A Vademecum to Continuations (continued)

Olivier Danvy

BRICS, University of Aarhus, Denmark
(danvy@brics.dk)

BRICS EWSCS March 1, 2005

Plan

- The CPS transformation: S, fib, map, Calder mobiles, printf, listing list prefixes, non-deterministic programming.
- A monadic one-pass transformation.
- Implementing first-class continuations.

The CPS transformation

- Names intermediate results.
- Sequentializes their computation.
- Introduces first-class functions (continuations).

Subplan

- S
- fib
- map
- Calder mobiles
- printf
- listing list prefixes
- non-deterministic programming

A simple example (1/3)

```
f x (g x)
```

A simple example (2/3)

```
f x (g x)


let v1 = f x
    v2 = g x
    v3 = v1 v2
in v3
```

A simple example (3/3)

```
f x (g x)
```

```
let v1 = f x
    v2 = g x
    v3 = v1 v2
in v3
```

```
\k.f x (\v1.
   g x (\v2.
   v1 v2 (\v3.
   k v3)))
```

Subplan

- S $\surd$
- fib
- map
- Calder mobiles
- printf
- listing list prefixes
- non-deterministic programming

The Fibonacci function (1/3)

```
fib n
= if n <= 1
  then n
  else fib(n - 1) + fib(n - 2)
```

The Fibonacci function (2/3)

```
fib n
= if n <= 1
  then n
  else let v1 = fib(n - 1)
           v2 = fib(n - 2)
       in v1 + v2
```

The Fibonacci function (3/3)

```
fib (n, k)
= if n <= 1
  then k n
  else fib(n - 1, \v1.
       fib(n - 2, \v2.
       k (v1 + v2)))
```

The Fibonacci function (4/3)

```
fib n
= if n <= 1 then n
  else let n1 = n - 1
           v1 = fib n1
           n2 = n - 2
           v2 = fib n2
       in v1 + v2
```

CPS or not CPS?

Q. When should we make a function continuation-passing?

CPS or not CPS?

Q. When should we make a function continuation-passing?

A. When it is pure and total.

Subplan

- S $\surd$
- fib $\surd$
- map
- Calder mobiles
- printf
- listing list prefixes
- non-deterministic programming

The map function (1/3)

```
fun map (f, nil)
      = nil
  | map (f, x :: xs)
      = (f x) :: (map (f, xs))
```

The map function (2/3)

```
fun map (f, nil)
      = nil
  | map (f, x :: xs)
      = let v1 = f x
            v2 = map (f, xs)
        in v1 :: v2
```

The map function (3/3)

```
fun map (f, nil, k)
      = k nil
  | map (f, x :: xs, k)
      = f (x, \v1.
          map (f, xs, \v2.
          k (v1 :: v2)))
```

Subplan

- S $\surd$
- fib $\surd$
- map $\surd$
- Calder mobiles
- printf
- listing list prefixes
- non-deterministic programming

Exercise: Calder mobiles

- Consider a binary tree of natural numbers.
- Compute whether it is well balanced.
- Do it in one traversal.

Correction of the exercise

Question: is a binary tree well balanced?

Solution: use an auxiliary function

$$\mathsf{aux} : \mathsf{bt} \longrightarrow \mathsf{Nat} + \mathsf{Unit}$$

- If $\mathfrak{t}$ is balanced and of weight $\mathfrak{n}$

  then aux $\mathfrak{t}$ = L $\mathfrak{n}$.

- If $\mathfrak{t}$ is not balanced then aux $\mathfrak{t}$ = R.

Criticism of the solution

One pass, yes,

but injecting and dispatching at each step.

Alternative: use CPS

$$\mathsf{bt} \to \big(\mathsf{Nat} + \mathsf{Unit} \to \mathsf{Bool}\big) \to \mathsf{Bool}$$

Better: use two continuations.

$$\mathsf{bt} \to \big(\mathsf{Nat} \to \mathsf{Bool}\big) \times \big(\mathsf{Unit} \to \mathsf{Bool}\big) \to \mathsf{Bool}$$

No more injecting and dispatching!

Alternative: use CPS (contd)

$$\mathsf{bt} \longrightarrow \big(\mathsf{Nat} \longrightarrow \mathsf{Bool}\big) \times \big(\mathsf{Unit} \longrightarrow \mathsf{Bool}\big) \longrightarrow \mathsf{Bool}$$

Simpler: use only one continuation and a failure value.

$$\mathsf{bt} \longrightarrow \big(\mathsf{Nat} \longrightarrow \mathsf{Bool}\big) \times \mathsf{Bool} \longrightarrow \mathsf{Bool}$$

Alternative: use CPS (contd)

$$\mathsf{bt} \longrightarrow \big(\mathsf{Nat} \longrightarrow \mathsf{Bool}\big) \times \mathsf{Bool} \longrightarrow \mathsf{Bool}$$

Even simpler: inline the failure value.

$$\mathsf{bt} \longrightarrow \big(\mathsf{Nat} \longrightarrow \mathsf{Bool}\big) \longrightarrow \mathsf{Bool}$$

Alternative: use CPS (ended)

$$\mathrm{bt} \longrightarrow \big(\mathrm{Nat} \longrightarrow \mathrm{Bool}\big) \longrightarrow \mathrm{Bool}$$

Further alternatives:

- Find another representation of $\mathrm{Nat} \longrightarrow \mathrm{Bool}$.
- Write aux in direct style: $\mathrm{bt} \longrightarrow \mathrm{Nat}$

  and use call/cc.

Another alternative

Implement bt $\longrightarrow$ Nat + Unit

as an exception-raising function

$$\text{bt} \xrightarrow{exn} \text{Nat}$$

Subplan

- S $\surd$
- fib $\surd$
- map $\surd$
- Calder mobiles $\surd$
- printf
- listing list prefixes
- non-deterministic programming

What is the type of printf?

```
printf ("Hello world.\n");

printf ("The %s is %d.\n",
        "answer",
        42);
```

The issue: concrete vs. abstract syntax

Alan Perlis's Epigram #34: "The string is a stark data structure and everywhere it is passed there is much duplication of process. It is a perfect vehicle for hiding information."

The issue: concrete vs. abstract syntax

Alan Perlis's Epigram #106: "It's difficult to extract sense from strings, but they're the only communication coin we can count on."

Abstract syntax of patterns

- `lit` for declaring literal strings;
- `eol` for declaring newlines;
- `int` for specifying integers; and
- `str` for specifying strings.

Plus an associative infix operator `oo`.

Example

```
sprintf
 (int oo lit " is " oo str)
: int -> string -> string

sprintf
 (int oo lit "/" oo int oo eol)
: int -> int -> string
```

The insight

- Recursion: no.
- Induction: yes.
- Use CPS to thread the constructed string throughout.
- Use polymorphism for the domain of answers.

eol

```
(string -> ’a) -> string
-> ’a
```

int

```
(string -> 'a) -> string
-> int -> 'a
```

The code (1/2)

```
fun lit x k s
    = k (s ^ x)

fun eol k s
    = k (s ^ "\n")
```

The code (2/2)

```
fun int k s (x:int)
    = k (s ^ (makestring x))

fun str k s x
    = k (s ^ x)
```

Glueing the directives

`oo` is just function composition.

```
int oo int :
(string -> 'a) -> string ->
int -> int -> 'a
```

Initialization

```
fun sprintf p
    = p (fn (s:string) => s) ""
```

Partial evaluation

```
sprintf
 (int oo lit " is " oo str oo eol)
```

specializes to

```
fn (x1:int) =>
fn x2 =>
(makestring x1) ^ " is " ^ x2 ^ "\n"
```

Reference

“Functional Unparsing”

JFP 8(6):621-625

BRICS RS-98-12 (extended version)

Subplan

- S $\surd$
- fib $\surd$
- map $\surd$
- Calder mobiles $\surd$
- printf $\surd$
- listing list prefixes
- non-deterministic programming

Listing list suffi xes

Easy.

Listing list prefi xes

Less easy.

Plan

- The CPS transformation: S, fib, map, Calder mobiles, printf, listing list prefixes, non-deterministic programming. $\surd$
- A monadic one-pass transformation.
- Implementing first-class continuations.

Topic: compiling from one language to another.

Area: from λ_v-terms to monadic normal forms.

Specifics: short-cut boolean evaluation.

Goal: efficiency at compile time and at run time.

Source and target languages

The λ_v-calculus: the prototypical idealized programming language.

Monadic normal forms: the prototypical idealized intermediate language, where

- all intermediate results are named, and
- their computation is sequentialized.

Short-cut boolean evaluation

- Left-to-right evaluation.
- Absorbant elements (false for conjunction, true for disjunction) prevail ASAP.

Goal

The translation should

- work in one pass,
- duplicate no code,
- be properly tail recursive,
- generate no chain of thunks.

Starting point: a one-pass transformation

for the λ_v-calculus.

Extension: boolean expressions and

short-cut evaluation.

Encoding λ_v-terms into monadic style:

$$
\begin{aligned}
\mathcal{E}_v[\![x]\!] &= \text{return } x \\
\mathcal{E}_v[\![\lambda x.e]\!] &= \text{return } \lambda x.\mathcal{E}_v[\![e]\!] \\
\mathcal{E}_v[\![e_0\ e_1]\!] &= \text{let } w_0 = \mathcal{E}_v[\![e_0]\!] \\
&\quad \text{in let } w_1 = \mathcal{E}_v[\![e_1]\!] \\
&\quad\ \text{in } w_0\ w_1
\end{aligned}
$$

Wanted

$$\begin{aligned}
\mathcal{B}_\nu[\![\mathrm{e}]\!] &= \mathcal{E}_\nu[\![\mathrm{e}]\!] \\
\mathcal{B}_\nu[\![\mathrm{b}_1 \wedge \mathrm{b}_2]\!] &= \text{if } \mathcal{B}_\nu[\![\mathrm{b}_1]\!] \text{ then } \mathcal{B}_\nu[\![\mathrm{b}_2]\!] \text{ else } \textit{false} \\
\mathcal{B}_\nu[\![\mathrm{b}_1 \vee \mathrm{b}_2]\!] &= \text{if } \mathcal{B}_\nu[\![\mathrm{b}_1]\!] \text{ then } \textit{true} \text{ else } \mathcal{B}_\nu[\![\mathrm{b}_2]\!] \\
\mathcal{B}_\nu[\![\neg \mathrm{b}]\!] &= \text{if } \mathcal{B}_\nu[\![\mathrm{b}]\!] \text{ then } \textit{false} \text{ else } \textit{true}
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}_v[\![\text{if } \mathrm{b}_2 \text{ then } \mathrm{b}_1 \text{ else } \mathrm{b}_0]\!] &= \text{if } \mathcal{B}_v[\![\mathrm{b}_2]\!] \\
&\quad \text{then } \mathcal{E}_v[\![\mathrm{b}_1]\!] \\
&\quad \text{else } \mathcal{E}_v[\![\mathrm{b}_2]\!]
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}_v[\![\text{if } \mathrm{b}_2 \text{ then } \mathrm{b}_1 \text{ else } \mathrm{b}_0]\!] &= \text{if } \mathcal{B}_v[\![\mathrm{b}_2]\!] \\
&\quad \text{then } \mathcal{B}_v[\![\mathrm{b}_1]\!] \\
&\quad \text{else } \mathcal{B}_v[\![\mathrm{b}_0]\!]
\end{aligned}$$

Problem

The result is not in normal form:

let-expressions are nested, not flat.

$$
\begin{aligned}
\mathcal{E}_v[\![x]\!] &= \text{return } x \\
\mathcal{E}_v[\![\lambda x.e]\!] &= \text{return } \lambda x.\mathcal{E}_v[\![e]\!] \\
\mathcal{E}_v[\![e_0\ e_1]\!] &= \text{let } w_0 = \mathcal{E}_v[\![e_0]\!] \\
&\quad \text{in let } w_1 = \mathcal{E}_v[\![e_1]\!] \\
&\quad\ \text{in } w_0\ w_1
\end{aligned}
$$

A simple solution

Add an extra pass to normalize.

A better solution

A continuation-based one-pass solution:

$$\begin{cases} \mathcal{E} & : \ \Lambda \to \Lambda^{c} \\ \mathcal{E}' & : \ \Lambda \to (\Lambda^{v} \to \Lambda^{c}) \to \Lambda^{c} \end{cases}$$

A better solution

A continuation-based one-pass solution:

$$\begin{cases} \mathcal{E} & : \ \Lambda \to \Lambda^{c} \\ \mathcal{E}' & : \ \Lambda \to (\Lambda^{v} \to \Lambda^{c}) \to \Lambda^{c} \end{cases}$$

We build on this solution.

The insight: split the continuation

$$\mathit{Bool} \to \Lambda^{C}_{ml} \equiv (\mathit{Unit} \to \Lambda^{C}_{ml}) \times (\mathit{Unit} \to \Lambda^{C}_{ml})$$

Byproduct (for example):

$$\mathcal{B}[\![\neg \mathrm{b}]\!] \langle \kappa_1, \kappa_0 \rangle \;=\; \mathcal{B}_{cc}[\![\mathrm{b}]\!] \langle \kappa_0, \kappa_1 \rangle$$

No context duplication

Boolean contexts in non-tail position

are named in let expressions.

(Just beware of chains of jumps.)

All in all: four functions

$$\mathcal{B}_{vv} : \Lambda^{\mathrm{B}} \to \Lambda^{\mathrm{V}}_{ml} \times \Lambda^{\mathrm{V}}_{ml} \to \Lambda^{\mathrm{C}}_{ml}$$

$$\mathcal{B}_{cv} : \Lambda^{\mathrm{B}} \to (1 \to \Lambda^{\mathrm{C}}_{ml}) \times \Lambda^{\mathrm{V}}_{ml} \to \Lambda^{\mathrm{C}}_{ml}$$

$$\mathcal{B}_{vc} : \Lambda^{\mathrm{B}} \to \Lambda^{\mathrm{V}}_{ml} \times (1 \to \Lambda^{\mathrm{C}}_{ml}) \to \Lambda^{\mathrm{C}}_{ml}$$

$$\mathcal{B}_{cc} : \Lambda^{\mathrm{B}} \to (1 \to \Lambda^{\mathrm{C}}_{ml}) \times (1 \to \Lambda^{\mathrm{C}}_{ml}) \to \Lambda^{\mathrm{C}}_{ml}$$

+ an interface with $\mathcal{E}$ and $\mathcal{E}'$.

Results

- One pass: the source term is traversed once and the target term is not re-traversed.
- Monadic normal form: target grammar as ML data type.
- No chain of jumps: structural induction.

See BRICS RS-02-52 (CC'03) for more detail.

$\lambda x.g\ (h\ (\text{if } a \text{ then if } b_2 \text{ then } b_1 \text{ else } b_0 \text{ else } x))$

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

$\lambda x.\text{if } a_1 \wedge a_2 \wedge a_3 \wedge a_4 \text{ then } x \text{ else } g\ (h\ x)$

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

Assessment in theory

“Nice hack,

but what do I make of it?”

$\longrightarrow$ It is another instance of

Normalization by Evaluation.

Assessment in practice

“Nice functional program,

but what do I make of it?”

$\longrightarrow$ All continuations are stackable,

so the program can be defunctionalized

into an efficient C program.

Question from a functional programmer

“Is this similar to Bird’s circular programs?”

$\longrightarrow$ No it isn’t (see appendix in paper).

Question from a software engineer

“Is the new compiler easy to maintain?”

$\longrightarrow$ It’s hard to say (cf. JIT issues).

- The translator becomes unintuitive

  (i.e., we need a formal proof).

- Its effectiveness becomes unpredictable

  (e.g., SML/NJ 0.93).

Summary

- A simple and effective solution,
  directly applicable in a JIT compiler.
- A new illustration of
  compiling with continuations.
- A new instance of normalization by evaluation.
- A possible application to SSA transformations.

Plan

- The CPS transformation: S, fib, map, Calder mobiles, printf, listing list prefixes, non-deterministic programming. $\surd$
- A monadic one-pass transformation. $\surd$
- Implementing first-class continuations.

Alternatives to continuations

☺ Theorem: For all situations, there exists an alternative to continuations.

- But it never seems to be the same alternative.
- However, it is always the same continuations.