Efficient rewriting in mCRL2

An account of implementing applicative term rewriting

Muck van Weerdenburg

WRS'06, August 11, 2006

Outline

- Introduction
  - What is mCRL2?
  - Why efficient rewriting?
  - Why not reuse existing solutions?
- Rewriting
  - mCRL2 data language
  - Implementation
    - Code generation
    - Normal forms
    - Pattern matching
- Conclusions

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.

Higher-order, so toolsets like μCRL won't do.

Rewriting on open terms, so fast functional language implementation won't do either.

Why not reuse existing solutions? - Open terms

A (linearised) mCRL2 specification could be as follows:

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

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.

Why not reuse existing solutions? - Open terms

$$
\begin{array}{lcl}
n < S(S(0)) & & \\
\quad\downarrow & & \\
0 < S(S(0)) & \rightarrow & \mathit{true} \\
S(n') < S(S(0)) & \rightarrow & n' < S(0) \\
 & & \quad\downarrow \\
 & & 0 < S(0) \quad \rightarrow \quad \mathit{true} \\
 & & S(n'') < S(0) \quad \rightarrow \quad \mathit{false}
\end{array}
$$

Outline

mCRL2 data language

We use the core data language, as specifications are first translated to this core.

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): 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).

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

Equations can be conditional.

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

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

Implementation - Strategies

- *Innermost*

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

- *JITty* (with automatic strategy generation)

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

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

Implementation - Common optimisations

- Code generation.
  Each function symbol gets it own dedicated rewrite function.
- Implicit substitutions.
  $rewr((x > 5)[0/x])$ vs. $rewr(x > 5, x \mapsto 0)$
- Avoiding rewriting normal forms
- Pattern matching using trees
  (Only on linear patterns)

Implementation - New optimisations

- Code generation (JITty)
- Avoiding rewriting normal (JITty)
  Without additional run time cost
- Pattern matching using trees
  On nonlinear (higher-order) patterns

Code generation - Innermost

function $innermost(\ f(t_1, \ldots, t_n)\ )$
  for $i \in \{1, \ldots, n\}$ **do**
    $t_i\ := innermost(\ t_i\ )$
  return $(get\_rewr\_func(f, n))(t_1, \ldots, t_n)$

$$get\_rewr\_func = \{\quad (g,0) \mapsto rewrite_g^0,\ (g,1) \mapsto rewrite_g^1,$$
$$(h,0) \mapsto rewrite_h^0,$$
$$\ldots\}$$

Implementation of $rewrite_f^n$ is based on match tree.

Code generation - JITty

function $JITty(\ f(t_1, \ldots, t_n)\ )$
  for $i \in \{1, \ldots, n\}$ **do**
    $t_i := JITty(\ t_i\ )$
  return $(get\_rewr\_func(f, n))(t_1, \ldots, t_n)$

$$get\_rewr\_func = \{ \quad (g,0) \mapsto rewrite_g^0,\ (g,1) \mapsto rewrite_g^1,$$
$$(h,0) \mapsto rewrite_h^0,$$
$$\ldots\}$$

Implementation of $rewrite_f^n$ is based on strategy.

Normal forms - Innermost

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.

Normal forms - JITty

JITty requires annotations to detect normal forms.

Example:

Rewrite $f(t, u)$*:*

Rewrite argument t *to get* $f(t', u')$

Apply rule $f(x, y) \rightarrow g(h(x), y)$

Rewrite $g(h(\nu(t')), u)$

Or rewrite $g^{00}(h^{1}(t'), u)$

We use the last method, which does not add run time cost.

Pattern matching

Higher-order matching at least NP-complete.

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.

$$
\begin{array}{llllll}
s_0 == s_0 & \rightarrow & \mathit{true} & \ldots & s_n == s_0 & \rightarrow \quad \mathit{false} \\
s_0 == s_1 & \rightarrow & \mathit{false} & \ldots & s_n == s_1 & \rightarrow \quad \mathit{false} \\
\quad \vdots & & & & \quad \vdots & \\
s_0 == s_n & \rightarrow & \mathit{false} & \ldots & s_n == s_n & \rightarrow \quad \mathit{true}
\end{array}
$$

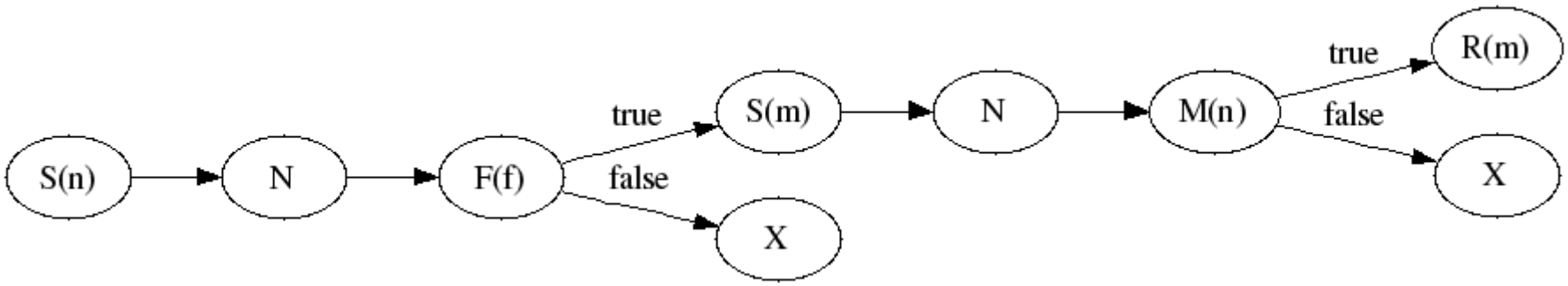
We use match trees for efficient pattern matching.

Match trees also translate nicely to code.

Pattern matching

Each rule is transformed to a simple tree.

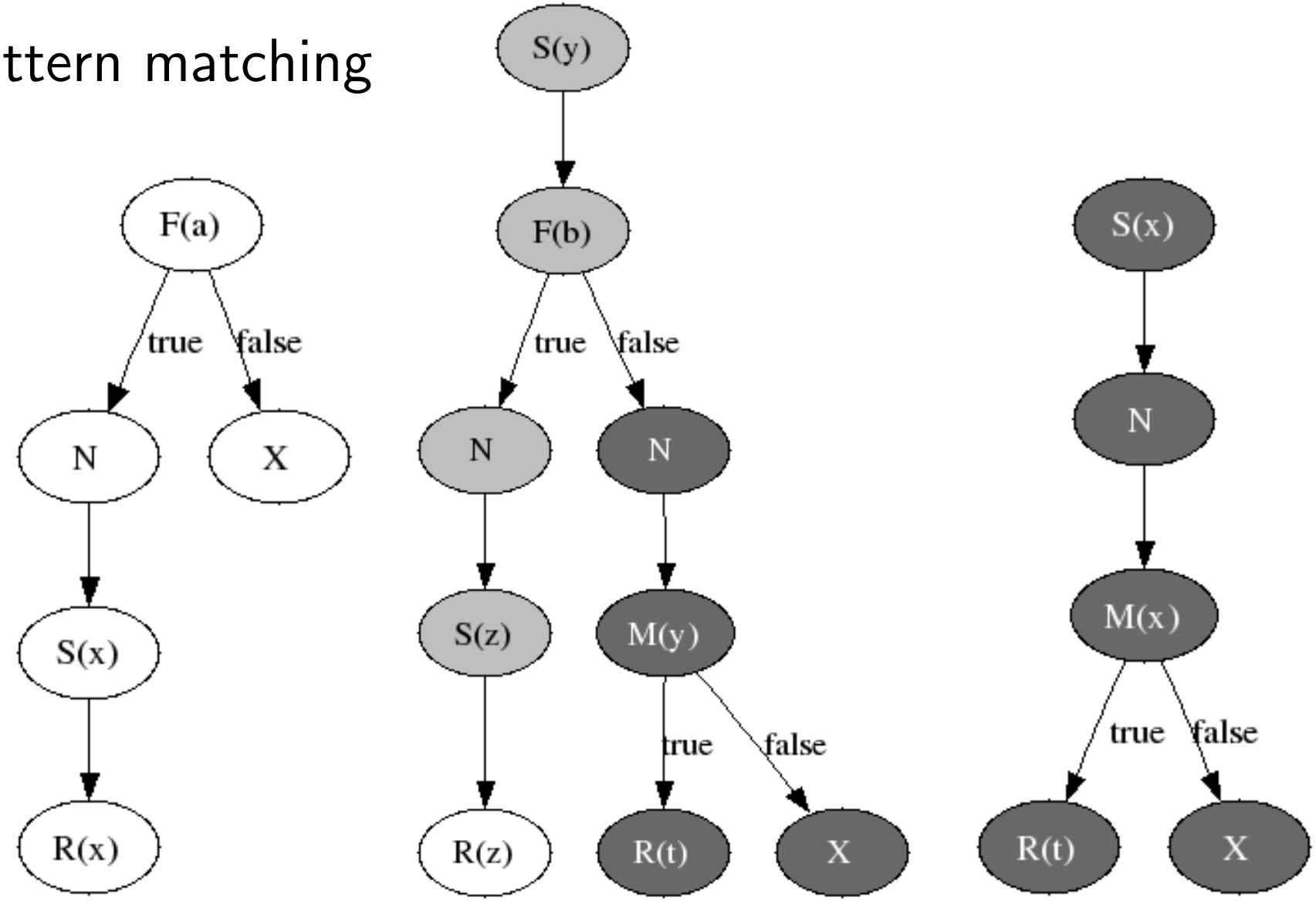
$h(n)(f(m)(n)) \rightarrow m$ would be:



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

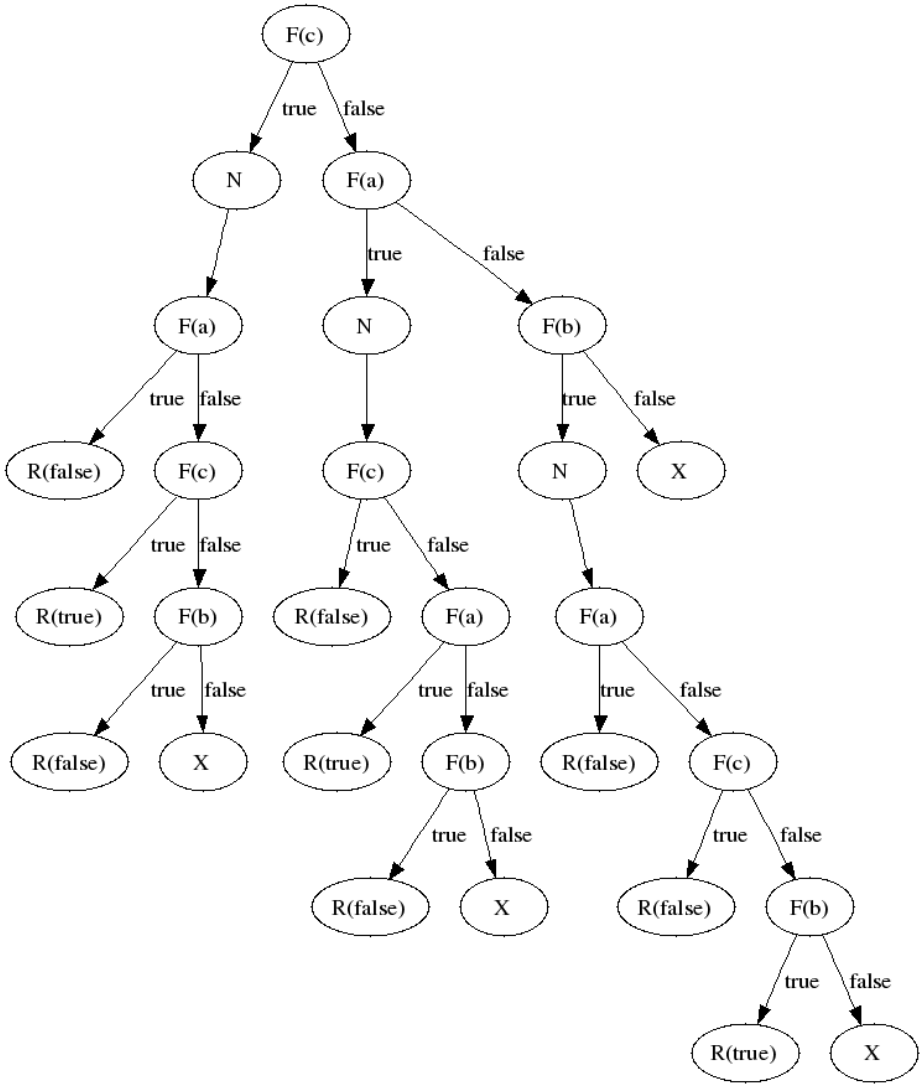
All trees of rules for a specific function are combined into one tree.

Pattern matching



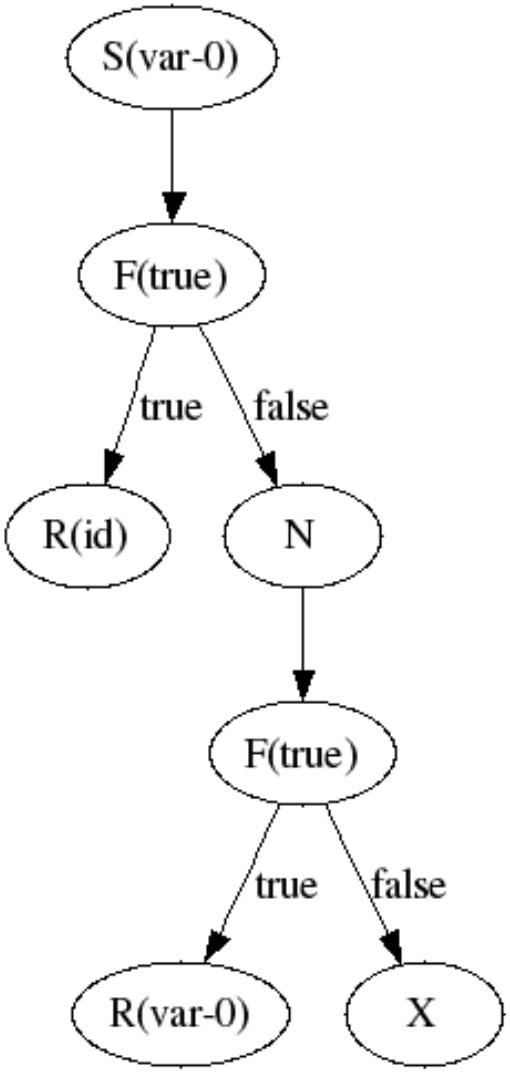
Pattern matching

Equality on $\{a, b, c\}$



Pattern matching

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



Outline

Conclusions

Benchmarks show that ...

- ... our JITty rewriter is faster than μCRL
  (for state space generation)
- ... open term rewriting is as fast as the fastest rewriters
  (GHC, Clean)
- ... our JITty rewrite is inefficient in closed term rewriting
  (due to construction of terms)

What's left? Avoiding high term construction cost for JITty.

(And hope function language implementations will support open term rewriting.)

Thank you for your attention.