Compiling Strict Functional Languages using Continuation Passing Style

Anthony M. Sloane

Programming Languages Research Group
Department of Computing
Macquarie University

`Anthony.Sloane@mq.edu.au`, `@inkytonik`

A Comprehensive Account

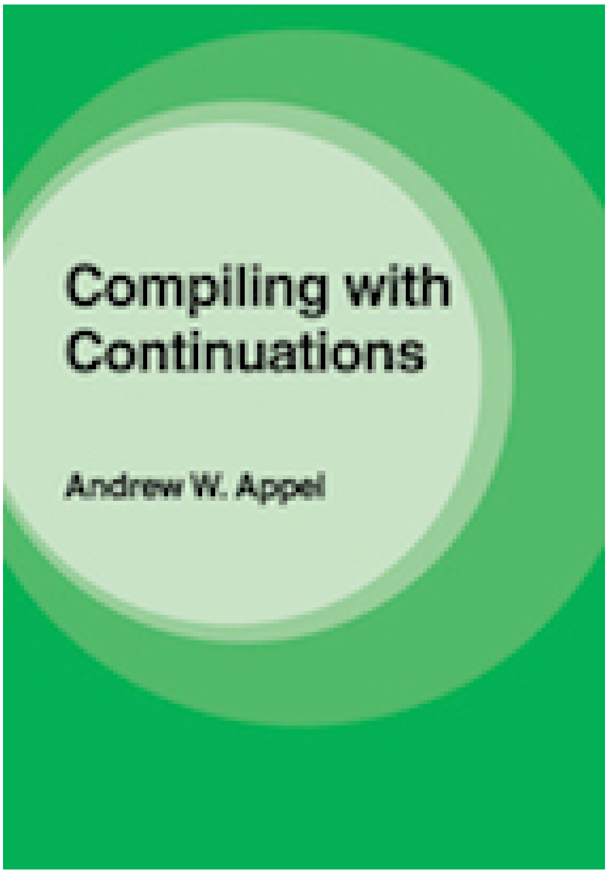
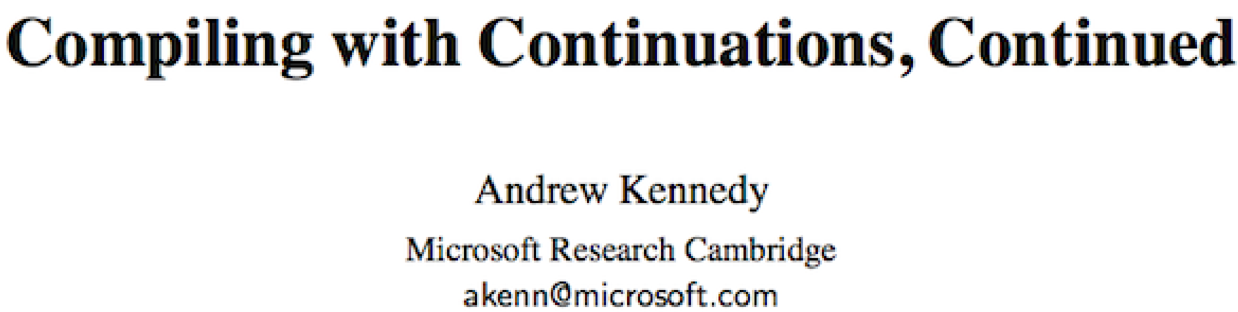


Figure: Cambridge University Press, 1992

ICFP 2007



Compiling with Continuations, Continued

Andrew Kennedy

Microsoft Research Cambridge

akenn@microsoft.com

Figure: A More Recent View

Standard ML Fragment

```
> 42
42

> 2 + 3
5

> let val x = 1 in x + 2
3

> if true then 3 else 4
3

> (fn x => x + 1) (5)
6
```

Standard ML Fragment

$$
\begin{aligned}
\mathrm{ML} \ni e \quad ::= \quad & x \mid e\,e' \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid \texttt{(}e\texttt{,}e'\texttt{)} \mid \texttt{\#}i\ e \mid \texttt{()} \\
& \mid \texttt{in}i\ e \mid \texttt{let val}\ x\ \texttt{=}\ e\ \texttt{in}\ e'\ \texttt{end} \\
& \mid \texttt{case}\ e\ \texttt{of in1}\ x_1\ \texttt{=>}\ e_1 \mid \texttt{in2}\ x_2\ \texttt{=>}\ e_2
\end{aligned}
$$

We assume a datatype declared by

```
datatype ('a,'b) sum = in1 of 'a | in2 of 'b
```

Figure: ML Terms

Continuation Passing Style (1)

```
42              letval $0 = 42 in
                  halt $0

2 + 3           letval $5 = 2 in
                  letval $6 = 3 in
                    letprim $4 = IntAdd $5 $6 in
                      halt $4

let x = 1 in    letcont c1 x = letval $2 = 2 in
  x + 2                          letprim $1 = IntAdd x $2 in
                                   halt $1 in
                  letval $3 = 1 in
                    c1 $3
```

Continuation Passing Style (2)

```
if true then 3 else 4
```

```
letval $7 = true in
  letcont c3 _ = letval $5 = 3 in
                 halt $5 in
    letcont c4 _ = letval $6 = 4 in
                   halt $6 in
      case $7 in c3 c4
```

Continuation Passing Style (3)

```
((x : Int) => x + 1) (5)
```

```
letfun $1 c2 x = letval $3 = 1 in
                   letprim $2 = IntAdd x $3 in
                     c2 $2 in
  letval $4 = 5 in
    letcont c1 $0 = halt $0 in
      $1 c1 $4
```

Continuation Passing Style (4)

$$
\begin{array}{llll}
\text{(terms)} & \mathrm{CTm} \ni K, L & ::= & \mathsf{letval}\ x = V\ \mathsf{in}\ K \\
 & & \mid & \mathsf{let}\ x = \pi_i\, x\ \mathsf{in}\ K \\
 & & \mid & \mathsf{letcont}\ k\ x = K\ \mathsf{in}\ L \\
 & & \mid & k\ x \\
 & & \mid & f\ k\ x \\
 & & \mid & \mathsf{case}\ x\ \mathsf{of}\ k_1 \,[]\, k_2 \\
\text{(values)} & \mathrm{CVal} \ni V, W & ::= & () \mid (x, y) \mid \mathsf{in}_i\, x \mid \lambda k\, x.K
\end{array}
$$

Figure: CPS Terms

Runtime Representations

Runtime values: $r ::= () \mid (r_1, r_2) \mid \mathsf{in}_i\, r \mid \langle \rho, \lambda k\, x.K \rangle$
Continuation values: $c ::= \langle \rho, \lambda x.K \rangle$
Environments: $\rho ::= \bullet \mid \rho, x \mapsto r \mid \rho, k \mapsto c$

Interpretation of values:

$$
\begin{array}{rcl@{\qquad}rcl}
[\![()]\!]\,\rho & = & () & [\![(x,y)]\!]\,\rho & = & (\rho(x), \rho(y)) \\
[\![\mathsf{in}_i\, V]\!]\,\rho & = & \mathsf{in}_i\,(\rho(x)) & [\![\lambda k\, x.K]\!]\,\rho & = & \langle \rho, \lambda k\, x.K \rangle
\end{array}
$$

Figure: CPS Runtime Data

Execution Example (1)

```
let val x = 1 in x + 2

1. <empty environment>

   letcont c1 x = letval $2 = 2 in
                    letprim $1 = IntAdd x $2 in
                      halt $1 in
     letval $3 = 1 in
       c1 $3

2. c1 -> <continuation>

   letval $3 = 1 in
     c1 $3
```

Execution Example (2)

3.
```
$3 -> 1
c1 -> <continuation>

c1 $3
```

4.
```
x -> 1

letval $2 = 2 in
  letprim $1 = IntAdd x $2 in
    halt $1
```

5.
```
x -> 1
$2 -> 2

letprim $1 = IntAdd x $2 in
  halt $1
```

Execution Example (3)

```
6. x -> 1
   $1 -> 3
   $2 -> 2

   halt $1
```

Evaluation Rules

$$\text{(e-let)}\ \frac{\rho, x \mapsto [\![V]\!]\ \rho \vdash K \Downarrow}{\rho \vdash \mathsf{letval}\ x = V\ \mathsf{in}\ K \Downarrow}$$

$$\text{(e-letc)}\ \frac{\rho, k \mapsto \langle \rho, \lambda x.K \rangle \vdash L \Downarrow}{\rho \vdash \mathsf{letcont}\ k\ x = K\ \mathsf{in}\ L \Downarrow}$$

$$\text{(e-proj)}\ \frac{\rho, y \mapsto r_i \vdash K \Downarrow}{\rho \vdash \mathsf{let}\ y = \pi_i\ x\ \mathsf{in}\ K \Downarrow}\ \rho(x) = (r_1, r_2)$$

$$\text{(e-appc)}\ \frac{\rho', y \mapsto \rho(x) \vdash K \Downarrow}{\rho \vdash k\ x \Downarrow}\ \rho(k) = \langle \rho', \lambda y.K \rangle$$

$$\text{(e-case)}\ \frac{\rho', y \mapsto r \vdash K \Downarrow}{\rho \vdash \mathsf{case}\ x\ \mathsf{of}\ k_1\ [\!]\ k_2 \Downarrow}\ \begin{array}{l}\rho(x) = \mathsf{in}_i\ r \\ \rho(k_i) = \langle \rho', \lambda y.K \rangle\end{array}$$

$$\text{(e-app)}\ \frac{\rho', j \mapsto \rho(k), y \mapsto \rho(x) \vdash K \Downarrow}{\rho \vdash f\ k\ x \Downarrow}\ \rho(f) = \langle \rho', \lambda j\ y.K \rangle$$

$$\text{(e-halt)}\ \frac{}{\rho \vdash \mathsf{halt}\ x \Downarrow}$$

Figure: CPS Evaluation

ML to CPS Translation (1)

$$
\begin{array}{rcl}
[\![\cdot]\!] & : & \mathrm{ML} \rightarrow (\mathrm{Var} \rightarrow \mathrm{CTm}) \rightarrow \mathrm{CTm} \\
[\![x]\!]\ \kappa & = & \kappa(x) \\
[\![()]\!]\ \kappa & = & \mathsf{letval}\ x = ()\ \mathsf{in}\ \kappa(x) \\
[\![e_1\ e_2]\!]\ \kappa & = & [\![e_1]\!]\ (\lambda z_1. \\
 & & \quad [\![e_2]\!]\ (\lambda z_2. \\
 & & \qquad \mathsf{letcont}\ k\ x = \kappa(x)\ \mathsf{in}\ z_1\ k\ z_2)) \\
[\![(e_1, e_2)]\!]\ \kappa & = & [\![e_1]\!]\ (\lambda z_1. \\
 & & \quad [\![e_2]\!]\ (\lambda z_2. \\
 & & \qquad \mathsf{letval}\ x = (z_1, z_2)\ \mathsf{in}\ \kappa(x)))
\end{array}
$$

Figure: Translation Rules (1)

ML to CPS Translation (2)

$$
\begin{aligned}
[\![\texttt{in}i\ e]\!]\ \kappa &= [\![e]\!]\ (\lambda z.\mathsf{letval}\ x = \mathsf{in}_i\ z\ \mathsf{in}\ \kappa(x)) \\
[\![\texttt{\#}i\ e]\!]\ \kappa &= [\![e]\!]\ (\lambda z.\mathsf{let}\ x = \pi_i\ z\ \mathsf{in}\ \kappa(x)) \\
[\![\texttt{fn}\ x\ \texttt{=>}\ e]\!]\ \kappa &= \mathsf{letval}\ f = \lambda k\, x.[\![e]\!]\ (\lambda z.k\ z)\ \mathsf{in}\ \kappa(f)
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{let val}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end}]\!]\ \kappa = \\
&\quad \mathsf{letcont}\ j\ x = [\![e_2]\!]\ \kappa\ \mathsf{in}\ [\![e_1]\!]\ (\lambda z.j\ z)
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{case}\ e\ \texttt{of in1}\ x_1\ \texttt{=>}\ e_1 \mid \texttt{in2}\ x_2\ \texttt{=>}\ e_2]\!]\ \kappa = \\
&\quad [\![e]\!]\ (\lambda z.\mathsf{letcont}\ k_1\ x_1 = [\![e_1]\!]\ \kappa\ \mathsf{in} \\
&\qquad\qquad \mathsf{letcont}\ k_2\ x_2 = [\![e_2]\!]\ \kappa\ \mathsf{in} \\
&\qquad\qquad\quad \mathsf{case}\ z\ \mathsf{of}\ k_1\ [\!]\ k_2)
\end{aligned}
$$

Figure: Translation Rules (2)

Why do we care?

- Every aspect of data and control flow is explicit.
- Good code can be generated directly from CPS.
- Arguably it's easier to perform transformations than in other popular representations:
  - tail call optimisation is direct
  - beta reduction (inlining) is sound
  - sharing is represented directly

Tail Call Optimisation (1)

$$[\![\texttt{fn}\ x\ \texttt{=>}\ e]\!]\ \kappa \quad = \quad \text{letval}\ f = \lambda k\, x.[\![e]\!]\ (\lambda z.k\ z)\ \text{in}\ \kappa(f)$$

Figure: Original Rule

We can do better since the continuation *k* is statically known.

$$[\![\texttt{fn}\ x\ \texttt{=>}\ e]\!]\ \kappa \quad = \quad \text{letval}\ f = \lambda k\, x.\ (\!|e|\!)\ k\ \ \text{in}\ \kappa(f)$$

Figure: New Rule

Tail Call Optimisation (2)

$$
\begin{array}{rcl}
(\!|\cdot|\!) & : & \mathrm{ML} \to \mathrm{CVar} \to \mathrm{CTm} \\
(\!|x|\!)\ k & = & k\ x \\
(\!|e_1\ e_2|\!)\ k & = & [\![e_1]\!]\ (\lambda x_1.[\![e_2]\!]\ (\lambda x_2.x_1\ k\ x_2)) \\
(\!|\texttt{fn}\ x\ \texttt{=>}\ e|\!)\ k & = & \mathsf{letval}\ f = \lambda j\ x.(\!|e|\!)\ j\ \mathsf{in}\ k\ f \\
(\!|\texttt{(}e_1\texttt{,}e_2\texttt{)}|\!)\ k & = & [\![e_1]\!]\ (\lambda x_1.[\![e_2]\!]\ (\lambda x_2.\mathsf{letval}\ x = (x_1, x_2)\ \mathsf{in}\ k\ x)) \\
(\!|\texttt{in}i\ e|\!)\ k & = & [\![e]\!]\ (\lambda z.\mathsf{letval}\ x = \mathsf{in}_i\ z\ \mathsf{in}\ k\ x) \\
(\!|\texttt{()}|\!)\ k & = & \mathsf{letval}\ x = ()\ \mathsf{in}\ k\ x \\
(\!|\texttt{\#}i\ e|\!)\ k & = & [\![e]\!]\ (\lambda z.\mathsf{let}\ x \Leftarrow \pi_i\ z\ \mathsf{in}\ k\ x)
\end{array}
$$

Figure: Some Tail Call Translation Rules (2)

Beta Reduction (Inlining) is Sound

- Not so much in lambda calculus:
  - In $(\lambda x.0)(f\ y)$ it is not sound to reduce to 0, since f may not terminate (or in ML, may have a side-effect).
- The CPS form of the expression is

$$\lambda k_1.f\ (\lambda z.(\lambda k_2.\lambda x.k_2\ 0)\ k_1\ z)\ y$$

  which can be safely reduced to

$$\lambda k_1.f\ (\lambda z.k_1\ 0)\ y$$

A-Normal Form

- "The Essence of Compiling with Continuations", Flanagan et al., PLDI 1993.
- Every intermediate computation is named using a let construct.
- Many transformations need a renormalisation step.
- For example,

  $$\text{let } x = (\lambda y.\text{let } z = a\ b \text{ in } c)\ d \text{ in } e$$

  reduces to

  $$\text{let } x = (\text{let } z = a\ b \text{ in } c) \text{ in } e$$

  which is not in A-normal form.

Sharing

- Compiling some constructs can lead to undesirable duplication.

$$\text{let } z = (\lambda x.\text{if } x \text{ then } a \text{ else } b)\ c \text{ in } M$$

  reduces to non-normal form

$$\text{let } z = (\text{if } c \text{ then } a \text{ else } b) \text{ in } M$$

- One option to return to normal form is to duplicate M in conditional:

$$\text{if } c \text{ then let } z = a \text{ in } M \text{ else let } z = b \text{ in } M$$

- Better is to factor M out and reuse:

$$\text{let } k\ x = M \text{ in if } c \text{ then let } z = a \text{ in } k\ z \text{ else let } z = b \text{ in } k\ z$$

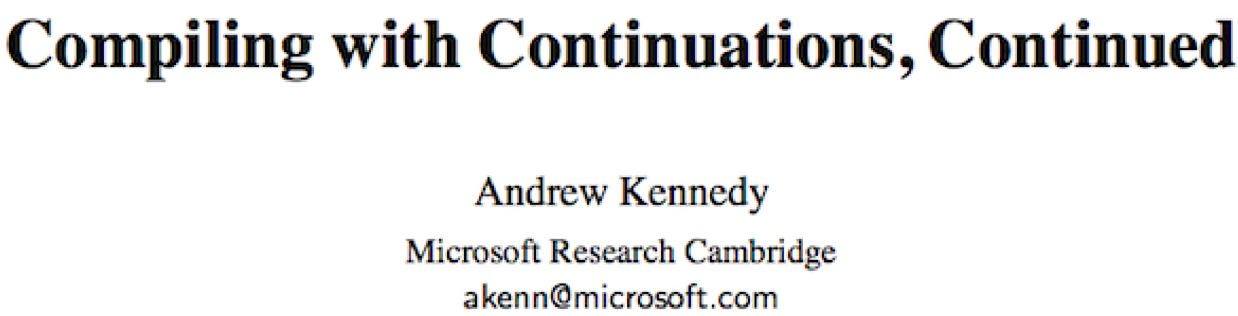
  which is essentially creating a continuation-based form!

Read On...

Kennedy's paper contains much more:

- Proper discussion of CPS vs ANF and monadic style
- Full definition of a typed CPS language with exceptions
- Extension of ML-like language to exceptions and recursive functions
- Efficient graph-based implementation of CPS
- Extended example: transform functions into continuations where possible

Questions?



Compiling with Continuations, Continued

Andrew Kennedy

Microsoft Research Cambridge

akenn@microsoft.com

Figure: ICFP 2007

http://research.microsoft.com/pubs/64044/compilingwithcontinuationscontinued.pdf