On Graph Rewriting, Reduction and Evaluation

Ian Zerny

Department of Computer Science, Aarhus University, Denmark
zerny@cs.au.dk

Trends in Functional Programming, 2009

Graph reduction

What? Represent terms as graphs instead of trees

Why? Avoid redundant computation

How? Two main approaches to graph reduction

- Reduction machines à la Turner
- Graph rewriting systems à la Barendregt et al.

This talk

Goal: Connecting reduction machines à la Turner with graph rewriting systems à la Barendregt.

Means: Mechanical program derivation based on Danvy's AFP 2008 presentation.

The syntactic correspondence

Danvy and students:

$$\text{calculi} \quad \xleftrightarrow[\text{correspondence}]{\text{syntactic}} \quad \text{abstract machines}$$

The syntactic correspondence

Danvy and students:

calculi $\xleftrightarrow[\text{correspondence}]{\text{syntactic}}$ abstract machines

a.o. λ-calculus $\longleftrightarrow$ CK

a.o. $\lambda\widehat{\rho}$-calculus $\longleftrightarrow$ CEK

n.o. $\lambda\widehat{\rho}$-calculus $\longleftrightarrow$ KAM

The syntactic correspondence

Danvy and students:

calculi $\xleftrightarrow[\text{correspondence}]{\text{syntactic}}$ abstract machines

a.o. λ-calculus $\longleftrightarrow$ CK

a.o. $\lambda\widehat{\rho}$-calculus $\longleftrightarrow$ CEK

n.o. $\lambda\widehat{\rho}$-calculus $\longleftrightarrow$ KAM

$\longleftrightarrow$ SECD

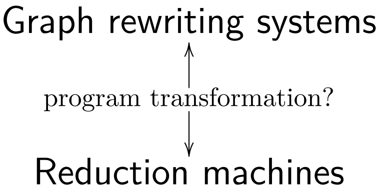
$\longleftrightarrow$ CAM

The syntactic correspondence

Danvy and students:

calculi	$\xleftrightarrow[\text{correspondence}]{\text{syntactic}}$	abstract machines
a.o. λ-calculus	$\longleftrightarrow$	CK
a.o. $\lambda\widehat{\rho}$-calculus	$\longleftrightarrow$	CEK
n.o. $\lambda\widehat{\rho}$-calculus	$\longleftrightarrow$	KAM
	$\longleftrightarrow$	SECD
	$\longleftrightarrow$	CAM
λ_{sec}-calculus	$\longleftrightarrow$	
σ-calculus	$\longleftrightarrow$	

What about graph reduction?



Graph rewriting systems

program transformation?

Reduction machines

Key issues

- Side effects
- Modification of executing code
- Non-functional formalizations

Domain of discourse

The simplest setting: the S, K and I combinators

$$\begin{aligned}\mathrm{S}xyz &= xz(yz)\\ \mathrm{K}xy &= x\\ \mathrm{I}x &= x\end{aligned}$$

- No loss of generality

Formalization of graphs with Standard ML references

```
datatype atom  = S | K | I
datatype node  = C of atom
               | A of graph × graph
withtype graph = node ref
```

Domain of discourse

The simplest setting: the S, K and I combinators

$$\begin{aligned} \mathrm{S}xyz &= xz(yz) \\ \mathrm{K}xy &= x \\ \mathrm{I}x &= x \end{aligned}$$

- No loss of generality

Formalization of graphs with Standard ML references

```
datatype atom  = S | K | I
datatype node  = C of atom
               | A of graph × graph
withtype graph = node ref
```

Overview

- Formalization of a reduction machine
- Formalization of a graph rewriting system
- Derivation
- Towards graph evaluation
- Conclusion

Reduction machines à la Turner

- Set of combinators and primitive operations
- Stack unwinding routine
- Application routine by graph transformation

Our reduction machine for S, K and I

- Restricted to only S, K and I
- Full normal form reduction
- Stack management by stack marking
- Transition functions `unwind` and `apply`
- Fits on a single slide

Our reduction machine for S, K and I

```
(* unwind_RM : graph × stack → graph *)
fun unwind_RM (g as ref (A (g0, g1)), gs)
    = unwind_RM (g0, PUSH (g, gs))
  | unwind_RM (g as ref (C a), gs)
    = apply_RM (a, g, gs)

(* apply_RM : atom × graph × stack → graph *)
and apply_RM (_, g, EMPTY)
    = g
  | apply_RM (I, _, PUSH (r as ref (A (_, x)), gs))
    = (r := !x;
       unwind_RM (r, gs))
  | apply_RM (K, _, PUSH (      ref (A (_,  x)),
                     PUSH (r as ref (A (g0, y as g1)),
                      gs)))
    = (g0 := C I;
       g1 := !x;
       r  := !x;
       unwind_RM (r, gs))
  | apply_RM (S, _, PUSH (       ref (A (_,  x)),
                     PUSH (      ref (A (_,  y)),
                      PUSH (r as ref (A (g0, z as g1)),
                       gs))))
    = (g0 := A (ref (!x), ref (!z));
       g1 := A (ref (!y), ref (!z));
       unwind_RM (r, gs))
  | apply_RM (a, _, PUSH (g as ref (A (_, g1)), gs))
    = unwind_RM (g1, MARK (a, g, gs))
  | apply_RM (_, _, MARK (a, g, gs))
    = apply_RM (a, g, gs)
```

Overview

- Formalization of a reduction machine
- Formalization of a graph rewriting system
- Derivation
- Towards graph evaluation
- Conclusion

Graph rewriting systems à la Barendregt

- Graph: N, F, $N \to F$, $N \rightharpoonup N \times N$
- Rewrite rules
- Reduction strategy

Rewriting

- Rewriting à la Barendregt
    - Extensional
    - Rewiring is induced by the formalism
- Rewriting à la Plasmeijer
    - Intensional
    - Rewiring is a computational axiom

Rewriting

Rewrite rules for I, K and S à la Barendregt

$$\begin{array}{lllll}
\langle & r : \mathrm{A}(\mathrm{I}, x), & r, & x & \rangle \\
\langle & r : \mathrm{A}(\mathrm{A}(\mathrm{K}, x), y), & r, & x & \rangle \\
\langle & r : \mathrm{A}(\mathrm{A}(\mathrm{A}(\mathrm{S}, x), y), z) + r' : \mathrm{A}(\mathrm{A}(x, z), \mathrm{A}(y, z)), & r, & r' & \rangle
\end{array}$$

Rewrite rules I, K and S à la Plasmeijer

$$\begin{array}{llll|llll|llll}
r : & \mathrm{AI}x & \rightarrow & r := x & r : & \mathrm{A}sy & \rightarrow & r := x & r : & \mathrm{A}sz & \rightarrow & u : \mathrm{A}xz \\
 & & & & s : & \mathrm{A}cx & & & s : & \mathrm{A}ty & & v : \mathrm{A}yz \\
 & & & & c : & \mathrm{K} & & & t : & \mathrm{A}cx & & w : \mathrm{A}uv \\
 & & & & & & & & c : & \mathrm{S} & & r := w \\
 & & & & & & & & & & & u, v, w \text{ are fresh}
\end{array}$$

Rewriting

Computational axioms for rewiring

```
(*  replace : graph × node × node → graph *)
fun replace (g as ref (A (g0, g1)), g0', g1')
    = (g0 := g0'; g1 := g1'; g)

(*  rewire : graph × node → graph *)
fun rewire (g, x)
    = (g := x; g)
```

Rewriting

Formalization of rewriting in Standard ML

```
datatype redex
  = RED_I of graph × graph
  | RED_K of graph × graph
  | RED_S of graph × graph × graph × graph

(*  contract : redex → graph *)
fun contract (RED_I (r, ref x))
    = rewire (r, x)
  | contract (RED_K (r, ref x))
    = rewire (replace (r, C I, x), x)
  | contract (RED_S (r, ref x, ref y, ref z))
    = replace (r, A (ref x, ref z), A (ref y, ref z))
```

The rest of the story

Remaining operations: the reduction strategy

- Finding the next redex *(decomposition)*
- Rewriting the redex *(contraction)*
- Reconstructing the resulting graph *(recomposition)*
- Repeating if the result is not a normal form *(iteration)*

Usually left implicit

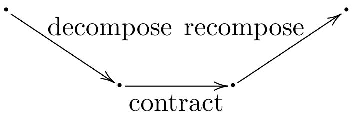
Decomposition and recomposition

```
datatype decomposition
  = NF  of graph
  | DEC of redex × context

(*  decompose : graph → decomposition *)
fun decompose g = ...

(*  recompose : graph × context → graph *)
fun recompose (g, c) = ...
```

One-step reduction

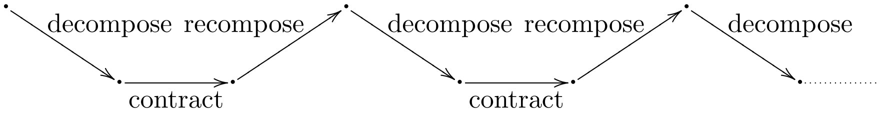


decompose recompose

contract

```
(*  reduce : graph → graph *)
fun reduce g
  = (case decompose g
      of NF g'
         ⇒ g'
       | DEC (red, c)
         ⇒ recompose (c, contract red))
```

Reduction-based normalization



```
(*  iterate : decomposition → graph *)
fun iterate (NF g)
      = g
  | iterate (DEC (g, c))
      = iterate (decompose (recompose (c, contract g)))

(*  normalize : graph → graph *)
fun normalize g
      = iterate (decompose g)
```

Overview

- Formalization of a reduction machine
- Formalization of a graph rewriting system
- Derivation
- Towards graph evaluation
- Conclusion

This work

Key observation

- Side effects are restricted to axioms
- Navigation is without pointer swapping

This work

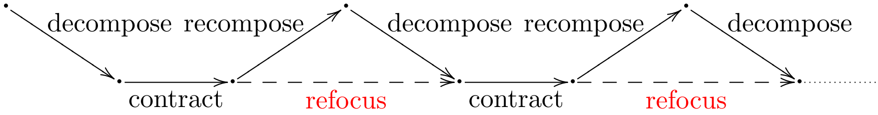
Key observation

- Side effects are restricted to axioms
- Navigation is without pointer swapping

Consequence

- Driving machinery is functional
- Amenable to the syntactic correspondence starting with refocusing

Refocusing


decompose
recompose
decompose
recompose
decompose
contract
refocus
contract
refocus

Derivation steps

Graph rewriting system

Danvy and Nielsen's refocusing

inlining

Ohori and Sasano's lightweight fusion

transition compression

Abstract machine

Result: an abstract machine

This abstract machine
coincides
with Turner's reduction machine.

Rewriting system to reduction machine

Summary

- Side effects are restricted to axioms
- Driving machinery is functional
- Syntactic correspondence applies

Overview

- Formalization of a reduction machine
- Formalization of a graph rewriting system
- Derivation
- Towards graph evaluation
- Conclusion

Towards graph evaluation

Background: Reynolds's functional correspondence.

$$\begin{tikzcd}
\text{Evaluator} \arrow[d, "\text{closure conversion}"] \\
\bullet \arrow[d, "\text{CPS transformation}"] \\
\bullet \arrow[d, "\text{defunctionalization}"] \\
\text{Abstract machine}
\end{tikzcd}$$

The functional correspondence

Danvy and students:

abstract machines $\xleftrightarrow[\text{correspondence}]{\text{functional}}$ evaluators

CEK $\longleftrightarrow$

KAM $\longleftrightarrow$

SECD $\longleftrightarrow$

$\longleftrightarrow$ monadic evaluator

Towards graph evaluation

- To defunctionalized form:
  stack marking to list of stack frames
- Refunctionalization
- Direct-style transformation

Towards graph evaluation

Result:

- A graph evaluator
- Resembles the one of Okasaki, Lee and Tarditi

Overview

- Formalization of a reduction machine
- Formalization of a graph rewriting system
- Derivation
- Towards graph evaluation
- Conclusion

Conclusion

- Danvy et al.'s syntactic correspondence

  $$\text{terms} \longrightarrow \text{graphs}$$

- Barendregt et al. and Turner's graph reduction

  $$\text{graph rewriting} \longleftrightarrow \text{reduction machines}$$

- Reynolds's functional correspondence

  $$\text{reduction machines} \longleftrightarrow \text{graph evaluators}$$

Conclusion

- Danvy et al.'s syntactic correspondence

  terms ⟶ graphs

- Barendregt et al. and Turner's graph reduction

  graph rewriting ⟷ reduction machines

- Reynolds's functional correspondence

  reduction machines ⟷ graph evaluators

Thank you