URL: http://www.elsevier.nl/locate/entcs/volume44.html 25 pages

Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment

Ralf Lämmel[1]

CWI & Vrije Universiteit
Amsterdam, The Netherlands

Guido Wachsmuth[2]

Fachbereich Informatik
Universität Rostock
Rostock, Germany

Abstract

We describe FST—a *F*ramework for *S*DF *T*ransformation. FST supports the adaptation (in a broad sense) of grammars based on the syntax definition formalism SDF. We further describe the prototype implementation of FST in the ASF+SDF Meta-Environment. Grammar transformations form an important concept of grammar reengineering, implementation, recovery and others. Tool support for grammar transformations is essential to automate the corresponding processes.

1 Introduction

Adaptation of grammars

Grammars are software artifacts. They are of prime importance in many application domains, especially in the domain of generic language technology. By grammars, we mean concrete syntax definitions, abstract syntax definitions, intermediate and exchange formats. Like any software, grammars need to be developed and maintained. Since grammars serve usually as important contracts for other software components, grammars should be adapted with care. This paper reports on FST—a *F*ramework for *S*DF *T*ransformation. As the name points out, the syntax definition formalism SDF [9,22] is the grammar notation covered by FST. The framework essentially provides a suite of

[1] Email: Ralf.Laemmel@cwi.nl
[2] Email: guwac@informatik.uni-rostock.de

©2001 Published by Elsevier Science B. V. Open access under CC BY-NC-ND license.

operators to describe grammar transformations. The resulting style of grammar programming can be conceived as *transformational grammar programming.* The operator suite and all other FST concepts were prototyped in ASF+SDF. The resulting algebraic specification is executable in the (new) ASF+SDF Meta-Environment [11,5].

Transformation sample

We will use the VS COBOL II language [10] as the running example of the paper. Grammar transformations have been used extensively in correcting and completing the grammar contained in the standard. The corresponding grammar recovery project is described in [14].

Let us consider one specific problem in the standard. The sort `Subscript` was, for example, used but not defined. The sort is meant to describe proper forms of subscripts for data items in the VS COBOL II language. Instead of proper subscripts, a related construct was defined via the sort `Subscripting`. The problem with the available definition of `Subscripting` is that it defines the complete form of a data name reference *with a subscript involved.* Consequently, we need to identify that part of `Subscripting` which corresponds to `Subscript`. This idea can be expressed in a precise manner by the transformation in Figure 1. The first transformation step `fold`s the relevant part to a new nonterminal. For convenience, the `focus` for the transformation is made explicit. The second step is meant to `unify` the introduced and the required sort. The remaining steps `rename` the candidate to the proper name, and `eliminate` the obsolete nonterminal `Subscripting`.

FST—*A Framework for SDF Transformations*

The concepts offered by FST are illustrated in Figure 2. The figure actually depicts the modular hierarchy of the FST prototype in the ASF+SDF Meta-Environment. The primary concepts offered by FST are operators. There are *primitive* operators like `add` for adding productions, `replace` for replacing phrases (i.e., extended BNF expressions) within productions, `substitute` for replacing sorts. Using *combinators*, derived operators can be defined which are meant to embody proper steps of transformations. There are several groups of derived operators, namely operators for grammar *refactoring*, for *construction* and *destruction* of grammars. Furthermore, FST provides several supplementary concepts to enable transformational grammar programming. We need, for example, a concept of *focus* to restrict the focus for the application of transformations. We also need a concept of *condition* to formulate pre- and post-conditions of transformations.

Structure of the paper

In Section 2, the platform for the prototype implementation of FST, especially the relevant formalisms, are explained. In Section 3 and Section 4, the various

Available definition of `Subscripting`

```
Condition-name | Data-name "("
    ( Integer
    | (Data-name  ("+" | "-" Integer)?)
    | (Index-name ("+" | "-" Integer)?))+
")" -> Subscripting
```

Transformation

```
focus on sort Subscripting do
 fold ( Integer
      | (Data-name  ("+"|"-" Integer)?)
      | (Index-name ("+"|"-" Integer)?) )+
 to Subscript-candidate;
unify Subscript to Subscript-candidate;
rename Subscript-candidate to Subscript;
eliminate Subscripting;
```

Resulting definition of `Subscript`

```
( Integer
| (Data-name  ("+"|"-" Integer)?)
| (Index-name ("+"|"-" Integer)?)
)+ -> Subscript
```

Fig. 1. A grammar transformation for VS COBOL II

concepts offered by FST are defined in some detail. For convenience, we separate basic concepts and derived operators. In Section 5, we report on applications of grammar transformations. The paper is concluded in Section 6.

Acknowledgement The work of the first author was supported, in part, by the Dutch Research Organisation *NWO*, in the project 612.014.006 "*Generation of Program Transformation Systems*". The work of the second author was supported by Universiteit van Amsterdam, Programming Research Group, through a grant for a research visit. We are grateful for Paul Klint's support of this project. We are grateful to Jan Heering, Merijn de Jonge, Jan Kort, Jurgen Vinju, Joost Visser, and Dave Wile for discussions on this project.

2 Prototyping FST in ASF+SDF

FST has been prototyped in the ASF+SDF Meta-Environment. In the present section, we explain this environment. By coincidence, SDF is also the grammar formalism of choice for FST. First, the SDF syntax definition formalism is briefly described. Then, ASF(+SDF) used for the (algebraic) specification of the grammar transformations is briefly described. Finally, the recent extension of ASF+SDF by traversal functions is motivated. Traversal functions allow us to describe grammar transformations much more concise.

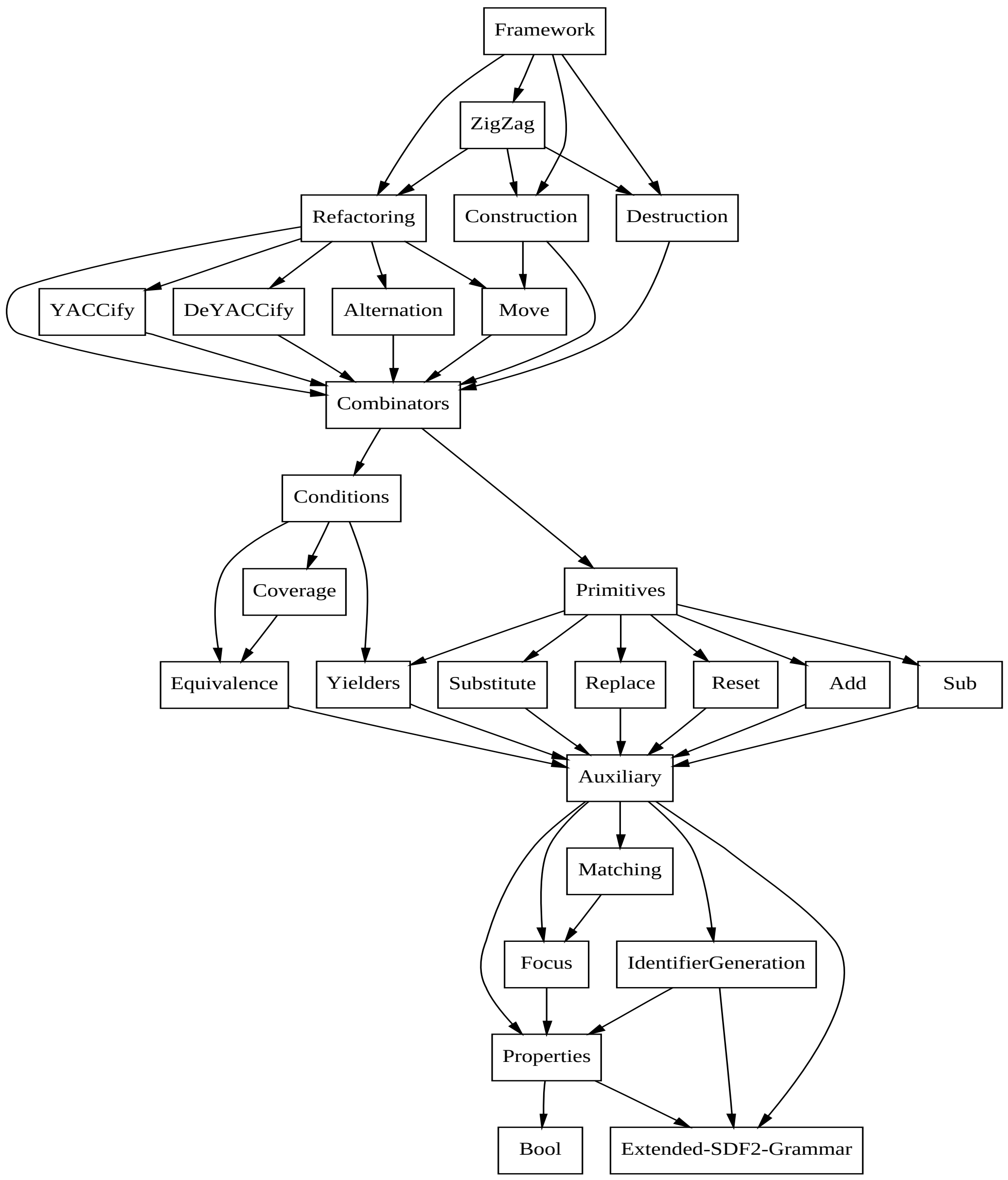

Fig. 2. Concepts of FST

2.1 SDF

SDF [9,22] is a syntax definition formalism offering not just extended BNF expressiveness but also constructs for modular syntax and disambiguation of syntax. SDF is complemented by a parser generator *pgen* and a table-driven parser *sglr* supporting scannerless generalised LR parsing [21,18]. A SDF production is of the form `Symbol* -> Symbol`. Note that left-hand side and right-hand side are flipped compared to standard BNF notation. Note also that the right-hand side symbol is usually a sort if the production describes syntax. In general, arbitrary symbols can occur on the right-hand side. This would be useful for rewriting functions. Primitive symbols are sorts (`Sort`;

another term for nonterminals) and literals (`Literal`; another term for terminals). Symbols are composed using extended BNF notation. We saw already some grammar fragments in SDF notation in the introductory example in Figure 1.

```
module Regular-Sdf-Syntax
 imports Kernel-Sdf-Syntax IntCon
 exports
  context-free syntax
   Symbol "?"                       -> Symbol
   Symbol "+"                       -> Symbol
   Symbol "*"                       -> Symbol
   Symbol "|" Symbol                -> Symbol {right}
   ...
  context-free priorities
   {Symbol "?"              -> Symbol
    Symbol "*"              -> Symbol
    Symbol "+"              -> Symbol} >
    Symbol "|" Symbol       -> Symbol
   ...
```

Fig. 3. SDF syntax of SDF

Part of the syntax of SDF is defined in SDF notation itself in Figure 3. Note that FST relies on SDF2 [22] rather than SDF1 [9] as it covers SDF2-specific constructs, e.g., optionals and alternatives. The shown module fragment lists some forms of compound symbols, namely optionals (cf. `Symbol "?"`), plus-lists (cf. `Symbol "+"`), star-lists (cf. `Symbol "*"`), and alternatives (cf. `Symbol | Symbol`). We also see an example of the declarative disambiguation constructs offered by SDF. In the `context-free priorities` section the binding of some operators is regulated. In particular, "?", "*" and "+" bind stronger than "|".

SDF is indeed an excellent basis for syntax definition because of its orthogonality and expressiveness. Also, the available implementation does not restrict grammars to LR(1) or any other subclass of context-free grammars. Meanwhile, SDF grammars have been developed for many languages. SDF is a challenging target for grammar transformations. Due to its complexity (modules, extended BNF, permutation phrases, variables, disambiguation constructs and others), many design decisions are due.

2.2 ASF+SDF

The formalism couple ASF+SDF supports algebraic specification (cf. [3] for ASF) based on concrete syntax. ASF+SDF is supported within the ASF+SDF Meta-Environment [11,5]. ASF accomplishes conditional rewrite rules. Some decent form of control can be specified, that is, one can point out default rules.

In Figure 4, we illustrate ASF+SDF with an excerpt from one module of FST. Part of the module `Combinators` is shown. The module defines the

```
module Combinators
 imports Conditions Primitives
 exports
  sorts Trafo
  context-free syntax
   Trafo ";" Trafo -> Trafo {right}
   "focus" "on" FocusYielder "do" Trafo -> Trafo
   "if" Cond "then" Trafo "else" Trafo -> Trafo
   Trafo "effectively" -> Trafo
   "guard" Cond -> Trafo
   ...
equations
[IT-1] &SDF1 = IT(&Trafo0,&SDF0,&Focus0),
       &SDF2 = IT(&Trafo1,&SDF1,&Focus0)
       ========================================
       IT(&Trafo0; &Trafo1, &SDF0, &Focus0)=&SDF2
...
[default-IT-6] IT (&Trafo0, &SDF0, &Focus0) = undefined
[guard-1] guard &Cond0 = if &Cond0 then id else fail
```

Fig. 4. ASF+SDF module for combinators for grammar transformations

interpretation of combinators for grammar transformations. The SDF part indicates that there are, for example, combinators for sequential composition (cf. `...; ...`), focused transformation (cf. `focus ...`), and conditional transformation (cf. `if ...`). The interpretation function for transformations is `IT`. We only show two ASF equations for `IT`. They are tagged `[IT-1]` and `[default-IT-6]`. The former defines the interpretation of sequential composition. The latter is a default rule to return `undefined` if none of the other rules triggers. As for the conditional rewrite rule `[IT-1]`, we see that premises and conclusion are separated by `===...===`, conditions are separated by comma. Note that we usually use the sort identifier for the stem of many-sorted variables, e.g., the variable `&Trafo0` is used as a place-holder for sort `Trafo` in the rewrite rule `[IT-1]`.

As an aside, algebraic specifications in the context of program transformation, analysis, interpretation and compilation are often conveniently written in the constructive style. That is, there is a separation of constructors and defined operations, and rewrite rules perform a kind of case discrimination on constructor patterns. In fact, FST is written in the constructive style (+ disciplined default equations and use of implicit traversals) so that we can easily claim confluence for our specifications. Since ASF+SDF favours concrete syntax, constructors correspond to SDF productions, and SDF is also used to declare profiles of ASF operations.

2.3 Traversal functions

The ASF+SDF formalism is currently being extended with support for traversal functions facilitating concise definitions of traversals [4]. We use this extension for FST in order to specify several of the program transformations and program analyses underlying FST. They key idea is that distinguished function symbols perform top-down traversals. The programmer provides rewrite rules with such function symbols as outermost symbols to refine in a sense the default traversal. This mechanism pays off if the traversal is specific for few patterns. This is indeed very often the case in the context of program transformation and analysis. Without explicit support, traversals are usually encoded as explicit case discriminations.

In Figure 5, one primitive transformation operator offered by FST is defined. The operator performs sort substitution all over an SDF definition. We will explain details of this specification in a second. Just note that the transformation, which is applicable to the full SDF syntax, can be specified with just *three* rewrite rules (refining an implicit traversal). The prototype implementation of FST comprises many such traversals which benefit from the extension for traversal functions. **Disclaimer**: The traversal extension for ASF+SDF including the underlying notation is still subject to change. In the current paper, we take a snapshot. The ultimate reference is [4] to be published soon.

```
module Substitute
 imports Focus
 exports
  context-free syntax
   "substitute" "(" SDF "," Focus "," Sort "," Sort ")" -> SDF {traverse}
equations
[subst-1] &Module0= module &ModuleName0 &ImpSection*0 &Sections0,
          focused(&ModuleName0,&Focus0)=false
          ===================================================
          substitute(&Module0,&Focus0, &Sort1, &Sort2)=&Module0
[subst-2] &Production0=&Symbols0 -> &Sort0 &Attributes0,
          focused(&Sort0,&Focus0)=false
          ===========================================================
          substitute(&Production0,&Focus0, &Sort1, &Sort2)=&Production0
[subst-3] substitute(&Sort0, &Focus0,&Sort0, &Sort1)=&Sort1
```

Fig. 5. Definition of sort substitution with traversal function

Traversal functions assume that the data structure to be traversed is passed as the first parameter to them. There are two essential scenarios covered by the current traversal extension. Firstly, one can model type-preserving functions where the traversed type coincides with the result type. Secondly, one can accumulate during traversal starting from a supplied initial value. Actually, both schemata can also be combined in a beneficial manner. Program transformations adhere to the former scheme, whereas program analyses can often be encoded following the latter scheme.

Substitution as described in Figure 5 is an example of the type-preserving scenario. There are a few things worth mentioning. The function `substitute` is attributed with `{traverse}` to indicate that it is a traversal function. Although, the function `substitute` is declared for type `SDF` (see the first parameter), the function is implicitly overloaded for all sorts reachable from `SDF`. The rewrite rules `[subst-1]` to `[subst-3]` deal with other types, namely `Module`, `Production`, and `Sort`. The core rule is `[subst-3]`. Encountering a sort during traversal which also coincides with the sort to be substituted, the sort is rewritten to the new sort. The other two equations deal with focus issues. The traversal should only descend into a module if the module is focused (cf. `[subst-1]`). Similarly, the traversal should only descend into a production if the nonterminal defined by the production is focused (cf. `[subst-2]`).

3 Basic concepts of FST

We describe the basic concepts underlying our framework FST. Essentially, we deliver a core language for grammar transformations. First, primitive grammar transformations are identified. Then, some auxiliary concepts are supplied, namely conditions, a focus concept, and yielders for symbolic operands in transformations. Finally, transformation combinators are added.

3.1 Primitives

Grammar transformations can be constructed from a small set of primitives. We will later see how combinators can be used to compose primitives, and to constrain them by pre- and post-conditions. The primitives for grammar transformations are declared in Figure 6. We use a dedicated sort `Trafo` to describe forms of transformations.

There are simple grammar transformations for identity and failure denoted by `id` and `fail`. There is an operator to discard the focused part of the given grammar denoted by `reset`. There are operators `add` and `sub` to add and to remove a production from a grammar. There are operators `replace` and `substitute` to replace SDF symbols or to substitute sorts. Finally, there are primitives to `introduce`, `delete` and `rename` modules. We declare an interpretation function `IT` to interpret transformations, that is, to apply them to a given grammar. The interpretation function takes a focus parameter as argument. Thereby, a transformation can be restricted to apply only to a certain part of the grammar. We only show the simple equation for the operator `substitute`. Interpretation is ultimately defined in terms of the traversal function which we defined before in Figure 5.

```
module Primitives
 imports Substitute Add Sub Replace Reset Yielders
 exports
  sorts Trafo
  context-free syntax
   "id" -> Trafo
   "fail" -> Trafo
   "reset" -> Trafo
   "add" ProductionYielder -> Trafo
   "sub" ProductionYielder -> Trafo
   "substitute" Sort "by" Sort -> Trafo
   "replace" SymbolsYielder "by" SymbolsYielder -> Trafo
   "introduce" "module" ModuleName -> Trafo
   "delete" "module" ModuleName -> Trafo
   "rename" "module" ModuleName "to" ModuleName -> Trafo
   "IT" "(" Trafo "," SDF "," Focus ")" -> SDF
   ...
equations
...
[IT-6] &SDF1=substitute(&SDF0,&Focus0,&Sort0,&Sort1)
       ====================================================
       IT(substitute &Sort0 by &Sort1,&SDF0,&Focus0)=&SDF1
...
```

Fig. 6. Primitives

3.2 Focus

Transformational grammar programming is to a large extent concerned with local changes. One way to realise this aspect in the framework is to restrict grammar transformations so that they only apply in a certain focus. This concept was already mentioned above for the primitive operator `reset`. The operators `replace` and `substitute` are also often applied in a focus. In Figure 7, all forms of focus and some helper functions are declared. The actual application of a grammar transformation in a focus relies on the `focus` combinator (cf. combinators in Figure 4).

As the first production in Figure 7 details, a focus is basically a compound entity with one part declaring what modules are in the focus, and another part for sorts. There are wild-card focus forms, that is, we can express that `all` modules or sorts are focused. Finally, there are two auxiliary functions `focused(...)` to check if a given module name or a given sort is covered by the focus at hand.

3.3 Yielders

We can basically apply primitive transformation operators to concrete symbols, productions and others. Often it is convenient to apply these operators rather to *symbolic* operands. By symbolic we mean that we are concerned with

```
module Focus
 imports Properties Bool
 exports
  sorts ModuleFocus SortFocus Focus
  context-free syntax
   "modules" ModuleFocus "sorts" SortFocus -> Focus
   Error -> Focus
   ModuleName+ -> ModuleFocus
   "all" -> ModuleFocus
   Sort+ -> SortFocus
   "all" -> SortFocus
  "focused" "(" ModuleName "," Focus ")" -> Bool
  "focused" "(" Sort "," Focus ")" -> Bool
  ...
equations
...
```

Fig. 7. Focus

forms which are only turned into proper concrete symbols etc. by evaluation. We call these forms yielders, and the corresponding function for evaluation of yielders is called `IY`. This function is overloaded for each different type of yielders. In a sense, yielders model basic operations for the types used in grammar transformations.

In Figure 8, some forms of yielders and profiles for the overloaded evaluation function `IY` are defined. One non-trivial form covered by the figure is `definition of` s. This yielder evaluates to the definition of the sort s—if it exists, and there is only a single production for s.

3.4 Conditions

A disciplined style of transformational grammar programming is achieved if for each step the conditions of applicability are well-understood and enforced. If we want to, for example, eliminate a sort, we pretend that it is not needed anymore. Thus, in terms of pre- and post-conditions, we need to check that the sort is fresh once we removed its definition. Several other properties for sorts, modules, and symbols are defined in Figure 9. Conditions are used to formulate conditional transformations based on `if ...` (cf. combinators in Figure 4).

There are sort conditions to check if a sort is a

- `fresh` sort, i.e., it does not occur at all,
- `bottom` sort, i.e., it is used but not defined,
- `defined` sort, i.e., there is at least one defining production,
- `top` sort, i.e., it is not used, except maybe in its own definition.

As for modules, we can test if a module is `empty` and others. For two sequences

```
module Yielders
 imports Auxiliary
 exports
  sorts SymbolsYielder ProductionYielder
  context-free syntax
   Symbols -> SymbolsYielder
   Pattern+ -> SymbolsYielder
   "definition" "of" Sort -> SymbolsYielder
   "permutation" "of" SymbolsYielder -> SymbolsYielder
   SymbolsYielder "->" Sort Attributes -> ProductionYielder
   ...
   "all" "sorts"-> SortFocusYielder
   "sorts" Sort+ -> SortFocusYielder
   ...
   "all" "modules" -> ModuleFocusYielder
   "modules" ModuleName+ -> ModuleFocusYielder
   ...
   SortFocusYielder "in" ModuleFocusYielder -> FocusYielder

   "IY" "(" SymbolsYielder "," SDF "," Focus ")" -> Symbols
   "IY" "(" ProductionYielder "," SDF "," Focus  ")" -> Production
   "IY" "(" FocusYielder "," SDF ")" -> Focus
   ...
equations
...
```

Fig. 8. Yielders

```
module Conditions
 imports Covers Yielders
 exports
  sorts Cond Bool
  context-free syntax
   "fresh" "sort" Sort -> Cond
   "bottom" "sort" Sort -> Cond
   "defined" "sort" Sort -> Cond
   "top" "sort" Sort -> Cond
   SymbolsYielder "covers" SymbolsYielder -> Cond
   SymbolsYielder "equal" SymbolsYielder -> Cond
   "fresh" "module" ModuleName -> Cond
   "defined" "module" ModuleName -> Cond
   "empty" "module" ModuleName -> Cond
   "IC" "(" Cond "," SDF ")" -> Bool
   "not" Cond -> Cond
   ...
equations
...
```

Fig. 9. Conditions

of symbols, we can check if they are `equal` modulo some simplification rules, or if the first `covers` the second, that is, the first can be derived into the second.

The symbol $A|A$, for example, is equal to A. The symbols $A*\ B$, for example, cover the symbols $A\ B$. We should point out that many of these conditions are implemented as traversal functions, in particular those conditions dealing with sorts and modules in Figure 9. The function `IC` evaluates conditions. It is convenient, to assume forms of conditions for the common logical operations (cf. `not ...` in Figure 9).

3.5 Combinators

The module for combinators was already briefly explained in Section 2.2 (also refer to Figure 4). We only want to summarise these combinators. Sequential composition of transformation t_1 and t_2 is denoted by $t_1; t_2$. To apply a transformation t in a focus f, we use the form `focus on` f `do` t. The focus parameter f is actually an expression *yielding* a focus. Thereby, we can express things like "focus on all sorts reachable from ...". Conditional transformations are of the form `if` c `then` t_1 `else` t_2. Here, both t_1 and t_2 are transformations, and c is a condition. The concept of conditional transformation is used to enforce pre- and post-conditions. We have an extra shorthand `guard` c to prefix or postfix transformations by conditions, say guards. The shorthand is defined as `if` c `then id else fail` (cf. rule `[guard-1]` in Figure 4). Finally, there is the form t `effectively` which behaves like t but with the only exception that it fails if t preserves the given grammar. We usually want to refuse transformation steps which do not have any effect.

4 Derived operators

We define a number of derived transformation operators which correspond to transformation techniques needed in actual transformational programming. We describe groups for refactoring, construction, destruction. There also hybrids (cf. module `ZigZag` in Figure 2) which we will skip for brevity.

4.1 Refactoring

The group of transformations for refactoring comprises several semantics-preserving transformations. By semantics-preserving we mean that the language generated by the grammar is preserved. The corresponding formal discussion can be found in [12]. The general idea of refactoring is to change the structure of a grammar so that the grammar becomes more comprehensible, better accessible for subsequent changes, or feasible or more efficient for implementation in a certain way. Several operators which we have identified are also very common in other settings of semantics-preserving transformation, e.g., fold/unfold transformations in the context of logic and functional programming [17].

The types of the refactoring operators are declared in Figure 10. For brevity, we only show the actual definition of the operator `rename`. As the

```
module Refactoring
 imports Combinators Move YACCify DeYACCify Alternation
 exports
  context-free syntax
   "rename" Sort "to" Sort -> Trafo
   "fold" SymbolsYielder "to" Sort -> Trafo
   "unfold" Sort -> Trafo
   "introduce" Sort "as" SymbolsYielder -> Trafo
   "eliminate" Sort -> Trafo
   "equate" Sort "to" Sort -> Trafo
   "simplify" SymbolsYielder "to" SymbolsYielder -> Trafo
   "eliminate" "module" ModuleName -> Trafo
equations
[ren-1] rename &Sort0 to &Sort1 =
          guard fresh sort &Sort1;
          substitute &Sort0 by &Sort1 effectively;
          guard fresh sort &Sort0
...
```

Fig. 10. Refactoring

rewrite rule `[ren-1]` details, the operator `rename` is derived from the primitive `substitute`. The way the operator is constrained, it is made sure that the first sort is consistently renamed to the second. As an aside, the post-condition can only fail if the operator is applied in a (too restrictive) focus. *All* derived operators are compositions of primitives constrained by guards to ensure pre- and post-conditions.

Let us carry on with the other refactorings declared in Figure 10. There are operators to `fold` and `unfold` productions. There are two other dual operators to `introduce` and to `eliminate` sorts (say definitions). There is an operator to `equate` two sorts because they are defined the same anyway, and there is an operator to `simplify` SDF symbols. The operator `eliminate` is overloaded so that one can also eliminate modules. Recall that the introduction of modules was already covered by a corresponding primitive (cf. Figure 6).

Some of the slightly more complex refactorings are not declared directly in the `Refactoring` module, but rather in the imported modules (cf. `imports ...` in Figure 10). The module `Move` defines an operator `move` which facilitates intra-modular and inter-modular moves of productions. Within a module, moves are sometimes needed to enforce a certain favoured order. Inter-modular moves are needed for modularisation or demodularisation. The concepts from the other imported modules `YACCify`, `DeYACCify`, and `Alternation` will be explained in the application section (cf. Section 5).

4.2 Construction and destruction

As a matter of fact, refactoring is not sufficient in transformational grammar programming. When a grammar evolves during development or maintenance, the grammar usually has to be changed in a more general way. We consider two deviations from refactoring. The first class covers so-called construction operators. The other class covers the opposite concept, that is, destruction. Let us first consider construction. The idea here is that these operators extend the grammar, that is, they are in a sense constructive. There are in turn two ways how construction can be performed. We might provide definitions for undefined sorts. We might also generalise existing definitions.

```
module Construction
 imports Combinators Move
 exports
  context-free syntax
   "generalise" SymbolsYielder "to" SymbolsYielder -> Trafo
   "include" SymbolsYielder "in" Sort -> Trafo
   "resolve" Sort "as" SymbolsYielder -> Trafo
   "unify" Sort "to" Sort -> Trafo
   ...
equations
[gen-1] generalise &SymbolsYielder1 to &SymbolsYielder2 =
          guard &SymbolsYielder2 covers &SymbolsYielder1;
          replace &SymbolsYielder1 by &SymbolsYielder2 effectively
...
[unify-1] unify &Sort1 to &Sort2 =
            guard bottom sort &Sort1;
            guard not fresh sort &Sort2;
            replace &Sort1 by &Sort2;
            guard fresh sort &Sort1
```

Fig. 11. Construction

In Figure 11, several forms of construction are defined. We can `generalise` symbols. Note the pre-condition to ensure that the phrase corresponding to the first operand indeed is subsumed by the phrase corresponding to the second operand. Other ways of construction are to `include` a production, and to `resolve` a so-far undefined sort. The last operator for construction which we want to mention here is `unify`. The operator is supposed to connect, in a way, a given bottom sort with another sort. Again, the guards convey essential ideas. The pre-condition checks that the second sort is not fresh. Otherwise, unification degenerates to renaming. There is also a post-condition for `unify` checking that sort substitution was done in exhaustive manner, that is, the focus was not set too specific.

In Figure 12, the opposites of the construction operators are declared. Symbols can be `restrict`ed. Productions can be `exclude`d (as long as the underlying sort does not get undefined thereby). Definitions of sorts can be entirely `reject`ed. The definition of the operator `reject` is a suitably restricted `reset`.

```
module Destruction
 imports Combinators
 exports
  context-free syntax
   "restrict" SymbolsYielder "to" SymbolsYielder -> Trafo
   "exclude" SymbolsYielder "from" Sort -> Trafo
   "reject" Sort -> Trafo
   "reject" "module" ModuleName -> Trafo
   "seperate" Sort "as" Sort -> Trafo
equations
...
```

Fig. 12. Destruction

5 Applications of FST

We discuss a few applications of grammar transformations. We illustrate the applications with IBM's COBOL dialect VS COBOL II. This, of course, does not imply in any sense that our technology is restricted to COBOL-like languages. First, we study applications in the context of grammar *implementation.* Then, we deal with grammar *recovery.* Finally, grammar transformations are considered in the context of grammar *reengineering.*

5.1 Grammar implementation

Transformations are useful in grammar implementation as we will indicate with a few scenarios. Transformations can be used, for example, in conflict resolution and disambiguation, YACCification, and grammar minimalisation.

Disambiguation

Grammars serving as references are tuned towards readability. When it comes to parser implementation, the grammars at least need to be disambiguated. Depending on the chosen parser technology, conflict resolution has to be performed, too. Let us consider a typical VS COBOL II sample. According to COBOL terminology, we should separate plain data items, and structured, possibly nested record descriptions. The VS COBOL II standard [10] encourages this separation by using two distinct identifiers, namely

- `Record-description-entry` and
- `Data-item-description-entry`.

Unfortunately, at some point the standard regulates that the syntax for such entries is described by `Data-description-entry`. The latter nonterminal describes flat entries which might be part of a record or not. Indeed, it is hardly possible to syntactically separate data items and records because of the way level numbers are used in COBOL to describe the nesting structure of records. The input grammar fragment in Figure 13 is clearly ambiguous because of the coinciding definition of `Record-description-entry` and `Data-item-description-entry`. To disambiguate the grammar, we perform the transformation steps shown in the figure. Firstly, we unfold the definitions of the aforementioned sorts. Secondly, we simplify the ambiguous expression without affecting the generated language. Finally, we can eliminate the definitions of the obsolete nonterminals.

Ambiguous grammar fragment

```
...
("FILE" "SECTION" "."
 (File-and-sort-description-entry Record-description-entry+)*)?
("WORKING-STORAGE" "SECTION" "."
 (Record-description-entry | Data-item-description-entry)*)?
("LINKAGE" "SECTION" "."
 (Record-description-entry | Data-item-description-entry)*)?
   -> Data-division-content
Data-description-entry -> Data-item-description-entry
Data-description-entry -> Record-description-entry
...
```

Semantics-preserving transformation

```
focus on sort Data-division-content do
 unfold Record-description-entry;
focus on sort Data-division-content do
 unfold Data-item-description-entry;
focus on sort Data-division-content do
 simplify Data-description-entry | Data-description-entry
       to Data-description-entry;
eliminate Record-description-entry;
eliminate Data-item-description-entry;
```

Disambiguated grammar fragment

```
...
("FILE" "SECTION" "."
 (File-and-sort-description-entry Data-description-entry+)*)?
("WORKING-STORAGE" "SECTION" "."
 Data-description-entry*)?
("LINKAGE" "SECTION" "."
 Data-description-entry*)?
  -> Data-division-content
...
```

Fig. 13. Disambiguation sample for VS COBOL II

YACCification

If the parser description language at hand does not support extended BNF, or if the actual application of a grammar requires that only simple BNF forms are to be used (e.g., in the case of standard attribute grammars), then we

```
[optional] eliminate-optional &Symbol1 use &Sort1 =
           introduce &Sort1 as &Symbol1;
           include  in &Sort1;
           replace &Symbol1? by &Sort1 effectively
```

Fig. 14. Elimination of optionals (from module `YACCify`)

need to perform an adaptation for elimination of extended BNF patterns (optionals, lists, and nested alternatives). We call this kind of transformation YACCification (cf. module `YACCify` in Figure 2). To eliminate, for example, an optional, we resort to a scheme where an optional is modelled with an auxiliary nonterminal. This simple idea is illustrated with the derived operator `eliminate-optional` in Figure 14.

The elimination of all extended BNF phrases can be done by the repeated application of operators like the one given. Another possibility is that YACCification is done automatically in exhaustive manner. Then, unique sort names for the auxiliary nonterminals have to be generated (cf. concept *IdentifierGeneration* in Figure 2). The resulting grammar will however be less readable. Therefore, it is often favourable to leave it in the responsibility of the grammar programmer to flatten the grammar step by step using transformations. Thereby, sensible sort names and phrases can be identified.

Grammar minimalisation

The scenario of grammar minimalisation is less of a standard idea. By minimalisation we mean a kind of grammar specialisation according to an available code base. Such a minimalisation is motivated by applications from automated software renovation [20,14], where one needs to implement program transformations or source-to-source translations. Automatic grammar minimalisation helps to reduce the effort in this domain. If the code base at hand covers only a smaller part of the full grammar, this can be measured and made explicit in a minimised grammar. The resulting grammar serves as a much more precise contract for grammar-based tools. The effort for developing tools for automated software renovation indeed strongly depends on the number of patterns to be handled, and on the fact if the necessity to handle a certain pattern can be safely and easily determined and maintained. The idea of automated grammar minimalisation is illustrated in Figure 15.

We do not explain here how coverage measurement is to be performed (cf. [13] for the underlying theory). Once the coverage is available, the grammar can be specialised by `restrict`, `exclude`, and `eliminate`. To gain some precision, a "context-dependent" coverage notion can be taken into account where coverage of productions is measured relative to the occurrence in which the relevant nonterminal occurs. To realise this more precise measurement in the minimalised grammar, applications of `fold` and `unfold` are also due.

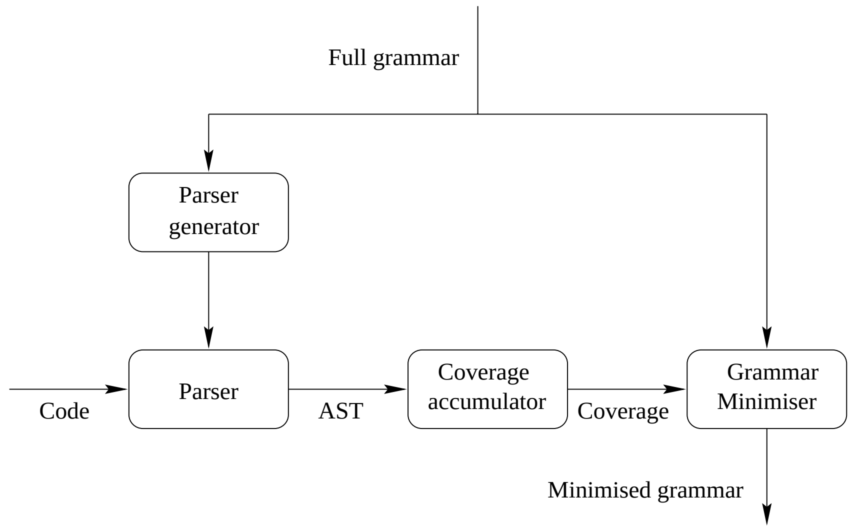

Fig. 15. Grammar minimalisation

5.2 Grammar recovery

The transformation from the introduction provides a good example for grammar recovery. In the example, the aim was to recover the correct definition of subscripts from IBM's VS COBOL II standard. A global account on grammar recovery is given in [14]. To recover a relatively correct and complete VS COBOL II grammar from the raw grammar contained in IBM's standard, we had to perform about 300 transformation steps. We can hardly include all the transformations verbatim. Let us describe the kinds of correctness and completeness problems we encountered:

Preparation As a result of the extraction from a semi-formal language reference, some sort names were entirely unsuitable because some of them had to be generated or composed using heuristics. Also, some obviously redundant or obsolete definitions could be removed.

Connectivity The introductory example from Figure 1 was an example for connectivity problems: A sort for a certain construct is not defined, although the intended construct is subsumed by some other sort. This problem pops up for IBM's VS COBOL II reference because the developers preferred to define some constructs rather in their context than separately. This style was chosen for a debatable convenience of reading, but it destroys the connectivity of the grammar.

Lack of definition The definitions of certain sorts were entirely missing in the grammar contained in the standard, e.g., arithmetic expressions. The definitions are then usually defined in natural language. With some decent COBOL-knowledge one could turn the text into productions. We could not identify a good reason to resort to textual explanations instead of concise context-free grammar productions.

Extensions and relaxations Many symbols were too restrictive. A com-

Restrictive grammar fragment

```
...
Alphabet-clause*
Symbolic-characters-clause*
Class-clause*
Currency-sign-clause?
Decimal-point-clause? -> Special-names-paragraph-clauses
...
```

Generalising transformation

```
focus on
 sort Special-names-paragraph-clauses
do
 generalise
  definition of
   Special-names-paragraph-clauses
 to
  permutation of
   definition of
    Special-names-paragraph-clauses
```

Grammar fragment with permutation phrase

```
...
<< Alphabet-clause*
   Symbolic-characters-clause*
   Class-clause*
   Currency-sign-clause?
   Decimal-point-clause? >> -> Special-names-paragraph-clauses
...
```

Fig. 16. Sample for introduction of permutation phrase

mon example is that certain constructs were obligatory in Standard COBOL, but optional in IBM's VS COBOL II. There are more difficult examples, e.g., regarding abbreviated combined relation conditions. There are 12 informal rules for how parentheses can be placed in abbreviations. Most other problems for extensions or relaxations were easy to identify and implement. A subclass of problems concerns permutation phrases [8], that is, if the order of certain subphrases is immaterial. In Figure 16, we show how a permutation phrase is enforced for the `SPECIAL NAMES` paragraph of COBOL. We `generalise` the `definition` of the nonterminal for the relevant clauses to the corresponding `permutation` phrase. Note that the form `permutation of ...` is a form of yielder. The SDF notation for permutation phrases is `<< ... >>`.

Grammar transformations trace adaptations of a grammar in a convenient manner. In the VS COBOL II project, we modularised the transformation script so that it becomes clear what transformation is concerned with what problems. We have 15 subscripts which deal, for example, with the aforementioned class of problems called preparation, with certain parts of the COBOL syntax, and other problems like the proper resolution of bottom sorts in terms of lexical sorts.

5.3 Grammar reengineering

Software reengineering for grammars can unsurprisingly be called grammar reengineering. We refer to [20] for a global account on reengineering language descriptions and related issues. The overall idea of grammar reengineering is that syntax descriptions are reengineered in order to become fit for other purposes, e.g., for redocumentation, and tool-generation. Reengineering comprises activities like the following:

- removal of semantic actions intertwined with a parser description,
- refactoring or normalisation in a broad sense,
- modularisation, and
- deYACCification (introduction of optionals, lists and that alike).

Let us discuss some activities in detail.

Modularisation
Modular grammars are fully supported by SDF and the accompanying tools. Modularity is useful simply for the reason that a modular grammar is usually more comprehensible. Given a complex language like COBOL, the modular hierarchy provides a good entry point to study the grammar and the language. The module hierarchy which we derived for our VS COBOL II case is shown in Figure 17. Modular grammars are also appropriate to cope with dialect problems. Together with some features of SDF, modularisation also allows us to separate the core syntax from declarations for disambiguation.

Modularisation is easily accomplished by transformations. Note that modularisation can be conceived as a form of refactoring because it does not affect the language generated by the grammar. We have the operators `introduce` and `eliminate` for the introduction and elimination of modules. We use the operator `move` to move productions between the modules. In Figure 18, we illustrate the introduction and inhabitation of one module for VS COBOL II, namely the module for arithmetic expressions. If we do not want to enumerate all sorts to be moved, we can also use a closure operator to include all sorts reachable from some distinguished set.

FST supports modularisation and modular syntax not just in the sense that corresponding operators are offered. The framework also takes care of the module interfaces. The imports of modules, and the exports of sorts are automatically derived from the productions at hand. This approach does not just enforce consistent interfaces, but it also promotes a homogeneous style of import and export.

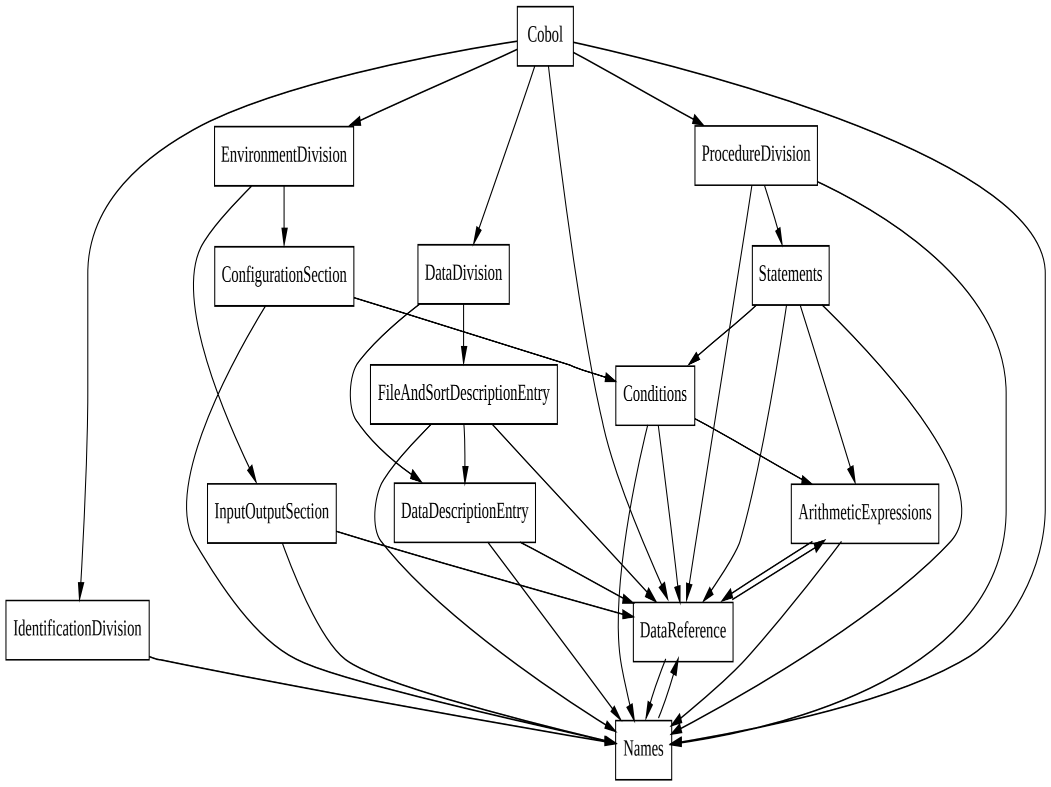

Fig. 17. Modular COBOL grammar

```
introduce module ArithmeticExpression;
focus on module Cobol do
 move
  sorts Arithmetic-expression Times-div Power Basis
 to
  module ArithmeticExpression;
```

Fig. 18. Excerpt of the modularisation script for VS COBOL II

DeYACCification

Given a grammar which does not use extended BNF, one can derive a more readable, richer grammar with optionals, list constructs, and others. This has been, for example, suggested in [20]. We call the corresponding grammar reengineering technique deYACCification (cf. module `DeYACCify` in Figure 2). Clearly, this approach is the inverse of YACCification discussed earlier. The technique is also related to the problem of the derivation of abstract from concrete syntax studied in [23]. In Figure 19, we show some schemata for deYACCification.

In the schemata, the sort variable $\langle Y \rangle$ is the place-holder for the auxiliary nonterminal used in the puritanical notation (usually YACC). The replacement is shown as a grammar transformation including elimination steps. As for the schemata dealing with separated lists, we only eliminate $\langle R \rangle$ (which is intended to cover the tail of a separated list) if it is a top sort.

Description	Pattern	Replacement
Optionals	$\rightarrow \langle Y\rangle$ $\langle X\rangle \rightarrow \langle Y\rangle$	replace$\langle Y\rangle$ by $\langle X\rangle?$; eliminate $\langle Y\rangle$
Star-lists	$\rightarrow \langle Y\rangle$ $\langle X\rangle\ \langle Y\rangle \rightarrow \langle Y\rangle$	replace$\langle Y\rangle$ by $\langle X\rangle^*$; eliminate $\langle Y\rangle$
Plus-lists	$\langle X\rangle \rightarrow \langle Y\rangle$ $\langle X\rangle\ \langle Y\rangle \rightarrow \langle Y\rangle$	replace$\langle Y\rangle$ by $\langle X\rangle^+$; eliminate $\langle Y\rangle$
Separated star-lists	$\rightarrow \langle Y\rangle$ $\langle X\rangle\ \langle R\rangle \rightarrow \langle Y\rangle$ $\rightarrow \langle R\rangle$ $\langle S\rangle\ \langle X\rangle\ \langle R\rangle \rightarrow \langle R\rangle$	replace$\langle Y\rangle$ by $\{\langle X\rangle\ \langle S\rangle\}^*$; eliminate $\langle Y\rangle$; if top $\langle R\rangle$ eliminate $\langle R\rangle$ else id
Separated plus-lists	$\langle X\rangle\ \langle R\rangle \rightarrow \langle Y\rangle$ $\rightarrow \langle R\rangle$ $\langle S\rangle\ \langle X\rangle\ \langle R\rangle \rightarrow \langle R\rangle$	replace$\langle Y\rangle$ by $\{\langle X\rangle\ \langle S\rangle\}^+$; eliminate $\langle Y\rangle$; if top $\langle R\rangle$ eliminate $\langle R\rangle$ else id

Fig. 19. Schemata for deYACCification

Normal-forms

Besides deYACCification, there are other schemata useful in grammar reengineering. Sometimes we need to derive a flat definition of $\langle X\rangle$ from a non-flat definition and vice versa. These concepts are illustrated in Figure 20 (and they are exported by the FST module `Alternation`). By flat we mean that the $\langle X\rangle$ is defined as list of top-level alternatives in one production. Consequently, by non-flat we mean that $\langle X\rangle$ is defined by multiple productions, where each single production has no top-level alternatives. A non-flat form is usually more readable especially if the alternatives are complex. A flat definition is needed if $\langle X\rangle$ should be unfolded.

$$\begin{array}{l} \langle A_1\rangle \rightarrow \langle X\rangle \\ \ldots \\ \langle A_n\rangle \rightarrow \langle X\rangle \end{array} \quad \Longleftrightarrow \quad \langle A_1\rangle \mid \cdots \mid \langle A_n\rangle \rightarrow \langle X\rangle$$

Fig. 20. Non-flat vs. flat definition of $\langle X\rangle$

Implementation of schemata

Conceptually, it is clear that the schemata like those in Figure 19 and Figure 20 describe refactorings which are, of course, semantics-preserving. Technically, schemata are not expressible in the current framework as derived operators because the required kind of matching is not (yet) possible in FST. We only *can* encode the transformation for replacement underlying a scheme as shown in Figure 19. However, schemata can be encoded as rewrite rules (as opposed to derived operators). Such a rewrite rule matches the given grammar against patterns like those in Figure 19 so that schema variables get bound, and then grammar transformations are performed using these bound variables. The application of the "scheme as rewrite rule" idea is a bit tedious. We have to enforce certain side conditions: We should not be sensitive w.r.t. the order of the productions in the input scheme. We also have to test that the instantiated rules from the input scheme are all productions for $\langle Y \rangle$.

Exhaustive scheme application corresponds simply to repeated application of rewrite rules encoding a scheme. Note that a set of schemata, when considered as rewrite rules in a naive manner, might violate confluence. Consider, for example, the given scheme for optionals. Its application might disable intended applications of the scheme for separated lists.

6 Concluding remarks

We have reported on the design, the prototype implementation, and the applications of FST—a *F*ramework for *S*DF *T*ransformation. FST supplies primitive operators, auxiliary concepts, and derived operators for refactoring, construction, and destruction. The formal underpinnings of grammar adaptation are described in [12].

There is hardly related work on grammar transformations. There are some folklore transformations, like EBNF to BNF reduction, elimination of left-recursion, transformation to Chomsky-Normal-form, but previously, there was no framework covering the transformations performed by grammar programmers manually otherwise. In the emerging XT project [24] for program transformation, some grammar transformation tools are also included or developed. The transformational style to grammar adaptation supported by FST has been largely motivated by the general idea of transformational programming [16].

Operator suites and corresponding transformation systems exist for other domains than grammar programming, e.g., in logic programming [1,7], or for main-stream programming languages [2]. Operator suites and tool support for program adaptation are also well-established concepts in object-oriented programming. In [19,15], for example, refactoring tools are described.

The choice for the ASF+SDF Meta-Environment as platform for a prototype implementation was driven by some of its key features. The integration with the powerful syntax definition formalism SDF allows us to cover a representative part of notation for concrete (and abstract) syntax. The algebraic specification formalism ASF allows us to develop formal and executable specifications of the FST concepts. The recent addition of traversal functions provides us with a means to define the many transformations and analyses in FST in a concise manner.

A preliminary form of traversal functions based on generating the traversal functions was described in [6]. The present implementation fully transparently integrates traversal functions with ASF+SDF and the rewrite engine. The design and the implementation is discussed in full detail in [4]. Many of our grammar transformations and the contributing grammar analyses take advantage of traversal functions. At the time of writing this article, FST uses 24 traversal functions with only a few rewrite rules per function. The SDF grammar itself has about 100 relevant productions. This is a remarkable indication for the usefulness of the support for traversal functions. In worst case, we would have to deal with about 2400 rewrite rules otherwise.

Finally, we want to point out a few directions for future work. We have not yet fully understood the role of SDF disambiguation constructs in the context of grammar transformations. A conceptual problem with the current FST is that schemata as for deYACCification are not first-class citizens, i.e., they cannot be encoded as derived operators. We would like to extend FST so that concepts of matching and iteration can be applied for operator definition.

References

[1] F. Alexandre, K. Bsaies, J.P. Finance, and A. Quere. Spes: A System for Logic Program Transformation. In A. Voronkov, editor, *Logic Programming and Automated Reasoning, LPAR'92*, volume 624 of *LNCS*, pages 445–447. Springer-Verlag, 1992.

[2] I. Attali, V. Pascual, and C. Roudet. A language and an integrated environment for program transformations. Rapport de recherche 3313, INRIA, December 1997.

[3] J. A. Bergstra, J. Heering, and P. Klint. The Algebraic Specification Formalism ASF. In *Algebraic Specification*, chapter 1, pages 1–66. The ACM Press in cooperation with Addison-Wesley, 1989.

[4] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term Rewriting with Traversal Functions. Technical report, CWI, Amsterdam, 2001. in preparation.

[5] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In *Compiler Construction 2001 (CC 2001)*, LNCS. Springer, 2001. To appear.

[6] M.G.J. van den Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000.

[7] J. Brunekreef. TransLog, an Interactive Tool for Transformation of Logic Programs. Technical Report P9512, University of Amsterdam, Programming Research Group, December 1995.

[8] R.D. Cameron. Extending context-free grammars with permutation phrases. *ACM Letters on Programming Languages and Systems*, 2(4):85–94, March 1993.

[9] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[10] IBM Corporation. *VS COBOL II Application Programming Language Reference*, 1993. Release 4, Document number GC26-4047-07.

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

[12] R. Lämmel. Grammar Adaptation. In *Proc. Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, Springer-Verlag, 2001.

[13] R. Lämmel. Grammar Testing. In *Proc. Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, Springer-Verlag, 2001.

[14] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. Submitted, available at `http://www.cwi.nl/~ralf/`, July 2000.

[15] I. Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proc. OOPSLA '96: Object-Oriented Programming Systems, Languages, and Applications*, pages 235–250. ACM Press, 1996.

[16] H.A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[17] A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, June 1996.

[18] J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992.

[19] D. Roberts, J. Brant, and R.E. Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[20] A. Sellink and C. Verhoef. Development, Assessment, and Reengineering of Language Descriptions. In J. Ebert and C. Verhoef, editors, *Proc. Fourth European Conference on Software Maintenance and Reengineering*, pages 151–160. IEEE Computer Society, March 2000.

[21] M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers, 1985.

[22] E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997.

[23] D.S. Wile. Abstract Syntax From Concrete Syntax. In *Proc. of the 1997 International Conference on Software Engineering*, pages 472–480. ACM Press, 1997.

[24] XT: Program Transformation Tools, 2000–2001. http://www.program-transformation.org/xt/.