Reduction contexts without headaches

— a functional pearl —

Olivier Danvy

(danvy@brics.dk)

IFIP WG 2.8 July 17, 2007

This could happen to you

Write a reduction semantics for, e.g.,

conditional arithmetic expressions.

Terms and values

```
datatype term =
     NUM of int
   | ADD of term * term
   | BOO of bool
   | CND of term * term * term

datatype value =
     VNUM of int
   | VBOO of bool
```

Potential redexes

```
datatype potredex =
     SUM of value * value
   | SEL of value * term * term
```

A redex may be an actual one or a stuck one.

Contraction

```
(*  contract : potredex -> term option  *)
fun contract (SUM (VNUM n1, VNUM n2))
    = SOME (NUM (n1 + n2))
  | contract (SEL (VBOO b, t2, t3))
    = SOME (if b then t2 else t3)
  | contract r
    = NONE
```

Reduction strategey

Depth-first, left-to-right.

Challenge

Wanted:

- the grammar of reduction contexts,
- the corresponding plug function,
- a decomposition function, and
- a unique-decomposition property.

Root of the problem (1/2)

Contexts are notoriously tricky to get right:

- Are all cases covered?
- Are some of them redundant?
- Are they “outside in” or “inside out”?

Root of the problem (2/2)

Contexts are notoriously tricky to get right:

- Shouldn’t they be like stack frames or something?
- Does the unique-decomposition property hold?

(cf. the Flatt test)

Plan

- a 2-step method
- a variant of the 2nd step
- lessons learned
- perspectives

Step 1

Write a compositional function

```
term -> potredex option
```

searching for the first redex in a term.

Step 2

Expand the search function
to return a fill function

```
(potredex * (term -> term)) option
```

- applying it to the potential redex
  yields back the term
- applying it to the contractum
  yields the next term in the reduction sequence

Step 2a: synthesizing the fill function

- constructing the fill function at return time

Step 2a: synthesizing the fill function

- constructing the fill function at return time
- defunctionalize the fill function

Step 2a: synthesizing the fill function

- constructing the fill function at return time
- defunctionalize the fill function

Result: outside-in reduction contexts
+ plug function

Step 2b: inheriting the fill function

- constructing the fill function at call time

Step 2b: inheriting the fill function

- constructing the fill function at call time
- defunctionalize the fill function

Step 2b: inheriting the fill function

- constructing the fill function at call time
- defunctionalize the fill function

Result: inside-out reduction contexts
+ plug function

Variant of the 2nd step

- fill is an endofunction
- represent it with explicit function composition
- replace the monoid of functions
  by a monoid of lists
- defunctionalize each elementary function
  in the list

Results:

- synthesized: plug done with right fold
- inherited: plug done with left fold
- inherited: LIFO list of control-stack frames

Lessons learned

- simple, mechanical way
  of obtaining reduction contexts

- scales up

- unique-decomposition property
  follows as corollary

- clarification of outside-in and
  inside-out contexts

Perspectives: small-step semantics (1/4)

function implementing a structural operational semantics $\xrightarrow[\text{transf.}]{\text{CPS}} \quad \xrightarrow{\text{defunct.}}$ function implementing a reduction semantics

Perspectives: big-step semantics (2/4)

function implementing a natural semantics $\xrightarrow[\text{transf.}]{\text{CPS}}$ $\xrightarrow{\text{defunct.}}$ function implementing a big-step abstract machine

Reynolds, 1972

Perspectives: small-step semantics (3/4)

function implementing a reduction semantics $\xrightarrow{\text{refocusing}}$ function implementing a small-step abstract machine

Danvy and Nielsen, BRICS RS-04-26

Perspectives: abstract machines (4/4)

$$\begin{array}{c}\text{function}\\\text{implementing}\\\text{a small-step}\\\text{abstract}\\\text{machine}\end{array} \xrightarrow[\text{fusion}]{\text{lightweight}} \begin{array}{c}\text{function}\\\text{implementing}\\\text{a big-step}\\\text{abstract}\\\text{machine}\end{array}$$

Ohori and Sasano, POPL'07

Danvy and Millikin, BRICS RS-07-08

Conclusions

- abstract machines: a natural meeting ground
- reduction contexts = evaluation contexts
  (They are in defunctionalized form, and
  it’s their apply function that matters.)
- the ubiquitous zipper
  (Ditto.)

“There is a striking analogy

between computing a program

and assigning semantics to it.”

“Intuitionistic model constructions

and normalization proofs”

Thierry Coquand and Peter Dybjer, 1993