Practical Conversion from CPS to Direct Style

Kavon Farvardin and John Reppy

University of Chicago

MWPLS

December 2, 2016
CPS is great for compilers

- Evaluation order is made explicit.
- Control-flow is regularized.
- Useful for both high-level and low-level representations.
- Easily supports non-local control-flow; exceptions, call/cc, etc.
Bringing Continuations to LLVM

• Ongoing work to explore implementations of continuations.

• Native codegen is a pain; using LLVM is easier.

• Recent work: heap-allocated, first-class conts with LLVM

• How can a CPS-based compiler use LLVM with a stack?
CPS with a stack

```
fun fib (n, k) =
    if n <= 2 then k 1
    else
    cont minus2 f_1 =
    cont add f_2 =
    k (f_1 + f_2)
  in fib (n-2, add)
in fib (n-1, minus2)
```

minus2’s closure dies here

Downward funargs
Undoing CPS *in theory*

**Key Observation***

Most continuations created by CPS are well-behaved.

* by Danvy, Kelsey, etc.
Undoing CPS in practice

It starts with a good intermediate representation:

- Continuations and functions are different.
- Continuation parameters added by CPS are distinguished.

\[
\text{cont } k () = _ \text{ in } _ \leftrightarrow \text{ throw } k ()
\]

\[
\text{fun } f (x, y / k) = _ \leftrightarrow f (1, 2 / k')
\]
Noninvasive Compiler Upgrades

\[ \ldots \rightarrow DS \rightarrow CPS \text{ convert} \rightarrow CPS \rightarrow \text{ Closure convert} \rightarrow CFG \text{ (first-order)} \rightarrow LLVM \]
Noninvasive Compiler Upgrades

... → DS → CPS convert → CPS → Closure convert → CFG (first-order) → LLVM

- Classification of Continuations
- Closure & DS convert
Classifying Continuations

fun g x = x
fun f x y = if x > 10 then h((g x) + y) else h x

fun g (x / k) = throw k x
fun f (x, y / k) =
cont doH z = h (z + y / k) in
if x > 10
  then g (x / doH)
  else h (x / k)

CPS

Higher-order DS

Return throw
Non-tail call
Tail call

Return continuations are only ever used or passed from the same function.
Converting to Direct Style

Higher-order CPS

\[
\begin{align*}
\text{fun } g (x / k) &= \text{throw } k x \\
\text{fun } f (x, y / k) &= \\
&\quad \text{cont doH } z = h (z + y / k) \text{ in} \\
&\quad \text{if } x > 10 \\
&\quad \quad \text{then } g (x / \text{doH}) \\
&\quad \text{else } h (x / k)
\end{align*}
\]

First-order DS

\[
\begin{align*}
\text{fun } g (_, x) &= \text{return } x \\
\text{fun } f (ep, x, y) &= \\
&\quad \text{block doH } (ep, z, y) = \\
&\quad \quad \text{tailcall } h (z + y) \\
&\quad \text{if } x > 10 \\
&\quad \quad \text{then } z = \text{call } g x \\
&\quad \quad \quad \text{goto doH } (ep, z, y) \\
&\quad \text{else } \text{tailcall } h x
\end{align*}
\]
Taming CPS Optimizations

• Arity raising

• Expansive inlining

• ... maybe others?

fun foo t = let
    val x = #1(t)
    val t = #2(t)
    ...

Unbox tuple

fun foo x y = let
    ...


Taming CPS Optimizations

```haskell
fun foo (_, / fooRet) =
  fun bar (_, / barRet) = throw barRet ()

fun g (_, / gRet) =
  if ...
  then bar (_, / gRet)
  else throw gRet ()

cont joinK () =
  ...
  throw fooRet ()
in
  g (_, / joinK)
```
fun foo (_ / fooRet) =
  fun bar (_ / barRet) = throw barRet ()

fun g (_ / gRet) =
  if ... then bar (_ / gRet)
  else throw gRet ()

cont joinK () =
  ...
  throw fooRet ()
in
  g (_ / joinK)

CFA says barRet = joinK, so we inline the throw to barRet.
fun foo (_, fooRet) = 
  fun bar (_) = ... **throw fooRet ()**

fun g (_, gRet) = 
  if ... 
  then bar (_) 
  else **throw gRet ()**

**cont** joinK () = 
  ... 
  **throw fooRet ()**

in 
  g (_, / joinK)

Stack (grows down)

foo’s Return Cont.

g’s Return Cont.
Conclusion and Ongoing Work

• Direct style conversion can be done easily during closure conversion.

• Ongoing Work
  • Dedicated stack-based cont primitives (newStack, etc.)
  • Extending LLVM to support first-class stack-based conts.