Proxac: an editor for program transformation

Jan L.A. van de Snepscheut

This note is an introduction to an editor oriented to the production of programs and proofs.

1 Introduction and overview

The development of programs from their specification via a process known as transformational programming is in principle well-known. Yet it is rarely practiced because the process is often tedious and error-prone. We have developed an editor that supports this programming method by automating the tedious parts of it.

The editor, called `proxac` for program and proof transformation and calculation, can be found in `~modula3/edit/proxac`. Writing this editor was, and is, a challenging undertaking. Diana Finley was instrumental in getting this project underway. Greg Davis contributed many ideas and helped getting the program to the point where it actually became usable. My thanks go to both of them.

Figure 1 contains a view of the editor. It shows the configuration after expression

```
max@sum@segs
```

has been entered (on the first `EDIT` line) and the three definitions in the rightmost window have been applied to it. The example as well as the transformation rules are taken from [1]. The notation used is some concise programming language, and doesn't concern us too much now. The underlining of the last line in the edit window indicates the expression that is the focus of attention. We illustrate how a transformation step is carried out. Figure 2 contains the configuration after the first transformation rule has been selected. This selection is carried out by moving the mouse to the rules window and depressing the $\downarrow$ and $\uparrow$-keys until the proper rule is underlined.

Next, pushing the `a`-key will apply the rule to the expression that is the focus of attention. The editor attempts to match the focus against the left-hand side or the right-hand side of the rule. In this

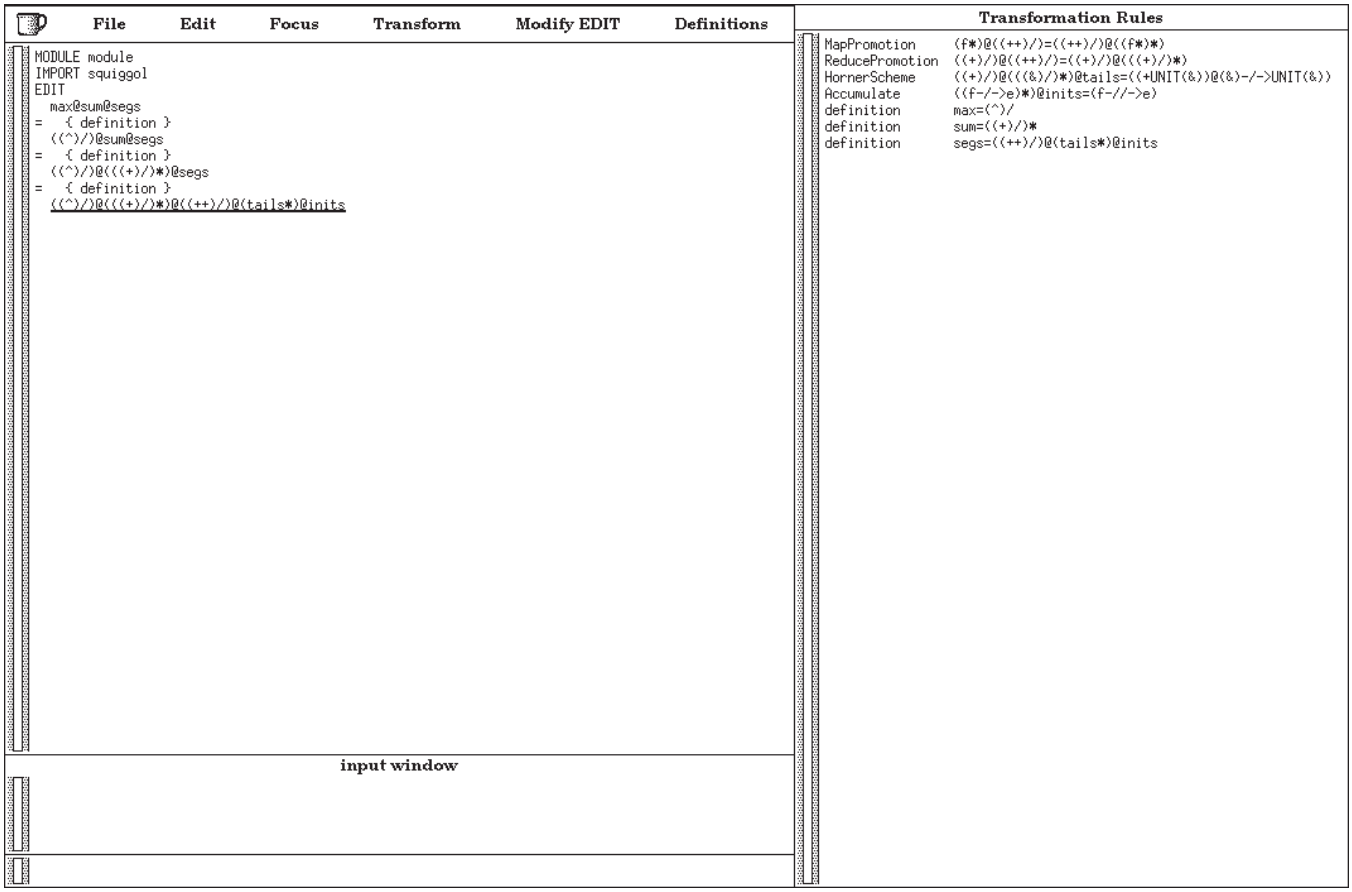

Figure 1: The window

case, no such match will be found and then the editor proceeds by attempting to match a subexpression of the focus against the rule. (To suppress this recursive matching, hold down the Control key when pushing the `a`-key.) In this particular case, the subexpression that matches the left-hand side of the rule is expression

```
((+/)*)@(++/)
```

provided that `+/` is substituted for `f` in the rule. Given this substitution, which is automatically generated by the editor, the right-hand side of the rule reduces to

```
((++/)*)@(((+/)*)*)
```

and these two results are then displayed by the editor as show in Figure 3. Line

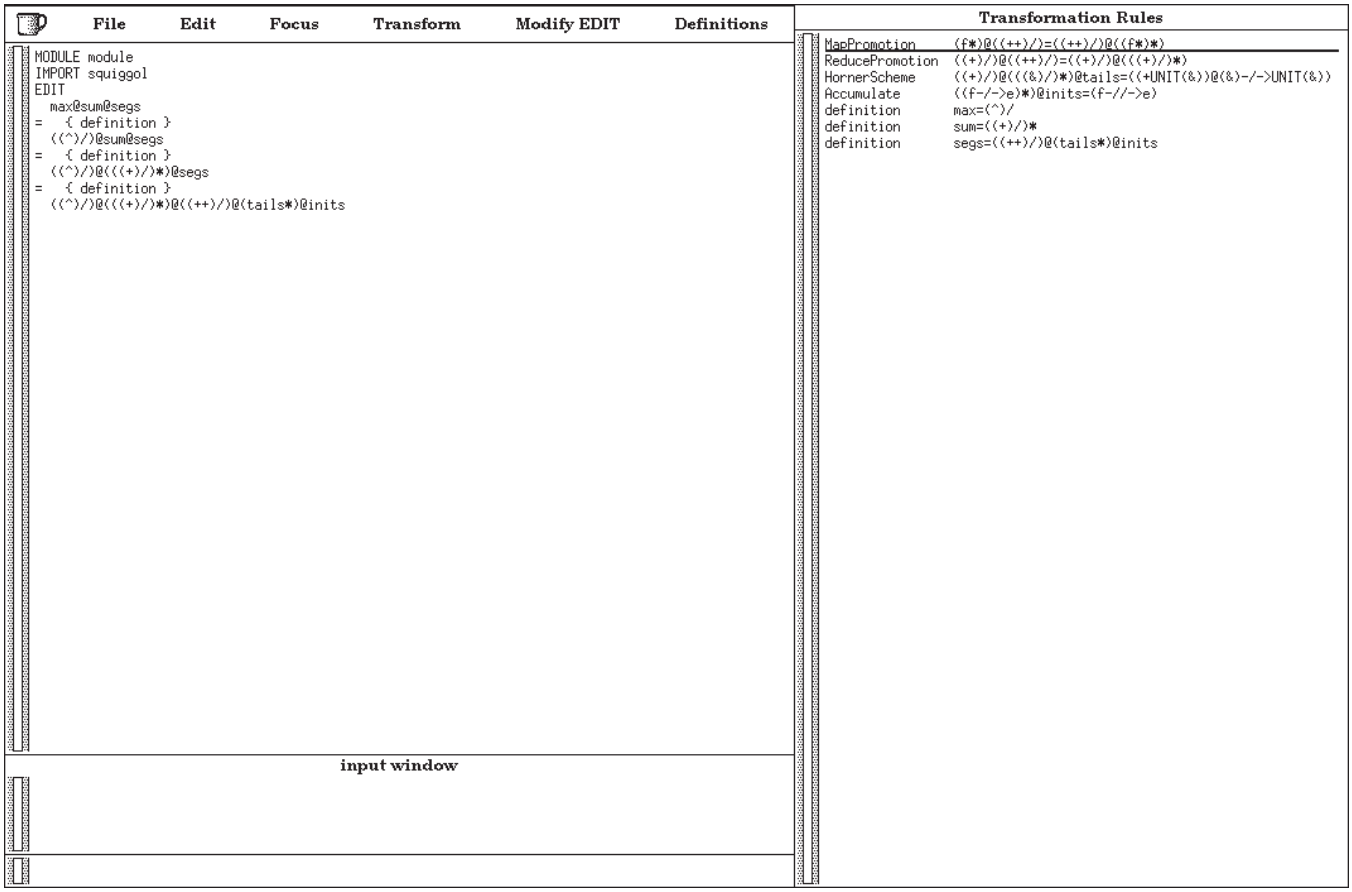

Figure 2: The window

```
= { MapPromotion[f:=+/] }
```

indicates that the programs preceding and following it are equal, and that their equality is justified by the rule labeled `MapPromotion`. The justification can be checked by substituting `+/` for `f` in the rule. The result of the replacement is shown in the new focus.

Figure 4 contains the configuration after a few more steps. The same text can be found in Appendix where the font size is not as minuscule.

The idea of the transformational programming method is that the first line in Figure 4 is the specification, a program that clearly expresses the job to be carried out, but possibly in an inefficient way. Each of the transformation steps is known to be correct, they are part of the theory that comes with the programming language at hand. Each transformation preserves the program's correctness and may

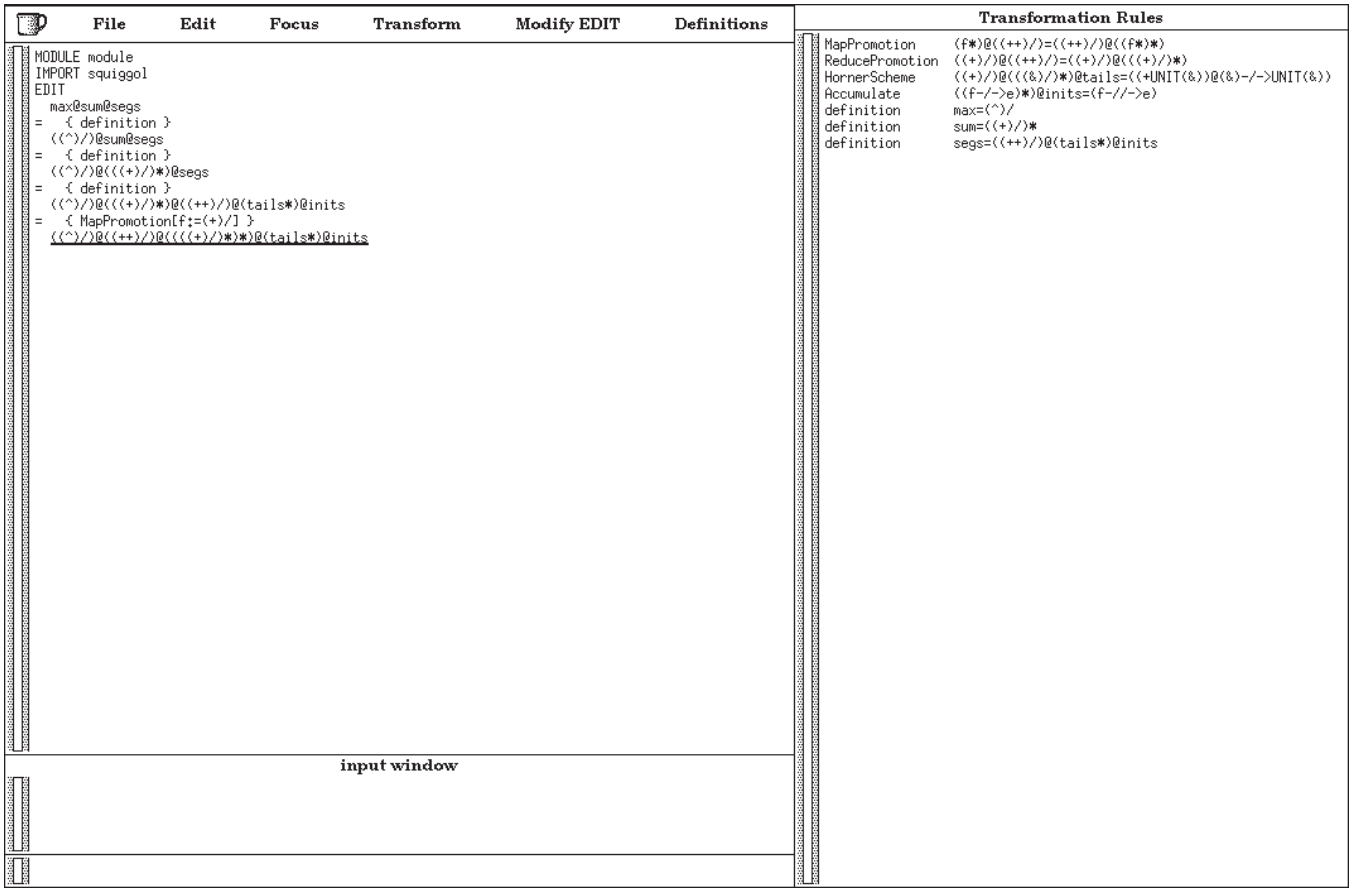

Figure 3: The window

improve its efficiency. In the example, the program is given a list of integers, not necessarily all positive. A segment is a consecutive sublist of the given list and each segment has a sum. The problem is to compute the maximum of all those segment sums. The initial program uses some standard functions to first generate all segments, then compute all their sums, and finally compute the maximum thereof. If the given list has length n, the running time of this program is proportional to n^3. The final program is much trickier but runs in time proportional to n. If we know that the transformation rules are correct, then the resulting efficient program is known to be correct also.

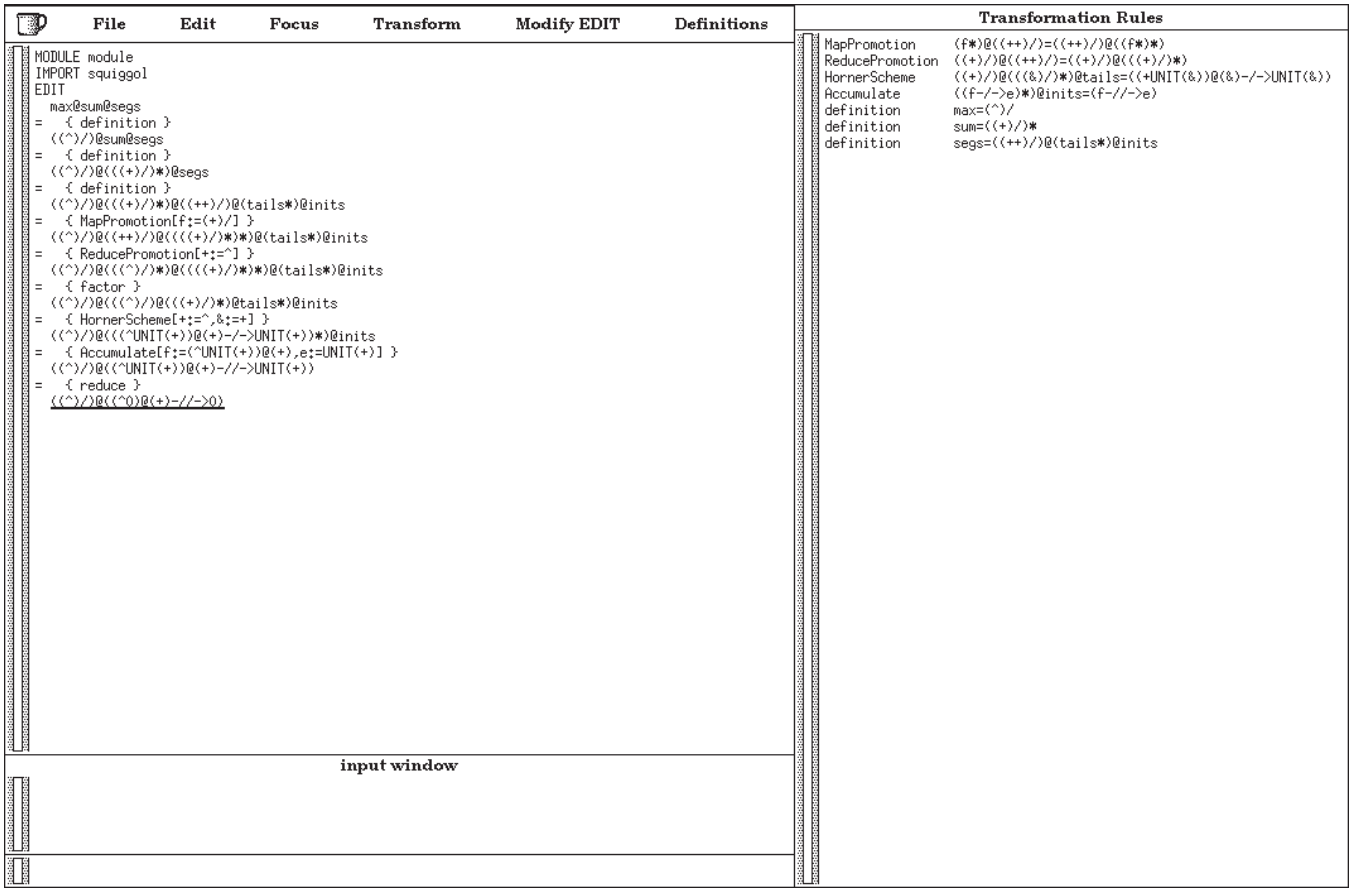

Figure 4: The window

2 Expressions

In this section we describe the notation used for expressions. This part is strongly influenced by a proof editor reported in [2]. Compared to most theorem provers or proof checkers, the outstanding feature is that the notation is based on infix (and prefix and postfix) expressions instead of on function application. The main reason is that being able to write

$$a + b + c$$

instead of the cumbersome

$$sum(a, sum(b, c))$$

(let alone having to distinguish between this and $sum(sum(a,b),c)$) is such an enormous gain that infix expressions are a sine qua non. Every operator has a number of attributes that determine its interpretation. One of its attributes is whether it is an infix, prefix, or postfix operator. Every infix operator $\oplus$ has a number of additional attributes.

- Every operator has one of the following associativity attributes:
  - associative, in which case parentheses can be omitted from expressions like $x \oplus (y \oplus z)$. We write `ASSOCIATIVE`$(\oplus)$ for associative operators.
  - left-associative, in which case $x \oplus y \oplus z$ stands for $(x \oplus y) \oplus z$. We write `LASSOCIATIVE`$(\oplus)$;
  - right-associative, in which case $x \oplus y \oplus z$ stands for $x \oplus (y \oplus z)$. We write `RASSOCIATIVE`$(\oplus)$;
  - not associative, in which case $x \oplus y \oplus z$ is unacceptable.

  One may write `LASSOCIATIVE`$(\oplus) \ \wedge \$ `RASSOCIATIVE`$(\oplus)$ for `ASSOCIATIVE`$(\oplus)$.
- An infix operator may be conjunctive. For conjunctive operator $\leq$, expression $x \leq y \leq z$ equals $x \leq y \ \wedge \ y \leq z$. A conjunctive operator is not associative. We write `CONJUNCTIVE`$(\leq)$. One can mix various conjunctive operators. If both $\leq$ and $<$ are conjunctive, $x \leq y < z$ equals $x \leq y \ \wedge \ y < z$.
- An infix operator may be transitive. For transitive operator $\leq$, expression $x \leq y \ \wedge \ y \leq z$ implies $x \leq z$. We write `TRANSITIVE`$(\leq)$.
- An infix operator may be idempotent. For idempotent operator $\oplus$, expression $x \oplus x$ equals x. We write `IDEMPOTENT`$(\oplus)$.
- An infix operator may or may not have a dual. If operator $\leq$ has dual $\geq$ then $x \leq y$ and $y \geq x$ are equal. An infix operator is symmetric (or commutative) just when it is its own dual. We write `DUAL`$(\leq) = \geq$.
- An infix operator may have a left unit `LUNIT`$(\oplus)$, and it may have a right unit `RUNIT`$(\oplus)$. If `LUNIT`$(\oplus) = l$ then $l \oplus e = e$. If `RUNIT`$(\oplus) = r$ then $e \oplus r = e$. If both are defined, they have to be equal and we write `UNIT`$(\oplus)$ for the unit of $\oplus$.
- An infix operator may have a left zero `LZERO`$(\oplus)$, and it may have a right zero `RZERO`$(\oplus)$. If `LZERO`$(\oplus) = l$ then $l \oplus e = l$. If `RZERO`$(\oplus) = r$ then $e \oplus r = r$. If both are defined, they have to be equal and we write `ZERO`$(\oplus)$ for the zero of $\oplus$.

Following Dijkstra, we write an explicit infix operator for function application, such as $fib.n$ for the application of function fib to argument n. If function max has two arguments, we write $max.x.y$ and stipulate that . associates to the left. This boils down to saying that the previous function application can be read as $(max.x).y$ and this in turn corresponds to how we usually Curry a function with more than one argument: max is a function of one argument that returns a function of one argument.

We write `@` for function composition, which is an associative operator. We have

$$(f@g@h).x \ = \ f.(g.(h.x))$$

We use the following notation for functions. It is inspired by our notation for quantification. For example,

$$\forall(n \mid n \geq 0 \triangleright (n+1)! = (n+1) * n!)$$

stands for the universal quantification of term $(n+1)! = (n+1) * n!$ over all n for which $n \geq 0$ holds. Similarly,

$$\exists(n \mid n \geq 0 \triangleright (n+1)! = (n+1) * n!)$$

stands for the existential quantification of the same terms. It is therefore useful to think of

$$(n \mid n \geq 0 \triangleright (n+1)! = (n+1) * n!)$$

as a unit in its own right to which a quantifier can be applied. We think of it as the function that maps n to $(n+1)! = (n+1) * n!$ for all n for which $n \geq 0$ holds. Quantifier Q is then a prefix operator that is applied to a function. It has two attributes in addition to being a prefix operator. One is the infix operator $\texttt{INFIXOPERATOR}(Q)$ that Q is the continued version of, and the other is the constructor function $\texttt{CONSTRUCTOR}(Q)$ that is applied to all terms before subjecting them to the infix operator. Infix operator $\texttt{INFIXOPERATOR}(Q)$ should be symmetric and associative. Here are some examples. Universal quantification $\forall$ has attributes $\wedge$ and *id*. Summation $\sum$ has attributes $+$ and *id*. Numeric quantification $\mathbf{N}$ has attributes $+$ and the function that maps *true* to 1 and *false* to 0. Set formation has attributes $\cup$ and the function that maps x to $\{x\}$.

The range of the bound variable(s) is written between the symbols $\mid$ and $\triangleright$. The construct $\mid$*true* may be omitted. For example, the function that yields the average of two numbers is written

$$(x, y \triangleright (x+y)/2) \quad .$$

By the way, we switched from the notation

$$(x, y \ : \ x > 0 \ : \ (x+y)/2)$$

to

$$(x, y \mid x > 0 \triangleright (x+y)/2)$$

to keep the door open for the introduction of type declarations, which might look like

$$(x, y : \mathit{real} \mid x > 0 \triangleright (x+y)/2) \quad .$$

We admit operators as function parameters also. For example, we write

$$(x, \texttt{PREFIX}\ \pi, y \triangleright ...)$$

for a function whose first argument is any expression, whose second argument is a prefix operator, and whose third argument is an any expression. Similarly, one can write

`POSTFIX` !

`INFIX` *number* $\oplus$

`QUANTIFIER` Q

for other operators. Prefix and postfix operators are unary. The integer in the case of an infix operator indicates its precedence level. Any integer number can be used as a precedence level; a higher number implies that the operator binds more strongly. Prefix and postfix operators bind even more strongly than infix operators.

Below we describe how attributes of operators (such as their associativity, unit elements, and so on) can be described and one may include requirements on operator parameters in the function's range as one would for other parameters. Those properties are also used by the parser when parsing the function's body.

In some applications (such as the Bird/Meertens calculus [1]) one finds infix expressions from which the leftmost or the rightmost operand has been omitted, such as $*2$. These are functions, but it would be painful having to write $(x \mid true \triangleright x*2)$ or $(x \triangleright x*2)$ instead of simply $*2$. These incomplete expressions have become known as left and right sections (the adjective indicates which operand is missing).

Quite often, we have a need for writing an expression together with some substitutions that need to be carried out. The substitutions are understood to be performed simultaneously, so that $(x < y)[x := y+1, y := x+1]$ is equivalent to $y+1 < x+1$.

The goal of the editor is to come up with a sequence of transformation steps that connect a number of expressions. Such a sequence is called a calculation. It is a sequence of one or more formulae, and each formula is connected to the next by an operator. The connective is accompanied by a hint. It is typically something like

```
    ((a+b)*n)[n:=2]
=     { distribution }
    (a*n+b*n)[n:=2]
=     { reduce }
    a*2+b*2
```

and we treat such a calculation also as an expression. It is the presence of the hints, written in curly braces, that distinguishes them from other expressions.

3 The transformation menu

As shown in Figure 4, the last transformation step is not an application of one of the rules from the rule window, but an applications of one of the predefined rules called `Reduce`. In this section, we describe the transformations carried out by `Reduce` and other predefined transformations.

3.1 Reduction

The `Reduce` action is activated by either pressing the `r`-key in the edit window, or by downclicking the mouse button on the `Transformation` menu, which will pop up a menu, followed by an upclick on the

`Reduce` option. Reductions are first applied recursively to all subexpressions of the current focus, and then to the expression itself. To suppress this recursive application, hold down the Control key when activating the Reduce action. The reduction is applied exactly to the focus, the underlined expression, and not to some subexpression thereof. The reductions are tabulated below.

Let π be a prefix operator, ! a postfix operator, and $\oplus$ an infix operator.

`LUNIT(` $\oplus$ `)`	the left unit element of $\oplus$	
`RUNIT(` $\oplus$ `)`	the right unit element of $\oplus$	
`UNIT(` $\oplus$ `)`	the unit element of $\oplus$	
`LZERO(` $\oplus$ `)`	the left zero element of $\oplus$	
`RZERO(` $\oplus$ `)`	the right zero element of $\oplus$	
`ZERO(` $\oplus$ `)`	the zero element of $\oplus$	
`TRANSITIVE(` $\oplus$ `)`	the boolean transitivity attribute of $\oplus$	
`ASSOCIATIVE(` $\oplus$ `)`	similar	
`LASSOCIATIVE(` $\oplus$ `)`	similar	
`RASSOCIATIVE(` $\oplus$ `)`	similar	
`CONJUNCTIVE(` $\oplus$ `)`	similar	
`IDEMPOTENT(` $\oplus$ `)`	similar	
`DUAL(` $\oplus$ `)`	the dual of $\oplus$	
`INFIXOPERATOR(Q)`	the infix operator of quantifier `Q`	
`CONSTRUCTOR(Q)`	the constructor of quantifier `Q`	
`x` $=$ `x`	`true`	
`x` $\leq$ `x`	`true`	
`x` $\geq$ `x`	`true`	
`x` $\neq$ `x`	`false`	
`x` $<$ `x`	`false`	
`x` $>$ `x`	`false`	
`(x,y|range` $\triangleright$ `expr).a.b`	$\texttt{expr}_{\texttt{a,b}}^{\texttt{x,y}}$	provided $\texttt{range}_{\texttt{a,b}}^{\texttt{x,y}}$ holds
`(x|range` $\triangleright$ `expr).a.b`	$\texttt{expr}_{\texttt{a}}^{\texttt{x}}$`.b`	provided $\texttt{range}_{\texttt{a}}^{\texttt{x}}$ holds
`(x,y,z|range` $\triangleright$ `expr).a.b`	$(\texttt{z}\vert\texttt{range}_{\texttt{a,b}}^{\texttt{x,y}} \triangleright \texttt{expr}_{\texttt{a,b}}^{\texttt{x,y}})$	
`(f@g@h).x`	`f.(g.(h.x))`	
`(` $\oplus$ `a` $\oplus$ `b` $\oplus$ `).x`	`x` $\oplus$ `a` $\oplus$ `b` $\oplus$	
`(` $\oplus$ `a` $\oplus$ `b` $\oplus$ `c).x`	`x` $\oplus$ `a` $\oplus$ `b` $\oplus$ `c`	
`(a` $\oplus$ `b` $\oplus$ `c` $\oplus$ `).x`	`a` $\oplus$ `b` $\oplus$ `c` $\oplus$ `x`	
`(` $\oplus$ `a` $\oplus$ `b` $\oplus$ `c` $\oplus$ `).x.y`	`x` $\oplus$ `a` $\oplus$ `b` $\oplus$ `c` $\oplus$ `y`	
π`.x`	π `x`	
`!.x`	`x!`	
$\oplus$`.x.y`	`x` $\oplus$ `y`	
$\oplus$`.x`	`x` $\oplus$	
`id.x`	`x`	
`l` $\oplus$ `x`	`x`	if `l=LUNIT(` $\oplus$ `)`
`x` $\oplus$ `r`	`x`	if `r=RUNIT(` $\oplus$ `)`
`l` $\oplus$ `x`	`l`	if `l=LZERO(` $\oplus$ `)`

`x` $\oplus$ `r`	`r`	if `r=RZERO(` $\oplus$ `)`
`x` $\oplus$ `x`	`x`	if `IDEMPOTENT(` $\oplus$ `)`
`(x` $\triangleright$ `x)`	`id`	
`(x` $\triangleright$ π `x)`	π	if $\pi \neq x$
`(x` $\triangleright$ `x !)`	`!`	if $! \neq x$
`(x` $\triangleright$ `a` $\oplus$ `x)`	`a` $\oplus$	if `x` not free in `a` $\oplus$
`(x` $\triangleright$ `x` $\oplus$ `a)`	$\oplus$ `a`	if `x` not free in $\oplus$ `a`
`(x,z` $\triangleright$ `x` $\oplus$ `y` $\oplus$ `z)`	$\oplus$ `y` $\oplus$	if `x,z` not free in $\oplus$ `y` $\oplus$
`expr[x:=e,y:=f]`	$\mathtt{expr}^{\mathtt{x,y}}_{\mathtt{e,f}}$	
for quantifier `Q`, let $\oplus$ = `INFIXOPERATOR(Q)` and `c=CONSTRUCTOR(Q)`		
`Q(i | false` $\triangleright$ `expr)`	`UNIT(` $\oplus$ `)`	
`Q(i | i=k` $\triangleright$ `expr)`	`c.(expr[i:=k])`	provided `i` does not occur in `k`
`Q(i | i=k` $\triangleright$ `expr)`	`expr[i:=k]`	if `c=id` and `i` does not occur in `k`
`Q(i | a` $\vee$ `b` $\triangleright$ `expr)`	`Q(i | a` $\wedge \neg$ `b` $\triangleright$ `expr)` $\oplus$ `Q(i | b` $\triangleright$ `expr)`	
`Q(i | a` $\vee$ `b` $\triangleright$ `expr)`	`Q(i | a` $\triangleright$ `expr)` $\oplus$ `Q(i | b` $\triangleright$ `expr)`	
		if $\oplus$ is idempotent or
		if $a \wedge b = \mathit{false}$
`a` $<$ `{h1}b` $\leq$ `{h2}c`	`a` $<$ `{h1; h2}c`	if $<$ and $\leq$ can be combined to $<$

In the last rule, for reducing a calculation, there is the proviso that $<$ and $\leq$ can be combined. The rule for combining operators is tabulated below. For any infix operator $\sim$

$\sim$	and $=$	can be combined, giving $\sim$	
$\sim$	$\sim$	$\sim$	provided $\sim$ is a transitive operator
$\leq$	$<$	$<$	
$\geq$	$>$	$>$	

The last two lines indicate that we should establish some relation between operators $\sim$ and $\simeq$ for which we have $x \simeq y \;=\; (x \sim y \vee x = y)$. Right now, it is an ad-hoc thing.

3.2 Factorization and distribution

A transformation that is carried out frequently is to switch from $a*(b+c)$ to $(a*b)+(a*c)$, which is called distribution, or vice versa, which is called factorization. The `Transformation` menu offers two distribution and two factorization operations: one that operates on the left, and one on the right. Going from $a*(b+c)$ to $(a*b)+(a*c)$ is called a left distribution, whereas going from $(b*a)+(c*a)$ to $(b+c)*a$ is called a right factorization.

Left distribution is applied to an infix expression, say $x*y*z$. It distributes the whole expression minus the last term, that is section $x*y*$, through the last term, that is through z. The result depends on the form of z. Notice that $x*y*$ acts just like a prefix operation, and one might correctly suspect that expression πz works in a similar fashion. It is also perfectly okay for x to be missing, in which case we have left-sectioned results.

`x*y*(a+b)`	transforms to	`(x*y*a)+(x*y*b)`	
`x*y*(v | R ▷ t)`		`(v | R ▷ x*y*t)`	provided `v` does not occur in `x*y*`
`x*y*Q(v | R ▷ t)`		`Q(v | R ▷ x*y*t)`	where `Q` is a quantifier and `v` does not occur in `x*y*`

Factorization is just the opposite.

Observe that the transformation from $a * (b + c)$ to $(a * b) + (a * c)$ makes sense only if $*$ left-distributes through $+$ in the theory that one is working with. This condition is checked by the editor. Also observe that the transformation from $f.(x + y)$ to $f.x + f.y$ is an instance of a left distribution.

3.3 Reverse

This transformation reverses the order of the operands in an infix expression and changes the operator to its dual. If the focus is a conjunction, it reverses the order of the operands and operators, and transforms each operator into its dual. If the focus is a calculation, all steps are put in reverse order, and all connectives are replaced by their dual.

3.4 Flatten

If an expression happens to contain superfluous parentheses, as in $a + (b + c)$, they can be removed by `Flatten` which transforms the expression into $a + b + c$.

3.5 Copy

This operation makes a plain copy of the expression without any transformation being carried out.

3.6 Swap Exprs

This operation requires the selection of a secondary focus in addition to the first focus. See section 4 for a description on selecting a secondary focus. The effect of executing this operation is to swap the two expressions. They are assumed to be siblings: operands of one and the same symmetric infix operation.

3.7 Regroup

This operation requires the selection of a secondary focus in addition to the first focus. They are assumed to be adjacent siblings. Regrouping inserts a pair of parentheses around these two expressions. For example, regrouping expression $a + b + c$ when a and b are the primary and secondary focus, yields $(a + b) + c$.

3.8 SplitConj

This operation requires the focus to be a conjunctive expression, like $a \leq b < c$, and splits it into the conjunction of its individual terms, like $a \leq b \quad \wedge \quad b < c$.

3.9 Rule with Hint

This operation is a form of rule application where the editor is first given a hint as to which substitution to apply. For example, if rule

$$\texttt{RULE CaseAnalysis} : (\texttt{a}, \texttt{b} \triangleright \texttt{a} \;=\; (\texttt{a} \wedge \texttt{b}) \vee (\texttt{a} \wedge \neg \texttt{b}))$$

is applied to expression

$$\texttt{x} = \texttt{y}$$

one might give the hint to choose $\texttt{s} > \texttt{0}$ for $\texttt{b}$ to produce the new expression

$$((\texttt{x} = \texttt{y}) \wedge (\texttt{s} > \texttt{0})) \;\vee\; ((\texttt{x} = \texttt{y}) \wedge \neg(\texttt{s} > \texttt{0})) \qquad .$$

There is no way that the editor could have determined this substitution for $\texttt{b}$ without further input. We would write

$$\texttt{hint}[\texttt{b} := \texttt{s} > \texttt{0}]$$

in the input window to supply the hint, focus on the rule, and then apply the transformation.

3.10 Substitute

This operation is a form of rule application in which the rule is not given in the rules window, but is given as an expression in the edit window. This expression is taken from the secondary focus (see next section) and is often used to transform an expression like

$$\texttt{a} \uparrow \texttt{2} \leq \texttt{r} < \texttt{b} \uparrow \texttt{2} \;\wedge\; \texttt{a} = \texttt{0}$$

into

$$\texttt{0} \uparrow \texttt{2} \leq \texttt{r} < \texttt{b} \uparrow \texttt{2} \;\wedge\; \texttt{a} = \texttt{0}$$

by selecting $\texttt{a} = \texttt{0}$ with the secondary focus, by selecting the initial $\texttt{a}$ with the primary focus, and by then applying the Substitute transformation.

4 The focus menu

So far, the focus of attention has always been the last line of the calculation in the edit window. Sometimes, we want the focus to be narrowed to a subexpression thereof. And sometimes the focus is on an expression that is not part of the last line. The operations in the selection menu allow another expression to be selected as the focus of attention. Six operations are provided: `In`, `Out`, `Left`, `Right`, `Up`, and `Down`. Their respective effect is to shift the focus to a subexpression, to a superexpression, to the expression to the left (in some list), to the right, to the expression one line up, or to the expression one line down. For example, to shift the focus from c to b in expression $(a+b)*(c+d+e)$, one cannot just apply `Left` because c is already the leftmost expression in its list. Instead, one would go through `Out` to shift the focus to $c+d+e$, then `Left` to shift the focus to $a+b$, then `In` to shift the focus to a, and finally `Right` to shift the focus to b. The `Left`, `Right`, `Up`, and `Down` operations can also be activated via the arrow keys.

We have mentioned the need for a secondary focus. Operation `Set Focus 2` sets the secondary focus to whatever expression holds the primary focus. The primary focus remains unchanged. The primary focus is indicated by solid underlining, and the secondary focus is indicated by dotted underlining. When the two foci coincide, the expression is highlighted instead of underlined.

The prseent version of the editor suffers from some inconveniences that are to some extent due to my inability to understand the underlying window system. (The program is written in Modula3 and uses `FormsVBT` to set up the windows and communicate with them.) When the program is started, the windows are set up, but no underlining is shown. Even so, there is a focus but for some mysterious reason it is not shown. Click in the edit window (with the left mouse button) and use the arrow keys to change the focus. You should now see an underlining appear and this is the focus. Whenever you change your action to another window, for example when you were editing and then want to move up or down the focus in the rules window, you have to click in the new window first. This causes the underlining in the old window to disappear. It does not remove the focus, but it does remove the underlining.

5 The Edit Menu

The `Edit` menu offers four buttons.

- `From Keyboard`

  Replaces the expression that holds the focus with the expression read from the input window.

- `To Keyboard`

  Writes the expression that holds the focus to the input window.

- `Delete Step`

  Removes the last step in a calculation, provided it is the focus.

- `Undo`

  Not yet implemented.

6 The File Menu

The `File` menu offers a number of buttons, including the following.

- `Open`

  To read a module from a file.

- `Save`

- `Save As`

  To save a module in a file.

- `Print LaTeX`

  Not yet implemented.

- `Quit`

  To terminate the program.

7 Notations

We have explained that the editor is primarily based on expressions written in infix notation. Sometimes one encounters expressions written in a very different format. For example, one writes $|x|$ for the absolute value of x, or `WHILE` b `DO` s for a loop in some program notation. It is, therefore, possible to introduce notations other than infix operators. Right now, this mechanism is sometimes a bit clumsy and not thought through; it is likely to change a bit.

A notation like the loop construct can be declared by writing

```
NOTATION WHILE ! DO !
```

In this example, there are two so-called keywords and two parameters. The parameters are anonymous and their position is indicated by exclamation marks. One may substitute any expression for such a parameter. A keyword need not be a sequence of letters; it may also be an operator, or any of the special symbols `[`, `]`, `,`, `:`, or `|` that cannot be used as operators. Keywords and parameters need not alternate. One restriction in the present version is that the first element is a keyword, not a parameter.

So much for the simple case in which one expression is substituted for a parameter. Sometimes it is ever so nice to be able to substitute a list of expressions for a single parameter. For example, when writing functions, we write a list of variables between the opening `(` and the `|` or $\triangleright$ symbols. The appendix contains an example of the specification statement, which is written

```
SPEC r,q:[true,(r^2<=s<q^2) /\ (q=r+1)]
```

in which keyword `SPEC` is followed by a list of names (interpreted as the names of those variables that can be assigned a new value through the statement). It is obviously highly undesirable to having to write lots of declarations

```
NOTATION SPEC ! : [!, !]
NOTATION SPEC !,! : [!, !]
NOTATION SPEC !,!,! : [!, !]
```

Instead, we write

```
NOTATION SPEC !LIST : [!, !]
```

and instantiate it with as many expressions as desired (well, one or more). They are separated by commas. There is no way to restrict the expressions to being identifiers only.

In some cases, the parameters are actually introducing new scopes. This is the case for the bound variables of functions, and it might be the case for newly introduced constructs as well. In such a case, we do not write `!` but write `?` instead. For example,

```
FOR ? := ! TO ! DO ! END
```

could be an attempt to write a for loop in some programming language with the intent that the control variable be local to the loop. To indicate the scope of the newly introduced variable, in this case to indicate that the scope is the body of the loop but not the two bounds, we name the variable and list it in the other parameters.

```
FOR ?(v) := ! TO ! DO !(v) END
```

For the sake of completeness, we also have

```
?LIST(v)
```

to allow a list of new bound variables, and

```
!(v,w)
```

to indicate that this parameter is included in the scope of both the `v` and the `w` variables. Warning: Not all of these constructs have been implemented completely (yet).

8 Modules

In this section we discuss the module structure that has been ignored so far. As can be seen from the Figures, the edit window starts with the lines

```
MODULE MaximumSegmentSum
IMPORT squiggol
```

whereas the actual editing was done on the expression following the line

```
EDIT
```

In this section we discuss the role of modules. First, what does a module look like? A module starts with a heading that lists the name of the module. In the example, the name is `MaximumSegmentSum`. A module contains a list of items. Each item is a declaration of an identifier, an operator, or a notation, is an import, is the statement of a property, or is a rule. The second line of our example imports the module named `squiggol`. It is read from file `squiggol.mod`. Here is the text of that module.

```
MODULE squiggol

DECLARE INFIX 0 ++

PROPERTY ALL(f,g|>((f@g)*) = ((f*) @ (g*)))

DECLARE INFIX 0 -/->

DECLARE INFIX 0 -//->

DECLARE tails, inits

RULE MapPromotion: (f |>
  (f *) @ ((++) /) = ((++) /) @ ((f *) *))

RULE ReducePromotion: (INFIX 0 + |
  | ASSOCIATIVE(+) |>
  ((+) /) @ ((++) /) = ((+) /) @ (((+) /) *))

RULE HornerScheme: (INFIX 0 +, INFIX 0 & |
  ASSOCIATIVE(+) /\ ASSOCIATIVE(&) /\ ALL(a,b,c|>((a+b)&c)=((a&c)+(b&c))) |>
  ((+) /) @ (((&) /) *) @ tails = (((+ UNIT(&)) @ (&)) -/-> UNIT(&)))

RULE Accumulate: (f,e |>
  ((f -/-> e) *) @ inits = (f -//-> e))

DECLARE INFIX 0 ^

PROPERTY ASSOCIATIVE(^)

PROPERTY ALL(a,b,c|>((a^b)+c)=((a+c)^(b+c)))

RULE definition: max = (^) /

RULE definition: sum = ((+) /) *

RULE definition: segs = ((++) /) @ (tails *) @ inits
```

This module contains examples of many of the constructs. For example,

```
DECLARE INFIX 0 ++
```

declares ++ to be an infix operator of precedence level 0. Prefix and postfix operators are declared with the constructs

```
DECLARE POSTFIX !
```

and

```
DECLARE PREFIX not
```

and a quantifier is declared as

```
DECLARE QUANTIFIER SUM
```

The squiggol module also shows the declaration of identifiers `tails` and `inits`. One often declares identifiers that are not operators, but special constants. In the module above, `tails` and `inits` are functions that are not defined in the module, although some of their properties are revealed in the rules that follow. Properties can also be revealed using a `PROPERTY`. An example is

```
PROPERTY INFIXOPERATOR(SUM)=+ /\ CONSTRUCTOR(SUM)=id
```

which states that quantifier `SUM` is the well-known summation quantifier. Often, properties of operators are given in such a `PROPERTY` statement. One might, for example, wonder why

```
PROPERTY DUAL(^)=^
```

was not included in the module. (It was not needed.) The editor uses the information revealed by declarations and properties when parsing expressions, and when checking the applicability conditions of certain reductions and transformation rules.

The `RULE` construct is the one to introduce rules. We have seen that rules are crucial in the operation of the editor, and the example shows how they are introduced. The keyword `RULE` is followed by the name of the rule, a colon, and the rule itself. A rule is, in general, a function whose body is an equation. Here is an example. Horner's rule is

$$\begin{array}{l} \quad \oplus/ \ @ \ \otimes / * \ @ \ tails \\ = \\ \quad (\oplus \mathtt{UNIT}(\otimes) \ @ \otimes) \not\rightarrow \mathtt{UNIT}(\otimes) \end{array}$$

for associative operators $\oplus$ and $\otimes$ provided

$$\forall(x, y, z :: (x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z))$$

holds. (The interpretation is not "known" to the editor, so it is not relevant to us here.) The applicability condition is similar in spirit to the condition on the range of a function, and that is why it is represented in the same way. The range indicates the condition under which the function can be applied to its parameters. The parameters are the operators $\oplus$ and $\otimes$ and the function body is

$$\oplus/ \ @ \ \otimes / * \ @ \ tails \quad = \quad (\oplus \texttt{UNIT}(\otimes) \ @ \otimes) \not\rightarrow \texttt{UNIT}(\otimes)$$

The body might also be written as a calculation (of two or more steps) and then the rule's body can be considered to be equal to the reduced version of that calculation (see section 3.1).

The complete rule for Horner's scheme is shown in the listing of the squiggol module. In the rules window, a shorthand version of the rule is printed. It consists of the function body, preceded by the rulke name only. If you want to see the full version of the rule that is presently selected (the one that is underlined), press the button on top of the rules window. It pops up a window with the rule in its complete form.

The main operator of a rule body need not be the equality operator. If a different operator is used, one should be careful in its application. Application of a rule to an expression extends a calculation with a hint, an operator, and a new expression. The operator is copied from the rule body. This may lead to "surprises" in what are sometimes called "negated" contexts. For example, rule

```
RULE twice : (a | a > 0 ▷ a < 2 * a)
```

applied to `x` in `-x` produces

```
   -x
<  { twice[a:=x] }
   -2*x
```

and not

```
   -x
>  { twice[a:=x] }
   -2*x
```

which would be slighty more correct. The editor attempts to verify that a rule is applied in a monotonic context, but these attempts are presently incomplete.

The squiggol module also contains examples of rules that have no parameters and, therefore, serve only as definitions.

When a module m is imported in another module n, then the scope of all declarations occurring in m extends to n. That is why we can use, for example, operator ++ in module `MaximumSegmentSum`. Also, all properties and rules declared in m extend to n. Declarations, properties, and rules that were imported into m via an import clause in m itself are not extended to n. When needed, n should import those directly.

9 The Definitions menu

The `Definitions` menu contains two buttons: pushing the `Operators` button produces a window with all operators that have been defined, and pushing the `Notations` button produces a window with all notations that have been defined. Either window is closed by pushing the `C` button in its left top corner.

The predefined operators are

.	function application	LUNIT=id, LASSOCIATIVE
@	function composition	UNIT=id, ASSOCIATIVE
~	boolean negation	
-	integer negation	
*	integer multiplication	UNIT=1,ZERO=0,ASSOCIATIVE,DUAL=*
/	integer division	RUNIT=1,LZERO=0,LASSOCIATIVE
\	integer remainder	LZERO=0,LASSOCIATIVE
+	integer addition	UNIT=0,ASSOCIATIVE,DUAL=+
-	integer subtraction	LASSOCIATIVE
/\	boolean conjunction	UNIT=true,ZERO=false,IDEMPOTENT,ASSOCIATIVE,DUAL= /\
\/	boolean disjunction	UNIT=false,ZERO=true,IDEMPOTENT,ASSOCIATIVE,DUAL= \/
=	equality	DUAL = =,CONJUNCTIVE,TRANSITIVE
#	inequality	DUAL = #,CONJUNCTIVE
<	less than	DUAL = >,CONJUNCTIVE,TRANSITIVE
<=	at most	DUAL = >=,CONJUNCTIVE,TRANSITIVE
>	greater than	DUAL = <,CONJUNCTIVE,TRANSITIVE
>=	at least	DUAL = <=,CONJUNCTIVE,TRANSITIVE
-->	implication	DUAL = <--,CONJUNCTIVE,TRANSITIVE,LUNIT=true
<--	explication	DUAL = -->,CONJUNCTIVE,TRANSITIVE,RUNIT=true
ALL	universal quantification	INFIXOPERATOR= /\ ,CONSTRUCTOR=id

There are no predefined notations.

10 The Modify EDIT menu

The `Modify EDIT` menu offers nine buttons. This part of the interface is sometimes rather clumsy, and will hopefully (make that definitely) be improved. The nine buttons and their functions are:

- `add EDIT`
  adds an `EDIT` item to the module in the edit window
- `delete EDIT`
  deletes the `EDIT` item containing the focus from the module
- `EDIT to RULE`
  changes the status of an `EDIT` item to a `RULE`. This adds the item to the rule window.

- `EDIT to PROPERTY`

  changes the status of an `EDIT` item to a `PROPERTY`

- `EDIT to IMPORT`

  changes the status of an `EDIT` item to an `IMPORT`

- `EDIT to DECLARE`

  changes the status of an `EDIT` item to a `DECLARE`

- `RULE to EDIT`

- `PROPERTY to EDIT`

- `DECLARE to EDIT`

  these three have not yet been implemented

11 Appendix A

This appendix shows the complete text of the maximum-segment-sum example. It is identical to the text in Figure 4, but the font size is not as minuscule.

```
MODULE module
IMPORT squiggol
EDIT
  max@sum@segs
=   { definition }
  max@sum@((++)/)@(tails*)@inits
=   { definition }
  max@(((+)/)*)@((++)/)@(tails*)@inits
=   { definition }
  ((^)/)@(((+)/)*)@((++)/)@(tails*)@inits
=   { MapPromotion[f:=(+)/] }
  ((^)/)@((++)/)@((((+)/)*)*)@(tails*)@inits
=   { ReducePromotion[+:=^] }
  ((^)/)@(((^)/)*)@((((+)/)*)*)@(tails*)@inits
=   { factor }
  ((^)/)@(((^)/)@(((+)/)*)@tails*)@inits
=   { HornerScheme[+:=^,&:=+] }
  ((^)/)@(((^UNIT(+))@(+)-/->UNIT(+))*)@inits
=   { Accumulate[f:=(^UNIT(+))@(+),e:=UNIT(+)] }
  ((^)/)@((^UNIT(+))@(+)-//->UNIT(+))
=   { reduce }
  ((^)/)@((^0)@(+)-//->0)
```

12 Appendix B

This appendix lists two modules (`refine` and `sqrt`) and show how they can be used to derive a program for computing square roots by binary search. This example is almost straight from Carroll Morgan's book [3]. The example shows both weaknesses and strengths of the current editor, but we refrain from making comments about it now. The first module introduces the refinement calculus. Not all rules and properties are used in the example. The main thing that is lacking is a statement of the monotonicity properties of the statement constructors. One comment might be in order. I have written

```
NOTATION VAR !LIST BEGIN ! END
```

instead of

```
NOTATION VAR ?LIST(v) BEGIN !(v) END
```

because program variables are not exactly like mathematical variables. I don't know how to do this well (yet).

```
MODULE refine

NOTATION DO ! -> ! OD

NOTATION IF ! -> ! | ! -> ! FI

NOTATION SPEC !LIST : [ ! , ! ]

NOTATION ASSIGN ! := !

NOTATION VAR !LIST BEGIN ! END

DECLARE INFIX 0 ;

DECLARE skip

PROPERTY UNIT(;)=skip

PROPERTY ASSOCIATIVE(;)

RULE Assignment: (x, E, P, Q | P --> Q[x:=E] |>
    SPEC x: [P, Q]
  <=
    (ASSIGN x:=E))

RULE Assignment: (v, x, E, P, Q | P --> Q[x:=E] |>
```

```
    SPEC v, x: [P, Q]
  <=
    (ASSIGN x:=E))

RULE Block: (v, w, P, Q |>
    (* provided w (which may be a list) does not occur in v, P or Q *)
    SPEC v:[P, Q]
  =
    VAR w BEGIN SPEC v, w:[P, Q] END)

RULE StrengthenPost: (v, P, Q, R | R --> Q |>
    SPEC v: [P, Q]
  <=
    SPEC v: [P, R])

RULE StrengthenPost: (v, P, Q, R |>
    SPEC v: [P, Q]
  <=
    SPEC v: [P, Q /\ R])

RULE Semicolon: (v, P, Q, R |>
    SPEC v: [P, R]
  <=
    (SPEC v: [P, Q]; SPEC v: [Q, R]))

RULE IfStatement: (v, P, Q, b0, b1 |>
    SPEC v:[ P /\ (b0 \/ b1), Q]
  <=
    IF b0 -> SPEC v:[P /\ b0, Q] | b1 -> SPEC v:[P /\ b1, Q] FI)

RULE IfStatement: (v, P, Q, b |>
    SPEC v:[ P, Q]
  <=
    IF b -> SPEC v:[P /\ b, Q] | ~b -> SPEC v:[P /\ ~b, Q] FI)

RULE Loop: (v, b, inv |>
    SPEC v: [inv, inv /\ ~b]
  <=
    DO b -> SPEC v: [inv /\ b, inv] OD)

RULE TerminatingLoop: (v, b, inv, bf | inv /\ b --> bf>0 |>
    SPEC v: [inv, inv /\ ~b]
  <=
    DO b -> SPEC v: [inv /\ b /\ (bf=BF), inv /\ (bf<BF)] OD)
```

```
RULE ContractFrame: (v, w, P, Q |>
    SPEC v, w: [P, Q]
  <=
    SPEC v: [P, Q])

RULE RightDistr: (b0, s0, b1, s1, s |>
    (IF b0 -> s0 | b1 -> s1 FI ; s)
   =
    IF b0 -> s0; s | b1 -> s1; s FI)

RULE Skip: (v, P, Q | P --> Q |>
    SPEC v:[P, Q]
  <=
    skip)
```

The second module defines functions `Floor` and `Sqrt`, or at least the properties that are relevant here. For example, the `Floor` rule states a relation between integer `x` and real number `y`. Those types can presently not be indicated. The second rule specifies what in the fashionable jargon is known as a Galois connection between `Sqrt` and squaring. The two rules called `NotPromotion` should not really be part of this module, but they are. One might argue that they should be properties of the operators.

```
MODULE sqrt

RULE Floor: (x,y |> (x = Floor.y) = ( x <= y < x+1))

DECLARE INFIX 10 ^

RULE Sqrt:  (x,y | x>=0 /\ y>=0 |> (x <= Sqrt.y) = ( x^2 <= y))

PROPERTY    (x,y |> (x^2<y^2) = (x<y))

RULE NotPromotion: (x,y |> (x < y) = ~(y <= x))

RULE NotPromotion: (x,y |> (x = y) = ~(x # y))
```

And here is the module that is produced by importing the two modules above, stating that a program is required for setting integer `r` to the value `Floor.(Sqrt.s)` and then applying lots and lots of transformations. Below is the output of the editor. I have omitted the output related to verification conditions. (The editor prints questions when it fails to verify side conditions.) Also, I have added some white space and split up lines because they became too wide. Furthermore, I have used asterisks to indicate all the hints that I had to give as input. No other inputs were supplied (other than mouse clicks).

```
MODULE module
```

```
IMPORT refine
IMPORT sqrt
EDIT
  SPEC r:[true,r=Floor.(Sqrt.s)]
  ******************************
=   { Floor[x:=r,y:=Sqrt.s] }
  SPEC r:[true,r<=Sqrt.s<r+1]
=   { split conjunction }
  SPEC r:[true,(r<=Sqrt.s)/\(Sqrt.s<r+1)]
=   { NotPromotion[x:=Sqrt.s,y:=r+1] }
  SPEC r:[true,(r<=Sqrt.s)/\~(r+1<=Sqrt.s)]
=   { Sqrt[x:=r,y:=s] }
  SPEC r:[true,(r^2<=s)/\~(r+1<=Sqrt.s)]
=   { Sqrt[x:=r+1,y:=s] }
  SPEC r:[true,(r^2<=s)/\~((r+1)^2<=s)]
=   { NotPromotion[x:=s,y:=(r+1)^2] }
  SPEC r:[true,r^2<=s<(r+1)^2]
=   { Block[v:=r,w:=q,P:=true,Q:=r^2<=s<(r+1)^2] }
                ****
  VAR q BEGIN SPEC r,q:[true,r^2<=s<(r+1)^2] END
<=   { StrengthenPost[v:=(r,q),P:=true,Q:=r^2<=s<(r+1)^2,R:=q=r+1] }
                                                        ********
  VAR q BEGIN SPEC r,q:[true,(r^2<=s<(r+1)^2)/\(q=r+1)] END
=   { q=r+1 }
  VAR q BEGIN SPEC r,q:[true,(r^2<=s<q^2)/\(q=r+1)] END
<=   { Semicolon[v:=(r,q),P:=true,Q:=r^2<=s<q^2,R:=(r^2<=s<q^2)/\(q=r+1)] }
                                  *************
  VAR q BEGIN SPEC r,q:[true,r^2<=s<q^2];SPEC r,q:[r^2<=s<q^2,(r^2<=s<q^2)/\(q=r+1)] END
<=   { Semicolon[v:=(r,q),P:=true,Q:=r=0,R:=r^2<=s<q^2] }
                                  ******
  VAR q
  BEGIN SPEC r,q:[true,r=0];SPEC r,q:[r=0,r^2<=s<q^2];
        SPEC r,q:[r^2<=s<q^2,(r^2<=s<q^2)/\(q=r+1)]
  END
<=   { Assignment[v:=q,x:=r,E:=0,P:=true,Q:=r=0] }
                        **** ****
  VAR q
  BEGIN (ASSIGN r:=0);SPEC r,q:[r=0,r^2<=s<q^2];
        SPEC r,q:[r^2<=s<q^2,(r^2<=s<q^2)/\(q=r+1)]
  END
<=   { Assignment[v:=r,x:=q,E:=s+1,P:=r=0,Q:=r^2<=s<q^2] }
                        **** ******
  VAR q
  BEGIN (ASSIGN r:=0);(ASSIGN q:=s+1);
```

```
        SPEC r,q:[r^2<=s<q^2,(r^2<=s<q^2)/\(q=r+1)]
  END
=   { NotPromotion[x:=q,y:=r+1] }
  VAR q
  BEGIN (ASSIGN r:=0);(ASSIGN q:=s+1);
        SPEC r,q:[r^2<=s<q^2,(r^2<=s<q^2)/\~(q#r+1)]
  END
<=   { TerminatingLoop[v:=(r,q),b:=q#r+1,inv:=r^2<=s<q^2,bf:=q-r] }
                                                        *******
  VAR q
  BEGIN (ASSIGN r:=0);(ASSIGN q:=s+1);
        DO q#r+1->SPEC r,q:[(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF),(r^2<=s<q^2)/\(q-r<BF)] OD
  END
=   { Block[v:=(r,q),w:=m,P:=(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF),Q:=(r^2<=s<q^2)/\(q-r<BF)] }
                    ****
  VAR q
  BEGIN (ASSIGN r:=0);(ASSIGN q:=s+1);
        DO q#r+1->
          VAR m
          BEGIN SPEC r,q,m:[(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF),(r^2<=s<q^2)/\(q-r<BF)] END
        OD
  END
<=   { Semicolon[v:=(r,q,m),P:=(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF),
                Q:=(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF)/\(r<m<q),
                ******************************************
                R:=(r^2<=s<q^2)/\(q-r<BF)] }
  VAR q
  BEGIN (ASSIGN r:=0);(ASSIGN q:=s+1);
        DO q#r+1->
          VAR m
          BEGIN SPEC r,q,m:[(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF),
                           (r^2<=s<q^2)/\(q#r+1)/\(q-r=BF)/\(r<m<q)];
                SPEC r,q,m:[(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF)/\(r<m<q),(r^2<=s<q^2)/\(q-r<BF)]
          END
        OD
  END
<=   { Assignment[v:=(r,q),x:=m,E:=(r+q)/2,P:=(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF),
                          **** **********
                 Q:=(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF)/\(r<m<q)] }
  VAR q
  BEGIN (ASSIGN r:=0);(ASSIGN q:=s+1);
        DO q#r+1->
          VAR m
          BEGIN (ASSIGN m:=(r+q)/2);
```

```
              SPEC r,q,m:[(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF)/\(r<m<q),(r^2<=s<q^2)/\(q-r<BF)]
        END
      OD
  END
<=   { IfStatement[v:=(r,q,m),P:=(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF)/\(r<m<q),
                  Q:=(r^2<=s<q^2)/\(q-r<BF),b:=s<m^2] }
                                          ********
  VAR q
  BEGIN (ASSIGN r:=0);(ASSIGN q:=s+1);
        DO q#r+1->
          VAR m
          BEGIN (ASSIGN m:=(r+q)/2);
                IF s<m^2->SPEC r,q,m:[(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF)/\(r<m<q)/\(s<m^2),
                                      (r^2<=s<q^2)/\(q-r<BF)]
                |~(s<m^2)->SPEC r,q,m:[(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF)/\(r<m<q)/\~(s<m^2),
                                      (r^2<=s<q^2)/\(q-r<BF)]
                FI
          END
        OD
  END
<=   { Assignment[v:=(r,m),x:=q,E:=m,P:=(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF)/\(r<m<q)/\(s<m^2),
                           **** ****
                  Q:=(r^2<=s<q^2)/\(q-r<BF)] }
  VAR q
  BEGIN (ASSIGN r:=0);(ASSIGN q:=s+1);
        DO q#r+1->
          VAR m
          BEGIN (ASSIGN m:=(r+q)/2);
                IF s<m^2->ASSIGN q:=m
                |~(s<m^2)->SPEC r,q,m:[(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF)/\(r<m<q)/\~(s<m^2),
                                      (r^2<=s<q^2)/\(q-r<BF)]
                FI
          END
        OD
  END
<=   { Assignment[v:=(q,m),x:=r,E:=m,P:=(r^2<=s<q^2)/\(q#r+1)/\(q-r=BF)/\(r<m<q)/\~(s<m^2),
                           **** ****
                  Q:=(r^2<=s<q^2)/\(q-r<BF)] }
  VAR q
  BEGIN (ASSIGN r:=0);(ASSIGN q:=s+1);
        DO q#r+1->
          VAR m
          BEGIN (ASSIGN m:=(r+q)/2);
                IF s<m^2->ASSIGN q:=m|~(s<m^2)->ASSIGN r:=m FI
```

```
        END
      OD
END
```

References

[1] R.S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, NATO ASI Series F, pages 151–216. Springer-Verlag, 1988.

[2] P. Chisholm. Calculation by Computer: System Manual. Technical report, Eindhoven University of Technology, 1990.

[3] C. Morgan. *Programming from Specifications.* Series in Computer Science (C.A.R. Hoare, ed.). Prentice-Hall International, 1990.