Efficient rewriting in mCRL2

Muck van Weerdenburg

OASE, February 2, 2006

Outline

What is mCRL2?

mCRL2 ...

- ... is a process algebra with data
- ... is the successor of μCRL
- ... has its own toolset
- ... has a higher-order data specification language
- ...

Why efficient rewriting?

One of the main uses of the mCRL2 toolset is for model checking.

This means state space generation.

Rewriting of data expressions consumes more than 90% of the time of state space generation.

Why not reuse existing solutions?

No existing language/implementation really matches our needs.

Really higher-order.

Open terms.

Why not reuse existing solutions? (Open terms.)

A (linearised) specification could be as follows:

$$
\begin{array}{rcl}
P(b : \mathbb{B}, n : \mathbb{N}) & = & \sum_{m:\mathbb{N}} \neg b \wedge m < 5 \rightarrow r(m).P(\neg b, m) \\
 & + & b \rightarrow s(n).P(\neg b, n)
\end{array}
$$

For any natural number m, action $r(m)$ is possible from state $P(b, n)$ if $\neg b \wedge m < 5$ holds (and results in state $P(\neg b, m)$).

Only by rewriting on open terms we can detect that we every number m above or equal to 5 will never satisfy the first guard.

mCRL2 data language

We use subset of the actual data language, as specifications are first translated to this subset.

Sorts are defined or composed with $\rightarrow$:

$$\mathbb{N}, \quad S, \quad (S \rightarrow \mathbb{N}) \rightarrow S \rightarrow \mathbb{N}$$

Data expressions can be the following:

- a variable: $x, \ \ y, \ \ \ldots$
- a function (symbol): $\mathit{fib}, \ \ 0, \ \ \ldots$
- an application of two data expressions: $\mathit{fib}(0), \ \ x(y)(\mathit{fib}), \ \ \ldots$

mCRL2 data language

Equations define functions (sort of).

$times(x)(0) = 0, \ \ plus(0) = times(1)$

Equations can be conditional.

$times(x)(y) = 0 \ \text{ if } x = 0 \vee y = 0$

Equations are used as rewrite rules (from left to right).

Outline

Rewriting

Implicit substitutions are used to avoid unnecessary work.

$rewr(x > 5[0/x])$ vs. $rewr(x > 5, x \to 0)$

$t[u/x]$, $rewr(t)$ and $rewr(t, s)$ are all in the "order" of $|t|$

Code generation for faster rewriting.

Each function symbol gets it own dedicated rewrite function.

Up to 20 times faster.

Strategies

Innermost

First rewrite arguments, then try to match term with match tree.
$\mathit{if}(0 < 1, 0, \mathit{fib}(50)) \to \mathit{if}(\mathit{true}, 0, 12586269025) \to 0$

JITty (with automatic strategy generation)

Rewrite arguments only when they are needed for a rule.
$\mathit{if}(0 < 1, 0, \mathit{fib}(50)) \to \mathit{if}(\mathit{true}, 0, \mathit{fib}(50)) \to 0$

JITty uses *strategies*: $[\{1\}, \{\alpha, \beta\}, \{2, 3\}]$
(α: $\mathit{if}(\mathit{true}, x, y) \to x$, β: $\mathit{if}(\mathit{false}, x, y) \to y$)

Strategies

JITty can avoid rewriting parts of a term that are not relevant.

Innermost can easily keep track of normal forms.

Example:

> *Rewrite $f(t, u)$:*
> *Rewrite arguments t and u to $f(t', u')$*
> *Apply rule $f(x, y) \to g(h(x), y)$*
> *Rewrite $h(t')$ to h'*
> *Rewrite $g(h', u')$*

Last two steps no longer include the rewriting of arguments.

Pattern matching

Higher-order unification is undecidable.

Pattern matching is done purely on syntax.
(This makes rewriting essentially first-order.)

What do we lose? (η-equivalence?)

In practice, first-order is often sufficient.

In other cases we need help (e.g. from provers).

Pattern matching

Naive pattern matching is inefficient.

$$\begin{array}{rcl} C(e_0, x_0, \ldots, x_n) & = & x_0 \\ & \vdots & \\ C(e_n, x_0, \ldots, x_n) & = & x_n \end{array}$$

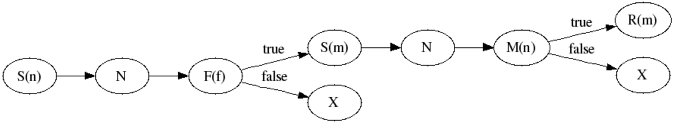
The avoid this we use match trees, extended for conditional rules *and* higher-order terms.

Match trees also translate nicely to code.

Pattern matching

Each rules is transformed to simple tree structure.

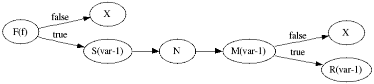
$h(n)(f(m)(n)) \rightarrow m$ would be:

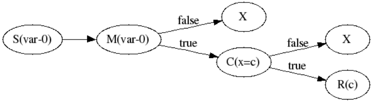


Also a conditional node is possible. E.g. $C(n < 5)$.

Pattern matching

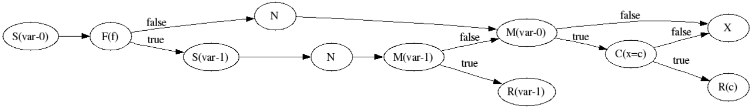
All trees of rules for a specific function are combined into one tree.





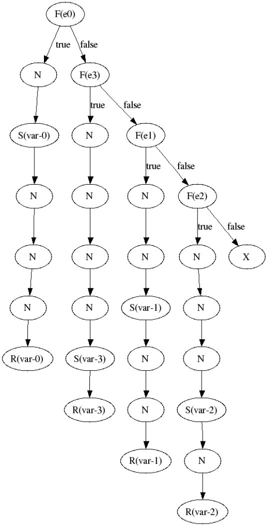
$h(f(x), x) \rightarrow x$ $\qquad$ $h(x, x) \rightarrow c \quad \text{if} \quad x = c$

$$\downarrow$$



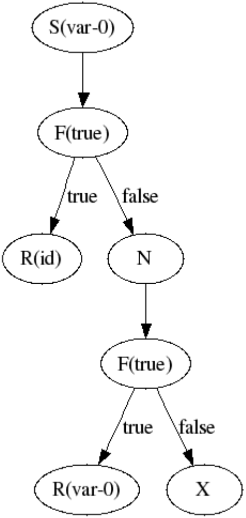
Pattern matching

$$
\begin{aligned}
C(e_0, x_0, \ldots, x_n) &= x_0 \\
&\vdots \\
C(e_n, x_0, \ldots, x_n) &= x_n
\end{aligned}
\longrightarrow
$$



Pattern matching

$$\begin{aligned} and(true) &= id \\ and(b, true) &= b \end{aligned}$$



Outline

Benchmarks (preliminary)

Single (closed) term rewriting:

	mCRL2		μCRL	GHC	Clean
	Innermost	*JITty*			
fib(30)	1.0s	7.9s		1.6s	0.8s - 2.3s
fib(33)	4.1s	33.7s		7.3s	3.6s - 8.7s
mfib	7.7s	42.1s	5.7s	14.7s	3.8s - 4.2s

State space generation:

	mCRL2		μCRL
	Innermost	*JITty*	
chatboxt	3.2s	3.0s	4.1s/5.8s
1394-fin	173.2s	78.9s	101.2s

Benchmarks

Preliminary benchmarks show that we ...

- ... are faster than μCRL
- ... can compete with the fastest implementations of functional languages (Glasgow Haskell Compiler, Clean).

Note that rewriting with only closed terms (and without higher-order function application) should be (much) more efficient.

Getting decent benchmarks is hard. Implementations and uses diverge.

Future work

- Extending JITty implementation to avoid rewriting of normal forms.
- Optimisation of terms for rewriter?
  $(b_0 \wedge b_1) \wedge (b_2 \wedge b_3)$ vs $b_0 \wedge (b_1 \wedge (b_2 \wedge b_3))$
- Other strategies than innermost or JITty?
  $len([]) = 0$, $len(a \triangleright s) = 1 + len(s)$ without rewriting s every time.
- ...

Thank you for your attention.