Rapid Development of a Program Transformation System with Attribute Grammars and Dynamic Transformations

Harald Vogt, Aswin van den Berg, Arend Freije

RUU-CS-90-23
June 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Rapid Development of a Program Transformation System with Attribute Grammars and Dynamic Transformations

Harald Vogt, Aswin van den Berg, Arend Freije

Technical Report RUU-CS-90-23
June 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

Rapid development of a program transformation system with attribute grammars and dynamic transformations*

Harald Vogt Aswin van den Berg Arend Freije

Department of Computer Science, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
E-Mail: harald@cs.ruu.nl

Abstract

Using the attribute grammar based Synthesizer Generator a prototype program transformation system has been developed in four man-months. This is very fast, compared with the development-time of other program transformation systems. The prototype supports the construction and manipulation of equational algorithm proofs and making derivations interspersed with text. Its intended use is in writing papers on algorithm design, automatic checking of the derivation and providing mechanic help during the derivation.

The editor supports dynamic transformations: they can be inserted and deleted during an edit-session, which is currently not supported by the Synthesizer Generator. Also the applicability and direction of applicability of a dynamic transformation on a formula in the program derivation is indicated and updated incrementally. Dynamic transformations were, until now, never implemented in any other proof- or program transformation system. The prototype, including the dynamic transformations, was written as a pure attribute grammar.

1 Introduction

This paper describes a prototype program transformation system made in four man-months with the attribute grammar based Synthesizer Generator (SG) [RTD 83]. The prototype transformation system (the BMF-editor) supports the interactive derivation of equational algorithm proofs in the Bird-Meertens formalism (BMF) [Bird 87, MRT 86]. Doing a derivation in BMF means repeatedly applying transformations to BMF-formulas.

*This is a copy of the manuscript presented at the International Workshop on Attribute Grammars and their Applications (WAGA), Paris, September 19-21, 1990

For a BMF-editor to be of practical use, the user should be able to add transformations which are derived during the derivation so they can be reused further on in the derivation. The transformations supported by the SG, however, can be only entered at editor-specification time. Dynamic transformations can be entered and deleted during the edit-session. Furthermore, the applicability and direction of applicability of a dynamic transformation on a formula is indicated and updated incrementally. Dynamic transformations were, until now, never implemented in any other proof- or program transformation system.

The dynamic transformations are implemented with an attribute grammar. Insertion and deletion of a transformation is already implemented with an attribute grammar in the CSG proof editor [Reps & Alpern 84]. In the attribute grammar based Interactive Proof Editor (IPE) [Ritchie 88] the applicabilaty of a dynamic transformation can be shown on demand but not incrementally.

The use of an attribute grammar based system like the SG was the key to easy and fast development of the BMF-editor. First, because the SG generates a user interface and environment for free. Second, because BMF-formulas, dynamic transformations and the derivation itself are represented easily by attribute grammars.

The BMF-editor lies somewhere between a program transformation system and a computer supported system for equational or formal reasoning. The construction time of program transformation systems like the PROSPECTRA system [Prospectra 86], the KIDS system, the TAMPR system and the CIP-S system (for an overview see [Partsch & Steinbruggen 83]), was considerably longer because almost all systems were totally written by hand without using any tools. The construction time of computer supported systems for formal reasoning like LCF, NuPRL, the Boyer-Moore theorem prover and the CSG proof editor (for an overview see [Lindsay 88]), was in the most cases also considerably longer for the same reasons.

The complete BMF-editor, including the dynamic transformations, was written in 3700 lines of pure SSL (the attribute grammar specification language of the SG), without using any non-standard SSL constructions. Therefore, the system is easily portable to any machine capable of running the SG.

The BMF-editor has been motivated and stimulated by ongoing research in the STOP project (Specification and Transformations Of Programs). The aim of the project is to further the research into program specification and transformation.

Section 2 introduces BMF and shows a sample derivation in BMF . Section 3 discusses the components, the look and feel, the abstract syntax and the dynamic transformations of the BMF-editor. A big example of a derivation with the editor is presented at the end of section 3. Further suggestions for improving the editor are discussed in section 4. Finally, the conclusions are presented in section 5.

2 The Bird-Meertens Formalism (BMF)

BMF is a lucid proof style based upon equational reasoning. A derivation in BMF consists of first specifying an inefficient algorithm which must be transformed into an efficient algorithm, and then making small, correctness-preserving transformation steps using a library of rules, until the desired algorithm is derived. Each transformation step replaces (part of) a formula, by another formula.

For a BMF-editor to be of practical use it should be possible to intersperse text together with the development of the program. This is similar to the WEB-system described in [Knuth 84]. The difference with the WEB-system is that we want to derive programs from specifications using small correctness-preserving transformations, instead of the Dijkstra style of programming. By using a transformation system which contains a library of rules, it is possible to check whether our derivation is correct, thereby overcoming the proof obligation still present in the WEB-system. Furthermore, as in the WEB-system, it should be possible to filter the final program out of the file containing the text and the program derivation. Just transforming would then be the same as writing articles in the system without writing text.

Because we believe that proofs (or derivations) have to be engineered by a human rather than by the computer we insist on manual operation. Therefore, the program transformation system is a kind of an editor.

2.1 Some basic BMF

Here we present some basic BMF . In the following section we use this in a small derivation. This short introduction was inspired upon [Bird 87]. All operators work on lists, list of lists, or elements of lists (integers or lists). Lists are finite sequences of values of the same type. Enumerated lists will be denoted using square brackets. The primitive operation on lists is concatenation, denoted by the sign $+\!\!+$. For example:

$$[1] +\!\!+ [2] +\!\!+ [1] \quad = \quad [1,2,1]$$

The operator / (pronounced "reduce") takes a binary operator on its left and a list on its right and "puts" the operator between the elements of the list. For example,

$$+\!\!+/\ [[1],[2],[1]] \quad = \quad [1] +\!\!+ [2] +\!\!+ [1]$$

Binary operators can be *sectioned*. For example $(\oplus 1)$ denotes the function

$$(\oplus 1)2 \quad = \quad 2 \oplus 1$$

The brackets here are essential and should not be omitted.

The operator $*$ (pronounced "map") takes a function on its left and a list on its right and applies the function to all elements of its list. For example,

$$(\mathit{plus}\ 1)*[1,2,1] \quad = \quad [(\mathit{plus}\ 1)\,1, (\mathit{plus}\ 1)\,2, (\mathit{plus}\ 1)\,1]$$

Functional composition is denoted by a centralised dot $(\cdot)$.

2.2 A sample derivation

The following transformations are used in the forthcoming derivation:

$$\begin{array}{rcll} \mathit{lif} & == & (\mathit{plus}\,1)* \cdot +\!\!\!+/ & \{ \text{Definition of lif} \} \\ F* \cdot +\!\!\!+/ & == & +\!\!\!+/ \cdot F** & \{ \text{Map promotion} \} \end{array}$$

The first rule defines the function *lif*, which concatenates all sublist of the list and then increments all elements of the list by one.

The *map promotion rule* states that first concatenating lists and then mapping F is the same as first mapping F to all the sublists of the list and then concatenating the results. Here F plays the rôle of a *program-variable*, which can be bound to any BMF-formula.

The following (short) derivation states that the function *lif* can be computed by first concatenating all sublist(s) of the list ($+\!\!\!+/$) and then adding one to all elements of the resulting list ($(\mathit{plus}\,1)*$) or by first adding one to all the elements of the sublist(s) of the list ($(\mathit{plus}\,1)**$) and then concatenating the result ($+\!\!\!+/$). The names of the applied transformation rules are shown between braces.

$$\begin{array}{rcl} \mathit{lif} & = & \left\{ \text{Definition of lif} \right\} \\ & & (\mathit{plus}\,1)* \cdot +\!\!\!+/ \\ & = & \left\{ \text{Map promotion} \right\} \\ & & +\!\!\!+/ \cdot (\mathit{plus}\,1)** \end{array}$$

In each transformation step the *selected transformation* is applied on the *selected term*. For example, in the second step of the sample derivation the map promotion rule is the selected transformation and $(\mathit{plus}\,1)* \cdot +\!\!\!+/$ is the selected term.

3 The BMF-editor

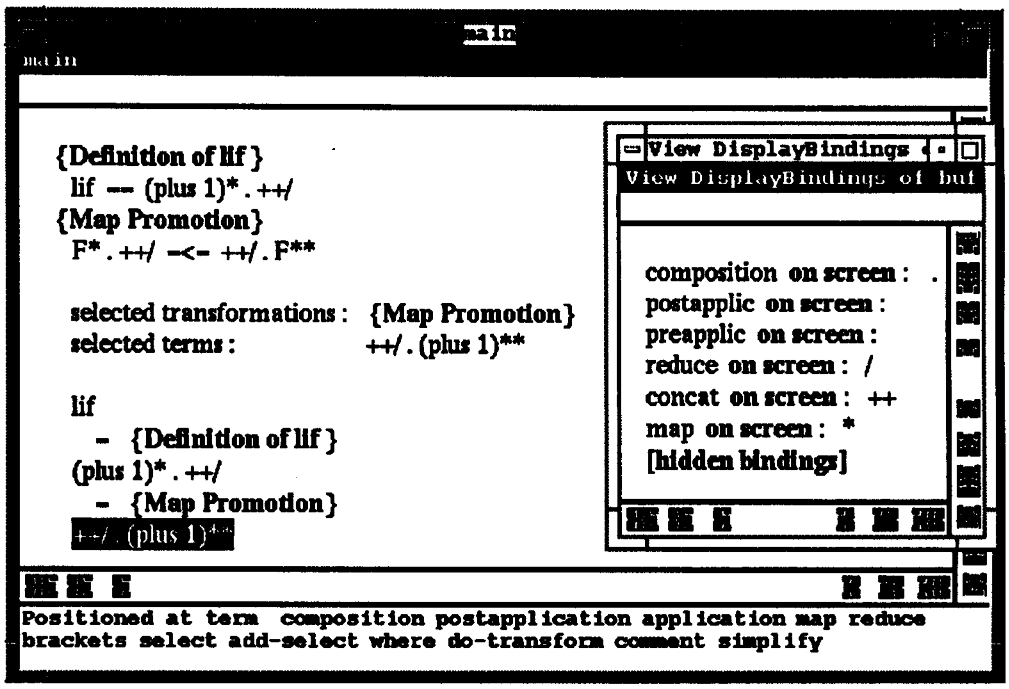
The BMF-editor supports the components used in the sample derivation. First these components will be discussed. Then the appearance to the user, the possible user actions, the abstract syntax of BMF-formulas and the system, and, finally, the dynamic transformations will be discussed.

A library of rules (the *dynamic transformations*) is supported and adding newly derived rules to this library is made simple. The direction in which (a subset of all) transformations are applicable on a newly selected (part of a) BMF-formula is updated incrementally and shown directly on the screen.

Just as in a written derivation, the system keeps track of the history of the derivation. Furthermore, it is possible to start a (different) subderivation anywhere in the tree. Therefore, a *forest of derivations* is supported, thus facilitating a trial and error approach to deriving algorithms.

Because the BMF-notation uses many non-ascii symbols, it is possible to select an arbitrary notation (e.g. LaTeX) as unparsing for the internal representation of a BMF-formula. For this purpose, the editor maintains an editable list of *displaybindings*.

3.1 Appearance to the user



```
main
main

{Definition of lif }
 lif == (plus 1)* . ++/
{Map Promotion}
 F* . ++/ =<= ++/ . F**

 selected transformations :  {Map Promotion}
 selected terms :            ++/ . (plus 1)**

 lif
   =  {Definition of lif }
 (plus 1)* . ++/
   =  {Map Promotion}
 ++/ . (plus 1)**
```

```
View DisplayBindings
View DisplayBindings of buf

 composition on screen :  .
 postapplic on screen :
 preapplic on screen :
 reduce on screen :  /
 concat on screen :  ++
 map on screen :  *
 [hidden bindings]
```

```
Positioned at term  composition postapplication application map reduce
brackets select add-select where do-transform comment simplify
```

Figure 1: The Base View and Display Bindings View of the sample derivation in the BMF-editor, the Display Bindings in the Base View are hidden.

The editor displays the definitions of the dynamic transformations and the derivation in almost the same order as in the sample derivation.

Transformations are shown as two BMF-formulas separated by an ==-sign. A transformation is preceded by its name. The direction in which a transformation is applicable on a BMF-formula is denoted by < and > signs in the ==-sign.

The *selected transformation* and the *selected term* are shown between the dynamic transformations and the forest of derivations.

Nodes in the derivation tree are labeled with BMF-formulas; the edges of the tree are marked with justifications. A justification is a reference to a transformation in the list of dynamic transformations. At all times only one path in the derivation tree is displayed. Left and right branches are indicated by ^-symbols.

A displaybinding is shown as the internal representation of the BMF-formula followed by the unparsing.

The dynamic transformation, selected transformation and term, the derivation and the displaybindings are shown on the **Base View** and **main** window. The dynamic transformations and the displaybindings in the Base View can be hidden by the user.

Beside the Base View, various other views on the main window are possible. There is one global cursor for all views. The following other views are available:

- **Transformations View** Displays all dynamic transformations.
- **Applicable Transformations View** Displays all the transformations that are applicable on a subterm of a selected term.
- **Transformable Terms View** Displays all (sub)terms in the whole derivation on which a selected transformation is applicable. These terms are shown together with the possible results of the transforming.
- **Display Bindings View** Displays all displaybindings.

Figure 1 shows the Base View and Display Bindings View of the sample derivation in the BMF-editor, the Display Bindings in the Base View are hidden.

3.2 User actions

A dynamic transformation can be inserted and deleted by edit-operations. A BMF-formula can be entered by structure editing or by typing the internal representation of a BMF-formula. There are shortcuts for frequently used BMF-constructions. For example, $f*$ is parsed correctly.

We will explain how to apply a transformation by doing the second transformation (map promotion) of the sample derivation. Commands to the system are given through built-in commands (SG-transformations), these will be indicated in **boldface** in the sequel of this section.

Before applying a transformation the user must duplicate (**dup**) the last BMF-formula in the derivation in order to keep the history of the derivation. Unfortunately, this must be done manually because the built-in SG-transformations do not allow to modify a tree which is not rooted by the node where the current cursor in the structure-tree is located.

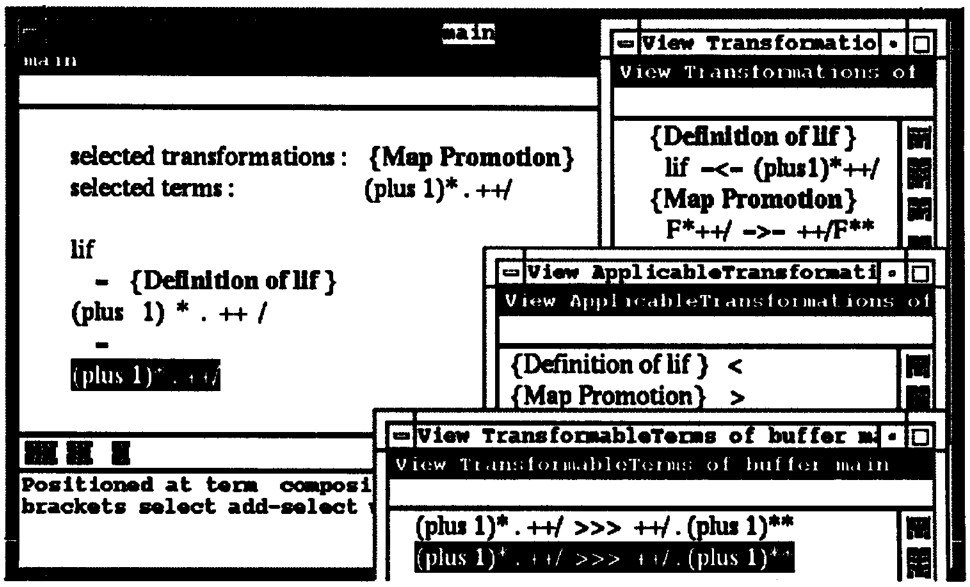
Then, the BMF-formula to be transformed is selected with the mouse and the **select** command. Now the system suggests which transformations are possible in the Transformations View or Applicable Transformations View. Because there is one global cursor for all views, clicking on one of the transformations in the Transformations View selects the corresponding transformation in the Base View. Selecting a dynamic transformation is done in the same way as selecting the term to be transformed. Both selections are shown as the selected transformation and the selected term. Figure 2 illustrates the situation before applying the map promotion rule.

Next, the transformation can be applied by giving the **do_transform** command. Figure 1 illustrates the situation after the transformation.

Several improvements on this scheme are implemented:

A set of dynamic transformations can be selected with the mouse and the **select** and **add_select** commands. Then, the system suggests which BMF-formulas in the derivation can be transformed with the selected transformations by showing them in the Transformable Terms View. Clicking on a result in the Transformable Terms View automatically selects the transformable term in the Base View (the highlighted parts in Figure 2), then the **do_transform** command can be given. In case there are more transformations possible, the user is asked to choose one.

Analogously, a set of terms can be selected. The Transformations and Applicable Transformations View display all applicable transformations on this set. Then the user can choose which transformation should be applied.


main
main
View Transformatio
View Transformations of
View ApplicableTransformati
View ApplicableTransformations of
View TransformableTerms of buffer m
View TransformableTerms of buffer main

```
selected transformations :  {Map Promotion}
selected terms :            (plus 1)* . ++/

lif
  =  {Definition of lif}
(plus  1) * . ++ /
  =
(plus 1)* . ++/
```

```
Positioned at term  composi
brackets select add-select
```

```
{Definition of lif}
  lif =<= (plus1)*++/
{Map Promotion}
  F*++/ =>= ++/F**
```

```
{Definition of lif}  <
{Map Promotion}  >
```

```
(plus 1)* . ++/ >>> ++/ . (plus 1)**
(plus 1)* . ++/ >>> ++/ . (plus 1)**
```

Figure 2: The sample derivation before applying map promotion and after duplication of the last BMF-formula in order to keep the history of the derivation. Note the various Views.

Other available commands are:

- **simplify** Simplify a BMF-formula (including removal of redundant brackets).
- **new_right, new_left, right** and **left** Focus on the (new) subderivation on the right or left and continue with a (different) subderivation.

- **comment** Insert text between derivation steps.

A displaybinding can be entered by giving ascii-symbols or their integer-values and choosing a suitable (LaTeX) font using SG-transformations.

Parts of the dynamic transformations, the derivation and the displaybindings can be saved and loaded with the built-in save and load facilities of the SG.

3.3 The abstract syntax

We have chosen a compact and uniform abstract syntax for BMF-formulas. The compact representation of BMF-formulas was necessary to minimize the attribution rules for the pattern-matching and program-variable binding in the BMF-formulas.

There is only one representation for BMF-formulas containing operators. For example, $a + b + c$ is represented as $(+, [a, b, c])$; the *infix* operator followed by a list of operands.

All operators in BMF are represented by infix operators in the grammar. In BMF three types of operators can be distinguished; *prefix*, *postfix* and *infix* operators. The *prefix* application $f x$ can be seen as the *infix* application

$$f\,preapplic\,x$$

where *preapplic* is the *infix* operator that applies its left operand to its right operand. Analogously, the *postapplic infix* operator can be defined.

There is no difference between operands and operators, they are both represented by Terms. A Term is described by the following production rules:

```
Term      ::=  TermConst
            |  TermVar
            |  ( Term, [ TermList ] ) ;
TermList  ::=  NoTerm
            |  Term, TermList ;
```

A Term can be a standard-term (*preapplic*, *postapplic*, *composition*, *map*, *reduce* and *list*) or a user-defined term, both described by TermConst, or a program-variable matching any term (TermVar). Program-variables start with an uppercase letter, standard and user-defined terms with a lowercase letter. Associated with each Term are fixed priorities. The terms *composition*, *map* and *reduce* denote the corresponding notion in BMF. The last term, *list*, is used to represent the lists of BMF.

As an example, the internal representation of $+\!\!+/$ is:

$$(postapplic, [+\!\!+, /])$$

In order to achieve the correct unparsing of this simple representation into BMF-notation, special unparsing rules for the standard terms are defined. For example:

$(\mathit{preapplic}, [f, x])$	is unparsed as	fx
$(\mathit{postapplic}, [f, *])$	is unparsed as	$f*$
$(\cdot, [f, g, h])$	is unparsed as	$f \cdot g \cdot h$
$(\mathit{list}, [1, 2, 1])$	is unparsed as	$[1, 2, 1]$

The root-production of the system is now as follows:

```
BMF-editor  ::=  TransList
                 Derivation
                 DisplayList ;
```

TransList represents the list of dynamic transformations, DisplayList represents the editable list of displaybindings of terms.

A dynamic transformation, named Label, is described by the following production:

```
Trans  ::=  { Label }
            Term == Term ;
```

A derivation is a list of terms separated by =-signs and the names of the transformation applied:

```
Derivation  ::=  Term
            |    Term
                 = { Label }
                 Derivation ;
```

In the actual implementation a more complicated grammar is used for the tree-structure of derivations and for the possibility to add comment in derivations.

3.4 Dynamic transformations

Transformations in the SG can be defined only at editor-specification time. Dynamic transformations can be entered and deleted at editor-run-time. Just as for standard SG-transformations the applicability of a dynamic transformation is computed incrementally.

In the Prospectra project [Prospectra 86] a brute force approach was taken. After adding a new transformation the complete *Prospectra Ada/Anna subset* editor was regenerated.

Our prototype emulates dynamic transformations using standard SSL attribute computation. This emulation will be explained hereafter.

As was said in section 3.3, a dynamic transformation consists of a name (Label) and a left hand side and right hand side pattern (Terms). A dynamic transformation is *applicable* on term T if the left hand side or the right hand side matches with term T.

For example the dynamic transformation

$$F* \cdot +\!\!+/ \quad == \quad +\!\!+/ \cdot F** \quad \{ \text{ Map promotion } \}$$

is applicable to the term

$$(\mathit{plus}\, 1)* \cdot +\!\!+/$$

which then can be transformed into

$$+\!\!+/ \cdot (\mathit{plus}\, 1)**$$

Note that the program-variable F is bound to $(\mathit{plus}\, 1)$.

The applicability test and actual application of a dynamic transformation to a term proceeds in four phases: pattern-matching, program-variable binding (both together are in fact unification), computation of the transformed term and replacement of the old term by the transformed term. Pattern-matching, program-variable binding and computation of the transformed term take place inside terms. The replacement of the old term by the transformed term takes place in the SG-transformation **do_transform** (see also section 3.2).

The first three phases (pattern-matching and program-variable binding and computation of the transformed term) require both the selected transformation and the selected term. To bring these together in an attribute grammar can be done in two complementary ways. Either the term to be transformed is inherited by the dynamic transformation or the dynamic transformation is inherited by the term to be transformed. Both ways are depicted in Figure 3.

The first way is used to compute the applicability direction: the selected term is an inherited attribute of the selected transformation. The second way is used to apply the selected transformation to the selected term: the selected transformation is an inherited attribute of the selected term. Also the Transformable Terms View is implemented in this way.

In order to keep the pattern-matching simple we do not take the associativity of operators into account. So the Term $1 \oplus H$ (represented as $(\oplus, [1, H])$) does not match with the Term $1 \oplus b \oplus c$ (represented as $(\oplus, [1, b, c])$). As a result, the match-time is linear in the size of the tree. Furthermore, a program-variable can be bound only once to another term.

Pattern-matching and computation of bindings use the inherited attribute *pat* and synthesized attributes *applic* and *bindings* of Term. A Term (the pattern-Term) is given as an inherited attribute to the Term it should match (the match-Term). A short description of each attribute is given.

- **pat** This attribute is used to distribute the pattern-Term over the tree representing the match-Term. Every node in this tree inherits that part of the pattern-Term it should match.

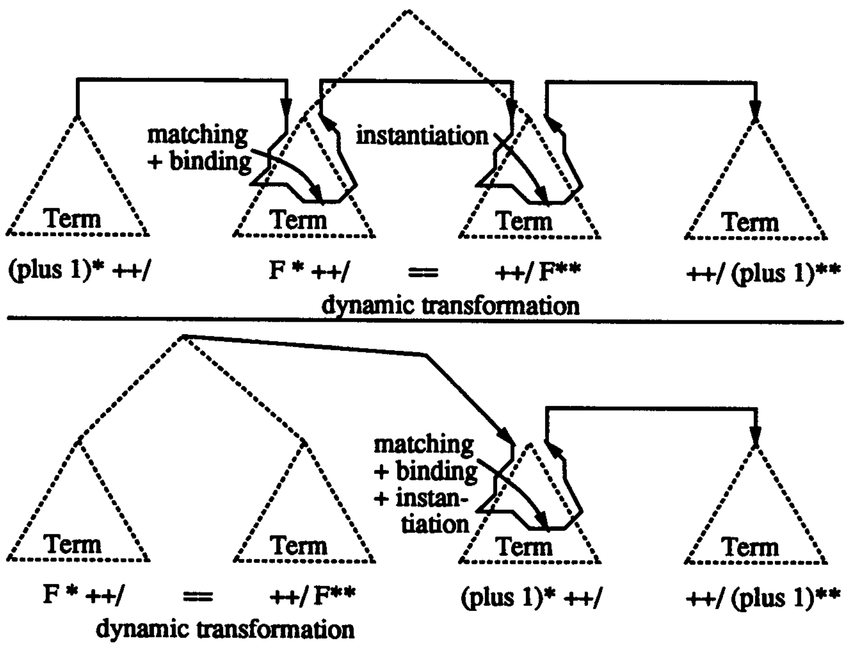


Figure 3: Two complementary ways of matching, binding of program-variables and computation of the transformed term.

- **applic** This boolean attribute is used to synthesize whether the pattern-Term matches. The top-most applic attribute in the tree representing the match-Term is *true* if all patterns in this tree match and there are no conflicting bindings.
- **bindings** This attribute contains the list of program-variable bindings.

3.5 A big example

This example, taken from [Bird 87], shows some steps in the derivation of an $O(n)$ algorithm for the *mss* problem. The *mss* problem is to compute the maximum of the sums of all segments of a given sequence of (possibly negative) numbers. This example illustrates the use of where-abstraction and conditions in the BMF-editor. The conditions are tabulated and automatically instantiated but not checked by the editor. First some definitions necessary to define *mss* are given.

The function *segs* returns a list of all segments of a list. For example,

$$segs[1,2,3] \;=\; [[],[1],[1,2],[2],[1,2,3],[2,3],[3]]$$

The *maximum* operator is denoted by $\uparrow$, for example

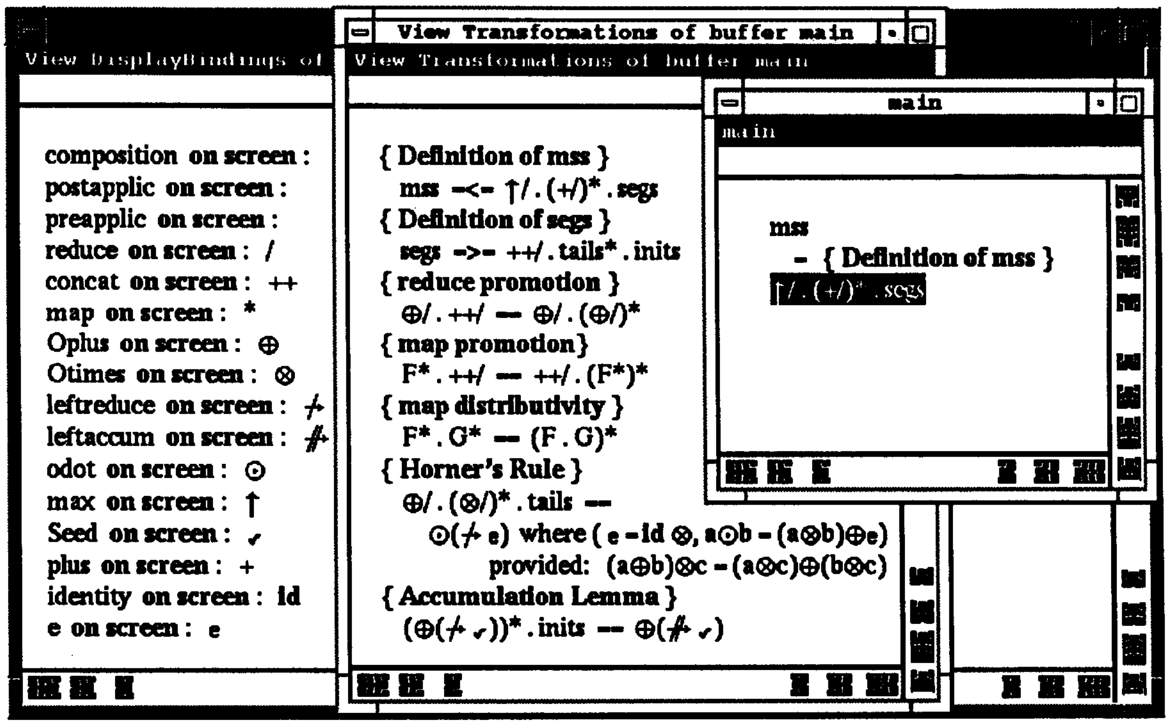
$$2\uparrow 4\uparrow 3 \;=\; 4$$

Now *mss* can be defined as follows

$$mss \;=\; \uparrow/ \cdot +/* \cdot segs$$

Direct evaluation of the right-hand-side of this equation requires $O(n^3)$ steps on a list of length n. There are $O(n^2)$ segments and each can be summed in $O(n)$ steps, giving $O(n^3)$ steps in all.

Without further explanation of the applied transformation rules we illustrate three situations in the derivation of a linear time algorithm for the *mss* problem. Figure 4 shows the start of the derivation together with all necessary displaybindings and transformations. Figure 5 illustrates the situation before applying Horner's rule. In Figure 6 the whole derivation is shown, note the instantiation of the where-abstraction and the conditions after applying Horner's rule.


View DisplayBindings of
View Transformations of buffer main
View Transformations of buffer main
main
main

```
composition on screen :
postapplic on screen :
preapplic on screen :
reduce on screen : /
concat on screen : ++
map on screen : *
Oplus on screen : ⊕
Otimes on screen : ⊗
leftreduce on screen : ↛
leftaccum on screen : ⇸
odot on screen : ⊙
max on screen : ↑
Seed on screen : ✓
plus on screen : +
identity on screen : id
e on screen : e
```

```
{ Definition of mss }
  mss =<= ↑/.(+/)*.segs
{ Definition of segs }
  segs =>= ++/.tails*.inits
{ reduce promotion }
  ⊕/.++/ == ⊕/.(⊕/)*
{ map promotion}
  F*.++/ == ++/.(F*)*
{ map distributivity }
  F*.G* == (F.G)*
{ Horner's Rule }
  ⊕/.(⊗/)*.tails ==
     ⊙(↛ e) where ( e = id ⊗, a⊙b = (a⊗b)⊕e)
            provided: (a⊕b)⊗c = (a⊗c)⊕(b⊗c)
{ Accumulation Lemma }
  (⊕(↛ ✓))*.inits == ⊕(⇸ ✓)
```

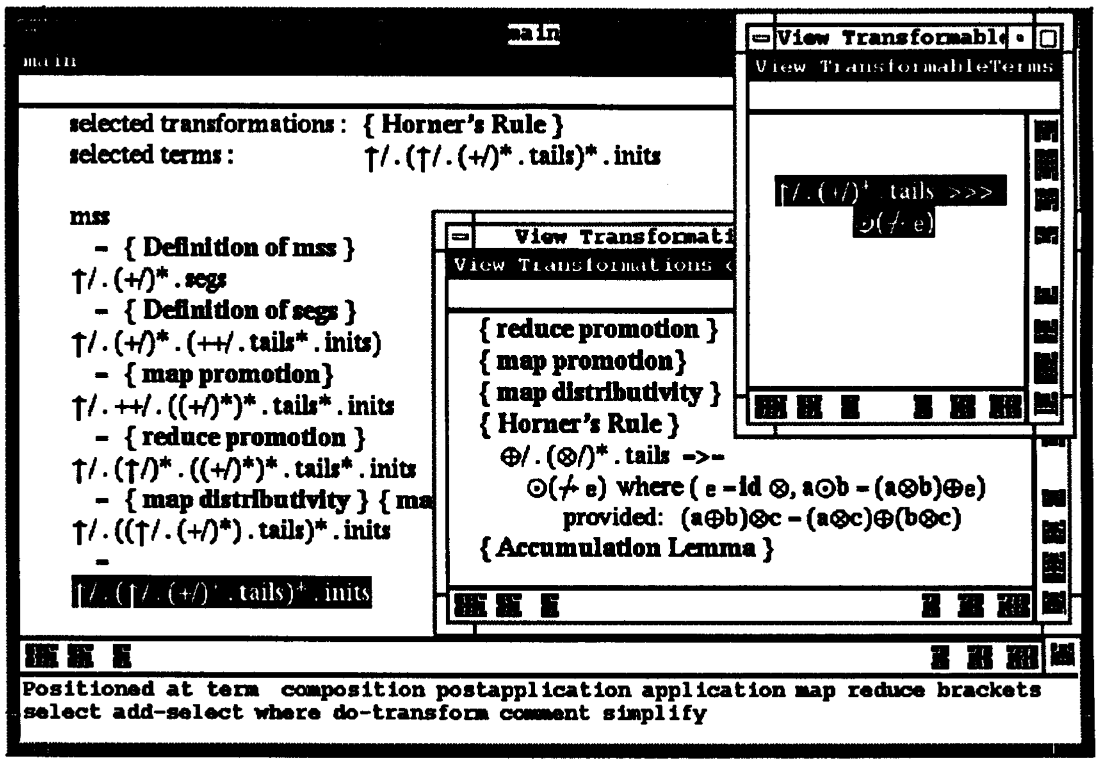
```
mss
  = { Definition of mss }
↑/.(+/)*.segs
```

Figure 4: The definition of *mss* and all necessary transformations and displaybindings for the derivation of a linear time algorithm for *mss*.

The last formula ($\uparrow/ \cdot \odot \not\!\!\!\#\!\!\!\!\to e$) is a maximum reduce composed with a *left-accumulation*. Left accumulation is expressed with the operator $\not\!\!\!\#\!\!\!\!\to$. For example,

$$\odot \not\!\!\!\#\!\!\!\!\to e\,[a_1, a_2, \ldots, a_n] \;=\; [e, e \odot a_1, \ldots, ((e \odot a_1) \odot a_2) \odot \ldots \odot a_n]$$

The maximum reduce composed with the left-accumulation can be easily translated into the following loop in an imperative language. Using hopefully straightforward notation, the value $\uparrow/ \cdot \odot \not\!\!\!\#\!\!\!\!\to e$ is the result delivered by the following imperative program ($a \odot b = (a + b) \uparrow 0$):



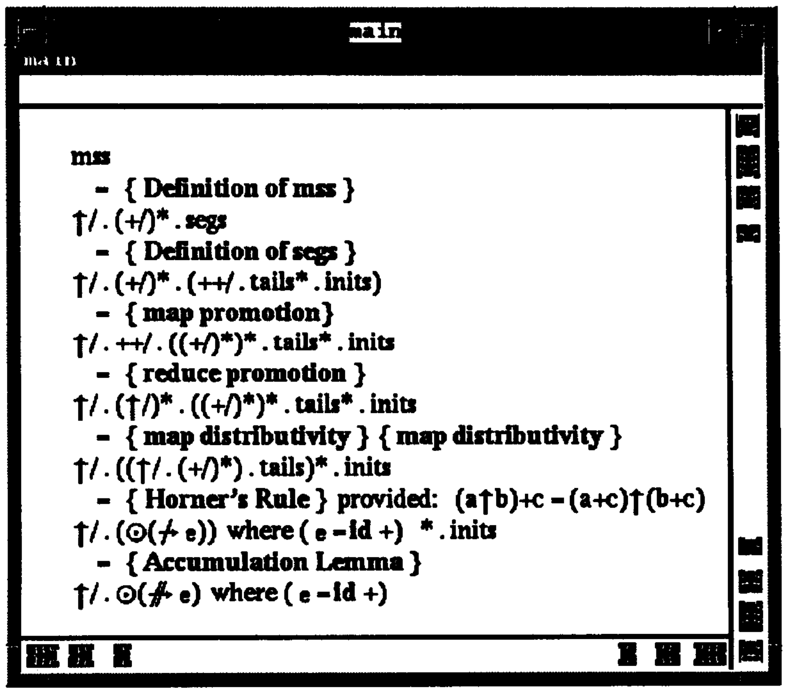
```
main
main

selected transformations : { Horner's Rule }
selected terms :           ↑/ . (↑/ . (+/)* . tails)* . inits

mss
  = { Definition of mss }
↑/ . (+/)* . segs
  = { Definition of segs }
↑/ . (+/)* . (++/ . tails* . inits)
  = { map promotion}
↑/ . ++/ . ((+/)*)* . tails* . inits
  = { reduce promotion }
↑/ . (↑/)* . ((+/)*)* . tails* . inits
  = { map distributivity } { ma
↑/ . ((↑/ . (+/)*) . tails)* . inits
  =
↑/ . (↑/ . (+/)* . tails)* . inits
```

```
View Transformati
View Transformations c

{ reduce promotion }
{ map promotion}
{ map distributivity }
{ Horner's Rule }
  ⊕/ . (⊗/)* . tails  =>=
    ⊙(↛ e) where ( e = id ⊗, a⊙b = (a⊗b)⊕e)
      provided:  (a⊕b)⊗c = (a⊗c)⊕(b⊗c)
{ Accumulation Lemma }
```

```
View Transformable
View TransformableTerms

↑/ . (+/)* . tails >>>
⊙(↛ e)
```

```
Positioned at term  composition postapplication application map reduce brackets
select add-select where do-transform comment simplify
```

Figure 5: The situation before applying Horner's rule.



```
main
main

mss
  = { Definition of mss }
↑/ . (+/)* . segs
  = { Definition of segs }
↑/ . (+/)* . (++/ . tails* . inits)
  = { map promotion}
↑/ . ++/ . ((+/)*)* . tails* . inits
  = { reduce promotion }
↑/ . (↑/)* . ((+/)*)* . tails* . inits
  = { map distributivity } { map distributivity }
↑/ . ((↑/ . (+/)*) . tails)* . inits
  = { Horner's Rule } provided:  (a↑b)+c = (a+c)↑(b+c)
↑/ . (⊙(↛ e)) where ( e = id +)  * . inits
  = { Accumulation Lemma }
↑/ . ⊙(⇻ e) where ( e = id +)
```

Figure 6: The whole derivation of a linear time algorithm for the *mss* problem. Note the instantiation of the where abstraction and the conditions after applying Horner's rule.

```
int a,b,t;
a := 0; t := 0;
for b in x
do a := max(a+b,0);
   t := max(t,a)
od
return t
```

4 Further suggestions

It should be possible to generate a LaTeX document by combining the comments and the derivation. Also program-code (for example Miranda) should be generated from the derivation. A first attempt of implementing both features is already done using the same technique as was used for the displaybindings.

Incremental type checking and consistency checking of the derivation (for example after deletion of a transformation) should be performed. The dynamic transformations now only use pattern-matching. The dynamic transformations could be extended to conditional and parameterized dynamic transformations (see also [Santos 88]).

A good way of organizing theories for making large derivations should be found, instead of one big theory. At edit-time, some complexity-measure of an algorithm should be indicated and updated incrementally.

5 Conclusion

A prototype program transformation system for BMF has been developed in four man-months with the attribute grammar based SG. The use of an attribute grammar based system has significantly speeded up the building of such a complex system. Dynamic transformations, which provide insertion and deletion of a transformation during an edit-session, are a great help for making derivations in an interactive program transformation system. Dynamic transformations are particular useful, because their applicability can be indicated and updated incrementally. Dynamic transformations were, until now, never implemented in any proof- or program transformation system.

Acknowledgements

Matthijs Kuiper, Jeroen Fokker and Johan Jeuring commented on previous versions of this paper. Finally, we would like to thank Doaitse Swierstra for the support while working on the BMF-editor.

References

[Bird 87] Bird, R. *An introduction to the theory of lists.* Logic of Programming and Calculi of Discrete Design (M. Broy,ed.), NATO ASI Series Vol. F.36, Springer Verlag 1987.

[Knuth 84] Knuth, D.E. *Literate Programming.* The Computer Journal, Vol. 27, 1984.

[Lindsay 88] Lindsay, P.A. *A survey of mechanical support for formal reasoning.* Software Engineering Journal, January 1988.

[MRT 86] Meertens, L.G.L.T. *Algorithmics – towards programming as a mathematical activity.* In: de Bakker, J.W., Hazewinkel, M., Lenstra, J.K. (eds.), Proc. CWI Symposium on Mathematics and Computer Science, CWI Monographs Vol. 1, 1986.

[Partsch & Steinbruggen 83] Partsch, H. and R. Steinbruggen. *Program Transformation Systems.* Computing Surveys, Vol. 15, No.3, September 1983.

[Prospectra 86] Krieg-Brückner, B., B. Hoffmann, H. Ganzinger, M. Broy, R. Wilhelm, U. Möncke, B. Weisberger, A. McGettrick, I.G. Campbell and G. Winterstein. *PROgram development by SPECification and TRAnsformation.* Proc. ESPRIT Conf. 86, North-Holland 1987.

[Reps & Alpern 84] Reps, T. and B. Alpern. *Interactive Proof Checking.* In the 11th Ann. ACM Symp. on Principles Of Programming Languages, pages 36-45, 1984.

[RTD 83] Reps, T., T. Teitelbaum and A. Demers. *Incremental Context-Dependent Analysis for Language Based Editors.* In ACM Transactions on Progr. Lang. and Systems, Vol. 5, No. 3, pages 449-477, July 1983.

[Ritchie 88] Ritchie, B. *The Design and Implementation of an Interactive Proof Editor.* Tech. Rep. CSF-57-88 and PhD. dissertation, Dept. of Computer Science, Univ. of Edinburgh, Oct. 1988.

[Santos 88] Santos, R.G. *Conditional and parameterized transformations in CSG.* PROSPECTRA Study Note S.1.5.C2-SN-2.0, 1988-24-5.