Memoïzation for Term (Hypergraph) Rewriting

Berthold Hoffmann
Universität Bremen*

(Extended Abstract)

Introduction

Memoïzation is an optimization technique for implementing functional programming languages: The result of a function application is stored in a table, in order to be looked up later on if the function is re-applied to equal arguments. Redundant re-evaluation of functions is avoided in this way, providing a side-effect-free notion of memory.

Commonly, memoïzation has been considered for the rather restricted classes of rewrite rules allowed in functional languages, and its semantics has been defined on the implementation level, usually depending on the evaluation strategy employed.

In this paper, we generalize memoïzation considerably:

- We consider two general classes of rules: convergent, and orthogonal rewrite systems.
- We define its semantics by a simple modification of the evaluation rules.
- Our results do not suppose a particular evaluation strategy.

The present paper goes beyond earlier work described in [Hof92]: Full memoïzation is extended to convergent rewrite systems, and lazy memoïzation for orthogonal rewrite systems is extended to cyclic graphs, representing infinite rational terms.

In what follows, we assume some familiarity with basic notions of (term) rewriting.

*Address: Fachbereich Mathematik und Informatik, Universität Bremen, Postfach 330 440, D-28334 Bremen, Germany. Usenet: hof%Informatik.Uni-Bremen.DE. Work is supported by the ESPRIT Basic Research Working Group COMPUGRAPH, Computing by Graph Transformation, under contract 7183.
Term Graphs

Hypergraphs are an extension of graphs where edges may connect an arbitrary number of nodes. In our setting, edges are applications (depicted by their label, a function symbol) that connect a value node (drawn above it) to an ordered sequence of argument nodes (drawn below it, from left to right, and connected by lines to the application).

A hypergraph is called a term graph if all its nodes are the value of at most one application. Nodes which are not value nodes are called open, and term graphs without open nodes are closed. Acyclic term graphs are called jungles.

We consider pointed term graphs and jungles. That is, we distinguish a result node in a graph (depicted by a filled circle). Only the applications and nodes contributing to this node matter for rewriting, whereas the rest is considered as garbage, and ignored.

For example, all hypergraphs depicted in the example below are pointed jungles. Pointed term graphs are collapsed representations of terms. In particular, cyclic term graphs represent rational terms — infinite terms with only finitely many distinct subterms. The unravelling of a term graph $T$ to a term is unique up to garbage, and denoted by $\hat{T}$.

Rewriting

Graph rewrite rules are straight-forward adaptations of term rewrite rules.

A pair $L \rightarrow R$ of pointed jungles is a rewrite rule if $L$ contains at least one application and is garbage-free, and if all open nodes of $R$ are in the interface graph $I = L \cap R$.

Rewriting a pointed term graph $T$ to a pointed term graph $U$ via some rewrite rule $L \rightarrow R$ works as follows:

- Identify a (possibly collapsed) image $\tilde{L}$ of $L$ in $T$, and remove the image of $L$'s top application.

- Add a copy of $R$ to this graph, and obtain $U$ by gluing the interface $I$ with its image in $\tilde{L}$, and by gluing $R$'s result node with the image of $L$'s result node.

(Formally, this is a special case of the double-pushout approach to graph grammars, see [Ehr79].)

For the rest of this paper, $\Rightarrow_R$ denotes rewriting via an arbitrary, fixed set $R$ of rewrite rules.

Collapsing

For jungles, the degree of sharing can be increased by collapsing rules $C$ that remove duplicate applications of some function to the same argument nodes, and identify their result nodes (see the example below). The collapsing relation $\Rightarrow_C$ induced by
these rules is strongly normalizing and confluent, and achieves maximal sharing of subterms.\footnote{This relation was called \textit{folding} in previous papers.}

\textbf{Example.} The function \texttt{FIB} computing Fibonacci numbers has the following recursive rewrite rule, and collapsing rule (dashed boxes indicate the applications to be removed):

\begin{center}
\begin{tikzpicture}
  \node (fib) {\texttt{FIB}};
  \node (plus) [right of=fib, xshift=0.5cm] {$+$};
  \node (s) [below of=fib, yshift=-0.5cm] {$S$};
  \node (plus_s) [right of=s, xshift=0.5cm] {$S$ \texttt{FIB}};
  \node (fib_s) [right of=plus_s, xshift=0.5cm] {$\texttt{FIB}$};
  \path (fib) edge (s);
  \path (s) edge (plus);
  \path (plus) edge (plus_s);
  \path (plus_s) edge (fib_s);
\end{tikzpicture}
\end{center}

\textbf{Jungle Evaluation ([HP91, Plu93])}

We combine rewriting and collapsing to the \textit{evaluation} relation $\Rightarrow_{\mathcal{R}}$ on closed pointed jungles.

Then every rewriting $T \Rightarrow_{\mathcal{R}} U$ models parallel term rewriting $\widehat{T} \Rightarrow_{\mathcal{R}} \widehat{U}$, at all copies of $\widehat{L}$ in $\widehat{T}$. This is equivalent to iterated sequential term rewriting $\overline{T} \Rightarrow_{\mathcal{R}} \overline{U}$. Every collapsing $T \Rightarrow_{\mathcal{C}} U$ preserves terms, i.e. $\overline{T} = \overline{U}$.

If term rewriting is convergent (i.e. strongly normalizing and confluent), jungle evaluation is convergent up to garbage. An evaluation $T \Rightarrow^{*}_{\mathcal{R}} U$ to normalform implements parallel term rewriting $\overline{T} \Rightarrow^{*}_{\mathcal{R}} \overline{U}$, where $\overline{U}$ is in normalform as well.

\textbf{Term Graph Rewriting ([Cor93])}

Let the rewrite rules $\mathcal{R}$ be orthogonal (i.e. have left hand sides that are trees and cannot overlap with each other). We consider the rewriting relation $\Rightarrow_{\mathcal{R}}$ on pointed term graphs. (Collapsing is dropped in this case, because it cannot achieve sharing of equal cyclic subgraphs.)

Then rewriting $T \Rightarrow_{\mathcal{R}} U$ within a cycle models parallel term rewriting $\overline{T} \Rightarrow_{\mathcal{R}} \overline{U}$, at infinitely many occurrences. However, this does not always correspond to a transfinite sequential rewriting $\overline{T} \Rightarrow^{*}_{\mathcal{R}} \overline{U}$ as in the Semagraph approach, see [KKSdV90]. Rather, it models \textit{continuous term rewriting} in the sense of [Cor93].

Term graph rewriting is (strongly) confluent. Every normalizing rewriting $T \Rightarrow^{*}_{\mathcal{R}} U$ yielding a cyclic term graph $U$ then implements a parallel term rewriting of $\overline{T}$ that converges to an infinite normalform $\overline{U}$, in the sense defined by [Cor93].
Term Graphs with Memory

In order to realize memoization, we need a table for the values of function applications. This table is represented by additional entry edges (drawn in a box) that correspond to applications. The value node of an entry represents the result of applying a function to its arguments. The so extended term graphs are called table graphs. All graphs in the example below are table graphs.

Tabulation and Lookup

Based on this representation of memory, rewriting with value tabulation is just a simple modification of rewriting, and lookup is just a variation of collapsing:

- The tabulation rules \( \mathcal{T} \) are obtained from the rewrite rules \( \mathcal{R} \) by extending their right hand sides with an entry corresponding to their left hand side.

- Lookup rules \( \mathcal{L} \) collap an application and a corresponding table entry with the same arguments.

Example. The function \( \text{FIB} \) has the following recursive tabulation rule, and lookup rule (an arrow points from a table entry to its value node):

\[
\begin{align*}
&\text{FIB} \rightarrow^* \text{FIB} \\
&\text{FIB} \rightarrow^* \text{FIB}
\end{align*}
\]

Full Memoization for Convergent Systems

We combine tabulation, collapsing, and lookup to the memo evaluation relation \( \Rightarrow_{\mathcal{R} \mathcal{C}} \), and restrict it to memo jungles, i.e. the language of table graphs generated by \( \Rightarrow_{\mathcal{R} \mathcal{C}} \) from closed pointed jungles.

Assume that term rewriting via \( \mathcal{R} \) is convergent, i.e. strongly normalizing and confluent. Then memo evaluation is convergent up to table entries. Every normalizing memo evaluation \( T \Rightarrow_{\mathcal{R} \mathcal{C}}^* U \) yields a memo jungle \( U \) whose result jungle unravels to a term in normalform.

If collapsing and lookup get priority over tabulation, memo evaluation performs full memoization in the sense of [Hug85]: A function application is never re-evaluated, if an application of the same function has already been evaluated with arguments that represent equal terms.

This scheme does not impose a strategy on tabulation. So this result can be applied to innermost as well as to outermost rewriting (sequential as well as parallel).
Lazy Memoïzation for Orthogonal Infinitary Systems

We combine tabulation and lookup to the memo rewriting relation $\Rightarrow_{\mathcal{T}}$, and restrict it to memo graphs, i.e. the language of table graphs derived by $\Rightarrow_{\mathcal{T}}^*$ from pointed term graphs.

Assume that the rewrite system $\mathcal{R}$ is orthogonal. Then memo evaluation is confluent up to table entries, and lookup $T \Rightarrow_{\mathcal{C}} U$ implements iterated parallel term rewriting $\tilde{T} \Rightarrow_{\mathcal{R}} \tilde{U}$.

If rewriting is looping, (i.e. if some term graph $G$ rewrites, in one step or more, to a term graph $H$ that contains $G$ as a subgraph), then a lookup $T \Rightarrow_{\mathcal{C}} U$ may introduce a cycle in the term graph underlying $U$. Then memo rewriting $T \Rightarrow_{\mathcal{T}} U$ implements transfinite parallel term rewriting $\tilde{T} \Rightarrow_{\mathcal{R}} \tilde{U}$.

If lookup gets priority over tabulation, memo rewriting performs lazy memoïzation: A function application is never re-evaluated, if an applications of the same function has already been evaluated with the same argument nodes.

This scheme does also not impose a strategy for applying tabulation rules.

Conclusions

We have defined memoïzation by a simple modification of rewriting and collapsing rules. This works for the general classes of convergent, and (infinitary) orthogonal term rewrite systems.

In practice, not all functions of a rewrite system need be tabulated, not even for all rules of a particular function. For example, the cost of the Fib function (without the cost of the additions) cuts down from exponential to linear if just its recursive rewrite rule is tabulated!

As with many optimization techniques, the practical relevance of memoïzation for the implementation of functional languages is a matter of debate. However, lazy memoïzation seems to be a good compromise for speeding up implementations of functional languages. Even an implementation of full memoïzation by St. Kahrs has given surprisingly good results, see [Kah92].

References


