Rewriter Implementation Notes

Wieger Wesselink

20th March 2023

1 Introduction

This document describes rewrite algorithms that can be applied in the mCRL2 tool set. Currently only a prototype implementation in python is available. Most of the content is based on [Wee09] and on [Wul09].

1.1 Higher order rewriting

There are several formalisms for higher order rewriting. We choose higher-order rewriting systems (HRSs) introduced by Nipkow. In [vR01] HRSs are summarized as follows:

In a HRS we work modulo the $\beta\eta$-relation of simply typed λ-calculus. *Types* are built from a non-empty set of base types and the binary type constructor $\to$ as usual. For every type we assume a countably infinite set of *variables* of that type, written as $x, y, z, \ldots$. A *signature* is a non-empty set of typed function symbols. The set of *preterms* of type A over a signature Σ consists exactly of the expressions $s$for which we can derive $s : A$ using the following rules:

1. $x : A$ for a variable x of type A,
2. $f : A$ for a function symbol f of type A in Σ,
3. if $A = A' \to A''$, and $x : A'$ and $s : A''$, then $(x.s) : A$,
4. if $s : A' \to A$ and $t : A'$, then $(s\ t) : A$.

The abstraction operator $\_.\_$ binds variables, so occurrences of x in s in the preterm $x.s$ are bound. We work modulo type-preserving α-conversion and assume that bound variables are renamed whenever necessary in order to avoid unintended capturing of free variables. Parentheses may be omitted according to the usual conventions. We make use of the usual notions of *substitution* of a preterm t for the free occurrences of a variable x in a preterm s, notation $s[x := t]$, and *replacement in a context*, notation $C[t]$. We write $s \supseteq s'$ if s' is a subpreterm of s, and use $\supset$ for the strict subpreterm relation.

The *β-reduction relation*, notation $\to_\beta$, is the smallest relation on preterms that is compatible with formation of preterms and that satisfies the following:

$$(x, s)t \to_\beta s[x := t]$$

The *restricted η-expansion relation*, notation $\to_{\overline{\eta}}$, is defined as follows. We have

$$C[s] \to_{\overline{\eta}} C[x.(s\ x)]$$

if $s : A \to B$, and $x : A$ is a fresh variable, and no β-redex is created (hence the terminology restricted η-expansion). The latter condition is satisfied if s is not an abstraction (so not of the form $z.s'$), and doesn't occur in $C[s]$ as the left part of an application (so doesn't occur in a sub-preterm of the form $(s\ s')$).

In the sequel we employ only preterms in $\overline{\eta}$-normal form, where every sub-preterm has the right number of arguments. Instead of $s_0 s_1 \dots s_m$ we often write $s_0(s_1, \dots, s_m)$. A preterm is then of the form $x_1 \dots x_n.s_0(s_1, \dots, s_m)$ with $s_0(s_1, \dots, s_m)$ of base type and all s_i in $\overline{\eta}$-normal form.

A *term* is a preterm in β-normal form. It is also in $\overline{\eta}$-normal form because $\overline{\eta}$-normal forms are closed under β-reduction. A term is of the form $x_1 \dots x_n.a(s_1, \dots, s_m)$ with a a function symbol or a variable. Because the $\beta\overline{\eta}$-reduction relation is confluent and terminating on the set of preterms, every $\beta\overline{\eta}$-equivalence class of preterms contains a unique term, which is taken as the representative of that class.

Because in the discussion we will often use preterms, we use here the notation s^σ for the replacement of variables according to the substitution σ (*without* reduction to β-normal form), and write explicitly $s^\sigma \downarrow_\beta$for its β-normal form. This is in contrast with the usual notations for HRSs.

A *rewrite rule* is a pair of terms (l, r), written as $l \to r$, satisfying the following requirements:

1. l and r are of the same base type,
2. l is of the form $f(l_1, \dots, l_n)$,
3. all free variables in r occur also in l,
4. a free variable x in l occurs in the form $x(y_1, \dots, y_n)$ with y_i η-equivalent to different bound variables.

The last requirement guarantees that the rewrite relation is decidable because unification of patterns is decidable. The rewrite rules induce a rewrite relation $\to$ on the set of terms which is defined by the following rules:

1. if $s \to t$ then $x(\dots, s, \dots) \to x(\dots, t, \dots)$,
2. if $s \to t$ then $f(\dots, s, \dots) \to f(\dots, t, \dots)$,
3. if $s \to t$ then $x.s \to x.t$,
4. if $l \to r$ is a rewrite rule and σ is a substitution then $l^\sigma \downarrow_\beta \to r^\sigma \downarrow_\beta$.

The last clause in this definition shows that HRSs use higher-order pattern matching, unlike AFSs, where matching is syntactic.

1.2 mCRL2 terms

In mCRL2 we have the following terms:

$$t := x \mid f \mid t(t, \cdots, t) \mid \lambda_x.t \mid \forall_x.t \mid \exists_x.t \mid t \ whr \ x = t'$$

where t is a term, x is a variable and f is a function symbol.

Remark 1 *This needs to be further elaborated. Terms are typed, and function symbols (and terms?) have an arity. The term $t(t, \cdots, t)$ is rather unusual, but it is covered by HRSs (?).*

Remark 2 *In fact the mCRL2 language uses slightly more general terms: $\lambda_{x_1 \dots x_n}.t$, $\forall_{x_1 \dots x_n}.t$, $\exists_{x_1 \dots x_n}.t$ and $t \ whr \ x_1 = t_1, \dots, x_n = t_n$.*

For a rewrite algorithm $rewr$ the following rules are suggested:

$$\begin{aligned} rewr(\lambda_x.t, \sigma) &= \lambda_{x'}.rewr(t, \sigma[x := x']) \\ rewr(\forall_x.t, \sigma) &= \forall_{x'}.rewr(t, \sigma[x := x']) \\ rewr(\exists_x.t, \sigma) &= \exists_{x'}.rewr(t, \sigma[x := x']) \\ rewr(t \ whr \ x &= t', \sigma) = rewr(t, \sigma[x := t']) \end{aligned}$$

where x' is a fresh variable not appearing in t.

Remark 3 *In a rewrite algorithm, the term types are unused. One can add correctness checks for proper typing however.*

Remark 4 *Types in mCRL2 need to be rewritten to normal form as well. A very simple rewrite system can be defined for this.*

Remark 5 *What about normal forms for terms containing λ-expressions and/or quantifiers? Expressions can be equal modulo alpha-conversion, so the ATerm equality doesn't work here.*

Remark 6 *In a rewrite algorithm one has to explicitly describe where α-conversion and β-reduction is being done. Doing $\overline{\eta}$-expansion is probably not necessary.*

1.3 Types

A *base type* is a non-function type, typical examples are the Booleans or Natural numbers. Let B be a non-empty set of base types and $b \in B$. The set of types is inductively defined as follows:

$$type ::= b \mid type \times type \mid type \to type,$$

where $\to$ is the function-type constructor. The type constructor associates to the left, for example:

$$b \to b \to b \text{ is the same as } (b \to b) \to b.$$

Product types are often not present in treatment of simply-typed lambda calculus. We need them later to type non-lambda terms.

The *arity* of a type A is a natural number, denoted $arity(A)$, which is inductively defined on the structure of A as follows:

$$\begin{array}{ll} arity(A) = 0 & \text{if } A \text{ is a base type,} \\ arity(A \to A') = arity(A') + 1 & \end{array}$$

A *signature* Σ is a non-empty set of *function symbols* each of which has a type. We write $f : A$ to denote that symbol f has type A and extend the notion arity to symbols such that if $f : A$ then $arity(f) = arity(A)$. Symbols with arity zero are called *constants*.

Let Σ be a signature and let χ_A be a countably finite set of variables of type A such that $\Sigma \cap \chi_A = \emptyset$, for each type A. The set of terms over Σ, denoted $\mathcal{T}(\Sigma)$, is inductively defined as

- Let $x \in \chi_A$ be a variable of type A then $x \in \mathcal{T}(\Sigma)$.
- Let $f \in \Sigma$ be a function symbol of type $A_1 \times \ldots \times A_n \to B$, and $t_i : A_i$ for all $i \in \{1, \ldots, n\}$, then $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma)$ is a term of type B.
- Let $t : A_1 \times \ldots \times A_n \to B$, and $t_i : A_i$ for all $i \in \{1, \ldots, n\}$, then $t(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma)$ is a term of type B.
- Let $x \in \chi_A$ be a variable of type A and t a term of type B, then $\lambda_x.(t) \in \mathcal{T}(\Sigma)$ is a term of type $A \to B$.

1.4 Simple terms

Simple terms are terms with the following syntax:

$$t := x \mid f \mid f(t, \cdots, t), \tag{1}$$

where t is a term, x is a variable and f is a function symbol.

1.5 Applicative terms

Applicative terms are an extension of simple terms:

$$t := x \mid f \mid t(t, \cdots, t). \tag{2}$$

The set of all variables is denoted by $\mathbb{V}$, the set of all function symbols by $\mathbb{F}$ and the set of all terms by $\mathbb{T}$. In this document we use the convention that $x, y \in \mathbb{V}$, that $t, u \in \mathbb{T}$, and that $f, g \in \mathbb{F}$.

We write $var(t)$ for the set of variables that occur in t. Formally:

$$\begin{array}{lcl} var(x) & = & \{x\} \\ var(f) & = & \emptyset \\ var(t(t_1, \cdots, t_n) & = & var(t) \cup \bigcup_{i=1 \cdots n} var(t_i). \end{array}$$

1.6 Subterms

To facilitate operations on subterms we inductively define positions ($\mathbb{P}$) as follows. A position is either ϵ (the empty position) or an index i (from 1,2,$\cdots$) combined with a position π, notation $i \cdot \pi$. We lift $\cdot$ to an associative operator on positions with ϵ as its unit element and often write just i for the position $i \cdot \epsilon$. We write the subterm of t at position π as $t|_\pi$ and we write term t with the subterm at position π replaced by u as $t[u]_\pi$. These operations are defined as follows.

$$\begin{array}{lcll} t|_\epsilon & = & t & \\ t(t_1, \cdots, t_n)|_{i \cdot \pi} & = & t_i|_\pi & \text{if } 1 \leq i \leq n \\ t[u]_\epsilon & = & u & \\ x[u]_{i \cdot \pi} & = & x & \\ f(t_1, \cdots, t_n)[u]_{i \cdot \pi} & = & f(t_1, \cdots, t_{i-1}, t_i[u]_\pi, t_{i+1}, \cdots, t_n) & \text{if } i \leq n \\ f(t_1, \cdots, t_n)[u]_{i \cdot \pi} & = & f(t_1, \cdots, t_n) & \text{if } i > n \end{array}$$

Some examples are:

$$\begin{array}{l} f(x, g(y))|_1 = x \\ f(x, g(y))|_{2 \cdot 1} = y \\ f(x, g(y))[h(x)]_2 = f(x, h(x)) \end{array}$$

1.7 Substitutions

A substitution is a function $\sigma : \mathbb{V} \to \mathbb{T}$. A substitution σ can also be applied to a term t. This is denoted by $t\sigma$ and it is defined as

$$\begin{array}{lcl} x\sigma & = & \sigma(x) \\ f\sigma & = & f \\ t(t_1, \cdots, t_n)\sigma & = & t\sigma(t_1\sigma, \cdots, t_n\sigma). \end{array}$$

1.8 Rewrite rules

A rewrite rule is a rule $l \to r$ **if** c, with $l, r, c \in \mathbb{T}$. We put three restrictions on rewrite rules:

$$\begin{array}{ll} 1) & l \text{ is a simple term} \\ 2) & l \notin \mathbb{V} \\ 3) & var(r) \cup var(c) \subseteq var(l) \end{array}$$

For a set R of rewrite rules we define the rewrite relation $\to_R$ as follows: $t \to_R u$ if there is a rule $l \to r$ **if** c in R , a position π and a substitution σ such that

$$t|_\pi = l\sigma \wedge u = t[r\sigma]_\pi \wedge \eta(c\sigma), \tag{3}$$

where η is a boolean function that determines if a condition is true. We write $\to$ instead of $\to_R$ if no confusion can occur. We write $\to_R^*$ for the reflexive and transitive closure of $\to_R$ and $t \nrightarrow_R$ if there is no u such that $t \to_R u$. A normal form is a term u such that $t \to_R^* u$ and $u \nrightarrow_R$.

1.9 Rewrite algorithm

We now fomulate an abstract rewrite algorithm *rewrite*, where we assume that R is a given, fixed set of rewrite rules.

function $rewrite(t)$
$u := t$
while $\{v \mid u \rightarrow_R v\} \neq \emptyset$ **do**
    choose v **such that** $u \rightarrow_R v$
    $u := v$
return u

Note that this algorithm does not need to terminate. In practice we are also interested in an algorithm $rewrite(t, \sigma)$, that applies a substitution σ to the variables in t during rewriting. The specification of this algorithm is simply

$$rewrite(t, \sigma) = rewrite(t\sigma).$$

The reason we are interested in such an algorithm is that it can be implemented more efficiently than the straightforward solution to first compute $u = t\sigma$ and then compute $rewrite(u)$.

2 Match trees

A match tree is a tree structure that represents a number of rewrite rules that have left hand sides with the same function symbol as head. It is used to compute all possible results of applying one of these rules to a term. Currently match trees are only defined for simple terms. A match tree consists of nodes of the following types:

- $F(f, T, U)$: If the current term has the form $f(t_1, \cdots, t_n)$ replace the top of the stack by $t_1 \rhd \cdots \rhd t_n$ and continue with T, otherwise continue with U.
- $S(x, T)$: Assign the current term to variable x and continue with T.
- $M(x, T, U)$: If the current term is equal to x continue with T, otherwise continue with U.
- $R(Q)$: Return Q
- X : Return the empty set.
- $N(n, T)$: Remove n elements from the stack and continue with T. We abbreviate $N(1, T)$ as $N(T)$.
- $E(T, U)$: If the stack is not empty continue with T, otherwise continue with U.
- $C(t, T, U)$: If t evaluates to *true*, continue with T, otherwise continue with U.

where f is a function symbol, x is a variable, t is a term, Q is a set of terms annotated with a rewrite rule, and T and U are match tree nodes.

2.1 Evaluating a match tree

Let l be a sequence of terms, and let σ be an arbitrary substitution function. Then the evaluation of a match tree with arguments l and σ is a set of terms and is defined as follows:

$$
\begin{array}{lcl}
F(f,T,U)(l,\sigma) & = & \begin{cases} \emptyset & \text{if } l = [] \\ T(m,\sigma) & \text{if } l = f \rhd m \\ T(t_1 \rhd \cdots \rhd t_n \rhd m,\sigma) & \text{if } l = f(t_1,\cdots,t_n) \rhd m \\ U(l,\sigma) & \text{if } l = g(t_1,\cdots,t_n) \rhd m \wedge f \neq g \\ U(l,\sigma) & \text{if } l = x \rhd m \end{cases} \\
X(l,\sigma) & = & \emptyset \\
R(Q)(l,\sigma) & = & \begin{cases} \{\sigma(t) \mid t^\alpha \in Q\} & \text{if } l = [] \\ \emptyset & \text{if } l \neq [] \end{cases} \\
S(x,T) & = & \begin{cases} \emptyset & \text{if } l = [] \\ T(l,\sigma[x \to t]) & \text{if } l = t \rhd m \end{cases} \\
M(x,T,U)(l,\sigma) & = & \begin{cases} \emptyset & \text{if } l = [] \\ T(l,\sigma) & \text{if } l = t \rhd m \wedge \sigma(x) = t \\ U(l,\sigma) & \text{if } l = t \rhd m \wedge \sigma(x) \neq t \end{cases} \\
N(n,T)(l,\sigma) & = & \begin{cases} \emptyset & \text{if } |l| < n \\ T(m,\sigma) & \text{if } l = t_1 \rhd \cdots \rhd t_n \rhd m \end{cases} \\
E(T,U)(l,\sigma) & = & \begin{cases} U(l,\sigma) & \text{if } l = [] \\ T(l,\sigma) & \text{if } l = t \rhd m \end{cases} \\
C(t,T,U)(l,\sigma) & = & \begin{cases} T(l,\sigma) & \text{if } t\sigma \text{ evaluates to } true \\ U(l,\sigma) & \text{if } t\sigma \text{ does not evalute to } true \end{cases}
\end{array}
$$

where T and U are match trees, f and g are function symbols, l and m are sequences of terms and t and t_i are terms. The evaluation of a match tree T in a single term t with substitution σ is defined as $T([t],\sigma)$.

2.2 Building a match tree

Let α be a rewrite rule given by $l \to r$.Then we define the match tree $match\_tree(\alpha) = \gamma([l],\{r^\alpha\},\emptyset)$, where γ is defined as:

$$
\begin{array}{lcl}
\gamma([],Q,V) & = & R(Q) \\
\gamma(x \rhd s,Q,V) & = & \begin{cases} S(x,N(\gamma(s,Q,V \cup \{x\}))) & \text{if } x \notin V \\ M(x,N(\gamma(s,Q,V \cup \{x\})),X) & \text{if } x \in V \end{cases} \\
\gamma(f(t_1,\cdots,t_n) \rhd s,Q,V) & = & F(f,\gamma(t_1 \rhd \cdots \rhd t_n \rhd s,Q,V),X)
\end{array}
$$

Match trees are only defined for rewrite rules with simple terms at the left hand side.

2.3 Joining match trees

Two match trees $left$ and $right$ can be joined into one using the operator $||$, which is defined as follows: $left \mid\mid right =$

$right$	if	$head(left) = X$		
$left$	if	$head(right) = X$		
$E(left, right)$	if	$head(right) = R$		
$E(right, left)$	if	$head(left) = R$		
$R(Q \cup Q')$	if	$left = R(Q)$	and	$right = R(Q')$
$S(x, T \mid\mid right)$	if	$left = S(x, T)$	and	$head(right) \in \{F, S, U\}$
$M(y, left \mid\mid U, left)$	if	$left = S(x, T)$	and	$right = M(y, U, V)$
$M(x, T \mid\mid right, T' \mid\mid right)$	if	$left = M(x, T, T')$	and	$head(right) \in \{F, M, N, S\}$
$S(x, left \mid\mid U)$	if	$left = F(f, T, T')$	and	$right = S(x, U)$
$M(x, left \mid\mid U, left)$	if	$left = F(f, T, T')$	and	$right = M(x, U, U')$
$F(f, T \mid\mid U, T')$	if	$left = F(f, T, T')$	and	$right = F(f, U, U')$
$F(f, T, T' \mid\mid right)$	if	$left = F(f, T, T')$	and	$right = F(g, U, U'),\ f \neq g$
$F(f, T \mid\mid N(ar(f), U), T' \mid\mid right)$	if	$left = F(f, T, T')$	and	$right = N(U)$
$S(x, left \mid\mid U)$	if	$left = N(T)$	and	$right = S(x, U)$
$M(x, left \mid\mid U, left \mid\mid U')$	if	$left = N(T)$	and	$right = M(x, U, U')$
$F(f, N(ar(f), T) \mid\mid U, left)$	if	$left = N(T)$	and	$right = F(f, U, X)$
$N(T \mid\mid U)$	if	$left = N(T)$	and	$right = N(U)$
$E(T, right \mid\mid T')$	if	$left = E(T, T')$	and	$head(right) \in \{F, M, N, R, S\}$,

where $head$ is defined as $head(F(f, T, U)) = F$, $head(R(Q)) = R$ etc.

2.4 Optimizing match trees

The result of joining match trees is often not optimal. This section gives two algorithms *reduce* and *clean* to optimize match trees.

$$
\begin{array}{lcll}
reduce(X) & = & X & \\
reduce(F(f,T,U)) & = & reduce_F(F(f,T,U),\emptyset) & \\
reduce(S(x,T)) & = & reduce_S(S(x,T),\emptyset) & \\
reduce(M(x,T,U)) & = & reduce_M(M(x,T,U),\emptyset,\emptyset) & \\
reduce(C(t,T,U)) & = & C(t,reduce(T),reduce(U)) & \\
reduce(N(n,T)) & = & N(n,reduce(T)) & \\
reduce(E(T,U)) & = & E(t,reduce(T),reduce(U)) & \\
reduce(R(Q)) & = & R(Q) &
\end{array}
$$

$$
\begin{array}{lcll}
reduce_F(X,F) & = & F & \\
reduce_F(F(f,T,U),F) & = & reduce_F(U,F) & \text{if } f \in F \\
reduce_F(F(f,T,U)) & = & F(f,reduce(T),reduce_F(U,F\cup\{f\})) & \text{if } f \notin F \\
reduce_F(N(n,T)) & = & reduce_M(M(x,T,U),\emptyset,\emptyset) &
\end{array}
$$

$$
\begin{array}{lcll}
reduce_S(X,\emptyset) & = & X & \\
reduce_S(X,\{x\}\cup V) & = & S(x,reduce(X[x/V],\emptyset)) & \\
reduce_S(F(f,T,U),\emptyset) & = & reduce_F(F(f,T,U),\emptyset) & \\
reduce_S(F(F,T,U),\{x\}\cup V) & = & S(x,reduce_F(F(f,T,U)[x/V],\emptyset) & \\
reduce_S(S(x,T),V) & = & reduce_S(T,V\cup\{x\}) & \\
reduce_S(N(n,T),\emptyset) & = & reduce(N(n,T),\emptyset) & \\
reduce_S(N(n,T),\{x\}\cup V) & = & S(x,reduce(N(n,T)[x/V])) &
\end{array}
$$

$$
\begin{array}{lcll}
reduce_M(X,M_t,M_f) & = & reduce(X) & \\
reduce_M(F(f,T,U),M_t,M_f) & = & reduce_F(F(f,T,U),\emptyset) & \\
reduce_M(S(x,T),M_t,M_f) & = & reduce_S(S(x,T),\emptyset) & \\
reduce_M(M(x,T,U),M_t,M_f) & = & reduce_M(T,M_t,M_f) & \text{if } x \in M_t \\
reduce_M(M(x,T,U),M_t,M_f) & = & reduce_M(U,M_t,M_f) & \text{if } x \in M_f \\
reduce_M(M(x,T,U),M_t,M_f) & = & M(x,reduce_M(T,M_t\cup\{x\},M_f), & \text{if } x \notin M_t \wedge x \notin M_f \\
 & & reduce_M(U,M_t\cup\{x\},M_f\cup\{x\})) & \\
reduce_M(N(n,T)) & = & reduce(N(n,T)), &
\end{array}
$$

with

$$
\begin{array}{lcll}
X[x/V] & = & X & \\
F(f,T,U)[x/V] & = & F(f,T[x/V],U[x/V]) & \\
S(x,T)[y/V] & = & S(x,T[y/(V\setminus\{x\})]) & \\
M(x,T,U)[y/V] & = & M(y,T[y/V],U[y/V]) & \text{if } x \in V \\
M(x,T,U)[y/V] & = & M(x,T[y/V],U[y/V]) & \text{if } x \notin V \\
C(t,T,U)[x/V] & = & C(t[x/y:y\in V],T[x/V],U[x/V]) & \\
N(n,T)[x/V] & = & N(n,T[x/V]) & \\
E(T,U)[x/V] & = & E(t,T[x/V],U[x/V]) & \\
R(Q)[x/V] & = & R(Q[x/y:y\in V]) &
\end{array}
$$

$$clean(T) \quad = \quad T' \quad \text{if } \chi(T) = \langle T', V\rangle\,,$$

where χ is defined as

$$
\begin{array}{lcll}
\chi(X) & = & \langle X, \emptyset \rangle & \\
\chi(F(f,T,U)) & = & \langle F(f,T',U'), V \cup W \rangle & \text{if } \chi(T) = \langle T', V \rangle \wedge \chi(U) = \langle U', W \rangle \\
\chi(S(x,T)) & = & \langle S(x,T'), V \setminus \{x\} \rangle & \text{if } \chi(T) = \langle T', V \rangle \wedge x \in V \\
\chi(S(x,T)) & = & \langle T', V \rangle & \text{if } \chi(T) = \langle T', V \rangle \wedge x \notin V \\
\chi(M(x,T,U)) & = & \langle T', V \rangle & \text{if } \chi(T) = \langle T', V \rangle \wedge \chi(U) = \langle U', W \rangle \wedge T' = U' \\
\chi(M(x,T,U)) & = & \langle M(x,T',U'), V \cup W \cup \{x\} \rangle & \text{if } \chi(T) = \langle T', V \rangle \wedge \chi(U) = \langle U', W \rangle \wedge T' \neq U' \\
\chi(C(t,T,U)) & = & \langle T', V \rangle & \text{if } \chi(T) = \langle T', V \rangle \wedge \chi(U) = \langle U', W \rangle \wedge T' = U' \\
\chi(C(t,T,U)) & = & \langle C(t,T',U'), V \cup W \cup var(t) \rangle & \text{if } \chi(T) = \langle T', V \rangle \wedge \chi(U) = \langle U', W \rangle \wedge T' \neq U' \\
\chi(N(n,T)) & = & \langle N(n,T'), V \rangle & \text{if } \langle T', V \rangle = \chi(T) \\
\chi(E(T,U)) & = & \langle T', V \rangle & \text{if } \chi(T) = \langle T', V \rangle \wedge \chi(U) = \langle X, W \rangle \\
\chi(E(T,U)) & = & \langle U', W \rangle & \text{if } \chi(T) = \langle X, V \rangle \wedge \chi(U) = \langle U', W \rangle \\
\chi(E(T,U)) & = & \langle E(T',U'), V \cup W \rangle & \text{if } \chi(T) = \langle T', V \rangle \wedge \chi(U) = \langle U', W \rangle \wedge T' \neq X \wedge U' \neq X \\
\chi(R(Q)) & = & \langle R(Q), var(Q) \rangle &
\end{array}
$$

2.5 Prioritized rewrite rules

By adding priorities to rewrite rules, the selection of rewrite rules that is considered for a term can be reduced. We model priorities of rewrite rules using a function φ, that returns the rules of highest priority for a set of rules. So $\varphi(R) \subseteq R$ and $\varphi(R) = \emptyset$ if and only if $R = \emptyset$. We define a function *prior* that applies a priority function φ to a match tree:

$$
\begin{array}{lcl}
prior(X, \varphi) & = & X \\
prior(F(f,T,U), \varphi) & = & F(f, prior(T,\varphi), prior(U,\varphi)) \\
prior(S(x,T), \varphi) & = & S(x, prior(T,\varphi)) \\
prior(M(x,T,U), \varphi) & = & M(x, prior(T,\varphi), prior(U,\varphi)) \\
prior(C(t,T,U), \varphi) & = & C(t, prior(T,\varphi), prior(U,\varphi)) \\
prior(N(n,T), \varphi) & = & N(n, prior(T,\varphi)) \\
prior(R(Q), \varphi) & = & R(\varphi(Q)) \\
prior(E(T,U), \varphi) & = & E(prior(T,\varphi), prior(U,\varphi)).
\end{array}
$$

The effect of applying *prior* to a match tree is that the R-nodes will contain less elements. This can be useful to remove unwanted results. Consider for example the rewrite system

$$
\left\{ \begin{array}{lcl} x = x & \rightarrow & true \\ x = y & \rightarrow & false \end{array} \right.
$$

This system can have both $true$ and $false$ as a result of rewriting the term $true = true$. But if we give the first equation a higher priority than the second, the undesired derivation $true = true \rightarrow false$ is eliminated.

3 Rewriting

In this section we describe rewriting strategies. For the moment we only consider innermost rewriting.

3.1 Rewriting using match trees

Suppose that we have a rewrite system, and that for each function symbol f a match tree M_f has been constructed that corresponds to rewrite rules with head symbol f. We define the function

$rewr_M$ as:

$$
\begin{array}{lcl}
rewr_M(x,\sigma) & = & \sigma(x) \\
rewr_M(f,\sigma) & = & \begin{cases} f & \text{if } M_f([f],\sigma) = \emptyset \\ u \in M_f([f],\sigma) & \text{if } M_f([f],\sigma) \neq \emptyset \end{cases} \\
rewr_M(f(t_1,\cdots,t_n),\sigma) & = & \begin{cases} f(t_1,\cdots,t_n) & \text{if } M_f([f(t_1,\cdots,t_n)],\sigma) = \emptyset \\ u \in M_f([f(t_1,\cdots,t_n)],\sigma) & \text{if } M_f([f(t_1,\cdots,t_n)],\sigma) \neq \emptyset \end{cases} \\
rewr_M(x(t_1,\cdots,t_n),\sigma) & = & x(t_1,\cdots,t_n) \\
rewr_M(u(u_1,\cdots,u_m)(t_1,\cdots,t_n),\sigma) & = & u(u_1,\cdots,u_m)(t_1,\cdots,t_n)
\end{array}
$$

3.2 Innermost rewriting

We now define an algorithm $rewr_I$ for innermost rewriting. It is defined for applicative terms. We assume that $\sigma(x)$ is always in normal form already.

$$
\begin{array}{lcl}
rewr_I(x,\sigma) & = & \sigma(x) \\
rewr_I(f,\sigma) & = & rewr_M(f,\sigma) \\
rewr_I(t(t_1,\cdots,t_n),\sigma) & = & rewr_M(rewr_I(t,\sigma)(rewr_I(t_1,\sigma),\cdots,rewr_I(t_n,\sigma)),\sigma)
\end{array}
$$

4 Further work

- Extend the definition of terms with lambda expressions and quantifier expressions, and extend the algorithms so they can handle them.
- Design an algorithm for rewriting using strategies as defined in [Wee09].
- Extend the rewrite algorithms so they handle evaluation of conditions (as is required in the evaluation of a C-node).
- Extend the rewrite algorithms for rewrite rules with more general left hand sides.
- Collect examples of higher order rewrite systems for testing the algorithms.

References

[vR01] Femke van Raamsdonk. On termination of higher-order rewriting. In Aart Middeldorp, editor, *RTA*, volume 2051 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2001.

[Wee09] "M.J. van Weerdenburg". *Efficient Rewriting Techniques*. PhD thesis, 2009.

[Wul09] "Jeroen van der Wulp". Notes for the design of a reusable higher-order conditional rewriting library, 2009.