Chapter 1

On Graph Rewriting, Reduction and Evaluation

Ian Zerny[1]

Abstract: We inter-derive two prototypical styles of graph reduction: reduction machines à la Turner and graph rewriting systems à la Barendregt et al. To this end, we adapt Danvy et al.'s mechanical program derivations from the world of terms to the world of graphs. We also outline how to inter-derive a third style of graph reduction: a graph evaluator.

1.1 INTRODUCTION

Graph reduction [39] has become a key subject in the specification and implementation of modern functional programming languages. As such, there is a need for models in which one can reason about the semantics of languages that make use of graph reduction. To this end, two general approaches have been developed: the foundational application of graph reduction as described by the term graph rewriting of Barendregt et al. [4] and the use of graph reduction as an implementation technique as pioneered by Turner [38]. Both approaches give rise to semantic descriptions of their own. The semantics are constructed separately, possess different properties, and are used for different purposes. Often however, the language theoretician has the need for an efficient implementation and the language implementer has the use of a more abstract model. Thus, the problem arises of relating such semantics as exemplified for λ-terms and for Landin's SECD machine by Plotkin [16, 37].

We are motivated to consider this problem in the setting of graphs since graph reduction is closer to the actual implementations of modern functional languages, and despite this, the two approaches have yet to be fully connected. Our approach to this problem is to mechanically inter-derive graph rewriting à la Barendregt

[1]Department of Computer Science, Aarhus University, Aabogade 34, DK-8200 Aarhus N, Denmark; zerny@cs.au.dk

and reduction machines à la Turner. Hereby, we maintain an explicit connection between the two semantic artifacts.

Our domain of discourse is the applicative language of S, K and I combinators with no extensions as defined by the equations:

$$\mathrm{S}xyz = xz(yz), \quad \mathrm{K}xy = x \quad \text{and} \quad \mathrm{I}x = x.$$

Since the language is not extended with primitives, we consider normalization to normal form as opposed to head normal form or weak head normal form. Normalization to normal form ensures reduction under right branches, which incidentally accounts for the operations needed to support functions that are strict in their arguments.

1.1.1 Rewriting à la Barendregt

The work of Barendregt et al. [4, 5] provides an account of term graph rewriting as an adaptation of term rewriting that includes the notion of graph reduction. Among other things, this work is used to model sharing [21], to aid language implementation [2, 25, 28], and is part of the foundational work on graph reduction [36].

We briefly exemplify Barendregt et al.'s work with a graph rewriting system for the language of S, K and I. For a more elaborate presentation, we refer readers to the original work [4]. An expression is given by a labeled acyclic graph over the function symbols $F = \{\mathrm{S}, \mathrm{K}, \mathrm{I}, \mathrm{A}\}$. A graph is defined by a set of nodes N; a label function $lab : N \to F$ mapping nodes to labels; and a partial successor function $succ : N \rightharpoonup N \times N$ from nodes to child nodes. In our case, *succ* is defined on exactly the nodes with label A, which denotes an application, and it produces the operator and operand of the application. A rewrite rule is a triple $\langle g, r, r' \rangle$ where the first component is a graph, and the second and third are nodes, named respectively the *left root* and the *right root*. A rewrite will in part consist of redirecting the left root to the right root. The rewrite rules for I, K and S are written below in the linear notation for graphs. Note that '$+$' is used to combine graphs, resulting in a possibly disconnected graph.

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

A rule of the form $\langle g, r, r' \rangle$ is said to be a redex of a graph $\hat{g}$ if there is a label and successor preserving morphism on graphs, say f, mapping the subgraph of g rooted at r to $\hat{g}$. In other words, f must be able to construct $\hat{g}$ by filling in placeholders in r and appending the result to some other graph (possibly the empty graph). Rewriting is performed in three steps: given a redex $\langle g, r, r' \rangle$, (1) build a copy of r', sharing any nodes contained in r; (2) redirect all occurrences of r to the copy of r'; and (3) garbage collect all nodes that are no longer accessible from the root of the graph. As usual, a graph is said to be a normal form if no redex exists.

1.1.2 Reduction à la Turner

By adapting graph reduction to combinatory logic, Turner created a convenient and efficient target for the implementation of functional languages [38]. He did so by cleverly combining the simple reduction mechanism of combinatory logic with graph reduction. A considerable body of work has since followed this path in the form of alternative translation techniques [10], different sets of combinators [23], concurrent and parallel extensions [8, 29], and alternative reduction machines [9, 27, 28, 34, 35], representing the current state of the art in functional-language implementations.

Turner's scheme operated by first translating λ-terms to a graph built by a set of basic combinators amounting to the assembly language of the reduction machine. The reduction machine executes by unwinding the spine of the graph while maintaining a 'left ancestor stack'. When a left hanging atom (a leaf node) is encountered the machine applies the contraction rule for the atom in question. The contraction itself is implemented in terms of a graph transformation, where the arguments are made available through the ancestor stack. By unwinding to the left, the machine implements normal-order reduction, where functions are reduced with possibly unevaluated arguments. However, for primitive operations, such as addition, the arguments must first be fully evaluated, which requires special care in representing and manipulating the ancestor stack.

1.1.3 On term rewriting, reduction and evaluation

Languages as defined by terms[2] have received considerable attention with respect to specifications, implementations and their interconnections. Interconnecting such semantic artifacts is often done by methods tailored to the concrete semantics under consideration. Such methods may provide elegant calculational connections [24], but these connections are provided on a case by case basis. An alternative approach is to mechanically inter-derive semantic artifacts by program derivation [12]. The correctness of the connection then follows as a corollary of the correctness of the individual transformations, where the calculations have been generically done once and for all. The derivational approach we consider here has successfully been used to connect a wide range of artifacts and to reflect changes made to one in the other [7, 14, 16, 26].

1.1.4 On graph rewriting, reduction and evaluation

In this article, we present a systematic method to derive a reduction machine from a graph rewriting system. We do so by adapting the methods used in the setting of term rewriting and evaluation to the new setting of graph reduction. Our work hinges on the fact that each inter-derived semantic artifact gives rise to the same trace of successive contractions [16]: the inter-derivation acts only on the functional glue between each contraction. We illustrate the method by deriving a

[2] By terms we mean trees where each node has at most one parent.

reduction machine for the language of S, K and I combinators as the direct result of a series of mechanical program transformations on a graph rewriting system for the same language. Our presentation focuses on the foundational aspects of graph reduction, and we will not be treating any issues with respect to high-performance reduction machines and language implementation. Our contribution is to connect the graph rewriting system for S, K and I à la Barendregt with the reduction machine for S, K and I à la Turner.

Prerequisites. All of the semantic specifications are presented as implementations in Core Standard ML with additional use of references to account for the implementation of the formal rewriting axioms. Readers acquainted with Standard ML [30] or a related functional language are equipped to follow the presentation. The reader should be familiar with the derivational approach of Danvy et al. [12, 13], in particular refocusing, lightweight fusion, refunctionalization, and the direct-style transformation. Some prior experience with combinatory logic, term rewriting, graph rewriting and reduction machines is preferable and can be obtained from many sources [3, 11, 34, 36].

Overview. The rest of this article is structured as follows. In Section 1.2, we begin by presenting a reduction machine for the language of S, K and I combinators that closely matches the original presentation of Turner. Then we take the calculus of S, K and I combinators in the style of Barendregt et al. and implement a full graph rewriting system in Section 1.3, where we have made explicit all of the implicit operations of such a system, using the same data types as in Section 1.2. With the graph rewriting system as starting point, in Section 1.4, we systematically derive the reduction machine of Section 1.2, presenting all intermediate transformations. Finally, in Section 1.5, we consider further transformations to the reduction machine and compare the results to other known artifacts. We conclude in Section 1.6. The complete derivation, along with tests, can be found on the author's home page.

1.2 GRAPH REDUCTION

In this section, we present a reduction machine à la Turner in the form of a state transition system. We start with the type for graphs. A graph is a reference to a node that can either be an atom, corresponding to a basic combinator, or an application of two graphs. We define graphs with the following ML data types, where a node-id is represented by an ML reference, i.e., a location in memory:

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

In order to reduce the right branch of a graph, we need to store information such that the state can be reestablished after completing the reduction of the branch. We cannot recursively unwind on the right branch, as Turner does

```
(*  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)

(*  normalize_RM : graph → graph *)
fun normalize_RM g
    = unwind_RM (g, EMPTY)
```

FIGURE 1.1. Reduction machine for S, K and I à la Turner

for functions with strict arguments, since such a call would violate the nature of a transition system as traditionally used to specify an abstract machine. Instead, we mark the ancestor stack and when completing reduction on the right branch we pop the mark and reestablish the active atom. By marking the stack, we separate the arguments of each function application. More on the various techniques for managing the spine stack can be found in Peyton Jones's treatise [34]. Our stack scheme gives rise to the following data type:

```
datatype stack
  = EMPTY
  | PUSH of graph × stack
  | MARK of atom  × graph × stack
```

Figure 1.1 displays the entire reduction machine as a state transition system with two functions $\texttt{unwind}_{RM}$ and $\texttt{apply}_{RM}$ that implement the operations described in Section 1.1.2. One issue arises when reducing K. The machine cannot replace the top most application node with the already existing graph given by x. Instead, following Turner [38, p. 43], an indirection node is installed in the form of an application of I to x and the machine proceeds to rewire the top of stack to point at x.

1.3 GRAPH REWRITING

In this section, we develop an implementation that is faithful to the graph rewriting system à la Barendregt as described in Section 1.1.1. We do so by investigating each of the required steps in turn, making explicit the algorithms and data structures involved following the style and terminology of Danvy et al. [12, 13].

We start by reusing the type of graphs from the previous section. For any instance of type `graph`, we can construct unique maps for

$$lab : \texttt{graph} \longrightarrow F$$
$$succ : \texttt{graph} \rightharpoonup \texttt{graph} \times \texttt{graph}$$

and ML references aptly account for a set of unique node identifiers. Thus, the implementation faithfully accounts for a graph.

Decomposition. We must consider how to find and represent a redex. To find a redex we need a method of traversing the graph. While traversing, we must maintain the graph structure relative to our current position in order to check for a matching rule. This is naturally done, in a functional style, with a zipper [22] using the following type of context:

```
datatype context
  = CTX_MT
  | CTX_L of graph × graph ×         context (* id × right       × context *)
  | CTX_R of graph × graph × atom × context (* id × left × atom × context *)
```

Here, `CTX_L` marks a traversal on a left branch, where we store the right branch along with the current node-id. Likewise, we store the left branch and node-id for `CTX_R` and in addition maintain the current function symbol. This addition is for purely practical reasons because it allows for a better typing discipline while decomposing the graph. We could dispense with the function symbol as well as the left and right branches since they are all accessible through the node-id reference.

We can implement a leftmost-outermost reduction strategy by a depth-first search of the graph. The search is implemented by the two mutually recursive functions: `decompose_graph` and `decompose_context`. The first dispatches on a graph to find the leftmost atom, and the second dispatches on the current context and atom checking for a matching redex. If the search fails, the graph is a normal form. If a redex is found, we pair the redex with its context as reflected in the `decomposition` data type. A redex is a match of the I-rule, K-rule or S-rule as described in Section 1.1.1. The rules imply that we must keep track of the left root, in order to redirect; and also, any node reachable from both the left and right root (roots inclusive), since such nodes must be shared. In the case of K, we must keep track of r and x. The entire process is implemented by the following definitions:

```
datatype redex
  = RED_I of graph × graph                   (* r × x         *)
  | RED_K of graph × graph                   (* r × x         *)
  | RED_S of graph × graph × graph × graph   (* r × x × y × z *)
```

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

(*  decompose_graph : graph × context → decomposition *)
fun decompose_graph (g as ref (A (g0, g1)), c)
    = decompose_graph (g0, CTX_L (g, g1, c))
  | decompose_graph (g as ref (C a), c)
    = decompose_context (c, g, a)

(*  decompose_context : context × graph × atom → decomposition *)
and decompose_context (CTX_MT, g, a)
    = NF g
  | decompose_context (CTX_L (r, x, c), _, I)
    = DEC (RED_I (r, x), c)
  | decompose_context (CTX_L (_, x, CTX_L (r, y, c)), _, K)
    = DEC (RED_K (r, x), c)
  | decompose_context (CTX_L (_, x, CTX_L (_, y, CTX_L (r, z, c))), _, S)
    = DEC (RED_S (r, x, y, z), c)
  | decompose_context (CTX_L (g, g1, c), g0, a)
    = decompose_graph (g1, CTX_R (g, g0, a, c))
  | decompose_context (CTX_R (g, g0, a, c), g1, _)
    = decompose_context (c, g, a)

(*  decompose : graph → decomposition *)
fun decompose g
    = decompose_graph (g, CTX_MT)
```

Note that `decompose`, which implements the decomposition process over acyclic graphs, is a pure and total function.

Contraction. Next we implement the rewriting axioms of the system. Recall that rewriting consisted of three phases: building, redirecting and garbage collection. We omit the treatment of garbage collection and simply rely on the underlying runtime system of Standard ML to collect unreachable graphs. For redirecting nodes we use two auxiliary procedures: `replace` that in-place updates the contents of an application node, and `rewire` that redirects a single reference. Their definitions are:

```
(*  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)
```

For a K-redex, we have nothing to build and proceed to redirect all references of the root of K, named *r*, to references of *x*. As noted in Section 1.2, the root of K is an application node and we cannot simply overwrite it by *x*. Instead we replace it with an indirection node and only rewire the immediate reference to the root. For an S-redex, we construct nodes for the parts not shared between the left and right root of the rewrite rule and then replace the root application with the newly constructed nodes. The rewriting axioms are implemented by the following definition:

```
(*  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))
```

Note that `contract`, which implements the rewriting axioms, is an impure and total function: impure since effects are used for in-place rewriting and total since contraction is defined on all redexes and as such does not give rise to stuck graphs.

Recomposition. After decomposing a graph and contracting the redex, we need to recreate the graph. Since contraction actually modifies the graph in place, recomposing is simply finding the root of the graph:

```
(*  recompose : context × graph → graph *)
fun recompose (CTX_MT, t)
    = t
  | recompose (CTX_L (g, g1, c), g0)
    = recompose (c, g)
  | recompose (CTX_R (g, g0, a, c), g1)
    = recompose (c, g)
```

Note that `recompose`, which implements the recomposition process, is a pure and total function.

One-step reduction. We implement one-step reduction as the process of decomposing, contracting and recomposing:

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

Normalization. We implement normalization as the iteration of one-step reduction. Since it is built on normal-order reduction, this iteration is known to terminate with a normal form should one exist. This result generalizes to the setting of graphs since normal-order reduction in combinatory logic is hypernormalizing [4]. Normalization is implemented by the following definitions:

```
(*  iterate0 : decomposition → graph *)
fun iterate0 (NF g)
    = g
  | iterate0 (DEC (g, c))
    = iterate0 (decompose (recompose (c, contract g)))

(*  normalize0 : graph → graph *)
fun normalize0 g
    = iterate0 (decompose g)
```

This concludes our implementation of the graph rewriting system. For each step, we have done no more then make explicit the operations that are implicit in the abstract account – remaining faithful to the calculus à la Barendregt.

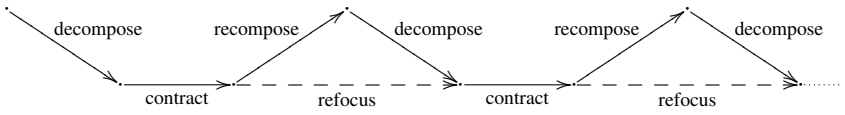
```
(*  refocus1 : graph × context → decomposition *)
fun refocus1 (g, c)
    = decompose_graph (g, c)

(*  iterate1 : decomposition → graph *)
fun iterate1 (NF g)
    = g
  | iterate1 (DEC (g, c))
    = iterate1 (refocus1 (contract g, c))

(*  normalize1 : graph → graph *)
fun normalize1 g
    = iterate1 (refocus1 (g, CTX_MT))
```

FIGURE 1.2. Small-step abstract machine obtained by refocusing

1.4 CONNECTING GRAPH REWRITING AND GRAPH REDUCTION

With the graph rewriting system of Section 1.3 as our starting point, we successively submit it to the program transformations of Biernacka and Danvy's syntactic correspondence [6, 7] lifted to the level of graphs. For an overview of the program transformations we refer to the work of Danvy et al. [12, 13].

Refocusing. Our first step is to refocus the reduction-based normalization from the previous section. Refocusing avoids repeated decomposition and recomposition, in effect deforesting the intermediate results. The result is a reduction-free normalization function that directly finds the next redex without first navigating to the top of the graph.



As mentioned in the previous section, recomposition and decomposition are pure and total functions and thus their composition is pure and total as well. In particular, so is their deforested composition: side effects are only used for the formal rewriting axioms of the system and are confined to `contract`. The deforested composition we choose is the optimal one due to Danvy and Nielsen [19], which simply consists in continuing the decomposition at the contraction site. The result is an abstract machine iterating contraction and refocusing. More precisely, as displayed in Figure 1.2, it is a small-step abstract machine with $\texttt{refocus}_1$ (which is pure) and then `contract` (which is impure) as the transition function and a 'driver loop', $\texttt{iterate}_1$.

Contraction unfolding. We then unfold `contract` into $\texttt{iterate}_1$ resulting in $\texttt{iterate}_2$, which case discriminates on the redex of a decomposition with one case per contraction rule:

```
(* iterate2 : decomposition → graph *)
fun iterate2 (NF g)
    = g
  | iterate2 (DEC (RED_I (r, ref x), c))
    = iterate2 (refocus2 (rewire (r, x), c))
  | iterate2 (DEC (RED_K (r, ref x), c))
    = iterate2 (refocus2 (rewire (replace (r, C I, x), x), c))
  | iterate2 (DEC (RED_S (r, ref x, ref y, ref z), c))
    = iterate2 (refocus2 (replace (r, A (ref x, ref z),
                                      A (ref y, ref z)), c))
```

Lightweight fusion. By lightweight fusion [15,31] of $\texttt{iterate}_2$ and $\texttt{refocus}_2$ (as defined by `decompose_graph` and `decompose_context`) we transform the small-step abstract machine into a big-step abstract machine in the sense that $\texttt{iterate}_3$, $\texttt{refocus\_graph}_3$, and $\texttt{refocus\_context}_3$ have become transition functions as shown in Figure 1.3. Here $\texttt{refocus\_graph}_3$ is the composition of $\texttt{iterate}_2$ and `decompose_graph`, while $\texttt{refocus\_context}_3$ is the composition of $\texttt{iterate}_2$ and `decompose_context` that directly calls $\texttt{iterate}_3$ instead of returning to the caller.

Compression of corridor transitions. We proceed to compress the corridor transitions, meaning we inline any call with a uniquely known target. For example, the call $\texttt{iterate}_3\texttt{(NF g)}$ is known to be handled by the first case in $\texttt{iterate}_3$ and we therefore replace it with `g`. Completing this process we obtain the code in Figure 1.4 where the iteration process has been completely inlined.

Notice how similar the machine of 1.4 is to the reduction machine of Section 1.2. The two are in fact the same where $\texttt{refocus\_graph}_4$ is replaced by $\texttt{unwind}_{RM}$; $\texttt{refocus\_context}_4$ is replaced by $\texttt{apply}_{RM}$ with permuted arguments; the type `context` is replaced by `stack` with an extra (and unneeded) graph component in `CTX_L` and `CTX_R`; and the definitions of `replace` and `rewire` have been inlined.

Thus, we have directly derived Turner's reduction machine from an implementation of Barendregt et al.'s graph rewriting system, using a series of simple and mechanical program transformations. This derivation is significant in two ways: (1) it connects Turner's reduction machine and Barendregt et al.'s graph rewriting system, which is new; and (2) it shows that Biernacka and Danvy's syntactic correspondence scales to term graph rewriting, which is also new.

1.5 TOWARDS GRAPH EVALUATION

In this section, we briefly investigate whether Reynolds's functional correspondence [1], which further connects abstract machines and evaluators for terms, can supply an evaluational counterpart for reduction machines.

The reduction machine of Section 1.2 is not in defunctionalized form [18]. However, we can obtain a reduction machine that is in defunctionalized form by using an alternative but equivalent stack scheme:

```
(*  refocus_graph3 : graph × context → graph *)
fun refocus_graph3 (g as ref (A (g0, g1)), c)
    = refocus_graph3 (g0, CTX_L (g, g1, c))
  | refocus_graph3 (g as ref (C a), c)
    = refocus_context3 (c, g, a)

(*  refocus_context3 : context × graph × atom → graph *)
and refocus_context3 (CTX_MT, g, a)
    = iterate3 (NF g)
  | refocus_context3 (CTX_L (r, x, c), _, I)
    = iterate3 (DEC (RED_I (r, x), c))
  | refocus_context3 (CTX_L (_, x, CTX_L (r, y, c)), _, K)
    = iterate3 (DEC (RED_K (r, x), c))
  | refocus_context3 (CTX_L (_, x, CTX_L (_, y, CTX_L (r, z, c))), _, S)
    = iterate3 (DEC (RED_S (r, x, y, z), c))
  | refocus_context3 (CTX_L (g, g1, c), g0, a)
    = refocus_graph3 (g1, CTX_R (g, g0, a, c))
  | refocus_context3 (CTX_R (g, g0, a, c), g1, _)
    = refocus_context3 (c, g, a)

(*  iterate3 : decomposition → graph *)
and iterate3 (NF g)
    = g
  | iterate3 (DEC (RED_I (r, ref x), c))
    = refocus_graph3 (rewire (r, x), c)
  | iterate3 (DEC (RED_K (r, ref x), c))
    = refocus_graph3 (rewire (replace (r, C I, x), x), c)
  | iterate3 (DEC (RED_S (r, ref x, ref y, ref z), c))
    = refocus_graph3 (replace (r, A (ref x, ref z),
                                  A (ref y, ref z)), c)

(*  normalize3 : graph → graph *)
fun normalize3 g
    = refocus_graph3 (g, CTX_MT)
```

FIGURE 1.3. Big-step abstract machine obtained by lightweight fusion

```
(*  refocus_graph4 : graph × context → graph *)
fun refocus_graph4 (g as ref (A (g0, g1)), c)
    = refocus_graph4 (g0, CTX_L (g, g1, c))
  | refocus_graph4 (g as ref (C a), c)
    = refocus_context4 (c, g, a)

(*  refocus_context4 : context × graph × atom → graph *)
and refocus_context4 (CTX_MT, g, a)
    = g
  | refocus_context4 (CTX_L (r, ref x, c), _, I)
    = refocus_graph4 (rewire (r, x), c)
  | refocus_context4 (CTX_L (_, ref x, CTX_L (r, y, c)), _, K)
    = refocus_graph4 (rewire (replace (r, C I, x), x), c)
  | refocus_context4 (CTX_L   (_, ref x,
                       CTX_L  (_, ref y,
                        CTX_L (r, ref z, c))), _, S)
    = refocus_graph4 (replace (r, A (ref x, ref z),
                                  A (ref y, ref z)), c)
  | refocus_context4 (CTX_L (g, g1, c), g0, a)
    = refocus_graph4 (g1, CTX_R (g, g0, a, c))
  | refocus_context4 (CTX_R (g, g0, a, c), g1, _)
    = refocus_context4 (c, g, a)

(*  normalize4 : graph → graph *)
fun normalize4 g
    = refocus_graph4 (g, CTX_MT)
```

FIGURE 1.4. Reduction machine obtained by transition compression

```
datatype stack_context
  = INIT  of graph
  | FRAME of stack × atom × stack_context
withtype stack = graph list
```

This reduction machine, shown in Figure 1.5, uses stack frames to manage right branches in the graph. A frame simply wraps the former parameters of $\texttt{apply}_{RM}$, and $\texttt{unwind}_{RM}$ constructs the frame prior to application.

Now the type `stack_context` together with the function $\texttt{apply}_{DF}$ is the first-order implementation of a higher-order function. Refunctionalizing [17] this machine gives an evaluator in continuation-passing style where all continuations are used in a linear fashion. The direct-style counterpart of this evaluator is shown in Figure 1.6. This evaluator handles right branching by recursively unwinding on the right hanging node, akin to Turner's original reduction machine. The evaluator also bears considerable resemblance to the graph reducer of Okasaki et al. [32], except they consider the Spineless G-machine [9], and they therefore have a treatment of closures and assignment that is unlike that of our evaluator. In short, this preliminary investigation suggests that Reynolds's functional correspondence scales to connecting graph reduction and graph evaluation.

1.6 CONCLUSION AND PERSPECTIVES

We have presented the first mechanical inter-derivation of graph rewriting, graph reduction, and graph evaluation. Based on the restrictive use of side effects, this derivation adapts Biernacka and Danvy's syntactic correspondence to the setting of graphs as opposed to terms. In so doing, we have connected the graph rewriting systems of Barendregt et al. to the reduction machines of Turner.

We have considered the simplest possible setting: combinatory logic with just the basic combinators S, K and I. However, in our experience, the methods scale to more involved settings. It is our experience that different rewriting systems lead to different variants of reduction machines. Furthermore, the methods provide the possibility of incrementally refining either of the semantic artifacts such that the refinements are reflected constructively in the derived semantic counterparts.

Future work would expand on the notions of graph rewriting, graph reduction, and graph evaluation as has already been successful for terms. We have sketched one such possibility in one treatment of graph evaluation. With this work, the many reduction machines could be connected to their foundational counterparts for better comparison and reasoning. We are also interested in the issue of factoring abstract graph machines into compilers and virtual graph machines. All of the above is being investigated in the author's forthcoming thesis [40].

Acknowledgments. Thanks to Olivier Danvy for his supervision and for his course on functional programming at Aarhus University from which this work originates. I am also grateful to Dennis Decker Jensen and the anonymous reviewers for their comments.

```
(*  unwind_DF : graph × stack × stack_context → graph *)
fun unwind_DF (g as ref (A (g0, g1)), gs, c)
    = unwind_DF (g0, g :: gs, c)
  | unwind_DF (ref (C a), gs, c)
    = apply_DF (FRAME (gs, a, c))

(*  apply_DF : stack_context → graph *)
and apply_DF (INIT g) = g
  | apply_DF (FRAME (gs, a, c))
    = (case (a, gs)
        of (I, (r as ref (A (_, x))) :: gs)
           ⇒ (r := !x;
              unwind_DF (r, gs, c))
         | (K,       ref (A (_,  x)) ::
                     (r as ref (A (g0, y as g1))) :: gs)
           ⇒ (g0 := C I; g1 := !x; r := !x;
              unwind_DF (r, gs, c))
         | (S,       ref (A (_,  x)) ::
                     ref (A (_,  y)) ::
                     (r as ref (A (g0, z as g1))) :: gs)
           ⇒ (g0 := A (ref (!x), ref (!z));
              g1 := A (ref (!y), ref (!z));
              unwind_DF (r, gs, c))
         | (_, (ref (A (_, g1))) :: gs)
           ⇒ unwind_DF (g1, [], FRAME (gs, a, c))
         | (_, [])
           ⇒ apply_DF c)

(*  normalize_DF : graph → graph *)
fun normalize_DF g
    = unwind_DF (g, [], INIT g)
```

FIGURE 1.5. Reduction machine in defunctionalized form

```
(*  unwind_GE : graph × stack → graph *)
fun unwind_GE (g as ref (A (g0, g1)), gs)
    = unwind_GE (g0, g :: gs)
  | unwind_GE (ref (C a), gs)
    = let fun apply_GE gs
              = (case (a, gs)
                   of (I, (r as ref (A (_, x))) :: gs)
                      ⇒ (r := !x;
                         unwind_GE (r, gs))
                    | (K,       ref (A (_,  x)) ::
                                (r as ref (A (g0, y as g1))) :: gs)
                      ⇒ (g0 := C I; g1 := !x; r := !x;
                         unwind_GE (r, gs))
                    | (S,       ref (A (_,  x)) ::
                                ref (A (_,  y)) ::
                                (r as ref (A (g0, z as g1))) :: gs)
                      ⇒ (g0 := A (ref (!x), ref (!z));
                         g1 := A (ref (!y), ref (!z));
                         unwind_GE (r, gs))
                    | (_, (ref (A (_, g1))) :: gs)
                      ⇒ (unwind_GE (g1, []); apply_GE gs)
                    | (_, [])
                      ⇒ ())
      in apply_GE gs
      end

(*  normalize_GE : graph → graph *)
fun normalize_GE g
    = (unwind_GE (g, []); g)
```

FIGURE 1.6. Graph evaluator in direct style

REFERENCES

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In D. Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, Aug. 2003. ACM Press.

[2] Z. M. Ariola and Arvind. Graph rewriting systems. Technical Report Computation Structures Group Memo 323-1, Laboratory for Computer Science, Massachusetts Institution of Technology, Nov. 1992.

[3] H. P. Barendregt. *Functional Programming and Lambda Calculus*, volume B, chapter 7, pages 321–364. The MIT Press, 1990.

[4] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In de Bakker et al. [20], pages 141–158.

[5] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Towards an intermediate language based on graph rewriting. In de Bakker et al. [20], pages 159–175.

[6] M. Biernacka and O. Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):1–30, 2007. Article #6. Extended version available as the research report BRICS RS-06-3.

[7] M. Biernacka and O. Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part II: Reduction semantics and abstract machines. In Palsberg [33], pages 186–206.

[8] T. Bülck, A. Held, W. E. Kluge, S. Pantke, C. Rathsack, S.-B. Scholz, and R. Schröder. Experience with the implementation of a concurrent graph reduction system on an nCube/2 platform. In B. Buchberger and J. Volkert, editors, *CONPAR*, volume 854 of *Lecture Notes in Computer Science*, pages 497–508. Springer, 1994.

[9] G. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless G-machine. In R. C. Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 244–258, Snowbird, Utah, July 1988. ACM Press.

[10] F. W. Burton. A linear space translation of functional programs to Turner combinators. *Inf. Process. Lett.*, 14(5):201–204, 1982.

[11] H. B. Curry, J. R. Hindley, and R. Feys. *Combinatory Logic: Volume II*. North Holland, 1972.

[12] O. Danvy. Defunctionalized interpreters for programming languages. In P. Thiemann, editor, *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, SIGPLAN Notices, Vol. 43, No. 9, pages 131–142, Victoria, British Columbia, Sept. 2008. ACM Press. Invited talk.

[13] O. Danvy. From reduction-based to reduction-free normalization. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming, Sixth International School*, number 5382 in Lecture Notes in Computer Science, pages 66–164, Nijmegen, The Netherlands, May 2008. Springer-Verlag. Lecture notes including 70+ exercises.

[14] O. Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part I: Denotational semantics, natural semantics, and abstract machines. In Palsberg [33], pages 162–185.

[15] O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008.

[16] O. Danvy and K. Millikin. A rational deconstruction of Landin's SECD machine with the J operator. *Logical Methods in Computer Science*, 4(4:12):1–67, Nov. 2008.

[17] O. Danvy and K. Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8):534–549, 2009. Extended version available as the research report BRICS RS-08-04.

[18] O. Danvy and L. R. Nielsen. Defunctionalization at work. In H. Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, Sept. 2001. ACM Press. Extended version available as the research report BRICS RS-01-23.

[19] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, Aarhus, Denmark, Nov. 2004. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[20] J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors. *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 15-19, 1987, Proceedings*, volume 259 of *Lecture Notes in Computer Science*. Springer, 1987.

[21] J. R. W. Glauert, R. Kennaway, and M. R. Sleep. Dactl: An experimental graph rewriting language. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 378–395. Springer, 1990.

[22] G. Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[23] J. Hughes. Super combinators: A new implementation method for applicative languages. In D. P. Friedman and D. S. Wise, editors, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, Pennsylvania, Aug. 1982. ACM Press.

[24] G. Hutton and J. Wright. Calculating an Exceptional Machine. In H.-W. Loidl, editor, *Trends in Functional Programming*, volume 5. Intellect, Feb. 2006. Selected papers from the Fifth Symposium on Trends in Functional Programming, Munich, November 2004.

[25] A. Jeffrey. A fully abstract semantics for concurrent graph reduction. In *LICS*, pages 82–91. IEEE Computer Society, 1994.

[26] J. Johannsen. An investigation of Abadi and Cardelli's untyped calculus of objects. Master's thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, June 2008. BRICS research report RS-08-6.

[27] T. Johnsson. Efficient compilation of lazy evaluation. In S. L. Graham, editor, *Proceedings of the 1984 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 19, No 6, pages 58–69, Montréal, Canada, June 1984. ACM Press.

[28] P. W. M. Koopman. *Functional Programs as Executable Specifications*. PhD thesis, University of Nijmegen, 1990.

[29] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. Michaelson, R. Pena, S. Priebe, Á. J. R. Portillo, and P. W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.

[30] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[31] A. Ohori and I. Sasano. Lightweight fusion by fixed point promotion. In M. Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 42, No. 1, pages 143–154, Nice, France, Jan. 2007. ACM Press.

[32] C. Okasaki, P. Lee, and D. Tarditi. Call-by-need and continuation-passing style. In C. L. Talcott, editor, *Special issue on continuations (Part II)*, Lisp and Symbolic Computation, Vol. 7, No. 1, pages 57–82, 1994.

[33] J. Palsberg, editor. *Semantics and Algebraic Specification: Essays dedicated to Peter D. Mosses on the occasion of his 60th birthday*, number 5700 in Lecture Notes in Computer Science. Springer-Verlag, 2009.

[34] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1987.

[35] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[36] M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[37] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[38] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, 1979.

[39] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Programming Research Group, Oxford University, 1971.

[40] I. Zerny. Master's thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, 2010. Forthcoming.