Program Transformations using ASF+SDF

M.G.J. van den Brand[1], S.M. Eijkelkamp[1], D.K.A. Geluk[1], H. Meijer[2], M.J.F. Polling[2] and H.R. Osborne[2]

[1]Programming Research Group, University of Amsterdam,
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands,
email: {markvdb,eijkelka,geluk}@fwi.uva.nl
[2]University of Nijmegen, Toernooiveld 1, NL-6525 ED
Nijmegen, The Netherlands,
email: {hans,manuelp,hugh}@cs.kun.nl

Abstract. This paper describes the ongoing work in the design and implementation of a program transformation system for educational purposes. The system is based on ASF+SDF for the implementation of the application of transformation rules and on SEAL for the implementation of the user interface. The transformation rules are given in a mixture of CIP-L and CLEAN: DISCIPLEAN. The DISCIPLEAN transformation rules are not 'hard-wired' in the system but they are stored in a kind of data base and they are available via an editor. This allows a flexible way of extending the set of transformation rules.

1 Introduction

The research area of Software Engineering studies the methods and tools applicable to the development of correct and efficient programs from problem specifications. In this paper the problem specification is assumed to be the formal outcome of a requirements engineering process [Par91] applied to an informal request from some customer.

There are many ways to obtain a correct and efficient program from a formal specification. One may (systematically or otherwise) 'invent' the program and prove it correct, or consider the specification as a postcondition to be established and develop the program from it, using some assertional calculus [Gri81]. In transformational programming a program or algorithm is derived from a formal problem specification —which may in general be descriptive, i.e. non-operational— by a series of applications of correctness-preserving transformation rules. Apart from numerous elementary rules there may be rules for the replacement of a quantifier by a suitable search-function, for the transformation from arbitrary recursion to tail-recursion and many others. In a sense, the repertoire of transformation rules represents a 'programming knowledge base'. The research in this area focusses on methodology, new transformation rules, strategies in which several rules are combined and tools for transformational programming.

There are many schools advocating a particular style of transformational programming. For instance, KIDS [Smi90] focusses on programming paradigms such as divide-and-conquer. The user chooses a program part and a paradigm, upon which the system more or less automatically develops that part along the lines of that paradigm. Another school is based on the Bird-Meertens Formalism [STO92]. In BMF, a specification is an executable, albeit generally very inefficient, program. Vogt [Vog93] developed a prototype system for transformational programming in BMF with (equational) verification, as a demonstration of Higher-Order Attribute Grammars.

In this work we adopt the framework of STOP [Par90a], which is an offspring of the Munich CIP-project [BBB+85]. In CIP (Computer-aided Intuition-guided

Proceedings of ASF+SDF95. A workshop on Generating Tools from Algebraic Specifications. May 11 & 12, 1995, CWI, Amsterdam, M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman & E. Visser (eds.) Technical Report P9504, Programming Research Group, University of Amsterdam

Programming) an *object language* called CIP-L is used in which one may express a functional or even descriptive specification. This allows for specifications which are as abstract as possible, and more abstract than those in, e.g., BMF. The specification is subsequently transformed into an equivalent (possibly very low-level) iterative, procedural program by applying transformation rules.

As an example,

$$f(x) = \min\{y : \text{prime}\, y \wedge x \leq y\}$$

specifies a program f with input x yielding the smallest prime number not smaller than x.

Transformation rules are instances of the statement

$$\vdash \mathcal{C}_1, \ldots, \mathcal{C}_n \Longrightarrow \mathcal{I} \supseteq \mathcal{O}$$

where $\mathcal{I}$ and $\mathcal{O}$ are program schemes in the object language (terms with instantiable variables) and the $\mathcal{C}_i$ are predicates over such schemes. The rule states that some instantiation of the input scheme $\mathcal{I}$ may be replaced by the output scheme $\mathcal{O}$ (likewise instantiated) provided that the application conditions $\mathcal{C}_i$ (likewise instantiated) hold. Note that the $\mathcal{C}_i$ are proof obligations for the *user* of the rule, whereas the symbol $\vdash$ represents a proof obligation for the *provider* (the inventor) of the rule. In CIP-L, the denotation (semantics) of an expression is the set of its 'non-deterministically' possible values. Therefore, the provider must prove that (assuming the $\mathcal{C}_i$) each value of $\mathcal{O}$ is a value of $\mathcal{I}$, under each possible instantiation. If $\mathcal{I} \supseteq \mathcal{O}$ and $\mathcal{O} \supseteq \mathcal{I}$ we write $\mathcal{I} \equiv \mathcal{O}$.

As a first step towards an operational version of f specified above, one may apply the transformation rule 'case introduction'

$$\vdash \mathrm{B}_1 \vee \cdots \vee \mathrm{B}_k \Longrightarrow \mathrm{E} \equiv \textbf{if}\ \mathrm{B}_1\ \textbf{then}\ \mathrm{E}\ [] \cdots []\ \mathrm{B}_k\ \textbf{then}\ \mathrm{E}\ \textbf{fi}$$

where the B_i and E are program scheme variables. Taking $k = 2$, 'prime x' for B_1, '$\neg$(prime x)' for B_2 and '$\min\{y : \text{prime}\, y \wedge x \leq y\}$' for E one obtains

$$\begin{aligned} f(x) = \quad & \textbf{if}\ \text{prime}\, x \\ & \textbf{then}\ \min\{y : \text{prime}\, y \wedge x \leq y\} \\ & \textbf{else}\ \min\{y : \text{prime}\, y \wedge x \leq y\} \\ & \textbf{fi} \end{aligned}$$

There are further elementary transformation rules (akin to case introduction) which allow one to transform this into

$$\begin{aligned} f(x) = \quad & \textbf{if}\ \text{prime}\, x \\ & \textbf{then}\ \min\{y : \text{prime}\, y \wedge x \leq y\} \\ & \textbf{else}\ \min(\{y : \text{prime}\, y \wedge x = y\} \cup \{y : \text{prime}\, y \wedge x < y\}) \\ & \textbf{fi} \end{aligned}$$

which, by possibly many elementary transformations may be simplified to

$$\begin{aligned} f(x) = \quad & \textbf{if}\ \text{prime}\, x \\ & \textbf{then}\ x \\ & \textbf{else}\ \min(\emptyset \cup \{y : \text{prime}\, y \wedge x + 1 \leq y\}) \\ & \textbf{fi} \end{aligned}$$

from which by 'folding' a recursive version of f may be obtained:

$$f(x) = \textbf{if}\ \text{prime}\, x\ \textbf{then}\ x\ \textbf{else}\ f(x+1)\ \textbf{fi}$$

Some applicability conditions are purely syntactical, others refer to the language's semantics and must be separately proven. As indicated by the name CIP, it is imperative that the transformational development is computer-supported. Typically, the user enters a specification, written in the object language and repeatedly (a) selects a part-to-be-transformed and (b) chooses a transformation rule from the available repertoire. Many variations are possible: the system may suggest the parts to be selected, it may, for each selection, show the (syntactically) applicable rules, it may enable backtracking from dead alleys encountered in the development, etc. An ample such wish list may be found in [BvdBvD+90].

The School of Informatics at the University of Nijmegen, which has a course on transformational programming based on the language CIP-L [BBB+85] defined in the CIP-project, explored the 'market' in the first half of 1994 [OL], to find that there was no useable, readily available support system, not even CIP-S [BEH+87], the system being developed in the CIP-project. However, it became clear that it would be relatively straightforward to construct such a system by means of the ASF+SDF Meta-environment [Kli93] as developed in the Programming Research Group of the University of Amsterdam and CWI.

Meanwhile it was decided to make a transition to a simpler object language in the School's transformational programming course. The language Clean [PvE93], being developed in the Computing Science Institute of the University of Nijmegen and well known to the Nijmegen students, was chosen with the understanding that some constructs were irrelevant for the purpose of transformational programming and that a few language constructs should be added, in particular for the formulation of descriptive specifications. This extended subset of Clean, which has a few constructs borrowed from CIP-L, is named DISCIPLEAN, for the time being.

Subsequently, a master's thesis project was defined to actually realize the system. It comprises several subtasks: the precise definition of DISCIPLEAN, the formulation of transformation rules in DISCIPLEAN, the description of the language and the transformation rules in ASF+SDF and the design of a user interface. This paper gives an overview of this project and some of its results.

The prototype will not yet support the verification of application conditions connected to the individual transformation rules. It will log the application conditions so that the user of the transformation tool can give the proof 'by hand'.

The project may have a follow-up, where a proof system is added to support fulfilling the proof obligations that are generated by many transformation rules. A still further follow-up could explore the similarities with e.g. computer algebra systems.

2 The language

While the program transformation system should, in principle, be language independent, practical considerations of the formulation and implementation of transformation rules impose restrictions on the choice of language. There are also more fundamental requirements, needed to ensure that the language will be suited at all to transformational programming. Briefly, the two major requirements are that the language be descriptive —it must be possible to give a good problem description at an abstract level without language or implementation dependent details; and that the language contains quantors and nondeterminism, enabling constructs such as **some** x: P(x).

Less essential but desirable features are that the language has a simple syntax, that the language is extensible, and that an implementation exists (for didactic purposes) of at least the "back end" of the transformations —i.e. of that subset of the language in which the final product of transformations will lie.

The language CIP-L, being specially designed for transformational programming, has these features. The type system of CIP-L, however, is rather baroque, at the cost of the legibility of programs in CIP-L. CIP-L was also designed with transformations in mind that had a lower level target language than that in which we are interested, and thus contains many features which are superfluous to our requirements.

Most, if not all, functional languages also satisfy most of the requirements, with the exception of the existence of quantors and nondeterminism. Practical considerations have led to the choice of Clean as basis for the specification language. These considerations are:

- close contacts with the research group responsible for the development of Clean, thus facilitating feedback between our project and the Clean research group; and
- the existence of an easily accessible and efficient implementation of Clean.

The next section describes the extended subset of Clean used in this project. It should be emphasised that the choice of Clean is in no way an essential aspect of the transformation system. There are, indeed, longer term plans to move to a higher level formalism such as Funmath [Bou].

2.1 DISCIPLEAN syntax

Although both Clean and the applicative part of CIP-L are functional languages, they differ in many ways. Apart from the semantics of both languages —lazy functional graph rewriting versus strict functional evaluation with a set of possible result values for each construct— the constructs available to the programmer are also very different.

Clean is a typical functional language in that it allows function definitions with patterns and guards for alternatives. Additionally the language has a few special constructs and a very rich type system with a lot of predefined types. CIP-L has declarative constructs such as non-deterministic choice between a finite number of alternatives, a number of different quantifiers and set comprehension. Clean lacks these, and this is an essential shortcoming since they play an important rôle in the paradigm of transformational programming.

A compromise is chosen to overcome these differences. A dialect of Clean, DISCIPLEAN, is established that has a Clean-like syntax but also incorporates the declarative constructs of CIP-L. The most important of these are universal and existential quantification and *comprehensive choice* i.e., **some** x: P(x). This means that functions can be specified in the usual functional way and that one is not required to specify a type with each identifier. Although deliberately kept small and simple at first, it is our intention to add as much of Cleans type system to DISCIPLEAN as possible, but it will probably be too much to include advanced concepts such as type classes.

The current syntax of DISCIPLEAN has a number of declarative constructs borrowed from CIP-L. Constructs of Clean that have an equivalent in CIP-L were maintained in DISCIPLEAN. The list below is a summary of all DISCIPLEAN constructs. The complete syntax of DISCIPLEAN can be found in the Appendices B and C.

- **function definition**: For example

  ```
  fib 1 = 1;
  fib n | n==2 = 1;
  fib n = fib (n-1) + fib (n-2);
  ```

- **local declaration/strict let**: There is only one kind of local declaration supported in the current version of DISCIPLEAN. For example

let `x = 1` **in** `fib x.`

- **quantified expressions**: The left hand side can be a pattern, but it may not contain constants. For example

  forall `x :  x < 2.`
  exists `(x,y,z) :  x*x+y*y == z*z.`
  some `x :  issorted(x).`
  that `n :  n*1 == 0.`

- **finite choice**: Represents a non-deterministic choice between a number of alternatives. For example

  `1 [|] 2 [|] 3`

- **conditional**: The unavoidable if-then-else construction. For example

  if `(x == 1) 2 3`

- **denotations for predefined types**: Tuples, maps, sets and sequences are predefined. For example

  `(1,x,f)` is a tuple;
  {| `x :  issorted x` |} is a set;
  < `5, 6, 7` > is a map; and
  `[1, 2, 3]` and `[ ]` are sequences.

- **operations on predefined types and constants**: The language is completed by the predefined basic types boolean and integer, the operators —both monadic and dyadic— for these types, and constants of both types. Relational operators on integers are provided as well.

2.2 DISCIPLEAN semantics

The semantics of the resulting new language will be twofold. First of all, a CIP-like semantics will be ascribed to the constructs, which allows us to keep the same view of the transformation rules. The shift from strict to lazy evaluation is not a problem and can be ignored safely, because lazy evaluation is less restrictive than eager evaluation in many ways. In cases where this leads to real problems, Clean already provides a way out by supplying strict annotations in its type system. If necessary all we have to do is assume implicit strictness annotations in all definitions and perhaps provide a way to make those annotations explicit to be able to obtain a working program. Secondly, a program can still be viewed as a Clean program i.e., graph rewrite semantics can be ascribed to it as the operational meaning of the program. This is only true, if a program does not contain descriptive constructs.

When transforming a program one does not only need the syntactical representation of that program. Like in most languages the Clean grammar accepts programs that are semantically incorrect. Although we could have waited with checking the semantics until the system was past the experimental stage and until then assume that users would not write erroneous programs, we decided to implement a type assignment system at an early stage. The advantage of already having a type system is two-fold: the first is the obvious advantage of having less concern about the correctness of test programs, the second advantage is more important: a number of transformation rules use type information in their applicability conditions. With a type system it is possible to check such conditions, instead of leaving that to the user.

The type assignment system for DISCIPLEAN is derived from the Curry type assignment system for lambda-expressions. The system employs both type inference and type checking, dependent upon whether user supplied type information is available or not. This results in an approach that is a mix of Milner and Mycroft type assignment algorithms [Mil78, Myc84]. Our implementation essentially follows that of [PvE93].

The biggest problem for type assignment systems for a functional language are mutually recursive functions. To infer the correct type of functions that are mutually recursive, a fix-point equation for all functions that depend on each other at once has to be solved. In other words, a system of equations has to be solved. If, instead of the fix-point calculation, Milner type inference is used, the functions have to be treated together too. This greatly complicates the type assignment and so we assume that mutually recursive function definitions have a previously supplied type. This allows us to limit type assignment to type checking, which does not have these problems. In practice, most programmers already supply types with their function definitions, so this is not a severe restriction. The ability for inferring types of mutually recursive functions can be added later if still needed.

3 The transformation system

The transformation rules used in the prototype of the program transformation system are based on the transformation rules found in Chapter 4 of [Par90b]. The transformation rules can be divided into two groups

- "simple" transformation rules; and
- more complex rules such as "fold" and "unfold".

3.1 Describing transformation rules

Besides the DISCIPLEAN syntax it is also necessary to design the syntax for defining transformation rules, applicability conditions, and syntactic constraints.

Transformation rules are specified in what could be called a scheme language. This language allows Clean syntax to be mixed with variables for complete expressions. These variables can be considered as a kind of meta-variables. The type of the expected expression is coded in the variable name, e.g. `BL[0-9]* -> BOOL`. Transformation rules have the following structure

```
<Input-template>
(⟶|⟷)
<Output-template>
⟶
<Conditions>
⟶
<Syntactic-constraints>
```

The arrows, $\longrightarrow$ and $\longleftrightarrow$, between the input template and output template indicate whether the transformation rule can be applied in only one direction, from left to right, or in both directions, from left to right and from right to left. The input template and output template are DISCIPLEAN language constructs.

Both templates may contain variables which are either instantiated with constructs from the specification to be transformed or filled out by the user of the transformation system.

3.1.1 Applicability conditions

A transformation rule can only be correct when the applicability conditions are valid.

The applicability conditions are Horn clauses over program schemes that may contain additional syntactic and semantic predicates. The system uses 7 applicability conditions which have the following structure

```
<Expression> ≡ <Boolean-Constant> ⊢ <Expression> ⊆ <Expression>
<Expression> ≡ <Boolean-Constant> ⊢ <Expression> ≡ <Boolean-Constant>
DEF [ <Expression> ]
DET [ <Expression> ]
<Expression> ≡ <Boolean-Constant>
DEF [ <Expression> ] ⊢ DEF [ <Expression> ]
EQUIV <Expression> <Expression>
```

Expression and Boolean-Constant are DISCIPLEAN constructs, which may contain variables. $\equiv$, $\vdash$, $\subseteq$, DEF, DET and EQUIV belong to the additional syntax, where $\equiv$, $\vdash$, $\subseteq$ have the usual meaning. DEF [<Expression>] and DET [<Expression>] are used for determining whether the Expression is **defined** and/or **determinate**. An expression is said to be defined if the task it describes is solvable, i.e. if it can be fulfilled. An expression is said to be determinate if the task it describes is either unsolvable, or solvable and the effect of fulfilling the task, i.e. the result, is unique. EQUIV <Expression> <Expression> checks if two expressions are equivalent under the following conditions:

- eq(x,x) $\equiv$ true
- eq(x,y) $\equiv$ eq(y,x)
- eq(x,y) $\equiv$ true & eq(y,z) $\equiv$ true $\Rightarrow$ eq(x,z) $\equiv$ true

The system only fills out the variables in the applicability conditions, writes them in the administration and then leaves the proof to the user.

3.1.2 Syntactic constraints

Syntactic constraints are syntactic restrictions on the entities used in the transformation rules. Chapter four of the book of Partsch [Par90b] mentions three syntactic constraints, namely:

KIND [x **is m**]
NOTOCCURS[x **in** *Expression*]
DECL [f **is** *Expression*]

Where KIND yields the kind of some syntactic entity. NOTOCCURS tests the absence of some identifier in a given syntactic entity and DECL yields the declaration of some entity.

The system only uses one of these constraints, namely

```
NOTOCCURS [<Identifier> in <Expression>]
```

where <Identifier> and <Expression> are DISCIPLEAN constructs. This syntactic constraint is handled as follows. Firstly, the variables are filled out by the system. Secondly, it is checked whether the Identifier actually occurs in the Expression. The outcome of this check determines whether the transformation rule is applied or not.

The other two constraints can be omitted. `KIND` can be omitted because all the used expressions are checked by a type checker. This type checker checks if every used entity is of the correct type. `DECL` is only used in the Fold-rules and Unfold-rules. These are handled as a special case. The `DECL` check is done internally.

3.2 Compact rules

Compact rules are summaries of frequently used transformation sequences. All the applicability conditions belonging to a certain transformation rule appear in the compact rule as well.

3.3 Simple transformation rules

Simple transformation rules describe transformations consisting of the replacement of one program fragment, i.e. expression, by another. In a transformation rule variables may be used whose values depend either on the program fragment to be transformed or on values introduced by the user of the system when applying the transformation rule.

For the simple transformation rules we worked out the following general mechanism.

1. The actual application of a transformation rule from left to right consists of a number of steps:

    - The underlying abstract syntax tree of the program fragment is compared with the abstract syntax tree of the input template of the transformation rule. If the structure of both underlying abstract syntax trees are identical, the values of the variables used in the input template of the transformation rule are retrieved from the underlying abstract syntax tree of the program fragment. These values are internally coupled to the variables so that they can be used later on.
    - The variables in the output template which occur in the input template of the transformation rule as well are initialized with the values obtained by matching the input template to the program fragment. The value of variables which do not occur in the input template have to be given by the user of the transformation system, see Section 4.4.
    - If all variables in the output template have a value and the syntactic constraints, if any, are satisfied, the original program fragment is replaced by the output template of the transformation rule with all variables filled out. This terminates the transformation.

    If the rule is applied from right to left the input and output template should be swapped in the steps above.

2. The administrations are updated. The undo administration is updated by storing the entire "old" DISCIPLEAN program, for more details see Section 4.5. The administration of the complete transformation is updated by storing the transformed program fragment, the applied transformation rule, and the applicability conditions which must be proved later on, see Section 4.6.

3.4 Fold/unfold transformation rules

The basic idea of fold/unfold transformation rules is the replacement of function calls by function bodies and vica versa. The fold/unfold rules are more complex because it is necessary to inspect the program to be transformed. This inspection is necessary to retrieve e.g. the function body of some function call which is to be unfolded.

3.4.1 The unfold rule

The unfold rule replaces a function call with a function body. The context-free syntax rules of the function declaration is

```
Id Pattern* Guarded-body+ -> Decl
where
Boolean-constant                          -> Pattern
Integer                                   -> Pattern
Identifier                                -> Pattern
"(" Identifier "," {Identifier ","}+ ")"  -> Pattern

Guard "=" Expression ";" -> Guarded-body

"|" Expr -> Guard
         -> Guard
```

The context-free syntax rule of the function call is

```
Expression Expression -> Expression
```

The function heading is represented by `Id Pattern*`. The underlying abstract syntax tree of the function call to be unfolded is matched against the abstract syntax tree of the function heading of the corresponding function declaration. The formal arguments are replaced by the values of the actual arguments of the function call and substituted in the function body. The modified function body replaces the function call to be unfolded.

Given the function call to be unfolded the transformation system tries to find all appropiate function declarations —there may be more than one valid declaration. All these declarations are presented to the user. The user has to select the desired declaration. The facilities offered by the user interface for performing this selection are discussed in Section 4.4.

3.4.2 The fold rule

The fold rule is the inverse of the unfold rule, this rule replaces a function body with a function call. The transformation system must find the appropiate function call to the body which is to be folded. In most cases transformations may have been applied to this function body. This problem is solved by "storing" the function which is to be used during folding, if the function body is modified during the transformation process. The effect is then that the user has to indicate explicitly that he is going to do a folding transformation in the future and which function he wants to use for folding. When the actual fold transformation is applied on a given function body, the system can take this original declaration. It then matches the abstract syntax tree of the function body in the declaration with the abstract syntax tree of the given function body, hereby retrieving the actual values of the variables. Notice that these values may be entire expressions. These values are then filled out in the function call and this function call then replaces the given function body.

3.5 An example

This section gives an example of the application of both a simple and a more complex transformation rule to a small DISCIPLEAN specification. The specification to be transformed is

```
boolexpression = True & False | False;
and x y        = x & y
```

The user of the transformation system may use the simple rule

BL1 | BL2 $\longleftrightarrow$!(!BL1 & !BL2) $\longrightarrow$ Conditions $\longrightarrow$ Constraints

and the fold transformation rule. The fold transformation rule could also be described in terms of input templates and output templates, but we have chosen for connecting the application of this rule to a button, for more details see Section 4.2.

The user focuses the part of the specification which is to be transformed. The transformation system first checks in which direction the transformation rule can be applied. This check is done by comparing the abstract syntax tree of the selected fragment with the abstract syntax tree of input template of the transformation rule. The tree structures are identical, see Figure 1, so the transformation rule can be applied from left to right.

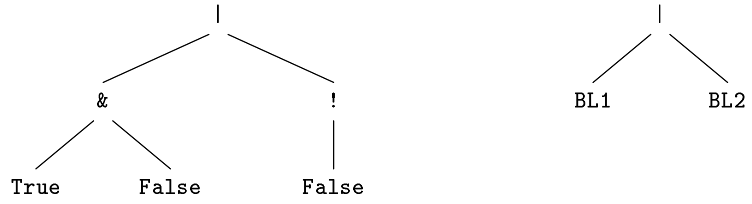

Figure 1: Specification fragment versus input template.

The abstract syntax tree of the output template is also compared with the abstract syntax tree of the selected fragment. These tree structures are not identical, see Figure 2, so the transformation rule can not be applied from right to left.

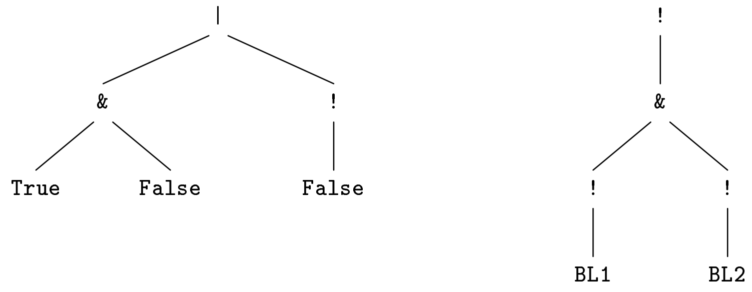

Figure 2: Specification fragment versus output template.

The values of the variables in the input template are obtained from the program fragment, `BL1 = True & False` and `BL2 = False`. The variables in the output template are replaced by the values obtained for the input template. This yields the instantiated output template `!(!(True & False) & !False)`. The original fragment is now replaced with the instantiated output template

```
boolexpression = !(!(True & False) & !False)
and x y        = x & y
```

This concludes the application of the simple transformation rule.

The user of the transformation system must indicate in advance which function is going to be used for folding, in this example the function `and` will be used of course. The expression `!(True & False) & !False` is the fragment which is to be folded. The abstract syntax tree of this fragment is compared with the abstract syntax tree of the body of the `and` function. The tree structures are identical, see Figure 3, so the fold transformation rule can be applied.

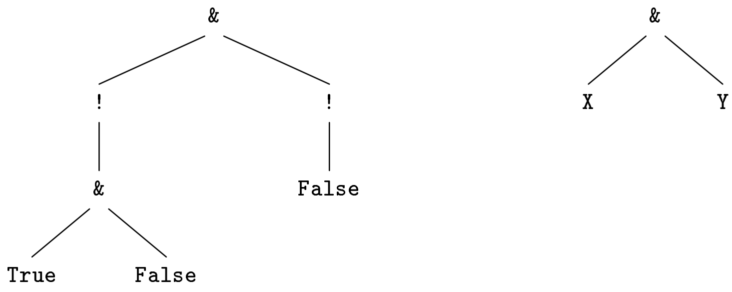

Figure 3: Specification fragment versus function body.

The values of the variables used in the function body are obtained from the selected program fragment, `x = !(True & False)` and `y = !False`. These values are used to initialize the variables in the corresponding function heading, yielding `and !(True & False) !False`. The original fragment is now replaced by this function heading

```
boolexpression = !and !(True & False) !False
and x y        = x & y
```

This concludes the application of the fold transformation.

3.6 Compact rules

Compact rules are summaries of frequently used transformation sequences. All the applicability conditions belonging to a certain transformation rule appear in the compact rule as well.

4 User interface

The user interface is one of the most important elements of a program transformation system. A transformation system is in essence a very sophisticated syntax directed editor. A traditional syntax directed editor allows the replacements of program fragments by placeholders (or meta-variables), or placeholders by templates. These substitutions are done in a context-free manner. The application of transformation rules is comparable to these actions but they need not to be context-free. Several conditions must be satisfied before a transformation rule may be applied. Some of these conditions are of a syntactic nature, the so-called syntactic constraints, others have a more semantic nature. The syntactic constraints must be checked by the transformation system before the transformation rule is applied. The semantic conditions should be checked by the user, the transformation system may offer some support in proving these conditions.

Various attempts have been made to use programming environment generators to generate program transformation systems [Vog93], transformation system constructed in this way are inflexible with respect to new transformation rules added by the user of the system. The transformation rules are mostly "hard-wired" in the specification of the transformation system, adding new transformation rules can only be done by modifying the specification and regenerating the transformation system. The transformation rules in the DISCIPLEAN transformation system are not "hard-wired" in the specification but are "stored" in a kind of data base. This data base is accessible via an editor. The user can add new transformation rules without recompiling the system. A number of frequently used transformation rules, e.g. fold and unfold, are made available via buttons and pull-down menus.

4.1 General user interface

Our initial idea of the DISCIPLEAN transformation system was that it should consist of a number of syntax directed editors over the DISCIPLEAN language extended with buttons and pull-down menus, and a number of editors with transformation rules from the "user-extensible" data bases.

Selecting a program fragment in one of the DISCIPLEAN editors results in the activation of buttons and menu entries of valid transformation rules as well as the creation of an editor with all valid transformation rules from the "user-extensible" data bases. Given the fact that there will be over one hundred transformation rules we decided that this will not lead to a workable system with respect to performance.

We have opted for a system with the same structure as above however the selection of a program fragment does not lead to the activation of any button, menu entry, or editor. Transformation rules should explicitly be selected by the user of the system, application conditions are checked on demand. The prototype of the DISCIPLEAN transformation system does not offer more than one editor for doing the transformations.

The user interface can by divided in to the following parts:

- The DISCIPLEAN window, in this window the user can do the actual transformation on a DISCIPLEAN program. This editor offers the possibility to transform more than one DISCIPLEAN program.
- Transformation windows, in these windows the transformation rules are presented.
- Help window, for the auxilary communication with the user.
- Undo window, in this window all intermediate results are stored, in order to backtrack over the transformations.
- Administration windows for the administration of the complete transformation, for authenticity checking, and the administration of all conditions that have to be proved by the user of the system.

There are buttons on the DISCIPLEAN and transformation windows for applying a selected transformation rule to a selected DISCIPLEAN program fragment.

4.2 DISCIPLEAN editor

The DISCIPLEAN window contains zero or more DISCIPLEAN programs on which the user can do transformations. The DISCIPLEAN window is fitted with six buttons, see Figure 4

These buttons do the following :

- **Clear**, to clear the DISCIPLEAN-, Undo- and Administration windows.
- **Load**, to load a DISCIPLEAN file.

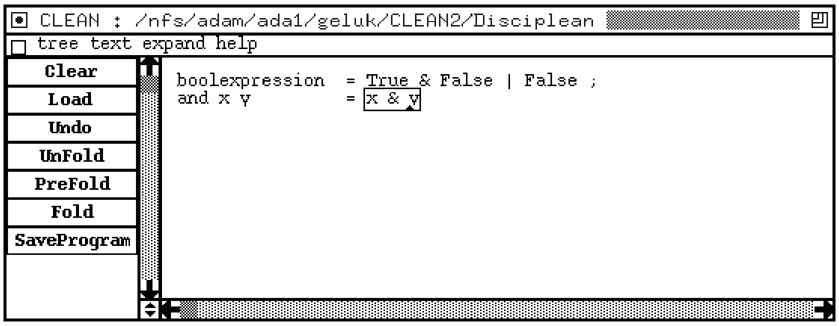

Figure 4: The DISCIPLEAN window with the example from Section.

- **Undo**, to undo the last transformation step by replacing the entire program by the last saved program in the Undo window.
- **Unfold**, replacing the focus in the DISCIPLEAN window with the matching function body, if this body can be found. If there is more than one matching body, the user is asked in the Help window to select one. If there is no match or the focus is not on a function call the system generates an error message in the Help window.
- **PreFold**, saving the focus of the DISCIPLEAN window in a separate "fold" section of the DISCIPLEAN window. The contents of this "fold" section will be used during a fold action.
- **Fold**, replacing the focus in the DISCIPLEAN window with the matching function call, if this function call can be found in the "fold" section. If there is more than one matching function call, the user is asked in the Help window to select one. If there is no match or the focus is not on a function body the system generates an error message in the Help window.
- **SaveProgram**, saving the entire program in the administration.

4.3 Transformation rule editors

The transformation rules are divided over several independent transformation rule editors, according to the basic and predefined data types in DISCIPLEAN. Each of these editors contains a collection of data type specific transformation rules, so there are editors for the transformation rules over the Booleans, Integers, Maps, Sequences, Tuples, etc. Furthermore, there is an editor with a collection of more general transformation rules. The transformation rule editors have three buttons, see Figure 5.

These buttons do the following :

- **Check**, the system checks if the selected transformation can be applied on the focus in the DISCIPLEAN window. If the transformation can be applied, in any direction, the system will return the transformation rule with all known variables filled out in the Help window. If the transformation rule can not be applied the system will return a message that the selected rule can not be applied on the actual focus.
- **ApplyLR**, if the selected transformation can be applied from left to right on the focus in the DISCIPLEAN window, the focus in the DISCIPLEAN window will be replaced by the result of the transformation. If the selected transformation

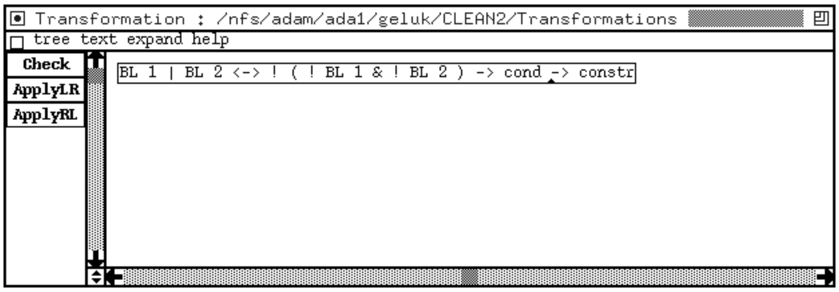

Figure 5: A transformation rule editor with the transformation rule used in Section.

can not be applied the system will return a message in the Help window that the selected rule could not be applied.

- **ApplyRL**, if the selected transformation can be applied from right to left on the focus in the DISCIPLEAN window, the focus in the DISCIPLEAN window will be replaced by the result of the transformation. If the selected transformation can not be applied the system will return a message in the Help window that the selected rule could not be applied.

It is not allowed for the user of the transformation system to edit in the transformation rule editors. The extension of the set of transformation rules with a new transformation rule should only be done via an "add" button. SEAL does however not offer any facilities to disable the editing in a window.

4.4 Help window

The Help window does not contain any buttons but is the most important window for communicating with the user. There are three different types of communications between system and user:

1. Messages from the system to the user.

   For example, the message

   ```
   Applying from Left to Right will lead to :
       True & False | False  ⟶  !(!(True & False) & !True)
   ```

   will appear if the user pushes the Check button on the Transformation window with one focus on the DISCIPLEAN fragment:

   ```
   True & False | False
   ```

   and the other on the transformation rule

   ```
   BL1 | BL2  ⟷  !(!BL1 & !BL2)
   ```

2. Variable input from the user. The user is asked to fill out some unknown variable in an output template of a transformation rule, see Figure 6.

3. Selection of various declarations for fold/unfold transformation. The user is asked to select the required declaration by mouse clicking on it, see Figure 7

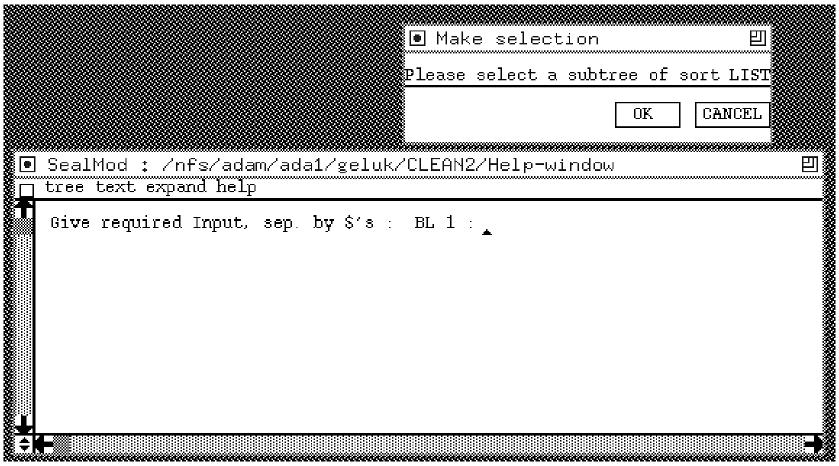

Figure 6: The Help window with a request to fill in the missing variable `BL1`.

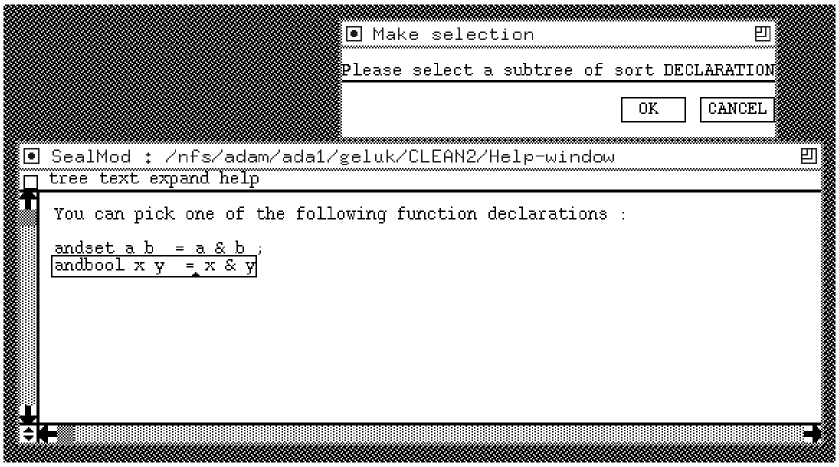

Figure 7: The Help window with a request to select the required declaration.

4.5 Undo facility

The user of the transformation system can backtrack over the performed transformations. The system supports an unlimited undo facility. If it is necessary the original DISCIPLEAN specification can even be retrieved.

The undo facility should have been only visible by means of an `undo` button which is activated whenever undo actions are possible. SEAL does not offer sophisticated file manipulation facilities however. The only way to implement the undo facility is by means of an editor, which is always visible for the user. The disabling of editing in an editor, see Section 4.3, is a feature which is also desirable for the undo editor.

Another feature of the undo facility should be to store only the modified part of the DISCIPLEAN specification. The prototype saves the entire program after each transformation. An undo operation does not result in restoring the focus in a correct way. This solution works not very satisfactory, because the focus is lost.

In order to implement an undo facility which does not lose the focus it is necessary to obtain the path from the focus to the root of the abstract syntax tree of the DISCIPLEAN specification. This can be done via small SEAL specification by zooming out until the root, and by recording the SEAL actions (`up` and `left`) necessary to reach the root. By reversing this list of actions and by translating each `up` action into a `down` action and each `left` action into a `right` action, the old focus can be reset. However this has an annoying visible effect in the window.

4.6 Administration window

The developed program transformation system will be used for the educational purposes. Therefore it is necessary to have an accurate administration of all transformation steps. The administration of transformation steps consists of recording the applied transformation rule, the resulting DISCIPLEAN specification, and the conditions which must be proved by the user of the system afterwards. This administration gives a complete history from the initial specification to the resulting transformed program.

The administration is done in the same way as the undo bookkeeping, the parts that we want to place in the administration are placed in a window which is always visible. The information of the administration window is only written to a file if the user closes the administration window and answers Yes to the question if he wants to save the file. Due to the fact that SEAL does not have a kill-window option **every** window the system uses has to be closed and saved by the user if he wants to preserve the information that is in it.

This editor must have some sort of protection, if the user clicks in it, the system will erase the contents and generate a message that the Administration is erased. This is a very crude way, but there is no other way to make sure the user has not modified the transformation history.

5 Conclusions and future work

The overall conclusion is that ASF+SDF is a suited formalism for developing a program transformation system. The ASF+SDF Meta-environment proved to be very helpful in the process of developing and designing the DISCIPLEAN language and the scheme language.

There are various shortcomings in the current prototype, some of which are the result of restrictions imposed by SDF, others are caused by problems with SEAL. Some problems are also caused by having not yet a clear understanding of how to express a desired functionality. For example, the problem with the scheme language is that it should allow a way of dealing with an unknown (but finite, of course) amount of expressions. For instance, it must be possible to apply a transformation rule about tuples to tuples of arbitrary size without having to specify the rule for all possible sizes.

5.1 Influences of ASF+SDF on the syntax definition

The current definition of the syntax has been fixed on a suitable subset of what DISCIPLEAN should become in the end. We would have liked to include a lot more of Cleans type system and features for function definitions in DISCIPLEAN. Algebraic type definitions and operator definitions were left out to limit the size of the project.

While there is currently no reason to assume that there will be problems because of limitations of ASF+SDF when adding algebraic type definitions to DISCIPLEAN, this is not true for user defined operators. Using the features of SDF for specifying the syntax, we were forced to include every predefined operator explicitly in the syntax and to fix all priorities and associativities in the syntax. As a result of this, allowing user defined operators or even changing or adding predefined operators requires changing the syntax. Again, it was not our aim to include user defined operators from the beginning. We did want however, to set up the language definition in a way that allowed easy extension of the language. In other words what we missed in the formalism was a way to decide between different parse trees based on ASF equations.

In the summary of the syntax of the Section 2.1 one might notice the dots that conclude quantified expressions and let-expressions. These dots were introduced due to a problem with priorities between expressions. We decided that a quantified expression should have a low priority, because this way a quantifier will take all expressions behind it as its predicate and we assumed that this is the usual way that quantified expressions are regarded. For instance, in an expression with the general form '**forall** x : P && R' normally one would regard 'P && R' to be the predicate over x and one would place brackets if only P was to be the predicate. It was quite well possible to give quantified expressions a lower priority and in cases where they were the left-most expressions this caused no problems. If however, the quantifier had an expression to its left with an operator with higher priority a problem did occur. The system was not able to create an unambiguous tree for an expression like '`1 +` **that** `x :  x ==` `1`'. The obvious correct parse tree for this expression is Figure 8.

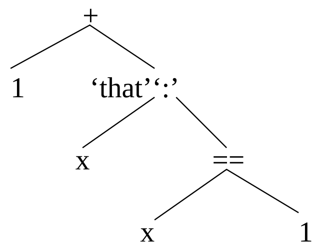

Figure 8: Operator tree of expression '`1 +` **that** `x :  x ==` `1`'

The parser can not construct this tree however, because '`+`' is supposed to have a higher priority and thus occur as a sub tree of the quantified expression. This problem is avoided by introducing the dot as a delimiter of quantified and let-expressions.

The conclusion to be drawn out of all this seems to be that the current scheme of declaring priorities and associativity in SDF, while easy to use and flexible, is still too limited for some applications. These limitations are obviously only problematic when specifying languages where the user can define the priority of operators[1] or where the number of predefined operators is very likely to be often extended[2].

5.2 Restrictions of SEAL

The entire user interface is built with SEAL [Koo94]. The use of SEAL has led to a number of unexpected problems and limitations which have influenced the resulting prototype in a considerable way.

As described in Section 4.1 the initial idea of the DISCIPLEAN transformation system was a collection of syntax directed editors over the DISCIPLEAN language extended with buttons and pull-down menus, and a number of editors with transformation rules from the "user-extensible" data bases. The selection of a program

[1]This does not mean that user defined operators are impossible in ASF+SDF; in C++ for instance, the user can overload operators thus creating a new operator, but this new operator inherits its priority and associativity from the existing operator

[2]This is of course only a problem if the application has a fundamental need for flexibility in the available operators. If the operators only change in the development phase it is just a result from rapid prototyping techniques and has to be lived with.

fragment in one of the DISCIPLEAN editors should have resulted in a collection of valid transformation rules which could be applied to the selected fragment. More than hunderd transformation rules should be inspected in order to decide whether they are valid. Due to expected performance problems of SEAL this elegant and simple idea is not realised.

The next "version" of the transformation system consists of a collection of syntax directed editors over the DISCIPLEAN language and a collection of editors containing transformation rules. The user selects not only a program fragment but also the transformation rule that should be applied. However, due to SEAL it was not possible to work with a collection of syntax directed editors. Although suggested otherwise in [Koo94], focus assignments in editors without fixed names are not allowed. So, we were forced to work with only one editor over the DISCIPLEAN language. The collection of DISCIPLEAN editors is simulated by this single editor by a list of DISCIPLEAN specifications. Each of the specifications in this list can be transformed.

One of the most annoying consequences of the use of SEAL is the enormous amount of editors active in the transformation system. Because of restricted file manipulation facilities, editors had to be created for the administration of undo information and the bookkeeping administration. In fact all information for administration should not be visible to the user of the transformation system.

Another quite big problem is the absence of facilities to manipulate the permission for editing in an editor. The transformation rules should not be editable by the user. SEAL does not offer an elegant way to obtain this behaviour.

5.3 Future work

The current version of the prototype is completely integrated in the ASF+SDF Meta-environment. The transformation system should become a stand-alone tool, independent of the ASF+SDF Meta-environment. The basis for a stand-alone transformation system will be the ToolBus [BK95]. A stand-alone system consists of a compiled version of the current specification in ASF+SDF, a user interface based on Tcl and Tk [Ous94], and a number of parsers to process the DISCIPLEAN specification and the transformation rules. A lot of the problems caused by SEAL will be solved when using Tcl and Tk for the design of the user interface. Such a stand-alone transformation system can be used to do experiments with connecting (existing) proof checkers and other relevant tools.

References

[BBB+85] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[BEH+87] F.L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP. Volume II: The Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[BK95] J.A. Bergstra and P. Klint. The discrete time toolbus. Technical Report P9502, University of Amsterdam, Programming Research Group, 1995.

[Bou] R.T. Boute. Funmath: a unifying formalism for applied mathematics and computing science. submitted for publication.

[BvdBvD+90] E.A. Boiten, M.G.J. van den Brand, N.W.P. van Diepen, C.H.A. Koster, H.A. Partsch, and N. Völker. USTOPIA requirements – Thoughts on a User-friendly System for Transformation Of Programs In Abstracto. Technical Report 90-12, University of Nijmegen, Department of Informatics, 1990.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, Berlin, 1981.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Koo94] J. W. C. Koorn. *Generating uniform user-interfaces for interactive programming environments*. PhD thesis, University of Amsterdam, 1994. ILLC Dissertation series 1994-2.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In Paul M. and Robinet B., editors, *Proc. International Conference on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–239. Springer-Verlag, 1984.

[OL] Hugh Osborne and Wil Lamain. Afleidingsmanoeuvres. In Dutch.

[Ous94] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Massachusetts, 1994.

[Par90a] H.A. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development*. Springer-Verlag, Berlin, 1990.

[Par90b] H.A. Partsch. *Specification and transformation of programs : a formal approach to software development*. Springer-Verlag, 1990.

[Par91] H. Partsch. *Requirements Engineering*, volume Band 5:5 of *Handbuch der Informatik*. Oldenbourg, München, 1991.

[PvE93] M.J. Plasmeijer and M.C.J.D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[Smi90] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[STO92] STOP. *Lecture Notes of the STOP 1992 Summer School on Constructive Algorithmics, Ameland*, 1992.

[Vog93] H.H. Vogt. *Higher Order Attribute Grammars*. PhD thesis, Rijksuniversiteit Utrecht, 1993.

A The Administration of a (very simple) example derivation

Let s and t be finite sequences and x an element.

The initial DISCIPLEAN **specification is:**

```
d s x t = t ++ delete s ;
delete a = if (isempty a)
              a
              if (first a == x)
                 (rest a)
                 (first a ++ delete (rest a));
```

- Step -

A prefold has been done on the focus:

```
d s x t = t ++ delete s ;
```

The function-call:

```
delete s
```

was unfolded to the function-body:

```
if (isempty s)
   s
   if (first s == x)
      (rest s)
      (first s ++ delete (rest s));
```

- Step -

The focus:

```
t ++ if (isempty s)
        s
        if (first s == x)
           (rest s)
           (first s ++ delete (rest s))
```

was replaced by:

```
if (isempty s)
   (t ++ s)
   (t ++ if   (first s == x)
              (rest s)
              (first s ++ delete (rest s)))
```

by applying the transformation:

```
EXPR1 ++ if  BL1
             EXPR2
             EXPR3
⟶
if BL1
   (EXPR1 ++ EXPR2)
   (EXPR1 ++ EXPR3)
⟶
⟶
```

- Step -

The focus:

```
t ++ if (first s == x)
        (rest s)
        (first s ++ delete (rest s))
```

was replaced by:

```
if (first s == x)
   (t ++ rest s)
   (t ++ (first s ++ delete (rest s)))
```

by applying the transformation:

$$\begin{array}{l} \mathit{EXPR1} \mathbin{+\!\!+} \text{if } \ \mathit{BL1} \\ \qquad\qquad\quad \mathit{EXPR2} \\ \qquad\qquad\quad \mathit{EXPR3} \\ \longrightarrow \\ \text{if } \mathit{BL1} \\ \quad (\mathit{EXPR1} \mathbin{+\!\!+} \mathit{EXPR2}) \\ \quad (\mathit{EXPR1} \mathbin{+\!\!+} \mathit{EXPR3}) \\ \longrightarrow \\ \longrightarrow \end{array}$$

- Step -

The focus:

```
t ++ s
```

was replaced by:

```
t ++ [ ]
```

by applying the transformation:

$$\begin{array}{l} \mathit{EXPR1} \\ \longrightarrow \\ \mathit{EXPR2} \\ \longrightarrow \\ \longrightarrow \end{array}$$

- Step -

The focus:

```
t ++ [ ]
```

was replaced by:

```
t
```

by applying the transformation:

$$\begin{array}{l} \mathit{SEQ1} \mathbin{+\!\!+} [\ ] \\ \longleftrightarrow \\ \mathit{SEQ1} \\ \longrightarrow \\ \longrightarrow \end{array}$$

- Step -

The focus:

```
t ++ (first s ++ delete (rest s))
```

was replaced by:

```
t ++ first s ++ delete (rest s)
```

by applying the transformation:

$$\begin{array}{l} \mathit{SEQ1} \mathbin{+\!\!+} (\mathit{SEQ2} \mathbin{+\!\!+} \mathit{SEQ3}) \\ \longleftrightarrow \end{array}$$

```
SEQ1 ++ SEQ2 ++ SEQ3
⟶
⟶
```

- Step -

The function-body:

```
t ++ first s ++ delete (rest s)
```

was folded to the function-call:

```
d (rest s) x (t ++ first s)
```

- Step -

The final Disciplean **program is:**

```
d s x t = if (isempty s)
               t
               if (first s == x)
                  (t ++ rest s)
                  (d (rest s) x (t ++ first s)) ;
delete a =  if (isempty a)
               a
               if (first a == x)
                  (rest a)
                  (first a ++ delete (rest a)) ;
_fold
d s x t = t ++ delete s ;
```

-End-

B Disciplean

imports Expressions^{C}
exports
sorts DECLARATION PATTERN GUARDED-BODY GUARD SCRIPT DECL-LIST OPT-LOCALS
exports
context-free syntax

DECLARATION	→ DECL-LIST
DECLARATION DECL-LIST	→ DECL-LIST
DECL-LIST	→ SCRIPT
	→ SCRIPT
IDENTIFIER PATTERN∗ GUARDED-BODY+	→ DECLARATION
CONSTANT	→ PATTERN
IDENTIFIER	→ PATTERN
"(" IDENTIFIER "," {IDENTIFIER ","}+ ")"	→ PATTERN
GUARD "=" EXPRESSION OPT-LOCALS ";"	→ GUARDED-BODY
	→ OPT-LOCALS
"*where*" DECL-LIST	→ OPT-LOCALS
"|" EXPRESSION	→ GUARD
	→ GUARD

C Expressions

DISCIPLEAN is an untyped language, everything is an expression.
imports Nat Bool Identifiers
exports
Formal patterns do not appear in the definitions of functions, they are introduced here, because they appear in quantified expressions. A formal pattern can not contain constants, only identifiers and tuple are allowed.

```
sorts FORMAL-PATTERN
context-free syntax
  IDENTIFIER                                 → FORMAL-PATTERN
  "(" IDENTIFIER "," {IDENTIFIER ","}+ ")"   → FORMAL-PATTERN
sorts UNDEFINED CONSTANT TUPLE SET SEQUENCE MAP
      CONDITIONAL QUANT-EXPRESSION LET-EXPRESSION
      EXPRESSION
lexical syntax
  "UNDEFINED" → UNDEFINED
context-free syntax
  "(" EXPRESSION "," {EXPRESSION ","}+ ")" → TUPLE
context-free syntax
  "{|" FORMAL-PATTERN ":" EXPRESSION "|}" → SET
  "{|" "|}"                               → SET

  "[" {EXPRESSION ","}* "]"               → SEQUENCE

  "<" {EXPRESSION ","}* ">"               → MAP

  "if" EXPRESSION EXPRESSION EXPRESSION   → CONDITIONAL

  UNDEFINED                               → EXPRESSION
  CONSTANT                                → EXPRESSION
  FORMAL-PATTERN                          → EXPRESSION
  TUPLE                                   → EXPRESSION
  SET                                     → EXPRESSION
  SEQUENCE                                → EXPRESSION
  MAP                                     → EXPRESSION
  CONDITIONAL                             → EXPRESSION
context-free syntax
  EXPRESSION EXPRESSION → EXPRESSION {left}
context-free syntax
  EXPRESSION "[|]" EXPRESSION → EXPRESSION {left}
context-free syntax
  "exists" FORMAL-PATTERN ":"
  EXPRESSION "."              → QUANT-EXPRESSION
  "forall" FORMAL-PATTERN ":"
  EXPRESSION "."              → QUANT-EXPRESSION
  "some" FORMAL-PATTERN ":"
  EXPRESSION "."              → QUANT-EXPRESSION
  "that" FORMAL-PATTERN ":"
  EXPRESSION "."              → QUANT-EXPRESSION

  "let" FORMAL-PATTERN "="
  EXPRESSION "in"
  EXPRESSION "."              → LET-EXPRESSION

  QUANT-EXPRESSION            → EXPRESSION
```

LET-EXPRESSION → EXPRESSION
“(” EXPRESSION “)” → EXPRESSION {**bracket**}
context-free syntax
BOOLEAN-CONSTANT → CONSTANT

EXPRESSION “&” EXPRESSION → EXPRESSION {**left**}
EXPRESSION “|” EXPRESSION → EXPRESSION {**left**}
“!” EXPRESSION → EXPRESSION

EXPRESSION “!&!” EXPRESSION → EXPRESSION {**left**}
EXPRESSION “!|!” EXPRESSION → EXPRESSION {**left**}
context-free syntax
INTEGER → CONSTANT

“+” EXPRESSION → EXPRESSION
“−” EXPRESSION → EXPRESSION

EXPRESSION “+” EXPRESSION → EXPRESSION {**left**}
EXPRESSION “−” EXPRESSION → EXPRESSION {**left**}
EXPRESSION “∗” EXPRESSION → EXPRESSION {**left**}
EXPRESSION “/” EXPRESSION → EXPRESSION {**left**}
context-free syntax
EXPRESSION “++” EXPRESSION → EXPRESSION {**left**}
EXPRESSION “*sub*” EXPRESSION → EXPRESSION {**left**}
context-free syntax
EXPRESSION “*union*” EXPRESSION → EXPRESSION {**left**}
EXPRESSION “*intersection*” EXPRESSION → EXPRESSION {**left**}
EXPRESSION “*minus*” EXPRESSION → EXPRESSION {**left**}

EXPRESSION “*in*” EXPRESSION → EXPRESSION
EXPRESSION “*notin*” EXPRESSION → EXPRESSION
context-free syntax
EXPRESSION “>” EXPRESSION → EXPRESSION {**left**}
EXPRESSION “≥” EXPRESSION → EXPRESSION {**left**}
EXPRESSION “≡” EXPRESSION → EXPRESSION {**left**}
EXPRESSION “≠” EXPRESSION → EXPRESSION {**left**}
EXPRESSION “≤” EXPRESSION → EXPRESSION {**left**}
EXPRESSION “<” EXPRESSION → EXPRESSION {**left**}
priorities
“*if*”EXPRESSION EXPRESSION EXPRESSION → CONDITIONAL > EXPRESSION EXPRESSION → EXPRESSION > {**non-assoc**: “*sub*”, “*in*”, “*notin*”} > “++” > {“+”EXPRESSION → EXPRESSION, “−”EXPRESSION → EXPRESSION} > {**left**: “∗”, “/”} > {**left**: EXPRESSION “+”EXPRESSION → EXPRESSION, EXPRESSION “−”EXPRESSION → EXPRESSION} > {**non-assoc**: “>”, “≥”, “≡”, “≠”, “≤”, “<”} > “!”EXPRESSION → EXPRESSION > {**left**: “&”, “!&!”} > {**left**: “|”, “!|!”} > “[|]” > “*intersection*” > {**left**: “*union*”, “*minus*”}