An Account of Implementing Applicative Term Rewriting

Muck van Weerdenburg
M.J.van.Weerdenburg@tue.nl

Abstract

Generation of labelled transition systems from system specifications is highly dependent on efficient rewriting (or related techniques). We give an account of the implementation of two rewriters of the mCRL2 toolset. These rewriters work on open terms and use nonlinear match trees. A comparison is made with other commonly used efficient rewriters.

1 Introduction

Verification of systems is an important field of research that is directly linked to the practical world. A widely used technique in this field is model checking. In short, this often means that a Labelled Transition System (LTS) is generated from a system specification and requirements are checked on this LTS. However, the task of generating LTSs is very time and space demanding. In cases where LTS generation is done with the help of rewriters, generation of typical LTSs of, say, 10^7 transitions requires at least a few times more than 10^7 calls to the rewriter. In fact, inspection of this process in the mCRL2 toolset [9], which supports modelling and verification of systems, shows that more than 90% of the time generating an LTS is spent rewriting.

Apart from on-the-fly LTS reductions (e.g. modulo some equivalence), there are two clear paths towards optimisation of LTS generation. One is to reduce the number of times the LTS generator uses the rewriter. The other, which we consider here, is to optimise the rewriting procedure.

We discuss two implementations we have made for the mCRL2 toolset. One uses innermost rewriting, the other just-in-time (JITty) rewriting [17]. The latter is a strategy close to lazy rewriting [7] (i.e. rewriting (sub)terms only when needed). An essential property of these rewriters is that they are *compiling* rewriters, meaning that a specialised rewriter is generated for a given specification. Also, they support rewriting of *open* terms (i.e. terms in which (free) variables may occur), which is required for LTS generation.

As mCRL2 has a higher-order data language, rewriting is on higher-order (applicative) terms. Due to the fact that higher-order matching is NP-hard [3], we restrict the rewriting to using only simple syntactic pattern matching. This

basically boils down to rewriting applicative terms without being able to do η-reductions. It seems that this restriction does not really impedes practical use of the rewriter (at least not with case studies, such as [12], so far), but the precise implications should be subject to future research. But even if our choice is too restricted for general-purpose use, a limited, but *fast* rewriter is still very useful for a large set of problems.

In order to implement efficient matching we use an adaptation of existing algorithms that, instead of matching each rule separately, combine sets of rules into a tree structure that allows for simultaneous matching of these rules. Although implementations of such algorithms often require left-hand sides to contain each variable at most once (i.e. the left-hand sides must be *linear*), our implementation does not have this restriction.

Another important optimisation is to avoid rewriting normal forms multiple times. Although this is fairly easy with innermost rewriting, it is much more involved in the JITty rewriter.

In short, we have implemented a compiling JITty rewriter for conditional rewrite rules on open applicative terms, making use of efficient matching of non-linear applicative terms. As far as we know, this is the first of its kind.

First, we introduce the part of the mCRL2 data language that is relevant for rewriting and the general architecture of our implementations in Sect. 2. In Sect. 3 we discuss the matching algorithm used and Sect. 4 and Sect. 5 contain the descriptions of the innermost and JITty rewriters, respectively. We conclude with an analysis of some benchmarks in Sect. 6.

2 Preliminaries

The data language we consider here is the core data language of mCRL2. It has only one operator, viz. application. The complete data language contains many additional constructs for ease of modelling (including λs), but they are all expressible in this core. From this point on, we will refer to this core simply as mCRL2.

The **signature** (Σ) of mCRL2 consists of a set of **basic sorts** $\mathbb{S}_B$, a set of **variables** $\mathbb{V}$ and a set of **function symbols** $\mathbb{F}$. Each variable or function symbol has a **sort**. Sorts s are defined as follows, where $b \in \mathbb{S}_B$ and $\to$ is right-associative:

$$s \;:=\; b \;\mid\; s \to s$$

With $x_s \in \mathbb{V}$ a variable of sort s and $f_s \in \mathbb{F}$ a function symbol of sort s, the definition of mCRL2 **terms** t_s of sort s is as follows:

$$t_s \;:=\; x_s \;\mid\; f_s \;\mid\; t_{s' \to s}(t_{s'})$$

Typical basic sorts are the booleans $\mathbb{B}$ or the integers $\mathbb{Z}$. Function symbols are, for example, *true* or *even*. The sorts as subscripts of terms are usually omitted. Given a term $f(t_1)\ldots(t_n)$ we call f the **head symbol** and t_i the ith

argument. The **arity** of a function symbol is the maximal number of arguments it can have. For readability we usually write terms with sequences of **applications** (i.e. terms $t(u)$) such as $((f(w))((g(x))(y)))(z)$ simply as $f(w, g(x, y), z)$.

Rewrite rules are of the form $t \to u$ **if** c, where terms t and u have the same sort. Term c of sort $\mathbb{B}$ is the condition of a rewrite rule indicating whether or not the rule may be applied (i.e. only when c rewrites to *true*). Often we omit this condition in the case c is (syntactically) equal to *true*.

We write $(\Sigma, \to)$ for a signature Σ and set of rewrite rules $\to$ to denote a Term Rewrite System (**TRS**) [6].

The architecture of the rewriters is as follows. The rewriters first preprocess the TRS by sorting the rules by head symbol. For each head symbol f and number of arguments n that f can have, we create a specialised function $rewr_f(t_1, \ldots, t_n)$ that returns a normal form of the term $f(t_1) \ldots (t_n)$. The implementation of a function $rewr_f$ takes care of the matching and applications of the rules for f by using the match trees of Sect. 3. Also a main rewrite function is added that takes a single term t and calls the specialised function for the head symbol of t. Depending on the strategy it also rewrites the arguments of a function symbol, before calling its specialised function.

For reasons of efficiency we use **implicit substitutions**. This means that, instead of first substituting specific values for variables and then rewriting the term, we apply substitution on-the-fly during rewriting (i.e. we rewrite in a *context* of substitutions). This basically boils down to replacing a variable with its value as soon as it is encountered. We can, however, also encounter terms of the form $x(t_1, \ldots, t_n)$. In the case that x is not bound to a value we can just ignore it and rewrite its arguments. Otherwise, we need to get the value of x, append the arguments $t_1, \ldots, t_n$ and then rewrite that term.

For the implementation of the data terms we use the ATerm [21] library. This automatically gives us term sharing[1] and constant time equality tests. Construction of terms, however, is more expensive.

3 Match trees

Straightforward implementations of rewrite systems will try to match the term to be rewritten with every left-hand side of a rewrite rule separately. For example, with the system $\{t_1 \to u_1, t_2 \to u_2\}$ one could first try to match a term with t_1 and afterwards, if it did not match, with t_2 (or vice versa).

That this is not a very efficient manner of matching can be seen clearly by looking at rules for equality functions. Assuming a sort S with n simple constructors (i.e. without arguments), the equality on S needs n^2 rules (for every pair in $S \times S$).[2] However, by combining these rules into a specific tree

[1] That is, equal (sub)terms are only stored once in memory. Note that changing a term in one place will not automatically change (equal) terms in other places.

[2] Note that many languages allow for more compact notations by assuming an order on rules. Such features are in general not safe when rewriting with open terms (e.g. rewriting

structure, we can test for a match in the order of n. Such trees are in essence decision trees for a matching algorithm.

The method we use for this is similar to the ones used in the ASF+SDF [19] rewriters [20] and ELAN [22]. For rules with linear left-hand sides (i.e. left-hand sides in which variables occur at most once), algorithms to create such trees can be found in [14, 2, 16]. As we have applicative terms and allow nonlinear rewrite rules, our approach deviates a bit. Note that in ASF+SDF nonlinear rules are also allowed, but converted to linear rules, which requires additional side conditions. Another related method is that of *definitional trees* [1]. These trees for linear rules are specialised for a combination of lazy rewriting and narrowing [6].

Match trees determine the way a term is matched; each node of a tree represents a basic instruction and guides the path through the tree. We start at the root and walk up the tree, choosing branches based on the result of matching so far. For example, one node could be to check whether a (sub)term has a specific head symbol. Matching continues with one branch if the symbol was found and with the other branch otherwise.

The way a term is traversed during matching is as follows. Matching a term $f(t_1, \ldots, t_k)$ according to a match tree m is done in a left-most way; it starts with argument t_1 of f and executes the specific functionality of m. We do not have to match f itself as we make a specialised rewrite function that handles only terms starting with f (for each symbol f). At any point during execution of the matching algorithm there is a context of values bound to variables (i.e. a context of substitutions) and a stack of terms to be matched. Initially the context is empty and the stack consists of the arguments t_1 to t_k of f (with t_1 on top). The matching algorithm always considers the top of the stack, which we refer to as $g(u_1, \ldots, u_l)$. During matching the context will be built up, resulting in a substitution that makes the left-hand side of the matching rule equal to $f(t_1, \ldots, t_k)$.

Our **match trees** m have the following structure, with $x \in \mathbb{V}$, $f \in \mathbb{F}$ and term t.

$$m ::= S(x, m) \mid M(x, m, m) \mid F(f, m, m) \mid N(m) \mid C(t, m, m) \mid R(t) \mid X$$

We give an intuition of functionality of the trees before giving the actual matching function. A $S(x, m)$ binds the top of the stack to variable x and continues with tree m. Such a value bound to x is tested for equality with the top of the stack with $M(x, m, n)$, which continues with tree m on equality and n otherwise. With $F(f, m, n)$ matching continues with $u_1, \ldots, u_l$ on top of the stack and tree m if f is equal to g. If not, tree n is used without changing the stack. Node $N(m)$ removes the top of the stack and continues with m. A condition b can be checked with $C(b, m, n)$. A successful match is indicated by $R(t)$, where t is the result of applying a matching rule. Unsuccessful matches occur with X and when the stack is empty (i.e. there are too few arguments).

$f(x)$ in the system $f(0) \rightarrow e$; $f(n) \rightarrow g(f(n-1))$ does not terminate). In mCRL2 we use standard conditional rewriting.

Let σ be a context, $\sigma[x \mapsto t]$ the context σ in which term t is bound to variable x and $\sigma(t)$ a term t in which every variable is replaced by the value bound to it in σ. Also let [] denote the empty stack and $t \rhd s$ term t on top of stack s. The definition of the **matching function** μ, which returns either X (no match) or $R(t)$ (match with result t), is as follows:

$$
\begin{array}{llcll}
\mu(m, & \sigma, & [\,]) & = & X \\
\mu(S(x,m), & \sigma, & t \rhd s) & = & \mu(m, \sigma[x \mapsto t], t \rhd s) \\
\mu(M(x,m,n), & \sigma, & t \rhd s) & = & \mu(m, \sigma, t \rhd s) & \text{if } \sigma(x) = t \\
\mu(M(x,m,n), & \sigma, & t \rhd s) & = & \mu(n, \sigma, t \rhd s) & \text{if } \sigma(x) \neq t \\
\mu(F(f,m,n), & \sigma, & g(u_1, \ldots, u_n) \rhd s) & = & \mu(m, \sigma, u_1 \rhd \ldots \rhd u_n \rhd s) & \text{if } f = g \\
\mu(F(f,m,n), & \sigma, & g(u_1, \ldots, u_n) \rhd s) & = & \mu(n, \sigma, g(u_1, \ldots, u_n) \rhd s) & \text{if } f \neq g \\
\mu(N(m), & \sigma, & t \rhd s) & = & \mu(m, \sigma, s) \\
\mu(C(b,m,n), & \sigma, & t \rhd s) & = & \mu(m, \sigma, t \rhd s) & \text{if } \sigma(b) \\
\mu(C(b,m,n), & \sigma, & t \rhd s) & = & \mu(n, \sigma, t \rhd s) & \text{if } \neg\sigma(b) \\
\mu(R(t), & \sigma, & t \rhd s) & = & R(\sigma(t))
\end{array}
$$

To illustrate the use of the match trees and give some intuition on how we build such trees, we consider the rewrite rules $f(g(x), x) \to x$ and $f(x, x) \to c$ if $h(x)$. In Fig. 1 the match tree for the first rule is shown. We can see that the root node (on the far left) checks whether the head symbol of the first argument is a g or not. If this is the case, it binds the argument of g to x and proceeds to the next argument. As g has only one argument, this means we look at the next argument of the enclosing function f. The M node checks to see if this argument is the same as the value of x and returns the result (also x) if this is the case. Note that the head symbol f does not occur as root in the tree. This is because we make one tree for all rules with head symbol f, thus removing the need to check for f itself in the tree.

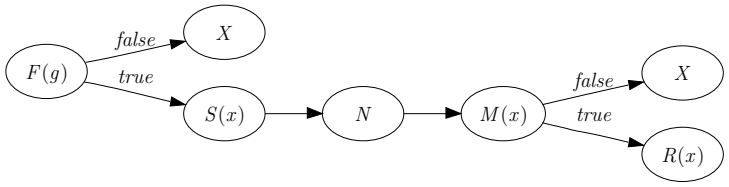

Figure 1: Match tree for $f(g(x), x) \to x$

The tree for the conditional rule is shown in Fig. 2. Here we see that the first argument is stored and the second argument is matched with the first argument. If they are the same, the condition $h(x)$ is checked, using the value bound to x, before returning the result c.

Finally, we combine both trees to the complete match tree for function symbol f, as shown in Fig. 3. Such a combination is made by weaving the trees together and synchronising on N nodes. The following rules give a simplified

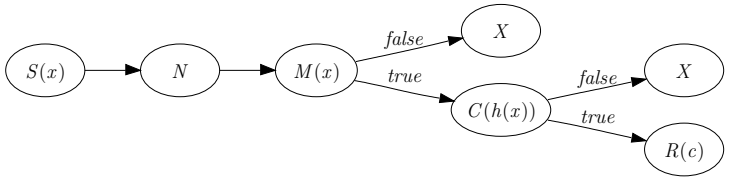

Figure 2: Match tree for $f(x, x) \to c$ if $h(x)$

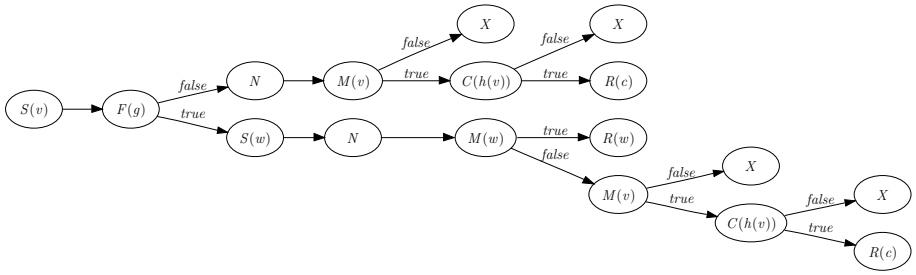

Figure 3: Combined match tree for f

version of our algorithm to compute $comb(T)$, the combination of the trees in T. If more than one rule can be applied, the one that occurs first in the list below is applied. In the case that one rule can be applied in different ways, one is chosen non-deterministically. We write T for a set of trees, which we can partition in T_f and $T \setminus T_f$, of which the former contains all F nodes that check for symbol f (and only those nodes). Projection functions π_1 and π_2 are used to filter a set of $F(f, m, n)$ nodes to the m, respectively n values. We write $N_f(T)$ for T with an N node added to the root of every tree in it; the amount of added nodes corresponds to the number of arguments f has (in the pattern). The substitution of a variable x by y in tree m is denoted by $m[y/x]$. With x' we indicate a fresh variable (i.e. one not occurring in any of the trees).

$$
\begin{array}{lcl}
comb(\{R(t)\} \cup T) & \to & R(t) \\
comb(\{C(t, m, n)\} \cup T) & \to & C(t, comb(\{m\} \cup T), comb(\{n\} \cup T)) \\
comb(\{M(x, m, n)\} \cup T) & \to & M(x, comb(\{m\} \cup T), comb(\{n\} \cup T)) \\
comb(\{S(x, m)\} \cup T) & \to & S(x', comb(\{m[x'/x]\} \cup T)) \\
comb(\{F(f, m, n)\} \cup T) & \to & F(f,\ comb(\pi_1(T_f) \cup N_f(T \setminus T_f)), \\
 & & \qquad comb(\pi_2(T_f) \cup (T \setminus T_f))\ ) \\
comb(\{N(m_0), \ldots, N(m_k)\}) & \to & N(comb(\{m_0, \ldots, m_k\})) \\
comb(\{X\} \cup T) & \to & comb(T) \\
comb(\emptyset) & \to & X
\end{array}
$$

The first rule indicates that as soon as there is a tree indicating a positive match, we can just return that match and ignore the other trees. In the rule for

S we introduce a fresh variable to avoid conflicts with variables in other trees. When applying the F rule for a symbol f, we consider all trees that have such a root node. This is done as the first subtree of an F processes arguments of the matched function symbol and this can only be done once (due to the matching function). Also, during matching of the arguments (of the subterm), the other trees that do not participate need to be ignored until f and its arguments are completely matched. For this reason we add the necessary N nodes to these trees.

There are several optimisations to the above. For example, between two N nodes, we can ensure that matching a variable occurs only once and we can combine all S nodes into one, as they all store the same term. In case both subtrees of an M or C node are the same, we can replace it with the subtree itself. Also, S nodes that bind a value to a variable that is never used in the subtree can be replaced by the subtree.

Note that in the first three cases there might be more than one way to choose T. As choosing

4 Compiling Innermost Rewriter

The implementation of the innermost rewriter is very similar to that of the μCRL toolset [4] and ASF+SDF. We discuss the main points. To achieve optimal performance, compilation of a specific rewrite system is essential. This is done as described in Sect. 2. The main rewrite function would be of the following form (not considering implicit substitutions and variables as head symbols):

$$\begin{array}{l}\textbf{function } \mathit{innermost}(f(t_1,\ldots,t_n))\\ \quad \textbf{for } i \in \{1,\ldots,n\} \textbf{ do}\\ \quad\quad t_i := \mathit{innermost}(t_i)\\ \quad \textbf{return } \mathit{rewr}_f(t_1,\ldots,t_n)\end{array}$$

A specialised function for a function symbol f uses the match tree for f to see if any rule can be applied. If this is the case, the right-hand side of that rule is built and the generic rewrite function is called on this term. If no rule matches, then the original term is built and returned. An example of the code that would be generated of a function with rewrite rule $f(c,x) = g(h(x),x)$ is as follows.

$$\begin{array}{l}\textbf{function } \mathit{rewr}_f(\mathit{arg}_1,\mathit{arg}_2)\\ \quad \textbf{if } \mathit{arg}_1 = c \textbf{ then}\\ \quad\quad \textbf{return } \mathit{rewrite}(g(h(\mathit{arg}_2),\mathit{arg}_2))\\ \quad \textbf{else}\\ \quad\quad \textbf{return } f(\mathit{arg}_1,\mathit{arg}_2)\end{array}$$

One important optimisation is that of avoiding needless traversal of *normal forms*. The main observation here is that one can assume that the arguments

of a specific rewrite function are already in normal form. This is the case when called from the main rewrite function, as it first explicitly rewrites these arguments, and also needs to be the case when called from a specific rewrite function.

We achieve this optimisation by taking the instantiations of the variables of the matching rewrite rule, which are in normal form by definition, and building up the term around it with the appropriate specific rewrite functions. For example, if we need to build a term $g(h(x), x)$, we call the specific rewrite function of h on the instantiation of x, returning the normal form of $h(x)$, and then call the specific rewrite function of g with the previous result and the instantiation of x. The rewrite function for f then becomes as follows:

$$
\begin{array}{l}
\textbf{function } \mathit{rewr}_f(\mathit{arg}_1, \mathit{arg}_2) \\
\quad \textbf{if } \mathit{arg}_1 = c \textbf{ then} \\
\quad\quad \mathit{tmp} \ := \ \mathit{rewr}_h(\mathit{arg}_2) \\
\quad\quad \textbf{return } \mathit{rewr}_g(\mathit{tmp}, \mathit{arg}_2) \\
\quad \textbf{else} \\
\quad\quad \textbf{return } f(\mathit{arg}_1, \mathit{arg}_2)
\end{array}
$$

In our case we also have to consider applicative terms. This means that a function of arity n has at most n arguments (instead of exactly n). This is solved by generating specific rewrite functions for each function symbol and number of arguments allowed. So, for f we would have two additional rewrite functions (i.e. one for one argument and another for no arguments at all).

5 Compiling JITty Rewriter

The JITty strategy delays rewriting of arguments for as long as they are not needed for matching. By doing so, it avoids rewriting terms that can be removed without ever being used. A typical example is the *if*, which often has the following rules:

$$
\begin{array}{lllll}
\alpha: & \mathit{if}(\mathit{true}, x, y) & \rightarrow & x \\
\beta: & \mathit{if}(\mathit{false}, x, y) & \rightarrow & y \\
\gamma: & \mathit{if}(b, x, x) & \rightarrow & x
\end{array}
$$

Instead of rewriting all arguments first and then matching these rules, like innermost rewriting does, JITty uses a strategy to, for example, only rewrite the first argument and then check rules α and β. Only if these rules do not match, the other arguments are rewritten and γ is matched. Such a strategy, written as $[\{1\}, \{\alpha, \beta\}, \{2, 3\}, \{\gamma\}]$ for the *if*, can be computed automatically. Note that strategies need to be *full* and *in-time* [17], which means that all rules and argument indices must occur in the strategy and every argument index must occur before the rules that need that argument for matching.

Concerning code generation, this strategy differs from innermost in the fact that the generic rewrite function (*JITty*) no longer rewrites the arguments of a

function f before calling its specific rewrite function $rewr_f$. Instead $rewr_f$ itself does this, as specified by the strategy for function symbol f. Also, where there is only one match tree for all rules (with the same head symbol) in innermost, with JITty we have a match tree per set of rewrite rules in the strategy. In the above example this would mean there is a tree matching both rule α and β and a tree matching γ.

The code for a strategy is generated such that the elements in the strategy are executed in order. For the *if* this would mean that the corresponding specific function will consist of first rewriting the first argument, then the code for the match tree of $\{\alpha, \beta\}$, etc., as can be seen in the following code.

```
function rewr_if(arg_1, arg_2, arg_3)
    arg_1 := JITty(arg_1)
    if arg_1 = true then
        return arg_2
    else if arg_1 = false then
        return arg_3
    else
        arg_2 := JITty(arg_2)
        arg_3 := JITty(arg_3)
        if arg_2 = arg_3 then
            return arg_2
        else
            return if(arg_1, arg_2, arg_3)
```

5.1 Strategy generation

Because we do not want to burden our users with supplying strategies themselves, we need to generate reasonable strategies from a given set of rewrite rules (i.e. one strategy per function symbol). This is done by observing which arguments need to be rewritten to be able to match a given rule. An argument that is needed for matching by *most* of the rules is added to the strategy, indicating it needs to be rewritten first. In the case that all arguments of a rule that are essential for matching are rewritten, this rule is added to the strategy. This process continues until all rules and arguments are in the strategy.

More formally, let $dep(r)$ be a function that returns the indices of the arguments that need to be rewritten before matching rule r, i.e. (with $vars(t)$ the variables occurring in t)

$$dep(f(t_1, \ldots, t_k) \to u) = \{i \ : \ t_i \notin \mathbb{V} \ \vee \ t_i \in \bigcup_{j \neq i} vars(t_j)\}$$

That is, a rule depends on argument i if the ith argument is either a specific term (not just a variable) or it is a variable that also occurs in another argument.

Also, let $occ(i, R_f)$ be a function that returns the number of rules of a set R_f that require argument i:

$$occ(i, R_f) = \#\{r \in R_f \ : \ i \in dep(r)\}$$

We denote the empty strategy with [] and a set S of argument indices or rewrite rules prepended to a strategy l by $S \rhd_c l$. Here, $\rhd_c$ only adds S to l if S is not empty (i.e. $\emptyset \rhd_c l = l$). A strategy for a set of rules R_f is generated with $strat(R_f, \emptyset)$, where $strat(R, I)$ is defined as follows, for any set of rules $R \subseteq R_f$ and set of indices $I \subseteq \{1, \ldots, ar(f)\}$ (with I the set of argument indices added to the strategy so far and $\uparrow$ the maximum quantifier):

$$\begin{array}{lcll} strat(\emptyset, I) & = & (\{1, \ldots, ar(f)\} \setminus I) \rhd_c [] & \\ strat(R, I) & = & T \rhd_c J \rhd_c strat(R \setminus T, I \cup J) & \text{if } R \neq \emptyset \\ & & \textbf{where} \ \ T = \{r \in R \ : \ dep(r) \subseteq I\}, & \\ & & \qquad J = \{i \ : \ i \notin I \ \wedge \ occ(i, R \setminus T) = \uparrow_{j \notin I} occ(j, R \setminus T)\} & \end{array}$$

For the *if* above we can now calculate $strat(\{\alpha, \beta, \gamma\}, \emptyset)$. As all rules depend on at least one argument, no rules will be added in the first step. And, as both α and β depend (solely) on the first argument, this argument will be added first. Thus we get $\emptyset \rhd_c \{1\} \rhd_c strat(\{\alpha, \beta, \gamma\}, \{1\})$. Then, as the first argument is now in the strategy, we can add α and β. Doing so means that there is only one rule left (γ) and it needs both remaining arguments, which we therefore add. This gives us $\emptyset \rhd_c \{1\} \rhd_c \{\alpha, \beta\} \rhd_c \{2, 3\} \rhd_c strat(\{\gamma\}, \{1, 2, 3\})$. As only γ remains to be added we get $\emptyset \rhd_c \{1\} \rhd_c \{\alpha, \beta\} \rhd_c \{2, 3\} \rhd_c \{\gamma\} \rhd_c \emptyset \rhd_c \emptyset \rhd_c []$, which is $[\{1\}, \{\alpha, \beta\}, \{2, 3\}, \{\gamma\}]$.

Our approach deviates from the *just-in-time* strategy as defined in [17] in two ways. First of all, we do not require arguments to be rewritten in order. This way we basically get the same strategy as before when we permute the arguments of the *if*. We also do not preserve in any way the order in which rules were specified by the user while *just-in-time* would (as far as a strategy allows this).

5.2 Normal forms

Unlike innermost rewriting, JITty rewriting does not allow for a simple build up mechanism (as described in Sect. 4). To avoid rewriting normal forms we want to tag terms to indicate that they are in normal form (or not). A simple way is to add an extra function symbol ν, such that $\nu(t)$ means that t is in normal form (which is done in [18]). However, such an addition results in a time penalty due to additional construction of terms.

Our approach is to introduce extra function symbols f^s for each original function symbol f. Each extra symbol f^s has an annotation s indicating which of its arguments is in normal form. For example, f^{011} indicates that the second and third arguments are in normal form. We will write ϵ for the absence of an annotation (i.e. f^ϵ is equal to f). Note that having these additional symbols

does not add extra costs in construction of terms as the construction only differs in which function symbol is used. And because of the way it is used, normal forms will always be built up of the original function symbols, thus matching does not change at all. The only change is the increase of the number of rewrite methods, which only effects initialisation time and needed (static) memory.

To use these annotations we need to convert the rewrite rules in such a way that they use the annotations. Given a set of variables N and a term t we define $\psi(t, N)$ to be the annotated version of t under the assumption that (the values bound to) the variables of N are in normal form. More precisely (where $[\mathit{true}] = 1$ and $[\mathit{false}] = 0$):

$$\begin{array}{lcl} \psi(x, N) & = & x \\ \psi(f(t_1, \ldots, t_n), N) & = & f^{[t_1 \in N] \ldots [t_n \in N]}(\psi(t_1, N), \ldots, \psi(t_n, N)) \end{array}$$

Let $ar(f)$ denote the arity of function symbol f, $\mathit{vars}(t)$ the set of variables occurring in t and $\mathit{dep}_f(r)$ the indices of arguments of f that the JITty strategy will have rewritten before trying to apply rewrite rule r. We define a **transformation function** ϕ on TRSs such that $\phi((\Sigma, \rightarrow)) = (\Sigma', \rightarrow')$, where Σ' and $\rightarrow'$ are defined as follows:

$$\begin{array}{lcll} \Sigma' & = & \{f^s \ : \ f \in \Sigma \ \wedge \ s \in \bigcup_{0 \leq i \leq ar(f)} \{0,1\}^i \ \} & \\ \rightarrow' & = & \{f^s(t_1, \ldots, t_n) \rightarrow u' \ \textbf{if} \ c' \ : & r = f(t_1, \ldots, t_n) \rightarrow u \ \textbf{if} \ c \ \wedge \\ & & & r \ \in \rightarrow \ \wedge \ s \ \in \ \{\epsilon\} \cup \{0,1\}^n \ \wedge \\ & & & N = \bigcup_{i \in dep_f(r) \ \vee \ s.i=1} \mathit{vars}(t_i) \ \wedge \\ & & & c' = \psi(c, N) \ \wedge \ u' = \psi(u, N) \\ & & \} \ \cup \ \{f^s \rightarrow f \ \textbf{if} \ \mathit{true} \ : \ s \neq \epsilon \ \wedge \ f^s \in \Sigma'\} & \end{array}$$

This translation adds the annotated function symbols and annotated copies of the rewrite rules. It makes sure that the right-hand side of rules correctly uses the annotations based on the annotation of the head symbol of the left-hand side and which arguments will be rewritten before application. It also adds rules to remove the annotations.

For these latter rules the code generation has to be adapted such that these are only applied in case no other rule matches. This way we make sure that normal forms are always without annotations, which ensures that matching does not have to consider annotations at all. The function symbols with an annotation indicating that none of the arguments are in normal form can be safely replaced by the unannotated version.

To illustrate the translation, we look at the following example. Assume the following rules (where [] is the empty list and $a \triangleright l$ is the list l prepended with a):

$$\begin{array}{lcl} \alpha \ : \ \mathit{len}([]) & \rightarrow & 0 \\ \beta \ : \ \mathit{len}(a \triangleright l) & \rightarrow & 1 + \mathit{len}(l) \end{array}$$

Given the above transformation, we obtain the following set of rules. Note that we have annotated the name of the rules as well with the effect that they have on the annotation of *len*.

$$
\begin{array}{lcll}
\alpha & : & len([]) & \to & 0 \\
\alpha^1 & : & len^1([]) & \to & 0 \\
\beta^{\to 1} & : & len(a \rhd l) & \to & 1 + len^1(l) \\
\beta^{1\to 1} & : & len^1(a \rhd l) & \to & 1 + len^1(l) \\
^{1\to} & : & len^1(l) & \to & len(l)
\end{array}
$$

Note that in practice it might not be feasible to use $\phi(R)$ instead of TRS R because of the exponential increase in size. However, it is often sufficient to limit the annotations to, say, 3 arguments.

6 Evaluation

We evaluate the implementations of our mCRL2 rewriters by looking at some benchmarks. These are divided into two parts, viz. benchmarks for rewriting a single closed term and benchmarks for generating labelled transition systems. The reason for this division is that LTS generation, at least as it is implemented in μCRL and mCRL2, uses rewriters in a very specific way.

6.1 LTS generation

The μCRL and mCRL2 toolsets first convert the specification to a *symbolic* LTS, which consists of a list of guarded transitions and the effect on the state these have. Such a guard is an open term that indicates under which valuation of the variables a transition can happen. To generate all such valuations we use a form of narrowing [6]; we repeatedly do case distinction on a variable and rewrite the guard to see if it evaluates to *true* or *false*.

As only a small change is made in each step, most of the time the rewriter will be busy reestablishing that large parts of the guard are still in normal form. Optimisations that avoid normal form rewriting are actually less effective in this setting, as they always need to traverse a term at least once to establish that it is a normal form.

For the LTS benchmarks we have taken four specifications (chatboxt, 1394-fin, ccp33 and commprot) from the μCRL toolset, converted them to symbolic LTSs that are easily translatable to LOTOS [10] (for the CADP toolset [8]) and mCRL2. The used specifications differ slightly from the versions in the μCRL toolset to be able to translate to CADP. Note that, unlike the μCRL and mCRL2 toolsets, CADP is not specialised in handling these symbolic LTSs, which can negatively influence their results. All tools were used on the same machine with 2 gigabytes of memory (of which the tools were only allowed to use 1.5 gigabytes to avoid swapping). Note that we write OoM (out of memory) in case a tool was terminated because it needed more than the allowed amount of memory. For this reason we included additional variants of benchmarks limited to an amount of states that all tools could handle.

Looking at Table 1, we see that our JITty implementation performs better on average than any of the others. The exact difference depends highly on

	# states	CADP	μCRL	mCRL2	
				Innermost	*JITty*
chatboxt	65536	1.3s	5.0s	4.0s	3.5s
1394-fin	400	65.3s	0.1s	0.5s	0.4s
1394-fin	371804	OoM	103.8s	212.1s	92.3s
ccp33	7000	25.5s	27.6s	61.8s	8.7s
ccp33	20000	OoM	79.0s	171.9s	26.2s
commprot	700	53.9s	11.0s	12.4s	13.0s
commprot	5000	OoM	77.8s	92.1s	93.0s

Table 1: LTS generation benchmarks

the chosen example, as some depend more heavily on functions that allow for JITty techniques. In the CADP column we see several OoMs indicating the tool needed more than the allowed amount of memory.

Our innermost implementation is about two times as slow as μCRL in the, calculational-wise, heavier cases. This could be either because μCRL also applies JITty-like techniques in a limited fashion or because their implementation does not need to deal with applicative terms. The implementation is otherwise very similar. Given the times in Table 1 it is clear that only in case there is a significant difference in execution time between the mCRL2 implementations there is also a significant difference with μCRL. This seems to support the idea that our innermost rewriter is slower than μCRL because the latter also applies some JITty techniques.

6.2 Closed term rewriting

To investigate the performance of our rewriters in a more general setting we look at the benchmarks in Table 2. These benchmarks consist of only a single closed data term that needs to be rewritten to normal form. In order to test the rewriters of the LTS generation tools we again use μCRL specifications as before, only with a single process that can do precisely one transition which has the term to be rewritten as an argument (such that these tools are effectively only rewriting that term). In addition to the LTS generation tools we also consider the functional language tools Maude [5], Glasgow Haskell Compiler (GHC) [11], Clean [15] and ASF+SDF. For these tools the process part of the specification is discarded in the conversion.

The benchmarks we use are a naive Fibonacci implementation (fib(32)), benchmarks as used in [13] (evalexp, evalsym, evaltree) and a binary search (b.search). Fibonacci and evalsym are mainly calculational benchmarks, evalexpr differentiates eager and lazy implementations and evaltree is a memory extensive benchmark. The binary search is a benchmark that takes an increasing function, a value and a bound and searches that function (in the domain determined by the bound) for the given value. This benchmark is mainly a test for applicative terms (as the search function takes a function as argument), but

also requires a lazy implementation for reasonable execution. The function we use as argument is the Fibonacci function. We write NA (not applicable) in Table 2 for tools that do not support applicative terms.

	Maude	GHC	Clean	ASF	CADP	μCRL	mCRL2	
							Inner.	*JITty*
fib(32)	23.4s	4.0s	2.6s	2.7s	2.4s	2.3s	4.0s	11.2s
evalexpr	3.3s	0.4s	0.3s	OoM	0.5s	OoM	OoM	5.4s
evalsym	231.3s	18.7s	15.8s	36.3s	OoM	19.0s	49.3s	254.2s
evaltree	16.7s	OoM	2.1s	1.6s	0.6s	1.0s	1.9s	25.6s
b.search	NA	4.5s	2.5s	NA	NA	NA	OoM	10.8s

Table 2: Closed term rewriting benchmarks

From the benchmarks in Table 2 we can see that in general the rewriters of the LTS generators can compete with the fastest rewriters for functional languages available, which seems to indicate that supporting open term rewriting and implicit substitution is not a bottleneck. We can also see that our JITty implementation is often significantly slower than the others and is more comparable to Maude, which uses an interpreting rewriter. This is likely due to the fact that JITty always has to build the result of rule application before rewriting that term, which is very expensive in our implementation. The memory extensive evaltree benchmarks, where JITty is about twelve times slower than our innermost rewriter, seems to support this. Also note that the evalsym benchmark, meant to test pure calculation speed, favors those that use a lazy implementation (ASF+SDF and the mCRL2 innermost rewriter are the only strict innermost rewriters).

7 Conclusion

We have described the implementation of the rewriters of the mCRL2 toolset. The implementation of the innermost rewriter is very similar to the implementation of the μCRL rewriter and the rewriter used in ASF+SDF. The second implementation is that of a compiling JITty rewriter, which is, as far as we know, the first of its kind.

Benchmarks are given to illustrate the improvement this JITty rewriter is over the innermost rewriters used for LTS generation. For closed term rewriting we have shown that our innermost rewriter can compete with the best rewriters currently available (ignoring the effects of lazy rewriting) and that JITty is a bit slower. The latter is likely due to the fact that in this implementation more intermediate terms have to be constructed, which is quite expensive.

The fact that the rewriters used for LTS generation can clearly compete with the fastest rewriters for functional languages seems to suggest that adapting the latter to support open term rewriting (which is essential for LTS generation) should not be a problem. That is, unless these functional languages support

additional features with respect to the more basic languages used in process specifications that are fundamentally in conflict with efficient open term rewriting. In any case, such an adaptation would allow developers and users of tools centered around process behaviour and theorem proving (and most likely other fields as well) to have direct access to the functionality offered by the expertise of the functional programming community.

Most significant future work will be the improvement of the JITty rewriter for closed term rewriting and especially the study of the implications of the restrictions we have put on higher-order rewriting.

References

[1] S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proceedings of the 3rd International Conference on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 143–157. Springer, 1992.

[2] L. Augustsson. Compiling pattern matching. In J.-P. Jouannaud, editor, *Proceedings of a Conference on Functional Programming Languages and Computer Architecture (FPCA)*, volume 523 of *Lecture Notes in Computer Science*, pages 368–381. Springer-Verlag, 1985.

[3] L. D. Baxter. *The complexity of unification.* PhD thesis, University of Waterloo, 1976.

[4] S. C. C. Blom, W. J. Fokkink, J. F. Groote, I. van Langevelde, B. Lisser, and J. C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification: 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag, 2001.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[6] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 243–320. MIT Press, 1990.

[7] W. Fokkink, J. Kamperman, and P. Walters. Lazy rewriting on eager machinery. *ACM Transactions on Programming Languages and Systems*, 22(1):45–86, 2000.

[8] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4, 2002.

[9] J. F. Groote, A. H. J. Mathijssen, M. J. van Weerdenburg, and Y. S. Usenko. From μCRL to mCRL2: Motivation and outline. In L. Aceto and A. D. Gordon, editors, *Proc. Workshop Algebraic Process Calculi: The First Twenty Five Years and Beyond*, volume 162 of *Electronic Notes in Theoretical Computer Science*, pages 191–196, 2006.

[10] ISO. ISO 8807: Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour. Standard, International Standards Organization, Geneva, Switzerland, 15 February 1987. First edition.

[11] J. Launchbury and P. M. Sansom, editors. *The Glasgow Haskell Compiler: A Retrospective.*, Workshops in Computing. Springer, 1993.

[12] A. H. J. Mathijssen and A. J. Pretorius. Specification, analysis and verification of an automated parking garage. Technical Report 05/25, Eindhoven University of Technology, ISSN 0926-4515, 2005.

[13] P. Olivier. *A Framework for Debugging Heterogeneous Applications.* PhD thesis, University of Amsterdam, 2000.

[14] S. L. Peyton Jones. *The implementation of functional programming languages.* Prentice-Hall, 1987.

[15] M. J. Plasmeijer. Clean: a programming environment based on term graph rewriting. *Electronic Notes in Theoretical Computer Science*, 2:215–221, 1995.

[16] P. Schnoebelen. Refined compilation of pattern-matching for functional languages. *Science of Computer Programming*, 11(2):133–159, 1988.

[17] J. van de Pol. Just-in-time: On strategy annotations. In B. Gramlich and S. Lucas, editors, *WRS 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming*, volume 57 of *Electronic Notes in Theoretical Computer Science*, pages 41–63, 2001.

[18] J. C. van de Pol. JITty: a rewriter with strategy annotations. In S. Tison, editor, *Rewriting Techniques and Applications : 13th International Conference, RTA 2002, Copenhagen, Denmark, July 22-24, 2002. Proceedings*, volume 2378 of *Lecture Notes in Computer Science*, pages 367–370. Springer-Verlag, 2002.

[19] M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: A component-based language development environment. In R. Wilhelm, editor, *Compiler Construction: 10th International Conference, CC 2001*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer, 2001.

[20] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):334–368, 2002.

[21] M. G. T. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software: Practice & Experience*, 30(3):259–291, 2000.

[22] M. Vittek. A compiler for nondeterministic term rewriting systems. In *Rewriting Techniques and Applications, 7th International Conference, RTA-96, New Brunswick, NJ, USA, July 27-30, 1996, Proceedings*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–167. Springer, 1996.