Abstract: We present a convenient notation for positive/negative-conditional equations. The idea is to merge rules specifying the same function by using case-, if-, match-, and let-expressions. Based on the presented macro-rule-construct, positive/negative-conditional equational specifications can be written on a higher level. A rewrite system translates the macro-rule-constructs into positive/negative-conditional equations.
1 Introduction

We present a macro-rule-construct for convenient specification with positive/negative-conditional equations as presented in Wirth & Gramlich (1993). Though separate equations building up the definition of one single function are advantageous under several theoretical and practical aspects, this separation does not correspond to the “natural” way of defining functions. As equational specification requires every reduction rule to be defined explicitly, various repetitions of common sub-expressions occur. In specifications with positive/negative-conditional equations, moreover, case distinctions lead to frequent numerous repetitions of only slightly changed left-hand sides and condition lists. This is rather tedious for the specifier and a source of errors. It also hides the actual structure of the specification.

To overcome these problems we introduce a macro-rule-construct for achieving the following aims:

- Concise notation: The specifier should be able to express the sharing of expressions in the specification language instead of having to spread copies of a common sub-expression all over a function’s definition.
- Logical modularization: Reduction rules for the same function should be combined and structured hierarchically.
- Explicit representation of case distinctive structures: The knowledge the specifier has in mind should be made explicit.
- Free choice of specification level: The language should also allow equational specification without using the structural features.

To explain some ideas of our approach we will use the following rules:

\[
\begin{align*}
\text{delete } x \text{ nil} &= \text{nil} \\
\text{delete } x \text{ cons } y \text{ k} &= \text{delete } x \text{ k} \quad \leftarrow x = y \\
\text{delete } x \text{ cons } y \text{ k} &= \text{cons } y \text{ delete } x \text{ k} \quad \leftarrow x \neq y \\
\text{delete } x \text{ l} &= l \quad \leftarrow \text{memberp } x \text{ l} \neq \text{true}
\end{align*}
\]
The main features of our macro-rule-construct are:

- Conditions of equations are written as lists and characteristic functions as predicates. For example the last delete-rule above may be written
  
  \((\text{delete } x \ l) = l \leftarrow ((\text{not } (\text{memberp } x \ l)))\)

- Contraction of right-hand sides and conditions into a new “meta-term”, changing the order of appearance:
  Instead of
  
  \(\text{delete } x \ \text{cons } y \ k = \text{delete } x \ k \leftarrow x = y\)
  
  we write
  
  \((\text{delete } x \ (\text{cons } y \ k)) = (\text{case } ((= x y)) \ (\text{delete } x \ k))\)

- Introduction of match-conditions \((\text{@ VAR TERM})\), which bind the variables in the term \(\text{TERM}\) by a required match from \(\text{TERM}\) to the value of the variable \(\text{VAR}\). This has the advantage that all left-hand sides of equations specifying the same function can be written in the same way.

The rules of our delete-specification can now be written like this:

\[
\begin{align*}
(\text{delete } x \ l) &= \text{case } ((\text{@ l nil})) \ \text{nil} \\
(\text{delete } x \ l) &= \text{case } ((\text{@ l (cons y k)}) \ (\text{delete } x \ k)) \\
(\text{delete } x \ l) &= \text{case } ((\text{@ l (cons y k)}) \ (\text{cons y (delete x k)})) \\
(\text{delete } x \ l) &= \text{case } ((\text{not } (\text{memberp } x \ l))) \ l
\end{align*}
\]

The match-atom \((\text{@ VAR TERM})\) connects the rule’s variable \(\text{VAR}\) with those variables that are introduced by \(\text{TERM}\) and may occur to the right of the match-atom. For avoiding reference problems, the variables in \(\text{TERM}\) must not occur to the left of \((\text{@ VAR TERM})\) in the rule. This restriction can be weakened to apply only to those variables that are not properly influenced by some let- or match-atom. Especially \(\text{VAR}\) may occur in \(\text{TERM}\) and to the left of \((\text{@ VAR TERM})\). E.g. in the above rules, we could replace the \(k\) with \(l\). In this case, the occurrences of \(\text{VAR}\) in the second argument of \((\text{@ VAR TERM})\) have the same meaning as the occurrences of \(\text{VAR}\) to the right of \((\text{@ VAR TERM})\), which is different from the meaning of \(\text{VAR}\) in the first argument of \((\text{@ VAR TERM})\) having the same meaning as the occurrences of \(\text{VAR}\) to the left of \((\text{@ VAR TERM})\). Thus (in case of \(\text{VAR}\) occurring in \(\text{TERM}\)) the borderline of the meaning of \(\text{VAR}\) in the rule goes right through the match-atom.
If, however, VAR does not occur in TERM, then the meaning of VAR to the left and to the right of the match-atom is the same. This persistence of the meaning of VAR can be useful.

As an (not really convincing) example the first delete-rule could be written:

\[
\text{(delete x l) } = \text{(case ((@ l nil)) l)}
\]

- A let-expression (let TERM VAR) may occur in condition lists and introduces VAR as a macro for TERM.

Each of the expressions (\(\emptyset\) VAR\(_2\) TERM\(_1\)) and (let TERM\(_2\) VAR\(_1\)) binds the variables occurring in its second argument (TERM\(_1\), VAR\(_1\), resp.) with the scope being the rest of the rule. If one of these variables is already bound in the context of the expression, then its old binding is lost in the scope of the expression. Since this is a common source for bugs in specifications, the specifier should\(^1\) be warned if such a re-binding occurs. E.g. (let (cons x l) l) re-binds l to the term (cons x l) where l refers to the old binding of l, which is lost for the rest of the rule. Similarly, if cons is the top symbol of l, then (@ l (cons x l)) binds x to the first argument of l and re-binds l to the second argument of the old binding of l, which again is lost for the rest of the rule.

The translation into rules removes an atom (let TERM\(_2\) VAR\(_1\)) by substituting TERM\(_2\) for all occurrences of VAR\(_1\) to the right of the atom. Similarly, an atom (\(\emptyset\) VAR\(_2\) TERM\(_1\)) is removed by substituting TERM\(_1\) for all occurrences of VAR\(_2\) to the left and (unless VAR\(_2\) occurs in TERM\(_1\)) to the right of the atom\(^2\).

- Equations with the same left-hand side are merged:

\[
\text{(macro-rule (delete x l) }
\begin{align*}
\text{(case } \\
\text{ ((@ l nil)) l) } \\
\text{ ((@ l (cons y k)) (del x y)) (del x k)) \\
\text{ ((@ l (cons y k)) (# x y)) (cons y (del x k))})
\end{align*}
\]

\(^1\)For the specifier who really wants to write (@ l (cons x l)) and does not want to be warned all the time, there is another match-atom having the form (@@ VAR\(_2\) TERM\(_1\)). It behaves similar to (@ VAR\(_2\) TERM\(_1\)) but does not warn if VAR\(_2\) occurs in TERM\(_1\), since it un-binds VAR\(_2\) before it binds the variables in TERM\(_1\) via matching TERM\(_1\) to the old binding of VAR\(_2\).

\(^2\)Similarly, an atom (@@ VAR\(_2\) TERM\(_1\)) is removed by substituting TERM\(_1\) for all VAR\(_2\) to the left the atom.
Negatable conditions may be used in the (conjunctive) condition lists of case-with-else- and if-expressions. The two latter cases of the above macro-rule-expression can be combined into:

\[
\begin{align*}
((@ l (cons y k))) \\
(if ((= x y)) \\
\quad (delete x k) \\
\quad (cons y (delete x k)))
\end{align*}
\]

For a condition list of length \( n + 1 \) an “if”-expression saves \( n + 1 \) condition literals and \( n \) repetitions of the meta-term of the else-case in the specification:

\[
\text{(macro-rule } l \\
\quad (if (L_0 \ldots L_n) r_0 \\
\quad \quad r_1))
\]

written in form of unstructured conditional equations is much longer:

\[
\begin{align*}
l & = r_0 \leftarrow L_0 \ldots L_n \\
l & = r_1 \leftarrow \text{not } L_0 \\
l & = r_1 \leftarrow \text{not } L_1 \\
\vdots & \quad \vdots \\
l & = r_1 \leftarrow \text{not } L_n
\end{align*}
\]

For a case-with-else-expression the saving has the complexity of the product of the lengths of the condition lists.

The possibility of nestling case- and if-expressions allows a quadratic saving in the number of condition literals:

\[
\text{(macro-rule } l \\
\quad (if (L_0) r_0 \\
\quad \quad (if (L_1) r_1 \\
\quad \quad \quad \vdots \\
\quad \quad \quad \quad (if (L_n) r_n \\
\quad \quad \quad \quad \quad r_{n+1}) \ldots )))
\]

written in form of unstructured conditional equations is much longer:

\[
\begin{align*}
l & = r_0 \leftarrow L_0 \\
l & = r_1 \leftarrow \text{not } L_0 \ L_1 \\
\vdots & \quad \vdots \\
l & = r_n \leftarrow \text{not } L_0 \ldots \text{not } L_{n-1} \ L_n \\
l & = r_{n+1} \leftarrow \text{not } L_0 \ldots \text{not } L_{n-1} \text{not } L_n
\end{align*}
\]
Propositional logic expressions using “not”, “and”, and “or” may occur in condition lists. For example

\begin{verbatim}
(macro-rule 1
  (if (L_0 \ldots \ L_n) is equivalent to:
    r_0
    r_1))

(macro-rule 1
  (case (L_0 \ldots \ L_n)
    r_0
    ((or (not L_0) \ldots (not L_n)))
    r_1))
\end{verbatim}

Note that the positive/negative-conditional rule system, denoted by an or-conditioned case contains in general more than one conditional equation differing only in the condition part. As we do not provide a certain order between positive/negative-conditional equations it is of no importance in which order the arguments are supplied in the or-expression unless its negation becomes relevant due to an outer “not” or a following else-case. In the denoted rule system the and-expression behaves rather different: As it refers to only one conditional equation, the order of appearance of arguments is preserved in the condition list.

A “sequential” (or* L_1 \ldots L_n) is also placed to the specifiers disposal. This expression guarantees, that all arguments from L_0 to L_{i-1} are not fulfilled when the validity of L_i is checked. To illustrate the difference between or and or* a characteristic function is specified. It tests, whether all elements in a list are equal. Here we assume (\text{car (cons x l)}) = x, (\text{cdr nil}) = \text{nil} (!) and (\text{cdr (cons x l)}) = l. The specification of car need not necessarily be complete.

\begin{verbatim}
(macro-rule (equal-l l)
  (if ((or* (= (cdr l) nil)
        (and (equal-l (cdr l))
             (= (car l)
                (car (cdr l)))))
       true
       false))
\end{verbatim}

The corresponding conditional equations for the true-case are:

\begin{verbatim}
(equal-l l) = true ←− (cdr l) = nil
(equal-l l) = true ←− (cdr l) \neq nil,
    (equal-l (cdr l)) = true,
    (car l) = (car (cdr l))
\end{verbatim}

The condition list of the second equation contains the negated first argument of or* besides the second one. If an or-expression were used in spite of the or* a termination problem would occur because this first negated condition would be removed:

\begin{verbatim}
(equal-l l) = true ←− (equal-l (cdr l)) = true,
    (car l) = (car (cdr l))
\end{verbatim}
As the dual of or*, an and*-expression is also included. The and- and the and*-expression are equivalent with respect to the positive/negative-conditional rules they denote unless its negation becomes relevant due to an outer “not” or a following else-case. The and*-expression should be used whenever the order of appearance of the arguments is relevant.

For an and*-condition with \( n+1 \) arguments the if-expression saves \( (n+1)*(n+2)/2 \) condition literals and \( n \) repetitions of the meta-term of the else-case in the specification:

\[
\text{written in form of unstructured conditional equations is much longer:}
\]

\[
\begin{align*}
l &= r_0 \leftarrow L_0 \ldots L_n \\
l &= r_1 \leftarrow (\text{not } L_0) \\
l &= r_1 \leftarrow L_0 \ (\text{not } L_1) \\
&\vdots \quad \vdots \\
l &= r_1 \leftarrow L_0 \ldots L_{n-1} \ (\text{not } L_n)
\end{align*}
\]

For a case-with-else-expression the saving has the complexity of the product of the squares of the numbers of arguments of the and*-expressions.

We now give a final version of our introducing delete-specification:

\[
\text{(macro-rule (delete x l))}
\]

\[
\begin{align*}
\text{(case}
\quad ((@ l \text{ nil)))} \\
\text{nil} \\
\quad ((@ l (\text{cons y k)))} \\
\quad \text{(let (delete x k) h))} \\
\quad (\text{if } ((= x y))) \\
\quad \text{h} \\
\quad \text{(cons y h))} \\
\quad ((\text{not (memberp x l)))} \\
\text{l))}
\end{align*}
\]

The last case really should be omitted. It is only present to remind the cursory reader that the cases must be neither complementary nor complete and that their ordering is (in contrast to LISP’s COND) relevant only for the order of the tests of an optional else-case of the case-expression.
All in all, this macro-rule-construct was designed as a tool for the specifier. Besides that, it is also useful for explicitly structuring an equational specification. This structuring must be done anyway:

- It reduces the number of matching and condition tests and therefore enhances efficiency of rewriting.

- More important for us is that it may exhibit the recursive construction of a function and therefore may help to find suitable structures for inductive proofs by giving hints for case distinctions and for the choice of covering sets of substitutions:

  For example, the “natural” way of proving inductive properties of the delete-function is to start with a covering set of substitutions given by “{l→nil}” and “{l→(cons y k)}”, and then to make a case distinction for the second case on whether “x=y” holds or not.
2 Examples

In this section we give some more examples.

Two specifications of the characteristic function of the member predicate:

```lisp
(macro-rule (memberp x l)
  (case
    ((@ l nil))
      false
    ((@ l (cons y m)))
      (if ((= x y))
        true
        (memberp x m))))
```

denotes

\[
\begin{align*}
\text{memberp } x \text{ nil} & = \text{false} \\
\text{memberp } x \text{ cons } y \text{ m} & = \text{true} \quad \leftarrow x = y \\
\text{memberp } x \text{ cons } y \text{ m} & = \text{memberp } x \text{ m} \quad \leftarrow x \neq y
\end{align*}
\]

while

```lisp
(macro-rule (memberp x l)
  (case
    ((@ l nil))
      false
    ((@ l (cons y m)))
      (if (((or (= x y) (memberp x m)))
        true
        false)))
```

denotes

\[
\begin{align*}
\text{memberp } x \text{ nil} & = \text{false} \\
\text{memberp } x \text{ cons } y \text{ m} & = \text{true} \quad \leftarrow x = y \\
\text{memberp } x \text{ cons } y \text{ m} & = \text{memberp } x \text{ m} \quad \leftarrow x \neq y, \text{ memberp } x \text{ m} \neq \text{true}.
\end{align*}
\]
Functions on natural numbers:

\[(\text{macro-rule } (p \ x))\]
\[(\text{case})\]
\[(((@ \ x \ (s \ u)))\]
\[u)\]

denotes
\[p \ s \ u = u\]

which is syntactically more restrictive and operationally more useful than

\[(\text{macro-rule } (p \ x))\]
\[(\text{case})\]
\[((= \ x \ (s \ u)))\]
\[u)\]

which denotes
\[p \ x = u \leftarrow x = s \ u.\]

\[(\text{macro-rule } (\text{max} \ x \ y))\]
\[(\text{case})\]
\[(((@ \ x \ 0))\]
\[y\]
\[(((@ \ y \ 0))\]
\[x\]
\[(((@ \ x \ (s \ u))\]
\[(@ \ y \ (s \ v)))\]
\[(s \ (\text{max} \ u \ v)))\]

\[(\text{macro-rule } (+ \ x \ y))\]
\[(\text{case})\]
\[(((@ \ x \ 0))\]
\[y\]
\[(((@ \ x \ (s \ u))\]
\[(s \ (+ \ u \ y))))\]

\[(\text{macro-rule } (* \ x \ y))\]
\[(\text{case})\]
\[(((@ \ x \ 0))\]
\[0\]
\[(((@ \ x \ (s \ u))\]
\[(+ \ y \ (* \ u \ y))))\]

\[(\text{macro-rule } (\text{pot} \ w \ x)) ; \text{computes } w^x\]
\[(\text{case})\]
\[(((@ \ x \ 0))\]
\[(s \ 0)\]
\[(((@ \ x \ (s \ u))\]
\[(* \ w \ (\text{pot} \ w \ u))))\]
Functions on binary trees:

(macro-rule (hight t)
  (case
    ((@ t nil))
    0
    ((@ t (mk-tree l node r)))
    (s (max (hight l)
            (hight r)))))

(macro-rule (count-nodes t)
  (case
    ((@ t nil))
    0
    ((@ t (mk-tree l node r)))
    (s (+ (count-nodes l)
          (count-nodes r))))

(macro-rule (completep t)
  (case
    ((@ t nil))
    true
    ((@ t (mk-tree r node l)))
    (if ((= (hight l) (hight r))); this is a conjunctive condition
        (completep l) ; list, just like with equational
        (completep r)) ; rules
      true
      false)))
3 Syntax

The syntax of the macro-rule-construct is defined by the following context-free grammar with starting symbol `<macro-rule>`. Note that the sets of variable, constant, and function names must be mutually disjoint. Furthermore, function names must be different from “case”, and “if” and should also be different from “=”, “#”, “def”, “@”, “@@”, “let”, “or”, “or*”, “and”, “and*”, and “not”.

```
<term> ::= <variable-name>
| <constant-name>
| (<function-name> <term>+)

<(in-)equality-atom> ::= (= <term> <term>)
| (# <term> <term>)

<predicate-atom> ::= <term>

<negatable-atom> ::= <(in-)equality-atom>
| <predicate-atom>

<def-atom> ::= (def <term>)

<basic-atom> ::= <negatable-atom>
| <def-atom>

<match-atom> ::= (@ <variable-name> <term>)
| (@@ <variable-name> <term>)

<let-atom> ::= (let <term> <variable-name>)

<negatable-condition> ::= <negatable-atom>
| (and <negatable-condition>*)
| (or <negatable-condition>*)
| (and* <negatable-condition>*)
| (or* <negatable-condition>*)
| (not <negatable-condition>)
```

3Here, “...*” denotes zero or more repetitions, “...+” denotes one or more repetitions, “...|...” denotes different possibilities, “<...>” denotes non-terminals, and typewriter font indicates grammar terminals.

4This is necessary if the function is specified as characteristic function and used in a predicate-atom.
\[\text{general-condition} := \text{negatible-condition} \]
\[| \text{basic-atom} \]
\[| \text{match-atom} \]
\[| \text{let-atom} \]
\[| (\text{and} \text{general-condition}^*) \]
\[| (\text{or} \text{general-condition}^*) \]
\[| (\text{and}^* \text{negatible-condition}^* \text{general-condition}) \]
\[| (\text{or}^* \text{negatible-condition}^* \text{general-condition}) \]

\[\text{negatible-condition-list} := (\text{negatible-condition}^*) \]
\[\text{general-condition-list} := (\text{general-condition}^*) \]

\[\text{negatible-case} := \text{negatible-condition-list} \]
\[\text{meta-term} \]
\[\text{else} := \text{else} \]
\[\text{meta-term} \]
\[\text{case} := \text{general-condition-list} \]
\[\text{meta-term} \]

\[\text{if-term} := (\text{if} \text{negatible-condition-list} \]
\[\text{meta-term} \]
\[\text{meta-term}) \]
\[\text{case-term-with-else} := (\text{case} \]
\[\text{negatible-case}^* \]
\[\text{else}) \]
\[\text{case-term} := (\text{case} \]
\[\text{case}^+) \]

\[\text{meta-term} := \text{term} \]
\[| \text{if-term} \]
\[| \text{case-term-with-else} \]
\[| \text{case-term} \]

\[\text{macro-rule} := (\text{macro-rule} \text{term} \text{meta-term}) \]
4 Semantics

The semantics of a sequence of macro-rule-expressions is a positive/negative-conditional rule system.

Let:

- $\text{VAR}_i \in L_G(<\text{variable-name}>)$\footnote{$L_G(<\text{sym}>)$ denotes the set of words generated by productions of our grammar starting from the symbol $<\text{sym}>$.}
- $\text{TERM}_i \in L_G(<\text{term}>)$
- $\text{PRED-ATOM}_i \in L_G(<\text{predicate-atom}>)$
- $\text{N-C}_i \in L_G(<\text{negatible-condition}>)$
- $\text{N-C-LIST}_i \in L_G(<\text{negatible-condition-list}>)$
- $\text{BASIC-ATOM}_i \in L_G(<\text{basic-atom}>)$
- $\text{MATCH}_i \in L_G(<\text{match-atom}>)$
- $\text{LET}_i \in L_G(<\text{let-atom}>)$
- $\text{GEN-COND}_i, \text{G-C}_i \in L_G(<\text{general-condition}>)$
- $\text{CASE}_i \in L_G(<\text{case}>)$
- $\text{META-TERM}_i \in L_G(<\text{meta-term}>)$

The denotation of the following “elementary” macro-rule-expressions is defined as follows:

$(\text{macro-rule } \text{TERM}_1 \text{ TERM}_2)$
denotes the unconditional rewrite rule

$\text{TERM}_1 = \text{TERM}_2$

and

$(\text{macro-rule } \text{TERM}_1$

(case

$(\text{BASIC-ATOM}_0 \cdots \text{BASIC-ATOM}_n)$

$\text{TERM}_2)$

$\text{TERM}_1 = \text{TERM}_2 \leftarrow \text{BASIC-ATOM}_0, \ldots, \text{BASIC-ATOM}_n$

A macro-rule-expression is non-erroneous iff it can be transformed into elementary macro-rule-expressions with the rewrite rules we will introduce in this section. Note that the semantics is declarative in so far as no precedence is imposed on the application of these rules. The resulting rewriting relation is confluent and Noetherian. Since all elementary macro-rule-expressions are irreducible, each macro-rule-expression denotes at most one positive/negative-conditional rule system.
“Predicate”-Removal
Predicates may be used as conditions. All these predicates are put into equations:

In the context of a general or negatable condition:
\[ \text{PRED-ATOM} \rightarrow (\ = \text{PRED-ATOM true}) \]

“if”-Removal
if-expressions are replaced by “case-with-else”-expressions:

\[
\begin{align*}
(\text{if} & \ N-C-\text{LIST} \\
\quad & \text{META-TERM}_1 \\
\quad & \text{META-TERM}_2) \\
\rightarrow & (\text{case} \\
\quad & N-C-\text{LIST} \\
\quad & \text{META-TERM}_1 \\
\quad & \text{else} \\
\quad & \text{META-TERM}_2)
\end{align*}
\]

“else”-Removal
As else-statements may cause trouble when replacing “case in case” (cf. below), they must be eliminated before:

\[
\begin{align*}
(\text{case} \\
\quad & (N-C_{1,1} \cdots N-C_{1,n_1}) \\
\quad & \vdots \\
\quad & (N-C_{m,1} \cdots N-C_{m,n_m}) \\
\quad & \text{else} \\
\quad & \text{META-TERM}_{m+1})
\end{align*}
\]

\[
\begin{align*}
\downarrow
(\text{case} \\
\quad & (N-C_{1,1} \cdots N-C_{1,n_1}) \\
\quad & \vdots \\
\quad & (N-C_{m,1} \cdots N-C_{m,n_m}) \\
\quad & ((\text{or} \ (\not N-C_{1,1}) \\
\quad & \quad \ (\not N-C_{1,2}) \\
\quad & \quad \vdots \\
\quad & \quad (\not N-C_{1,n_1}) \\
\quad & \quad \vdots) \\
\quad & (\text{or} \ (\not N-C_{m,1}) \\
\quad & \quad (\not N-C_{m,2}) \\
\quad & \quad \vdots \\
\quad & \quad (\not N-C_{m,n_m})) \\
\quad & \text{META-TERM}_{m+1})
\end{align*}
\]

If none of the preceding rewrite rules applies anymore, then all negatable atoms are (in-)equality atoms and no if- or else-expressions occur in the specification.
“not”-Removal

(not (not N-C)) → N-C
(not (and N-C₁ ⋯ N-Cₙ)) → (or (not N-C₁) ⋯ (not N-Cₙ))
(not (and* N-C₁ ⋯ N-Cₙ)) → (or* (not N-C₁) ⋯ (not N-Cₙ))
(not (or N-C₁ ⋯ N-Cₙ)) → (and (not N-C₁) ⋯ (not N-Cₙ))
(not (or* N-C₁ ⋯ N-Cₙ)) → (and* (not N-C₁) ⋯ (not N-Cₙ))
(not (= TERM₁ TERM₂)) → (# TERM₁ TERM₂)
(not (# TERM₁ TERM₂)) → (= TERM₁ TERM₂)

“or”-Removal

(case
  CASE₁
  ...
  CASEₙ
  (G-C₁ ⋯ G-Cₚ
   (or GEN-COND₁ ⋯ GEN-CONDₙ)
   G-Cₚ₊₁ ⋯ G-Cₚ₊ᵣ)
META-TERM
CASEₙ₊₁
  ...
  CASEₙ₊ₘ)
↓
(case
  CASE₁
  ...
  CASEₙ
  (G-C₁ ⋯ G-Cₚ GEN-COND₁ G-Cₚ₊₁ ⋯ G-Cₚ₊ᵣ ) META-TERM
  ...
  (G-C₁ ⋯ G-Cₚ GEN-CONDₙ G-Cₚ₊₁ ⋯ G-Cₚ₊ᵣ ) META-TERM
  CASEₙ₊₁
  ...
  CASEₙ₊ₘ)

Note that for application of this rule no ‘else’ may occur in the case-expression.
“or*”-Removal

\[
\text{(case} \\
\text{CASE}_1 \\
\vdots \\
\text{CASE}_n \\
\text{(G-C}_1 \ldots \text{G-C}_p \\
\text{(or* N-C}_1 \ldots \text{N-C}_r \\
\text{G-C}_{p+1} \ldots \text{G-C}_{p+q}) \\
\text{META-TERM} \\
\text{CASE}_{n+1} \\
\vdots \\
\text{CASE}_{n+m})
\]

\[
\downarrow
\]

\[
\text{(case} \\
\text{CASE}_1 \\
\vdots \\
\text{CASE}_n \\
\text{(G-C}_1 \ldots \text{G-C}_p \\
\text{N-C}_1 \\
\text{G-C}_{p+1} \ldots \text{G-C}_{p+q}) \\
\text{META-TERM} \\
\text{(G-C}_1 \ldots \text{G-C}_p \\
\text{(not N-C}_1) \text{N-C}_2 \\
\text{G-C}_{p+1} \ldots \text{G-C}_{p+q}) \\
\text{META-TERM} \\
\vdots \\
\text{(G-C}_1 \ldots \text{G-C}_p \\
\text{(not N-C}_1) \ldots \text{(not N-C}_{r-1}) \text{N-C}_r \\
\text{G-C}_{p+1} \ldots \text{G-C}_{p+q}) \\
\text{META-TERM} \\
\text{CASE}_{n+1} \\
\vdots \\
\text{CASE}_{n+m})
\]

Note that for application of this rule no ‘else’ may occur in the case-expression.
“and[*]”-Removal

\[
\text{(case} \quad \text{CASE}_1 \\
\vdots \\
\text{CASE}_n \\
(G-C_1 \cdots G-C_p \\
(\text{and}[*] \ \text{GEN-COND}_1 \cdots \text{GEN-COND}_r) \quad \rightarrow \\
G-C_{p+1} \cdots G-C_{p+q}) \\
\text{META-TERM} \quad \text{META-TERM} \\
\text{CASE}_{n+1} \\
\vdots \\
\text{CASE}_{n+m})
\]

Note that for application of this rule no ‘else’ may occur in the case-expression.

“\text{case-in-case}”-Removal

\[
\text{(case} \\
\text{CASE}_1 \\
\vdots \\
\text{CASE}_m \\
(G-C_1 \cdots G-C_p) \\
(\text{case} \\
\text{(GEN-COND}_{1,1} \cdots \text{GEN-COND}_{1,q_1}) \\
\vdots \\
\text{(GEN-COND}_{r,1} \cdots \text{GEN-COND}_{r,q_r}) \\
\text{META-TERM}_1 \\
\vdots \\
\text{META-TERM}_r \\
\text{CASE}_{m+1} \\
\vdots \\
\text{CASE}_{m+n})
\]

¶

\[
\text{(case} \\
\text{CASE}_1 \\
\vdots \\
\text{CASE}_m \\
(G-C_1 \cdots G-C_p \ \text{GEN-COND}_{1,1} \cdots \text{GEN-COND}_{1,q_1}) \ \text{META-TERM}_1 \\
\vdots \\
\text{(GEN-COND}_{r,1} \cdots \text{GEN-COND}_{r,q_r}) \ \text{META-TERM}_r \\
\text{CASE}_{m+1} \\
\vdots \\
\text{CASE}_{m+n})
\]

Note that for application of this rule no ‘else’ may occur in any of the two case-expressions.
"@"-Removal

As we do not want match-atoms in our final rule-system we replace all occurrences of a match-variable \( \text{VAR} \) preceding a match-atom \((@ \ \text{VAR} \ \text{TERM})\) with the match-term \(\text{TERM}\). If the match-variable does not occur in the match-term, we also have to replace all occurrences of the match-variable in the scope of the match-atom with the match-term. Let \(\mathcal{V} (\text{TERM})\) denote the set of variables occurring in \(\text{TERM}\).

If \( \text{VAR} \in \mathcal{V} (\text{TERM}) \), then the specifier should be warned like:

"WARNING: \((@ \ \text{VAR} \ \text{TERM})\) re-binds \text{VAR}"

and we reduce:

\[
(@ \ \text{VAR} \ \text{TERM}) \rightarrow (@@ \ \text{VAR} \ \text{TERM})
\]

Otherwise we reduce:

\[
(@ \ \text{VAR} \ \text{TERM}) \rightarrow (@@ \ \text{VAR} \ \text{TERM}) \\
(\text{let} \ \text{TERM} \ \text{VAR})
\]

"@@"-Shift-Left

In case of \( \mathcal{V} (\text{BASIC-ATOM}) \cap (\mathcal{V} (\text{TERM}) \setminus \{\text{VAR}\}) \neq \emptyset \) the condition list below is erroneous.

Otherwise we reduce:

\[
\begin{array}{c}
(G-\text{C}_1 \\
\vdots \\
G-\text{C}_m \\
\text{BASIC-ATOM} \\
@@ \ \text{VAR} \ \text{TERM} \\
G-\text{C}_{m+1} \\
\vdots \\
G-\text{C}_{m+n})
\end{array} \rightarrow \begin{array}{c}
(G-\text{C}_1 \\
\vdots \\
G-\text{C}_m \\
@@ \ \text{VAR} \ \text{TERM} \\
\text{BASIC-ATOM}\{\text{VAR} \mapsto \text{TERM}\} \\
G-\text{C}_{m+1} \\
\vdots \\
G-\text{C}_{m+n})
\end{array}
\]

"let"-Shift-Right

If \( \text{VAR} \in \mathcal{V} (\text{TERM}) \), then the specifier should be warned like:

"WARNING: \((\text{let} \ \text{TERM} \ \text{VAR})\) re-binds \text{VAR}"

The following inference rule is the dual of "@@"-shift-left.

\[
\begin{array}{c}
(G-\text{C}_1 \\
\vdots \\
G-\text{C}_m \\
(\text{let} \ \text{TERM} \ \text{VAR}) \\
\text{BASIC-ATOM} \\
G-\text{C}_{m+1} \\
\vdots \\
G-\text{C}_{m+n})
\end{array} \rightarrow \begin{array}{c}
(G-\text{C}_1 \\
\vdots \\
G-\text{C}_m \\
\text{BASIC-ATOM}\{\text{VAR} \mapsto \text{TERM}\} \\
(\text{let} \ \text{TERM} \ \text{VAR}) \\
G-\text{C}_{m+1} \\
\vdots \\
G-\text{C}_{m+n})
\end{array}
\]
“let”-“@@”-Swap

This is the only non-trivial rewrite rule.

\[
\begin{align*}
(G-C_1) & \\
\vdots & \\
G-C_m & \\
(\text{let TERM}_1 \ VAR_1) & \quad \rightarrow \\
(\text{@@ VAR}_2 \ TERM_2) & \\
G-C_{m+1} & \\
\vdots & \\
G-C_{m+n}) &
\end{align*}
\]

with \(<X>\) defined as follows:

\(\text{VAR}_1 = \text{VAR}_2: \text{ERROR.}\)

There is no reasonable semantics for this unless \(\text{TERM}_1 \sigma = \text{TERM}_2 \xi \sigma\) for some \(\xi\) replacing the variables of \(\mathcal{V}(\text{TERM}_1) \cap \mathcal{V}(\text{TERM}_2)\) with new distinct variables and \(\sigma\) being a most general unifier for \(\text{TERM}_1\) and \(\text{TERM}_2\). This case, however, is too unlikely and not important enough to give semantics for, since this would make single pass error checking more difficult.

\(\text{VAR}_1 \in \mathcal{V}(\text{TERM}_2) \setminus \{\text{VAR}_2\}:\)

\begin{align*}
<X> &= (\text{@@ VAR}_2 \ TERM_2) \\
\text{The let-term is removed since \text{VAR}_1 is re-bound by the match-atom. Often, this will not be the intention of the specifier. Therefore a warning should be given.}
\end{align*}

\(\text{VAR}_1 \notin \{\text{VAR}_2\} \cup \mathcal{V}(\text{TERM}_2):\)

\begin{align*}
<X> &= (\text{@@ VAR}_2 \ TERM_2) \\
(\text{let TERM}_1\{\text{VAR}_2 \mapsto \text{TERM}_2\} \ VAR_1) \\
\text{This should be the normal case.}
\end{align*}

Note that errors and warnings (case one and two) can be detected easily by one single pass over the specification before starting the rewriting. This allows error and warning messages to refer to the original macro-rule-constructs, which is necessary for being understandable for the specifier.

\[<X> = (\text{@@} \mathcal{V}(\text{TERM}_2) | \sigma) (\text{let}^* (\mathcal{V}(\text{TERM}_2))((\sigma))^{-1})\] would correspond to our intention. E.g. for

\[
\begin{align*}
(\text{let} (\text{mt l y l}) k) \\
(\text{@@ k (mt h}_1 (\text{cons} y m) h_2))
\end{align*}
\]

we would choose \(\xi := \{y \mapsto z\}; \sigma := \{y \mapsto (\text{cons} z m), h_1 \mapsto l, h_2 \mapsto l\}\) and get

\[
<Z> = (\text{@@} y (\text{cons} z m)) (\text{let} l h_1) (\text{let} l h_2) (\text{let} z y).
\]

However, this definition would destroy the confluence of \(\rightarrow\). E.g. consider the following condition-list where \(y\) is an alias for \(u\):

\[
\begin{align*}
\{& (\text{@@} x (s u)) (\text{@@} x (s y)) \} \rightarrow \\
& ((\text{@@} x (s u)) (\text{let} (s u) x) (\text{@@} x (s y)) (\text{let} (s y) x)).
\end{align*}
\]

The latter condition-list reduces in two ways. First with \(\xi := \{\}, \sigma := \{u \mapsto y\}:\)

\[
\rightarrow ((\text{@@} x (s u)) (\text{@@} u y) (\text{let} (s y) x)).
\]

Second with \(\xi := \{\}, \sigma := \{y \mapsto u\}:\)

\[
\rightarrow ((\text{@@} x (s u)) (\text{let} u y) (\text{let} (s y) x)).
\]

Now the first version reports an error if \(y\) occurs to the left while the second does not. Furthermore, the first version will use the variable \(y\) in its scope while the second will use \(u\) instead.
Splitting

\[
\begin{align*}
\text{(macro-rule T}_1\text{ )} & \quad \text{(macro-rule T}_2\text{ )} \\
\text{(case } & \quad \text{(case } C_1\text{ )} \\
C_1 & \quad \vdots \\
C_n \text{ )} & \quad \text{(case } C_n\text{ )} \\
\end{align*}
\]

By application of the inference rules introduced above, all non-erroneous macro-rule-expressions can be transformed into the following form:

\[
\begin{align*}
\text{(macro-rule T}_1\text{ )} \\
\text{(case } & \quad \text{(case } \text{ MATCH}_1 \cdots \text{ MATCH}_m \\
\text{BASIC-ATOM}_1 \cdots \text{ BASIC-ATOM}_p \\
\text{LET}_1 \cdots \text{ LET}_n \text{ )} \\
\text{T}_2 \text{ )} & \quad \text{(case } \text{ META-TERM} \text{ )}) \\
\text{or} & \quad \text{or} \\
\text{\vdots} & \quad \text{\vdots} \\
\end{align*}
\]

The transformation into an elementary macro-rule-expression is attained by the last three rules.

“@@”-removal

In case of \( \mathcal{V}(T_1) \cap (\mathcal{V}(T_2) \setminus \{V\}) \neq \emptyset \) the specification is erroneous. Otherwise we reduce:

\[
\begin{align*}
\text{(macro-rule T}_1\text{ )} & \quad \text{(macro-rule T}_1\{V\leftarrow T_2\}\{\{V\}\}) \\
\text{(case } & \quad \text{(case } \text{ MATCH}_1 \cdots \text{ MATCH}_m \\
\text{G-C}_1 \cdots \text{ G-C}_m \text{ )} \\
\text{META-TERM} \text{ )}) \\
\end{align*}
\]

↓

\[
\begin{align*}
\text{(macro-rule T}_1\{V\leftarrow T_2\}\{\{V\}\}) & \quad \text{(macro-rule T}_1\text{ )} \\
\text{(case } & \quad \text{(case } \text{ G-C}_1 \cdots \text{ G-C}_m \text{ )} \\
\text{META-TERM} \text{ )}) \\
\end{align*}
\]
“let”-removal

(case
  CASE_1
  ...
  CASE_m
  (G-C_1 \cdots G-C_m \ (\text{let} \ \text{TERM}_1 \ \text{VAR})) \ \text{TERM}_2
  CASE_{m+1}
  ...
  CASE_{m+n})

↓

(case
  CASE_1
  ...
  CASE_m
  (G-C_1 \cdots G-C_m) \ \text{TERM}_2\{\text{VAR}\rightarrow\text{TERM}_1\}
  CASE_{m+1}
  ...
  CASE_{m+n})

“case-with-empty-condition”-Removal

(macro-rule \text{TERM}_1 \ (case () \ \text{TERM}_2)) \rightarrow (macro-rule \text{TERM}_1 \ \text{TERM}_2)

References

Claus-Peter Wirth, Bernhard Gramlich (1993). A Constructor-Based Approach for Positive/Negative-Conditional Equational Specifications. 3rd CTRS 1992, LNCS 656, pp. 198–212, Springer. Revised and extended version in J. Symbolic Computation (1994) 17, pp. 51–90, Academic Press (Elsevier).