Continuations, Processes, and Sharing

Paul Downen Luke Maurer
Zena M. Ariola
University of Oregon
{pdownen,maurerl,ariola}@cs.uoregon.edu

Daniele Varacca
Université Paris Diderot
varacca@pps.univ-paris-diderot.fr

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—process models; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—lambda calculus and related systems

Keywords continuation-passing style, transforms, effects, call-by-need, processes, π-calculus, bisimulation

Abstract

Continuation-passing style (CPS) transforms have long been important tools in the study of programming. They have been shown to correspond to abstract machines and, when combined with a naming transform that expresses shared values, they enjoy a direct correspondence with encodings into process calculi such as the π-calculus. We present our notion of correctness and discuss the sufficient conditions that guarantee the correctness of transforms. We then consider the call-by-value, call-by-name and call-by-need evaluation strategies for the λ-calculus and present their CPS transforms, abstract machines, π-encodings, and proofs of correctness. Our analysis covers a uniform CPS transform, which differentiates the three evaluation strategies only by the treatment of function calls. This leads to a new CPS transform for call-by-need requiring a less expressive form of side effect, which we call *constructive update*.

1. Introduction

For formally specifying the semantics of a programming language, there are a variety of different tools—from operational semantics to natural semantics, abstract machines to continuation-passing style (CPS) transforms, and many others. On the one hand, no one tool is objectively the best. Each semantic artifact has its own advantages, and so we can use each one to bring out different intuition and explore separate aspects of computation. On the other hand, if we refer to multiple semantics of a language, we need to be sure that they all agree. To that end, Danvy et al. [2] give a technique that not only demonstrates that two semantic artifacts correspond, but provides a mechanism for systematically interderiving one from the other. This lets us generate a variety of semantics from one specification with confidence that they all give the same interpretation.

In that vein, we can also use a process calculus such as the π-calculus [16, 20] as a basis for understanding the semantics of a language by encoding programs into processes [15]. In the same way CPS transforms target the λ-calculus, π-encodings elaborate the meaning of a program by hardwiring implementation details, such as the evaluation strategy, into the syntax of a process. However, certain computational phenomena which are more difficult to express by a CPS transform come out naturally in the π-calculus. For example, the π-encoding of memoization in a call-by-need language [20], which requires a change of state, is no more difficult to express than basic call-by-value or call-by-name evaluation.

Here, we aim to incorporate the π-calculus and π-encodings into the Danvy et al. [2] inter-derivation of semantic artifacts. We build on Sangiorgi's derivation [19] of π-encodings from CPS transforms by enriching the basic CPS language with a new effect called *constructive update*, and show how it reflects constructions that occur naturally in the π-calculus. Constructive update is a weaker effect than mutable state and more precisely describes memoization in call-by-need languages. For example, even though we don't consider recursive bindings, a particular implementation technique for recursion (*i.e.,* "black holes" in GHC) appears naturally in the abstract machine from the encoding into constructive update. In the end, we present π-encodings for the full CPS languages, with and without constructive update. These can be used to systematically derive a π-encoding from *any* CPS transform expressed in the language, simply by composition—no *ad hoc* reasoning is required.

In order to establish the correctness of the various transforms, we introduce a general proof methodology based on bisimulation. We integrate techniques used by Leroy [14] and Danvy and Zerny [10], giving us a robust and general framework for simplifying the different correctness proofs. We use this methodology to establish the correctness of the π-encoding of the CPS target language, as well as a less studied *uniform* [20] CPS transform, which distinguishes call-by-value from call-by-name evaluation by varying only the treatment of function application.

We start, in Section 2, with the semantics of the call-by-name and call-by-value λ-calculi. We present a uniform CPS transform for both, from which we derive the corresponding uniform abstract machine. We then outline a general methodology in Section 3 for establishing that a transform preserves the behavior of programs, using it in Section 4 to show the correctness of the uniform transform. Next, we demonstrate in Section 5 how π-encodings of the call-by-value and call-by-name λ-calculi can be systematically derived from a naming transform and a syntactic embedding of the CPS language, which are both shown to be correct. In Section 6, we consider call-by-need evaluation and present a novel call-by-need CPS transform in terms of constructive update. We demonstrate the correctness of this new transform in Section 7, and in Section 8 we derive the call-by-need π-encoding by translating constructive update to a form of

PPDP '14, September 8–10, 2014, Canterbury, UK.

ACM 978-1-4503-2947-7/14/09…$15.00.
http://dx.doi.org/10.1145/2643135.2643155

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

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

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

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

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

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

process that did not appear in the call-by-value and call-by-name encodings.

2. Call-by-Name and -Value

The λ-calculus is a simple yet powerful model of computation. It consists of only three parts: *functions*, *variables*, and *applications*:

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

As the λ-calculus can be seen as a foundation of both strict and lazy languages, a λ-term can be evaluated according to either a *call-by-name* (CBN) or a *call-by-value* (CBV) strategy. These two distinct semantics are captured by the two reduction rules, β_n and β_v (see Figs. 1 and 2) and by an *evaluation context*, which concisely specifies where in a term evaluation may take place [11]. Notice how β_v restricts β_n by only allowing a value to be substituted. CBV evaluation contexts allow computation to take place in the argument of a function application once the function has already been reduced to a value. The effect of these two alterations changes the order of evaluation: where CBN begins evaluating a function body before the argument to the function, CBV evaluates the argument first.

Notations

Throughout the paper, we will make use of the following notation: we write $M \longrightarrow N$ if N follows by a reduction of some subterm of M, and write $M \longmapsto N$, read "M standard reduces to N," if M decomposes as $E[M']$ for some E and M', $M' \longrightarrow N'$ by a reduction at the top level of M', and $N \triangleq E[N']$. Note that $\longmapsto$ can be seen as a restriction on $\longrightarrow$: the former only reduces the "standard" redex in a term (which must occur inside an evaluation context), whereas the latter can reduce *any* redex in a term (which may occur inside an arbitrary context). We write $\longrightarrow\!\!\!\!\rightarrow$ for the reflexive and transitive closure of $\longrightarrow$, denoting zero or more steps; $\longrightarrow^+$ for the transitive closure, denoting at least one step; and $=$ for the reflexive, *symmetric*, and transitive closure, denoting zero or more steps in either direction. The relations $\longmapsto\!\!\!\!\rightarrow$ and $\longmapsto^+$ follow from $\longmapsto$ similarly.

Finally, we introduce some notation for the possible outcomes of computation, which is to say, the *observations* we can make by running a program. For each calculus, we have a subset of terms called *answers*—that is, computed results. In the basic λ-calculus setting, an answer is simply a λ-abstraction. Then we write:

- $M \downarrow$ if M is an answer;
- $M \Downarrow$ if $M \longmapsto\!\!\!\!\rightarrow N$ and N is an answer;
- $M \not\downarrow$ if M is *stuck*, that is, M is *not* an answer and it cannot reduce any further;
- $M \not\Downarrow$ if $M \longmapsto\!\!\!\!\rightarrow N$ and N is stuck; or

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

Figure 3. A uniform CPS transform

- $M \Uparrow$ if M takes infinitely many evaluation steps.

A Uniform CPS Transform

As an alternative to specifying the semantics of a language in terms of rewrite rules for programs, we can give a function that "compiles," or transforms, programs into some lower-level form. The advantage is that analyzing lower-level terms 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 called; hence it plays a similar role to the *call stack* used in most computer architectures. Since their evaluation contexts differ, we can elucidate the difference between CBN and CBV languages by translating each into 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, given an application of the form MN, once M has evaluated to a function v, the continuation in the CBN transform invokes v immediately, passing the unevaluated CPS term $\lambda k'.\, \mathcal{C}[\![N]\!]k'$ as the argument. Evaluating a variable is done by invoking it with a continuation, so each evaluation of a variable within the body of a function will evaluate the argument. The CBV transform evaluates M the same way, but its continuation does not use v immediately; instead, it first evaluates N to a value w. Now, the argument to v is a function that simply passes w to its continuation, and so each evaluation of the argument immediately returns the precomputed value w.

The CBN portion of the uniform CPS transform corresponds to Plotkin's CBN CPS transform [18] up to currying and η-expansion. The CBV fragment, however, differs fundamentally from Plotkin's CBV transform:[1] it doesn't follow the conventional β_v rule. In the CBV λ-calculus, a variable is considered a value, and hence the term $(\lambda x.\, \lambda y.\, y)z$ reduces to $\lambda y.\, y$. However, a typical implementation would attempt to evaluate z and raise an "unbound variable" error. Accordingly, the CBV portion of the uniform CPS transform produces a term that becomes stuck on z rather than reducing to a value.

Since the CPS transform and the CBV λ-calculus do not agree on stuck states involving free variables, the uniform CPS transform does not implement the calculus given in Fig. 2. Rather, it implements a calculus that further restricts values to only include λ-abstractions.

[1] Instead, the CBV fragment is a "callee-save" version of Reynold's CPS transform (see section 4.2 of [12]).

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

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

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

$$\begin{array}{rl} \text{Terms:} & M, N ::= V(W^+) \\ \text{Values:} & V, W ::= x \mid \lambda(x^+).\, M \end{array}$$

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

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

The restriction impacts the β_v rule to apply only when the argument is a λ-abstraction, and limits evaluation contexts to exclude contexts like xE. From now on, when we speak of the call-by-value calculus, we will refer to the version in Fig. 4. In most cases, the change 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 it is a value or not.

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 values for arguments; we denote this $V(W^+)$ (standing for $V(W)$ and $V(W, W')$, we will omit parentheses when there is one argument). Likewise, $M\{V^+/x^+\}$ denotes the simultaneous substitution of one or two values for the same number of variables. A value is a variable or a λ-abstraction. Note that the body of an abstraction must again be a CPS term—that is, an application. In this CPS calculus, all function calls are tail calls, *ad infinitum*.

Uniform Abstract Machine

We can derive other artifacts from the uniform CPS transform. In particular, through the functional correspondence [2], we obtain a uniform abstract machine for both CBN and CBV evaluation (see Fig. 6), where we begin execution of M with the initial state: $\langle M, \mathsf{Ret}\langle\rangle, \epsilon\rangle_M$. The CBN fragment of the machine is essentially the same as the well-known Krivine machine for CBN evaluation [13]. On closed terms, the CBV fragment behaves similarly to the CEK machine [11] (without control operators). Unlike most environment-based abstract machines, ours does not exclude open terms, and thus its behavior can 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.

3. Preservation of Observations

In order to relate transforms like the uniform CPS transform to reduction in their source language, we consider what we mean when a transform is "correct." First, we expect it to preserve termination:

Criterion 1. $M \Downarrow$ *iff* $\mathcal{T}[\![M]\!] \Downarrow$.

In the following we will refer to the left-to-right implication as the forward direction and its converse as backward. In order to prove Criterion 1, we would want to proceed by induction on the evaluation steps and establish an *invariant*: something that is true at the beginning of evaluation and remains true after each step. The

$$\begin{array}{rl} \text{Terms:} & M, N ::= x \mid \lambda x.\, M \mid MN \\ \text{Conts.:} & k ::= \mathsf{Ret}\langle\rangle \mid \mathsf{AppN}\langle M, k, \rho\rangle \mid \\ & \quad \mathsf{AppV1}\langle M, k, \rho\rangle \mid \mathsf{AppV2}\langle \lambda x.\, M, k, \rho\rangle \\ \text{Envs.:} & \rho ::= \epsilon \mid \rho[x = \mathsf{Clos}\langle M, \rho\rangle] \\ \text{States:} & S ::= \langle M, k, \rho\rangle_M \mid \langle k, \lambda x.\, M, \rho\rangle_K \mid \\ & \quad \langle \lambda x.\, M, \rho\rangle_H \end{array}$$

$$\begin{array}{rl} \langle x, k, \rho\rangle_M & \longmapsto \langle M, k, \rho'\rangle_M \\ \text{where } \rho(x) & \equiv \mathsf{Clos}\langle M, \rho'\rangle \\ \langle \lambda x.\, M, k, \rho\rangle_M & \longmapsto \langle k, \lambda x.\, M, \rho\rangle_K \\ \langle MN, k, \rho\rangle_M & \longmapsto \langle M, k', \rho\rangle_M \\ \text{where } k' & \triangleq \begin{cases} \mathsf{AppN}\langle N, k, \rho\rangle & \text{CBN} \\ \mathsf{AppV1}\langle N, k, \rho\rangle & \text{CBV} \end{cases} \end{array}$$

$$\begin{array}{rl} \langle \mathsf{Ret}\langle\rangle, \lambda x.\, M, \rho\rangle_K & \longmapsto \langle \lambda x.\, M, \rho\rangle_H \\ \langle \mathsf{AppN}\langle N, k, \rho'\rangle, \lambda x.\, M, \rho\rangle_K & \longmapsto \langle M, k, \rho''\rangle_M \\ \text{where } \rho'' & \triangleq \rho[x = \mathsf{Clos}\langle N, \rho'\rangle] \\ \langle \mathsf{AppV1}\langle N, k, \rho'\rangle, \lambda x.\, M, \rho\rangle_K & \longmapsto \langle N, k', \rho'\rangle_M \\ \text{where } k' & \triangleq \mathsf{AppV2}\langle \lambda x.\, M, k, \rho\rangle \\ \langle \mathsf{AppV2}\langle \lambda y.\, N, k, \rho'\rangle, \lambda x.\, M, \rho\rangle_K & \longmapsto \langle N, k, \rho''\rangle_M \\ \text{where } \rho'' & \triangleq \rho'[y = \mathsf{Clos}\langle \lambda x.\, M, \rho\rangle] \end{array}$$

Figure 6. Uniform abstract machine for CBN and CBV

simplest such invariant is:

$$\begin{array}{ccc} M & \longmapsto\!\!\!\!\!\!\longrightarrow & N \\ \sim & & \sim \\ \mathcal{T}[\![M]\!] & \longmapsto\!\!\!\!\!\!\twoheadrightarrow & \mathcal{T}[\![N]\!] \end{array} \tag{1}$$

In words, whenever M reduces to N in one step, $\mathcal{T}[\![M]\!]$ reduces to $\mathcal{T}[\![N]\!]$ in many steps, where the dashed arrows indicate the existence of such transformations. The transformed program can take multiple steps because in general a transformation may introduce *administrative redexes* into a term. These are intermediate computations that do not correspond to actual reductions in the source language. (Non-administrative redexes are called *proper*.) Unfortunately, there is a serious issue with (1). As noted by Plotkin [18], with CPS transforms administrative reductions do not line up in this way. Because $\mathcal{T}[\![N]\!]$ introduces administrative redexes of its own, the true situation is this:

$$\begin{array}{ccccc} M & \longmapsto\!\!\!\!\!\!\!\!\longrightarrow & & & N \\ \sim & & & & \sim \\ \mathcal{T}[\![M]\!] & \longmapsto\!\!\!\!\!\!\twoheadrightarrow & P & \twoheadleftarrow\!\!\!\!\!\!- & \mathcal{T}[\![N]\!] \end{array} \tag{2}$$

Plotkin's solution was to derive a new transform that eliminated these initial administrative redexes, thus regaining (1). The problem with this and similar solutions [8, 9] 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, higher-order, or require multiple passes. Danvy and Nielsen [9] show how to avoid such issues

for developing compact, administrative-reduction free transforms. However, this approach does not generalize to handle other cases when the source and target programs get out of synch, as with different amounts of sharing caused by copying references to values in Section 5.

Instead of changing the transform, we can further loosen the invariant using the *bisimulation* technique, which requires only that we find *some* suitable relation to act as the invariant, allowing us to still relate program states that have moved away from the transform. Bisimulation gives us a single proof methodology that readily applies to all the transforms studied here, solving issues caused by administrative reductions and sharing. Given a relation $\sim$ between source and target terms, we can prove Criterion 1 so long as the following hold:

$$\begin{array}{ccccc} M & & M \longmapsto N & & M \vdash\!\!\dashrightarrow\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow N \\ \sim & & \sim & & \sim \\ \mathcal{T}[\![M]\!] & & P \vdash\!\!\dashrightarrow\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow Q & & P \longmapsto Q \end{array} \qquad \begin{array}{ccc} M\downarrow & & M \vdash\!\!\dashrightarrow\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow N\downarrow \\ \sim & & \sim \\ P \vdash\!\!\dashrightarrow\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow Q\downarrow & & Q\downarrow \end{array} \tag{3}$$

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.

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

Criterion 2. *$M \not\Downarrow$ iff $\mathcal{T}[\![M]\!] \not\Downarrow$.*

To prove Criterion 2, we require two more properties of the simulation $\sim$, in addition to those in (3):

$$\begin{array}{ccc} M\not\downarrow & & M \vdash\!\!\dashrightarrow\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow N\not\downarrow \\ \sim & & \sim \\ P \vdash\!\!\dashrightarrow\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow Q\not\downarrow & & P\not\downarrow \end{array} \tag{4}$$

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

The final observation we should preserve is divergence:

Criterion 3. *$M \Uparrow$ iff $\mathcal{T}[\![M]\!] \Uparrow$.*

However, because evaluation is deterministic in our calculi, we can get Criterion 3 "for free" from Criteria 1 and 2: if one term diverges, it can neither reduce to an answer nor get stuck, and hence the other term can only diverge.

We can simplify the proof methodology thanks to Leroy's observation [14] that if the source language is deterministic, the forward direction is sufficient, so long as we ensure that no infinite sequence of evaluation steps in the source maps to a finite sequence in the target, as this might relate a divergent term with an answer. It suffices to restrict the second diagram of (3) so that each source evaluation step maps to at least one evaluation step in the target. However, this is too restrictive due to the presence of administrative steps in the source. Borrowing ideas from Danvy and Zerny [10], we require that each proper step in the source (written $\longmapsto_{pr}$) map to at least one step in the target, whereas an administrative step in the source (written $\longmapsto_{ad}$) may map to zero steps (*i.e.,* equality). We

$$\begin{array}{ccccc} M & & M \xmapsto{pr} N & & M \xmapsto{ad} N \\ \sim & & \sim & & \sim \\ \mathcal{T}[\![M]\!] & & P \vdash\!\!\dashrightarrow^{+} Q & & P \vdash\!\!\dashrightarrow\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow Q \end{array} \qquad \begin{array}{ccc} M\downarrow & & M\not\downarrow \\ \sim & & \sim \\ P \vdash\!\!\dashrightarrow\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow Q\downarrow & & P \vdash\!\!\dashrightarrow\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow Q\not\downarrow \end{array}$$

Figure 7. Sufficient conditions for correctness of $\mathcal{T}$

have, however, the additional requirement that administrative steps must be strongly normalizing.

Theorem 4. *Given the sufficient conditions of Fig. 7, for any source term M:*

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

Proof. The forward direction follows directly from induction on the reduction sequence. For the backward direction of 1, 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 the reduction sequence must contain an infinite number of proper steps (because administrative reduction by itself is strongly normalizing), so by the second diagram it must hold that $\mathcal{T}[\![M]\!]$ also diverges and by determinacy it cannot reduce to an answer. Similarly, if M gets stuck, $\mathcal{T}[\![M]\!]$ must get stuck, and hence cannot reduce to an answer by determinacy. The reasoning for the backward directions of 2 and 3 is similar. □

We will use the following general procedure for establishing Theorem 4 from the sufficient conditions. First, we have to define what we mean by the transform, answers, stuck terms, and the simulation relation. Second, we need to specify which reduction steps in the source are administrative, if there are any, and show that they are strongly normalizing. Third, we need to establish the sufficient conditions in Fig. 7: that a source term is similar to its transformed term; that they remain similar after each step in the source; and that the relation preserves end states so that the observable outcomes correspond as well.

This procedure employs *forward* reasoning, in the sense that we begin with source reductions and give corresponding target reductions. It is sometimes easier to reason *backwards* instead. For instance, the target language of a transform $\mathcal{T}$ may express additional information which is difficult to recover in the source. To reason backward, we can instead prove correctness of an inverse transform $\mathcal{T}^{-1}$. Then we can use the fact that $\mathcal{T}^{-1}\langle\!\langle \mathcal{T}[\![M]\!] \rangle\!\rangle \triangleq M$ to derive the correctness of $\mathcal{T}$ from the correctness of $\mathcal{T}^{-1}$.

4. Correctness of the Uniform CPS Transform

In order to define the transform into the CPS calculus, let us consider what it means for a CPS program to be evaluated. A term $\mathcal{C}[\![M]\!]$ is an inert λ-abstraction; it must be applied to a continuation 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 consider a free variable called $\mathbf{ret}$ to be the initial continuation, and evaluate M in the CPS calculus as $\mathcal{C}[\![M]\!]\,\mathbf{ret}$. If we think of a term $\mathcal{C}[\![M]\!]$ as meaning "evaluate M and then," then $\mathcal{C}[\![M]\!]\,\mathbf{ret}$ reads

"evaluate M and then return." Intuitively, $\mathbf{ret}$ is analogous to the C function `exit`, which terminates execution and yields its argument as the result of the program. Thus, we let $\mathcal{T}[\![M]\!]$ be $\mathcal{C}[\![M]\!]\,\mathbf{ret}$.

An answer in the source CBV and CBN λ-calculi is a λ-abstraction value V. A corresponding term in the target CPS language has the form $\mathbf{ret}\,V$, so this is the form of a CPS answer. Additionally, a stuck term in the source calculi has the form of a free variable in an evaluation context, $E[x]$, whereas a stuck term in the target is one with a free variable other than $\mathbf{ret}$ in head position, xV.

Next, we want to define the simulation $\sim$ by relating terms in a way that ignores all administrative reductions. We can keep track of these reductions in the CPS term by marking certain λ-abstractions as administrative with the notation $\bar{\lambda}k.\,M$. Referring to Fig. 3, all λ-abstractions except the one representing a source λ-abstraction (that is, the $\lambda(x,k).\,M$) are considered administrative. Administrative reductions are then defined as the β-reductions of the marked λ-abstractions, as given by the *administrative β-rule*:

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

We can now define our simulation relation $\sim$ in a way that allows us to reason up to β_{ad} equality of the CPS transform:

Definition 5. *$M \sim P$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} =_{ad} P$.*

We are now left to show that the conditions in Fig. 7 hold. The initial condition is trivial, since $M \sim \mathcal{C}[\![M]\!]\,\mathbf{ret}$ by reflexivity. Demonstrating the inductive step (the second diagram of Fig. 7), however, is more challenging. First, observe that substitution may be pushed through the CPS transform, at the cost of reducing some non-standard administrative reductions.

Proposition 6. $\mathcal{C}[\![M]\!]\{\mathcal{C}[\![N]\!]/x\} \longrightarrow\!\!\!\!\!\rightarrow_{ad} \mathcal{C}[\![M\{N/x\}]\!]$.

Using Proposition 6, it is straightforward to show by calculation that, if $M \longrightarrow N$ at the top level, then

$$\mathcal{C}[\![M]\!]\,\mathbf{ret} \longmapsto\!\!\!\!\!\rightarrow_{ad} \longmapsto_{pr} =_{ad} \mathcal{C}[\![N]\!]\,\mathbf{ret} \tag{5}$$

However, this statement needs to be generalized in two ways: (1) in the source, the reduction $M \longrightarrow N$ may occur in any evaluation context, and (2) in the target, we may begin with any $P =_{ad} \mathcal{C}[\![M]\!]\,\mathbf{ret}$.

For the first generalization, we must determine how evaluation contexts in the source are treated by the CPS transform. As it happens, administrative reductions of the CPS-transformed term convert a source-level evaluation context into a continuation, thereby lifting the next redex to the top of the term. In other words, we begin with $\mathcal{C}[\![E[M]]\!]\,\mathbf{ret}$ and build toward $\mathcal{C}[\![M]\!]K'$, where K' is a function that encodes the context E.

Proposition 7. *For each evaluation context E in λ_n or λ_v, there is a continuation K' such that for every term M, $\mathcal{C}[\![E[M]]\!]\,\mathbf{ret} \longmapsto\!\!\!\!\!\rightarrow_{ad} \mathcal{C}[\![M]\!]K'$.*

This allows us to lift reduction into any evaluation context, so that if $E[M] \longmapsto E[N]$ by any standard reduction then (5) still holds by (i) applying Proposition 7 to convert the evaluation context E into a function K', (ii) applying (5) as usual with K' substituted for $\mathbf{ret}$, and (iii) applying Proposition 7 again to relate the result with $\mathcal{C}[\![E[N]]\!]\,\mathbf{ret}$.

The second generalization of (5) means that we lose the benefit of starting directly with the transformation of our source term. Rather, we may be given a CPS term that has gotten out of sync by some number of administrative steps. Our goal is then to show that this extra administration at the start does not get in the way of evaluation. In particular, we can show that any administrative equality can be deferred past a proper standard reduction while still retaining a non-trivial reduction sequence.

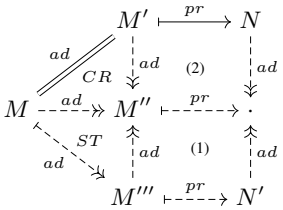


Figure 8. A summary of the proof of Lemma 8.

Lemma 8. *For λ_{cps} terms M, M', N, if $M =_{ad} M' \longmapsto_{pr} N$, then there is N' such that $M \longmapsto^+ N' =_{ad} N$.*

Proof. To begin, we note some facts about the administrative subset of the CPS language, which consists only of the β_{ad} rule. First, note that the next standard reduction that M' takes is a proper step so it is an *administrative answer*, defined as the terms that cannot take a standard step by the β_{ad} rule (that is, there is no $M' \longmapsto_{ad} N$). For example, both $\mathbf{ret}\,V$ and $(\lambda(x,k).\,M)(V,V')$ are administrative answers because they cannot take an administrative step, but only the second one can take a proper step. Second, administrative reduction ($M \longrightarrow N$ by β_{ad} on any subterm of M) in λ_{cps} is confluent and enjoys 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 $M \longmapsto\!\!\!\!\!\rightarrow_{ad} M' \longrightarrow\!\!\!\!\!\rightarrow_{ad} N$. Third, non-standard administrative reductions commute with proper standard reductions, so they do not create or destroy the top β-redex:

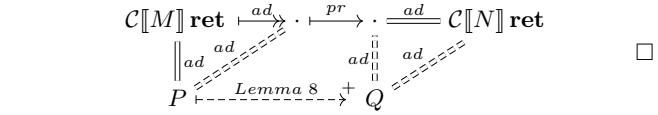
1. If $M \longrightarrow\!\!\!\!\!\rightarrow_{ad} M' \longmapsto_{pr} N$ and M is an administrative answer, then there is N' with $M \longmapsto_{pr} N' \longrightarrow\!\!\!\!\!\rightarrow_{ad} N$.
2. If $M \longleftarrow\!\!\!\!\!\leftarrow_{ad} M' \longmapsto_{pr} N$, then there is N' with $M \longmapsto_{pr}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

Finally, we use confluence and standardization, along with the above properties, to establish commutation of administrative equality with proper reduction, as shown in Fig. 8. □

By using Lemma 8, we can strengthen (5) by postponing any preceding administrative equality to the very end, giving us the second diagram of Fig. 7.

Proposition 9. *If $M \longmapsto N$ and $M \sim P$ then there is a $N \sim Q$ such that $P \longmapsto^+ Q$.*

Proof. Beginning with (5) for $M \longmapsto N$, we have



□

The remaining diagrams establishing the preservation of observations follow from the fact that "answerness" and "stuckness" are preserved by administrative operations:

Proposition 10. *1. If $P =_{ad} P'$ and $P \downarrow$, then $P' \Downarrow$.*
2. If $P =_{ad} P'$ and $P \not\downarrow$, then $P' \not\Downarrow$.

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

$$\begin{array}{rl}\text{Processes:} & P, Q ::= \overline{x}\langle y^+\rangle \mid x(y^+).\, P \mid (P \mid Q) \mid\, !P \mid \boldsymbol{\nu} z\, P \\ \text{Eval. Cxts.:} & E ::= [\,] \mid (E \mid Q) \mid \boldsymbol{\nu} z\, E\end{array}$$

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

$$P \mid Q \equiv Q \mid P \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \quad !P \equiv P \mid !P$$
$$\boldsymbol{\nu} x\, \boldsymbol{\nu} y\, P \equiv \boldsymbol{\nu} y\, \boldsymbol{\nu} x\, P \quad (\boldsymbol{\nu} z\, P) \mid Q \equiv \boldsymbol{\nu} z\, (P \mid Q) \;\; z \notin FV(Q)$$
$$\boldsymbol{\nu} x\, P \equiv \boldsymbol{\nu} y\, P\{y/x\} \qquad x(y^+).\, P \equiv x(z^+).\, P\{z^+/y^+\}$$

Figure 9. A fragment of the (polyadic) π-calculus.

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

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

Figure 10. The value-named CPS λ-calculus, $\lambda_{cps,vn}$

channel and, in parallel, another is reading 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. 9. 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 they provide 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 $\equiv$, which expands upon the usual notion of α-equivalence. The rules are given in Fig. 9, and the $E[P]$ operation is considered up to structural congruence. 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.

Despite the radically different approaches to expressing computation, CPS transforms and π-encodings are interrelated. 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 send names, rather than processes, over channels. In other words, λ is higher-order, but π is first-order. We address this issue by introducing an environment-based CPS transform, which uses names to refer to values indirectly.

Take note that the translation into the π-calculus, both the naming step and the transformation of functions into processes, is defined generally for the entire CPS calculus. The translation from the CPS λ-calculus to the π-calculus is not tied to the uniform CPS transform discussed in Section 2 in particular, but can be used to convert *any* CPS transform expressed in λ_{cps} into a π-encoding. For example, we can derive a π-encoding for both Plotkin's [18] original call-by-value CPS transform and Danvy and Nielsen's [9] first-order one-pass CPS transform by composing them with the naming and π-calculus translations.[2]

Environment-Based CPS Transform

So far, we have expressed argument passing by *substitution*: each β-reduction substitutes the arguments for the free occurrences of the corresponding variables. Effectively the term is rewritten so the argument is copied in place of 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, 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 it can be captured by extending the syntax with a **let** construct [5]: a bound name identifies a node in a graph. However, we present an alternative syntax which explicitly expresses the dynamic allocation of fresh names. We write $\nu x.\, x \coloneqq \lambda(x^+).\, M \;\mathbf{in}\; N$ 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. 10. Each term is now an application inside some number of bindings, which effectively serve as the environment. Each function application must have only variables as the arguments.

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. It is also important to observe that a ν specifies the static scope of the introduced name, so that it is subject to α-equivalence in the same way as a λ-abstraction. Furthermore, the *deref* rule carries the restriction that the the free variables of $\lambda(x^+).\, M$ are not captured by a ν in the binding context E, so that dereferencing a value does not unintentionally cause capture of free names.

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, referring to each λ-abstraction indirectly by a new variable.

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

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 naming transform may also be made compositional by unfolding the definition of $\mathcal{N}$ for every combination of named and unnamed arguments (splitting clauses like $\mathcal{N}[\![V(\lambda(x^+).\, M, W)]\!]$ into cases for W and then unfolding all non-compositional applications of $\mathcal{N}$ on the right-hand side). Fig. 11 shows the composition of the uniform CPS transform and the naming transform.

[2] Additionally, multiple-argument functions produced by the CPS transforms would need to be uncurried, as λ_{cps} does not allow for curried functions.

$$\mathcal{C}_{vn}[\![x]\!] \triangleq \lambda k.\, xk$$

$$\mathcal{C}_{vn}[\![\lambda x.\, M]\!] \triangleq \lambda k.\, \nu f.\, f \coloneqq \lambda(x, k').\, \mathcal{C}_{vn}[\![M]\!]k' \textbf{ in } kf$$

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

Figure 11. Uniform CPS transform in named form

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 the 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, and how it only duplicates the name for $(\lambda x.\, N)$:

$$\begin{aligned} \mathcal{N}[\![M]\!] &\triangleq \nu y.\, y \coloneqq \lambda x.\, \mathcal{N}[\![N]\!] \textbf{ in } (\lambda x.\, f(x,x))y \\ &\longmapsto \nu y.\, y \coloneqq \lambda x.\, \mathcal{N}[\![N]\!] \textbf{ in } f(y,y) \end{aligned}$$

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

$$\begin{aligned} &\mathcal{N}[\![f(\lambda z.\, N, \lambda z.\, N)]\!] \triangleq \\ &\quad \nu x.\, x \coloneqq \lambda z.\, \mathcal{N}[\![N]\!] \textbf{ in } \nu y.\, y \coloneqq \lambda z.\, \mathcal{N}[\![N]\!] \textbf{ in } f(x,y) \end{aligned}$$

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.

The simplest way to build a simulation that relates terms up to sharing is to remove all sharing from the terms and compare the unnamed forms. This suggests a transform that "flattens" a term's bound variables:

$$\mathcal{N}^{-1}\langle\!\langle V(x^+) \rangle\!\rangle \triangleq \mathcal{N}^{-1}\langle\!\langle V \rangle\!\rangle (x^+)$$

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

$$\mathcal{N}^{-1}\langle\!\langle f \rangle\!\rangle \triangleq f \qquad \mathcal{N}^{-1}\langle\!\langle \lambda(x^+).\, M \rangle\!\rangle \triangleq \lambda(x^+).\, \mathcal{N}^{-1}\langle\!\langle M \rangle\!\rangle$$

Note that $\mathcal{N}^{-1}\langle\!\langle \mathcal{N}[\![M]\!] \rangle\!\rangle$ is the same as M, so correctness of $\mathcal{N}^{-1}$ implies that $\mathcal{N}[\![M]\!]$ is also correct.

Now we then wish to relate a final state of the form $E[f(x^+)]$, where f is not bound by E, with a similar one $f(V^+)$ in the unnamed calculus. We say that such a state is an answer if f is $\mathbf{ret}$ and stuck otherwise. Our simulation relation then becomes just the unnaming transform.

Definition 11. *For a named CPS term P and an unnamed CPS term M, let $P \sim M$ when $M \equiv \mathcal{N}^{-1}\langle\!\langle P \rangle\!\rangle$ by α-equivalence.*

When we check that reduction in the named CPS calculus maps to reduction in the unnamed CPS calculus, we find that *deref* steps disappear since they are erased by the $\mathcal{N}^{-1}$ transform. Therefore, we must consider *deref* to be an administrative step, which is strongly normalizing since the next standard reduction is always a β step. With this definition of administrative reduction in the named calculus, the sufficient conditions from Fig. 7 follow immediately. Observe that answers and stuck states in $\lambda_{cps,vn}$ are preserved by the substitution performed by $\mathcal{N}^{-1}$. The remaining diagrams from Fig. 7 can be checked for the (proper) β and (administrative) $deref$ reductions of $\lambda_{cps,vn}$.

$$\mathcal{E}[\![x]\!]_k \triangleq \overline{x}\langle k \rangle$$

$$\mathcal{E}[\![\lambda x.\, M]\!]_k \triangleq \boldsymbol{\nu} f\, (\overline{k}\langle f \rangle \mid !f(x,k).\, \mathcal{E}[\![M]\!]_k)$$

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

Figure 12. The uniform π-encoding for CBN and CBV

Deriving a π-Encoding from the CPS Transform

We can now see that the *value-named* CPS language $\lambda_{cps,vn}$ is "first-order" in much the same way as the π-calculus. 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, transmitting the tuple (y^+), and then halts.
- Each binding $\nu x.\, x \coloneqq \lambda(y^+).\, N \textbf{ in } M$ becomes a process of the form $\boldsymbol{\nu} x\, (P \mid !x(y^+).\, Q)$, where P and Q correspond to M and N, respectively. This process allocates a fresh channel name x, then runs a process P in parallel with the process $!x(y^+).\, Q$. The latter acts as a "server": it listens on the channel x for a request, then runs the process Q with the request's values as arguments. Notice that the read, $x(y^+)$, is always prefixed by replication, !. This lets the server process handle any number of requests over time. For example, when the process $!x(k).\, \overline{k}\langle a \rangle$ receives a request on channel x, it responds with a and creates a duplicate of itself to do the same thing on the next request.

The only terms without counterparts are applications with λ-abstractions in head position—that is, β-redexes. But we can handle them by having them eliminated as part of the translation. Thus, we can faithfully translate $\lambda_{cps,vn}$ to the π-calculus:

$$\mathcal{P}[\![V(y^+)]\!] \triangleq \mathcal{P}[\![V]\!]_{y^+}$$

$$\mathcal{P}[\![\nu x.\, x \coloneqq \lambda(y^+).\, N \textbf{ in } M]\!] \triangleq \boldsymbol{\nu} x\, (\mathcal{P}[\![M]\!] \mid !x(y^+).\, \mathcal{P}[\![N]\!])$$

$$\mathcal{P}[\![f]\!]_{y^+} \triangleq \overline{f}\langle y^+ \rangle$$

$$\mathcal{P}[\![\lambda(x^+).\, M]\!]_{y^+} \triangleq \mathcal{P}[\![M]\!]\{y^+/x^+\}$$

The subscripted form of $\mathcal{P}$ translates a term, given the arguments it is being applied to, performing β-reduction as needed. These reductions are strongly normalizing (as we will see shortly) and the transform is compositional and defined by structural induction on terms, ensuring that $\mathcal{P}$ is well-defined. Finally, we can derive the π-calculus encoding, $\mathcal{E}$, corresponding to *any* CPS transform, $\mathcal{C}$, targeting λ_{cps} by composing it with the naming transform $\mathcal{N}$ and the π-calculus translation $\mathcal{P}$:

$$\mathcal{E}[\![M]\!]_k \triangleq \mathcal{P}[\![\mathcal{N}[\![\mathcal{C}[\![M]\!]k]\!]]\!]$$

The π-encoding derived from the uniform CPS transform in this way is shown in Fig. 12. The final product coincides with the established uniform π-encoding [20].[3]

Correctness of the π-Encoding

To prove the transform $\mathcal{P}$ correct, we need to be able to relate the final states of the named CPS calculus with observations in the π-calculus. For a π-calculus term P, we can observe whether

[3] The final transform in Fig. 12 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).

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

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

Figure 13. The call-by-need λ-calculus, λ_{need}

P is capable of performing a write on the free channel name $\mathbf{ret}$ (possibly after some reductions). Therefore an answer of the form $E[\mathbf{ret}\, V]$ should relate to some process with an externally visible write on channel $\mathbf{ret}$. Stuck states similarly correspond to observable writes. The simulation relation, $M \sim P$, is just the transform up to structural congruence, $\mathcal{P}[\![M]\!] \equiv P$. One detail to note is that all available β-reductions in the named CPS calculus are performed during the $\mathcal{P}$ transform. This means that we need to consider β to be an administrative reduction; it is strongly normalizing since every β-reduction reduces the number of λ-abstractions in a term (since only names can be duplicated). From this definition of the simulation relation and administrative reduction, the sufficient conditions in Fig. 7 follow directly. Observe that $\mathcal{P}$ turns answers and stuck states in $\lambda_{cps,vn}$ into processes with externally visible writes, and the remaining diagrams can be checked for the (proper) $deref$ and (administrative) β reductions.

6. Call-by-Need and Constructive Update

In order to better describe the behavior of implementations of lazy languages, Ariola et al. [4] devised the *call-by-need λ-calculus*, shown in Fig. 13, which accounts for *memoization* of computed values. It introduces answers as a syntactic category: an answer is a value surrounded by some number of $\mathbf{let}$ bindings. As $\mathbf{let}$ is understood to be lazy, control passes into the bound expression only when the value of a binding is needed. This is expressed by evaluation contexts of the shape $\mathbf{let}\ x = E\ \mathbf{in}\ F[x]$: if the inner expression has the form $F[x]$ for some evaluation context F, that means the value of x is required, so computation should take place inside the binding for x.

There does exist a call-by-need CPS transform due to Okasaki et al. [17] It requires mutable storage, which our CPS languages do not support. However, suppose we borrow the $:=$ 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:

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

This is similar to the Okasaki CPS transform. Unfortunately, it is not valid $\lambda_{cps,vn}$ syntax, as the operator $:=$ is only allowed to assign to a variable immediately inside the ν that introduces it. The inner continuation, $\lambda w.\, x := (\lambda k''.\, k'' w)\ \mathbf{in}\ k' w$, violates this restriction by attempting to "overwrite" x. Of course, this is precisely what we wish to happen: the term bound to x *should* change in order to cache the computed value. However, we don't need the full power of mutable storage; a much weaker effect will suffice.

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

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

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. 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 $\mathbf{let}$ is not recursive). Therefore, the initial thunk is only used once; since that very thunk performs the second assignment, the first lookup (via $deref$) 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. 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.

Call-by-Need in the Uniform Transform

Writing $x := V\ \mathbf{in}\ N$ for a permanent assignment and $x :=_1 V\ \mathbf{in}\ N$ for an ephemeral assignment, we can modify the call-by-need CPS transform so that it does not require destructive update, as shown in Fig. 14 (the cases for variables and abstractions are the same as before). The initial thunk is now assigned ephemerally.

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 other words, update never changes the value associated with a name, but only gives new meaning to names with no assigned value. In this language, updates only construct, never destroy; hence we call the phenomenon *constructive update*. The syntax and semantics for permanent and ephemeral assignment are given in Fig. 15. The $deref_1$ rule is similar to $deref$, only it also 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 12. *A $\lambda_{cps}^{:=_1}$ 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 13. *For any term M, $\mathcal{C}[\![M]\!]\,\mathbf{ret}$ is safe.*

$$
\begin{array}{rrl}
\text{Terms:} & M, N ::= & V(V^+) \mid \nu x.\, M \mid \\
& & x \coloneqq \lambda(x^+).\, M \textbf{ in } N \mid \\
& & x :=_1 \lambda(x^+).\, M \textbf{ in } N \\
\text{Values:} & V ::= & x \mid \textbf{ret} \mid \lambda(x^+).\, M \\
\text{Eval. Cxts.:} & E ::= & [\,] \mid \nu x.\, E \mid x \coloneqq \lambda(x^+).\, M \textbf{ in } E \mid \\
& & x :=_1 \lambda(x^+).\, M \textbf{ in } E
\end{array}
$$

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

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

Call-by-Need 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. 16. There are 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 maps *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][4], except that a suspended computation is removed when it is retrieved from the environment. In this way, it resembles the original call-by-need abstract machine by Sestoft [21]. 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 [10], 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 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.

7. Correctness of the Call-by-Need CPS Transform

We would like to follow the same general outline used in Section 4 for the uniform CPS transform to show the correctness of the call-by-need transform as well. However, attempting to do so directly raises two serious issues:

1. The transformations of a function application and a **let** binding introduce code that memoizes the result of a bound term so that the result may be reused in the future. That means that the first time a bound term is accessed and a result is returned, that memoization code is "used up." In other words, a term like $\textbf{let } x = V \textbf{ in } M$ means something different in the CPS transform before and after the first time M accesses the variable x.

2. The proof in Section 4 relied on Proposition 7, which implies that for a source term $E[R]$ where the next redex R is hidden in an evaluation context, administrative reduction of the CPS transformed term is strong enough to bring out the redex. This would suffice for the call-by-need calculus except for evaluation contexts of the form $\textbf{let } x = E \textbf{ in } F[x]$. In this case, reduction of the transformed term will have to look up x before finding the redex in E, which we may not want to consider a mere administrative step.

In both cases, the root problem is that the call-by-need CPS transform brings out some of the inherent statefulness of memoization which is not expressed by the source calculus. Therefore we address these issues by introducing annotations that track such stateful transitions and allow us to devise reduction rules that more closely mimic what is done by the CPS terms. Thus we factor the correctness of the call-by-need CPS transform into two parts: correctness of the annotations, and correctness of the annotated CPS transform.

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

$$
\begin{array}{rcl}
\langle x, \sigma, \rho, k\rangle_M & \longmapsto & \langle t, \sigma, k\rangle_T \\
\multicolumn{3}{c}{\text{where } \rho(x) \equiv \ell \text{ and } \sigma(\ell) \equiv t} \\
\langle \lambda x.\, M, \sigma, \rho, k\rangle_M & \longmapsto & \langle k, \mathsf{Clos}\langle x, M, \rho\rangle, \sigma\rangle_K \\
\langle MN, \sigma, \rho, k\rangle_M & \longmapsto & \langle M, \sigma, \rho, \mathsf{Apply}\langle N, \rho, k\rangle\rangle_M \\
\\
\langle \mathsf{Susp}\langle M, \ell, \rho\rangle, \sigma, k\rangle_T & \longmapsto & \langle M, \sigma[\ell = \perp], \rho, \mathsf{Update}\langle \ell, k\rangle\rangle_M \\
\langle \mathsf{Memo}\langle f\rangle, \sigma, k\rangle_T & \longmapsto & \langle k, f, \sigma\rangle_K \\
\\
\langle \mathsf{Ret}\langle\rangle, f, \sigma\rangle_K & \longmapsto & \langle f, \sigma\rangle_H \\
\langle \mathsf{Apply}\langle M, \rho, k\rangle, f, \sigma\rangle_K & \longmapsto & \langle f, \ell, \sigma[\ell = \mathsf{Susp}\langle M, \ell, \rho\rangle], k\rangle_F \\
\multicolumn{3}{c}{\text{where } \ell \notin \sigma} \\
\langle \mathsf{Update}\langle \ell, k\rangle, f, \sigma\rangle_K & \longmapsto & \langle k, f, \sigma[\ell = \mathsf{Memo}\langle f\rangle]\rangle_K \\
\\
\langle \mathsf{Clos}\langle x, M, \rho\rangle, \ell, \sigma, k\rangle_F & \longmapsto & \langle M, \sigma, \rho[x = \ell], k\rangle_M
\end{array}
$$

Figure 16. The abstract machine derived from $\mathcal{C}$

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

$$
\begin{array}{rrl}
\text{Expressions:} & M, N ::= & x \mid V \mid MN \mid \mathbf{let_s}\, x = M \,\mathbf{in}\, N \mid \\
& & \mathbf{let_a}\, x = M \,\mathbf{in}\, E[x] \mid \\
& & \mathbf{let_v}\, x = V \,\mathbf{in}\, N \\
\text{Values:} & V ::= & \lambda x.\, M \\
\text{Eval. Cxts.:} & E, F ::= & [\,] \mid EM \mid \mathbf{let_s}\, x = M \,\mathbf{in}\, E \mid \\
& & \mathbf{let_a}\, x = E \,\mathbf{in}\, F[x] \mid \\
& & \mathbf{let_v}\, x = V \,\mathbf{in}\, E \\
\text{Bind. Cxts.:} & B ::= & [\,] \mid \mathbf{let_s}\, x = M \,\mathbf{in}\, B \mid \\
& & \mathbf{let_v}\, x = V \,\mathbf{in}\, B
\end{array}
$$

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

Figure 17. The annotated call-by-need λ-calculus, λ^a_{need}

Annotations

As discussed previously, there is an implicit statefulness in memoization that is made manifest by the CPS transform. Specifically, there are three stages in the life cycle of a $\mathbf{let}$ binding.

Suspended Initially, in $\mathbf{let}\, x = M \,\mathbf{in}\, N$ the binding 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.

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. The act rule simply marks the $\mathbf{let}$ as active, while $deact$ is more involved, as it must take the computed answer $B[V]$, add the bindings B to the surrounding context, mark the $\mathbf{let}$ as a value, and finally plug V into the context that demanded x. Next, we can simplify the language by considering the *lift* and $assoc$ rules as administrative, as they only serve to move parts of a redex closer together. We can avoid this administrative work in the annotated 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. This lets us merge every sequence of *lift*s with a β step, and every sequence of $assoc$s with a $deref$. These modifications to reduction give us the annotated λ^a_{need}-calculus shown in Fig. 17.

We can transform an unannotated term into an annotated one, written $\mathcal{A}[\![M]\!]$, by annotating every $\mathbf{let}$ with $\mathbf{s}$. However, it is easier to begin with an annotated term since it contains more information, and to work the other way by removing all the annotations with the inverse transform $\mathcal{A}^{-1}$. Note that $\mathcal{A}^{-1}\langle\!\langle \mathcal{A}[\![M]\!] \rangle\!\rangle \triangleq M$, so that the correctness of $\mathcal{A}^{-1}$ implies the correctness of $\mathcal{A}$. We want to relate answers of the form $B[V]$ and stuck terms of the form $E[x]$, for a free variable x, in their respective calculi. The simulation relation that we will use, $M \sim P$, relates annotated terms to their erasure, $P \triangleq \mathcal{A}^{-1}\langle\!\langle M \rangle\!\rangle$. With this relation, we can see that the act rule in the annotated calculus is administrative, and it is strongly normalizing since it reduces the number of $\mathbf{s}$ annotations in a term. The sufficient conditions in Fig. 7 then follow immediately. Observe that removing the annotations preserves answers and stuck states, and the remaining diagrams can be checked by cases for the proper (β_a, $deact$, $deref_a$) and administrative (act) reductions of λ^a_{need}. Note that β_a incorporates some $lift$ steps in λ_{need}, and $deact$ incorporates some $assoc$ steps.

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

Figure 18. The call-by-need CPS transform on annotated terms.

Annotated CPS Transform

We would like to follow the proof in Section 4 of the correctness of the uniform CPS transform as closely as possible. To that end, we define our transform similarly as $\mathcal{C}_a[\![M]\!]\,\mathbf{ret}$, as shown in Fig. 18. Notice that in the transformation of $\mathbf{let_a}$, by choosing a fresh y we can use an administrative reduction substitute the memoized term exactly once into the hole in E. Our goal is then to ensure that an answer in the annotated calculus, which is of the form $B[V]$, relates to an answer in the CPS transform, which is of the form $B[\mathbf{ret}\, V]$. Likewise, we also want evaluation to stuck states of the form $E[x]$ in the annotated calculus to relate to some stuck state of the form $B[xV]$ in the CPS transform.

As before, we need to define the simulation relation $\sim$ to relate terms in a way that lets us ignore administrative reductions. The administrative reductions include the β_{ad} rule for marked λ-abstractions, which are shown in Fig. 18. However, we will find that this alone is not enough, since reduction of the CPS transform gets out of sync with the original term in another way. Specifically, the $deact$ rule for a term like $\mathbf{let_a}\, x = B[V] \,\mathbf{in}\, E[x]$ remembers the shared value by updating the binding of a variable x. In the CPS transform this updated assignment to x occurs *inside* of $E[x]$, but if we were to transform the resulting term $B[\mathbf{let_v}\, x = V \,\mathbf{in}\, E[V]]$ we would find the assignment to x at the introduction of the $\mathbf{let}$. Therefore, we also need to be able to relate terms up to a stronger notion of congruence that ignores different orders of assignments, and so we extend the administrative reductions with a rule

$$x := \lambda(x^+).\, N \,\mathbf{in}\, E[M] \longrightarrow_{ad} E[x := \lambda(x^+).\, N \,\mathbf{in}\, M] \quad lift$$

that transports an assignment. As a side condition, *lift* requires that x is not captured by a νx in E.

In the end, we define the simulation relation $\sim$ in terms of β_{ad} and *lift* congruence:

Definition 14. $M \sim P$ *iff* $\mathcal{C}_a[\![M]\!]\,\mathbf{ret} =_{ad} P$.

Note that the *lift* rule is purely for mediating the simulation and is never a standard reduction, since the next β or $deref$ step neither depends on nor is affected by *lift*.

We now need to demonstrate the conditions in Fig. 7, and as in Section 4 the initial condition is trivial. Next, because the reduction rules of λ^a_{need} make use of evaluation contexts, we need to understand how the CPS transform interprets evaluation contexts to show the inductive step. What we find is similar to Proposition 7, except now administrative standard reduction converts an evaluation context in the annotated calculus into a continuation *and* a binding context in the CPS calculus.

Proposition 15. *For each evaluation context E in λ^a_{need}, there is a continuation K' and binding context B' such that for every term M, $\mathcal{C}_a[\![E[M]]\!]\,\mathbf{ret} \longmapsto\!\!\!\!\rightarrow_{ad} B'[\mathcal{C}_a[\![M]\!]K']$.*

Intuitively, the continuation K' represents the applicative part of the evaluation context of the form EM and the binding context B' represents the part of the evaluation context that goes under a **let** binding. We can then make use of Proposition 15 to show by calculation that if $M \longrightarrow N$ at the top level, then

$$\mathcal{C}_a[\![M]\!]\,\mathbf{ret} \longmapsto\!\!\!\!\rightarrow_{ad} \longmapsto_{pr} =_{ad} \mathcal{C}[\![N]\!]\,\mathbf{ret} \tag{6}$$

For most rules, the final $=_{ad}$ is just β_{ad}, but for the act rule, the final $=_{ad}$ must make use of *lift* to pull out the updated assignment of the bound variable. Generalizing this statement to establish the inductive step of the simulation follows the same procedure as before in Section 4. First, in order to establish (6) for reduction in an evaluation context, we make further use of Proposition 15 to bring out the redex inside of M and the result of reduction inside of N. Second, in order to establish (6) for any starting point in the relation, we need to commute administrative equality across proper reductions, as we did before with Lemma 8.

Lemma 16. *If $M =_{ad} M' \longmapsto_{pr} N$ then there is an N' such that $M \longmapsto^+ N' =_{ad} N$.*

Proof. The same as the proof of Lemma 8, again relying on similar properties of confluence, standardization, and commutation of administrative and proper steps for the administrative reduction relation of $\lambda^{:=_1}_{cps}$. □

This allows us to complete the diagrams in Fig. 7; the remainder of the proof follows similarly to Section 4.

Proposition 17. *If $M \longmapsto N$ and $M \sim P$ then there is a $N \sim Q$ such that $P \longmapsto^+ Q$.*

Proposition 18. *1. If $P =_{ad} P'$ and $P \downarrow$ then $P \Downarrow'$.*
2. If $P =_{ad} P'$ and $P \not\downarrow$, then $P' \not\Downarrow$.

8. Call-by-Need and Processes

In Section 5, the π-encoding of the $\lambda_{cps,vn}$-calculus reached a very small subset of the π-calculus. Recall that the "server" processes, created by permanent assignments, always respond in exactly the same way every time they are invoked. However, the full π-calculus allows us to write server processes that change their behavior over the course of their lifetime, responding differently from one request to the next. This is achieved by having processes with read operations that are *not* replicated. For example, we could have a process like $x(k).\,(\overline{k}\langle b\rangle \mid !x(k).\,\overline{k}\langle a\rangle)$, which responds to its first request with the answer b and then becomes a replicated process that always responds with a. This type of statefulness is key for encoding call-by-need, as a thunk's behavior changes after its first evaluation.

Naming and Constructive Update

The naming transform of $\lambda^{:=_1}_{cps}$ is effectively the same as before in Section 5, with the small difference that we are now starting with terms that already have some amount of naming in them. Therefore, we extend the previous transform to preserve any existing naming in a term, and to give names to function arguments as before:

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

Note that this new $\mathcal{N}$ transform converts a $\lambda^{:=_1}_{cps}$ term into a subset of the full $\lambda^{:=_1}_{cps}$-calculus where function arguments must be variables.

Before, we proved correctness of the naming transform by instead considering the inverse transform that removes all sharing from a term. However, since we now begin with a term that has some amount of sharing, an inverse transform is no longer so straightforward. For that reason, we instead show the correctness of naming directly, which forces us to reason up to "extra sharing." This relation $M \prec P$, read "M has less sharing than P," is defined as the reflexive, transitive closure of the rule

$$M\{V/x\} \prec \nu y.\,y := V\ \mathbf{in}\ M\{y/x\}$$

where y is not a free variable of V or M. Intuitively, this relation states that substitution can just as well be implemented by a permanent assignment on a private name without changing the behavior of a term. We then define the simulation relation as the naming transform up to extra sharing.

Definition 19. *$M \sim P$ iff $\mathcal{N}[\![M]\!] \prec P$.*

We now need to establish the sufficient conditions for correctness. The initial condition is satisfied by reflexivity, and the final conditions come from the fact that if $M \prec P$ then M is an answer iff P is (and likewise for stuck terms). We therefore only need to demonstrate the inductive step. We can show that if $M \longmapsto N$, then there is a Q such that:

$$\mathcal{N}[\![M]\!] \longmapsto Q \qquad \mathcal{N}[\![N]\!] \prec Q \tag{7}$$

So it remains to show that any P with more sharing than $\mathcal{N}[\![M]\!]$ will do something similar.

Proposition 20. *If $M \longmapsto N$ and $M \prec P$ then there is a Q such that $P \longmapsto^+ Q$ and $N \prec Q$.*

For the $deref$ and $deref_l$ rules, this is immediate. The most difficult case is for the β rule, where in the worst case P may have to do a $deref$ step to lookup the extra name followed by a β step. Therefore, all the conditions of Fig. 7 hold.

Deriving a π-Encoding with Update

We also extend our previous π-encoding from Section 5 to account for both permanent and ephemeral assignments as follows:

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

As discussed, a permanent assignment corresponds to a replicated server process that continually responds in the same way, whereas for ephemeral assignment the server is not replicated and therefore disappears when it is first invoked. Composing together the call-by-need CPS, naming, and π transforms, so that $\mathcal{E}[\![M]\!]_k \triangleq \mathcal{P}[\![\mathcal{N}[\![\mathcal{C}[\![M]\!]k]\!]]\!]$, we arrive at the same call-by-need π-encoding (Fig. 19) found in the literature [6, 20] (up to replication of processes that are only used once).

Correctness of the extended $\mathcal{P}$ transform follows the same general proof as from Section 5: finished terms of the form $B[xV]$ correspond to processes with an externally visible write on channel x, and the simulation relation is just the $\mathcal{P}$ transform. The $deref_l$ rule for ephemeral assignment is exactly simulated by the behavior

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

$$
\mathrm{memo}_x(M) \triangleq x(k).\, \boldsymbol{\nu} h\ (\mathcal{E}[\![M]\!]_h \mid !h(w).\, (\overline{k}\langle w\rangle \mid !x(k').\, \overline{k'}\langle w\rangle))
$$

Figure 19. The π-encoding for call-by-need

of an unreplicated server process. The conditions from Fig. 7 follow directly, since $\mathcal{P}$ preserves the final states and reductions of $\lambda^{:=1}_{cps}$.

9. Related Work

Brock and Ostheimer [6] gave a proof of correctness for the call-by-need π-calculus encoding, but without connecting it to CPS. Okasaki *et al.* [17] 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 [15], 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 [19], 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] arrives at different CBN and CBV π-encodings by taking another approach entirely: he begins by extending the source calculi with explicit substitutions, employing a *linear head reduction* strategy reminiscent of call-by-need. Since linear head reduction performs substitutions one at a time and only when needed, reductions in the modified calculi then correspond to π-calculus reductions one-to-one, making for straightforward proofs of correctness for the derived encodings. His development is related to that by Toninho, Caires, and Pfenning [22], which relates the simply-typed λ-calculus to the π-calculus via linear logic.

The advantages of using distance rules to eliminate administrative work have been explored by Chang and Felleisen [7], who reformulate the call-by-need λ-calculus using a single β-rule; and by Accattoli [1], who expresses both (modified) λ-calculi and 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 [6] and Danvy and Zerny [10] include versions of what we call $\mathbf{let_a}$, the "active $\mathbf{let}$."

10. Conclusion and Future Work

Separate lines of work have connected CPS transforms to the π-calculus and to abstract machines, both in the context of call-by-value and call-by-name evaluation. In this paper, we bring them together, extending the inter-derivation of semantic artifacts to include π-encodings, and moreover we show that a more limited effect than mutable state in the target CPS calculus is required to give a direct account of memoization and to extend the full correspondence to call-by-need evaluation. The transformation from CPS to processes applies to an entire language of CPS programs, giving a general and systematic method for deriving a π-encoding from CPS transforms. To show this, we study a less-known uniform CPS transform with variations for each evaluation strategy and present how to derive the different semantic artifacts. Considering the correspondence between the call-by-need CPS transform and π-encoding has brought to light a new effect which more precisely expresses memoization and which may be interesting in its own right. As future work, we plan to capitalize on this connection by exploring whether the type systems for the π-calculus [20] can delineate a typed call-by-need CPS transform.

Acknowledgements

We would like to thank Olivier Danvy for his encouragement in pursuing this line of research, as well as the anonymous reviewers for their feedback and help in improving this paper. Paul Downen, Luke Maurer, and Zena M. Ariola have been supported by NSF grant CCF-0917329. Luke Maurer was also supported by an INRIA internship in the joint team πr^2.

References

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

[2] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP*, pages 8–19, 2003.

[3] R. Amadio. A decompilation of the pi-calculus and its application to termination. (CNRS 7126), 2011.

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

[5] Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *IC*, 139, 1996.

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

[7] S. Chang and M. Felleisen. The call-by-need lambda calculus, revisited. In *PLS*, pages 128–147. 2012.

[8] O. Danvy and A. Filinski. Representing control: A study of the CPS transformation. *MSCS*, 2(04), 1992.

[9] O. Danvy and L. R. Nielsen. A first-order one-pass CPS transformation. *TCS*, 308(1):239–257, 2003.

[10] O. Danvy and I. Zerny. A synthetic operational account of call-by-need evaluation. In *PPDP*, 2013.

[11] M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the λ-calculus. In *FDPC*, 1986.

[12] J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *POPL*, pages 458–471, 1994.

[13] J.-L. Krivine. A call-by-name lambda-calculus machine. *HOSC*, 20(3):199–207, 2007.

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

[15] R. Milner. Functions as processes. *MSCS*, 2(02):119–141, 1992.

[16] R. Milner. Elements of interaction: Turing award lecture. *Commun. ACM*, 36(1):78–89, Jan. 1993.

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

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

[19] D. Sangiorgi. From λ to π; or, rediscovering continuations. *MSCS*, 9(4):367–401, July 1999.

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

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

[22] B. Toninho, L. Caires, and F. Pfenning. Functions as session-typed processes. In *FoSSaCS*, pages 346–360, 2012.