BRICS

Basic Research in Computer Science

A New One-Pass Transformation into Monadic Normal Form

Olivier Danvy

BRICS Report Series **RS-02-52**

ISSN 0909-0878 **December 2002**

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/02/52/`

A New One-Pass Transformation into Monadic Normal Form *

Olivier Danvy

BRICS †
Department of Computer Science
University of Aarhus ‡

December 2002

Abstract

We present a translation from the call-by-value λ-calculus to monadic normal forms that includes short-cut boolean evaluation. The translation is higher-order, operates in one pass, duplicates no code, generates no chains of thunks, and is properly tail recursive. It makes a crucial use of symbolic computation at translation time.

*To appear in the proceedings of CC 2003.

†Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.

‡Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
E-mail: `danvy@brics.dk`

Contents

1 Introduction

Program transformation and code generators offer typical situations where symbolic computation makes it possible to merge several passes into one. The CPS transformation is a canonical example: it transforms a term in direct style into one in continuation-passing style (CPS) [39, 43]. It appears in several Scheme compilers, including the first one [30, 33, 42], where it is used in two passes: one for the transformation proper and one for the simplifications entailed by the transformation (the so-called "administrative redexes"). One-pass versions have been developed that perform administrative reductions at transformation time [2, 15, 48]. They form one of the first, if not the first, instances of higher-order and natively executable two-level specifications.

The notion of binding times was discovered early by Jones and Muchnick [27] in the context of programming languages. Later it proved instrumental for partial evaluation [28], for program analysis [37], and for code generation [50]. It was then soon noticed that two-level specifications (i.e., 'staged' [29], or 'binding-time separated' [35], or again 'binding-time analyzed' [25] specifications) were directly expressible in languages such as Lisp and Scheme that offer quasiquote and unquote—a metalinguistic capability that has since been rediscovered in 'C [19], cast in a typed setting in MetaML [45], and connected both to modal logic [18] and to temporal logic [17]. In Lisp, quasiquote and unquote are used chiefly to write macros [5], an early example of symbolic computation during code generation [32]. In partial evaluation [10, 26], two-level specifications are called 'generating extensions'. Nesting quasiquote and unquote yields macros that generate macros and multi-level generating extensions.

The goal of this article is to present a one-pass transformer into monadic normal forms [23, 36] that performs short-cut boolean evaluation, duplicates no code, generates no chains of thunks, and is properly tail recursive. We consider the following source language:

$$\begin{array}{rcl}
\Lambda^E \ni e & ::= & \ell \mid x \mid \lambda x.e \mid e\,e \mid \textsf{if}\ b\ \textsf{then}\ e\ \textsf{else}\ e \\
\Lambda^B \ni b & ::= & e \mid b \wedge b \mid b \vee b \mid \neg b \mid \textsf{if}\ b\ \textsf{then}\ b\ \textsf{else}\ b
\end{array}$$

We translate programs in this source language into programs in the following target language:

$$\begin{array}{rcl}
\Lambda^C_{ml} \ni c & ::= & \textsf{return}\ v \mid \\
 & & \textsf{let}\ x = v\,v\ \textsf{in}\ c \mid v\,v \mid \\
 & & \textsf{if}\ v\ \textsf{then}\ c\ \textsf{else}\ c \mid \\
 & & \textsf{let}\ x = \lambda().c\ \textsf{in}\ c \mid x\,() \\
\Lambda^V_{ml} \ni v & ::= & \ell \mid x \mid \lambda x.c
\end{array}$$

The source language is that of the call-by-value λ-calculus with literals, conditional expressions, and computational effects. The target language is that of monadic normal forms (sometimes called A-normal forms [21]), with a syntactic separation between computations (c, the serious expressions) and values

(v, the trivial expressions), as traditional since Reynolds and Moggi [36, 41]. The return production is the unit and the first let production is the bind of monadic style [47]. Computations are carried out by applications, which can either be named with a let expression or occur in tail position. Conditional expressions exclusively occur in tail position. The last two productions specify the declaration and activation of thunks, which are used to ensure that no code is duplicated.

For example, a source term such as

$$\lambda x.g_0\ (h_0\ (\mathsf{if}\ (g_1\ (h_1\ x)) \vee x\ \mathsf{then}\ g_2\ (h_2\ x)\ \mathsf{else}\ x))$$

is translated into the following target term (automatically pretty printed in Standard ML for clarity), in one pass.

```
return (fn x => let val k0 = fn w1 => let val w2 = h0 w1
                                      in g0 w2
                                      end
                    val t5 = fn () => let val w3 = h2 x
                                          val w4 = g2 w3
                                      in k0 w4
                                      end
                    val w6 = h1 x
                    val w7 = g1 w6

                in if w7
                   then t5 ()
                   else if x
                        then t5 ()
                        else k0 x
                end)
```

In this target term, the source context $g_0\ (h_0\ [\cdot])$ is translated into the function `k0`, where the outside call occurs tail recursively. Because of the disjunction in the test, a thunk `t5` is created for the then branch. In this thunk, the outside call occurs tail recursively. The composition of g_1 and h_1 is sequentialized and its result is tested. If it holds true, `t5` is activated; otherwise, the second half of the disjunction is tested. If it holds true, `t5` is activated (the code for `t5` is shared). Otherwise, the value of `x` is passed to the (sequentialized) composition of g_0 and h_0. Free variables (i.e., g_0, h_0, g_1, h_1, g_2, and h_2) have been translated to themselves (i.e., `g0`, `h0`, `g1`, `h1`, `g2`, and `h2`, respectively).

Monadic normal forms offer the main advantages of CPS (i.e., all intermediate results are named and their computation is sequentialized),[1] and they have been used in compilers for functional languages [6,7,21–23,38,40,46]. Therefore, a one-pass transformation into monadic normal form with short-cut boolean evaluation could well be of practical use (i.e., outside academia).

The rest of this article is organized as follows. We present a standard, two-pass translation from the source language to the target language (Section 2),

[1] The jury is still out about the other advantages of CPS [40].

and then its one-pass counterpart (Section 3). We then illustrate it (Section 4), assess it (Section 5), and then review related work and conclude (Section 6).

2 A Standard, Two-Pass Translation

The first part of the translation is simple enough: it is the standard encoding of the call-by-value λ-calculus into the computational metalanguage, straightforwardly extended to handle conditional expressions.

$$
\begin{array}{rcl}
\mathcal{E}_v[\![\ell]\!] & = & \mathsf{return}\,\ell \\
\mathcal{E}_v[\![x]\!] & = & \mathsf{return}\,x \\
\mathcal{E}_v[\![\lambda x.e]\!] & = & \mathsf{return}\,\lambda x.\mathcal{E}_v[\![e]\!] \\
\mathcal{E}_v[\![e_0\; e_1]\!] & = & \mathsf{let}\; w_0 = \mathcal{E}_v[\![e_0]\!]\;\mathsf{in}\;\mathsf{let}\; w_1 = \mathcal{E}_v[\![e_1]\!]\;\mathsf{in}\; w_0\; w_1 \\
\mathcal{E}_v[\![\mathsf{if}\; b\;\mathsf{then}\; e_1\;\mathsf{else}\; e_0]\!] & = & \mathsf{if}\;\mathcal{B}_v[\![b]\!]\;\mathsf{then}\;\mathcal{E}_v[\![e_1]\!]\;\mathsf{else}\;\mathcal{E}_v[\![e_0]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{B}_v[\![e]\!] & = & \mathcal{E}_v[\![e]\!] \\
\mathcal{B}_v[\![b_1 \wedge b_2]\!] & = & \mathsf{if}\;\mathcal{B}_v[\![b_1]\!]\;\mathsf{then}\;\mathcal{B}_v[\![b_2]\!]\;\mathsf{else}\;\mathit{false} \\
\mathcal{B}_v[\![b_1 \vee b_2]\!] & = & \mathsf{if}\;\mathcal{B}_v[\![b_1]\!]\;\mathsf{then}\;\mathit{true}\;\mathsf{else}\;\mathcal{B}_v[\![b_2]\!] \\
\mathcal{B}_v[\![\neg b]\!] & = & \mathsf{if}\;\mathcal{B}_v[\![b]\!]\;\mathsf{then}\;\mathit{false}\;\mathsf{else}\;\mathit{true} \\
\mathcal{B}_v[\![\mathsf{if}\; b_2\;\mathsf{then}\; b_1\;\mathsf{else}\; b_0]\!] & = & \mathsf{if}\;\mathcal{B}_v[\![b_2]\!]\;\mathsf{then}\;\mathcal{B}_v[\![b_1]\!]\;\mathsf{else}\;\mathcal{B}_v[\![b_0]\!]
\end{array}
$$

The second pass of the translation consists in performing monadic simplifications [24] and in unnesting conditional expressions until the simplified term belongs to Λ^C_{ml}.

3 A One-Pass Translation

In this section, we build on the full one-pass transformation into monadic normal form for the call-by-value λ-calculus:

$$
\begin{array}{rcl}
\mathcal{E} & : & \Lambda^E \rightarrow \Lambda^C_{ml} \\
\mathcal{E}[\![\ell]\!] & = & \underline{\mathsf{return}}\,\ell \\
\mathcal{E}[\![x]\!] & = & \underline{\mathsf{return}}\,x \\
\mathcal{E}[\![\lambda x.e]\!] & = & \underline{\mathsf{return}}\,\underline{\lambda} x.\mathcal{E}[\![e]\!] \\
\mathcal{E}[\![e_0\; e_1]\!] & = & \mathcal{E}_c[\![e_0]\!]\;\overline{\lambda} v_0.\mathcal{E}_c[\![e_1]\!]\;\overline{\lambda} v_1.v_0 \,\underline{@}\, v_1 \\
 & & \\
\mathcal{E}_c & : & \Lambda^E \rightarrow (\Lambda^V_{ml} \rightarrow \Lambda^C_{ml}) \rightarrow \Lambda^C_{ml} \\
\mathcal{E}_c[\![\ell]\!]\;\kappa & = & \kappa \,\overline{@}\, \ell \\
\mathcal{E}_c[\![x]\!]\;\kappa & = & \kappa \,\overline{@}\, x \\
\mathcal{E}_c[\![\lambda x.e]\!]\;\kappa & = & \kappa \,\overline{@}\, \underline{\lambda} x.\mathcal{E}[\![e]\!] \\
\mathcal{E}_c[\![e_0\; e_1]\!]\;\kappa & = & \mathcal{E}_c[\![e_0]\!]\;\overline{\lambda} v_0.\mathcal{E}_c[\![e_1]\!]\;\overline{\lambda} v_1.\underline{\mathsf{let}}\; w = v_0 \,\underline{@}\, v_1 \;\underline{\mathsf{in}}\; \kappa \,\overline{@}\, w
\end{array}
$$

The function $\mathcal{E}$ is applied to subterms occurring in tail position, and the function $\mathcal{E}_c$ to the other subterms; it is indexed with a functional accumulator κ.[2] This transformation is higher-order (witness the type of $\mathcal{E}_c$) and it is also two level: the underlined terms are hygienic syntax constructors and the overlined terms are reduced at transformation time (@ denotes infix application). We show in appendix how to program it in ML. This transformation is similar to a higher-order one-pass CPS transformation, which can be transformationally derived from a two-pass specification [16].

The question now is to generalize this one-pass transformation to the full Λ^E and Λ^B from Section 1. Our insight is to index the translation of each boolean expression with the translation of the corresponding consequent and alternative. Each of them can be the name of a thunk, which we can use non-linearly, or a thunk, which we should only use linearly since we want to avoid code duplication. Enumerating, we define four translation functions for boolean expressions:

$$\begin{array}{lcl}
\mathcal{B}_{cc} & : & \Lambda^B \to (1 \to \Lambda^C_{ml}) \times (1 \to \Lambda^C_{ml}) \to \Lambda^C_{ml} \\
\mathcal{B}_{vv} & : & \Lambda^B \to \Lambda^V_{ml} \times \Lambda^V_{ml} \to \Lambda^C_{ml} \\
\mathcal{B}_{cv} & : & \Lambda^B \to (1 \to \Lambda^C_{ml}) \times \Lambda^V_{ml} \to \Lambda^C_{ml} \\
\mathcal{B}_{vc} & : & \Lambda^B \to \Lambda^V_{ml} \times (1 \to \Lambda^C_{ml}) \to \Lambda^C_{ml}
\end{array}$$

The problem then reduces to following the structure of the boolean expressions and introducing residual let expressions to name computations if their result needs to be used more than once.

$$\begin{array}{rcl}
\mathcal{B}_{cc} & : & \Lambda^B \to (1 \to \Lambda^C_{ml}) \times (1 \to \Lambda^C_{ml}) \to \Lambda^C_{ml} \\
\mathcal{B}_{cc}[\![b_1 \wedge b_2]\!]\,\langle \kappa_1, \kappa_0 \rangle & = & \underline{\mathsf{let}}\; t_0 = \underline{\lambda}().\kappa_0 \;\overline{@}\; () \\
 & & \underline{\mathsf{in}}\; \mathcal{B}_{cv}[\![b_1]\!]\,\langle \overline{\lambda}().\mathcal{B}_{cv}[\![b_2]\!]\,\langle \kappa_1, t_0 \rangle, t_0 \rangle \\
\mathcal{B}_{cc}[\![b_1 \vee b_2]\!]\,\langle \kappa_1, \kappa_0 \rangle & = & \underline{\mathsf{let}}\; t_1 = \underline{\lambda}().\kappa_1 \;\overline{@}\; () \\
 & & \underline{\mathsf{in}}\; \mathcal{B}_{vc}[\![b_1]\!]\,\langle t_1, \overline{\lambda}().\mathcal{B}_{vc}[\![b_2]\!]\,\langle t_1, \kappa_0 \rangle \rangle \\
\mathcal{B}_{cc}[\![\neg b]\!]\,\langle \kappa_1, \kappa_0 \rangle & = & \mathcal{B}_{cc}[\![b]\!]\,\langle \kappa_0, \kappa_1 \rangle \\
\mathcal{B}_{cc}[\![\mathsf{if}\; b_2 \;\mathsf{then}\; b_1 \;\mathsf{else}\; b_0]\!]\,\langle \kappa_1, \kappa_0 \rangle & = & \underline{\mathsf{let}}\; t_1 = \underline{\lambda}().\kappa_1 \;\overline{@}\; \underline{()} \\
 & & \underline{\mathsf{in}}\; \underline{\mathsf{let}}\; t_0 = \underline{\lambda}().\kappa_0 \;\overline{@}\; () \\
 & & \quad \underline{\mathsf{in}}\; \mathcal{B}_{cc}[\![b_2]\!]\,\langle \overline{\lambda}().\mathcal{B}_{vv}[\![b_1]\!]\,\langle t_1, t_0 \rangle, \\
 & & \qquad\qquad \overline{\lambda}().\mathcal{B}_{vv}[\![b_0]\!]\,\langle t_1, t_0 \rangle \rangle
\end{array}$$

For example, let us consider $\mathcal{B}_{cc}[\![b_1 \wedge b_2]\!]\,\langle \kappa_1, \kappa_0 \rangle$, i.e., the translation of a conjunction in the presence of two thunks κ_1 and κ_0. The activation of κ_1 and κ_0 will yield the translation of the consequent and of the alternative of this

[2] We refrain from referring to κ as a continuation since it is not applied tail recursively.

conjunction. Naively, we could want to define the translation as follows:

$$\mathcal{B}_{cc}[\![b_1]\!]\,\langle\overline{\lambda}().\mathcal{B}_{cc}[\![b_2]\!]\,\langle\kappa_1,\kappa_0\rangle,\kappa_0\rangle$$

Doing so, however, would duplicate κ_0, i.e., the translation of the alternative of the conjunction. Therefore we name its result with a let. The rest of the translation follows the same spirit.

$$\begin{array}{rcl}
\mathcal{B}_{vv} & : & \Lambda^B \to \Lambda^V_{ml} \times \Lambda^V_{ml} \to \Lambda^C_{ml} \\
\mathcal{B}_{vv}[\![b_1 \wedge b_2]\!]\,\langle v_1, v_0\rangle & = & \mathcal{B}_{cv}[\![b_1]\!]\,\langle\overline{\lambda}().\mathcal{B}_{vv}[\![b_2]\!]\,\langle v_1, v_0\rangle, v_0\rangle \\
\mathcal{B}_{vv}[\![b_1 \vee b_2]\!]\,\langle v_1, v_0\rangle & = & \mathcal{B}_{vc}[\![b_1]\!]\,\langle v_1, \overline{\lambda}().\mathcal{B}_{vv}[\![b_2]\!]\,\langle v_1, v_0\rangle\rangle \\
\mathcal{B}_{vv}[\![\neg b]\!]\,\langle v_1, v_0\rangle & = & \mathcal{B}_{vv}[\![b]\!]\,\langle v_0, v_1\rangle \\
\mathcal{B}_{vv}[\![\mathsf{if}\ b_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_0]\!]\,\langle v_1, v_0\rangle & = & \mathcal{B}_{cc}[\![b_2]\!]\,\langle\overline{\lambda}().\mathcal{B}_{vv}[\![b_1]\!]\,\langle v_1, v_0\rangle, \\
 & & \qquad \overline{\lambda}().\mathcal{B}_{vv}[\![b_0]\!]\,\langle v_1, v_0\rangle\rangle \\
 & & \\
\mathcal{B}_{cv} & : & \Lambda^B \to (1 \to \Lambda^C_{ml}) \times \Lambda^V_{ml} \to \Lambda^C_{ml} \\
\mathcal{B}_{cv}[\![b_1 \wedge b_2]\!]\,\langle \kappa_1, v_0\rangle & = & \mathcal{B}_{cv}[\![b_1]\!]\,\langle\overline{\lambda}().\mathcal{B}_{cv}[\![b_2]\!]\,\langle \kappa_1, v_0\rangle, v_0\rangle \\
\mathcal{B}_{cv}[\![b_1 \vee b_2]\!]\,\langle \kappa_1, v_0\rangle & = & \underline{\mathsf{let}}\ t_1 = \underline{\lambda}().\kappa_1\ \overline{@}\ () \\
 & & \underline{\mathsf{in}}\ \mathcal{B}_{vc}[\![b_1]\!]\,\langle t_1, \overline{\lambda}().\mathcal{B}_{vv}[\![b_2]\!]\,\langle t_1, v_0\rangle\rangle \\
\mathcal{B}_{cv}[\![\neg b]\!]\,\langle \kappa_1, v_0\rangle & = & \mathcal{B}_{vc}[\![b]\!]\,\langle v_0, \kappa_1\rangle \\
\mathcal{B}_{cv}[\![\mathsf{if}\ b_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_0]\!]\,\langle \kappa_1, v_0\rangle & = & \underline{\mathsf{let}}\ t_1 = \underline{\lambda}().\kappa_1\ \overline{@}\ () \\
 & & \underline{\mathsf{in}}\ \mathcal{B}_{cc}[\![b_2]\!]\,\langle\overline{\lambda}().\mathcal{B}_{vv}[\![b_1]\!]\,\langle t_1, v_0\rangle, \\
 & & \qquad \overline{\lambda}().\mathcal{B}_{vv}[\![b_0]\!]\,\langle t_1, v_0\rangle\rangle \\
 & & \\
\mathcal{B}_{vc} & : & \Lambda^B \to \Lambda^V_{ml} \times (1 \to \Lambda^C_{ml}) \to \Lambda^C_{ml} \\
\mathcal{B}_{vc}[\![b_1 \wedge b_2]\!]\,\langle v_1, \kappa_0\rangle & = & \underline{\mathsf{let}}\ t_0 = \underline{\lambda}().\kappa_0\ \overline{@}\ () \\
 & & \underline{\mathsf{in}}\ \mathcal{B}_{cv}[\![b_1]\!]\,\langle\overline{\lambda}().\mathcal{B}_{vv}[\![b_2]\!]\,\langle v_1, t_0\rangle, t_0\rangle \\
\mathcal{B}_{vc}[\![b_1 \vee b_2]\!]\,\langle v_1, \kappa_0\rangle & = & \mathcal{B}_{vc}[\![b_1]\!]\,\langle v_1, \overline{\lambda}().\mathcal{B}_{vc}[\![b_2]\!]\,\langle v_1, \kappa_0\rangle\rangle \\
\mathcal{B}_{vc}[\![\neg b]\!]\,\langle v_1, \kappa_0\rangle & = & \mathcal{B}_{cv}[\![b]\!]\,\langle \kappa_0, v_1\rangle \\
\mathcal{B}_{vc}[\![\mathsf{if}\ b_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_0]\!]\,\langle v_1, \kappa_0\rangle & = & \underline{\mathsf{let}}\ t_0 = \underline{\lambda}().\kappa_0\ \overline{@}\ () \\
 & & \underline{\mathsf{in}}\ \mathcal{B}_{cc}[\![b_2]\!]\,\langle\overline{\lambda}().\mathcal{B}_{vv}[\![b_1]\!]\,\langle v_1, t_0\rangle, \\
 & & \qquad \overline{\lambda}().\mathcal{B}_{vv}[\![b_0]\!]\,\langle v_1, t_0\rangle\rangle
\end{array}$$

As for the connection between translating a boolean expression and translating an expression, we make it using a functional accumulator that will generate a conditional expression when it is applied.

$$
\begin{array}{rcl}
\mathcal{B}_{cc}[\![e]\!]\,\langle\kappa_1,\kappa_0\rangle & = & \mathcal{E}_c[\![e]\!]\,\overline{\lambda} v.\underline{\mathsf{if}}\ v\ \underline{\mathsf{then}}\ \kappa_1\ \overline{@}\ ()\ \underline{\mathsf{else}}\ \kappa_0\ \overline{@}\ ()\\
\mathcal{B}_{vv}[\![e]\!]\,\langle v_1,v_0\rangle & = & \mathcal{E}_c[\![e]\!]\,\overline{\lambda} v.\underline{\mathsf{if}}\ v\ \underline{\mathsf{then}}\ v_1\ \underline{@}\ ()\ \underline{\mathsf{else}}\ v_0\ \underline{@}\ ()\\
\mathcal{B}_{cv}[\![e]\!]\,\langle\kappa_1,v_0\rangle & = & \mathcal{E}_c[\![e]\!]\,\overline{\lambda} v.\underline{\mathsf{if}}\ v\ \underline{\mathsf{then}}\ \kappa_1\ \overline{@}\ ()\ \underline{\mathsf{else}}\ v_0\ \underline{@}\ ()\\
\mathcal{B}_{vc}[\![e]\!]\,\langle v_1,\kappa_0\rangle & = & \mathcal{E}_c[\![e]\!]\,\overline{\lambda} v.\underline{\mathsf{if}}\ v\ \underline{\mathsf{then}}\ v_1\ \underline{@}\ ()\ \underline{\mathsf{else}}\ \kappa_0\ \overline{@}\ ()
\end{array}
$$

Finally we connect translating an expression and translating a boolean expression as follows.

$$
\begin{array}{rcl}
\mathcal{E}[\![\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0]\!] & = & \mathcal{B}_{cc}[\![b]\!]\,\langle\overline{\lambda}().\mathcal{E}[\![e_1]\!],\overline{\lambda}().\mathcal{E}[\![e_0]\!]\rangle\\
\mathcal{E}_c[\![\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0]\!]\ \kappa & = & \underline{\mathsf{let}}\ k=\underline{\lambda} w.\kappa\ \overline{@}\ w\\
 & & \underline{\mathsf{in}}\ \mathcal{B}_{cc}[\![b]\!]\,\langle\overline{\lambda}().\mathcal{E}_v[\![e_1]\!]\ k,\overline{\lambda}().\mathcal{E}_v[\![e_0]\!]\ k\rangle\\
 & & \\
\mathcal{E}_v & : & \Lambda^E\to\Lambda^V_{ml}\to\Lambda^C_{ml}\\
\mathcal{E}_v[\![\ell]\!]\ k & = & k\ \underline{@}\ \ell\\
\mathcal{E}_v[\![x]\!]\ k & = & k\ \underline{@}\ x\\
\mathcal{E}_v[\![\lambda x.e]\!]\ k & = & k\ \underline{@}\ \underline{\lambda} x.\mathcal{E}[\![e]\!]\\
\mathcal{E}_v[\![e_0\ e_1]\!]\ k & = & \mathcal{E}_c[\![e_0]\!]\ \overline{\lambda} v_0.\mathcal{E}_c[\![e_1]\!]\ \overline{\lambda} v_1.\underline{\mathsf{let}}\ w=v_0\ \underline{@}\ v_1\ \underline{\mathsf{in}}\ k\ \underline{@}\ w\\
\mathcal{E}_v[\![\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0]\!]\ k & = & \mathcal{B}_{cc}[\![b]\!]\,\langle\overline{\lambda}().\mathcal{E}_v[\![e_1]\!]\ k,\overline{\lambda}().\mathcal{E}_v[\![e_0]\!]\ k\rangle
\end{array}
$$

In the second equation, a let expression is inserted to name the context (and to avoid its duplication). $\mathcal{E}_v$ is there to avoid generating chains of thunks when translating nested conditional expressions.

The result can be directly coded in ML (see appendix): the source and target languages are implemented as data types and the translation as a function. A side benefit of using ML is that its type inferencer acts as a theorem prover to tell us that the translation maps terms from the source language into terms in the target language (a bit more reasoning, however, is necessary to show that the translation generates no chains of thunks). Finally, since the translation is specified compositionally, it does operate in one pass.

4 Two Examples

4.1 No chains of thunks

The term $\lambda x.g\ (h\ (\mathsf{if}\ a\ \mathsf{then\ if}\ b_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_0\ \mathsf{else}\ x))$ is translated into the following target term in one pass.

```
return (fn x => let val k0 = fn v1 => let val v2 = h v1
                                      in g v2
                                      end
                in if a
                   then if b2
                        then k0 b1
                        else k0 b0
                   else k0 x
                end)
```

Each conditional branch directly calls `k0`.

4.2 Short-cut boolean evaluation

The term $\lambda x.\mathsf{if}\ a_1 \wedge a_2 \wedge a_3 \wedge a_4\ \mathsf{then}\ x\ \mathsf{else}\ g\ (h\ x)$ is translated into the following target term in one pass.

```
return (fn x => let val f1 = fn () => let val v0 = h x
                                      in g v0
                                      end
                in if a1
                   then if a2
                        then if a3
                             then if a4
                                  then return x
                                  else f1 ()
                             else f1 ()
                        else f1 ()
                   else f1 ()
                end)
```

All the else branches directly call `f1`.

5 Assessment

A similar development yields, mutatis mutandis, a CPS transformation that is higher-order, operates in one pass, duplicates no code, generates no chain of thunks, and is properly tail recursive.

The author has implemented both transformations in his academic Scheme compiler. Their net effect is to fuse two compiler passes into one and to avoid, in effect, an entire copy of the source program. In particular, an escape analysis of the transformations themselves shows that all of their higher-order functions are stack-allocatable [4]. The transformations therefore have a minimal footprint in that they only allocate heap space to construct their result, making them well suited in a JIT situation.

6 Related Work, Conclusion, and Future Work

We have presented a two-level program transformation that encodes call-by-value λ-terms into monadic normal form and achieves short-cut boolean evaluation. The transformation operates in one pass in that it directly constructs the normal form without intermediate representations that need further processing. As usual with two-level specifications, erasing all overlines and underlines yields something meaningful—here an interpreter for the call-by-value λ-calculus in the monadic metalanguage.

The program transformation can be easily adapted to other evaluation orders.

Short-cut evaluation is a standard topic in compiling [1,9,34]. The author is not aware of any treatment of it in one-pass CPS transformations or in one-pass transformations into monadic normal form.

Our use of higher-order functions and of an underlying evaluator to fuse a transformation and a form of normalization is strongly reminiscent of the notion of *normalization by evaluation* [8,11,13,20]. And indeed the author is convinced that the present one-pass transformation could be specified as a formal instance of normalization by evaluation—a future work.

Monadic normal forms and CPS terms are in one-to-one correspondence [12], and Kelsey and Appel have noticed the correspondence between continuation-passing style and static single assignment form (SSA) [3, 31]. Therefore, the one-pass transformation with short-cut boolean evaluation should apply directly to the SSA transformation [49]—another future work.

Acknowledgments: Thanks are due to Mads Sig Ager, Jacques Carette, Samuel Lindley, and the anonymous reviewers for comments.

This work is supported by the ESPRIT Working Group APPSEM II (`http://www.tcs.informatik.uni-muenchen.de/~mhofmann/appsem2/`).

A Two-Level Programming in ML

We briefly outline how to program the one-pass translation of Section 2 [14].

First, we assume a type for identifiers as well as a module generating fresh identifiers in the target abstract syntax:

```
type ide = string

signature GENSYM = sig
                     val init : unit -> unit
                     val new : string -> ide
                   end
```

Given this type, the source and the target abstract syntax (without conditional expressions) are defined with two data types:

```
structure Source = struct
                     datatype e = VAR of ide
                                | LAM of ide * e
                                | APP of e * e
                   end

structure Target = struct
                     datatype e = RETURN of t
                                | TAIL_APP of t * t
                                | LET_APP of ide * (t * t) * e
                          and t = VAR of ide
                                | LAM of ide * e
                   end
```

Given a structure `Gensym : GENSYM`, the two translation functions $\mathcal{E}$ and $\mathcal{E}_c$ are recursively defined as two ML functions `trans0` and `trans1`. In particular, `trans1` is uncurried and higher order. For readability of the output, the main translation function `trans` initializes the generator of fresh identifiers before calling `trans0`:

```
(*  trans0 : Source.e -> Target.e                           *)
(*  trans1 : Source.e * (Target.t -> Target.e) -> Target.e  *)
fun trans0 (Source.VAR x)
    = Target.RETURN (Target.VAR x)
  | trans0 (Source.LAM (x, e))
    = Target.RETURN (Target.LAM (x, trans0 e))
  | trans0 (Source.APP (e0, e1))
    = trans1 (e0,
              fn v0 => trans1 (e1,
                               fn v1 => Target.TAIL_APP (v0, v1)))
and trans1 (Source.VAR x, k)
    = k (Target.VAR x)
  | trans1 (Source.LAM (x, e), k)
    = k (Target.LAM (x, trans0 e))
  | trans1 (Source.APP (e0, e1), k)
    = trans1 (e0,
              fn v0 => trans1 (e1,
                               fn v1 => let val v = Gensym.new "v"
                                        in Target.LET_APP
                                             (v,
                                              (v0, v1),
                                              k (Target.VAR v))
                                        end))

(*  trans : Source.e -> Target.e  *)
fun trans e
    = (Gensym.init (); trans0 e)
```

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* World Student Series. Addison-Wesley, Reading, Massachusetts, 1986.

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, 1992.

[3] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, April 1998.

[4] Anindya Banerjee and David A. Schmidt. Stackability in the typed call-by-value lambda calculus. *Science of Computer Programming*, 31(1):47–73, 1998.

[5] Alan Bawden. Quasiquotation in Lisp. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical report BRICS-NS-99-1, University of Aarhus, pages 4–12, San Antonio, Texas, January 1999. Available online at `http://www.brics.dk/~pepm99/programme.html`.

[6] Nick Benton and Andrew Kennedy. Monads, effects, and transformations. In *Third International Workshop on Higher-Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*, pages 19–31, Paris, France, September 1999.

[7] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java byte-codes. In Paul Hudak and Christian Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 129–140, Baltimore, Maryland, September 1998. ACM Press.

[8] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer-Verlag, 1998.

[9] Keith Clarke. One-pass code generation using continuations. *Software—Practice and Experience*, 19(12):1175–1192, 1989.

[10] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[11] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[12] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[13] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[14] Olivier Danvy. Programming techniques for partial evaluation. In Friedrich L. Bauer and Ralf Steinbrüggen, editors, *Foundations of Secure Computation*, NATO Science series, pages 287–318. IOS Press Ohmsha, 2000.

[15] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

[16] Olivier Danvy and Lasse R. Nielsen. A first-order one-pass CPS transformation. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002*, number 2303 in Lecture Notes in Computer Science, pages 98–113, Grenoble, France, April 2002. Springer-Verlag. Extended version available as the technical report BRICS RS-01-49. To appear in TCS.

[17] Rowan Davies. A temporal-logic approach to binding-time analysis. In Edmund M. Clarke, editor, *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[18] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Steele Jr. [44], pages 258–283.

[19] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoe. ‘C: A language for high-level, efficient, and machine-independent dynamic code generation. In Steele Jr. [44], pages 131–144.

[20] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer-Verlag.

[21] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN’93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

[22] Matthew Fluet and Stephen Weeks. Contification using dominators. In Xavier Leroy, editor, *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 36, No. 10, pages 2–13, Firenze, Italy, September 2001. ACM Press.

[23] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 458–471, Portland, Oregon, January 1994. ACM Press.

[24] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, pages 507–541, 1997. Extended version available as the technical report BRICS RS-96-34.

[25] Neil D. Jones. Tutorial on binding time analysis. In Paul Hudak and Neil D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, New Haven, Connecticut, June 1991. ACM Press.

[26] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London, UK, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/`.

[27] Neil D. Jones and Steven S. Muchnick. Some thoughts towards the design of an ideal language. In Susan L. Graham, editor, *Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 77–94. ACM Press, January 1976.

[28] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[29] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In Mark Scott Johnson and Ravi Sethi, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 86–96, St. Petersburg, Florida, January 1986. ACM Press.

[30] Richard A. Kelsey. *Compilation by Program Transformation*. PhD thesis, Computer Science Department, Yale University, New Haven, Connecticut, May 1989. Research Report 702.

[31] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In Michael Ernst, editor, *ACM SIGPLAN Workshop on Intermediate Representations*, SIGPLAN Notices, Vol. 30, No 3, pages 13–22, San Francisco, California, January 1995. ACM Press.

[32] Oleg Kiselyov. Macros that compose: Systematic macro programming. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the 2002 ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, number 2487 in Lecture Notes in Computer Science, pages 202–217, Pittsburgh, Pennsylvania, October 2002. Springer-Verlag.

[33] David A. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Computer Science Department, Yale University, New Haven, Connecticut, February 1988. Research Report 632.

[34] George Logothetis and Prateek Mishra. Compiling short-circuit boolean expressions in one pass. *Software—Practice and Experience*, 11:1197–1214, 1981.

[35] Torben Æ. Mogensen. Separating binding times in language specifications. In Joseph E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 14–25, London, England, September 1989. ACM Press.

[36] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[37] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[38] Dino P. Oliva and Andrew P. Tolmach. From ML to Ada: strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

[39] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[40] John Reppy. Optimizing nested loops using local CPS conversion. *Higher-Order and Symbolic Computation*, 15(2/3), 2002. To appear.

[41] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[42] Guy L. Steele Jr. Lambda, the ultimate declarative. AI Memo 379, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, November 1976.

[43] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.

[44] Guy L. Steele Jr., editor. *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996. ACM Press.

[45] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, Portland, Oregon, 1999. CSE-99-TH-002.

[46] David Tarditi, Greg Morrisett, Perry Cheng, and Chris Stone. TIL: a type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 31, No 5, pages 181–192. ACM Press, June 1996.

[47] Philip Wadler. The essence of functional programming (invited talk). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

[48] Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Mathematical Foundations of Programming Semantics*, number 598 in Lecture Notes in Computer Science, pages 294–311, Pittsburgh, Pennsylvania, March 1991. Springer-Verlag. 7th International Conference.

[49] Mark N. Wegman and F. Ken Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 3(2):181–210, 1991.

[50] Zhe Yang. *Language Support for Program Generation: Reasoning, Implementation, and Applications*. PhD thesis, Computer Science Department, New York University, New York, New York, August 2001.

Recent BRICS Report Series Publications

RS-02-52 Olivier Danvy. ***A New One-Pass Transformation into Monadic Normal Form*****. December 2002. 16 pp. To appear in Hedin, editor,** ***Compiler Construction, 12th International Conference*****, CC '03 Proceedings, LNCS, 2003.**

RS-02-51 Gerth Stølting Brodal, Rolf Fagerberg, Anna Östlin, Christian N. S. Pedersen, and S. Srinivasa Rao. ***Computing Refined Buneman Trees in Cubic Time*****. December 2002. 14 pp.**

RS-02-50 Kristoffer Arnsfelt Hansen, Peter Bro Miltersen, and V. Vinay. ***Circuits on Cylinders*****. December 2002. 16 pp.**

RS-02-49 Mikkel Nygaard and Glynn Winskel. ***HOPLA—A Higher-Order Process Language*****. December 2002. 18 pp. Appears in Brim, Jančar, Křetínský and Antonín, editors,** ***Concurrency Theory: 13th International Conference*****, CONCUR '02 Proceedings, LNCS 2421, 2002, pages 434–448.**

RS-02-48 Mikkel Nygaard and Glynn Winskel. ***Linearity in Process Languages*****. December 2002. 27 pp. Appears in Plotkin, editor,** ***Seventeenth Annual IEEE Symposium on Logic in Computer Science*****, LICS '02 Proceedings, 2002, pages 433–446.**

RS-02-47 Zoltán Ésik. ***Extended Temporal Logic on Finite Words and Wreath Product of Monoids with Distinguished Generators*****. December 2002. 16 pp. To appear in** ***6th International Conference, Developments in Language Theory*****, DLT '02 Revised Papers, LNCS, 2002.**

RS-02-46 Zoltán Ésik and Hans Leiß. ***Greibach Normal Form in Algebraically Complete Semirings*****. December 2002. 43 pp. An extended abstract appears in Bradfield, editor,** ***European Association for Computer Science Logic: 16th International Workshop*****, CSL '02 Proceedings, LNCS 2471, 2002, pages 135–150.**

RS-02-45 Jesper Makholm Byskov. ***Chromatic Number in Time $O(2.4023^n)$ Using Maximal Independent Sets*****. December 2002. 6 pp.**