigma_E \\ &\longmapsto \mathcal{N}^{-1}\langle\!\langle Q \rangle\!\rangle\{y^+/x^+\}\sigma_E \\ &\equiv \mathcal{N}^{-1}\langle\!\langle E[Q\{y^+/x^+\}] \rangle\!\rangle \\ &\equiv \mathcal{N}^{-1}\langle\!\langle P' \rangle\!\rangle \end{aligned}$$

2. If $P \longmapsto P'$ by the $deref$ rule, we have that:

$$\begin{aligned} P &\equiv E[\nu f.\, f \coloneqq \lambda(x^+).\, Q \textbf{ in } E'[f(y^+)]] \\ P' &\equiv E[\nu f.\, f \coloneqq \lambda(x^+).\, Q \textbf{ in } E'[(\lambda(x^+).\, Q(y^+)]] \\ \mathcal{N}^{-1}\langle\!\langle P \rangle\!\rangle &\triangleq f(y^+)\sigma_{E'}\{\lambda(x^+).\, Q/f\}\sigma_E \\ &\equiv (\lambda(x^+).\, Q)(y^+)\sigma_{E'}\{\lambda(x^+).\, Q/f\}\sigma_E \\ &\equiv \mathcal{N}^{-1}\langle\!\langle P' \rangle\!\rangle \end{aligned}$$

□

Now we can show that $\mathcal{N}^{-1}$ preserves observable behavior:

Theorem 25.

1. If $\mathcal{N}[\![M]\!] \Downarrow$ then $M \Downarrow$.
2. If $\mathcal{N}[\![M]\!] \Uparrow$ then $M \Uparrow$.
3. If $\mathcal{N}[\![M]\!] \not\Downarrow$ then $M \not\Downarrow$.

Proof. The initial condition $M \sim \mathcal{N}[\![M]\!]$ follows from the fact that $\mathcal{N}$ and $\mathcal{N}^{-1}$ form a retraction pair: for any unnamed CPS term M, $\mathcal{N}^{-1}\langle\!\langle \mathcal{N}[\![M]\!] \rangle\!\rangle \equiv M$. The invariant holds by Proposition 24. Since answers and stuck terms are virtually the same between unnamed and named terms, the final conditions are trivial. It only remains to show that divergence is preserved. This is easy, however, as each $deref$-reduction always produces a β-redex, so M and $\mathcal{N}[\![M]\!]$ must take the same number of β-reductions. □

As we did for the uniform CPS transform (only in reverse), we get the forward directions from the backward ones, completing the correctness proof.

Corollary 26.

1. $M \Downarrow$ iff $\mathcal{N}[\![M]\!] \Downarrow$.
2. $M \Uparrow$ iff $\mathcal{N}[\![M]\!] \Uparrow$.
3. $M \not\Downarrow$ iff $\mathcal{N}[\![M]\!] \not\Downarrow$.

Proof. The backward directions are Theorem 25. The forward directions use case analysis and determinism in the same way as Corollary 22. □

2.5 CPS and Processes

The π-calculus describes computation as the exchange of simple messages by independent agents, called *processes*. Each term in the π-calculus describes a process, and processes are built by *composing* them together in parallel, *prefixing* them with I/O actions, and *replicating* them. Communication takes place over *channels*, each of which has a *name*; processes interact when one is writing to a channel and, in parallel, another is reading

$$\text{Processes:} \qquad P, Q ::= \overline{x}\langle y^+\rangle \mid x(y^+).\, P \mid (P \mid Q) \mid !P \mid \boldsymbol{\nu} z\, P$$

$$\frac{}{\overline{x}\langle y^+\rangle \mid x(z^+).\, P \longrightarrow P\{y^+/z^+\}}$$

$$\frac{P =_{st} P' \qquad P' \longrightarrow Q' \qquad Q' =_{st} Q}{P \longrightarrow Q}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \frac{P \longrightarrow P'}{\boldsymbol{\nu} z\, P \longrightarrow \boldsymbol{\nu} z\, P'}$$

$$P \mid Q =_{st} Q \mid P \qquad\qquad !P =_{st} !P \mid P$$

$$(P \mid Q) \mid R =_{st} P \mid (Q \mid R) \qquad\qquad (\boldsymbol{\nu} z\, P) \mid Q =_{st} \boldsymbol{\nu} z\, (P \mid Q)$$

$$\boldsymbol{\nu} x\, \boldsymbol{\nu} y\, P =_{st} \boldsymbol{\nu} y\, \boldsymbol{\nu} x\, P \qquad\qquad \text{if } z \text{ not free in } Q$$

Figure 10. A fragment of the π-calculus.

from it. The values sent over the channels are themselves channel names, so processes can discover each other dynamically. Names act much like variables in the λ-calculus, with α-equivalent terms identified in the same way. They can be allocated by the $\boldsymbol{\nu}$ construct, which guarantees that the name it binds will be distinct from any other allocated or free name.

The syntax and semantics for the fragment of the π-calculus we are considering are given in Fig. 10. This fragment is called the *asynchronous π-calculus* because there are no processes of the form $\overline{x}\langle y\rangle.\, P$. In other words, no process is ever blocked waiting for a write operation to complete. This property reflects the behavior of CPS terms: They never wait for a subterm to compute, instead providing a continuation that performs the remaining work.

Processes in the π-calculus are meant to be considered up to a relation called *structural congruence*, which we write as $=_{st}$.[4] The rules (other than those making $=_{st}$ an equivalence relation and a congruence) are given in Fig. 10. Reductions are closed up to structural congruence (as well as parallel composition and name allocation). Besides eliminating unimportant differences such as the order of parallel composition, structural congruence accounts for the spawning of replicated processes and the scoping of allocated names.

2.5.1 π from CPS

Despite the radically different approaches to expressing computation, CPS transforms and π-encodings are interrelated. In particular, given a CPS transform, one can systematically *derive* a π-encoding from it [27, 28]. The major difficulty in deriving a π-encoding from a CPS transform arises from an important difference: Functions in the λ-calculus can take functions as arguments, but processes in the π-calculus do not send processes over channels, only names. In other words, λ is higher-order, but π is first-order.

But we have already addressed this mismatch: The *value-named* CPS language $\lambda_{cps,vn}$ is "first-order" in much the same way. In fact, nearly every construct in the named CPS calculus $\lambda_{cps,vn}$ corresponds directly to a construct in the π-calculus:

- An application $x(y^+)$ becomes a process $\overline{x}\langle y^+\rangle$, which performs a *write* on channel x, then halts. The tuple (y^+) is transmitted over x.
- Each binding $\nu x.\, x \coloneqq \lambda(y^+).\, N$ **in** M becomes a process of the form $\boldsymbol{\nu} x\ (P \mid !x(y^+).\, Q)$. This process allocates a fresh channel name x, then runs a process P in parallel with the process $!x(y^+).\, Q$. The latter acts

[4] In the π-calculus literature, structural congruence is usually written as $\equiv$.

as a "server": It listens on the channel x for a request, then runs the process Q with the request's values as arguments. The ! makes the server process replicated, so that it handles any number of requests over time.

The only terms without couterparts are applications with λ-abstractions in head position—that is, β-redexes. But we can handle these by reducing them during translation.

Thus we can faithfully translate $\lambda_{cps,vn}$ to the π-calculus:

$$\begin{aligned}
\mathcal{P}[\![V(y^+)]\!] &\triangleq \mathcal{P}[\![V]\!]_{y^+} \\
\mathcal{P}[\![\nu x.\, x \coloneqq \lambda(y^+).\, N \textbf{ in } M]\!] &\triangleq \boldsymbol{\nu} x\, (\mathcal{P}[\![M]\!] \mid !x(y^+).\, \mathcal{P}[\![N]\!]) \\
\mathcal{P}[\![f]\!]_{y^+} &\triangleq \overline{f}\langle y^+\rangle \\
\mathcal{P}[\![\mathbf{ret}]\!]_{y^+} &\triangleq \overline{\mathbf{ret}}\langle y^+\rangle \\
\mathcal{P}[\![\lambda(x^+).\, M]\!]_{y^+} &\triangleq \mathcal{P}[\![M]\!]\{y^+/x^+\}
\end{aligned}$$

The subscripted form of $\mathcal{P}$ translates a term, given the arguments it is being applied to, performing β-reduction as needed. To translate $\mathbf{ret}$, we simply assume some fresh π-calculus channel name $\mathbf{ret}$.

Finally, we obtain the π-calculus encoding (Fig. 11) by running the uniform CPS transform $\mathcal{C}$ through the naming transform $\mathcal{N}$ and then through the π-calculus translation $\mathcal{P}$. The final product coincides with the established uniform π-encoding [28].[5]

2.5.2 Correctness

For a π-calculus term P, if P is capable of performing a write on the free channel name k (possibly after some reductions), we write $P \Downarrow_{\overline{k}}$. A named CPS term signals termination by invoking initial continuation $\mathbf{ret}$, which is translated to a write on $\mathbf{ret}$. Hence we expect the π-encoded term to write on $\mathbf{ret}$ if and only if the named CPS term would invoke $\mathbf{ret}$:

Lemma 27. *For any $\lambda_{cps,vn}$ term M, $M \Downarrow$ iff $\mathcal{P}[\![M]\!] \Downarrow_{\overline{\mathbf{ret}}}$.*

Proof. We need only that our syntactic embedding of $\lambda_{cps,vn}$ into π is also a semantic embedding—in other words, that reductions in the CPS term correspond to reductions in the π-term and vice versa. The only catch is that β-redexes from the CPS term disappear under $\mathcal{P}$; however, $\mathcal{P}$ is still a bisimulation.

Note that any CPS term $E[M]$ will translate to processes representing the bindings in E in parallel with the process representing M. Thus we can consider translating E and M separately. Then:

$$\nu f.\, f \coloneqq \lambda(x^+).\, M \textbf{ in } E[f(y^+)] \longmapsto\!\!\!\!\rightarrow \nu f.\, f \coloneqq \lambda(x^+).\, M \textbf{ in } E[M\{y^+/x^+\}]$$

corresponds to

$$\boldsymbol{\nu} f\, (\overline{f}\langle y^+\rangle \mid \mathcal{P}[\![E]\!] \mid !f(x^+).\, \mathcal{P}[\![M]\!]) \longmapsto \boldsymbol{\nu} f\, (M\{y^+/x^+\} \mid \mathcal{P}[\![E]\!] \mid !f(x^+).\, \mathcal{P}[\![M]\!]).$$

It is straightforward to construct a grammar of possible π-terms produced by $\mathcal{P}$ to show that the correspondence works in both directions. □

Finally, we have the complete proof of the correctness of the uniform π-encoding, simply by composing:

Theorem 28. *For any λ_n or λ_v term M, $M \Downarrow$ if and only if $\mathcal{E}[\![M]\!]_{\mathbf{ret}} \Downarrow_{\overline{\mathbf{ret}}}$.*

Proof. Note that $\mathcal{E}[\![M]\!]_{\mathbf{ret}} \equiv \mathcal{P}[\![\mathcal{N}[\![\mathcal{C}[\![M]\!]\, \mathbf{ret}]\!]]\!]$; the result follows by Corollaries 22 and 26 and Lemma 27. □

2.6 Uniform Abstract Machine

In addition to the π-encoding, we can derive other artifacts from the uniform CPS transform. In particular, through the functional correspondence [2], we obtain a uniform abstract machine for CBN and CBV. See Fig. 12.

[5] In fact, the final transform in Fig. 11 differs slightly from the uniform π-encoding in the literature, in that all input processes are replicated, even those used at most once (*e.g.* continuation processes). However, this is harmless, as garbage collection is sound in the π-calculus (up to bisimulation). The call-by-need π-encoding will keep some processes unreplicated, as this is necessary for correctness.

$$
\begin{aligned}
\mathcal{C}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, k(\lambda(x, k').\, \mathcal{C}[\![M]\!]k') \\
\mathcal{C}[\![MN]\!] &\triangleq \begin{cases} \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, v(\lambda k'.\, \mathcal{C}[\![N]\!]k', k)) & \text{CBN} \\ \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, \mathcal{C}[\![N]\!](\lambda w.\, v(\lambda k'.\, k'w, k))) & \text{CBV} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_{vn}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}_{vn}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, \nu f.\, f \coloneqq \lambda(x, k').\, \mathcal{C}_{vn}[\![M]\!]k' \,\mathbf{in}\, kf \\
\mathcal{C}_{vn}[\![MN]\!] &\triangleq \begin{cases} \lambda k.\, \nu k'.\, k' \coloneqq (\lambda v.\, \nu x.\, x \coloneqq \lambda k'.\, \mathcal{C}_{vn}[\![N]\!]k' \,\mathbf{in} \\ \quad v(x, k)) \,\mathbf{in}\, \mathcal{C}_{vn}[\![M]\!]k' & \text{CBN} \\ \lambda k.\, \nu k'.\, k' \coloneqq \left(\begin{array}{l} \lambda v.\, \nu k''.\, k'' \coloneqq (\lambda w. \\ \quad \nu x.\, x \coloneqq \lambda k'.\, k'w \,\mathbf{in} \\ \quad v(x, k)) \,\mathbf{in}\, \mathcal{C}_{vn}[\![N]\!]k'' \end{array}\right) \mathbf{in} & \text{CBV} \\ \quad \mathcal{C}_{vn}[\![M]\!]k' \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![x]\!]_k &\triangleq \overline{x}\langle k\rangle \\
\mathcal{E}[\![\lambda x.\, M]\!]_k &\triangleq \boldsymbol{\nu} f\, (\overline{k}\langle f\rangle \mid\, !f(x, k).\, \mathcal{E}[\![M]\!]_k) \\
\mathcal{E}[\![MN]\!]_k &\triangleq \begin{cases} \boldsymbol{\nu} k'\, (\mathcal{E}[\![M]\!]_{k'} \mid \\ \quad !k'(v).\, \boldsymbol{\nu} x\, (\overline{v}\langle x, k\rangle \mid\, !x(k'').\, \mathcal{E}[\![N]\!]_{k''})) & \text{CBN} \\ \boldsymbol{\nu} k'\, (\mathcal{E}[\![M]\!]_{k'} \mid\, k'(v).\, \boldsymbol{\nu} k''\, (\mathcal{E}[\![N]\!]_{k''} \mid \\ \quad !k''(w).\, \boldsymbol{\nu} x\, (\overline{v}\langle x, k\rangle \mid\, !x(k').\, \overline{k'}\langle w\rangle))) & \text{CBV} \end{cases}
\end{aligned}
$$

Figure 11. The CPS transform, named CPS transform, and π-encoding for CBN and CBV

The CBN fragment of the machine is essentially the same as the well-known Krivine machine for CBN evaluation [17]. On closed terms, the CBV fragment strongly resembles the CEK machine [16] (without control operators). Unlike most environment-based abstract machines, ours does not exclude open terms, and thus its behavior can meaningfully be more finely specified: Variable lookup happens when a variable is evaluated, and only λ-abstractions are treated as values. We could instead delay the variable lookup until a λ-abstraction is required; this machine would implement the full CBV β-rule. Much as with the uniform CPS transform, we can observe the difference using a term such as $(\lambda x.\, \lambda y.\, y)z$, which is stuck according to the uniform abstract machine.

3. Call-by-Need and Constructive Update

As we have seen, CBV *usually* takes fewer evaluation steps to reach an answer than CBN. However, CBV evaluation wastes work whenever a function does not use its argument. The call-by-need λ-calculus [5, 7] is efficient in both cases: Unneeded arguments are never computed, yet each argument is evaluated at most once. Hence call-by-need models efficient implementations of lazy evaluation, which *memoize*, or cache, each computed value.

The syntax and semantics are given in Fig. 13. Rather than perform a substitution, the β_{need} rule suspends the argument in a **let** binding. The grammar for evaluation contexts expresses lazy evaluation order: Given a term $\mathbf{let}\, x = M\, \mathbf{in}\, N$, we evaluate N until it becomes either a value or a term of the form $F[x]$ for some evaluation context F. Such a term *needs* the value of x to continue, so now we evaluate the term M that x is bound to. But since this computation is done in place, it only needs to be done once: After M becomes a value, this value will

$$
\begin{array}{rl}
\text{Terms:} & M, N ::= x \mid \lambda x.\, M \mid MN \\
\text{Continuations:} & k ::= \mathsf{Ret}\langle\rangle \mid \mathsf{ApplyN}\langle M, k, \rho\rangle \mid \\
& \qquad \mathsf{ApplyV1}\langle M, k, \rho\rangle \mid \mathsf{ApplyV2}\langle \lambda x.\, M, k, \rho\rangle \\
\text{Environments:} & \rho ::= \epsilon \mid \rho[x = \mathsf{Clos}\langle M, \rho\rangle] \\
\text{States:} & S ::= \langle M, k, \rho\rangle_M \mid \langle k, \lambda x.\, M, \rho\rangle_K \mid \langle \lambda x.\, M, \rho\rangle_H
\end{array}
$$

$$
M \longmapsto \langle M, \mathsf{Ret}\langle\rangle, \epsilon\rangle_M
$$

$$
\begin{array}{rl}
\langle x, k, \rho\rangle_M & \longmapsto \langle M, k, \rho'\rangle_M \\
\text{where } \rho(x) & \equiv \mathsf{Clos}\langle M, \rho'\rangle \\
\langle \lambda x.\, M, k, \rho\rangle_M & \longmapsto \langle k, \lambda x.\, M, \rho\rangle_K \\
\langle MN, k, \rho\rangle_M & \longmapsto \langle M, k', \rho\rangle_M \\
\text{where } k' & \triangleq \begin{cases} \mathsf{ApplyN}\langle N, k, \rho\rangle & \text{CBN} \\ \mathsf{ApplyV1}\langle N, k, \rho\rangle & \text{CBV} \end{cases}
\end{array}
$$

$$
\begin{array}{rl}
\langle \mathsf{Ret}\langle\rangle, \lambda x.\, M, \rho\rangle_K & \longmapsto \langle \lambda x.\, M, \rho\rangle_H \\
\langle \mathsf{ApplyN}\langle N, k, \rho'\rangle, \lambda x.\, M, \rho\rangle_K & \longmapsto \langle M, k, \rho[x = \mathsf{Clos}\langle N, \rho'\rangle]\rangle_M \\
\langle \mathsf{ApplyV1}\langle N, k, \rho'\rangle, \lambda x.\, M, \rho\rangle_K & \longmapsto \langle N, \mathsf{ApplyV2}\langle \lambda x.\, M, k, \rho\rangle, \rho'\rangle_M \\
\langle \mathsf{ApplyV2}\langle \lambda y.\, N, k, \rho'\rangle, \lambda x.\, M, \rho\rangle_K & \longmapsto \langle N, k, \rho'[y = \mathsf{Clos}\langle \lambda x.\, M, \rho\rangle]\rangle_M
\end{array}
$$

Figure 12. Uniform abstract machine for CBN and CBV

simply be substituted directly (by the *deref* rule) if x is needed again. If N instead becomes a value, then we say the whole term is an *answer*—a value surrounded by a number of **let** bindings.

An answer in call-by-need is "almost a value." Evaluation stops when a term becomes an answer, but it's not a value for the purposes of the β or *deref* rule. When a subterm evaluates to an answer, either the *lift* or the *assoc* rule moves each binding into the outer environment.

To illustrate, we turn to our previous example, $(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z))$. It reduces as follows:

$$
\begin{array}{l}
(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z)) \\
\quad \longmapsto_{\beta_{need}} \mathbf{let}\ x = (\lambda y.\, y)(\lambda z.\, z)\ \mathbf{in}\ xx \\
\quad \longmapsto_{\beta_{need}} \mathbf{let}\ x = (\mathbf{let}\ y = \lambda z.\, z\ \mathbf{in}\ y)\ \mathbf{in}\ xx \\
\quad \longmapsto_{\mathit{deref}} \mathbf{let}\ x = (\mathbf{let}\ y = \lambda z.\, z\ \mathbf{in}\ \lambda z.\, z)\ \mathbf{in}\ xx \\
\quad \longmapsto_{\mathit{assoc}} \mathbf{let}\ y = \lambda z.\, z\ \mathbf{in}\ \mathbf{let}\ x = \lambda z.\, z\ \mathbf{in}\ xx \\
\quad \longmapsto_{\mathit{deref}} \mathbf{let}\ y = \lambda z.\, z\ \mathbf{in}\ \mathbf{let}\ x = \lambda z.\, z\ \mathbf{in}\ (\lambda z.\, z)x \\
\quad \longmapsto_{\beta_{need}} \mathbf{let}\ y = \lambda z.\, z\ \mathbf{in}\ \mathbf{let}\ x = \lambda z.\, z\ \mathbf{in}\ \mathbf{let}\ z = x\ \mathbf{in}\ z \\
\quad \longmapsto_{\mathit{deref}} \mathbf{let}\ y = \lambda z.\, z\ \mathbf{in}\ \mathbf{let}\ x = \lambda z.\, z\ \mathbf{in}\ \mathbf{let}\ z = \lambda z.\, z\ \mathbf{in}\ z \\
\quad \longmapsto_{\mathit{deref}} \mathbf{let}\ y = \lambda z.\, z\ \mathbf{in}\ \mathbf{let}\ x = \lambda z.\, z\ \mathbf{in}\ \mathbf{let}\ z = \lambda z.\, z\ \mathbf{in}\ \lambda z.\, z
\end{array}
$$

$$
\begin{array}{rl}
\text{Expressions:} & M, N ::= x \mid \lambda x.\, M \mid MN \mid \mathbf{let}\, x = M\, \mathbf{in}\, N \\
\text{Values:} & V ::= \lambda x.\, M \\
\text{Answers:} & A ::= V \mid \mathbf{let}\, x = M\, \mathbf{in}\, A \\
\text{Evaluation Contexts:} & E, F ::= [] \mid EM \mid \mathbf{let}\, x = E\, \mathbf{in}\, F[x] \mid \\
& \qquad\qquad \mathbf{let}\, x = M\, \mathbf{in}\, E
\end{array}
$$

$$
\begin{array}{rll}
(\lambda x.\, M)N & \longrightarrow \mathbf{let}\, x = N\, \mathbf{in}\, M & \beta_{need} \\
(\mathbf{let}\, y = L\, \mathbf{in}\, A)N & \longrightarrow \mathbf{let}\, y = L\, \mathbf{in}\, AN & \mathit{lift} \\
\mathbf{let}\, x = V\, \mathbf{in}\, E[x] & \longrightarrow \mathbf{let}\, x = V\, \mathbf{in}\, E[V] & \mathit{deref} \\
\mathbf{let}\, x = (\mathbf{let}\, y = L\, \mathbf{in}\, A)\, \mathbf{in}\, E[x] & \longrightarrow \mathbf{let}\, y = L\, \mathbf{in}\, \mathbf{let}\, x = A\, \mathbf{in}\, E[x] & \mathit{assoc}
\end{array}
$$

Figure 13. The call-by-need λ-calculus, λ_{need}, of Ariola et al.[7]

Evaluation begins in a call-by-name manner, in the sense that the outer β-redex is reduced immediately. The argument is suspended in a **let** binding. Next, since x is in head position in the body, we need to evaluate it, and so we reduce the inner β-redex. However, since the reduction is done in place, this step will only be done once. In all, three β-reductions are performed, as few as is done by call-by-value.

Unlike call-by-value, when a function ignores its argument, call-by-need does not waste work. In extreme cases, call-by-value *never finishes* when call-by-name or call-by-need would. For example, consider the term

$$(\lambda x.\, \lambda y.\, y)\Omega,$$

where Ω is a term $(\lambda x.\, xx)(\lambda x.\, xx)$ that diverges. Call-by-name substitutes the Ω immediately (the substitution is trivial since x does not occur in the body). Call-by-need similarly suspends the Ω without attempting to evaluate it. Call-by-value insists on computing the argument first, and thus is caught in an infinite loop.

3.1 Continuation-Passing Style

There does exist a call-by-need CPS transform due to Okasaki et al. [22] It requires mutable storage, which our CPS languages do not support. However, suppose we borrow the assignment syntax from the named CPS language $\lambda_{cps,vn}$. Then we can build on the uniform CPS transform (Fig. 3) and use a call-by-need application rule:

$$
\begin{array}{l}
\mathcal{C}[\![MN]\!] \triangleq \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, \nu x.\, x := \\
\qquad \big(\lambda k'.\, \mathcal{C}[\![N]\!](\lambda w.\, x := \lambda k''.\, k'' w\, \mathbf{in}\, k' w)\big)\, \mathbf{in}\, v(x, k))
\end{array}
$$

This is roughly the same as in the Okasaki CPS transform. Unfortunately, this is not valid $\lambda_{cps,vn}$ syntax, as the assignment operator $:=$ is only allowed immediately inside a ν binding for the variable assigned to. As a result, $\lambda_{cps,vn}$ only allows an assignment to a variable that presently has no value. The inner continuation, $\lambda w.\, x := \lambda k''.\, k'' w\, \mathbf{in}\, k' w$, violates this restriction by attempting to "overwrite" x. We will call this a *double assignment*.

Of course, this is precisely what we wish to happen: The term bound to x *should* change, in order to cache the computed value. But we don't need the full power of mutable storage; a much weaker effect will suffice.

To see this, suppose for a moment we allow $:=$ anywhere, with the semantics of destructive update (that is, each assignment overwrites any previous one). Inspecting the rule, we see that each variable is now assigned to (at most) twice: Once when it is initialized with a thunk, and again when the thunk's result is memoized.

$$
\begin{aligned}
\mathcal{C}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, k(\lambda(x, k').\, \mathcal{C}[\![M]\!]k') \\
\mathcal{C}[\![MN]\!] &\triangleq \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, \nu x.\, x :=_1 \\
&\qquad \big(\lambda k'.\, \mathcal{C}[\![N]\!](\lambda w.\, x := \lambda k''.\, k''w \,\mathbf{in}\, k'w)\big) \,\mathbf{in}\, v(x, k)) \\
\mathcal{C}[\![\mathbf{let}\, x = L \,\mathbf{in}\, M]\!] &\triangleq \lambda k.\, \nu x.\, x :=_1 \\
&\qquad \big(\lambda k'.\, \mathcal{C}[\![L]\!](\lambda w.\, x := \lambda k''.\, k''w \,\mathbf{in}\, k'w)\big) \,\mathbf{in}\, \mathcal{C}[\![M]\!]k
\end{aligned}
$$

Figure 14. A call-by-need CPS transform using constructive update

However, after the *second* assignment, the stored value never changes *again*. Furthermore, note that the initial thunk cannot refer to x, even indirectly, as x is not in the scope of the computation (our **let** is not recursive). Therefore the initial thunk is only used once; since that very thunk performs the second assignment, the first lookup must precede the second assignment, with no other accesses in between.

In the language of data-flow analysis, after the first lookup, x cannot be *live*. Hence its value does not matter. In other words, it may as well *have no value*. If we clear x after the first lookup, then the second assignment is just like the first: It is giving a value to a variable that currently has none. There is no double assignment.

This analysis suggests a special assignment operation that always clears the variable the next time it is used. The assigned value will therefore only be used once, and thus the assignment is *ephemeral*, as opposed to *permanent*. After a permanent assignment, the variable will never be cleared, so permanent assignments are final.

3.1.1 The Transform

Writing $x := M \,\mathbf{in}\, N$ for a permanent assignment and $x :=_1 M \,\mathbf{in}\, N$ for an ephemeral assignment, we can modify the call-by-need CPS transform so that it does not require destructive update:

$$
\begin{aligned}
\mathcal{C}[\![MN]\!] \triangleq \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, \nu x.\, x :=_1 \\
\big(\lambda k'.\, \mathcal{C}[\![N]\!](\lambda w.\, x := \lambda k''.\, k''w \,\mathbf{in}\, k'w)\big) \,\mathbf{in}\, v(x, k))
\end{aligned}
$$

Since the initial thunk is now assigned ephemerally, there is never a double assignment. In fact, we can prove so: By the above data-flow analysis, x is unassigned before the second assignment. Each assignment is performed by a term that is used at most once, and thus no further assignments will be attempted.

Note what has happened here: x takes on different values over time, due to multiple assignments. Therefore it is fair to say it was *updated*. However, no previous value was destroyed by any update, and in fact a previous value *cannot* be destroyed. In this language, updates only construct, never destroy; hence we call the phenomenon *constructive update*.

This CPS transform, summarized in Fig. 14, is the one we will relate to the π-calculus. The syntax and semantics for permanent and ephemeral assignment are given in Fig. 15. The $deref_1$ rule is similar to $deref$, only it removes the ephemeral assignment.

We have shown that terms produced by the call-by-need CPS transform never attempt a *double assignment*—that is, they never reduce to a term such as $x := V \,\mathbf{in}\, x := W \,\mathbf{in}\, M$. Let us call such terms *safe*:

Definition 29. *A $\lambda^{:=_1}_{cps}$ term M is* safe *when it does not reduce to a term with a subterm of the form*

$$x :=_* V \,\mathbf{in}\, E[x :=_* W \,\mathbf{in}\, N],$$

where $:=_*$ stands for either $:=$ or $:=_1$ in each appearance.

Proposition 30. *For any M and K, if K is a variable,* **ret***, or $\lambda v.\, P$ where P is safe, then $\mathcal{C}[\![M]\!]K$ is safe.*

$$
\begin{array}{rrl}
\text{Terms:} & M, N ::= & V(V^+) \mid \nu x.\, M \mid \\
& & \quad x \coloneqq \lambda(x^+).\, M \,\mathbf{in}\, N \mid \\
& & \quad x :=_1 \lambda(x^+).\, M \,\mathbf{in}\, N \\
\text{Values:} & V ::= & x \mid \mathbf{ret} \mid \lambda(x^+).\, M \\
\text{Binding Contexts:} & B ::= & [\,] \mid \nu x.\, B \mid x \coloneqq \lambda(x^+).\, M \,\mathbf{in}\, B \mid \\
& & \quad x :=_1 \lambda(x^+).\, M \,\mathbf{in}\, B \\
\text{Evaluation Contexts:} & E ::= & B
\end{array}
$$

$$
\begin{array}{rll}
(\lambda(x^+).\, M)(V^+) & \longrightarrow M\{V^+/x^+\} & \beta \\
f \coloneqq \lambda(x^+).\, M \,\mathbf{in}\, E[f(V^+)] & \longrightarrow f \coloneqq \lambda(x^+).\, M \,\mathbf{in} & \\
& \quad E[(\lambda(x^+).\, M)(V^+)] & \mathit{deref} \\
f :=_1 \lambda(x^+).\, M \,\mathbf{in}\, E[f(V^+)] & \longrightarrow E[(\lambda(x^+).\, M)(V^+)] & \mathit{deref}_1
\end{array}
$$

Figure 15. The CPS λ-calculus with constructive update, $\lambda^{:=_1}_{cps}$

Proof. See the above data-flow analysis. □

3.2 Naming and π-Encoding

Now that we have the call-by-need CPS transform, we need only adapt the development in Sections 2.3 and 2.5 to derive the π-calculus encoding.

In Section 2, we were somewhat sloppy in deriving a π-encoding from the CPS—all input processes in the encoding were replicated, even those that are provably used at most once. Now that we have ephemeral assignment, we can be more precise. Some λ-abstractions are *affine*, i.e. never duplicated; these are the ones representing continuations or suspended computations. We can mark the λs in the CPS transform to indicate which values are affine:

$$
\begin{array}{rl}
\mathcal{C}[\![x]\!] \triangleq & \lambda k.\, xk \\
\mathcal{C}[\![\lambda x.\, M]\!] \triangleq & \lambda k.\, k(\lambda(x, k').\, \mathcal{C}[\![M]\!]k') \\
\mathcal{C}[\![MN]\!] \triangleq & \lambda k.\, \mathcal{C}[\![M]\!](\lambda_1 v.\, \nu x.\, x :=_1 \\
& \qquad \big(\lambda k'.\, \mathcal{C}[\![N]\!](\lambda_1 w.\, x \coloneqq \lambda k''.\, k''w \,\mathbf{in}\, k'w)\big) \,\mathbf{in}\, v(x, k)) \\
\mathcal{C}[\![\mathbf{let}\, x = L \,\mathbf{in}\, M]\!] \triangleq & \lambda k.\, \nu x.\, x :=_1 \\
& \qquad \big(\lambda k'.\, \mathcal{C}[\![L]\!](\lambda_1 w.\, x \coloneqq \lambda k''.\, k''w \,\mathbf{in}\, k'w)\big) \,\mathbf{in}\, \mathcal{C}[\![M]\!]k
\end{array}
$$

Now we augment the naming transform to treat affine values specially:

$$
\mathcal{N}[\![V(\lambda_1(x^+).\, M)]\!] \triangleq \nu y.\, y :=_1 \lambda(x^+).\, \mathcal{N}[\![M]\!] \,\mathbf{in}\, \mathcal{N}[\![V(y)]\!]
$$

Finally, the π-calculus translation $\mathcal{P}$ should use an unreplicated process to simulate ephemeral assignment:

$$
\mathcal{P}[\![x :=_1 \lambda(y^+).\, M \,\mathbf{in}\, N]\!] \triangleq \mathcal{P}[\![N]\!] \mid x(y^+).\, \mathcal{P}[\![M]\!]
$$

Putting these transforms together (Fig. 16), we arrive at the same call-by-need π-encoding found in the literature [8, 28].

$$
\begin{aligned}
\mathcal{C}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, k(\lambda(x, k').\, \mathcal{C}[\![M]\!]k') \\
\mathcal{C}[\![MN]\!] &\triangleq \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, \nu x. \\
&\qquad x :=_1 \left(\lambda k'.\, \mathcal{C}[\![N]\!](\lambda w.\, x := \lambda k''.\, k''w \,\mathbf{in}\, k'v)\right) \mathbf{in}\, v(x, k)) \\
\mathcal{C}[\![\mathbf{let}\, x = L \,\mathbf{in}\, M]\!] &= \lambda k.\, \nu x.\, x :=_1 (\lambda k'.\, \mathcal{C}[\![L]\!](\lambda w.\, x := \lambda k''.\, k''w \,\mathbf{in}\, kw)) \,\mathbf{in}\, \mathcal{C}[\![M]\!]k
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_{vn}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}_{vn}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, \nu f.\, f := \lambda(x, k').\, \mathcal{C}_{vn}[\![M]\!]k' \,\mathbf{in}\, kf \\
\mathcal{C}_{vn}[\![MN]\!] &\triangleq \lambda k.\, \nu h.\, h :=_1 \lambda v.\, \nu x.\, x :=_1 \left(\begin{array}{l} \lambda k'.\, \nu h'.\, h' :=_1 \\ \quad \lambda v.\, x := \lambda k''.\, k''v \,\mathbf{in}\, k'v \\ \mathbf{in}\, \mathcal{C}_{vn}[\![N]\!]h' \end{array}\right) \mathbf{in}\, v(x, k) \\
&\qquad \mathbf{in}\, \mathcal{C}_{vn}[\![M]\!]h \\
\mathcal{C}_{vn}[\![\mathbf{let}\, x = L \,\mathbf{in}\, M]\!] &\triangleq \lambda k.\, \nu x.\, x :=_1 \left(\begin{array}{l} \lambda k.\, \nu h'.\, h' :=_1 \\ \quad \lambda v.\, x := \lambda k.\, kv \,\mathbf{in}\, kv \\ \mathbf{in}\, \mathcal{C}_{vn}[\![L]\!]h' \end{array}\right) \mathbf{in}\, \mathcal{C}_{vn}[\![M]\!]k
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![x]\!]_k &\triangleq \overline{x}\langle k\rangle \\
\mathcal{E}[\![\lambda x.\, M]\!]_k &\triangleq \boldsymbol{\nu} f\, (\overline{k}\langle f\rangle \mid !f(x, k).\, \mathcal{E}[\![M]\!]_k) \\
\mathcal{E}[\![MN]\!]_k &\triangleq \boldsymbol{\nu} h\, (\mathcal{E}[\![M]\!]_h \mid h(v).\, \boldsymbol{\nu} x\, (\overline{v}\langle x, k\rangle \mid \\
&\qquad x(k').\, \boldsymbol{\nu} h'\, (\mathcal{E}[\![N]\!]_{h'} \mid h'(w).\, (\overline{k'}\langle w\rangle \mid !x(k'').\, \overline{k''}\langle w\rangle)))) \\
\mathcal{E}[\![\mathbf{let}\, x = L \,\mathbf{in}\, M]\!]_k &\triangleq \boldsymbol{\nu} x\, (\mathcal{E}[\![M]\!]_k \mid x(k').\, \boldsymbol{\nu} h'\, (\mathcal{E}[\![L]\!]_{h'} \mid h'(w).\, (\overline{k'}\langle w\rangle \mid !x(k'').\, \overline{k''}\langle w\rangle)))
\end{aligned}
$$

Figure 16. The CPS transform, named CPS transform, and π-encoding for call-by-need

3.3 Correctness

Now we establish the correctness of the call-by-need CPS transform $\mathcal{C}$. We do so by further decomposing it into three steps: A switch to a call-by-need calculus with rules that act *at a distance*; an annotation step; and simulation proofs for the CPS transform on annotated terms.

3.3.1 Distance Rules

As presented, λ_{need} (Fig. 13) has certain reductions—the *lift* and *assoc* rules—that hardly seem to perform any computation. They only shuffle bindings around in preparation for a β_{need}- or *deref*-reduction, respectively. In fact, if $\mathcal{C}[\![M]\!]$ is always an administrative λ-abstraction[6], then *lift*- and *assoc*-reductions are simulated as administrative reductions alone. In a sense, the *lift* and *assoc* rules *are* administrative: They only serve to bring the parts of a redex together.

We can avoid administrative work in the source calculus by using a suggestion of Accattoli [1] for the π-calculus: We express λ_{need} using rules that apply *at a distance*, that is, where parts of a redex are separated by an evaluation context.[7] The new calculus, λ^{d}_{need}, supplants the fine-grained *lift* and *assoc* rules with coarser β and *deref* rules. The syntax is the same as for λ_{need} (Fig. 13), except that we specify that some evaluation contexts are *binding contexts*; the reductions are given in Fig. 17.

[6] We could always not mark these as administrative, but then we would lose the flexibility that administrative congruence provides.

[7] Of course, the *deref* rule already works this way in part.

$$\text{Binding Contexts:} \qquad B ::= [] \mid \mathbf{let}\, x = M\, \mathbf{in}\, B$$

$$B[\lambda x.\, M]N \longrightarrow B[\mathbf{let}\, x = N\, \mathbf{in}\, M] \qquad \beta_d$$
$$\mathbf{let}\, x = B[V]\, \mathbf{in}\, E[x] \longrightarrow B[\mathbf{let}\, x = V\, \mathbf{in}\, E[V]] \qquad \mathit{deref}_d$$

Figure 17. Reductions for the distance call-by-need λ-calculus, λ^d_{need}

Proposition 31. *An λ_{need} term reduces to an answer, diverges, or gets stuck by the distance rules if and only if it does so by the original rules.*

Proof. Since $\longmapsto_{\beta_d}$ is the same as $\longmapsto\!\!\!\!\rightarrow_{\mathit{lift}} \longmapsto_{\beta_{need}}$, $\longmapsto_{\mathit{deref}_d}$ is the same as $\longmapsto\!\!\!\!\rightarrow_{\mathit{assoc}} \longmapsto_{\mathit{deref}}$, and the language is deterministic, it suffices to define a backward simulation that compares terms up to *lift* and $assoc$. □

3.3.2 Annotations

The single $\mathbf{let}$ construct does not tell the full story of a standard reduction sequence in λ_{need}: There is an implicit statefulness that is made manifest by the CPS transform. Specifically, there are three stages in the life cycle of a $\mathbf{let}$ binding:

Suspended Initially, the binding $\mathbf{let}\, x = M\, \mathbf{in}\, N$ represents a *suspended* computation. Computation takes place within N.

Active For x to be demanded, N must reduce to the form $E[x]$. Then the binding becomes *active*, with the form $\mathbf{let}\, x = M\, \mathbf{in}\, E[x]$, and computation takes place within M.

Memoized Eventually, M becomes an answer $B[V]$, and the body $E[x]$ receives V while the bindings in B are added to the environment. Subsequently, the binding is $\mathbf{let}\, x = V\, \mathbf{in}\, N$, and computation takes place within N.

The CPS translation exposes this state, which makes it difficult to relate a term to its CPS form: In what state is $\mathbf{let}\, x = V\, \mathbf{in}\, E[x]$? In fact, it could be in *any* of the three states, and thus the running CPS program could have any of three forms.

Therefore we annotate each $\mathbf{let}$, giving it a subscript $\mathbf{s}$, $\mathbf{a}$, or $\mathbf{v}$ (for *suspended*, *active*, or *value*, respectively). We will need new reduction rules, which we call act and $deact$, to represent binding state transitions; from the perspective of λ^d_{need}, these will be administrative. See Fig. 18 for the resulting calculus λ^a_{need}. We write $\mathcal{A}[\![M]\!]$ for the annotation of a λ^d_{need} term M, which consists simply of tagging each $\mathbf{let}$ with $\mathbf{s}$.

Proposition 32.

1. *$M \Downarrow$ if and only if $\mathcal{A}[\![M]\!] \Downarrow$.*
2. *$M \Uparrow$ if and only if $\mathcal{A}[\![M]\!] \Uparrow$.*
3. *$M \not\Downarrow$ if and only if $\mathcal{A}[\![M]\!] \not\Downarrow$.*

3.3.3 Annotated CPS Transform

Now we must show that the call-by-need CPS transform, as a function from λ^a_{need} to $\lambda^{:=_1}_{cps}$, is correct. First, we need to extend the CPS transform to the annotated terms. See Fig. 19, where we have also marked which λ-abstractions are administrative.

Most of the first part of $\mathcal{C}_a$ is unsurprising: A $\mathbf{let_s}$ translates as a suspended computation, as appears in $\mathcal{C}_a[\![MN]\!]$. A $\mathbf{let_v}$ translates as a memo-thunk. However, $\mathbf{let_a}$ is a challenge. To see why, consider a suspended computation:

$$\mathcal{C}_a[\![\mathbf{let_s}\, x = M\, \mathbf{in}\, N]\!]K =_{ad} \nu x.\, x :=_1 \bar{\lambda} k.\, \cdots\, \mathbf{in}\, \mathcal{C}_a[\![N]\!]K$$

$$
\begin{array}{rrl}
\text{Expressions:} & M, N ::= & x \mid V \mid MN \mid \\
& & \mathbf{let_s}\, x = M \,\mathbf{in}\, N \mid \\
& & \mathbf{let_a}\, x = M \,\mathbf{in}\, E[x] \mid \\
& & \mathbf{let_v}\, x = V \,\mathbf{in}\, N \\
\text{Evaluation Contexts:} & E, F ::= & [\,] \mid EM \mid \\
& & \mathbf{let_s}\, x = M \,\mathbf{in}\, E \mid \\
& & \mathbf{let_a}\, x = E \,\mathbf{in}\, F[x] \mid \\
& & \mathbf{let_v}\, x = V \,\mathbf{in}\, E \\
\text{Binding Contexts:} & B ::= & [\,] \mid \mathbf{let_s}\, x = M \,\mathbf{in}\, B \mid \\
& & \mathbf{let_v}\, x = V \,\mathbf{in}\, B
\end{array}
$$

$$
\begin{array}{rll}
B[\lambda x.\, M]N \longrightarrow B[\mathbf{let_s}\, x = N \,\mathbf{in}\, M] & & \beta_a \\
\mathbf{let_s}\, x = M \,\mathbf{in}\, E[x] \longrightarrow \mathbf{let_a}\, x = M \,\mathbf{in}\, E[x] & & act \\
\mathbf{let_a}\, x = B[V] \,\mathbf{in}\, E[x] \longrightarrow B[\mathbf{let_v}\, x = V \,\mathbf{in}\, E[V]] & & deact \\
\mathbf{let_v}\, x = V \,\mathbf{in}\, E[x] \longrightarrow \mathbf{let_v}\, x = V \,\mathbf{in}\, E[V] & & deref_a
\end{array}
$$

Figure 18. The annotated call-by-need λ-calculus, λ^a_{need}

The computation of x is suspended, pending its need. Then, x will be needed when N reduces to a term of the form $E[x]$. At that point in the computation, we expect $\mathcal{C}_a[\![E[x]]\!]$ to have evaluated to some term $E'[xK']$, where E' is a CPS evaluation context and K' is a continuation.

$$\mathcal{C}_a[\![\mathbf{let_s}\, x = M \,\mathbf{in}\, E[x]]\!]K =_{ad} \nu x.\, x :=_1 \bar{\lambda}k.\, \cdots \,\mathbf{in}\, E'[xK']$$

What happens next is that the $deref_1$ rule fires:

$$\nu x.\, x :=_1 \bar{\lambda}k.\, \cdots \,\mathbf{in}\, E'[xK'] \longmapsto \nu x.\, E'[(\bar{\lambda}k.\, \cdots)K']$$

Since this $deref_1$-reduction is what activates the computation of x, we expect that $\nu x.\, E'[(\bar{\lambda}k.\, \cdots)K']$ should be the shape of the CPS term corresponding to an active **let**. But this means we must be able to translate *evaluation contexts* as well as terms. For the uniform CPS transform, we were content to have a lemma (Proposition 10) that merely *asserted* that there was some K that served to translate a source evaluation context. Now that contexts are a crucial part of the CPS transform, we need to make the translation explicit. Hence the context part of the CPS transform, which we write using round brackets, like $\mathcal{C}_a(\!|E|\!)$. Its definition is easily derived from that of $\mathcal{C}_a$, so that we have:

Proposition 33. $\mathcal{C}_a[\![E[M]]\!] \equiv \mathcal{C}_a(\!|E|\!)[\mathcal{C}_a[\![M]]\!]$

Proof. An easy induction on E. □

Again we define an administrative congruence relation. We will want to be able to rearrange bindings when it is safe to do so; accordingly, we adopt a *lift* rule, a generalization of the *lift* rule from call-by-need:

$$
\begin{array}{rll}
E[E'[C]] \longrightarrow_{ad} E'[E[C]] & & lift \\
(\bar{\lambda}(x^+).\, M)(V^+) \longrightarrow_{ad} M\{V^+/x^+\} & & \beta_{ad}
\end{array}
$$

$$
\begin{aligned}
\mathcal{C}_a[\![x]\!] &\triangleq \bar{\lambda}k.\, xk \\
\mathcal{C}_a[\![\lambda x.\, M]\!] &\triangleq \bar{\lambda}k.\, k(\bar{\lambda}(x,k').\, \mathcal{C}_a[\![M]\!]k') \\
\mathcal{C}_a[\![MN]\!] &\triangleq \bar{\lambda}k.\, \mathcal{C}_a[\![M]\!](\lambda v.\, \nu x. \\
&\qquad x :=_1 \bar{\lambda}k'.\, \mathcal{C}_a[\![N]\!](\lambda v.\, x := \lambda k''.\, k''v \;\mathbf{in}\; k'v) \\
&\qquad \mathbf{in}\; v(x,k)) \\
\mathcal{C}_a[\![\mathbf{let_s}\; x = M \;\mathbf{in}\; N]\!] & \\
&\triangleq \bar{\lambda}k.\, \nu x.\, x :=_1 \bar{\lambda}k'.\, \mathcal{C}_a[\![M]\!](\lambda v.\, x := \bar{\lambda}k''.\, k''v \;\mathbf{in}\; k'v) \\
&\qquad \mathbf{in}\; \mathcal{C}_a[\![N]\!]k \\
\mathcal{C}_a[\![\mathbf{let_a}\; x = M \;\mathbf{in}\; E[x]]\!] & \\
&\triangleq \bar{\lambda}k.\, \nu x.\, \mathcal{C}_a(\!|E|\!)[\bar{\lambda}k'.\, \mathcal{C}_a[\![M]\!](\lambda v.\, x := \bar{\lambda}k''.\, k''v \;\mathbf{in}\; k'v)]k \\
\mathcal{C}_a[\![\mathbf{let_v}\; x = V \;\mathbf{in}\; N]\!] & \\
&\triangleq \bar{\lambda}k.\, \nu x.\, x := \mathcal{C}_a[\![V]\!] \;\mathbf{in}\; \mathcal{C}_a[\![N]\!]k
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_a(\!|[\,]|\!) &\triangleq [\,] \\
\mathcal{C}_a(\!|EN|\!) &\triangleq \bar{\lambda}k.\, \mathcal{C}_a(\!|E|\!)(\lambda v.\, \nu x. \\
&\qquad x :=_1 \bar{\lambda}k'.\, \mathcal{C}_a[\![N]\!](\lambda v.\, x := \bar{\lambda}k''.\, k''v \;\mathbf{in}\; k'v) \\
&\qquad \mathbf{in}\; v(x,k)) \\
\mathcal{C}_a(\!|\mathbf{let_s}\; x = M \;\mathbf{in}\; E|\!) & \\
&\triangleq \bar{\lambda}k.\, \nu x.\, x :=_1 \bar{\lambda}k'.\, \mathcal{C}_a[\![M]\!](\lambda v.\, x := \bar{\lambda}k''.\, k''v \;\mathbf{in}\; k'v) \\
&\qquad \mathbf{in}\; \mathcal{C}_a(\!|E|\!)k \\
\mathcal{C}_a(\!|\mathbf{let_a}\; x = E \;\mathbf{in}\; F[x]|\!) & \\
&\triangleq \bar{\lambda}k.\, \nu x.\, \mathcal{C}_a(\!|F|\!)[\bar{\lambda}k'.\, \mathcal{C}_a(\!|E|\!)(\lambda v.\, x := \bar{\lambda}k''.\, k''v \;\mathbf{in}\; k'v)]k \\
\mathcal{C}_a(\!|\mathbf{let_v}\; x = V \;\mathbf{in}\; E|\!) & \\
&\triangleq \bar{\lambda}k.\, \nu x.\, x := \mathcal{C}_a[\![V]\!] \;\mathbf{in}\; \mathcal{C}_a(\!|E|\!)k
\end{aligned}
$$

Figure 19. The call-by-need CPS transform on annotated terms and contexts, $\mathcal{C}_a$

As usual, the transitive closure of $\longrightarrow_{ad}$ is $\longrightarrow_{ad}^{+}$, and its reflexive and transitive closure is $\longrightarrow\!\!\!\!\rightarrow_{ad}$. Its reflexive, symmetric, and transitive closure, restricted[8] to safe terms, is $=_{ad}$.

We will also use $\longmapsto_{ad}$ to refer to an invocation of β_{ad} at the top level, which we call an *administrative standard reduction*. (A *lift*-reduction is never proper, as it is never necessary.) As before, an *administrative answer* is a term that cannot take an administrative standard reduction.

The confluence, standardization, and commutativity results we need are similar to before:

Proposition 34. *The relation* $=_{ad}$

1. *is* confluent*, so that if* $M =_{ad} M'$*, then there is some* N *such that* $M \longrightarrow\!\!\!\!\rightarrow_{ad} N$ *and* $M' \longrightarrow\!\!\!\!\rightarrow_{ad} N$*; and*
2. *has the* standardization *property, so that if* $M \longrightarrow\!\!\!\!\rightarrow_{ad} N$ *and* N *is an administrative answer, then there is an administrative answer* M' *such that* $M \longmapsto\!\!\!\!\rightarrow_{ad} M' \longrightarrow\!\!\!\!\rightarrow_{ad} N$.

[8] The restriction is necessary because safety is not closed backward, *i.e.,* there are unsafe terms that reduce to safe ones.

Proof.

1. Because *lift* and β_{ad} do not overlap and they are both left-linear, we can use modularity [4] to prove confluence from the confluence of each rule separately. The *lift* rule is symmetric and thus trivially confluent. The β_{ad} rule is simply the β rule restricted to a subcalculus, so it is also confluent.
2. Since *lift* and β_{ad} trivially commute (they do not even interact), we can perform the *lift* steps last, giving $M \longrightarrow\!\!\!\!\!\rightarrow_{\beta_{ad}} N' \longrightarrow\!\!\!\!\!\rightarrow_{\mathit{lift}} N$. Since $\longrightarrow\!\!\!\!\!\rightarrow_{\mathit{lift}}$ cannot create a proper standard reduction, N' must also be an administrative answer; hence standardization of β-reduction applies, completing the proof. □

Proposition 35.

1. If $M \longrightarrow\!\!\!\!\!\rightarrow_{ad} M' \longmapsto_{pr} N$ and M is an administrative answer, then there is N' with $M \longmapsto_{pr} N' \longrightarrow\!\!\!\!\!\rightarrow_{ad} N$.
2. If $M \longleftarrow\!\!\!\!\!\leftarrow_{ad} M' \longmapsto_{pr} N$, then there is N' with $M \longmapsto_{pr} N' \longleftarrow\!\!\!\!\!\leftarrow_{ad} N$.

Proof. The *lift* rule can neither create nor destroy *any* redexes, and internal reductions still cannot destroy a standard redex, so the proof is essentially the same as that of Proposition 8. □

Lemma 36. *If $M =_{ad} \longmapsto_{pr} N$, then $M \longmapsto^{+}_{pr^1} =_{ad} N$.*

Proof. The same as the proof of Lemma 9, this time using Propositions 34 and 35. □

For the uniform CPS transform, we characterized the action of translation on contexts as Proposition 10: $\mathcal{C}_a[\![E[M]]\!]K$ will reduce to $\mathcal{C}[\![M]\!]K'$, where K' is some continuation that represents E. This case will be more complex, however: In λ_{need}, an evaluation context contains both bindings *and* work to be done.[9] A continuation alone only captures the latter. Therefore, for call-by-need, we will need to translate the bindings as well. Our approach is to split context translation into a function $\mathcal{B}$ providing a CPS binding context and a function $\mathcal{K}$ providing a continuation (in which the bindings from $\mathcal{B}$ are in scope). Putting them together, we will be able to relate the original context transform $\mathcal{C}_a(\!|-|\!)$ to the split transform.

$$
\begin{aligned}
\mathcal{B}(\!|[\,]|\!) &\triangleq [] \\
\mathcal{B}(\!|EM|\!) &\triangleq \mathcal{B}(\!|E|\!) \\
\mathcal{B}(\!|\mathbf{let_s}\, x = M \,\mathbf{in}\, E|\!) &\triangleq \nu x.\, x :=_1 \bar{\lambda} k'.\, \mathcal{C}_a[\![M]\!](\lambda v.\, x := \bar{\lambda} k''.\, k'' v \,\mathbf{in}\, k' v) \,\mathbf{in}\, \mathcal{B}(\!|E|\!) \\
\mathcal{B}(\!|\mathbf{let_a}\, x = E \,\mathbf{in}\, F[x]|\!) &\triangleq \nu x.\, \mathcal{B}(\!|F|\!)[\mathcal{B}(\!|E|\!)] \\
\mathcal{B}(\!|\mathbf{let_v}\, x = V \,\mathbf{in}\, E|\!) &\triangleq \nu x.\, x := \mathcal{C}_a[\![V]\!] \,\mathbf{in}\, \mathcal{B}(\!|E|\!)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{K}(\!|[\,]|\!) : K &\triangleq K \\
\mathcal{K}(\!|EM|\!) : K &\triangleq \mathcal{K}(\!|E|\!) : \lambda v.\, \nu x.\, x :=_1 \bar{\lambda} k'.\, \mathcal{C}_a[\![M]\!](\lambda v.\, x := \bar{\lambda} k''.\, k'' v \,\mathbf{in}\, k' v) \,\mathbf{in}\, v(x, K) \\
\mathcal{K}(\!|\mathbf{let_s}\, x = M \,\mathbf{in}\, E|\!) : K &\triangleq \mathcal{K}(\!|E|\!) : K \\
\mathcal{K}(\!|\mathbf{let_a}\, x = E \,\mathbf{in}\, F[x]|\!) : K &\triangleq \mathcal{K}(\!|E|\!) : \lambda v.\, x := \bar{\lambda} k''.\, k'' v \,\mathbf{in}\, (\mathcal{K}(\!|F|\!) : K) v \\
\mathcal{K}(\!|\mathbf{let_v}\, x = V \,\mathbf{in}\, E|\!) : K &\triangleq \mathcal{K}(\!|E|\!) : K
\end{aligned}
$$

The following proposition shows the relationship between $\mathcal{C}_a$, $\mathcal{B}$, and $\mathcal{K}$. Note that, like Proposition 10, it demonstrates the action of administrative reductions: In this case, they serve to bring the continuation inward while preserving the bindings in the context. We use T here to denote a value that is, in particular, a thunk of the form $\lambda k.\, M$.

[9] Danvy and Zerny [14] make a similar observation about call-by-need evaluation contexts; they then derive a call-by-need language that separates the two parts.

Proposition 37. $\mathcal{C}_a(\!|E|\!)[T]K \equiv \mathcal{B}(\!|E|\!)[T(\mathcal{K}(\!|E|\!) : K)]$

Proof. By induction on E. First, some shorthand will clarify:

$$\mathbf{let}_\ell\, x = V \,\mathbf{in}\, P \triangleq \nu x.\, x :=_1 \bar{\lambda} k.\, V(\lambda v.\, x := \bar{\lambda} k.\, kv \,\mathbf{in}\, kv) \,\mathbf{in}\, P$$

Now:

- For $E \equiv [\,]$:

$$\begin{aligned}
\mathcal{C}_a(\!|E|\!)[T]K &\equiv [\,][T]K \\
&\triangleq TK \\
&\triangleq [\,][TK] \\
&\triangleq \mathcal{B}(\!|[\,]|\!)[T(\mathcal{K}(\!|[\,]|\!) : K)] \\
&\equiv \mathcal{B}(\!|E|\!)[T(\mathcal{K}(\!|E|\!) : K)]
\end{aligned}$$

- For $E \equiv E'M$:

$$\begin{aligned}
\mathcal{C}_a(\!|E|\!)[T]K &\equiv \mathcal{C}_a(\!|E'M|\!)[T]K \\
&\triangleq (\bar{\lambda} k.\, \mathcal{C}_a(\!|E'|\!)(\lambda v.\, \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!] \,\mathbf{in}\, v(x,k)))[T]K \\
&\equiv (\bar{\lambda} k.\, \mathcal{C}_a(\!|E'[T]|\!)(\lambda v.\, \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!] \,\mathbf{in}\, v(x,k)))K \\
&\longmapsto_{ad} \mathcal{C}_a(\!|E'[T]|\!)(\lambda v.\, \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!] \,\mathbf{in}\, v(x,K)) \\
&\longmapsto\!\!\!\!\rightarrow_{ad} \mathcal{B}(\!|E'|\!)[T(\mathcal{K}(\!|E'|\!) : \lambda v.\, \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!] \,\mathbf{in}\, v(x,K))] && \text{(by I.H.)} \\
&\triangleq \mathcal{B}(\!|E'M|\!)[T(\mathcal{K}(\!|E'M|\!) : K)] \\
&\equiv \mathcal{B}(\!|E|\!)[T(\mathcal{K}(\!|E|\!) : K)]
\end{aligned}$$

- For $E \equiv \mathbf{let_s}\, x = M \,\mathbf{in}\, E'$:

$$\begin{aligned}
\mathcal{C}_a(\!|E|\!)[T]K &\equiv \mathcal{C}_a(\!|\mathbf{let_s}\, x = M \,\mathbf{in}\, E'|\!)[T]K \\
&\triangleq (\bar{\lambda} k.\, \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!] \,\mathbf{in}\, \mathcal{C}_a(\!|E'|\!)k)[T]K \\
&\equiv (\bar{\lambda} k.\, \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!] \,\mathbf{in}\, \mathcal{C}_a(\!|E'[T]|\!)k)K \\
&\longmapsto_{ad} \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!] \,\mathbf{in}\, \mathcal{C}_a(\!|E'[T]|\!)K \\
&\longmapsto\!\!\!\!\rightarrow_{ad} \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!] \,\mathbf{in}\, \mathcal{B}(\!|E'|\!)[T(\mathcal{K}(\!|E'|\!) : K)k] && \text{(by I.H.)} \\
&\triangleq \mathcal{B}(\!|\mathbf{let_s}\, x = M \,\mathbf{in}\, E'|\!)[T(\mathcal{K}(\!|\mathbf{let_s}\, x = M \,\mathbf{in}\, E'|\!) : K)] \\
&\equiv \mathcal{B}(\!|E|\!)[T(\mathcal{K}(\!|E|\!) : K)]
\end{aligned}$$

- For $E \equiv \mathbf{let_a}\, x = E' \,\mathbf{in}\, F[x]$:

$$\begin{aligned}
\mathcal{C}_a(\!|E|\!)[T]K &\equiv \mathcal{C}_a(\!|\mathbf{let_a}\, x = E' \,\mathbf{in}\, F[x]|\!)[T]K \\
&\equiv (\bar{\lambda} k.\, \nu x.\, \mathcal{C}_a(\!|F|\!)[\bar{\lambda} k'.\, \mathcal{C}_a(\!|E'|\!)(\lambda v.\, x := \bar{\lambda} k''.\, k''v \,\mathbf{in}\, k'v)]k)[T]K \\
&\equiv (\bar{\lambda} k.\, \nu x.\, \mathcal{C}_a(\!|F|\!)[\bar{\lambda} k'.\, \mathcal{C}_a(\!|E'|\!)[T](\lambda v.\, x := \bar{\lambda} k''.\, k''v \,\mathbf{in}\, k'v)]k)K \\
&\longmapsto_{ad} \nu x.\, \mathcal{C}_a(\!|F|\!)[\bar{\lambda} k'.\, \mathcal{C}_a(\!|E'|\!)[T](\lambda v.\, x := \bar{\lambda} k''.\, k''v \,\mathbf{in}\, k'v)]K \\
&\longmapsto\!\!\!\!\rightarrow_{ad} \nu x.\, \mathcal{B}(\!|F|\!)[(\bar{\lambda} k'.\, \mathcal{C}_a(\!|E'|\!)[T](\lambda v.\, x := \bar{\lambda} k''.\, k''v \,\mathbf{in}\, (\mathcal{K}(\!|F|\!) : k')v))K] && \text{(by I.H.)} \\
&\longmapsto_{ad} \nu x.\, \mathcal{B}(\!|F|\!)[\mathcal{C}_a(\!|E'|\!)[T](\lambda v.\, x := \bar{\lambda} k''.\, k''v \,\mathbf{in}\, (\mathcal{K}(\!|F|\!) : K)v)] \\
&\longmapsto\!\!\!\!\rightarrow_{ad} \nu x.\, \mathcal{B}(\!|F|\!)[\mathcal{B}(\!|E'|\!)[T(\mathcal{K}(\!|E'|\!) : \lambda v.\, x := \bar{\lambda} k''.\, k''v \,\mathbf{in}\, (\mathcal{K}(\!|F|\!) : K)v))]] && \text{(by I.H.)} \\
&\triangleq \nu x.\, \mathcal{B}(\!|\mathbf{let_a}\, x = E' \,\mathbf{in}\, F[x]|\!)[T(\mathcal{K}(\!|\mathbf{let_a}\, x = E' \,\mathbf{in}\, F[x]|\!) : K)] \\
&\equiv \mathcal{B}(\!|E|\!)[T(\mathcal{K}(\!|E|\!) : K)]
\end{aligned}$$

- For $E \equiv \mathbf{let_v}\ x = V\ \mathbf{in}\ E'$:

$$\begin{aligned}
\mathcal{C}_a(\!|E|\!)[T]K &\equiv \mathcal{C}_a(\!|\mathbf{let_v}\ x = V\ \mathbf{in}\ E'|\!)[T]K \\
&\triangleq (\bar{\lambda}k.\, \nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!]\ \mathbf{in}\ \mathcal{C}_a(\!|E'|\!)k)[T]K \\
&\equiv (\bar{\lambda}k.\, \nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!]\ \mathbf{in}\ \mathcal{C}_a(\!|E'[T]|\!)k)K \\
&\longmapsto_{ad} \nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!]\ \mathbf{in}\ \mathcal{C}_a(\!|E'[T]|\!)K \\
&\longmapsto\!\!\!\!\rightarrow_{ad} \nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!]\ \mathbf{in}\ \mathcal{B}(\!|E'|\!)[T(\mathcal{K}(\!|E'|\!) : K)] \qquad \text{(by I.H.)} \\
&\triangleq \mathcal{B}(\!|\mathbf{let_v}\ x = V\ \mathbf{in}\ E'|\!)[T(\mathcal{K}(\!|\mathbf{let_v}\ x = V\ \mathbf{in}\ E'|\!) : K)] \\
&\equiv \mathcal{B}(\!|E|\!)[T(\mathcal{K}(\!|E|\!) : K)]
\end{aligned}$$

□

Now we can show how a term in a context is transformed:

Corollary 38. $\mathcal{C}_a[\![E[M]]\!]K \longmapsto\!\!\!\!\rightarrow_{ad} \mathcal{B}(\!|E|\!)[\mathcal{C}_a[\![M]\!](\mathcal{K}(\!|E|\!) : K)]$.

Proof. From Propositions 33 and 37. □

Not all λ_{need} contexts affect the continuation. In particular, binding contexts never alter the continuation at all. This is not surprising, since binding contexts are precisely those that do not affect the flow of control.

Proposition 39. *For any λ^a_{need} binding context B and $\lambda^{:=_1}_{cps}$ continuation K, $\mathcal{K}(\!|B|\!) : K \equiv K$.*

Proof. Trivial induction on B. Note that the clauses of $\mathcal{K}$ that do nothing but recurse are precisely those for the parts of a binding context. □

Corollary 40. *For any λ^a_{need} binding context B, λ^a_{need} term M, and $\lambda^{:=_1}_{cps}$ continuation K, $\mathcal{C}_a[\![B[M]]\!]K \longmapsto\!\!\!\!\rightarrow_{ad} \mathcal{B}(\!|B|\!)[\mathcal{C}_a[\![M]\!]K]$.*

Proof. Immediate from Corollary 38 and Proposition 39. □

Now we turn to correctness. Our forward simulation follows the same construction as that of the uniform CPS transform:

Definition 41. *For a λ^a_{need} term M and $\lambda^{:=_1}_{cps}$ term P, let $M \sim P$ when $\mathcal{C}_a[\![M]\!]\,\mathbf{ret} =_{ad} P$.*

To prove that $\sim$ is a forward simulation, we go through the diagrams in Eq. (6) once again. The first, third, and fourth are exactly as before:

Lemma 42. $M \sim \mathcal{C}_a[\![M]\!]\,\mathbf{ret}$. □

Lemma 43. *If $M \sim P$ and $M \downarrow$, then $P \Downarrow$.* □

Lemma 44. *If $M \sim P$ and $M \not\downarrow$, then $P \not\Downarrow$.* □

The second diagram can be proved using much the same strategy as for Lemma 20, but the more complex source and target languages make the calculations heavier. To review, the steps we take to prove the simulation are:

1. Show that, if $M \longmapsto N$ by a reduction in the empty context, then we have $\mathcal{C}_a[\![M]\!]K \longmapsto\!\!\!\!\rightarrow\!\!=_{ad} \mathcal{C}_a[\![N]\!]K$.
2. Allow the reduction to occur in any evaluation context, not only at the top of the term.
3. Let the CPS term be *any* $P =_{ad} \mathcal{C}_a[\![M]\!]$.

Now we begin with step one immediately:

Lemma 45. *If $M \longmapsto N$ by a reduction in the empty context, then $\mathcal{C}_a[\![M]\!]K \longmapsto^+_{pr^1}\!\!=_{ad} \mathcal{C}_a[\![N]\!]K$.*

Proof. This time we have four reduction rules, so there are four cases.

- For a β-reduction:

$$
\begin{aligned}
&\mathcal{C}_a[\![B[\lambda x.\, M]N]\!]K \\
&\quad \triangleq (\bar{\lambda}k.\, (\mathcal{C}_a[\![B[\lambda x.\, M]]\!](\lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\ \mathbf{in}\ v(x,k)))K \\
&\quad \longmapsto_{ad} \mathcal{C}_a[\![B[\lambda x.\, M]]\!](\lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\ \mathbf{in}\ v(x,K)) \\
&\quad \longmapsto\!\!\!\!\rightarrow_{ad} \mathcal{B}(\!|B|\!)[\mathcal{C}_a[\![\lambda x.\, M]]\!](\lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\ \mathbf{in}\ v(x,K))] && \text{(by Corollary 40)} \\
&\quad \triangleq \mathcal{B}(\!|B|\!)[(\bar{\lambda}k.\, k(\bar{\lambda}(x,k').\, \mathcal{C}_a[\![M]\!]k'))(\lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\ \mathbf{in}\ v(x,K))] \\
&\quad \longmapsto_{ad} \mathcal{B}(\!|B|\!)[(\lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\ \mathbf{in}\ v(x,K))(\bar{\lambda}(x,k').\, \mathcal{C}_a[\![M]\!]k')] \\
&\quad \longmapsto_{pr} \mathcal{B}(\!|B|\!)[\mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\ \mathbf{in}\ (\bar{\lambda}(x,k').\, \mathcal{C}_a[\![M]\!]k')(x,K)] \\
&\quad \longmapsto_{ad} \mathcal{B}(\!|B|\!)[\mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\ \mathbf{in}\ \mathcal{C}_a[\![M]\!]K] \\
&\quad \longleftarrow\!\!\!\!\mapsto_{ad} \mathcal{B}(\!|B|\!)[(\bar{\lambda}k.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\ \mathbf{in}\ \mathcal{C}_a[\![M]\!]k)K] \\
&\quad \twoheadleftarrow\!\!\!\!\mapsto_{ad} \mathcal{C}_a(\!|B|\!)[\mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\ \mathbf{in}\ \mathcal{C}_a[\![M]\!]]K && \text{(by Corollary 40)} \\
&\quad \longleftarrow\!\!\!\!\mapsto_{ad} (\bar{\lambda}k.\, \mathcal{C}_a(\!|B|\!)[\mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\ \mathbf{in}\ \mathcal{C}_a[\![M]\!]]k)K \\
&\quad \triangleq \mathcal{C}_a[\![B[\mathbf{let}_{\mathbf{s}}\, x = N\ \mathbf{in}\ M]]\!]K
\end{aligned}
$$

- For an act-reduction:

$$
\begin{aligned}
&\mathcal{C}_a[\![\mathbf{let}_{\mathbf{s}}\, x = M\ \mathbf{in}\ E[x]]\!]K \\
&\quad \triangleq (\bar{\lambda}k.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!]\ \mathbf{in}\ \mathcal{C}_a[\![E[x]]\!])K \\
&\quad \longmapsto_{ad} \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!]\ \mathbf{in}\ \mathcal{C}_a[\![E[x]]\!]K \\
&\quad \longmapsto\!\!\!\!\rightarrow_{ad} \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!]\ \mathbf{in}\ \mathcal{B}(\!|E|\!)[\mathcal{C}_a[\![x]\!](\mathcal{K}(\!|E|\!) : K)] && \text{(by Corollary 38)} \\
&\quad \triangleq \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!]\ \mathbf{in}\ \mathcal{B}(\!|E|\!)[(\bar{\lambda}k.\, xk)(\mathcal{K}(\!|E|\!) : K)] \\
&\quad \longmapsto_{ad} \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!]\ \mathbf{in}\ \mathcal{B}(\!|E|\!)[x(\mathcal{K}(\!|E|\!) : K)] \\
&\quad \triangleq \nu x.\, x :=_1 \bar{\lambda}k'.\, \mathcal{C}_a[\![M]\!](\lambda v.\, x := \bar{\lambda}k''.\, k''v\ \mathbf{in}\ k'v)\ \mathbf{in}\ \mathcal{B}(\!|E|\!)[x(\mathcal{K}(\!|E|\!) : K)] \\
&\quad \longmapsto_{pr} \nu x.\, \mathcal{B}(\!|E|\!)[(\bar{\lambda}k'.\, \mathcal{C}_a[\![M]\!](\lambda v.\, x := \bar{\lambda}k''.\, k''v\ \mathbf{in}\ k'v))(\mathcal{K}(\!|E|\!) : K)] \\
&\quad \twoheadleftarrow\!\!\!\!\mapsto_{ad} (\bar{\lambda}k.\, \nu x.\, \mathcal{C}_a(\!|E|\!)[\bar{\lambda}k'.\, \mathcal{C}_a[\![M]\!](\lambda v.\, x := \bar{\lambda}k''.\, k''v\ \mathbf{in}\ k'v)])K && \text{(by Corollary 38)} \\
&\quad \triangleq \mathcal{C}_a[\![\mathbf{let}_{\mathbf{a}}\, x = M\ \mathbf{in}\ E[x]]\!]K
\end{aligned}
$$

- For a $deact$-reduction:

$$
\begin{aligned}
&\mathcal{C}_a[\![\mathbf{let}_{\mathbf{a}}\, x = B[V]\ \mathbf{in}\ E[x]]\!]K \\
&\quad \triangleq (\bar{\lambda}k.\, \nu x.\, \mathcal{C}_a(\!|E|\!)[\bar{\lambda}k'.\, \mathcal{C}_a[\![B[V]]\!](\lambda v.\, x := \bar{\lambda}k''.\, k''v\ \mathbf{in}\ k'v)])K \\
&\quad \longmapsto_{ad} \nu x.\, \mathcal{C}_a(\!|E|\!)[\bar{\lambda}k'.\, \mathcal{C}_a[\![B[V]]\!](\lambda v.\, x := \bar{\lambda}k''.\, k''v\ \mathbf{in}\ k'v)]K \\
&\quad \longmapsto\!\!\!\!\rightarrow_{ad} \nu x.\, \mathcal{B}(\!|E|\!)[(\bar{\lambda}k'.\, \mathcal{C}_a[\![B[V]]\!](\lambda v.\, x := \bar{\lambda}k''.\, k''v\ \mathbf{in}\ k'v))(\mathcal{K}(\!|E|\!) : K)] && \text{(by Proposition 37)} \\
&\quad \longmapsto_{ad} \nu x.\, \mathcal{B}(\!|E|\!)[\mathcal{C}_a[\![B[V]]\!](\lambda v.\, x := \bar{\lambda}k''.\, k''v\ \mathbf{in}\ (\mathcal{K}(\!|E|\!) : K)v)] \\
&\quad \longmapsto\!\!\!\!\rightarrow_{ad} \nu x.\, \mathcal{B}(\!|E|\!)[\mathcal{B}(\!|B|\!)[\mathcal{C}_a[\![V]\!](\lambda v.\, x := \bar{\lambda}k''.\, k''v\ \mathbf{in}\ (\mathcal{K}(\!|E|\!) : K)v)]] && \text{(by Corollary 40)}
\end{aligned}
$$

Letting $\bar{\lambda}k.\, kW \triangleq \mathcal{C}_a[\![V]\!]$:

$$
\begin{aligned}
&\triangleq \nu x.\, \mathcal{B}(\!|E|\!)[\mathcal{B}(\!|B|\!)[(\bar{\lambda}k.\, kW)(\lambda v.\, x \coloneqq \bar{\lambda}k''.\, k''v \textbf{ in } (\mathcal{K}(\!|E|\!) : K)v)]] \\
&\longmapsto_{ad} \nu x.\, \mathcal{B}(\!|E|\!)[\mathcal{B}(\!|B|\!)[(\lambda v.\, x \coloneqq \bar{\lambda}k''.\, k''v \textbf{ in } (\mathcal{K}(\!|E|\!) : K)v)W]] \\
&\longmapsto_{pr} \nu x.\, \mathcal{B}(\!|E|\!)[\mathcal{B}(\!|B|\!)[x \coloneqq \bar{\lambda}k''.\, k''W \textbf{ in } (\mathcal{K}(\!|E|\!) : K)W]] \\
&\longrightarrow_{ad} \mathcal{B}(\!|B|\!)[\nu x.\, \mathcal{B}(\!|E|\!)[x \coloneqq \bar{\lambda}k''.\, k''W \textbf{ in } (\mathcal{K}(\!|E|\!) : K)W]] \\
&\longrightarrow_{ad} \mathcal{B}(\!|B|\!)[\nu x.\, x \coloneqq \bar{\lambda}k''.\, k''W \textbf{ in } \mathcal{B}(\!|E|\!)[(\mathcal{K}(\!|E|\!) : K)W]] \\
&\longleftarrow_{ad} \mathcal{B}(\!|B|\!)[\nu x.\, x \coloneqq \bar{\lambda}k''.\, k''W \textbf{ in } \mathcal{B}(\!|E|\!)[(\bar{\lambda}k.\, kW)(\mathcal{K}(\!|E|\!) : K)]] \\
&\triangleq \mathcal{B}(\!|B|\!)[\nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!] \textbf{ in } \mathcal{B}(\!|E|\!)[\mathcal{C}_a[\![V]\!](\mathcal{K}(\!|E|\!) : K)]] \\
&\twoheadleftarrow_{ad} \mathcal{B}(\!|B|\!)[\nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!] \textbf{ in } \mathcal{C}_a[\![E[V]]\!]K] && \text{(by Corollary 38)} \\
&\twoheadleftarrow_{ad} \mathcal{C}_a[\![B[\mathbf{let_v}\, x = V \textbf{ in } E[V]]]\!]K && \text{(by Corollary 40)}
\end{aligned}
$$

- For a *deref*-reduction:

$$
\begin{aligned}
&\mathcal{C}_a[\![\mathbf{let_v}\, x = V \textbf{ in } E[x]]\!]K \\
&\quad \triangleq \bar{\lambda}k.\, (\nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!] \textbf{ in } \mathcal{C}_a[\![E[x]]\!]k)K \\
&\quad \longmapsto_{ad} \nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!] \textbf{ in } \mathcal{C}_a[\![E[x]]\!]K \\
&\quad \longmapsto\!\!\!\!\rightarrow_{ad} \nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!] \textbf{ in } \mathcal{B}(\!|E|\!)[x(\mathcal{K}(\!|E|\!) : K)] && \text{(by Corollary 38)} \\
&\quad \longmapsto_{pr} \nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!] \textbf{ in } \mathcal{B}(\!|E|\!)[\mathcal{C}_a[\![V]\!](\mathcal{K}(\!|E|\!) : K)] \\
&\quad \twoheadleftarrow_{ad} \nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!] \textbf{ in } \mathcal{C}_a[\![E[V]]\!]K && \text{(by Corollary 38)} \\
&\quad \longleftarrow_{ad} (\bar{\lambda}k.\, \nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!] \textbf{ in } \mathcal{C}_a[\![E[V]]\!]k)K \\
&\quad \triangleq \mathcal{C}_a[\![\mathbf{let_v}\, x = V \textbf{ in } E[V]]\!]K
\end{aligned}
$$

□

Now we proceed to the second step, which allows the reduction to take place in a larger context:

Proposition 46. *If $M \longmapsto N$, then $\mathcal{C}_a[\![M]\!]K \longmapsto^{+}_{pr^1} =_{ad} \mathcal{C}_a[\![N]\!]K$.*

Proof. As before, by definition of $\longmapsto$, we have $M \equiv E[M']$ and $N \equiv E[N']$, where $M' \longmapsto N'$ at top level.

$$
\begin{aligned}
\mathcal{C}_a[\![M]\!]K &\equiv \mathcal{C}_a[\![E[M']]\!]K \\
&\longmapsto\!\!\!\!\rightarrow_{ad} \mathcal{B}(\!|E|\!)[\mathcal{C}_a[\![M']\!](\mathcal{K}(\!|E|\!) : K)] && \text{(by Corollary 38)} \\
&\longmapsto^{+}_{pr^1} =_{ad} \mathcal{C}_a(\!|E|\!)[\mathcal{C}_a[\![N']\!](\mathcal{K}(\!|E|\!) : K)] && \text{(by Lemma 45)} \\
&\twoheadleftarrow_{ad} \mathcal{C}_a[\![E[N']]\!]K && \text{(by Corollary 38)} \\
&\equiv \mathcal{C}_a[\![N]\!]K
\end{aligned}
$$

□

Finally, we use Lemma 36 to generalize, completing the third step:

Lemma 47. *If $M \longmapsto N$ and $M \sim P$, then there is Q such that $P \longmapsto^{+} Q$ and $N \sim Q$.*

Proof. We have $M \sim P$, so $P =_{ad} \mathcal{C}_a[\![M]\!]\,\mathbf{ret}$. $M \longmapsto N$, so $\mathcal{C}_a[\![M]\!]\,\mathbf{ret} \longmapsto^{+}_{pr^1} =_{ad} \mathcal{C}_a[\![N]\!]\,\mathbf{ret}$ by Proposition 46. So $P =_{ad} \longmapsto^{+}_{pr^1} =_{ad} \mathcal{C}_a[\![N]\!]\,\mathbf{ret}$. Since $\longmapsto^{+}_{pr^1}$ is short for $\longmapsto\!\!\!\!\rightarrow_{ad} \longmapsto_{pr} \longmapsto\!\!\!\!\rightarrow_{ad}$, we have $P =_{ad} \longmapsto_{pr} =_{ad} \mathcal{C}_a[\![N]\!]\,\mathbf{ret}$. Then, by Lemma 36, we have $P \longmapsto^{+} Q =_{ad} \mathcal{C}_a[\![N]\!]\,\mathbf{ret}$ for some Q. □

And now we have the correctness result for the CPS transform on annotated terms:

Lemma 48. *For any λ^{a}_{need}-term M:*

1. $M \Downarrow$ iff $\mathcal{C}_a[\![M]\!]\,\mathbf{ret} \Downarrow$.

2. $M \Uparrow$ *iff* $\mathcal{C}_a[\![M]\!]\,\mathbf{ret} \Uparrow$.
3. $M \not\Uparrow$ *iff* $\mathcal{C}_a[\![M]\!]\,\mathbf{ret} \not\Uparrow$.

Proof. By Lemmas 42 to 44 and 47, using determinacy. □

3.3.4 From λ_{need} to $\lambda^{:=_1}_{cps}$

From these pieces, we assemble the correctness of the call-by-need CPS transform:

Theorem 49. *For any λ_{need}-term M:*

1. $M \Downarrow$ *iff* $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Downarrow$.
2. $M \Uparrow$ *iff* $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Uparrow$.
3. $M \not\Uparrow$ *iff* $\mathcal{C}[\![M]\!]\,\mathbf{ret} \not\Uparrow$.

Proof. Immediate from Propositions 31 and 32 and Lemma 48. □

3.3.5 Naming and π-encoding

Since we have introduced some of the naming mechanism into the "unnamed" CPS language $\lambda^{:=_1}_{cps}$, the simulation proof for the naming transform needs to be more subtle. The readback function $\mathcal{N}^{-1}$ we defined before undoes *all* assignments; now that the source CPS calculus is only *mostly* unnamed, this is too blunt an instrument. Instead, we will use a relation that can *selectively* eliminate sharing.

First, we need to restrict our terms so that we can reason about what variables may be assigned to.

Definition 50. *A $\lambda^{:=_1}_{cps}$ term or value is* localized *if no variable appearing on the left of an assignment subterm is bound by a λ.*[10]

For example, the value $\lambda(x, y).\, x := \lambda k.\, ky\ \mathbf{in}\ kx$ is not allowed, since x is bound by the λ. Since free and ν-bound variables are not subject to substitution by β-reduction, localized terms are closed under reduction. In particular, we can say with certainty which variables in a localized term may ever be assigned to, no matter the context—only those assigned to by *subterms* of the term.

Proposition 51. *If K is localized, then $\mathcal{C}_a[\![M]\!]K$ is localized. In particular, $\mathcal{C}_a[\![M]\!]\,\mathbf{ret}$ is localized.*

Proof. Easy induction on M. □

Now, keeping in mind that $\lambda^{:=_1}_{cps,vn}$ is a subset of $\lambda^{:=_1}_{cps}$:

Definition 52. *The relation $\prec$ is the restriction to $\lambda^{:=_1}_{cps} \times \lambda^{:=_1}_{cps,vn}$ of the reflexive, transitive, and congruent closure of the following rules on localized terms:*

$$M\{\lambda y^+.\, N/x\} \prec \nu x.\, x := \lambda y^+.\, N\ \mathbf{in}\ M \qquad \textit{(if } x \textit{ not assigned in } M\textit{)}$$
$$M\{\lambda_1 y^+.\, N/x\} \prec \nu x.\, x :=_1 \lambda y^+.\, N\ \mathbf{in}\ M \qquad \textit{(if } x \textit{ is affine and not assigned in } M \textit{ or } N\textit{)}$$
$$M \prec \nu x.\, M \qquad \textit{(if } x \textit{ not free in } M\textit{)}$$

This suffices to prove the correctness of the naming transform:

Lemma 53. *For any $\lambda^{:=_1}_{cps}$-term M and variable k:*

1. $M \Downarrow$ *iff* $\mathcal{N}[\![M]\!] \Downarrow$.
2. $M \Uparrow$ *iff* $\mathcal{N}[\![M]\!] \Uparrow$.
3. $M \not\Uparrow$ *iff* $\mathcal{N}[\![M]\!] \not\Uparrow$.

[10] We borrow the term *localized* from the π-calculus literature. The localized π-calculus is a subcalculus that forbids processes from listening on channels they have received from other processes. As in the π-calculus, this restriction can be made finer using a type system.

Proof. As before, we show that $M \prec \mathcal{N}[\![M]\!]$ and that $\prec$ is a backward bisimulation that preserves outcomes in the backward direction. The result follows as always from these observations and determinism.

Initial Condition That $M \prec \mathcal{N}[\![M]\!]$ can be found by an easy induction, as $\prec$ simply undoes the manipulations performed by $\mathcal{N}$.

Simulation We need that $M \prec P$ and $P \longmapsto Q$ imply that there is N with $M \longmapsto\!\!\!\!\!\rightarrow N$ and $N \prec Q$.

If $P \longmapsto Q$, then we have a reduction by β, $deref$, or $deref_1$. β-reductions are unchanged by $\prec$. For $deref$, we have $P \equiv E[f \coloneqq \lambda(x^+).\, P' \textbf{ in } E'[f(y^+)]$ and $Q \equiv E[f \coloneqq \lambda(x^+).\, P' \textbf{ in } E'[(\lambda(x^+).\, P')(y^+)]]$. Now consider how M might relate to P: Applications of $\prec$ inside E, M, or E' would not interfere with the $deref$-reduction (we can apply the rules in Q instead). If f was substituted into the body of P, then we can simply take $M = N$. Otherwise, we can contract M to find N; either way, $N \prec Q$.

The case for $deref_1$ (*i.e.,* for an ephemeral assignment rather than a permanent one) is similar, only to get $N \prec Q$ at the end, we need to apply the third rule of $\prec$ to collect the ν as garbage.

Outcomes As with Corollary 26, $\downarrow$ and $\not\downarrow$ are invariant under $\prec$, and every $deref$ (or $deref_1$) creates a standard β-redex, so answers, stuck states, and divergence are preserved. □

The augmentation of $\mathcal{P}$ for ephemeral assignment is easy to prove correct:

Lemma 54. *For any $\lambda^{:=_1}_{cps,vn}$-term M, $M \Downarrow$ iff $\mathcal{P}[\![M]\!] \Downarrow_{\overline{\mathbf{ret}}}$*

Proof. To the proof of Lemma 27, we need only add consideration of ephemeral assignment, which corresponds just as strongly as permanent assignment. □

Finally we have our proof of the correctness of the call-by-need π-encoding:

Theorem 55. *For any λ_{need}-term M, M reduces to an answer iff $\mathcal{E}[\![M]\!]_{\mathbf{ret}} \Downarrow_{\overline{\mathbf{ret}}}$.*

Proof. Immediate from Theorem 49 and Lemmas 53 and 54, since $\mathcal{E}[\![M]\!]_k \triangleq \mathcal{P}[\![\mathcal{N}[\![\mathcal{C}[\![M]\!]k]\!]]\!]$. □

3.4 Abstract Machine

Just as we did with the uniform CPS transform, we can derive an abstract machine from the call-by-need transform. First, we represent ephemeral assignment in store-passing style: A thunk assigned ephemerally should be erased from store when it is accessed. We use the symbol $\perp$ to denote such a "missing" value; the store will bind $\perp$ to a variable that has been allocated (by a ν) but currently has no value.

Using this representation, the functional correspondence gives us the abstract machine in Fig. 20. There are different machine states for examining a term, a thunk, a continuation, or a closure, and a halt state returning the final value and store. The store is a map from *locations* to thunks, and the environment maps local variables to locations in the store.

Notably, up to a few transition compressions, this abstract machine is the same as one derived by Ager, Danvy, and Midtgaard [2][11], except that when a suspended computation is retrieved from the environment, it is removed. In this way, it resembles the original call-by-need abstract machine by Sestoft [29]. Without a **letrec** form in the source, however, this difference in behavior cannot be observed, since the symbol binding a computation cannot appear free in the term being computed.

It is also quite similar to one derived recently by Danvy and Zerny [14], which they call the lazy Krivine machine. The mechanisms are superficially different in a few ways. For them, a thunk is simply an unevaluated term, whereas we remember whether the thunk has been evaluated before (and a few bookkeeping details). However, this is merely a different choice for the division of responsibility: We hand control to the thunk, and then the thunk determines whether to set up an update or simply return a value. The lazy Krivine machine instead inspects the thunk when it is retrieved: If it is a value, it is returned immediately, and otherwise it is evaluated. Hence our thunks are *tagged* and theirs are not. The tags are largely an artifact of the connection to

[11] Specifically, it resembles the first variant mentioned in section 3 of [2].

$$
\begin{array}{rrl}
\text{Locations:} & & \ell, \ldots \\
\text{Terms:} & M, N & ::= x \mid \lambda x.\, M \mid MN \\
\text{Thunks:} & t & ::= \mathsf{Susp}\langle M, \ell, \rho\rangle \mid \mathsf{Memo}\langle f\rangle \mid \bot \\
\text{Continuations:} & k & ::= \mathsf{Ret}\langle\rangle \mid \mathsf{Apply}\langle M, \rho, k\rangle \mid \mathsf{Update}\langle \ell, k\rangle \\
\text{Closures:} & f & ::= \mathsf{Clos}\langle x, M, \rho\rangle \\
\text{Stores:} & \sigma & ::= \epsilon \mid \sigma[\ell = t] \\
\text{Environments:} & \rho & ::= \epsilon \mid \rho[x = \ell] \\
\text{States:} & S & ::= \langle M, \sigma, \rho, k\rangle_M \mid \langle t, \sigma, k\rangle_T \mid \\
 & & \quad \langle k, f, \sigma\rangle_K \mid \langle f, \ell, \sigma, k\rangle_F \mid \\
 & & \quad \langle f, \sigma\rangle_H
\end{array}
$$

$$
M \longmapsto \langle M, \epsilon, \epsilon, \mathsf{Ret}\langle\rangle\rangle_M
$$

$$
\begin{array}{rl}
\langle x, \sigma, \rho, k\rangle_M & \longmapsto \langle t, \sigma, k\rangle_T \\
\text{where } \rho(x) \equiv \ell \text{ and } \sigma(\ell) \equiv t & \\
\langle \lambda x.\, M, \sigma, \rho, k\rangle_M & \longmapsto \langle k, \mathsf{Clos}\langle x, M, \rho\rangle, \sigma\rangle_K \\
\langle MN, \sigma, \rho, k\rangle_M & \longmapsto \langle M, \sigma, \rho, \mathsf{Apply}\langle N, \rho, k\rangle\rangle_M
\end{array}
$$

$$
\begin{array}{rl}
\langle \mathsf{Susp}\langle M, \ell, \rho\rangle, \sigma, k\rangle_T & \longmapsto \langle M, \sigma[\ell = \bot], \rho, \mathsf{Update}\langle \ell, k\rangle\rangle_M \\
\langle \mathsf{Memo}\langle f\rangle, \sigma, k\rangle_T & \longmapsto \langle k, f, \sigma\rangle_K
\end{array}
$$

$$
\begin{array}{rl}
\langle \mathsf{Ret}\langle\rangle, f, \sigma\rangle_K & \longmapsto \langle f, \sigma\rangle_H \\
\langle \mathsf{Apply}\langle M, \rho, k\rangle, f, \sigma\rangle_K & \longmapsto \langle f, \ell, \sigma[\ell = \mathsf{Susp}\langle M, \ell, \rho\rangle], k\rangle_F \\
\text{where } \ell \notin \sigma & \\
\langle \mathsf{Update}\langle \ell, k\rangle, f, \sigma\rangle_K & \longmapsto \langle k, f, \sigma[\ell = \mathsf{Memo}\langle f\rangle]\rangle_K
\end{array}
$$

$$
\langle \mathsf{Clos}\langle x, M, \rho\rangle, \ell, \sigma, k\rangle_F \longmapsto \langle M, \sigma, \rho[x = \ell], k\rangle_M
$$

Figure 20. The abstract machine derived from $\mathcal{C}$

the π-calculus translation—whether an argument has been evaluated is evident from the structure of the process representing it. Another difference is that the lazy Krivine machine lacks an environment, relying entirely on the store, but again this is superficial.

4. Related Work

Brock and Ostheimer [8] gave a proof of correctness for the call-by-need π-calculus encoding, but without connecting it to CPS. Okasaki et al. [22] gave a CPS transform for call-by-need by targeting a calculus with mutable storage.

The first proof that the λ-calculus could be encoded in the π-calculus was due to Milner [20], who gave encodings (quite different from those considered here) for call-by-value and call-by-name. The connection to CPS was developed by Sangiorgi [27], who showed that CBN and CBV CPS transforms could be translated into the *higher-order π-calculus*, which can then be compiled to the usual first-order π-calculus. The compilation is analogous to the naming transforms considered here. Our approach, where names and sharing are handled outside of the π-calculus, was introduced by Amadio [3].

Accattoli [1] pursues a different approach to relating the CBN and CBV λ-calculi to π: Where we take the CPS language and alter it to match the π-calculus, he refines the source calculus so that its reductions line up with those in π. Namely, he reformulates the λ-calculus in terms of explicit substitutions and *linear weak head reduction*. Terms in this *linear substitution calculus* can then readily be compared with their encodings in the π-calculus, as substitution and sharing work in a similar way.

The advantages of using distance rules to eliminate administrative work have been explored by Chang and Felleisen [9], who reformulate the call-by-need λ-calculus using a single β-rule; and by Accattoli [1], who expresses the π-calculus using distance rules.

Finally, while the exact form of the annotated call-by-need λ-calculus considered here is new, such "preprocessed" forms have been considered elsewhere and are similar. In particular, both Brock and Ostheimer [8] and Danvy and Zerny [14] include versions of what we call $\mathbf{let_a}$, the "active $\mathbf{let}$."

5. Conclusion and Future Work

The connection between CPS transforms, the π-calculus, and graph reduction has been considered only in the call-by-value and call-by-name worlds. We have shown that only a modest extension to the target CPS calculus is required in order to put call-by-need on an equal footing. Hopefully, we can find further uses for the constructive update calculus; for instance, given the closeness to π-calculus channel operations, it is possible that other π-calculus encodings can be reinterpreted as CPS transforms as well. In another direction, since many type systems for the π-calculus have been proposed [28], it seems worth exploring whether we can use the techniques outlined here to consider a typed CPS transform and a typed encoding into the π-calculus.

References

[1] B. Accattoli. Evaluating functions as processes. In *TERMGRAPH*, pages 41–55, 2013.

[2] M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004.

[3] R. Amadio. A decompilation of the pi-calculus and its application to termination. Technical Report UMR CNRS 7126, Université Paris Diderot, 2011.

[4] C. Appel, V. van Oostrom, and J. G. Simonsen. Higher-order (non-)modularity. In *RTA*, pages 17–32, 2010.

[5] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Functional Programming*, 7(3):265–301, 1997.

[6] Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Information and Computation*, 139, 1996.

[7] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *POPL*, pages 233–246. ACM, 1995.

[8] S. Brock and G. Ostheimer. Process semantics of graph reduction. In *CONCUR*, pages 471–485, 1995.

[9] S. Chang and M. Felleisen. The call-by-need lambda calculus, revisited. In *Programming Languages and Systems*, pages 128–147. Springer, 2012.

[10] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 2:33, 346–366, 1932.

[11] O. Danvy and A. Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, Copenhagen, Denmark, 1989.

[12] O. Danvy and A. Filinski. Representing control: A study of the cps transformation. *Mathematical structures in computer science*, 2(04):361–391, 1992.

[13] O. Danvy and L. R. Nielsen. A first-order one-pass cps transformation. *Theoretical Computer Science*, 308(1): 239–257, 2003.

[14] O. Danvy and I. Zerny. A synthetic operational account of call-by-need evaluation. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, pages 97–108. ACM, 2013.

[15] M. Felleisen and D. Friedman. Control operators, the secd machine, and the lambda calculus. In *Formal Descriptions of Programming Concepts*, pages 193–219, 1986.

[16] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the λ-calculus. Technical Report 197, Indiana University, Computer Science Department, 1986.

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

[18] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[19] A. Meyer and S. Cosmadakis. Semantical Paradigms: Notes for an Invited Lecture. Technical report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, July 1988.

[20] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(02):119–141, 1992.

[21] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Automata, Languages and Programming*, pages 685–695. Springer, 1992.

[22] C. Okasaki, P. Lee, and D. Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7 (1):57–81, 1994.

[23] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[24] J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

[25] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.

[26] A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):916–941, 1997.

[27] D. Sangiorgi. From λ to π; or, rediscovering continuations. *Mathematical Structures in Computer Science*, 9(4):367–401, July 1999. ISSN 1469-8072. URL `http://journals.cambridge.org/article_S0960129599002881`.

[28] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge Univ. Press, 2003.

[29] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(03):231–264, 1997.