Writing Positive/Negative-Conditional Equations Conveniently

Claus-Peter Wirth, Rüdiger Lunde

Searchable Online Edition
December 22, 1994

SEKI-WORKING-PAPER SWP–94–04 (SFB)

Fachbereich Informatik,
Universitæt Kaiserslautern,
D–67653 Kaiserslautern

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.

Contents

This research was supported by the Deutsche Forschungsgemeinschaft, SFB 314 (D4-Projekt)

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:

```
delete x nil      = nil
delete x cons y k = delete x k         ⟵  x = y
delete x cons y k = cons y delete x k  ⟵  x ≠ y
delete x l        = l                  ⟵  memberp x l ≠ true
```

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

  ```
  (delete x l) = l  ⟵  ((not (memberp x l)))
  ```

- Contraction of right-hand sides and conditions into a new “meta-term”, changing the order of appearance:

  Instead of

  ```
   delete x cons y k = delete x k  ⟵  x = y
  ```

  we write

  ```
   (delete x (cons y k)) = (case  ((= x y))  (delete x k))
  ```

- Introduction of match-conditions `(@ VAR TERM)`, which bind the variables in the term `TERM` by a required match from `TERM` to the value of the variable `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:

  ```
  (delete x l)  =  (case ((@ l nil))          nil                  )

  (delete x l)  =  (case ((@ l (cons y k))
                         (= x y))             (delete x k)         )

  (delete x l)  =  (case ((@ l (cons y k))
                         (# x y))             (cons y (delete x k)))

  (delete x l)  =  (case ((not (memberp x l)))  l                    )
  ```

  The match-atom `(@ VAR TERM)` connects the rule’s variable `VAR` with those variables that are introduced by `TERM` and may occur to the right of the match-atom. For avoiding reference problems, the variables in `TERM` must not occur to the left of `(@ 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 `VAR` may occur in `TERM` and to the left of `(@ VAR TERM)`. E.g. in the above rules, we could replace the `k` with `l`. In this case, the occurrences of `VAR` in the second argument of `(@ VAR TERM)` have the same meaning as the occurrences of `VAR` to the right of `(@ VAR TERM)`, which is different from the meaning of `VAR` in the first argument of `(@ VAR TERM)` having the same meaning as the occurrences of `VAR` to the left of `(@ VAR TERM)`. Thus (in case of `VAR` occurring in `TERM`) the borderline of the meaning of `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:

```
(delete x l)  =  (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 $\texttt{(@ VAR}_2\ \texttt{TERM}_1\texttt{)}$ and $\texttt{(let TERM}_2\ \texttt{VAR}_1\texttt{)}$ binds the variables occurring in its second argument ($\texttt{TERM}_1$, $\texttt{VAR}_1$, 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 $\texttt{(let TERM}_2\ \texttt{VAR}_1\texttt{)}$ by substituting $\texttt{TERM}_2$ for all occurrences of $\texttt{VAR}_1$ to the right of the atom. Similarly, an atom $\texttt{(@ VAR}_2\ \texttt{TERM}_1\texttt{)}$ is removed by substituting $\texttt{TERM}_1$ for all occurrences of $\texttt{VAR}_2$ to the left and (unless $\texttt{VAR}_2$ occurs in $\texttt{TERM}_1$) to the right of the atom.[2]

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

```
(macro-rule (delete x l)
    (case

       ((@ l nil))
       nil

       ((@ l (cons y k))
        (= x y))
       (delete x k)

       ((@ l (cons y k))
        (# x y))
       (cons y (delete x k))))
```

[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 $\texttt{(@@ VAR}_2\ \texttt{TERM}_1\texttt{)}$. It behaves similar to $\texttt{(@ VAR}_2\ \texttt{TERM}_1\texttt{)}$ but does not warn if $\texttt{VAR}_2$ occurs in $\texttt{TERM}_1$, since it un-binds $\texttt{VAR}_2$ before it binds the variables in $\texttt{TERM}_1$ via matching $\texttt{TERM}_1$ to the old binding of $\texttt{VAR}_2$.

[2] Similarly, an atom $\texttt{(@@ VAR}_2\ \texttt{TERM}_1\texttt{)}$ is removed by substituting $\texttt{TERM}_1$ for all $\texttt{VAR}_2$ to the left the atom.

- Negatible 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:

  ```
  ⋮
  ((@ l (cons y k)))
  (if ((= x y))
     (delete x k)
     (cons y (delete x k)))
  ⋮
  ```

  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:

  ```
  (macro-rule l
     (if (L$_0$ ... L$_n$)
        r$_0$
        r$_1$))
  ```

  written in form of unstructured conditional equations is much longer:

  ```
  l  =  r$_0$  ⟵  L$_0$ ... L$_n$
  l  =  r$_1$  ⟵  (not L$_0$)
  l  =  r$_1$  ⟵  (not L$_1$)
  ⋮     ⋮         ⋮
  l  =  r$_1$  ⟵  (not L$_n$)
  ```

  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:

  ```
  (macro-rule l
     (if (L$_0$) r$_0$
     (if (L$_1$) r$_1$
     ⋮        ⋮
     (if (L$_n$) r$_n$
              r$_{n+1}$)...)))
  ```

  written in form of unstructured conditional equations is much longer:

  ```
  l  =  r$_0$      ⟵  L$_0$
  l  =  r$_1$      ⟵  (not L$_0$) L$_1$
  ⋮     ⋮             ⋮
  l  =  r$_n$      ⟵  (not L$_0$) ... (not L$_{n-1}$) L$_n$
  l  =  r$_{n+1}$  ⟵  (not L$_0$) ... (not L$_{n-1}$) (not L$_n$)
  ```

- Propositional logic expressions using “not”, “and”, and “or” may occur in condition lists. For example

```
(macro-rule l
   (if (L_0 ... L_n)
      r_0
      r_1))
```

is equivalent to:

```
(macro-rule l
  (case
     (L_0 ... L_n)
     r_0
     ((or (not L_0) ... (not L_n)))
     r_1))
```

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 ... 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 (car (cons x l)) = x, (cdr nil) = nil (!) and (cdr (cons x l)) = l. The specification of car need not necessarily be complete.

```
(macro-rule (equal-l l)
   (if ((or* (= (cdr l) nil)
             (and (equal-l (cdr l))
                  (= (car l)
                     (car (cdr l))))))
       true
       false))
```

The corresponding conditional equations for the true-case are:

```
(equal-l l)  =  true  ⟵  (cdr l) = nil
(equal-l l)  =  true  ⟵  (cdr l) ≠ nil,
                          (equal-l (cdr l)) = true,
                          (car l) = (car (cdr l))
```

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:

```
(equal-l l)  =  true  ⟵  (equal-l (cdr l)) = true,
                          (car l) = (car (cdr l))
```

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:

```
(macro-rule l
   (if ((and* $L_0$ ... $L_n$))
      $r_0$
      $r_1$))
```

written in form of unstructured conditional equations is much longer:

$$\begin{array}{lllll}
\mathtt{l} & = & \mathtt{r}_0 & \longleftarrow & \mathtt{L}_0\ \mathtt{...}\ \mathtt{L}_n \\
\mathtt{l} & = & \mathtt{r}_1 & \longleftarrow & \mathtt{(not}\ \mathtt{L}_0\mathtt{)} \\
\mathtt{l} & = & \mathtt{r}_1 & \longleftarrow & \mathtt{L}_0\ \mathtt{(not}\ \mathtt{L}_1\mathtt{)} \\
\vdots & & \vdots & & \vdots \\
\mathtt{l} & = & \mathtt{r}_1 & \longleftarrow & \mathtt{L}_0\ \mathtt{...}\ \mathtt{L}_{n-1}\ \mathtt{(not}\ \mathtt{L}_n\mathtt{)}
\end{array}$$

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:

```
(macro-rule (delete x l)
  (case

     ((@ l nil))
     nil

     ((@ l (cons y k))
      (let (delete x k) h))
     (if ((= x y))
        h
        (cons y h))

     ((not (memberp x l)))
     l))
```

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`$\mapsto$`nil`$\}$" and "$\{$`l`$\mapsto$`(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:

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

denotes

```
memberp x nil       =  false
memberp x cons y m  =  true         ⟵  x = y
memberp x cons y m  =  memberp x m  ⟵  x ≠ y
```

while

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

denotes

```
memberp x nil       =  false
memberp x cons y m  =  true   ⟵  x = y
memberp x cons y m  =  true   ⟵  memberp x m = true
memberp x cons y m  =  false  ⟵  x ≠ y,  memberp x m ≠ true .
```

Functions on natural numbers:

```
(macro-rule (p x)
  (case
     ((@ x (s u)))
     u))
```

denotes

```
  p s u  =  u
```

which is syntactically more restrictive and operationally more useful than

```
(macro-rule (p x)
  (case
     ((= x (s u)))
     u))
```

which denotes

```
  p x  =  u  ⟵  x = s u .
```

```
(macro-rule (max x y)
  (case
     ((@ x 0))
     y
     ((@ y 0))
     x
     ((@ x (s u))
      (@ y (s v)))
     (s (max u v))))

(macro-rule (+ x y)
  (case
     ((@ x 0))
     y
     ((@ x (s u)))
     (s (+ u y))))

(macro-rule (* x y)
  (case
     ((@ x 0))
     0
     ((@ x (s u)))
     (+ y (* u y))))

(macro-rule (pot w x)  ; computes w^X
  (case
     ((@ x 0))
     (s 0)
     ((@ x (s u)))
     (* w
        (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[3] 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[4] 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>
      <negatible-atom>  :=  <(in-)equality-atom>
                         |  <predicate-atom>
            <def-atom>  :=  (def <term>)
          <basic-atom>  :=  <negatible-atom>
                         |  <def-atom>
          <match-atom>  :=  (@   <variable-name> <term>)
                         |  (@@  <variable-name> <term>)
            <let-atom>  :=  (let <term> <variable-name>)

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

[3] Here, " ...* " denotes zero or more repetitions, " ...$^{+}$ " denotes one or more repetitions, "...|..." denotes different possibilities, "<...>" denotes non-terminals, and typewriter font indicates grammar terminals.

[4] This is necessary if the function is specified as characteristic function and used in a predicate-atom.

```
           <general-condition>  :=  <negatible-condition>
                                 |  <basic-atom>
                                 |  <match-atom>
                                 |  <let-atom>
                                 |  (and   <general-condition>*)
                                 |  (or    <general-condition>*)
                                 |  (and*  <negatible-condition>* <general-condition>)
                                 |  (or*   <negatible-condition>* <general-condition>)
  <negatible-condition-list>  :=  (<negatible-condition>*)
     <general-condition-list>  :=  (<general-condition>*)


            <negatible-case>  :=  <negatible-condition-list>
                                    <meta-term>
                        <else>  :=  else
                                    <meta-term>
                        <case>  :=  <general-condition-list>
                                    <meta-term>


                     <if-term>  :=  (if  <negatible-condition-list>
                                      <meta-term>
                                      <meta-term>)
        <case-term-with-else>  :=  (case
                                      <negatible-case>*
                                      <else>)
                   <case-term>  :=  (case
                                      <case>+)


                   <meta-term>  :=  <term>
                                 |  <if-term>
                                 |  <case-term-with-else>
                                 |  <case-term>


                  <macro-rule>  :=  (macro-rule <term> <meta-term>)
```

4 Semantics

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

Let:

VAR_i	$\in$	L_G(<variable-name>)[5]
TERM_i	$\in$	L_G(<term>)
$\mathrm{PRED\text{-}ATOM}_i$	$\in$	L_G(<predicate-atom>)
$\mathrm{N\text{-}C}_i$	$\in$	L_G(<negatible-condition>)
$\mathrm{N\text{-}C\text{-}LIST}_i$	$\in$	L_G(<negatible-condition-list>)
$\mathrm{BASIC\text{-}ATOM}_i$	$\in$	L_G(<basic-atom>)
MATCH_i	$\in$	L_G(<match-atom>)
LET_i	$\in$	L_G(<let-atom>)
$\mathrm{GEN\text{-}COND}_i$, $\mathrm{G\text{-}C}_i$	$\in$	L_G(<general-condition>)
CASE_i	$\in$	L_G(<case>)
$\mathrm{META\text{-}TERM}_i$	$\in$	L_G(<meta-term>)

The denotation of the following "*elementary*" `macro-rule`-expressions is defined as follows:

```
(macro-rule TERM$_1$ TERM$_2$)
```

denotes the unconditional rewrite rule

$$\mathrm{TERM}_1 \quad = \quad \mathrm{TERM}_2$$

and

```
(macro-rule TERM$_1$
   (case
      (BASIC-ATOM$_0$ $\cdots$ BASIC-ATOM$_n$)
      TERM$_2$))
```

denotes the following rewrite rule with nonempty condition

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

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.

[5] L_G(<sym>) denotes the set of words generated by productions of our grammar starting from the symbol <sym>.

“Predicate”-Removal

Predicates may be used as conditions. All these predicates are put into equations:

In the context of a general or negatible condition:

```
PRED-ATOM ⟶ (= PRED-ATOM true)
```

“if”-Removal

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

```
(if N-C-LIST                        (case
  META-TERM$_1$            ⟶           N-C-LIST
  META-TERM$_2$)                         META-TERM$_1$
                                       else
                                       META-TERM$_2$)
```

“else”-Removal

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

```
(case
  (N-C$_{1,1}$ ··· N-C$_{1,n_1}$ )                    META-TERM$_1$
  ⋮                                          ⋮
  (N-C$_{m,1}$ ··· N-C$_{m,n_m}$)                    META-TERM$_m$
  else                                       META-TERM$_{m+1}$)

↓

(case
  (N-C$_{1,1}$ ··· N-C$_{1,n_1}$ )                    META-TERM$_1$
  ⋮                                          ⋮
  (N-C$_{m,1}$ ··· N-C$_{m,n_m}$)                    META-TERM$_m$
  ((or (not N-C$_{1,1}$)
   .   (not N-C$_{1,2}$)
   .   ⋮
   .   (not N-C$_{1,n_1}$))
   ⋮
   (or (not N-C$_{m,1}$)
       (not N-C$_{m,2}$)
       ⋮
       (not N-C$_{m,n_m}$)))                    META-TERM$_{m+1}$)
```

If none of the preceding rewrite rules applies anymore, then all negatible 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-C$_1$ ⋯ N-C$_n$))       ⟶   (or   (not N-C$_1$) ⋯ (not N-C$_n$))

(not (and* N-C$_1$ ⋯ N-C$_n$))       ⟶   (or*  (not N-C$_1$) ⋯ (not N-C$_n$))

(not (or   N-C$_1$ ⋯ N-C$_n$))       ⟶   (and  (not N-C$_1$) ⋯ (not N-C$_n$))

(not (or*  N-C$_1$ ⋯ N-C$_n$))       ⟶   (and* (not N-C$_1$) ⋯ (not N-C$_n$))

(not (= TERM$_1$ TERM$_2$))          ⟶   (# TERM$_1$ TERM$_2$)

(not (# TERM$_1$ TERM$_2$))          ⟶   (= TERM$_1$ TERM$_2$)
```

“or”-Removal

```
(case
   CASE$_1$
   ⋮
   CASE$_n$
   (G-C$_1$    ⋯ G-C$_p$
    (or GEN-COND$_1$ ⋯ GEN-COND$_r$)
    G-C$_{p+1}$  ⋯ G-C$_{p+q}$)
   META-TERM
   CASE$_{n+1}$
   ⋮
   CASE$_{n+m}$)

↓

(case
   CASE$_1$
   ⋮
   CASE$_n$
   (G-C$_1$ ⋯ G-C$_p$ GEN-COND$_1$ G-C$_{p+1}$ ⋯ G-C$_{p+q}$ ) META-TERM
   ⋮                                    ⋮
   (G-C$_1$ ⋯ G-C$_p$ GEN-COND$_r$ G-C$_{p+1}$ ⋯ G-C$_{p+q}$ ) META-TERM
   CASE$_{n+1}$
   ⋮
   CASE$_{n+m}$)
```

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

“or*”-Removal

```
(case
   CASE$_1$
   ⋮
   CASE$_n$
   (G-C$_1$   $\cdots$ G-C$_p$
    (or* N-C$_1$ $\cdots$ N-C$_r$)
    G-C$_{p+1}$  $\cdots$ G-C$_{p+q}$)
   META-TERM
   CASE$_{n+1}$
   ⋮
   CASE$_{n+m}$)
```

↓

```
(case
   CASE$_1$
   ⋮
   CASE$_n$
   (G-C$_1$   $\cdots$ G-C$_p$
    N-C$_1$
    G-C$_{p+1}$  $\cdots$ G-C$_{p+q}$)
   META-TERM
   (G-C$_1$   $\cdots$ G-C$_p$
    (not N-C$_1$) N-C$_2$
    G-C$_{p+1}$  $\cdots$ G-C$_{p+q}$)
   META-TERM
   ⋮
   (G-C$_1$   $\cdots$ G-C$_p$
    (not N-C$_1$) $\cdots$ (not N-C$_{r-1}$) N-C$_r$
    G-C$_{p+1}$  $\cdots$ G-C$_{p+q}$)
   META-TERM
   CASE$_{n+1}$
   ⋮
   CASE$_{n+m}$)
```

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

“and[*]”-Removal

```
(case                                                (case
   CASE_1                                               CASE_1
   ⋮                                                    ⋮
   CASE_n                                               CASE_n
   (G-C_1    ··· G-C_p                                  (G-C_1    ··· G-C_p
    (and[*] GEN-COND_1 ··· GEN-COND_r)   ⟶               GEN-COND_1 ··· GEN-COND_r
    G-C_{p+1}   ··· G-C_{p+q})                           G-C_{p+1}   ··· G-C_{p+q})
   META-TERM                                            META-TERM
   CASE_{n+1}                                           CASE_{n+1}
   ⋮                                                    ⋮
   CASE_{n+m})                                          CASE_{n+m})
```

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

“`case`-in-`case`”-Removal

```
 (case
   CASE_1
   ⋮
   CASE_m
   (G-C_1 ··· G-C_p)
   (case
      (GEN-COND_{1,1} ··· GEN-COND_{1,q_1} )        META-TERM_1
      ⋮                                             ⋮
      (GEN-COND_{r,1} ··· GEN-COND_{r,q_r} )        META-TERM_r)
   CASE_{m+1}
   ⋮
   CASE_{m+n})

↓

 (case
   CASE_1
   ⋮
   CASE_m
   (G-C_1 ··· G-C_p GEN-COND_{1,1} ··· GEN-COND_{1,q_1} ) META-TERM_1
   ⋮                                                  ⋮
   (G-C_1 ··· G-C_p GEN-COND_{r,1} ··· GEN-COND_{r,q_r} ) META-TERM_r
   CASE_{m+1}
   ⋮
   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 VAR preceding a match-atom (@ VAR TERM) with the match-term 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 $\mathcal{V}(\texttt{TERM})$ denote the set of variables occurring in TERM.

If $\texttt{VAR} \in \mathcal{V}(\texttt{TERM})$, then the specifier should be warned like:
“WARNING: (@ VAR TERM) re-binds VAR”
and we reduce:

$$\texttt{(@ VAR TERM)} \quad \longrightarrow \quad \texttt{(@@ VAR TERM)}$$

Otherwise we reduce:

$$\texttt{(@ VAR TERM)} \quad \longrightarrow \quad \begin{array}{l} \texttt{(@@ VAR TERM)} \\ \texttt{(let TERM VAR)} \end{array}$$

“@@”-Shift-Left

In case of $\mathcal{V}(\texttt{BASIC-ATOM}) \cap (\mathcal{V}(\texttt{TERM}) \backslash \{\texttt{VAR}\}) \neq \emptyset$ the condition list below is erroneous. Otherwise we reduce:

$$\begin{array}{l} \texttt{(G-C}_1 \\ \;\vdots \\ \;\texttt{G-C}_m \\ \;\texttt{BASIC-ATOM} \\ \;\texttt{(@@ VAR TERM)} \\ \;\texttt{G-C}_{m+1} \\ \;\vdots \\ \;\texttt{G-C}_{m+n}\texttt{)} \end{array} \quad \longrightarrow \quad \begin{array}{l} \texttt{(G-C}_1 \\ \;\vdots \\ \;\texttt{G-C}_m \\ \;\texttt{(@@ VAR TERM)} \\ \;\texttt{BASIC-ATOM}\{\texttt{VAR}\mapsto\texttt{TERM}\} \\ \;\texttt{G-C}_{m+1} \\ \;\vdots \\ \;\texttt{G-C}_{m+n}\texttt{)} \end{array}$$

“let”-Shift-Right

If $\texttt{VAR} \in \mathcal{V}(\texttt{TERM})$, then the specifier should be warned like:
“WARNING: (let TERM VAR) re-binds VAR”

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

$$\begin{array}{l} \texttt{(G-C}_1 \\ \;\vdots \\ \;\texttt{G-C}_m \\ \;\texttt{(let TERM VAR)} \\ \;\texttt{BASIC-ATOM} \\ \;\texttt{G-C}_{m+1} \\ \;\vdots \\ \;\texttt{G-C}_{m+n}\texttt{)} \end{array} \quad \longrightarrow \quad \begin{array}{l} \texttt{(G-C}_1 \\ \;\vdots \\ \;\texttt{G-C}_m \\ \;\texttt{BASIC-ATOM}\{\texttt{VAR}\mapsto\texttt{TERM}\} \\ \;\texttt{(let TERM VAR)} \\ \;\texttt{G-C}_{m+1} \\ \;\vdots \\ \;\texttt{G-C}_{m+n}\texttt{)} \end{array}$$

“let”-“@@”-Swap

This is the only non-trivial rewrite rule.

$$\begin{array}{lcl}
\texttt{(G-C}_1 & & \texttt{(G-C}_1 \\
\ \vdots & & \ \vdots \\
\ \texttt{G-C}_m & & \ \texttt{G-C}_m \\
\ \texttt{(let TERM}_1\texttt{ VAR}_1\texttt{)} & & \ \texttt{<X>} \\
\ \texttt{(@@ VAR}_2\texttt{ TERM}_2\texttt{)} & \longrightarrow & \ \texttt{G-C}_{m+1} \\
\ \texttt{G-C}_{m+1} & & \ \vdots \\
\ \vdots & & \ \texttt{G-C}_{m+n}\texttt{)} \\
\ \texttt{G-C}_{m+n}\texttt{)} & &
\end{array}$$

with <X> defined as follows:

$\texttt{VAR}_1 = \texttt{VAR}_2$: ERROR.

> There is no reasonable semantics for this unless $\texttt{TERM}_1\sigma = \texttt{TERM}_2\xi\sigma$ for some ξ replacing the variables of $\mathcal{V}(\texttt{TERM}_1) \cap \mathcal{V}(\texttt{TERM}_2)$ with new distinct variables and σ being a most general unifier for $\texttt{TERM}_1$ and $\texttt{TERM}_2\xi$.[6] This case, however, is too unlikely and not important enough to give semantics for, since this would make single pass error checking more difficult.

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

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

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

> $\texttt{<X>} = \texttt{(@@ VAR}_2\texttt{ TERM}_2\texttt{)}$
>
> $\qquad\texttt{(let TERM}_1\{\texttt{VAR}_2{\mapsto}\texttt{TERM}_2\}\texttt{ VAR}_1\texttt{)}$
>
> This should be the normal case.

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.

[6] $\texttt{<X> = (@@}^{*}\ {}_{\mathcal{V}(\texttt{TERM}_1)}\!\upharpoonleft\!\sigma\texttt{) (let}^{*}\ ({}_{\mathcal{V}(\texttt{TERM}_2)}\!\upharpoonleft\!(\xi\sigma))^{-1}\texttt{)}$ would correspond to our intention. E.g. for
`(let (mt l y l) k)`
$\texttt{(@@ k (mt h}_1\texttt{ (cons y m) h}_2\texttt{))}$
we would choose $\xi := \{\, \texttt{y} \mapsto \texttt{z} \,\}$; $\sigma := \{\, \texttt{y} \mapsto \texttt{(cons z m)},\ \texttt{h}_1 \mapsto \texttt{l},\ \texttt{h}_2 \mapsto \texttt{l} \,\}$ and get
$\texttt{<X> = (@@ y (cons z m)) (let l h}_1\texttt{) (let l h}_2\texttt{) (let z y) .}$

However, this definition would destroy the confluence of ‘⟶’. E.g. consider the following condition-list where y is an alias for u: `((@ x (s u)) (@ x (s y)))` ⟶
`((@@ x (s u)) (let (s u) x) (@@ x (s y)) (let (s y) x))`.
The latter condition-list reduces in two ways. First with $\xi := \{\,\},\ \sigma := \{\, u \mapsto y \,\}$:
⟶ `((@@ x (s u)) (@@ u y) (let (s y) x))`.
Second with $\xi := \{\,\},\ \sigma := \{\, y \mapsto u \,\}$:
⟶ `((@@ x (s u)) (let u y) (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

```
 (macro-rule TERM                        (macro-rule TERM
   (case                                    (case CASE_1))
      CASE_1                      ⟶      ⋮
      ⋮                                  (macro-rule TERM
      CASE_n))                              (case CASE_n))
```

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

```
(macro-rule TERM_1
   (case (MATCH_1 ··· MATCH_m
          BASIC-ATOM_1 ··· BASIC-ATOM_p
          LET_1 ··· LET_n)
         TERM_2))
```

or

```
(macro-rule TERM_1 TERM_2).
```

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

“@@”-removal

In case of $\mathcal{V}(\texttt{TERM}_1) \cap (\mathcal{V}(\texttt{TERM}_2)\setminus\{\texttt{VAR}\}) \neq \emptyset$ the specification is erroneous. Otherwise we reduce:

```
(macro-rule TERM_1
   (case
      ((@@ VAR TERM_2) G-C_1 ··· G-C_m)
      META-TERM))
```

↓

```
(macro-rule TERM_1{VAR↦TERM_2}
   (case
      (G-C_1 ··· G-C_m)
      META-TERM))
```

“let”-removal

$$
\begin{array}{l}
\texttt{(case}\\
\qquad\texttt{CASE}_1\\
\qquad\vdots\\
\qquad\texttt{CASE}_m\\
\qquad\texttt{(G-C}_1\ \cdots\ \texttt{G-C}_m\ \texttt{(let TERM}_1\ \texttt{VAR))}\qquad\texttt{TERM}_2\\
\qquad\texttt{CASE}_{m+1}\\
\qquad\vdots\\
\qquad\texttt{CASE}_{m+n}\texttt{)}
\end{array}
$$

$$\downarrow$$

$$
\begin{array}{l}
\texttt{(case}\\
\quad\texttt{CASE}_1\\
\quad\vdots\\
\quad\texttt{CASE}_m\\
\quad\texttt{(G-C}_1\ \cdots\ \texttt{G-C}_m\texttt{)}\qquad\qquad\qquad\qquad\qquad\texttt{TERM}_2\{\texttt{VAR}\mapsto\texttt{TERM}_1\}\\
\quad\texttt{CASE}_{m+1}\\
\quad\vdots\\
\quad\texttt{CASE}_{m+n}\texttt{)}
\end{array}
$$

“case-with-empty-condition”-Removal

$$\texttt{(macro-rule TERM}_1\ \texttt{(case () TERM}_2\texttt{))}\ \longrightarrow\ \texttt{(macro-rule TERM}_1\ \texttt{TERM}_2\texttt{)}$$

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).