Needed Computations
Shortcutting Needed Steps

Sergio Antoy
Portland State University
Joint work with Jacob Johannsen and Steven Libby

Termgraph 2014 – Vienna, Austria
Thanks NSF CCF-1317249
Outline

- Part 1: an *optimal* implementation.
- Part 2: a *better* one.

What is the meaning of “optimal”? 
A bit of theory

The fundamental result of computations in Orthogonal Term Rewriting Systems:

Every reducible term has a redex that is reduced in every computation to its normal form, if it exists.

Significant consequences and some subtleties ...
Just do it

- If you discover that a redex is needed, then reduce it.
- Can’t do any better.
- Basis for optimal computations (including non-determinism and narrowing).
However

Needed redexes:

- can’t always be known,
- move around,
- may be cloned.
Our Environment

- *Graph* not term rewriting.
- Inductively sequential systems.
- Needed redexes easily found.
- Redex has identity, no clones.
Inductive sequentiality

Operations are defined by constructor-based left-linear rules.

A definition, e.g., the usual list concatenation:

\[
[] + y \rightarrow y
\]

\[
(x : xs) + y \rightarrow x : (xs + y)
\]

is like a structural induction case distinction.
Definitional Trees

A hierarchical structure of the rules of an operation that captures the structural induction-like case distinction of a definition:
Compilation

• Data: graphs labeled by signature symbols.

• Two functions: $H$ (head) and $N$(norm).

• $H$ derives argument to head constructor form. Generated by a traversal of definitional trees.

• $N$ derives argument to constructor form. Obtained from signature.
Object Code (\(H\))

Head function for ++.
Pattern matching is a notational convenience.
Apply rules in textual order.

\[
\begin{align*}
    H([++][]) & = [] \\
    H([++](y:ys)) & = (y:ys) \\
    H([++]y) & = H(y) \\
    H((x:xs)++]y) & = x:(xs++y) \\
    H(x++]y) & = H(H(x)++]y)
\end{align*}
\]
Object Code ($\mathcal{N}$)

Norm function for List constructors and $++$. Pattern matching is a notational convenience.

\[
\begin{align*}
\mathcal{N}([],) &= [] \\
\mathcal{N}(x:xs) &= \mathcal{N}(x):\mathcal{N}(xs) \\
\mathcal{N}(x++y) &= \mathcal{N}(\mathcal{H}(x++y))
\end{align*}
\]
Properties

- Call-by-value (innermost).
- "As if" only needed redexes are reduced.
- Normalizing computations (for values).
- A "very good" implementation/strategy.
  - optimal
  - no lookahead
  - simple
A more Efficient Implementation

- Transform object code
- Shortcut some rewrite steps
- Fewer allocated nodes
- Fewer (pattern) matched nodes
- “Fewer” may be zero
Transformation

Two phases:

- Remove application of $H$ to a variable (specialize the variable).

- Introduce composition of $H$ and symbol (from the above).
Phase 1 Example

An earlier rule of $H$ for $++$:

$$H([\cdot]++y) = H(y)$$

specialize $y$:

$$H([\cdot]++(u++v)) = H(u++v)$$

do same for every other (list) operation.
Phase 2 Example

Rule obtained from phase 1:

\[ H([]++(u++v)) = H(u++v) \]

Introduce \( H++ \) as \( H \circ ++ \):

\[ H++([],u++v) = H++(u,v) \]

do same everywhere in object code. \( H \) is no longer invoked.
Benchmark Program

Define:

\[
\begin{align*}
\text{length } [] &= 0 \\
\text{length } (_{:}xs) &= 1 + \text{length } xs
\end{align*}
\]

Object code:

\[
\begin{align*}
\text{H}_{\text{length}}([]) &= 0 \\
\text{H}_{\text{length}}(_{:}xs) &= \text{H}_+(1, \text{length}(xs)) \\
&\ldots
\end{align*}
\]

Integers are built-in, addition defined accordingly.
$l_1$ and $l_2$ are long lists.

<table>
<thead>
<tr>
<th>length$(l_1++l_2)$</th>
<th>$C_R$</th>
<th>$T_R$</th>
<th>$O_R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>rewrite steps</td>
<td>10</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>shortcut steps</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>node allocations</td>
<td>20</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>node matches</td>
<td>40</td>
<td>26</td>
<td>18</td>
</tr>
</tbody>
</table>

Entries are normalized with respect to the number of rewrite steps of $C_R$. 
Pushing the Idea Further

$O_R$ replaces in phase 1 rule:

$H(\text{length}(_:\text{xs})) = H(1+\text{length}(\text{xs}))$

with:

$H(\text{length}(_:\text{xs})) = H(1+H(\text{length}(\text{xs})))$

and then applies phase 2 of the transformation:

$H_{\text{length}}(_:\text{xs}) = H_+(1,H_{\text{length}}(\text{xs}))$

Can still save the allocations of 1.
Conclusion

- Two implementations of rewriting.
- One remarkably simple and optimal.
- One shortcuts needed steps (rewriting?).
- Executes with same building blocks.
- Conceptually interesting (optimal strategies).
- Practically useful (fewer resources).
Thank you