Graph reduction using combinators

Pedro Vasconcelos

March 11, 2024

Bibliography

- *Foundations for Functional Programming*, Section 7, Lawrence C. Paulson.
- *Implementation of Functional Languages*, Chapters 10, 11, 12, 16
- Video presentation @ Strange Loop Conf: https://youtu.be/GawiQQCn3bk

Strict vs. non-strict semantics

How should we evaluate $((\lambda x. M)\ N)$?

Strict semantics

Call-by-value evaluate N once (even if not used)

Non-strict semantics

Call-by-name evaluate N zero or multiple times (every time it is used)

Call-by-need, lazy evaluation evaluate N zero ou once (only if it is used)

Strict vs. non-strict semantics (cont.)

Strict semantics

- Usual in all imperative languages
- Also Scheme, Standard ML, OCaml and F#
- But: non-strict semantics is used in particular cases
  - logic operations `&&`, `||`
  - special parameters declared *call-by-name*
  - some data structures (e.g. *streams*)
  - *macros*

Non-strict semantics

- *Call-by-name* is too inefficent for general use
- *Call-by-need/lazy evaluation* some pure functional languages (Miranda, Clean, Haskell)

Dificulties in implementation

- *Call-by-name* is inefficient: duplicates computations
- *Call-by-need/lazy evaluation*:
  - we need to represent suspended computations (*thunks*)
  - after evaluation *update* the thunk with its result
  - share the result of the thunk for every use

Examples

$$\begin{aligned} \mathsf{sqr} &= \lambda x.\, \mathsf{mult}\ x\ x \\ \mathsf{main} &= \mathsf{sqr}\ (\mathsf{sqr}\ 5) \end{aligned}$$

Let us compute the expression “main”.

Call-by-name reduction

$$
\begin{aligned}
\underline{\mathsf{sqr}}\ (\mathsf{sqr}\ 5) &\to \underline{\mathsf{mult}}\ \underbrace{(\mathsf{sqr}\ 5)\ (\mathsf{sqr}\ 5)}_{\text{duplication}} \\
&\to \mathsf{mult}\ (\underline{\mathsf{sqr}}\ 5)\ (\mathsf{sqr}\ 5) \\
&\to \mathsf{mult}\ (\underline{\mathsf{mult}}\ 5\ 5)\ (\mathsf{sqr}\ 5) \\
&\to \mathsf{mult}\ 25\ (\underline{\mathsf{sqr}}\ 5) \\
&\to \mathsf{mult}\ 25\ (\underline{\mathsf{mult}}\ 5\ 5) \\
&\to \underline{\mathsf{mult}}\ 25\ 25 \\
&\to 625
\end{aligned}
$$

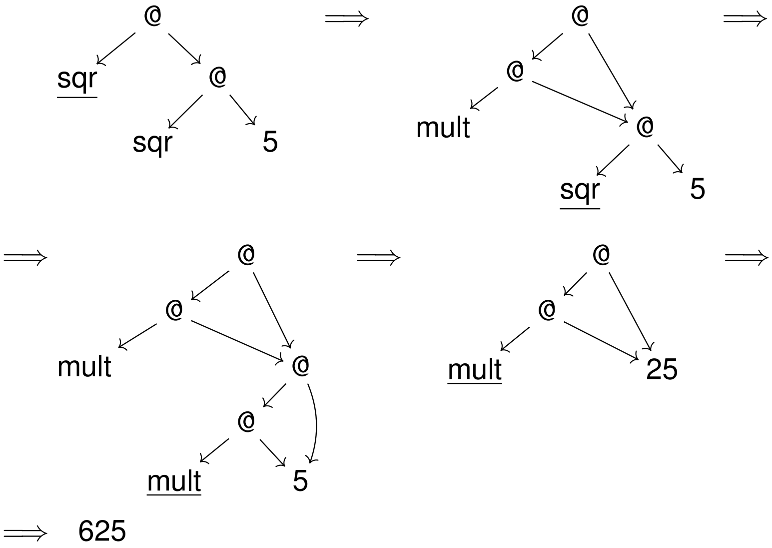
Graph reduction

- Represent the expression as a graph where applications are binary nodes:

$$\begin{tikzcd} & \texttt{@} \arrow[dl] \arrow[dr] & \\ \mathit{fun} & & \mathit{arg} \end{tikzcd}$$

- Look for the outermost and leftmost *redex*
- After reducing a sub-term: update the graph with the result

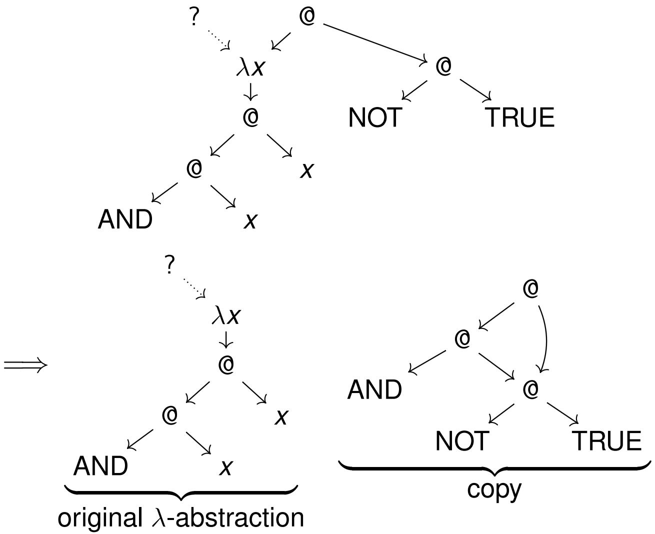
Example



Observations

- Sharing of sub-graphs avoid duplication of computations
- The graph can be implemented in memory as a linked data structure
- Each re-write step modifies pointers in this structure
- Problem: we need to copy the body of the function when we perform a β-reduction

Example


?
@
λx
@
NOT
TRUE
@
x
AND
x
?
λx
@
@
AND
@
NOT
TRUE
x
AND
x
original λ-abstraction
copy

Avoiding copying

We can avoid the need for copying if we use combinators instead of λ-terms.

Combinators

The terms of *combinatory logic* are

1. variables $x, y, z \ldots$;
2. constants $\mathbf{S}$, $\mathbf{K}$, $\mathbf{I}$;
3. applications $(M\, N)$ where M e N are terms.

Examples:

$$((\mathbf{S}\, x)\, \mathbf{I}) \qquad ((\mathbf{S}\, \mathbf{K})\, \mathbf{K}) \qquad ((\mathbf{S}\, (\mathbf{K}\, \mathbf{S}))\, \mathbf{K})$$

As in the the λ-calculus, we omit parenthesis by following the convention that applications associate to the left, e.g.

$$\mathbf{S}\, \mathbf{K}\, \mathbf{K} \equiv ((\mathbf{S}\, \mathbf{K})\, \mathbf{K})$$

Combinators (cont.)

Reduction rules $\rightarrow_w$ (*weak* reduction):

$$\begin{aligned} \mathbf{I}\,P &\rightarrow_w P \\ \mathbf{K}\,P\,Q &\rightarrow_w P \\ \mathbf{S}\,P\,Q\,R &\rightarrow_w P\,R\,(Q\,R) \end{aligned}$$

Example:

$$\mathbf{S}\,\mathbf{K}\,\mathbf{K}\,x \rightarrow_w \mathbf{K}\,x\,(\mathbf{K}\,x) \rightarrow_w x$$

Weak head normal form

A combinator term of the form

$$H\ E_1\ E_2 \ldots E_n$$

is in weak head normal form if

1. H is a constant combinator or a variable; and
2. $H\ E_1\ E_2 \ldots\ E_k$ is not a *redex* for any $k \leq n$

In WHNF:	Not in WHNF:
x	$\mathbf{K}\ x$
$\mathbf{I}$	$\mathbf{K}\ \mathbf{I}\ y$
$\mathbf{S}\ (\mathbf{K}\ x)$	$\mathbf{S}\ (\mathbf{K}\ x)\ (\mathbf{I}\ \mathbf{I})\ y$

Translation into combinators

We can translate any λ-term into combinators using two transformations:

$(-)_{CL}$ converts a λ-term into a combinator term;

$\lambda^* x$ auxiliary transformation to abstract a variable.

$$
\begin{aligned}
(x)_{CL} &\equiv x \\
(M\,N)_{CL} &\equiv (M)_{CL}(N)_{CL} \\
(\lambda x.\,M)_{CL} &\equiv \lambda^* x.(M)_{CL}
\end{aligned}
$$

$$
\begin{aligned}
\lambda^* x.\,x &\equiv \mathbf{I} \\
\lambda^* x.\,P &\equiv \mathbf{K}\,P && x \notin fv(P) \\
\lambda^* x.\,P\,Q &\equiv \mathbf{S}\,(\lambda^* x.\,P)\,(\lambda^* x.\,Q)
\end{aligned}
$$

Example translation

$$
\begin{aligned}
(\lambda xy.\, y\, x)_{CL} &\equiv \lambda^* x.\, \lambda^* y.\, (y\, x)_{CL} \\
&\equiv \lambda^* x.\, \lambda^* y.\, (y\, x) \\
&\equiv \lambda^* x.\, \mathbf{S}\, (\lambda^* y.\, y)\, (\lambda^* y.\, x) \\
&\equiv \lambda^* x.\, (\mathbf{S}\, \mathbf{I})\, (\mathbf{K}\, x) \\
&\equiv \mathbf{S}\, (\lambda^* x.\, \mathbf{S}\, \mathbf{I})\, (\lambda^* x.\, \mathbf{K}\, x) \\
&\equiv \mathbf{S}\, (\mathbf{K}\, (\mathbf{S}\, \mathbf{I}))\, (\mathbf{S}\, (\lambda^* x.\, \mathbf{K})\, (\lambda^* x.\, x)) \\
&\equiv \mathbf{S}\, (\mathbf{K}\, (\mathbf{S}\, \mathbf{I}))\, (\mathbf{S}\, (\mathbf{K}\, \mathbf{K})\, \mathbf{I})
\end{aligned}
$$

Properties of the translation

- $(M)_{CL}$ is equivalent to M (see the bibliography — *Foundations of FP*)
- The translation does *not* preserve normal forms (not important for an implementation)
- The translation λ^* may duplicate the number of applications
- The combinator term can be *exponentially larger* than the original λ-term
- The translation can be improved to *quadratic complexity* using more combinators (Turner, 1979)
- We can also need to add combinators for integers, arithmetic operations and the *fixed point combinator* (for recursion)

Improved translation

Extra combinators (Turner):

$$\mathbf{B}\, P\, Q\, R \rightarrow_w P\,(Q\,R)$$
$$\mathbf{C}\, P\, Q\, R \rightarrow_w (P\,R)\,Q$$

Improved translation (cont.)

$$\begin{aligned}
(x)_{CL} &\equiv x \\
(M\,N)_{CL} &\equiv (M)_{CL}(N)_{CL} \\
(\lambda x.\, M)_{CL} &\equiv \lambda^T x.(M)_{CL}
\end{aligned}$$

$$\begin{aligned}
\lambda^T x.\, x &\equiv \mathbf{I} & & \\
\lambda^T x.\, P &\equiv \mathbf{K}\, P & & x \notin fv(P) \\
\lambda^T x.\, P\, x &\equiv P & & x \notin fv(P) \\
\lambda^T x.\, P\, Q &\equiv \mathbf{B}\, P\, (\lambda^T x.\, Q) & & x \notin fv(P) \\
\lambda^T x.\, P\, Q &\equiv \mathbf{C}\, (\lambda^T x.\, P)\, Q & & x \notin fv(Q) \\
\lambda^T x.\, P\, Q &\equiv \mathbf{S}\, (\lambda^T x.\, P)\, (\lambda^T x.\, Q) & & x \in fv(P),\ x \in fv(Q)
\end{aligned}$$

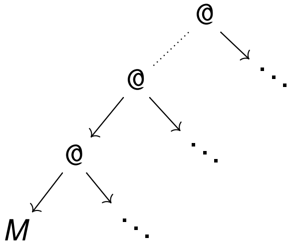
Example revisited

$$
\begin{aligned}
(\lambda xy.\, y\, x)_{CL} &\equiv \lambda^T x.\, \lambda^T y.\, (y\, x)_{CL} \\
&\equiv \lambda^T x.\, \lambda^T y.\, (y\, x) \\
&\equiv \lambda^T x.\, \mathbf{C}\, (\lambda^T y. y)\, x \\
&\equiv \lambda^T x.\, \mathbf{C}\, \mathbf{I}\, x \\
&\equiv \mathbf{C}\, \mathbf{I}
\end{aligned}
$$

Exercise: Verify that **C I** behaves like the lambda term $\lambda xy.\, y\, x$ when applied to two arguments.

Graph reduction with combinators

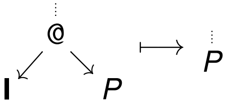
Step 1 (Unwind): Find the leftmost, outermost *redex* by descending left on the spine of @-nodes:

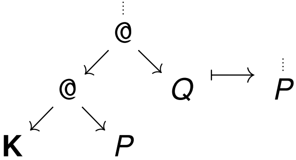


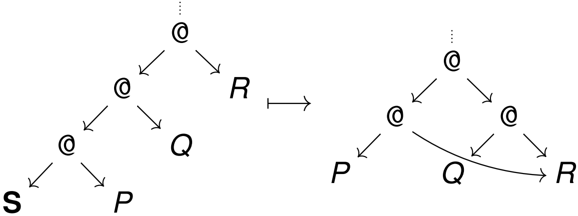
such that M is not a @-node

Graph reduction with combinators (cont.)

Step 2 (Rewrite): If we find a *redex*, perform local re-writing.

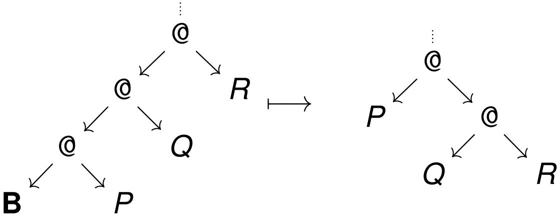


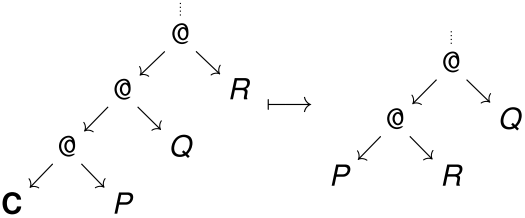




Note the *sharing* of R in the reduction for **S**.

Graph reduction with combinators (cont.)





Graph reduction with combinators (cont.)

If there are not enough arguments for the redex in Step 2, then the graph is in *weak head normal form*.

Graph reduction with combinators (cont.)

Step 3: Repeat step 1 and continue until we reach WHNF.

Graph reduction with combinators (cont.)

Strict primitive operations (e.g. ifzero, +, *, etc.) must recursively evaluate their arguments before reducing.

Reduction example

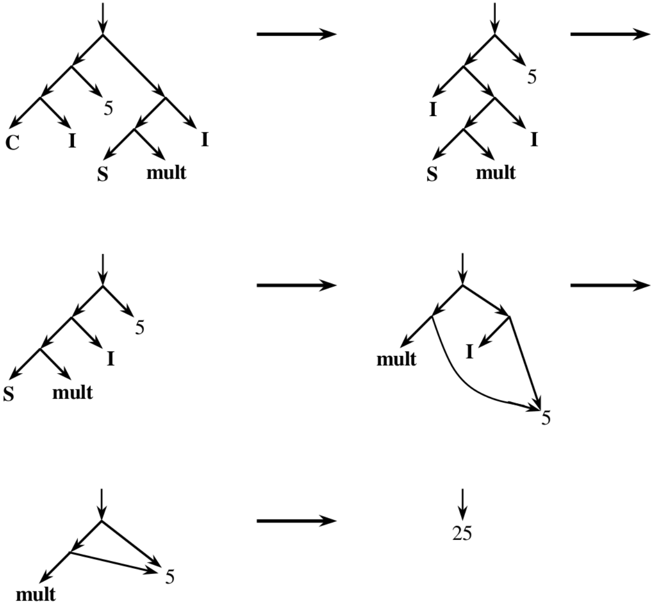
Let us translate

$$\textbf{let } sqr = \lambda x.\, x * x \textbf{ in } sqr\ 5$$

into combinators:

$$
\begin{aligned}
& (\lambda^T f.\, f\,5)\,(\lambda^T x.\, \mathbf{mul}\ x\ x) \\
\equiv & \mathbf{C}\,\mathbf{I}\,5\,(\mathbf{S}\,(\lambda^T x. \mathbf{mul}\ x)\,(\lambda^T x.\, x)) \\
\equiv & \mathbf{C}\,\mathbf{I}\,5\,(\mathbf{S}\,\mathbf{mul}\,\mathbf{I})
\end{aligned}
$$

Reduction example (cont.)



Historical context

- The translation into combinators was used in implementation of *Miranda* in the early 1980s
- Early implementations of Haskell in 1990s used translation into *supercombinators* (GHC, Gofer)
- Special *graph reductions machines* were also built between 1980-1990 (SKIM, NORMA, GRIP)
- No longer state-of-art:
  - special purpose hardware cannot compete with hardware advances in general-purpose CPUs
  - better compilation techniques employ program transformations of the functional program and run on off-the-shelf general purpose architectures
- But is *still* used e.g. in MicroHs: `https://github.com/augustss/MicroHs`