
Thunks (continued) *

Olivier Danvy, John Hatcliff

*Department of Computing and Information Sciences
Kansas State University
Manhattan, Kansas 66506, USA
e-mail: (danvy, hatcliff)@cis.ksu.edu*

Abstract: Call-by-name can be simulated in a call-by-value setting using “thunks” (*i.e.*, parameterless procedures) or *continuation-passing-style* (CPS). In this paper we uncover a relationship between the two simulations. We prove that the call-by-value CPS transformation \mathcal{C}_v , when applied to a term $\mathcal{T}(t)$ which simulates call-by-name using thunks, yields a term identical to the call-by-name CPS transformation \mathcal{C}_n applied directly to t (modulo renaming):

$$\mathcal{C}_v \circ \mathcal{T} \equiv \mathcal{C}_n$$

This result sheds new light on the call-by-name CPS transformation — it can be factored into two conceptually distinct steps:

- the suspension of argument evaluation (captured in \mathcal{T});
- the sequentialization of function application to give the usual tail-calls of CPS terms (captured in \mathcal{C}_v).

Keywords: λ -calculus, call-by-name, call-by-value, continuation-passing style transformation.

1. Introduction

Among many possible implementations of call-by-name in Algol 60, one still stands the test of time: Ingerman’s “thunks” [8]. A thunk forms the representation of a function argument. As it turns out, Ingerman’s implementational view of thunks coincides with the representation of a parameterless procedure.

*Proceedings of the Second International Workshop on Static Analysis WSA’92, volume 81-82 of Bigre Journal, pages 3-11, Bordeaux, France, September 1992. IRISA, Rennes, France. This work was partly supported by NSF under grant CCR-9102625.

Consequently, parameterless procedures (traditionally referred to as “thunks”) have become part of the programmer’s toolbox to implement call-by-name in a call-by-value language such as Standard ML or Scheme [1, 3]. Using side effects, thunks also enable one to implement call-by-need (as in the Yale implementations of lazy functional languages [2]).

In fact, thunks can be introduced systematically by wrapping each actual parameter in a parameterless procedure. Let \mathcal{T} denote this “thunkifying” program transformation.

Call-by-name (CBN) can also be simulated under call-by-value with continuation-passing-style (CPS). This simulation was formalized by Plotkin in the early ’70s [9]. The call-by-name CPS transformation \mathcal{C}_n is a more drastic program transformation than \mathcal{T} in that it changes the structure of a term more radically. \mathcal{C}_n explicitly encodes the CBN evaluation strategy into the transformed term, using functions called “continuations”. Continuations express how values will be used during the evaluation of the “rest of the program”.

Correspondingly, call-by-value (CBV) can also be simulated under call-by-name with CPS. The CBV CPS transformation \mathcal{C}_v also explicitly encodes the CBV evaluation strategy into the transformed term, using continuations.

In fact, \mathcal{C}_n and \mathcal{C}_v are best known to yield terms that are independent of the evaluation order [10]. They are used in many applications, *e.g.*, to compile eager functional programs [1, 11].

One may wonder, however, about a connection between the introduction of continuations and the introduction of thunks. We have found that the call-by-name CPS transformation is the composition of the thunkifier with the call-by-value CPS transformation:

$$\mathcal{C}_n \equiv \mathcal{C}_v \circ \mathcal{T}$$

This result sheds new light on \mathcal{C}_n — it can be factored into two conceptually distinct steps:

- the suspension of argument evaluation (captured in \mathcal{T});
- the linearization of function application actions to give the usual tail-calls of CPS terms (captured in \mathcal{C}_v).

This factorization is illustrated in Figures 1, 2, and 3. Figure 1 displays an uncurried version of the S combinator in Scheme. Figure 2 displays the

```

;;; [A * B -> C] * [A -> B] * A -> C
(lambda (f g x)
  (f x (g x)))

```

Figure 1. Uncurried S combinator in Scheme

```

(lambda (f g x)
  ((f) (lambda ()
        (x))
   (lambda ()
     ((g) (lambda ()
           (x))))))

```

Figure 2. Thunkified counterpart of Figure 1

```

(lambda (k0)
  (k0 (lambda (f g x k1)
        (f (lambda (v0)
            (v0 (lambda (k2)
                  (x k2)))
              (lambda (k3)
                (g (lambda (v1)
                     (v1 (lambda (k4)
                           (x k4))
                         k3))))
              k1))))))

```

Figure 3. CBN CPS-counterpart of Figure 1 & CBV CPS-counterpart of Figure 2

“thunkified” version of the uncurried S combinator. Figure 3 displays the CBN CPS counterpart of Figure 1 *and* the CBV CPS counterpart of Figure 2. The alert reader will identify many possible improvements to the expressions in Figures 2 and 3, all of them based on strictness properties.

1.1. Overview

Section 2 presents the transformations \mathcal{T} , \mathcal{C}_n , and \mathcal{C}_v . Section 3 formally establishes the connection between the two simulations \mathcal{T} and \mathcal{C}_n . This is done by composing \mathcal{C}_v and \mathcal{T} symbolically. We show that the composed transformation is identical to \mathcal{C}_n . Finally, Section 5 concludes.

2. The Program Transformations

2.1. The source language

We consider the pure λ -calculus generalized with uncurried functions (as in Scheme). The BNF of λ -terms is defined as follows.

$$e ::= x \mid \lambda(x_1, \dots, x_m).e \mid e_0(e_1, \dots, e_m)$$

NB: For conciseness, we forgo using parentheses for unary functions:

$$\begin{aligned} \lambda(x).... &= \lambda x.... \\ f(x) &= f x \end{aligned}$$

2.2. A thunkifier

Terms in the source language can be “thunkified” automatically. Figure 4 gives the definition of the thunkifier function \mathcal{T} . This figure contains two essential points:

- *identifiers* are mapped into parameterless applications;
- *function arguments* are mapped into parameterless procedures (thunks).

Otherwise, \mathcal{T} is structure preserving. In the extended version of this paper [5], we prove the correctness of \mathcal{T} formally in that it effectively simulates call-by-name in a call-by-value setting.

$$\begin{aligned}
\mathcal{T}[[x]] &= x () \\
\mathcal{T}[[\lambda(x_1, \dots, x_m).e]] &= \lambda(x_1, \dots, x_m). \mathcal{T}[[e]] \\
\mathcal{T}[[e_0(e_1, \dots, e_m)]] &= \mathcal{T}[[e_0]](\lambda(). \mathcal{T}[[e_1]], \dots, \lambda(). \mathcal{T}[[e_m]])
\end{aligned}$$

Figure 4. Thunkifier for simulating call-by-name under call-by-value

2.3. The call-by-name CPS transformation

Figure 5 gives Plotkin's call-by-name CPS transformation \mathcal{C}_n extended to n -ary functions [9].

$$\begin{aligned}
\mathcal{C}_n[[x]] &= x \\
\mathcal{C}_n[[\lambda(x_1, \dots, x_m).e]] &= \lambda \kappa. \kappa(\lambda(x_1, \dots, x_m, k). \mathcal{C}_n[[e]] k) \\
\mathcal{C}_n[[e_0(e_1, \dots, e_m)]] &= \lambda \kappa. \mathcal{C}_n[[e_0]](\lambda f. f(\mathcal{C}_n[[e_1]], \dots, \mathcal{C}_n[[e_m]], \kappa))
\end{aligned}$$

Figure 5. Call-by-name CPS transformation of λ -terms

The result of transforming a term e into CPS in an empty context is given by

$$\mathcal{C}_n[[e]](\lambda a. a)$$

2.4. Call-by-value CPS transformation

Figure 6 gives Plotkin's call-by-value CPS transformation \mathcal{C}_v extended to n -ary functions [9].

The result of transforming a term e into CPS in an empty context is given by

$$\mathcal{C}_v[[e]](\lambda a. a)$$

$$\begin{aligned}
\mathcal{C}_v[x] &= \lambda \kappa . \kappa x \\
\mathcal{C}_v[\lambda(x_1, \dots, x_m).e] &= \lambda \kappa . \kappa (\lambda(x_1, \dots, x_m, k) . \mathcal{C}_v[e] k) \\
\mathcal{C}_v[e_0(e_1, \dots, e_m)] &= \\
&\lambda \kappa . \mathcal{C}_v[e_0] \\
&\quad (\lambda f . \mathcal{C}_v[e_1] \\
&\quad\quad (\lambda v_1 . \dots \mathcal{C}_v[e_m] \\
&\quad\quad\quad (\lambda v_m . f(v_1, \dots, v_m, \kappa)) \dots))
\end{aligned}$$

Figure 6. Call-by-value CPS transformation of λ -terms

3. Relating the Call-by-Name Simulations

We now derive a new CPS transformation \mathcal{C}_t by composing \mathcal{C}_v with \mathcal{T} symbolically. We will make use of the following lemma.

Lemma 1 $\mathcal{C}_v[\lambda().e_i](\lambda v_i \dots) = (\lambda v_i \dots)(\mathcal{C}_v[e_i])$

Proof:

$$\begin{aligned}
\mathcal{C}_v[\lambda().e_i](\lambda v_i \dots) &= (\lambda v_i \dots)(\lambda k . \mathcal{C}_v[e_i] k) \\
&= (\lambda v_i \dots)(\mathcal{C}_v[e_i]) \quad \dots\eta\text{-reduction}
\end{aligned}$$

The η -reduction step is valid because evaluating $\mathcal{C}_v[e_i]$ terminates. \square

The derivation for each construct follows:

- $$\begin{aligned}
\bullet \mathcal{C}_t[x] &= \mathcal{C}_v \circ \mathcal{T}[x] \\
&= \mathcal{C}_v[x()] \\
&= \lambda \kappa . \mathcal{C}_v[x](\lambda f . f \kappa) \\
&= \lambda \kappa . x \kappa \\
&= x
\end{aligned}$$
- $$\begin{aligned}
\bullet \mathcal{C}_t[\lambda(x_1, \dots, x_m).e] &= \mathcal{C}_v \circ \mathcal{T}[\lambda(x_1, \dots, x_m).e] \\
&= \mathcal{C}_v[\lambda(x_1, \dots, x_m). \mathcal{T}[e]] \\
&= \lambda \kappa . \kappa (\lambda(x_1, \dots, x_m, k) . (\mathcal{C}_v \circ \mathcal{T}[e]) k) \\
&= \lambda \kappa . \kappa (\lambda(x_1, \dots, x_m, k) . \mathcal{C}_t[e] k)
\end{aligned}$$

$$\begin{aligned}
& \bullet \mathcal{C}_t[e_0(e_1, \dots, e_m)] \\
&= \mathcal{C}_v \circ \mathcal{T}[e_0(e_1, \dots, e_m)] \\
&= \mathcal{C}_v[\mathcal{T}[e_0](\lambda().\mathcal{T}[e_1], \dots, \lambda().\mathcal{T}[e_m])] \\
&= \lambda\kappa.(\mathcal{C}_v \circ \mathcal{T}[e_0])(\lambda f. \mathcal{C}_v[\lambda().\mathcal{T}[e_1]] \\
&\quad (\lambda v_1. \dots \mathcal{C}_v[\lambda().\mathcal{T}[e_m]] \\
&\quad \quad (\lambda v_m. f(v_1, \dots, v_m, \kappa)) \dots)) \\
&= \lambda\kappa.(\mathcal{C}_v \circ \mathcal{T}[e_0])(\lambda f. f(\mathcal{C}_v \circ \mathcal{T}[e_1], \dots \mathcal{C}_v \circ \mathcal{T}[e_m], \kappa)) \\
&\quad \dots \text{Lemma 1 and } \beta\text{-reduction (m times)} \\
&= \lambda\kappa. \mathcal{C}_t[e_0](\lambda f. f(\mathcal{C}_t[e_1], \dots \mathcal{C}_t[e_m], \kappa))
\end{aligned}$$

Theorem 1 $\mathcal{C}_n \equiv \mathcal{C}_v \circ \mathcal{T}$.

Proof: By inspection, \mathcal{C}_n and \mathcal{C}_t are identical. When applied to identical terms they yield identical terms (modulo renaming). \square

4. Related Work

Continuations are seeing a renewed interest today in their applications to logic, constructive mathematics, programming languages, and programming [6]. This paper builds on Plotkin's fundamental work on CPS [9].

Thunks were introduced to implement parameter transmission in Algol 60 [8]. Today they are a common programming tool to simulate call-by-name or call-by-need in a call-by-value setting [2].

To the best of our knowledge, no connection has been drawn between CPS and thunks so far.

5. Conclusion

This paper stems from the serendipitous observation that the call-by-name CPS transformation yields the same result as composing the call-by-value CPS transformation with a thunkifier. We have formalized and proven this coincidence. Let us now analyze it.

The two flavors of the CPS transformation (*i.e.*, \mathcal{C}_v and \mathcal{C}_n) act alike in that they both yield evaluation-order independent and continuation-passing λ -terms. The fact that each function is not only applied to its regular arguments, but also to a continuation — a representation of how succeeding

evaluation will use the value produced by the function — leads to the tail-call property of CPS terms.

\mathcal{C}_v and \mathcal{C}_n act differently because of the different treatment of function arguments in the CBN and CBV strategies. CBV functions must be applied to values only and thus the CBV CPS transformation explicitly encodes the evaluation of all function arguments before the function is applied. Under CBN, there is no distinction between arbitrary argument expressions and values — all evaluation of arguments is suspended. Thus each argument is explicitly encoded as a suspended computation that simply needs a continuation before evaluation can proceed. This leads to the evaluation-strategy independence of CPS terms.

The crux of the matter is that the thunking transformation explicitly encodes the CBN treatment of arguments as suspensions into the term. Since a suspension (*i.e.*, a thunk) coincides with the CBV notion of value, \mathcal{C}_v has nothing further to do than to add the tail-call property (via continuations).

Conversely, the existence of a Direct Style transformer \mathcal{D}_v , *i.e.*, of a program transformation that is the inverse of \mathcal{C}_v [4], suggests another way to discover the staging of \mathcal{C}_n . The idea is to map a λ_n -term into CPS, and, based on the evaluation-order independence of CPS, map this term back to CBV direct style. This transformation naturally thunkifies the original λ -term.

$$\begin{aligned} \mathcal{C}_n &= \mathcal{C}_v \circ \mathcal{T} \\ \Rightarrow \mathcal{D}_v \circ \mathcal{C}_n &= \mathcal{D}_v \circ \mathcal{C}_v \circ \mathcal{T} \\ \Rightarrow \mathcal{D}_v \circ \mathcal{C}_n &= \mathcal{T} \end{aligned}$$

We are currently investigating the connection of this staging with Filinski's duality between CBN and CBV [7].

Acknowledgements

Thanks are due to Julia L. Lawall for commenting an earlier version of this paper.

References

- [1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- [2] Adrienne Bloss, Paul Hudak, and Jonathan Young. Code optimization for lazy evaluation. *LISP and Symbolic Computation*, 1:147–164, 1988.
- [3] William Clinger and Jonathan Rees (editors). Revised⁴ report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.
- [4] Olivier Danvy. Back to direct style. In Bernd Krieg-Brückner, editor, *Proceedings of the Fourth European Symposium on Programming*, number 582 in Lecture Notes in Computer Science, pages 130–150, Rennes, France, February 1992.
- [5] Olivier Danvy and John Hatcliff. Thunks (continued). Technical Report CIS-92-28, Kansas State University, Manhattan, Kansas, 1992.
- [6] Olivier Danvy and Carolyn L. Talcott, editors. *Proceedings of the ACM SIGPLAN Workshop on Continuations*, Technical report STAN-CS-92-1426, Stanford University, San Francisco, California, June 1992.
- [7] Andrzej Filinski. Declarative continuations: An investigation of duality in programming language semantics. In David H. Pitt et al., editors, *Category Theory and Computer Science*, number 389 in Lecture Notes in Computer Science, pages 224–249, Manchester, UK, September 1989.
- [8] Peter Z. Ingerman. Thunks, a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961.
- [9] Gordon D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [10] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, 1972.
- [11] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.