

Compiling with Continuations, Continued

Andrew Kennedy

Microsoft Research Cambridge

Continuations are Dead

- Shivers:
“In 2002, then, CPS would appear to be a lesson abandoned.”
- Flanagan, Sabry, Duba, Felleisen:
“Yet, compiler writers abandoned CPS over the next ten years following our paper anyway.”



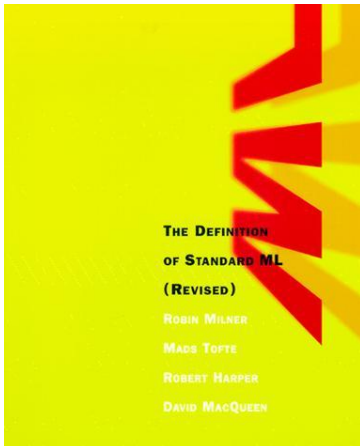
Long live Continuations

- This paper:
 - “Compiler writers, give continuations a second chance.”
 - I’ll try to convince you...
- Along the way, a couple of technical contributions:
 - An improved graph representation for terms
 - A new characterization of contification, based on term rewriting

Some context

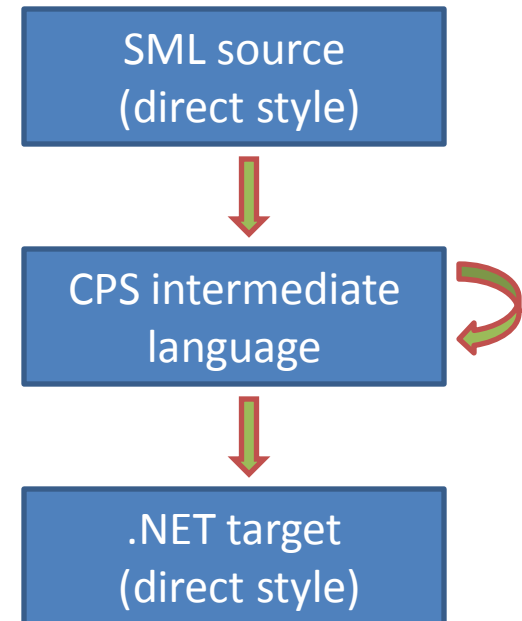
SML language features

- Call-by-value
- Impure
(state, I/O, exceptions)
- No first-class control
(call/cc)

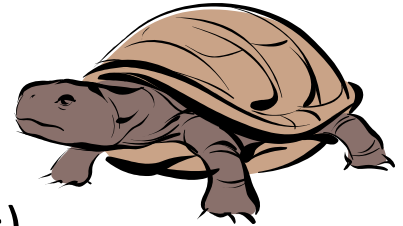


SML.NET compiler features

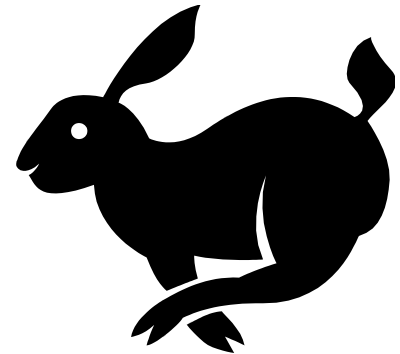
- Whole-program compilation
- High-level target (.NET IL)



Some history



- SML.NET is a *s / o w* compiler
 - Several minutes to (re-)compile itself (80k lines)
- To speed up compilation, we moved to a graph-based representation of intermediate language terms (cf Appel & Jim 1997)
 - Order-of-magnitude speedup
 - But monadic intermediate language (or ANF) not ideal
- We now use a graph-based representation of CPS terms
 - Further factor of 2 speedup
 - Can now self-compile in under 30 seconds.
 - CPS ideally suited to graph representation; other optimizations easier to express

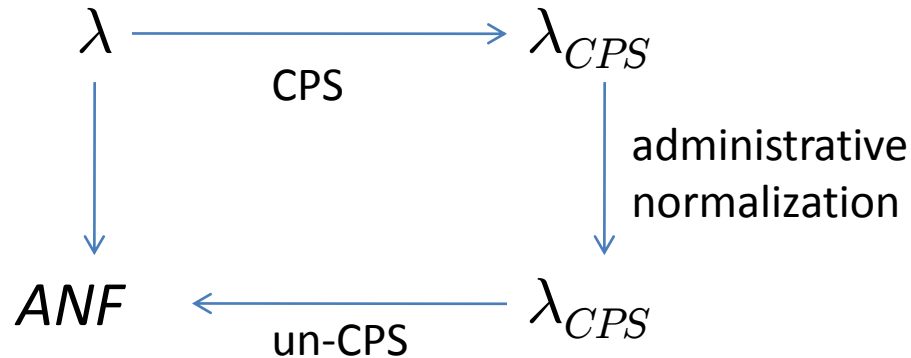


Compiler intermediate languages

- Lambda calculus? Problem: under CBV we can't always apply β -reduction:
 $(\lambda z.0) (f y) \rightarrow ?$
- Solution 1. Transform into continuation-passing style (CPS):
 $f y (\lambda w.(\lambda z.\lambda k.k 0) w k)$
- Solution 2: Transform into monadic language:
 $\text{let } w \Leftarrow f y \text{ in } (\lambda z. \text{val } 0) w$
- Solution 3: Transform into administrative normal form (ANF):
 $\text{let } w = f y \text{ in } (\lambda z.0) w$

All three make evaluation order explicit. How to choose?

Administrative Normal Form



Evaluation order made explicit.
Computations are linearized into a chain of lets.

Let-bound expression is a
value, application, or projection

$$\begin{aligned}
 A, B & ::= R \mid \text{let } x \Leftarrow R \text{ in } A \mid \text{case } v \text{ of } \text{in}_1 x_1.A_1 \mid \text{in}_2 x_2.A_2 \\
 R & ::= v \ w \mid \pi_i v \mid v \\
 v, w & ::= x \mid \lambda x.A \mid (v, w) \mid \text{in}_i v \mid ()
 \end{aligned}$$

ANF

- ANF has many of the nice properties of CPS. And it's easy to generate code from it.
- But it's not closed under standard β -reduction. We must “re-normalize” by floating lets:

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

↓ β reduce (inline function)

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

↓ re-normalize

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

Monadic intermediate language

Separate computations
from values

Let binds arbitrary
computations

$$\begin{aligned} M, N & ::= \text{val } v \mid \text{let } x \Leftarrow M \text{ in } N \mid v w \mid \pi_i v \\ & \mid \text{case } v \text{ of } \text{in}_1 x_1.M_1 \mid \text{in}_2 x_2.M_2 \\ v, w & ::= x \mid \lambda x.M \mid (v, w) \mid \text{in}_i v \mid () \end{aligned}$$

Separate value types τ from
computation types $T \tau$. Can hang effect
info ϵ on computation types $T_\epsilon \tau$

MIL

- MIL has many of the nice properties of CPS. And it's easy to generate code from it.
- But to expose reductions, it's sometimes necessary to apply "commuting conversions":

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



β reduce (inline function)

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



let-float commuting conversion

$\text{let } z \Leftarrow a \ b \ \text{in } \text{let } x \Leftarrow \text{val } c \ \text{in } \text{val } e$

Continuation Passing Style

$$K, L ::= V W \kappa \mid \kappa V \mid \text{let } x = \pi_i y \text{ in } K \mid \dots$$
$$V, W ::= () \mid (x, y) \mid \text{in}_i x \mid \lambda x k.K$$
$$\kappa ::= k \mid \lambda x.K$$

Return continuation

- CPS language is closed under β -reduction. There is no need for commuting conversions or “re-normalization”:

$$(\lambda y k.a b (\lambda z.k c)) d (\lambda x.e)$$


β reduce (argument & continuation)

$$a b (\lambda z.(\lambda x.e) c)$$

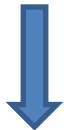
Case/if-floating and join points in ANF

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

↓ β reduce (inline function)

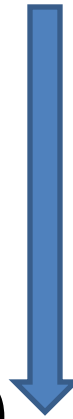
let $z \Leftarrow (\text{if } c \text{ then } a \text{ else } b)$ in M

case-float commuting conversion
(duplicating)



if c then let $z = a$ in M
else let $z = b$ in M

Or...



case-float commuting conversion
(sharing)

let $k = \text{val } (\lambda z. M)$

in if c then let $z = a$ in $k z$ else let $z = b$ in $k z$.

What is k ? It's a continuation!

Summary

- ANF needs a “super- β ” rule that re-normalizes.
(SabryWadler97, also cf Danvy IFL’07, leftist hereditary substitution?)

$$\text{let } x = (\lambda y.M) V \text{ in } N \rightarrow (x)N@(M[V/y])$$

- MIL needs commuting conversions to float lets and cases e.g.

$$\begin{aligned} \text{let } x \Leftarrow (\text{let } y \Leftarrow M_1 \text{ in } M_2) \text{ in } N &\rightarrow \\ \text{let } y \Leftarrow M_1 \text{ in } (\text{let } x \Leftarrow M_2 \text{ in } N) & \end{aligned}$$

- CPS just has standard β and η e.g.

$$\begin{aligned} (\lambda x k.M)V\kappa &\rightarrow \\ M[V/x, \kappa/k] & \end{aligned}$$

Commuting conversions harmful?

- Let-floating and case-floating seem benign. But
 - they complicate the term simplifier
 - to avoid code duplication we must introduce a kind of continuation anyway (the join point)
 - Appel & Jim showed how to perform “shrinking β reductions” in linear time. But commuting conversions aren’t shrinking; and the number required to shrink-reduce a term may be quadratic in size of term e.g.
let $f_n \Leftarrow \text{val } (\lambda x_n. \text{let } y_n \Leftarrow g x_n \text{ in } g y_n) \text{ in}$
let $f_{n-1} \Leftarrow \text{val } (\lambda x_{n-1}. \text{let } y_{n-1} \Leftarrow f_n x_{n-1} \text{ in } g y_{n-1}) \text{ in}$
 \vdots
let $f_1 \Leftarrow \text{val } (\lambda x_1. \text{let } y_1 \Leftarrow f_2 x_1 \text{ in } g y_1) \text{ in } f_1 a$

Continuation Passing Style, Revisited

All values are
named

Branches are
named

$K, L ::= \text{letval } x = V \text{ in } K \mid \text{let } x = \pi_i x \text{ in } K$
 $\mid \text{letcont } k \ x = K \text{ in } L \mid k \ x \mid f \ k \ x \mid \text{case } x \text{ of } k_1 \mid k_2$

Local continuation

Continuation
application

All function
applications take a
continuation argument

$V, W ::= () \mid (x, y) \mid \text{in}_i x \mid \lambda k \ x. K$

Explicit return
continuation

Features of our CPS language

- All intermediate values are named; all control points are named:

$\text{letval } x = V \text{ in } K \quad \lambda k x.K \quad \text{letcont } k x = K \text{ in } L$

- Consequence 1: only ever substitute variables for variables e.g.

$$\begin{aligned} \text{letval } f = \lambda k x.K \text{ in } C[f \ j \ y] &\rightarrow \\ \text{letval } f = \lambda k x.K \text{ in } C[K[j/k, y/x]] \end{aligned}$$

- Consequence 2: join points are in the term from the start. There is no need to create them to share continuations
- Continuations are *second-class*
 - passed to functions, but not returned, or stored in data structures
 - continuation applications can be implemented by jumps
- In full language, they also represent exceptional control flow (double-barrelled CPS), can be recursive, and are typed (see paper)

Scoping rules

Well-formed terms

$$\frac{\Gamma \vdash V \text{ ok} \quad \Gamma, x; \Delta \vdash K \text{ ok}}{\Gamma; \Delta \vdash \text{letval } x = V \text{ in } K \text{ ok}} \quad \frac{\Gamma, x; \Delta \vdash K \text{ ok} \quad \Gamma; \Delta, k \vdash L \text{ ok}}{\Gamma; \Delta \vdash \text{letcont } k \ x = K \text{ in } L \text{ ok}}$$

$$\frac{x \in \Gamma \quad \Gamma, y; \Delta \vdash K \text{ ok}}{\Gamma; \Delta \vdash \text{let } y = \pi_i x \text{ in } K \text{ ok}} \quad i \in 1, 2 \quad \frac{k \in \Delta, x \in \Gamma}{\Gamma; \Delta \vdash k \ x \text{ ok}}$$

$$\frac{k \in \Delta, f, x \in \Gamma}{\Gamma; \Delta \vdash f \ k \ x \text{ ok}} \quad \frac{x \in \Gamma, k_1, k_2 \in \Delta}{\Gamma; \Delta \vdash \text{case } x \text{ of } k_1 \mid k_2 \text{ ok}}$$

Well-formed values

$$\frac{x, y \in \Gamma}{\Gamma \vdash (x, y) \text{ ok}} \quad \frac{x \in \Gamma}{\Gamma \vdash \text{in}_i x \text{ ok}} \quad i \in 1, 2 \quad \frac{}{\Gamma \vdash () \text{ ok}} \quad \frac{\Gamma, x; k \vdash K \text{ ok}}{\Gamma \vdash \lambda k \ x. K \text{ ok}}$$

Well-formed programs

$$\frac{}{\{\}; \text{halt} \vdash K \text{ ok}}$$

Γ is list of ordinary variables
 Δ is list of continuation variables

Return
 continuation only

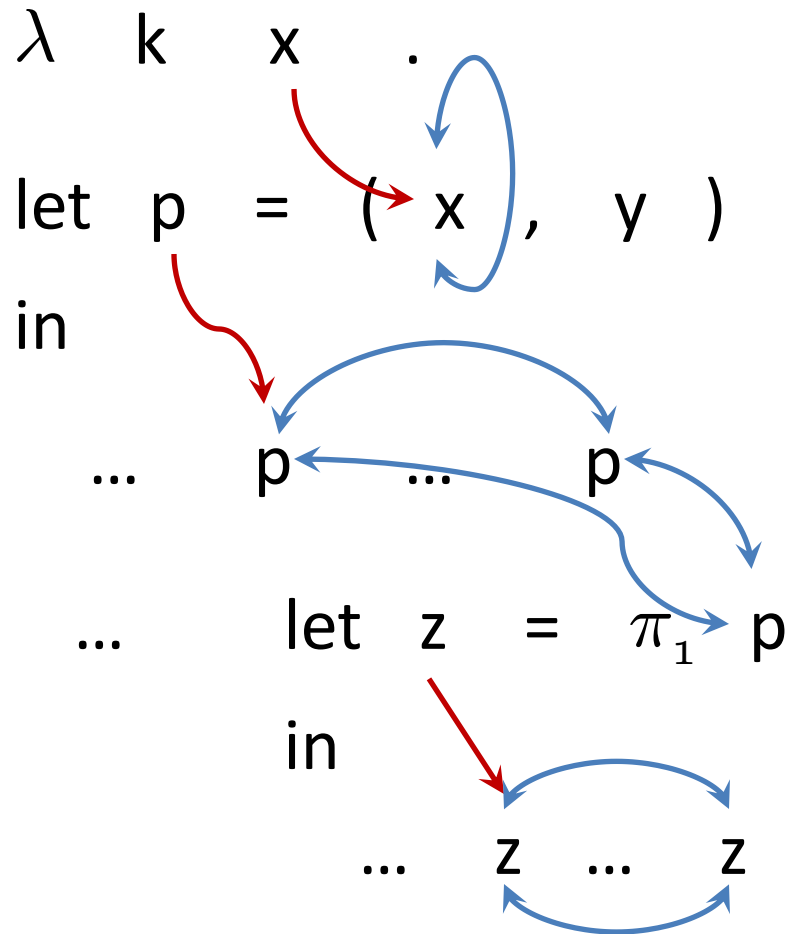
Graph representation

- CPS terms can be implemented in functional style: algebraic datatype + copy-with-diff rewriting
 - For large terms, this is expensive
- Alternative: graph representation + update-in-place rewriting
 - adaptation of Appel & Jim's data structure
 - crucial point: substitution (variable for variable) is constant-time
- Three ingredients
 - Doubly-linked tree for term structure
 - Pointer from bound variable to first free occurrence + doubly-linked circular list between occurrences
 - Union-find data structure to associate occurrences with bound variable



Graph representation

Links from bound to free



Graph representation

Links from free to bound

λ k x .

let p = (x , y)

in

... p ... p

... let z = π_1 p

in

... z ... z

Root of union-find tree

Contification

```
let val f = fn x => ...  
in  
  g (case d of in1 d1 => f y | in2 d2 => f d2)  
end
```

- Function *f* always returns to the “same place”
 - It can therefore be *contified*: compiled as a basic block, with calls compiled as jumps
- This much is obvious from the source code
 - After transformation into an intermediate language, and for more complex examples, it’s not so clear
 - Unless...the intermediate language is CPS-based

Contification

```
let val f = fn x => ...  
in g (case d of in1 d1 => f y | in2 d2 => f d2)  
end
```



CPS transform

```
letval f = ( $\lambda k x \dots k \dots$ ) in  
letcont  $k_0 w = g r w$  in  
letcont  $j_1 d_1 = f k_0 y$  in  
letcont  $j_2 d_2 = f k_0 d_2$  in  
case d of  $j_1$  |  $j_2$ 
```

contify

```
letcont  $k_0 w = g r w$  in  
letcont  $j x = \dots k_0 \dots$  in  
letcont  $j_1 d_1 = j y$  in  
letcont  $j_2 d_2 = j d_2$  in  
case d of  $j_1$  |  $j_2$ 
```

Hoist k_0 to bring it into scope

Common continuation

Replace f by continuation j

Substitute actual continuation arg for formal

Contifcation

CONT (f not free in \mathcal{C} , \mathcal{D} and \mathcal{D} minimal):

letval $f = \lambda k x.K$ in $\mathcal{C}[\mathcal{D}[f k_0 x_1, \dots, f k_0 x_n]]$

\rightarrow

$\mathcal{C}[\text{letcont } j x = K[k_0/k] \text{ in } \mathcal{D}[j x_1, \dots, j x_n]]$

- Generalizes to mutually recursive functions
- Really just common-argument elimination
- Iterating this reduction gives *optimal contifcation* in the sense of Fluet & Weeks (see paper for relationship to their dominator-based approach)

Summary and future work

- Moving to graph-based CPS representation was a lot of work (25,000 lines of code)
- Payoff is
 - Very fast optimizing compilation
 - New optimizations (contification)
 - Simplicity (mutable graphs notwithstanding...)
- Future work
 - CPS as an alternative to SSA for classical optimizations
 - Danvy has recently (IFL'07!) shown how to amortize cost of re-normalization in ANF

Example monadic term simplification

#1 ((fn x => (g x, x)) y)



CPS transform from SML source

let $z_2 \Leftarrow (\lambda x. \text{let } z_1 \Leftarrow g x \text{ in val } (z_1, x)) y \text{ in } \pi_1 z_2.$



β reduce (inline function)

let $z_2 \Leftarrow (\text{let } z_1 \Leftarrow g y \text{ in val } (z_1, y)) \text{ in } \pi_1 z_2.$



let-assoc (commuting conversion)

let $z_1 \Leftarrow g y \text{ in let } z_2 \Leftarrow \text{val } (z_1, y) \text{ in } \pi_1 z_2.$



β reduce (pair)

let $z_1 \Leftarrow g y \text{ in } z_1$



η reduce (let)

$g y$

Example CPS term simplification

#1 ((fn x => (g x, x)) y)

let $f = \lambda j_1 x.$
(letcont $j_2 z_1 =$ (letval $z_2 = (z_1, x)$ in $j_1 z_2$) in $g j_2 x$)
in letcont $j_3 z_3 =$ (let $z_4 = \pi_1 z_3$ in $k z_4$)



CPS transform from SML source

in $f j_3 y$



β reduce (inline function)

letcont $j_3 z_3 =$ (let $z_4 = \pi_1 z_3$ in $k z_4$)

in letcont $j_2 z_1 =$ (letval $z_2 = (z_1, y)$ in $j_3 z_2$) in $g j_2 y$



β reduce (continuation)

letcont $j_2 z_1 =$

(letval $z_2 = (z_1, y)$ in let $z_4 = \pi_1 z_2$ in $k z_4$)

in $g j_2 y.$



β reduce (pair)

letcont $j_2 z_1 = k z_1$ in $g j_2 y$



η reduce (cont)

$g k y$