LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1993
© 1993 Kluwer Academic Publishers – Manufactured in The Netherlands

Call-by-need and Continuation-passing Style

CHRIS OKASAKI (*cokasaki@cs.cmu.edu*)
PETER LEE* (*petel@cs.cmu.edu*)
DAVID TARDITI† (*dtarditi@cs.cmu.edu*)
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

Keywords: Call-by-need, Continuation-passing Style, Continuations, Lazy Evaluation, Functional Programming

Abstract. This paper examines the transformation of call-by-need λ terms into continuation-passing style (CPS). It begins by presenting a simple transformation of call-by-need λ terms into program graphs and a reducer for such graphs. From this, an informal derivation is carried out, resulting in a translation from λ terms into self-reducing program graphs, where the graphs are represented as CPS terms involving storage operations. Though informal, the derivation proceeds in simple steps, and the resulting translation is taken to be our canonical CPS transformation for call-by-need λ terms.

In order to define the CPS transformation more formally, two alternative presentations are given. The first takes the form of a continuation semantics for the call-by-need language. The second presentation follows Danvy and Hatcliff's two-stage decomposition of the call-by-name CPS transformation, resulting in a similar two-stage CPS transformation for call-by-need.

Finally, a number of practical matters are considered, including an improvement to eliminate the so-called administrative redexes, as well as to avoid unnecessary memoization and take advantage of strictness information. These improvements make it feasible to consider potential applications in compilers for call-by-need programming languages.

1. Introduction

One of the trends in compiler construction has been the use of λ terms written in continuation-passing style (CPS) as an intermediate representation [1, 19, 25]. Transformations into CPS for call-by-name and call-by-value languages are well known [8, 23, 24], but we are unaware of a similar transformation for languages implemented with a call-by-need evaluation strategy (also known as "lazy evaluation"). We find it natural to consider such a transformation. After all, one of the motivations for the use of CPS

*Supported in part by the National Science Foundation under PYI grant #CCR-9057567, with matching funds from Bell Northern Research.

†Supported by an AT&T Ph.D. scholarship.

is to expose low-level implementation details in a high-level intermediate language; as such this seems compatible with the efficiency considerations that motivate the call-by-need evaluation strategy.

We begin by considering graph reduction, which is a standard technique for implementing lazy evaluation. We examine a simple transformation of call-by-need λ terms into program graphs and a reducer for such graphs inspired by the Spineless G-machine [7]. Both the graphs and the reducer are expressed as ML programs. From this, we informally derive a translation from λ terms into self-reducing program graphs, where the graphs are represented as CPS ML terms involving storage operations. Though informal, the derivation proceeds in simple steps, and the resulting translation is taken to be our canonical CPS transformation for call-by-need λ terms.

The graph reducer memoizes the evaluation of subgraphs in order to achieve lazy behavior. Such memoization involves destructive updates of an underlying graph store; in our canonical CPS transformation such updates are expressed by assignments to ML-style mutable references. In order to make this mechanism more explicit and formal, we make an alternative presentation of the CPS transformation in the form of a continuation semantics. In this way, we provide a complete definition of the CPS terms and storage operations.

To provide yet another formal account of the call-by-need transformation, we follow Danvy and Hatcliff's decomposition of the call-by-name CPS transformation into two simpler transformations [9]. By making a simple extension to Danvy and Hatcliff's staging to account for memoization, we obtain a similar two-stage transformation, and show that this transformation is equivalent to our canonical transformation.

In the remainder of the paper we consider a number of practical matters, including an improvement to the call-by-need transformation to eliminate the so-called administrative redexes, based on the optimal CPS transformation of Danvy and Filinski [8]. Other optimizations include transformations to avoid unnecessary memoization and also to take advantage of strictness information. Finally, we conclude with a brief discussion of possible applications and directions for further work.

2. Graph Reduction and Continuation-passing Style

In graph reduction, a program is converted into a directed-graph representation. A graph reducer is then used to evaluate the program. Wadsworth was perhaps the first to use graph reduction to evaluate programs [27]. Turner popularized the idea with his implementation based on SK-combinators [26]. Many refinements to the concept have been proposed, including the Spine-

less G-machine [7] which serves as the inspiration for the model of graph reduction used here.

2.1. A graph reducer

For our purposes, programs are represented by binary DAGs (binary trees with shared subgraphs).[1] Interior nodes represent applications while leaves represent functions. (We assume programs have no free variables.) In addition, the roots of (potentially) shared subgraphs are distinguished by a special marker.

To evaluate a graph using the call-by-name strategy, the spine of the graph (that is, the path from the root to the left-most leaf) is traversed, pushing the arguments onto a stack, until a function is reached. The function is applied by building the graph corresponding to the body of the function, substituting arguments from the stack for occurrences of the function's parameters. This new graph is then evaluated with the remaining stack. Reduction is complete when a function is reached for which there are not enough arguments on the stack. (A function is considered to take n arguments if it is of the form $\lambda x_1 \ldots \lambda x_n . M$, where M is not a λ-abstraction.)

To achieve call-by-need evaluation, we must account for the memoization of shared subgraph evaluations. Memoized evaluation involves saving the current stack and evaluating the subgraph with an empty stack. The root of the subgraph is then overwritten with the result. Finally, the original stack is restored and evaluation of the new graph is continued.

More concretely, consider the datatype for graphs and the procedure for evaluating graphs presented in Figure 1 in an ML-like programming notation. Note that ML's notation for assignable references is used to implement the destructive update of graphs.

The representation of shared graphs requires a bit of explanation. The graph $\mathbf{Share}(r, m)$ is essentially an annotation indicating that the graph m is (potentially) shared. r is a self-reference to the annotated graph, which is used to facilitate its update for the purpose of memoization.

Figure 2 presents the transformation $\mathcal{G}$ of λ terms into ML terms of type $\mathbf{Graph\,ref}$. $\mathcal{G}^*$ and $\mathcal{F}$ transform shared graphs and functions respectively. Note that $\mathbf{fixref}\, r.e$ is shorthand for

$$\mathbf{let}\, r = \mathbf{ref}\, \langle dummy \rangle\, \mathbf{in}\, r := e;\, r$$

where r may occur in e.

[1] Most graph reducers also use cycles to represent recursion, but we will not consider recursive programs in this paper.

$$
\begin{array}{lcl}
\textbf{Graph} & = & \textbf{Clos}\ \text{of}\ ((\textbf{Stack} \to \textbf{Graph}) \times \textbf{Stack}) \\
& | & \textbf{App}\ \text{of}\ (\textbf{Graph ref} \times \textbf{Graph ref}) \\
& | & \textbf{Share}\ \text{of}\ (\textbf{Graph ref} \times \textbf{Graph ref}) \\
\\
\textbf{Stack} & = & \textbf{push}\ \text{of}\ (\textbf{Graph ref} \times \textbf{Stack}) \\
& | & \textbf{empty}
\end{array}
$$

$$
\begin{array}{l}
\textbf{eval} : \textbf{Graph} \to \textbf{Stack} \to \textbf{Graph} \\
\begin{array}{lcl}
\textbf{eval}\,(\textbf{Clos}\,(f, s'))\,s & = & f\,(\textbf{concat}\,s'\,s) \\
\textbf{eval}\,(\textbf{App}\,(m, n))\,s & = & \textbf{eval}\,(!\,m)\,(\textbf{push}\,(n, s)) \\
\textbf{eval}\,(\textbf{Share}\,(r, m))\,s & = & \textbf{let}\ t = \textbf{eval}\,(!\,m)\,\textbf{empty} \\
 & & \textbf{in}\ r := t;\ \textbf{eval}\,t\,s
\end{array}
\end{array}
$$

Figure 1: A simple graph reducer based on the Spineless G-machine.

Now, to evaluate a λ term M, the following ML program is executed:

$$\textbf{eval}\,(!M')\,\textbf{empty}$$

where $M' = \mathcal{G}[M]$.

Several points are worth mentioning about this graph-reduction scheme. First, the transformation $\mathcal{G}$ arranges for all arguments to be shared. This is actually overly conservative. A more practical approach is described in Section 7.1. Second, graph reducers typically assume that programs have been lambda-lifted [13, 14] prior to conversion into graph form. This greatly simplifies the handling of environments, but we make no such assumptions here. Finally, functions are "compiled" in the sense that a function is converted into an expression which, when executed, builds the graph of the function body and then reduces it. This, in fact, is the key idea of the G-machine [4, 15], one of the first practical graph reducers.

2.2. From graph reduction to continuation-passing style

We now informally derive a CPS transformation for call-by-need λ terms by incrementally modifying the representation of graphs until graphs are represented as CPS terms involving storage operations.

The first step is to remove the overhead of interpreting graphs by folding **eval** into $\mathcal{G}$, $\mathcal{G}^*$, and $\mathcal{F}$. In essence, the graphs are represented directly by

$$
\begin{array}{lcl}
\mathcal{G}[x] & = & x \\
\mathcal{G}[\lambda x_1 \ldots \lambda x_n . M] & = & \mathbf{ref}\,(\mathbf{Clos}\,(\mathcal{F}[\lambda x_1 \ldots \lambda x_n . M], \mathbf{empty}\,)) \\
\mathcal{G}[M\ N] & = & \mathbf{ref}\,(\mathbf{App}\,(\mathcal{G}[M], \mathcal{G}^*[N])) \\
\mathcal{G}^*[N] & = & \mathbf{fixref}\, r.\, \mathbf{Share}\,(r, \mathcal{G}[N])
\end{array}
$$

$$
\begin{array}{l}
\mathcal{F}[\lambda x_1 \ldots \lambda x_n . M] \;= \\
\quad \mathbf{fix}\, f.\, \lambda s.\, \mathbf{case}\, s\, \mathbf{of}\, \mathbf{push}\,(x_1, \ldots \mathbf{push}\,(x_n, s')) \Rightarrow \mathbf{eval}\,(!\,\mathcal{G}[M])\, s' \\
\qquad\qquad\qquad\qquad\quad \mathbf{otherwise} \qquad\qquad\quad \Rightarrow \mathbf{Clos}\,(f, s)
\end{array}
$$

Figure 2: Conversion of call-by-need λ terms into ML terms of type **Graph ref** .

functions derived from the reducer.[2] For instance, application graphs are represented as follows:

$$\mathcal{G}[M\ N] = \mathbf{ref}\,(\lambda s.(!\,\mathcal{G}[M])\,(\mathbf{push}\,(\mathcal{G}^*[N], s)))$$

where

$$
\begin{array}{rcl}
\mathcal{G}^*[N] & = & \mathbf{fixref}\, r.\, \lambda s.\, \mathbf{let}\ t = (!\,\mathcal{G}[N])\,\mathbf{empty} \\
& & \qquad\qquad \mathbf{in}\ r := t;\ t\ s
\end{array}
$$

In an implementation, such graphs may be represented as closures [22] or, if small enough, directly as sequences of machine code [3, 18]. (Note, however, that the latter approach involves self-modifying code, which is seldom feasible on modern computer architectures.)

This folding introduces a rather subtle effect on the time at which graphs are constructed. Previously the top-level program graph and the graphs representing the bodies of functions were built "all at once." In other words, the ML term produced by $\mathcal{G}$, when executed, built the entire graph. Now, however, graphs are built incrementally—subgraphs are constructed only when necessary. This occurs because the code for building the subcomponents of a graph (specifically, $\mathcal{G}[M]$ and $\mathcal{G}^*[N]$ in an application) is now delayed by a λ abstraction. In practice, this effect can be beneficial, for instance, when a very complicated argument is never evaluated. If for some reason such incremental behavior is not desired, the original behavior can be regained by "hoisting" the construction of the subgraphs outside

[2] This is comparable to the generation of compilers from interpreters by partial evaluation [11].

the scope of the abstraction. For instance:

$$\begin{aligned}\mathcal{G}[M\ N] = \textbf{let}\ & m = \mathcal{G}[M] \\ & n = \mathcal{G}^*[N] \\ \textbf{in}\ & \textbf{ref}\,(\lambda s.\,(!\,m)\,(\textbf{push}\,(n, s)))\end{aligned}$$

However, in all that follows, we will assume that incremental behavior is acceptable.

Next, we observe that only shared graphs are ever destructively updated. Thus, we can avoid placing any other graphs in the store.

$$\mathcal{G}[M\ N] = \lambda s.\,\mathcal{G}[M]\,(\textbf{push}\,(\mathcal{G}^*[N], s))$$

where

$$\begin{aligned}\mathcal{G}^*[N] = \textbf{let}\ & r = \textbf{fixref}\,r.\,\lambda s.\,\textbf{let}\ t = \mathcal{G}[N]\,\textbf{empty} \\ & \qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ r := t;\ t\,s \\ \textbf{in}\ & \lambda s.\,(!\,r)\,s\end{aligned}$$

Finally, we note that the argument stack is, in essence, a representation of a portion of the current continuation, namely, that part of the continuation arising from function application. More specifically, the argument stack

$$\textbf{push}\,(n_1, \textbf{push}\,(n_2, \textbf{push}\,(n_3, \ldots)))$$

is a representation of the continuation

$$(\lambda m_1.\,m_1 n_1 (\lambda m_2.\,m_2 n_2 (\lambda m_3.\,m_3 n_3 (\ldots))))$$

Expressing argument stacks as their equivalent continuations and introducing continuations to capture the sequencing implicit in the let-statements and storage operations leads to the translation shown in Figure 3. Note that without a stack, we cannot easily consume several arguments at once. Hence, we restrict our attention to single-argument (but higher-order) functions. Further, note that the SML storage primitives **ref**, !, and := have been replaced by the continuation-passing style storage primitives **new**, **deref**, and **assign**. These primitives will be defined formally in Section 4. However, note that for readability we will often use a semicolon (;) to delimit the third argument of **assign**. Thus, there is no difference between

$$\textbf{assign}\,r\,t\,;k\,n$$

and

$$\textbf{assign}\,r\,t\,(k\,n)$$

$$
\begin{array}{lcl}
\mathcal{G}[x] & = & x \\
\mathcal{G}[\lambda x.\, M] & = & \lambda k.\, k\,(\lambda x.\, \mathcal{G}[M]) \\
\mathcal{G}[M\ N] & = & \lambda k.\, \mathcal{G}[M]\,(\lambda m.\, \mathbf{new}\,(\lambda r. \\
 & & \quad \mathbf{assign}\, r\,(\lambda k.\, \mathcal{G}[N](\lambda n.\, \mathbf{assign}\, r\,(\lambda k.\, k\ n);\ k\ n))\,; \\
 & & \quad m\,(\lambda k.\, \mathbf{deref}\, r\,(\lambda t.\, t\ k))\ k))
\end{array}
$$

Figure 3: A CPS graph reducer. This transformation also serves as our canonical call-by-need CPS transformation. When viewed from that perspective, we rename the transformation $\mathcal{C}_l$.

$$
\begin{array}{lcl}
\mathcal{C}_n[x] & = & x \\
\mathcal{C}_n[\lambda x.\, M] & = & \lambda k.\, k\,(\lambda x.\, \mathcal{C}_n[M]) \\
\mathcal{C}_n[M\ N] & = & \lambda k.\, \mathcal{C}_n[M]\,(\lambda m.\, m\,\mathcal{C}_n[N]\,k)
\end{array}
$$

Figure 4: Plotkin's call-by-name CPS transformation.

From one point of view, the final transformation shown in Figure 3 serves as a high-level description of a form of graph reduction inspired by the Spineless G-machine. However, in a slight change of perspective, we can also view this transformation as a call-by-need CPS transformation. Hence, we rename the transformation $\mathcal{C}_l$.

It is instructive to compare this with the call-by-name CPS transformation shown in Figure 4. The only difference is in the rule for application where, in the call-by-need transformation, "memoization code" involving several storage operations is wrapped around the argument.

3. The Source and Target Languages

Throughout this paper we present transformations that convert terms in a source language to terms in a target language. These languages, though similar, may differ in the intended evaluation strategy and may have different nonstandard extensions. In order to avoid confusion, we use the following simple notation for naming the languages.

Each language is a variant of **Exp**, the untyped λ-calculus, as given by the following grammar:

$$M \;=\; x \mid \lambda x.\, M \mid M_1\ M_2$$

Occasionally, application will be written as

$$@\, M_1\, M_2$$

Variations of **Exp** are named by subscripts that describe the intended evaluation strategy and any nonstandard extensions. The possible subscripts are as follows:

Evaluation Orders

- call-by-name (N),
- call-by-need (L),
- call-by-value (V), and
- evaluation-order independent (CPS).

Non-Standard Extensions

- (memoized) suspension primitives ($Thunk$) and
- storage primitives ($Store$).

For example, $\mathbf{Exp}_{V+Thunk}$ is a call-by-value λ-calculus extended with **force** and **delay** operations for manipulating thunks. (The precise nature of these nonstandard extensions will be defined in the following sections.) Some of the languages employ a mixed evaluation strategy. For example, $\mathbf{Exp}_{L/V}$ employs both call-by-need and call-by-value evaluation (in a manner that will be made precise later). Grammars for each of these languages are included in the appendix.

As shown in Figure 3, our canonical CPS transformation for call-by-need λ terms has the following functionality:

$$\mathcal{C}_l \;:\; \mathbf{Exp}_L \;\rightarrow\; \mathbf{Exp}_{CPS+Store}$$

Similarly,

$$\mathcal{C}_n \;:\; \mathbf{Exp}_N \;\rightarrow\; \mathbf{Exp}_{CPS}$$

4. Continuation Semantics of Call-by-need Terms

Viewing denotational semantics as "a syntax-directed translation from a source language to...some version of the lambda-calculus" (Wand [28]), we see that, with the addition of an environment, a CPS transformation corresponds quite closely to a continuation semantics. Expressing the transformation as a continuation semantics yields important insight into the

$$
\begin{array}{rlll}
 & \mathbf{Ans} & = \ldots & \text{answers} \\
\varepsilon \in & \mathbf{Val} & = \mathbf{Clos} & \text{expressible values} \\
 & \mathbf{Clos} & = \mathbf{Thunk} \to \mathbf{Thunk} & \text{closures} \\
\rho \in & \mathbf{Env} & = \mathbf{Ide} \to \mathbf{Thunk} & \text{environments} \\
\kappa \in & \mathbf{ECont} & = \mathbf{Val} \to \mathbf{Ans} & \text{expression continuations} \\
\tau \in & \mathbf{Thunk} & = \mathbf{ECont} \to \mathbf{Ans} & \text{thunks}
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{E}_n \;:\; \mathbf{Exp}_N \to \mathbf{Env} \to \mathbf{Thunk}} \\
\mathcal{E}_n[x]\,\rho\,\kappa & = & (\rho[x])\,\kappa \\
\mathcal{E}_n[\lambda x.M]\,\rho\,\kappa & = & \kappa\,(\lambda\tau.\mathcal{E}_n[M]\,\rho\{x \mapsto \tau\}) \\
\mathcal{E}_n[M\,N]\,\rho\,\kappa & = & \mathcal{E}_n[M]\,\rho\,(\lambda\varepsilon.\varepsilon\,(\mathcal{E}_n[N]\,\rho)\,\kappa)
\end{array}
$$

Figure 5: A call-by-name continuation semantics corresponding to $\mathcal{C}_n$.

meanings of CPS terms and storage operations. The domain equations are particularly helpful in this respect.

To begin with a straightforward example, consider the call-by-name continuation semantics (corresponding to $\mathcal{C}_n$) and domain equations given in Figure 5. We see that the meaning of a transformed term is a thunk which takes an expression continuation and produces an answer.

Next, we add the domain of stores to obtain the call-by-need continuation semantics (corresponding to $\mathcal{C}_l$) and domain equations shown in Figure 6.[3] The semantics of the storage operators are given in Figure 7.

From these, we see that the meaning of a transformed term is a thunk which takes an expression continuation and produces a command continuation, which in turn takes a store and produces an answer. Note that each command continuation corresponds to a storage operation. Informally, we can think of a thunk as taking an expression continuation and performing all of the reductions up to the next storage operation before invoking that storage operation (*i.e.*, command continuation) with the current store.

5. A Two-stage Call-by-need CPS Transformation

Danvy and Hatcliff [9] demonstrate that the call-by-name CPS transformation $\mathcal{C}_n$ can be decomposed into two distinct stages: the introduction of

[3] Of course, in the absence of side-effects, call-by-need and call-by-name are denotationally indistinguishable. Still, a separate denotational semantics with explicit stores for call-by-need is useful as a framework for adding side-effects such as I/O.

$$
\begin{array}{lll}
 & \mathbf{Ans} & = \ldots \qquad \text{answers} \\
\varepsilon \in & \mathbf{Val} & = \mathbf{Clos} \qquad \text{expressible values} \\
 & \mathbf{Clos} & = \mathbf{Thunk} \to \mathbf{Thunk} \qquad \text{closures} \\
\rho \in & \mathbf{Env} & = \mathbf{Ide} \to \mathbf{Thunk} \qquad \text{environments} \\
\alpha \in & \mathbf{Loc} & \qquad \text{locations} \\
\sigma \in & \mathbf{Store} & = \mathbf{Loc} \to \mathbf{Thunk} \qquad \text{stores} \\
\theta \in & \mathbf{CCont} & = \mathbf{Store} \to \mathbf{Ans} \qquad \text{command continuations} \\
\kappa \in & \mathbf{ECont} & = \mathbf{Val} \to \mathbf{CCont} \qquad \text{expression continuations} \\
\tau \in & \mathbf{Thunk} & = \mathbf{ECont} \to \mathbf{CCont} \qquad \text{thunks}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E} : \mathbf{Exp}_L \to \mathbf{Env} \to \mathbf{Thunk} \\
\mathcal{E}_l[\![x]\!]\,\rho\,\kappa = (\rho[\![x]\!])\,\kappa \\
\mathcal{E}_l[\![\lambda x.\,M]\!]\,\rho\,\kappa = \kappa\,(\lambda\tau.\,\mathcal{E}_l[\![M]\!]\,\rho\{x \mapsto \tau\}) \\
\mathcal{E}_l[\![M\;N]\!]\,\rho\,\kappa = \mathcal{E}_l[\![M]\!]\,\rho\,(\lambda\varepsilon.\,\mathbf{new}\,(\lambda\alpha. \\
\qquad \mathbf{assign}\,\alpha\,(\lambda\kappa.\,\mathcal{E}_l[\![N]\!]\,\rho\,(\lambda\varepsilon.\,\mathbf{assign}\,\alpha\,(\lambda\kappa.\,\kappa\,\varepsilon);\\
\qquad\qquad\qquad\qquad \kappa\,\varepsilon))\,; \\
\qquad \varepsilon\,(\lambda\kappa.\,\mathbf{deref}\,\alpha\,(\lambda\tau.\,\tau\,\kappa))\,\kappa))
\end{array}
$$

Figure 6: A call-by-need continuation semantics corresponding to $\mathcal{C}_l$.

$$
\begin{array}{l}
\mathbf{new} : (\mathbf{Loc} \to \mathbf{CCont}) \to \mathbf{CCont} \\
\mathbf{new}\,\kappa_l\,\sigma = \kappa_l\,\alpha\,\sigma \qquad \text{where } \alpha \in \mathbf{free}\,\sigma \\
\\
\mathbf{deref} : \mathbf{Loc} \to (\mathbf{Thunk} \to \mathbf{CCont}) \to \mathbf{CCont} \\
\mathbf{deref}\,\alpha\,\kappa_t\,\sigma = \kappa_t\,(\sigma\,\alpha)\,\sigma \\
\\
\mathbf{assign} : \mathbf{Loc} \to \mathbf{Thunk} \to \mathbf{CCont} \to \mathbf{CCont} \\
\mathbf{assign}\,\alpha\,\tau\,\theta\,\sigma = \theta\,\sigma\{\alpha \mapsto \tau\}
\end{array}
$$

Figure 7: The semantics of the CPS storage operators.

$$\begin{array}{lcl}
\mathcal{T} & : & \mathbf{Exp}_N \rightarrow \mathbf{Exp}_{V+Thunk} \\
\mathcal{T}[x] & = & \mathbf{force}\, x \\
\mathcal{T}[\lambda x.\, M] & = & \lambda x.\, \mathcal{T}[M] \\
\mathcal{T}[M\ N] & = & \mathcal{T}[M]\,(\mathbf{delay}\, \mathcal{T}[N])
\end{array}$$

Figure 8: A transformation mapping call-by-name terms into a call-by-value language with explicit thunks.

thunks by a "thunkifying" transformation $\mathcal{T}$ and the introduction of continuations by the call-by-value CPS transformation $\mathcal{C}_v$ (extended to handle thunks). Thus $\mathcal{C}_n$ can be written as $\mathcal{C}_n = \mathcal{C}_v \circ \mathcal{T}$. We extend this result to call-by-need by accounting for memoization.

First, consider $\mathbf{Exp}_{V+Thunk}$, a call-by-value λ-calculus extended with two operations on thunks:

- **delay** M to construct a thunk for the term M, and
- **force** M to evaluate the thunk which is the value of M.

Call-by-name terms can be simulated in this language by the "thunkifying" transformation $\mathcal{T}$, shown in Figure 8, which delays each argument and forces each variable.

To extend this to call-by-need, we merely replace the rule for application with

$$\mathcal{T}[M\ N] = \mathcal{T}[M]\,(\mathbf{delay}^*\, \mathcal{T}[N])$$

where $\mathbf{delay}^*$ is identical to **delay**, except that its intended semantics include memoization.

Now, for $\mathcal{C}_v$, we simply take the usual call-by-value CPS transformation and extend it to handle the operations on thunks. The complete transformation is given in Figure 9. The **delay** and **force** operators are easily transformed as follows:

$$\begin{array}{lcl}
\mathcal{C}_v[\mathbf{delay}\, M] & = & \lambda k.\, k\,(\mathcal{C}_v[M]) \\
\mathcal{C}_v[\mathbf{force}\, M] & = & \lambda k.\, \mathcal{C}_v[M]\,(\lambda m.\, m\, k)
\end{array}$$

$\mathbf{delay}^*$ performs memoization in the usual way.

$$\begin{array}{l}
\mathcal{C}_v[\mathbf{delay}^*\, M] = \lambda k.\, \mathbf{new}\,(\lambda r. \\
\qquad \mathbf{assign}\, r\,(\lambda k.\, \mathcal{C}_v[M](\lambda m.\, \mathbf{assign}\, r\,(\lambda k.\, k\, m);\, k\, m))\,; \\
\qquad k\,(\lambda k.\, \mathbf{deref}\, r\,(\lambda t.\, t\, k)))
\end{array}$$

$$
\begin{array}{l}
\mathcal{C}_v \;:\; \mathbf{Exp}_{V+Thunk} \rightarrow \mathbf{Exp}_{CPS+Store} \\
\mathcal{C}_v[x] \;=\; \lambda k.\, k\, x \\
\mathcal{C}_v[\lambda x.\, M] \;=\; \lambda k.\, k\, (\lambda x.\, \mathcal{C}_v[M]) \\
\mathcal{C}_v[M\; N] \;=\; \lambda k.\, \mathcal{C}_v[M](\lambda m.\, \mathcal{C}_v[N](\lambda n.\, m\, n\, k)) \\
\mathcal{C}_v[\mathbf{force}\, M] \;=\; \lambda k.\, \mathcal{C}_v[M]\, (\lambda m.\, m\, k) \\
\mathcal{C}_v[\mathbf{delay}\, M] \;=\; \lambda k.\, k\, (\mathcal{C}_v[M]) \\
\mathcal{C}_v[\mathbf{delay}^*\, M] \;=\; \lambda k.\, \mathbf{new}\, (\lambda r. \\
\qquad\qquad \mathbf{assign}\, r\, (\lambda k.\, \mathcal{C}_v[M]\, (\lambda m.\, \mathbf{assign}\, r\, (\lambda k.\, k\, m); k\, m))\,; \\
\qquad\qquad k\, (\lambda k.\, \mathbf{deref}\, r\, (\lambda t.\, t\, k)))
\end{array}
$$

Figure 9: Plotkin's call-by-value CPS transformation extended to treat operations on thunks.

Theorem 1 *For any term M in $\mathbf{Exp}_L$,*

$$\mathcal{C}_l[M] = \mathcal{C}_v[\mathcal{T}[M]]$$

> **Proof:** By structural induction. Each case proceeds by expanding the definitions of $\mathcal{T}$ and $\mathcal{C}_v$, and reducing as appropriate. The three cases are shown in Figure 10. ∎

Less formally, we simply write $\mathcal{C}_l = \mathcal{C}_v \circ \mathcal{T}$. Decomposing $\mathcal{C}_l$ in this manner is convenient because many optimizations to the transformation may be localized to just one of the subcomponents. Improvements to $\mathcal{C}_v$ are described in Section 6, while improvements to $\mathcal{T}$ are presented in Section 7.

Alternatively, it may be useful to factor the description of memoization into a third stage $\mathcal{M}$, such that

$$\mathcal{C}_l = \mathcal{C}_v \circ \mathcal{M} \circ \mathcal{T}$$

In this approach, $\mathcal{M}$ transforms each **delay*** into appropriate combinations of **delay**'s, **force**'s, and call-by-value storage operations. Then $\mathcal{C}_v$ is extended to transform the (direct-style) call-by-value storage operations into CPS storage operations (but need no longer transform **delay***'s). The details of this factorization are left as an exercise for the motivated reader.

6. Eliminating Administrative Redexes

Literal implementations of transformations such as $\mathcal{C}_l$ introduce many redexes that simply manipulate continuations, doing no useful computation.

$$
\begin{array}{lcl}
\mathcal{C}_l[x] & = & \mathcal{C}_v[\mathcal{T}[x]] \\
 & = & \mathcal{C}_v[\mathbf{force}\, x] \\
 & = & \lambda k.\mathcal{C}_v[x](\lambda m.\, m\, k) \\
 & = & \lambda k.\, x\, k \\
 & = & x \\
\mathcal{C}_l[\lambda x.\, M] & = & \mathcal{C}_v[\mathcal{T}[\lambda x.\, M]] \\
 & = & \mathcal{C}_v[\lambda x.\, \mathcal{T}[M]] \\
 & = & \lambda k.\, k\, (\lambda x.\mathcal{C}_v[\mathcal{T}[M]]) \\
 & = & \lambda k.\, k\, (\lambda x.\mathcal{C}_l[M]) \\
\mathcal{C}_l[M\ N] & = & \mathcal{C}_v[\mathcal{T}[M\ N]] \\
 & = & \mathcal{C}_v[\mathcal{T}[M]\, (\mathbf{delay}^*\, \mathcal{T}[N])] \\
 & = & \lambda k.\mathcal{C}_v[\mathcal{T}[M]](\lambda m.\mathcal{C}_v[\mathbf{delay}^*\, \mathcal{T}[N]](\lambda n.\, m\, n\, k)) \\
 & = & \lambda k.\mathcal{C}_v[\mathcal{T}[M]](\lambda m.\mathbf{new}\, (\lambda r. \\
 & & \quad \mathbf{assign}\, r\, (\lambda k.\mathcal{C}_v[\mathcal{T}[N]]\, (\lambda n.\mathbf{assign}\, r\, (\lambda k.\, k\, n);\, k\, n))\,; \\
 & & \quad m\, (\lambda k.\mathbf{deref}\, r\, (\lambda t.\, t\, k))\, k)) \\
 & = & \lambda k.\mathcal{C}_l[M](\lambda m.\mathbf{new}\, (\lambda r. \\
 & & \quad \mathbf{assign}\, r\, (\lambda k.\mathcal{C}_l[N]\, (\lambda n.\mathbf{assign}\, r\, (\lambda k.\, k\, n);\, k\, n))\,; \\
 & & \quad m\, (\lambda k.\mathbf{deref}\, r\, (\lambda t.\, t\, k))\, k))
\end{array}
$$

Figure 10: Derivations in the proof of $\mathcal{C}_l[M] = \mathcal{C}_v[\mathcal{T}[M]]$.

$$
\begin{array}{l}
\mathcal{C}_{\underline{v}} \;:\; \mathbf{Exp}_{V+Thunk} \to \mathbf{Exp}_{CPS+Store} \\
\mathcal{C}_{\underline{v}}[x] = \underline{\lambda} c.\, \underline{@}\, c\, x \\
\mathcal{C}_{\underline{v}}[\lambda x.\, M] = \underline{\lambda} c.\, \underline{@}\, c\, (\underline{\lambda} x.\, \mathcal{C}_{\underline{v}}[M]) \\
\mathcal{C}_{\underline{v}}[M\ N] = \underline{\lambda} c.\, \overline{@}\, \mathcal{C}_{\overline{v}}[M]\, (\overline{\lambda} m.\, \overline{@}\, \mathcal{C}_{\overline{v}}[N]\, (\overline{\lambda} n.\, \underline{@}\, (\underline{@}\, m\, n)\, c)) \\
\mathcal{C}_{\underline{v}}[\mathbf{force}\, M] = \underline{\lambda} c.\, \overline{@}\, \mathcal{C}_{\overline{v}}[M]\, (\overline{\lambda} m.\, \underline{@}\, m\, c) \\
\mathcal{C}_{\underline{v}}[\mathbf{delay}\, M] = \underline{\lambda} c.\, \underline{@}\, c\, (\mathcal{C}_{\underline{v}}[M]) \\
\mathcal{C}_{\underline{v}}[\mathbf{delay}^*\, M] = \underline{\lambda} c.\, \underline{\mathbf{new}}\, (\underline{\lambda} r. \\
\qquad \underline{\mathbf{assign}}\, r\, (\underline{\lambda} c.\, \overline{@}\, \mathcal{C}_{\overline{v}}[M]\, (\overline{\lambda} m.\, \underline{\mathbf{assign}}\, r\, (\underline{\lambda} c.\, \underline{@}\, c\, m) \\
\qquad\qquad \underline{@}\, c\, m))\,; \\
\qquad \underline{@}\, c\, (\underline{\lambda} c.\, \underline{\mathbf{deref}}\, r\, (\underline{\lambda} t.\, \underline{@}\, t\, c)))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{C}_{\overline{v}} \;:\; \mathbf{Exp}_{V+Thunk} \to (\mathbf{Exp}_{CPS+Store} \to \mathbf{Exp}_{CPS+Store}) \to \mathbf{Exp}_{CPS+Store} \\
\mathcal{C}_{\overline{v}}[x] = \overline{\lambda} k.\, \overline{@}\, k\, x \\
\mathcal{C}_{\overline{v}}[\lambda x.\, M] = \overline{\lambda} k.\, \overline{@}\, k\, (\underline{\lambda} x.\, \mathcal{C}_{\underline{v}}[M]) \\
\mathcal{C}_{\overline{v}}[M\ N] = \overline{\lambda} k.\, \overline{@}\, \mathcal{C}_{\overline{v}}[M]\, (\overline{\lambda} m.\, \overline{@}\, \mathcal{C}_{\overline{v}}[N]\, (\overline{\lambda} n.\, \underline{@}\, (\underline{@}\, m\, n)\, (\underline{\lambda} a.\, \overline{@}\, k\, a))) \\
\mathcal{C}_{\overline{v}}[\mathbf{force}\, M] = \overline{\lambda} k.\, \overline{@}\, \mathcal{C}_{\overline{v}}[M]\, (\overline{\lambda} m.\, \underline{@}\, m\, (\underline{\lambda} a.\, \overline{@}\, k\, a)) \\
\mathcal{C}_{\overline{v}}[\mathbf{delay}\, M] = \overline{\lambda} k.\, \overline{@}\, k\, (\mathcal{C}_{\underline{v}}[M]) \\
\mathcal{C}_{\overline{v}}[\mathbf{delay}^*\, M] = \overline{\lambda} k.\, \underline{\mathbf{new}}\, (\underline{\lambda} r. \\
\qquad \underline{\mathbf{assign}}\, r\, (\underline{\lambda} c.\, \overline{@}\, \mathcal{C}_{\overline{v}}[M]\, (\overline{\lambda} m.\, \underline{\mathbf{assign}}\, r\, (\underline{\lambda} c.\, \underline{@}\, c\, m) \\
\qquad\qquad \underline{@}\, c\, m))\,; \\
\qquad \overline{@}\, k\, (\underline{\lambda} c.\, \underline{\mathbf{deref}}\, r\, (\underline{\lambda} t.\, \underline{@}\, t\, c)))
\end{array}
$$

Figure 11: A two-level specification of the extended call-by-value CPS transformation.

In a compiler, these "administrative" redexes may be eliminated in a post-pass. Alternatively, they may be reduced "on the fly" by a well-staged transformation in the style of Danvy and Filinski [8], which differentiates between administrative redexes (static terms) and abstract-syntax constructors (dynamic terms). By writing the call-by-value CPS transformation $\mathcal{C}_v$ in this form, we obtain a well-staged call-by-need CPS transformation for free via the decomposition $\mathcal{C}_l = \mathcal{C}_v \circ \mathcal{T}$.

Figure 11 contains the well-staged call-by-value CPS transformation. It can be read as a two-level specification [21], where the overlined terms (λ's and @'s) correspond to static (transformation-time) operations, and the underlined terms (λ's, @'s, and storage operations) correspond to dynamic (run-time) operations. (At transformation-time, the underlined terms are

just abstract-syntax constructors of $\mathbf{Exp}_{CPS+Store}$.) $\mathcal{C}_{\underline{v}}$ transforms terms with dynamic continuations while $\mathcal{C}_{\overline{v}}$ transforms terms with static continuations. $\mathcal{C}_{\underline{v}}$ and $\mathcal{C}_{\overline{v}}$ are related as follows:

Lemma 1 *For any term M in $\mathbf{Exp}_L$, static continuation k, and dynamic continuation c,*

$$\overline{@}\,\mathcal{C}_{\overline{v}}[M]\,k =_{\beta\eta} \underline{@}\,\mathcal{C}_{\underline{v}}[M]\,(\underline{\lambda} m.\,\overline{@}\,k\,m)$$

$$\underline{@}\,\mathcal{C}_{\underline{v}}[M]\,c =_{\beta\eta} \overline{@}\,\mathcal{C}_{\overline{v}}[M]\,(\overline{\lambda} m.\,\underline{@}\,c\,m)$$

Proof: By structural induction. ∎

Now, let $\mathcal{C}_{\underline{l}} = \mathcal{C}_{\underline{v}} \circ \mathcal{T}$ and $\mathcal{C}_{\overline{l}} = \mathcal{C}_{\overline{v}} \circ \mathcal{T}$.

Theorem 2 *For any term M in $\mathbf{Exp}_L$,*

$$\mathcal{C}_l[M] =_{\beta\eta} \mathcal{C}_{\underline{l}}[M]$$

Proof: Proceeds in the same fashion as Theorem 1, but the case for application requires one use of Lemma 1. ∎

There is one further refinement we can make to $\mathcal{C}_{\underline{v}}$. Note that

$$\begin{aligned} \mathcal{C}_{\underline{v}}[\mathbf{force}\,x] &= \underline{\lambda} c.\,\overline{@}\,\mathcal{C}_{\overline{v}}[x](\overline{\lambda} m.\,\underline{@}\,m\,c) \\ &= \underline{\lambda} c.\,\underline{@}\,x\,c \end{aligned}$$

we can eliminate this η-redex by adding the special case.

$$\mathcal{C}_{\underline{v}}[\mathbf{force}\,x] = x$$

$\mathcal{C}_{\overline{v}}$ requires no such modification since $\mathcal{C}_{\overline{v}}[\mathbf{force}\,x]$ does not produce an η-redex. Simple consequences of this modification are the identities

$$\mathcal{C}_{\underline{v}}[\mathbf{delay}\,(\mathbf{force}\,x)] = \mathcal{C}_{\underline{v}}[x] \qquad \text{and} \qquad \mathcal{C}_{\overline{v}}[\mathbf{delay}\,(\mathbf{force}\,x)] = \mathcal{C}_{\overline{v}}[x]$$

7. Optimizations

Actual implementations of functional programming languages often employ a number of optimizations based on compile-time analyses. In this section we consider two such optimizations. Rather than expressing them directly in $\mathcal{C}_l$, we confine our modifications to $\mathcal{T}$.

$$
\begin{array}{lcll}
\mathcal{D} & : & \mathbf{Exp}_L \to \mathbf{Exp}_{L/N} & \\
\mathcal{D}[x] & = & x & \\
\mathcal{D}[\lambda x.\, M] & = & \lambda x.\, \mathcal{D}[M] & \\
\mathcal{D}[M\ N] & = & @_n\, \mathcal{D}[M]\, \mathcal{D}[N] & \text{if } N = x \text{ or } N = \lambda x.\, M' \\
\mathcal{D}[M\ N] & = & @_l\, \mathcal{D}[M]\, \mathcal{D}[N] & \text{if } N = M'\ N'
\end{array}
$$

Figure 12: A simple dememoization optimizer. Applications whose arguments do not require memoization are converted to call-by-name.

7.1. Eliminating Unnecessary Memoization

The transformation presented thus far performs excessive memoization. Every argument is memoized when in fact, for many arguments, memoization is unnecessary and may be safely elided. Eliding memoization of an argument corresponds to replacing call-by-need application with call-by-name application. We call this optimization *dememoization*.

Consider an intermediate language $\mathbf{Exp}_{L/N}$ with both call-by-need and call-by-name applications, written as $@_l\, M\ N$ and $@_n\, M\ N$ respectively. Then, a dememoization optimizer

$$\mathcal{D} : \mathbf{Exp}_L \to \mathbf{Exp}_{L/N}$$

maps each application $M\ N$ to $@_n\, M\ N$ whenever it is safe to do so (*i.e.*, whenever N does not require memoization), and to $@_l\, M\ N$ otherwise. How can we tell which arguments require memoization?

First of all, neither λ-abstractions nor variables require memoization—λ-abstractions because they are already in weak head-normal form (*i.e.*, are not evaluated further) and variables because they are bound to other arguments, which are themselves memoized as required. A simple version of $\mathcal{D}$ which takes only these two cases into account appears in Figure 12.

Next, unshared arguments do not require memoization. An argument is *shared* if its value is required (*i.e.*, if its thunk is forced) more than once. Conversely, an argument is *unshared* if its value is required at most once. If an unshared argument is ever evaluated, we know that its value will never be required again. Therefore, saving that value is pointless. Incorporating a sharing analysis [12] into $\mathcal{D}$ increases its effectiveness by allowing further dememoization for unshared arguments.

Taking advantage of this optimization requires only a slight modification to $\mathcal{T}$ to substitute **delay** for **delay*** in call-by-name applications. The new transformation is shown in Figure 13.

$$
\begin{array}{lcl}
\mathcal{T} & : & \mathbf{Exp}_{L/N} \rightarrow \mathbf{Exp}_{V+Thunk} \\
\mathcal{T}[x] & = & \mathbf{force}\, x \\
\mathcal{T}[\lambda x.\, M] & = & \lambda x.\, \mathcal{T}[M] \\
\mathcal{T}[@_n\, M\, N] & = & \mathcal{T}[M]\,(\mathbf{delay}\, \mathcal{T}[N]) \\
\mathcal{T}[@_l\, M\, N] & = & \mathcal{T}[M]\,(\mathbf{delay}^*\, \mathcal{T}[N])
\end{array}
$$

Figure 13: Introducing thunks after dememoization.

There are two further points to make about the dememoization of variable arguments. First, such arguments need not even be delayed. However, the special case $\mathcal{C}_{\underline{v}}[\mathbf{force}\, x] = x$ will ensure that doing so causes no overhead since $\mathcal{C}_{\overline{v}}[\mathbf{delay}\,(\mathbf{force}\, x)] = \mathcal{C}_{\overline{v}}[x]$. Second, it may occasionally be useful to memoize a variable argument. If the sharing analysis can show that an argument is unshared along some paths, it may be worthwhile to memoize the argument only along those paths where it might be shared. Burn *et al.* [7] call this dynamic marking of shared thunks *dashing*. Consider the following application

$$(\lambda x.\, g\, x\, (h\, x))\, M$$

where g does not share either of its arguments, but h might share its argument. Then $\mathcal{D}$ might produce the term

$$@_l(\lambda x.\, @_n(@_n g\, x)\,(@_n h\, x))\,\mathcal{D}[M]$$

However, if we further knew that g might use one of its arguments but not both, $\mathcal{D}$ might produce the term

$$@_n(\lambda x.\, @_n(@_n g\, x)\,(@_l h\, x))\,\mathcal{D}[M]$$

In the first case, the argument $\mathcal{D}[M]$ is memoized. In the second case, the argument $\mathcal{D}[M]$ is not memoized until and unless it is passed to h (as the variable x).

7.2. Strictness Optimizations

For certain functions, known as *strict* functions, it is safe to use call-by-value even when the program is being evaluated under call-by-need. Since modern compiler technology typically generates more efficient code for call-by-value than for call-by-need, a common optimization is to use call-by-value for those strict functions which are detected by strictness analysis [6, 20]. We now demonstrate how to incorporate this optimization into our transformation.

$$
\begin{array}{lcl}
\mathcal{T} & : & \mathbf{Exp}_{L/V} \rightarrow \mathbf{Exp}_{V+Thunk} \\
\mathcal{T}[x_l] & = & \mathbf{force}\, x \\
\mathcal{T}[\lambda x_l.\, M] & = & \lambda x.\, \mathcal{T}[M] \\
\mathcal{T}[@_l\, M\, N] & = & \mathcal{T}[M]\,(\mathbf{delay}^*\, \mathcal{T}[N]) \\
\mathcal{T}[x_v] & = & x \\
\mathcal{T}[\lambda x_v.\, M] & = & \lambda x.\, \mathcal{T}[M] \\
\mathcal{T}[@_v\, M\, N] & = & \mathcal{T}[M]\mathcal{T}[N] \\
\mathcal{T}[\uparrow M] & = & \lambda x.\, M\,(\mathbf{force}\, x) \\
\mathcal{T}[\downarrow M] & = & \lambda x.\, M\,(\mathbf{delay}\, x)
\end{array}
$$

Figure 14: Introducing thunks after strictness optimizations.

Consider a mixed λ-calculus with both call-by-need and call-by-value terms [2, 10], indicated by subscripts of l and v respectively, as well as coercions between the two. A grammar for such a language $\mathbf{Exp}_{L/V}$ is given below.

$$
\begin{array}{rll}
M = & x_l \mid \lambda x_l.\, M \mid @_l\, M_1\, M_2 & \text{call-by-need terms} \\
\mid & x_v \mid \lambda x_v.\, M \mid @_v\, M_1\, M_2 & \text{call-by-value terms} \\
\mid & \uparrow M \mid \downarrow M & \text{coercions}
\end{array}
$$

In this framework, a strictness optimizer

$$\mathcal{S} : \mathbf{Exp}_L \rightarrow \mathbf{Exp}_{L/V}$$

replaces call-by-need (lazy) terms with their call-by-value (strict) counterparts whenever it is provably safe to do so. Of course, this must be done in a consistent manner [10]. For instance, in $@_l\, M\, N$, M must be a lazy function, while in $@_v\, M\, N$, M must be a strict function.

Figure 14 gives the modifications to $\mathcal{T}$ required after strictness optimizations. Call-by-need terms are transformed in the usual way (by adding $\mathbf{delay}^*$'s and $\mathbf{force}$'s), while call-by-value terms are unchanged by the transformation (since the target language is also call-by-value).

$\uparrow M$ coerces the call-by-value function M into a call-by-need function, while $\downarrow M$ does the opposite. A few examples illustrate their use.

First, consider the function

$$f \;=\; \lambda g.\, g\,(M\, N)$$

f is strict in its argument g. However, without further information, we must assume that g is a lazy function, yielding the mixed term

$$\mathcal{S}[f] \;=\; \lambda g_v.\, @_l\, g_v\, \mathcal{S}[M\, N]$$

Note that although g_v is a strict variable, the value to which it is bound is a lazy function.

Now, what happens when f appears in a lazy context, for instance, as an argument to itself? We simply insert a coercion.

$$@_v \mathcal{S}[f] (\uparrow \mathcal{S}[f])$$

The key is that we are not prevented from making f a strict function in spite of the fact that it is used in a lazy context. The effect of this sort of coercion is to allow functions to be analyzed and optimized in isolation from the contexts in which the functions appear (though, of course, taking those contexts into account may result in better optimizations).

As a second example, consider the function

$$f \;=\; \lambda g.\, g\, (\lambda x.\, M)$$

We could of course transform this as

$$\mathcal{S}[f] \;=\; \lambda g_v.\, @_l\, g_v\, \mathcal{S}[\lambda x.\, M]$$

and coerce any strict functions h to which f is applied ($\uparrow \mathcal{S}[h]$). Alternatively, we could transform this as

$$\mathcal{S}[f] \;=\; \lambda g_v.\, @_v\, g_v\, \mathcal{S}[\lambda x.\, M]$$

and coerce any lazy functions h to which f is applied ($\downarrow \mathcal{S}[h]$). In general, the latter translation is unsafe whenever g might be bound to a lazy function because the argument is always evaluated even though it might not be needed. Here, however, the argument evaluates trivially (since it is a λ-abstraction) so evaluating it does no harm even when it is not needed by g. Other arguments that evaluate trivially, such as strict variables or lazy variables which are certain to have previously been forced (detectable via a path analysis [5]), may be treated similarly. Which translation is preferable depends on the functions to which f is applied. If f is usually applied to lazy functions, the first translation may be preferable, and vice versa.

When combining strictness optimizations with the dememoization optimizations of the previous section, the strictness optimizations are performed first. Dememoization is then applied, changing some of the remaining call-by-need applications to call-by-name. The reason for this ordering is that strictness optimizations can change the sharing behavior of a program, causing some shared arguments to become unshared (and hence candidates for dememoization). This is because a thunk which is passed to a strict function is forced once, whereas previously it may have been forced many

times. Of course, to be able to apply dememoization after strictness optimizations, $\mathcal{D}$ must be modified to operate on the appropriate terms, *i.e.*

$$\mathcal{D} \ : \ \mathbf{Exp}_{L/V} \ \rightarrow \ \mathbf{Exp}_{L/N/V}$$

$\mathcal{T}$ must be extended similarly.

$$\mathcal{T} \ : \ \mathbf{Exp}_{L/N/V} \ \rightarrow \ \mathbf{Exp}_{V+Thunk}$$

8. Putting It All Together

The complete transformation from $\mathbf{Exp}_L$ to $\mathbf{Exp}_{CPS+Store}$ is now given by

$$\mathcal{C}_{\underline{v}} \circ \mathcal{T} \circ \mathcal{D} \circ \mathcal{S}$$

Composing just the last two stages ($\mathcal{C}_{\underline{v}}$ and $\mathcal{T}$) yields the CPS transformation on mixed terms, $\mathcal{C}_{\underline{m}}$, shown in Figure 15.

Finally, let us point out that the correctness of the complete transformation depends on the correctness of $\mathcal{D}$ and $\mathcal{S}$. A proof of this correctness would probably follow the lines of Wand [29]. However, note that if $\mathcal{D}$ and $\mathcal{S}$ are identity transformations (*i.e.*, introduce no call-by-name or call-by-value terms), then the above specializes to $\mathcal{C}_{\underline{l}}$.

9. Related Work

This work arose from a desire to give a CPS presentation of graph reduction, particularly of the so-called "tagless" models, in which graphs are directly executed rather than interpreted [22, 18]. Most previous descriptions of graph reduction have been presented either informally or as state transition systems.

Once we developed our first, rudimentary call-by-need CPS transformation, the work by Danvy *et al.* [8, 9, 10] on the call-by-name CPS transformation proved invaluable in guiding its evolution.

Three other authors have investigated similar topics involving call-by-need and continuations. Josephs [17] presents a continuation semantics for a lazy functional language, while Jørgensen [16] rewrites an interpreter for a lazy language into CPS to achieve binding-time improvements for the purpose of partial evaluation. Both of these employ a different memoization technique from ours, in which an explicit tag distinguishes between evaluated and unevaluated expressions. Wang [30] describes several methods for implementing lazy evaluation in Scheme using `call/cc` to access the continuations, including one, which she calls a "status-checking-free implementation," that is similar to ours.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{C}_{\underline{m}} \;:\; \mathbf{Exp}_{L/N/V} \;\to\; \mathbf{Exp}_{CPS+Store}} \\
\mathcal{C}_{\underline{m}}[x_l] & = & x \\
\mathcal{C}_{\underline{m}}[\lambda x_l.\, M] & = & \underline{\lambda} c.\, \underline{@}\, c\, (\underline{\lambda} x.\, \mathcal{C}_{\underline{m}}[M]) \\
\mathcal{C}_{\underline{m}}[@_n\, M\, N] & = & \underline{\lambda} c.\, \overline{\underline{@}}\, \mathcal{C}_{\overline{m}}[M]\, (\overline{\lambda} m.\, \underline{@}\, (\underline{@}\, m\, \mathcal{C}_{\underline{m}}[N])\, c) \\
\mathcal{C}_{\underline{m}}[@_l\, M\, N] & = & \underline{\lambda} c.\, \overline{@}\, \mathcal{C}_{\overline{m}}[M]\, (\overline{\lambda} m.\, \underline{\mathbf{new}}\, (\underline{\lambda} r. \\
 & & \quad \underline{\mathbf{assign}}\, r\, (\underline{\lambda} c.\, \overline{@}\, \mathcal{C}_{\overline{m}}[N] (\overline{\lambda} n.\, \underline{\mathbf{assign}}\, r\, (\underline{\lambda} c.\, \underline{@}\, c\, n)\, (\underline{@}\, c\, n))) \\
 & & \quad (\underline{@}\, (\underline{@}\, m\, (\underline{\lambda} c.\, \underline{\mathbf{deref}}\, r\, (\underline{\lambda} t.\, \underline{@}\, t\, c)))\, c))) \\
\mathcal{C}_{\underline{m}}[x_v] & = & \underline{\lambda} c.\, \underline{@}\, c\, x \\
\mathcal{C}_{\underline{m}}[\lambda x_v.\, M] & = & \underline{\lambda} c.\, \underline{@}\, c\, (\underline{\lambda} x.\, \mathcal{C}_{\underline{m}}[M]) \\
\mathcal{C}_{\underline{m}}[@_v\, M\, N] & = & \underline{\lambda} c.\, \overline{@}\, \mathcal{C}_{\overline{m}}[M]\, (\overline{\lambda} m.\, \overline{@}\, \mathcal{C}_{\overline{m}}[N]\, (\overline{\lambda} n.\, \underline{@}\, (\underline{@}\, m\, n)\, c)) \\
\mathcal{C}_{\underline{m}}[\uparrow M] & = & \underline{\lambda} c.\, \underline{@}\, c\, (\underline{\lambda} x.\, \underline{\lambda} c.\, \overline{@}\, \mathcal{C}_{\overline{m}}[M]\, (\overline{\lambda} m.\, \underline{@}\, x\, (\underline{\lambda} n.\, \underline{@}\, (\underline{@}\, m\, n)\, c))) \\
\mathcal{C}_{\underline{m}}[\downarrow M] & = & \underline{\lambda} c.\, \underline{@}\, c\, (\underline{\lambda} x.\, \underline{\lambda} c.\, \overline{@}\, \mathcal{C}_{\overline{m}}[M]\, (\overline{\lambda} m.\, \underline{@}\, (\underline{@}\, m\, (\underline{\lambda} c'.\, \underline{@}\, c'\, x))\, c)) \\
 & & \\
\multicolumn{3}{l}{\mathcal{C}_{\overline{m}} \;:\; \mathbf{Exp}_{L/N/V} \to (\mathbf{Exp}_{CPS+Store} \to \mathbf{Exp}_{CPS+Store}) \to \mathbf{Exp}_{CPS+Store}} \\
\mathcal{C}_{\overline{m}}[x_l] & = & \overline{\lambda} k.\, \underline{@}\, x\, (\underline{\lambda} a.\, \overline{@}\, k\, a) \\
\mathcal{C}_{\overline{m}}[\lambda x_l.\, M] & = & \overline{\lambda} k.\, \overline{@}\, k\, (\underline{\lambda} x.\, \mathcal{C}_{\underline{m}}[M]) \\
\mathcal{C}_{\overline{m}}[@_n\, M\, N] & = & \overline{\lambda} k.\, \overline{@}\, \mathcal{C}_{\overline{m}}[M]\, (\overline{\lambda} m.\, \underline{@}\, (\underline{@}\, m\, \mathcal{C}_{\underline{m}}[N])\, (\underline{\lambda} a.\, \overline{@}\, k\, a)) \\
\mathcal{C}_{\overline{m}}[@_l\, M\, N] & = & \overline{\lambda} k.\, \overline{@}\, \mathcal{C}_{\overline{m}}[M]\, (\overline{\lambda} m.\, \underline{\mathbf{new}}\, (\underline{\lambda} r. \\
 & & \quad \underline{\mathbf{assign}}\, r\, (\underline{\lambda} c.\, \overline{@}\, \mathcal{C}_{\overline{m}}[N] (\overline{\lambda} n.\, \underline{\mathbf{assign}}\, r\, (\underline{\lambda} c.\, \underline{@}\, c\, n)\, (\underline{@}\, c\, n))) \\
 & & \quad (\underline{@}\, (\underline{@}\, m\, (\underline{\lambda} c.\, \underline{\mathbf{deref}}\, r\, (\underline{\lambda} t.\, \underline{@}\, t\, c)))\, (\underline{\lambda} a.\, \overline{@}\, k\, a)))) \\
\mathcal{C}_{\overline{m}}[x_v] & = & \overline{\lambda} k.\, \overline{@}\, k\, x \\
\mathcal{C}_{\overline{m}}[\lambda x_v.\, M] & = & \overline{\lambda} k.\, \overline{@}\, k\, (\underline{\lambda} x.\, \mathcal{C}_{\underline{m}}[M]) \\
\mathcal{C}_{\overline{m}}[@_v\, M\, N] & = & \overline{\lambda} k.\, \overline{@}\, \mathcal{C}_{\overline{m}}[M]\, (\overline{\lambda} m.\, \overline{@}\, \mathcal{C}_{\overline{m}}[N]\, (\overline{\lambda} n.\, \underline{@}\, (\underline{@}\, m\, n)\, (\underline{\lambda} a.\, \overline{@}\, k\, a))) \\
\mathcal{C}_{\overline{m}}[\uparrow M] & = & \overline{\lambda} k.\, \overline{@}\, k\, (\underline{\lambda} x.\, \underline{\lambda} c.\, \overline{@}\, \mathcal{C}_{\overline{m}}[M]\, (\overline{\lambda} m.\, \underline{@}\, x\, (\underline{\lambda} n.\, \underline{@}\, (\underline{@}\, m\, n)\, c))) \\
\mathcal{C}_{\overline{m}}[\downarrow M] & = & \overline{\lambda} k.\, \overline{@}\, k\, (\underline{\lambda} x.\, \underline{\lambda} c.\, \overline{@}\, \mathcal{C}_{\overline{m}}[M]\, (\overline{\lambda} m.\, \underline{@}\, (\underline{@}\, m\, (\underline{\lambda} c'.\, \underline{@}\, c'\, x))\, c))
\end{array}
$$

Figure 15: The CPS transformation on mixed terms.

10. Conclusions and Future Work

In this paper, we have presented a call-by-need CPS transformation and explored several of its variations, culminating in one that produces no administrative redexes and takes advantage of both strictness and sharing information. There are a number of possible directions for further work.

An obvious application of these ideas is in the area of compiler construction. One approach is to use our transformation in the front-end of a CPS-based compiler for a lazy functional language. One could then use an existing CPS-based back-end that supports storage operations, even one originally developed for a call-by-value language! Our early experience with implementations of the techniques presented here gives us reason to believe that this approach is viable.

A second area for future investigation is the incorporation of other optimizations and analyses into our transformation. Strictness and sharing optimizations fit quite cleanly into this framework and it would be interesting to see if other optimizations do as well.

Acknowledgements

We are grateful to Mitch Wand, whose keen insight clarified much of our thinking. We also wish to thank Olivier Danvy for his suggestions and encouragement.

References

1. Appel, Andrew W. and Jim, Trevor. Continuation-passing, closure-passing style. In *Sixteenth ACM Symposium on Principles of Programming Languages* (1989) 293–302.

2. Asperti, Andrea. Integrating strict and lazy evaluation: the λ_{sl}-calculus. In Deransart, P. and Małuszyński, J., editors, *Programming Language Implementation and Logic Programming, Sweden*, Springer-Verlag (1990) 238–254.

3. Augusteijn, A. and van der Hoeven, G. Combinatorgraphs as self-reducing programs. (1984). Unpublished workshop presentation.

4. Augustsson, Lennart. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology (1987).

5. Bloss, Adrienne, Hudak, Paul, and Young, Jonathan. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation*, 1 (1988) 147–164.

6. Burn, Geoffrey L., Hankin, Chris, and Abramsky, Samson. Strictness analysis of higher-order functions. *Science of Computer Programming*, 7 (1986) 249–278.

7. Burn, Geoffrey L., Peyton Jones, Simon L., and Robson, John D. The Spineless G-Machine. In *Proceedings of the ACM Conference on Lisp and Functional Programming, Snowbird* (1988) 244–258.

8. Danvy, Olivier and Filinski, Andrzej. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2, 4 (December 1992) 361–391.

9. Danvy, Olivier and Hatcliff, John. Thunks (continued). In *Proceedings of the Workshop on Static Analysis WSA'92*, IRISA, Rennes, France, Bordeaux, France (September 1992) 3–11. Extended version available as Technical Report CIS-92-28, Kansas State University.

10. Danvy, Olivier and Hatcliff, John. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1, 3 (1993). To appear.

11. Futamura, Yoshihito. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2, 5 (1971) 45–50.

12. Goldberg, Benjamin. Detecting sharing of partial applications in functional programs. In Kahn, Gilles, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture, Portland*, Springer-Verlag (1987) 408–425.

13. Hughes, John. *The Design and Implementation of Programming Languages*. PhD thesis, Oxford University (1983).

14. Johnsson, Thomas. Lambda lifting: transforming programs to recursive equations. In Jouannaud, J.-P., editor, *Conference on Functional Programming Languages and Computer Architecture, Nancy*, Springer-Verlag (September 1985) 190–203.

15. Johnsson, Thomas. *Compiling Lazy Functional Languages*. PhD thesis, Department of Computer Sciences, Chalmers University of Technology (1987).

16. Jørgensen, Jesper. Generating a compiler for a lazy language by partial evaluation. In *Symposium on Principles of Programming Languages* (January 1992) 258–268.

17. Josephs, Mark B. The semantics of lazy functional languages. *Theoretical Computer Science*, 68 (1989) 105–111.

18. Koopman, Philip J., Lee, Peter, and Siewiorek, Daniel. Cache behavior of combinator graph reduction. *ACM Transactions on Programming Languages and Systems*, 14, 2 (April 1992) 265–297.

19. Kranz, David, Kelsey, Richard, Rees, Jonathan, Hudak, Paul, Philbin, James, and Adams, Norman. Orbit: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction* (July 1986) 219–233.

20. Mycroft, Alan. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the 4th International Symposium on Programming*, Springer-Verlag (1980) 269–281.

21. Nielson, Flemming and Nielson, Hanne Riis. Two-level semantics and code generation. *Theoretical Computer Science*, 56, 1 (January 1988) 59–133.

22. Peyton Jones, Simon L. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2, 2 (April 1992) 127–202.

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

24. Reynolds, John C. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM National Conference, New York, New York* (1972) 717–740.

25. Steele Jr., Guy L. *Rabbit: a compiler for Scheme*. Technical Report AI-TR-474, MIT (1978).

26. Turner, David A. A new implementation technique for applicative languages. *Software — Practice and Experience*, 9, 1 (January 1979) 31–49.

27. Wadsworth, Christopher P. *Semantics and Pragmatics of The Lambda Calculus*. PhD thesis, University of Oxford (1971).

28. Wand, Mitchell. A short proof of the lexical addressing algorithm. *Information Processing Letters*, 35 (1990) 1–5.

29. Wand, Mitchell. Specifying the correctness of binding-time analysis. In *Symposium on Principles of Programming Languages* (January 1993) 137–143.

30. Wang, Ching-lin. Obtaining lazy evaluation with continuations in Scheme. *Information Processing Letters*, 35 (1990) 93–97.

A. Grammars

$$\begin{array}{l}
\mathbf{Exp}_L, \mathbf{Exp}_N, \mathbf{Exp}_V, \mathbf{Exp}_{CPS} : \\
\quad M = \quad x \mid \lambda x.\, M \mid M_1\, M_2 \\
\mathbf{Exp}_{L/N} : \\
\quad M = \quad x \mid \lambda x.\, M \mid @_l\, M_1\, M_2 \mid @_n\, M_1\, M_2 \\
\mathbf{Exp}_{L/V} : \\
\quad M = \quad x_l \mid \lambda x_l.\, M \mid @_l\, M_1\, M_2 \\
\qquad \mid \quad x_v \mid \lambda x_v.\, M \mid @_v\, M_1\, M_2 \\
\qquad \mid \quad \uparrow M \mid \downarrow M \\
\mathbf{Exp}_{L/N/V} : \\
\quad M = \quad x_l \mid \lambda x_l.\, M \mid @_l\, M_1\, M_2 \mid @_n\, M_1\, M_2 \\
\qquad \mid \quad x_v \mid \lambda x_v.\, M \mid @_v\, M_1\, M_2 \\
\qquad \mid \quad \uparrow M \mid \downarrow M \\
\mathbf{Exp}_{V+Thunk} : \\
\quad M = \quad x \mid \lambda x.\, M \mid M_1\, M_2 \\
\qquad \mid \quad \mathbf{force}\, M \mid \mathbf{delay}\, M \mid \mathbf{delay}^*\, M \\
\mathbf{Exp}_{CPS+Store} : \\
\quad M = \quad x \mid \lambda x.\, M \mid M_1\, M_2 \\
\qquad \mid \quad \mathbf{new}\, M \mid \mathbf{deref}\, x\, M \mid \mathbf{assign}\, x\, M_1\, M_2
\end{array}$$