Declarative aspect-oriented programming*

Ralf Lämmel

Department of Computer Science
University of Rostock
D-18051 Rostock
Germany

Abstract

Aspect-oriented programming addresses the problem that the implementation of some properties such as error handling and optimization tends to cross-cut the basic functionality. To overcome that problem special languages are used to specify such properties—the so-called aspects—in isolation. The software application is obtained by weaving the aspect code and the implementation of properties corresponding to basic functionality—the so-called components. This paper investigates the suitability of functional meta-programs to specify aspects and to perform weaving. The proposal focuses on the declarative paradigm (logic programming, attribute grammars, natural semantics, constructive algebraic specification etc.) as far as components are concerned, whereas aspects are represented by program transformations. Weaving is regarded as a program composition returning a combination of the components satisfying all the aspects. The computational behaviour of the components is preserved during weaving. The proposal improves reusability of declarative programs. The approach is generic in the sense that it is applicable to several representatives of the declarative paradigm. Several roles of aspect code are defined and analysed.

1 Introduction

1.1 Aspect-oriented programming

Kiczales et al. recently proposed aspect-oriented programming (AOP) [KLM+97, AOP97, AOP98] as an extension of the traditional approach to programming coping well with functional decomposition. Procedures, functions, methods, modules, APIs, classes etc. are used in the traditional approach to implement all kind of properties. However, certain properties such as error handling and optimization tend to cross-cut the functionality resulting in tangled code which is then unclear and hard to modify and adapt. AOP attempts to close this well-known gap between requirements / design and implementation.

AOP is based on the following assumptions. Properties which must be implemented are subdivided into *components* corresponding to basic functionality and *aspects* corresponding to non-functional properties. A property is a *component* if it can be cleanly encapsulated in a procedure, a method etc. Otherwise it is an *aspect*. Components tend to be units of the system's functional decomposition, whereas aspects cross-cut the system's functionality and they usually affect the performance or semantics of the components in systemic ways. Thus, components are usually developed in an imperative / object-oriented language, whereas special language support is needed for the development of aspect code. The application is obtained by *weaving* the components and the aspects. Note that without AOP the properties corresponding to aspects have to be scattered throughout the code representing the basic functionality.

Example 1 Suppose persistency must be added to an object-oriented program. The implementation of the property "persistency" cross-cuts the classes implementing the basic functionality because the classes need to be adapted in a *systemic* way to "externalize" their state and to support "population" when objects are re-created.

Let us consider a very specific aspect in the context of persistency, that is to say *efficient* persistency where a persistency manager should keep track of the objects which have changed their state. Thereby, the update of the persistent storage can be done more efficiently.

Given a class *foo*, for example, the instances of which should be persistent, a method call like the one in the box below must be inserted for every change of the internal state.

```
class foo extends ··· {
// attributes
  type1 attr1;
  ···

// methods
  public result_type1 operation1 (···) {
    ···
    attr1 = ···; // changing the state
    this.keep_persistency_manager_informed();
    ···
  }
  ···
}
```

Thus, an aspect language is needed to specify this kind of systematic adaptation. Somehow it must be possible to specify the join point (i.e. the point where the aspect is woven into the component) "the statement after an assignment to an attribute". Weaving means to take some class definitions and the aspect code and to emit class definitions with the efficient persistency implemented. ◊

The original proposal of AOP [KLM+97] leaves open the question what actual languages for developing aspect code and what actual forms of weaving are appropriate. Since

*This work was supported, in part, by *Deutsche Forschungsgemeinschaft*, in the project *KOKS*.

then, several researchers have proposed potential aspect languages and forms of weaving, from readily available but dedicated weavers to sophisticated program transformations still to be implemented. One example in [KLM+97] deals with loop fusion (that is a kind of optimization), where the aspect code can be regarded as a set of rewrite rules acting on data flow diagrams derived from the component code. Weaving means here to build the data flow diagrams, to apply the rewrite rules and to emit C-code from the "optimized" data flow diagrams.

Thus, the fundamental question in AOP is what are the aspects, how to represent them and how to weave components and aspects. This paper attempts to answer these questions in a certain way.

1.2 Another instance of AOP

Our paper provides a general but still effective proposal for aspect code and weaving based on functional meta-programs. Components are declarative programs, whereas aspects are implemented by program transformations. Weaving is considered as a program composition combining components and aspects by essentially applying the transformations modelling aspects to components. Note that the original proposal of AOP and most work in the field focus on procedural (object-oriented) languages as far as components are concerned, whereas our concrete instance of AOP relies on declarative languages.

Typical examples of aspects in declarative programming are concerned with

- optimization, e.g. based on fold/unfold-strategies,
- failure handling, error recovery, exception handling,
- propagation, accumulation, synthesis of data,
- stylistic properties, e.g. CPS based on conversion, and
- refinements of the computational behaviour.

Our proposal for AOP improves *reusability* of declarative programs because aspects can be described in isolation. Thus, declarative programs can be programmed in a more modular fashion because they may abstract from the aspects. Our approach illustrates an amalgamation of program synthesis, program transformation, program analysis and program composition based on a generic framework for meta-programming in the declarative paradigm. The approach is generic in the sense that is applicable to several representatives of the declarative paradigm, e.g. natural semantics, attribute grammars, (constructive) algebraic specifications and logic programs. The framework for meta-programming is described in Section 2.

Some meta-programming operators modelling roles of aspect code are studied in Section 3. One particular form of weaving is presented in detail in Section 4. A general challenge in AOP is to ensure correctness of weaving, e.g. in the sense that the meaning of the components is preserved by the weaver. Otherwise, AOP would be unnerving because the person writing the aspects could foul the weaver and introduce unintended behaviour into the woven program. Most other works in the field of AOP focus on identification and characterization of aspects, aspect languages and possible forms of weaving. In contrast, this paper attempts to provide some theoretical grounds in the context of correctness. Reasoning is based on certain preservation properties of program transformations. The suggested form of weaving and the properties of the suggested roles of aspect code suffice to guarantee the preservation of the computational behaviour of the components by the weaver.

1.3 The running example

The discussion is rooted by small interpreter examples specified in the style of natural semantics[1]. Figure 1 shows an interpreter definition for a simple imperative language core. The natural semantics consists of two relations $do : \mathsf{C} \times \mathsf{ST} \to \mathsf{ST}$ describing how the execution of statements affects the store and $eval : \mathsf{E} \times \mathsf{ST} \to \mathsf{VAL}$ modelling expression evaluation free of side-effects. In the sense of AOP Figure 1 is a program implementing various properties related to the interpretation of basic language constructs. Note that it is not the intention of this paper to propose a certain style of developing interpreters for (simple) languages. The domain was rather chosen because the properties involved in simple interpreters are well-understood.

$$do(\mathsf{skip}, \mathsf{ST}) \to (\mathsf{ST}) \qquad [\mathsf{skip}]$$

$$\frac{do(\mathsf{C}_1, \mathsf{ST}) \to (\mathsf{ST}') \quad \wedge \quad do(\mathsf{C}_2, \mathsf{ST}') \to (\mathsf{ST}'')}{do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}) \to (\mathsf{ST}'')} \qquad [\mathsf{concat}]$$

$$\frac{eval(\mathsf{E}, \mathsf{ST}) \to (\mathsf{VAL}) \quad \wedge \quad update(\mathsf{ST}, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{ST}')}{do(\mathsf{assign}(\mathsf{ID}, \mathsf{E}), \mathsf{ST}) \to (\mathsf{ST}')} \qquad [\mathsf{assign}]$$

$$\ldots$$

$$\frac{apply(\mathsf{ST}, \mathsf{ID}) \to (\mathsf{VAL})}{eval(\mathsf{var}(\mathsf{ID}), \mathsf{ST}) \to (\mathsf{VAL})} \qquad [\mathsf{var}]$$

$$\ldots$$

Figure 1: An interpreter of a simple language

Now let us assume that we want to derive an interpreter coping with I/O as well. Figure 2 contains the corresponding interpreter rules. It is assumed that the input and the output are modelled by sequences (lists) of values. An expression of the form read is assumed to be evaluated to the head of the sequence of the current input; refer to the rule [read]. A statement of the form write(E) is assumed to output the value of the expression E; refer to the rule [write]. The rule [main] should be regarded as a kind of axiom for interpreting programs consuming an input and producing an output. The problem with the initial program in Figure 1 and the additional component in Figure 2 is that they are not compatible with each other. Figure 1 does not meet the properties "input propagation" and "output accumulation" which are obviously needed for the implementation of I/O in Figure 2. What is needed is an adaptation of Figure 1; refer to Figure 3. Meta-programs can be used to perform such adaptations, where the specifications in the style of natural semantics are regarded as target programs. Informally, the required adaptation can be performed in the following steps. Input and output positions of sort IN are added and the resulting parameters of sort IN are connected to code an accumulator. Output positions of sort OUT are added. Multiple parameters of sort OUT in a rule are combined in premises with the symbol *append*. For some rules the empty output is returned.

[1] The following conventions for natural semantics rules are assumed in this paper: $\to$ is used to separate inputs and outputs in propositions. Sort identifiers are used to derive variables of the sort by adding possibly quotes and indices, e.g. C_1 is a variable of sort C.

The key idea here is that "input propagation" and "output accumulation" are regarded as aspects modelled by program transformations. Weaving would be an elaboration of such a scenario, where several modules are adapted according to some aspects. All these issues are made clear in Section 4.

$$\frac{head(\mathsf{IN}) \to (\mathsf{VAL}) \quad \wedge \quad tail(\mathsf{IN}) \to (\mathsf{IN}')}{eval(\mathsf{read}, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{VAL}, \mathsf{IN}')} \text{ [read]}$$

$$\frac{eval(\mathsf{E}, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{VAL}, \mathsf{IN}') \quad \wedge \quad list(\mathsf{VAL}) \to (\mathsf{OUT})}{do(\mathsf{write}(\mathsf{E}), \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}, \mathsf{IN}', \mathsf{OUT})} \text{ [write]}$$

$$\frac{init \to (\mathsf{ST}) \quad \wedge \quad do(\mathsf{C}, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT})}{prog(\mathsf{C}, \mathsf{IN}) \to (\mathsf{OUT})} \text{ [main]}$$

Figure 2: I/O constructs

$$\frac{nil \to \mathsf{OUT}}{do(\mathsf{skip}, \mathsf{ST}, \boxed{\mathsf{IN}}) \to (\mathsf{ST}, \boxed{\mathsf{IN}}, \boxed{\mathsf{OUT}})} \text{ [skip]}$$

$$\frac{\begin{array}{l} do(\mathsf{C}_1, \mathsf{ST}, \boxed{\mathsf{IN}}) \to (\mathsf{ST}', \boxed{\mathsf{IN}'}, \boxed{\mathsf{OUT}_1}) \\ \wedge \quad do(\mathsf{C}_2, \mathsf{ST}', \boxed{\mathsf{IN}'}) \to (\mathsf{ST}'', \boxed{\mathsf{IN}''}, \boxed{\mathsf{OUT}_2}) \\ \wedge \quad \boxed{append(\mathsf{OUT}_1, \mathsf{OUT}_2) \to (\mathsf{OUT})} \end{array}}{do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}, \boxed{\mathsf{IN}}) \to (\mathsf{ST}'', \boxed{\mathsf{IN}''}, \boxed{\mathsf{OUT}})} \text{ [concat]}$$

$$\frac{\begin{array}{l} eval(\mathsf{E}, \mathsf{ST}, \boxed{\mathsf{IN}}) \to (\mathsf{VAL}, \boxed{\mathsf{IN}'}) \\ \wedge \quad update(\mathsf{ST}, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{ST}') \\ \wedge \quad \boxed{nil \to \mathsf{OUT}} \end{array}}{do(\mathsf{assign}(\mathsf{ID}, \mathsf{E}), \mathsf{ST}, \boxed{\mathsf{IN}}) \to (\mathsf{ST}', \boxed{\mathsf{IN}'}, \boxed{\mathsf{OUT}})} \text{ [assign]}$$

$$\ldots$$

$$\frac{apply(\mathsf{ST}, \mathsf{ID}) \to (\mathsf{VAL})}{eval(\mathsf{var}(\mathsf{ID}), \mathsf{ST}, \boxed{\mathsf{IN}}) \to (\mathsf{VAL}, \boxed{\mathsf{IN}})} \text{ [var]}$$

$$\ldots$$

Figure 3: An adaptation to cope with input/output

The remaining paper is structured as follows. First, a formal, generic framework for functional meta-programs is developed. Afterwards, certain operators facilitating program manipulation are outlined. These operators will be useful to implement aspects by means of meta-programs. We carry on by defining our instance of AOP. Finally, we comment on results, related work and future work.

2 The meta-programming framework

Our approach to the development of aspect code and to weaving is based on a meta-programming framework which is developed in the following steps. First, a representation for declarative target programs such as natural semantics, attribute grammars, logic programs and (constructive) algebraic specifications is declared. Second, the corresponding structural definitions are restricted to obtain the domains of proper target program fragments. These domains are embedded into a typed λ-calculus, where some further specification constructs are added as well. Finally, properties of meta-programs, e.g. preservation properties, are defined.

2.1 The representation of target programs

The representation of target programs (such as the rules in Figure 1 etc.) and fragments of them is given by certain domains which are intended to capture the common constructs in (first-order) declarative languages such as rules, propositions and parameters. There is, for example, a domain Rule which can be regarded as an abstraction from inference rules in natural semantics or syntactical rules together with the attributes + semantic rules in attribute grammars.

We assume the following domains:

- Rules *compatible* collections of rules
- Rule tagged rules
- Conclusion conclusions
- Premise premises
- Element parameterized names (propositions)
- Parameter annotated parameters such as variables
- Tag countable set of tags
- Name countable set of names
- Variable countable set of variable identifiers
- Sort countable set of sort identifiers

Example 2 The rules shown in Figure 1 form an element of Rules. There are rules (elements of Rule) tagged by [skip], [concat], [assign] and [var]. The conclusion of the rule [concat] is $do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}) \to (\mathsf{ST}'')$, whereas the premises of the rule are $do(\mathsf{C}_1, \mathsf{ST}) \to (\mathsf{ST}')$ and $do(\mathsf{C}_2, \mathsf{ST}') \to (\mathsf{ST}'')$. The conclusion and the premises are parameterized names, i.e. elements of Element. Parameters are variables, e.g. ST with several occurrences in [concat], or proper terms, e.g. $\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2)$. ♢

Refer to Figure 4 for the corresponding structural definition[2]. Note that barred names are used to point out that the structural definition needs to be restricted further to define the corresponding domain of *proper* fragments. Proper collections of rules, for example, are modelled by the domain $\overline{\mathsf{Rules}}$ obtained as a restriction of the structural definition $\overline{\mathsf{Rules}}$, where the restriction is concerned, for example, with compatibility of the types of all the single rules.

$$\begin{array}{lcl} \overline{\mathsf{Rules}} & = & \overline{\mathsf{Rule}}^* \\ \overline{\mathsf{Rule}} & = & \mathsf{Tag} \times \overline{\mathsf{Conclusion}} \times \overline{\mathsf{Premise}}^* \\ \overline{\mathsf{Conclusion}} & = & \overline{\mathsf{Element}} \\ \overline{\mathsf{Premise}} & = & \overline{\mathsf{Element}} + \cdots \\ \overline{\mathsf{Element}} & = & \mathsf{Name} \times \overline{\mathsf{Parameter}}^* \times \overline{\mathsf{Parameter}}^* \\ \overline{\mathsf{Parameter}} & = & (\mathsf{Variable} + \cdots) \times \mathsf{Sort} \end{array}$$

Refinement assumed in this paper

$$\begin{array}{lcl} \overline{\mathsf{Rules}} & = & \mathcal{P}(\overline{\mathsf{Rule}}) \\ \overline{\mathsf{Parameter}} & = & (\mathsf{Variable} + \overline{\mathsf{Term}}) \times \mathsf{Sort} \\ \overline{\mathsf{Term}} & = & \mathsf{Functor} \times \overline{\mathsf{Parameter}}^* \end{array}$$

Figure 4: Structural definition of representations

The structural definition has possibly to be refined and extended for particular instances. The domain Parameter for example needs to be extended for compound parameters in contrast to variables in the sense of terms in natural semantics. Another possible extension concerns the domain Premise which must be extended to cope with semantic rules

[2] $\mathcal{P}$ denotes the power set constructor. We are mainly concerned with finite subsets in this paper.

in attribute grammars. In this paper a refinement of the domains is assumed which copes with terms based on the following additional domains:

Term compound parameters (terms)
Functor countable set of functors for terms

Furthermore, we abstract from the order of rules in Rules. Thus, collections of rules are rather subsets of Rule than sequences. Refer to Figure 4 for the corresponding domain equations.

2.2 Properties of target programs

The structural definition from Figure 4 is restricted to obtain domains of proper fragments. Thereby, it can be guaranteed that meta-programs deal with rather proper fragments than arbitrary representations. Technically, inference rules are given to obtain the domains $\mathsf{Rules} \subset \overline{\mathsf{Rules}}$, ..., $\mathsf{Parameter} \subset \overline{\mathsf{Parameter}}$; refer to Figure 5[3]. Proper fragments are expected to satisfy

- well-formedness ($\mathcal{WF}$) in the sense of basic requirements, e.g. that the tags of a collection of rules are pairwise distinct or—more generally—that proper compound fragments are built only from proper fragments and
- well-typedness ($\mathcal{WT}$) in the sense of a type system with sorts and modes as used for example in logic programming [Boy96], i.e. it must be possible to associate profiles with all the symbols used in a target program fragment.

For *complete* target programs further properties can be relevant, e.g. certain completeness properties such as reducedness in the sense of context-free grammars (CFG) or non-circularity in the sense of attribute grammars. A particular property dealing with some minimal requirement concerning the data flow will be considered below.

$$\frac{\begin{array}{ll} & \overline{r}_i \in \mathsf{Rule} \text{ for } i = 1, \ldots, n \\ \wedge & \pi_{\mathsf{Tag}}(\overline{r}_i) \neq \pi_{\mathsf{Tag}}(\overline{r}_j) \text{ for } i, j = 1, \ldots, n, i \neq j \\ \wedge & \exists\, \Sigma : (\mathcal{WT}_{\mathsf{Rule}}(\Sigma, \overline{r}_i) \text{ for } i, \ldots, n) \end{array}}{\{\overline{r}_1, \ldots, \overline{r}_n\} \in \mathsf{Rules}} \quad [\mathsf{Rules}]$$

$$\frac{\begin{array}{ll} & \overline{r} \in \overline{\mathsf{Rule}} \\ \wedge & \pi_2(\overline{r}) \in \mathsf{Element} \\ \wedge & \pi_i(\pi_3(\overline{r})) \in \mathsf{Element} \text{ for } i = 1, \ldots, \#\pi_3(\overline{r}) \\ \wedge & \exists\, \Sigma : \mathcal{WT}_{\mathsf{Rule}}(\Sigma, \overline{r}) \end{array}}{\overline{r} \in \mathsf{Rule}} \quad [\mathsf{Rule}]$$

$$\frac{\begin{array}{ll} & \overline{e} \in \overline{\mathsf{Element}} \text{ is of the form } \langle n, in, out \rangle \\ \wedge & \pi_i(in) \in \mathsf{Parameter} \text{ for } i = 1, \ldots, \#in \\ \wedge & \pi_i(out) \in \mathsf{Parameter} \text{ for } i = 1, \ldots, \#out \\ \wedge & \exists\, \Gamma, \Sigma : \mathcal{WT}_{\mathsf{Element}}(\Gamma, \Sigma, \overline{e}) \end{array}}{\overline{e} \in \mathsf{Element}} \quad [\mathsf{Element}]$$

$$\frac{\begin{array}{ll} & \overline{p} \in \overline{\mathsf{Parameter}} \\ \wedge & \exists\, \Gamma, \Sigma : \mathcal{TYPE}_{\mathsf{Parameter}}(\Gamma, \Sigma, \overline{p}) = \pi_{\mathsf{Sort}}(\overline{p}) \end{array}}{\overline{p} \in \mathsf{Parameter}} \quad [\mathsf{Parameter}]$$

Figure 5: Properties of target program fragments

[3] π_i denotes the i-th projection for tuples and sequences. For a product $D = D_1 \times \cdots \times D_n$ the notation π_{D_i} is also used for π_i if the D is obvious from the context and the i is uniquely defined by the D_i. $\#s$ denotes the length of the sequence s.

$\mathcal{WF}$ is encoded directly in the inference rules in Figure 5, whereas most details of $\mathcal{WT}$ are defined by auxiliary relations shown in Figure 6.[4] Consider for example the inference rule [Rules] defining proper collections of rules. Its premises state the following properties:

- The single rules must be proper rules themselves ($\mathcal{WF}$).
- The tags must be pairwise distinct ($\mathcal{WF}$).
- The types of the rules must be compatible ($\mathcal{WT}$).

$$\frac{\begin{array}{ll} & \mathcal{WT}_{\mathsf{Rules}}(\Sigma, \overline{rs})) \\ \wedge & \Sigma \text{ is minimal, i.e. } \forall\, \Sigma' \neq \Sigma : \\ & \mathcal{WT}_{\mathsf{Rules}}(\Sigma', \overline{rs})) \Rightarrow |\Sigma| \leq \left|\Sigma'\right| \end{array}}{\mathcal{TYPE}_{\mathsf{Rules}}(\overline{rs}) \to \Sigma} \quad [\mathcal{WT}.1]$$

$$\frac{\mathcal{WT}_{\mathsf{Rule}}(\Sigma, \overline{r}_i) \text{ for } i = 1, \ldots, n}{\mathcal{WT}_{\mathsf{Rules}}(\Sigma, \{\overline{r}_1, \ldots, \overline{r}_n\})} \quad [\mathcal{WT}.2]$$

$$\frac{\exists\, \Gamma : (\mathcal{WT}_{\mathsf{Element}}(\Gamma, \Sigma, \overline{e}_i) \text{ for } i = 0, \ldots, n)}{\mathcal{WT}_{\mathsf{Rule}}(\Sigma, \langle t, \overline{e}_0, \langle \overline{e}_1, \ldots, \overline{e}_n \rangle \rangle)} \quad [\mathcal{WT}.3]$$

$$\frac{\begin{array}{ll} & s : \sigma_1 \times \cdots \times \sigma_m \to \sigma_{m+1} \times \cdots \times \sigma_k \in \Sigma \\ \wedge & \mathbf{in}_{\mathsf{Name}}(n) = s \\ \wedge & \mathcal{TYPE}_{\mathsf{Parameter}}(\Gamma, \Sigma, \overline{p}_i) \to \sigma_i \\ & \text{for } i = 1, \ldots, k \end{array}}{\mathcal{WT}_{\mathsf{Element}}(\Gamma, \Sigma, \langle n, \langle \overline{p}_1, \ldots, \overline{p}_m \rangle, \langle \overline{p}_{m+1}, \ldots, \overline{p}_k \rangle \rangle)} \quad [\mathcal{WT}.4]$$

$$\frac{\begin{array}{ll} & \exists\, v : \mathbf{in}_{\mathsf{Variable}}(v) = \pi_1(\overline{p}) \\ \wedge & \pi_{\mathsf{Sort}}(\overline{p}) = \sigma \\ \wedge & (v : \sigma) \in \Gamma \end{array}}{\mathcal{TYPE}_{\mathsf{Parameter}}(\Gamma, \Sigma, \overline{p}) \to \sigma} \quad [\mathcal{WT}.5]$$

Figure 6: The type system based on sorts and modes

Let us consider $\mathcal{WT}$ slightly more in detail. Symbols such as names used in elements (that is to say propositions) or functors used in terms get associated profiles based on sorts and modes, i.e. there are some input and some output positions each of a certain sort. We assume the following domains:

$$\begin{array}{lcll} \mathsf{Sigma} & \subseteq & \overline{\mathsf{Sigma}} & = \mathcal{P}(\overline{\mathsf{Profile}}) \\ \mathsf{Profile} & \subseteq & \overline{\mathsf{Profile}} & = \mathsf{Symbol} \times \mathsf{Sort}^\star \times \mathsf{Sort}^\star \\ \mathsf{Symbol} & = & & \mathsf{Name} + \mathsf{Functor} + \cdots \end{array}$$

In certain instances, proper profiles have to be restricted. The profile of a functor, for example, has a simple target in contrast to a proper Cartesian product. Signatures Σ are (finite) subsets of Profile. Again restrictions might be appropriate in certain instances. In the paper it is assumed for example that overloading is prohibited, i.e. a signature $\Sigma \in \mathsf{Sigma}$ must satisfy that $\forall p, p' \in \Sigma$:

$$\pi_{\mathsf{Symbol}}(p) = \pi_{\mathsf{Symbol}}(p') \Rightarrow p = p'.$$

The initial type system (without terms) is presented in Figure 6. $\mathcal{TYPE}_{\mathsf{Rules}}(rs)$, for example, denotes the type (i.e. the signature) of some rules rs. The type system can be refined to cope with specific constructs and properties in particular instances, e.g. terms.

[4] As far as coalesced sums $D = D_1 + \cdots + D_n$ are concerned, $\mathbf{in}_{D_i}(d)$ denotes the injection of $d \in D_i$ into D. Note that the D should be obvious from the context.

Example 3 $\mathcal{TYPE}_{\mathsf{Rules}}$(Figure 1) corresponds to the following set of profiles:

$$\begin{array}{lcl} do & : & \mathsf{C} \times \mathsf{ST} \to \mathsf{ST} \\ eval & : & \mathsf{E} \times \mathsf{ST} \to \mathsf{VAL} \\ update & : & \mathsf{ST} \times \mathsf{ID} \times \mathsf{VAL} \to \mathsf{ST} \\ apply & : & \mathsf{ST} \times \mathsf{ID} \to \mathsf{VAL} \\ \mathsf{skip} & : & \to \mathsf{C} \\ \mathsf{concat} & : & \mathsf{C} \times \mathsf{C} \to \mathsf{C} \\ \mathsf{assign} & : & \mathsf{ID} \times \mathsf{E} \to \mathsf{C} \\ \mathsf{var} & : & \mathsf{ID} \to \mathsf{E} \\ \ldots & & \end{array}$$

♢

Note that all target program fragments shown in the paper are proper fragments what implies that they are well-typed. Constructor operations according to the notation for axioms, inference rules and propositions used in Figure 1 etc. are assumed. These constructors must be regarded as partial in the sense that applications of them are defined iff the resulting fragment is a proper fragment.

As it was mentioned above *complete* target programs must probably satisfy further specific properties. The following definition provides some terms regarding a minimum requirement for the completeness of the data flow.

Definition 1 Given a rule $r \in \mathsf{Rule}$, the output (resp. input) positions of the conclusion and the input (resp. output) positions of the premises in r are called *applied* (resp. *defining*) *positions in* r. *Applied* resp. *defining variable occurrences* are variables on applied resp. defining positions. The set of applied resp. defining variable occurrences in r is denoted by $\mathcal{AO}(r)$ resp. $\mathcal{DO}(r)$.

The *data flow* in a collection of rules $rs \in \mathsf{Rules}$ is called *complete* if $\mathcal{DFC}(rs)$ holds.

$$\frac{\mathcal{AO}(\overline{r}_i) \subseteq \mathcal{DO}(\overline{r}_i) \text{ for } i = 1, \ldots, n}{\mathcal{DFC}(\{\overline{r}_1, \ldots, \overline{r}_n\})} \qquad [\mathcal{DFC}]$$

♢

Thus, completeness of the data flow in a collection of rules means that for each rule r the applied variable occurrences ($\mathcal{AO}(r)$) are contained in the defining variable occurrences ($\mathcal{DO}(r)$). The idea behind the terms applied and defining positions is that the variables with occurrences on applied positions are expected to be "computed" in terms of variables with occurrences on the defining positions. These terms are used in much the same way in, for example, extended attribute grammars [WM77]. Thereby, we may speak of *undefined* and *unused variables*, where a variable v is undefined in the rule r if $v \in \mathcal{AO}(r) \setminus \mathcal{DO}(r)$; dually for unused variables.

Example 4 All the variables in Figure 1 have a defining occurrence. Thus, $\mathcal{DFC}$(Figure 1) holds and there are no undefined variables. Actually, there are no unused variables either because all variables with a defining occurrence have an applied occurrence. ♢

$\mathcal{DFC}$ should not be required for intermediate results of program transformations but only for final results, that is to say complete programs (modules). To transform Figure 1 into Figure 3, for example, it is very suitable to insert first the additional parameter positions resulting in an intermediate result which does not satisfy $\mathcal{DFC}$. The additional premises are inserted and the data flow is established afterwards. Refer to Figure 8 for the intermediate result developed in the next section.

Instantiating and refining the framework probably other or more refined properties than just $\mathcal{DFC}$ will be relevant for complete programs, e.g. non-circularity in attribute grammars, call-correctness in logic programs [Boy96], unknowns in natural semantics, reducedness or interface conformance properties in the sense of module systems.

Another notion is needed to reason about target programs. The notion of a skeleton is similar in intent to the notion of the underlying CFG of an attribute grammar. Roughly, the skeleton of some rules is obtained by considering only the shapes of the rules, where the shape of a rule is a triple consisting of its tag, the name of the conclusion and the sequence of names of those premises which are meant to contribute to the skeleton. The "contributing" symbols are at least the defined symbols, i.e. symbols with an occurrence in a conclusion. Thus, in a sense a skeleton is degenerated collection of rules, without parameterization. Skeletons are a useful tool in meta-programs to abstract from the structure of target programs (components). Moreover, the notion facilitates pairing of rules. These applications will be clarified later on.

Definition 2 Let be $rs = \{r_1, \ldots, r_n\} \in \mathsf{Rules}$ and $ss \in \mathcal{P}(\mathsf{Name})$. The *defined symbols in* rs are denoted by $\mathcal{DS}(rs)$.

$$\mathcal{DS}(rs) = \bigcup_{i=1}^{n} \pi_{\mathsf{Name}}(\pi_{\mathsf{Conclusion}}(r_i))$$

The *skeleton of* rs *w.r.t.* ss is the set $\{sh_1, \ldots, sh_n\} \in \mathcal{P}(\mathsf{Tag} \times \mathsf{Name} \times \mathsf{Name}^\star)$ such that

$$\begin{array}{rcl} \pi_{\mathsf{Tag}}(sh_i) & = & \pi_{\mathsf{Tag}}(r_i) \\ \pi_{\mathsf{Name}}(sh_i) & = & \pi_{\mathsf{Name}}(\pi_{\mathsf{Conclusion}}(r_i)) \\ \pi_{\mathsf{Name}^\star}(sh_i) & = & \langle s_1, \ldots, s_m \rangle \end{array}$$

with $s_1, \ldots, s_m \in \mathcal{DS}(rs) \cup ss$ and there are natural numbers $q_1, \ldots, q_m$ such that $1 \leq q_1 < \cdots < q_m \leq premises$, $s_j = name_{q_j}$ for $j = 1, \ldots, m$ and $\forall k \in \{1, \ldots, premises\} \setminus \{q_1, \ldots, q_m\}$: $name_k \notin \mathcal{DS}(rs) \cup ss$, where

$$\begin{array}{rcl} premises & = & \#\pi_{\mathsf{Premise}^\star}(r_i) \\ name_x & = & \pi_{\mathsf{Name}}(\pi_x(\pi_{\mathsf{Premise}^\star}(r_i))) \end{array}$$

for $i = 1, \ldots, n$.

The skeleton of rs w.r.t. ss is denoted by

$$\mathcal{SKELETON}(rs, ss).$$

♢

The role of the names ss in the above definition is to specify more skeleton symbols, i.e. symbols contributing to the skeleton, than just the defined symbols. That is necessary if incomplete target programs, e.g. modules in the sense of components, are taken into consideration. Note that if $\mathcal{SKELETON}(rs_1, ss) = \mathcal{SKELETON}(rs_2, ss)$, then $\mathcal{SKELETON}(rs_1, ss') = \mathcal{SKELETON}(rs_2, ss')$ for $ss' \subseteq ss$. Figure 7 provides the corresponding structural definition and the restriction of it to characterize proper skeletons. For proper collections of shapes it must hold that the tags are pairwise distinct, similarly to proper collections of rules.

Example 5 Using a CFG notation $\mathcal{SKELETON}$(Figure 1, $\emptyset$) can be represented as follows:

$$\begin{array}{lll} [\mathsf{skip}] & do & : \ . \\ [\mathsf{concat}] & do & : \ do, do. \\ [\mathsf{assign}] & do & : \ eval. \\ \ldots & & \\ [\mathsf{var}] & eval & : \ . \\ \ldots & & \end{array}$$

Structure

$\overline{\mathsf{Skeleton}} = \mathcal{P}(\mathsf{Shape})$ skeletons
$\mathsf{Shape} = \mathsf{Tag} \times \mathsf{Name} \times \mathsf{Name}^\star$ shapes of rules

Proper skeletons

$$\frac{sh_i \in \mathsf{Shape} \text{ for } i = 1, \ldots, n \quad \wedge \quad \pi_{\mathsf{Tag}}(sh_i) \neq \pi_{\mathsf{Tag}}(sh_j) \text{ for } i, j = 1, \ldots, n, i \neq j}{\{sh_1, \ldots, sh_n\} \in \mathsf{Skeleton}} \quad [\mathsf{Skeleton}]$$

Figure 7: Skeletons

Note that Figure 3 has the same skeleton. In both cases all the auxiliary computations accessing the store, producing and combining outputs etc. do not contribute to the skeleton. ♢

Finally, a few remarks on equality on Rules are in place. Structural equality modulo renaming of variables is denoted by $(\_ = \_)$. Furthermore, equivalence classes in Rules are considered in order to abstract from the order of parameter positions. $rs' \in [rs]_{ss} \subset \mathsf{Rules}$ means that rs' can be transformed into some $rs'' = rs$ by only changing consistently the order of parameter positions of elements e with $\pi_{\mathsf{Name}}(e) \in ss$ all over rs'. Note that changing the order of parameter positions does not introduce technical problems as long as uniqueness for sorts on input and output positions is assumed.[5]

2.3 Functional meta-programs

To obtain a meta-programming language, it is proposed to embed the data types for meta-programming into a typed λ-calculus. Functional meta-programs are preferred because of the applicability of equational reasoning for proving properties and the suitability of higher-order functional programming to write abstract program manipulations.

Furthermore, the following specification language constructs are assumed in the resulting calculus:

- *foldl* / *foldr*, non-recursive / recursive *let*,
- the Boolean data type and the conditional $b \rightarrow e_1, e_2$,
- products ($\times$), sequences (*), sets ($\mathcal{P}$),
- maybe types ($D? = D + \{?\}$),
- an error element $\top$ for strict error propagation,
- impure constructs to generate fresh variables etc.

The error element $\top$ is regarded as an element of any type. Embedding the data types for meta-programming the application of a basic operation, e.g. for the construction of a fragment, returns $\top$ whenever the underlying operation is not defined. Evaluating a term is strict w.r.t. $\top$ with the common exception of the conditional.

To reason about (un)definedness it is assumed in the sequel that $\mathcal{DEF}(t)$ means that neither the term t is evaluated to $\top$ nor the evaluation of t diverges.

2.4 Properties of meta-programs

Certain properties of meta-programs which are useful to characterize operators and to facilitate well-founded program manipulation are considered. In the sequel the term *transformation* refers to functions on Rules. The type definition $\mathsf{Trafo} = \mathsf{Rules} \rightarrow \mathsf{Rules}$ is assumed.

The first definition concerns (α[6]-) total transformations. In general, a transformation does not need to be total because of partial fragment constructors and $\top$. However, for many operators, we can show that they are total.

Definition 3 A transformation $f \in \mathsf{Trafo}$ is α-*total* if $\forall rs \in \alpha \subseteq \mathsf{Rules}$: $\mathcal{DEF}(f(rs))$. ♢

Let us consider now a very simple preservation property, that is to say type preservation. It is often desirable to keep the output of a transformation compatible (i.e. interchangeable as far as the profiles of the symbols are concerned) with the input.

Definition 4 A transformation $f \in \mathsf{Trafo}$ is α-*type-preserving* if $\forall rs \in \alpha \subseteq \mathsf{Rules}$:

$$\mathcal{DEF}(f(rs)) \Rightarrow \mathcal{DEF}(\mathcal{TYPE}_{\mathsf{Rules}}(rs) \sqcup \mathcal{TYPE}_{\mathsf{Rules}}(f(rs))).$$

♢

Another simple preservation property is skeleton preservation. Skeleton-preservation is a valuable property in several ways. Consider for example transformations on attribute grammars, where the term skeleton corresponds (almost) to the term underlying CFG. Obviously, it is a desirable property for transformations focusing on attributes and semantic rules that they do not modify the skeleton. Moreover, the property facilitates composition based on superimposing rules with the same shape; refer to Subsection 3.3. Furthermore, skeleton preservation is necessary to be able to abstract from the structure of target programs as far as their skeletons are concerned. Our instance of weaving, for example, first accumulates a skeleton from all the components. Transformations modelling aspects may depend on the skeleton. To make sense the skeleton must be preserved during weaving.

Definition 5 A transformation $f \in \mathsf{Trafo}$ is α-*skeleton-preserving w.r.t.* $ss \in \mathcal{P}(\mathsf{Name})$ if $\forall rs \in \alpha \subseteq \mathsf{Rules}$:

$$\mathcal{DEF}(f(rs)) \Rightarrow \mathcal{SKELETON}(rs, ss) = \mathcal{SKELETON}(f(rs), ss).$$

♢

Note that if f is α-skeleton-preserving w.r.t. ss then f is also α-skeleton-preserving w.r.t. $ss' \subseteq ss$.

Let us consider a more advanced preservation property. If a given declarative program is adapted, for example, to cope with some additional computational aspects, the original computational behaviour mostly must be preserved. The semantics of the original interpreter from Figure 1, for example, is preserved by the adapted version in Figure 3 coping with I/O because a "syntactical" preservation property holds, that is to say the original interpreter can be regarded as a *projection* of the adapted interpreter where projection means that some premises and parameter positions can be removed and some occurrences of variables can be replaced by fresh variables.

Definition 6 Let be $rs, rs' \in \mathsf{Rules}$. rs is a *projection* of rs' (rs' is an *extension* of rs) if

[5] Thinking of an instance of the framework for attribute grammars, for example, uniqueness for positions of grammar symbols is a natural assumption if attributes are modelled by sorts.

[6] If some property holds only for some inputs $\alpha \subseteq \mathsf{Rules}$, the property is qualified with α.

1. $\forall \tau \in \mathcal{TYPE}_{\mathsf{Rules}}(rs) : \exists \tau' \in \mathcal{TYPE}_{\mathsf{Rules}}(rs') : \tau$ is a projection of τ', i.e.

   if $\tau' = s\ \sigma_1^{\downarrow} \times \cdots \times \sigma_{n'}^{\downarrow} \to \sigma_1^{\uparrow} \times \cdots \sigma_{m'}^{\uparrow}$

   then $\exists in_1, \ldots, in_n, out_1, \ldots, out_m$ such that

   - the in_i are pairwise distinct,
   - the out_j are pairwise distinct,
   - each $in_i \in \{1, \ldots, n'\}$,
   - each $out_j \in \{1, \ldots, m'\}$ and
   - $\tau = s\ \sigma_{in_1}^{\downarrow} \times \cdots \times \sigma_{in_n}^{\downarrow} \to \sigma_{out_1}^{\uparrow} \times \cdots \times \sigma_{out_m}^{\uparrow}$

   for $i = 1, \ldots, n$, $j = 1, \ldots, m$.

2. $|rs| = |rs'|$,

3. For every rule r in rs, there must be a rule r' in rs' as follows: $\pi_{\mathsf{Tag}}(r) = \pi_{\mathsf{Tag}}(r')$ and there is a type-consistent substitution θ such that $\theta(c) = \Pi(c')$ and $\theta(p_1) = \Pi(p'_{w_1}), \ldots, \theta(p_l) = \Pi(p'_{w_l})$, where

$$\begin{array}{rcl} c & = & \pi_{\mathsf{Conclusion}}(r) \\ c' & = & \pi_{\mathsf{Conclusion}}(r') \\ \langle p_1, \ldots, p_l \rangle & = & \pi_{\mathsf{Premise}^\star}(r) \\ \langle p'_1, \ldots, p'_{l'} \rangle & = & \pi_{\mathsf{Premise}^\star}(r') \end{array}$$

   and $w_1, \ldots, w_l$ are some natural numbers with $1 \leq w_1 < \cdots < w_l \leq l'$ and $\Pi : \mathsf{Element} \to \mathsf{Element}$ projects parameters in elements according to (1.).

$\Diamond$

Definition 7 A transformation $f \in \mathsf{Trafo}$ is *α-projection-preserving* if $\forall rs \in \alpha \subseteq \mathsf{Rules}$:

$$\mathcal{DEF}(f(rs)) \Rightarrow rs \text{ is a projection of } f(rs).$$

$\Diamond$

The term projection *preservation* makes sense here because if rs is a projection of $f(rs)$ all the projections of rs will be projections of $f(rs)$ as well. For several "sensible" instances of the framework projection-preserving transformations preserve computational behaviour because in some sense the given behaviour is extended and possibly further constrained but not adapted in any more specific sense. Kirschbaum, Sterling et al. have shown in [KSJ93], for example, that program maps—a tool similar to our projection-preserving transformations—preserve the computational behaviour of a Prolog program, if we assume that behaviour is manifested by the SLD computations of the program. It is also easy to observe that the notion is applicable to attribute grammars and natural semantics. In more general terms projections can be considered as one kind of data refinement (data transformation) [Heh93]. Obviously, not all interesting transformations are projection-preserving.

Finally, some properties concerning $\mathcal{DFC}$ are in place. Consider a transformation which preserves $\mathcal{DFC}$ in the sense that $\forall rs \in \mathsf{Rules} : \mathcal{DFC}(rs) \Rightarrow \mathcal{DFC}(f(rs))$ provided the result is defined. Such a preservation property is too weak to characterize transformations w.r.t. $\mathcal{DFC}$ because it does not apply to situations where $\mathcal{DFC}(rs)$ is not satisfied in intermediate results within a compound transformation. The following definition is useful to characterize transformations w.r.t. $\mathcal{DFC}$ in a more general sense.

Definition 8 Let be $f, f' \in \mathsf{Trafo}$, $\alpha \subseteq \mathsf{Rules}$, $\mathcal{F}$ a family $\{f_i \in \mathsf{Trafo}\}_{i \in \mathcal{I}}$ of transformations.

- The transformation f is *α-$\mathcal{DFC}$-preserving w.r.t. $\mathcal{F}$* if $\forall rs \in \alpha$: $\forall i_1, \ldots, i_n \in \mathcal{I}$:

$$\begin{array}{c} (\mathcal{DEF}(f^\star(rs)) \wedge \mathcal{DEF}(f^\star(f(rs)))) \Rightarrow \\ (\mathcal{DFC}(f^\star(rs)) \Rightarrow \mathcal{DFC}(f^\star(f(rs)))), \end{array}$$

  where $f^\star$ denotes $f_{i_n} \circ \cdots \circ f_{i_1}$.

- The *α-$\mathcal{DFC}$-preservation w.r.t. $\mathcal{F}$ for f is recovered by f'* if $\forall rs \in \alpha$: $\forall i_1, \ldots, i_n \in \mathcal{I} : \forall k : 1 \leq k \leq n$:

$$\begin{array}{c} (\mathcal{DEF}(f^\star(rs)) \wedge \mathcal{DEF}(f'^\star(f(rs)))) \Rightarrow \\ (\mathcal{DFC}(f^\star(rs)) \Rightarrow \mathcal{DFC}(f'^\star(f(rs)))), \end{array}$$

  where $f^\star$ denotes $f_{i_n} \circ \cdots \circ f_{i_1}$, whereas $f'^\star$ denotes $f_{i_n} \circ \cdots \circ f_{i_{k+1}} \circ f' \circ f_{i_k} \cdots \circ f_{i_1}$.

$\Diamond$

Note that the above weak characterization is captured by $\mathcal{DFC}$-preservation w.r.t. $\emptyset$. Recoverability of α-$\mathcal{DFC}$-preservation for f by f' means that f and f' can be composed in a sense to construct an α-$\mathcal{DFC}$-preserving transformation. This property is useful for non-$\mathcal{DFC}$-preserving transformations because it tells that f' compensates for the undefined variables introduced by f. Note that it is slightly more general to say that the α-$\mathcal{DFC}$-preservation w.r.t. $\mathcal{F}$ for f is recovered by f' than to say that $f' \circ f$ is α-$\mathcal{DFC}$-preserving. In the paper we assume that $\mathcal{F}$ in Definition 8 corresponds to the set of transformations derivable as instances from the operators introduced in the paper.

3 An operator suite

A few operators for program manipulation are introduced below. The actual selection is example-driven, i.e. the described operators suffice to describe some semantic aspects in our running interpreter example and a certain weaver. The emphasis is on the properties of the operators facilitating semantics-preserving transformation. In [Läm98] we have investigated a more expressive operator suite including the actual definition of the operators by means of meta-programs. In fact, all the operators presented in this paper can be rigorously defined by meta-programs in the framework from the previous section.

To approach to a classification, the operators are grouped to facilitate either program transformation, program analysis or program composition.

3.1 Program transformation

Four simple operators for program transformation are introduced in the sequel. To illustrate the effect of the transformations the adaptation which is necessary to transform the simple interpreter in Figure 1 into Figure 3 coping with I/O is performed in various small steps.

3.1.1 Adding positions

The operator **Add** _ : Position $\to$ Trafo with Position = Io $\times$ Name $\times$ Sort, Io = {**Input**, **Output**} is used to add parameter positions to symbols. Consider for example the transformation **Add** $\langle$**Input**, $s, \sigma\rangle$ applied to some rules $rs \in$ Rules. All the conclusions and premises in rs are transformed systematically as follows. An element $s'(p_1, \ldots, p_n) \to (p'_1, \ldots, p'_m)$ keeps unchanged if $s \neq s' \vee \pi_{\mathsf{Sort}}(p_1) = \sigma \vee \cdots \vee \pi_{\mathsf{Sort}}(p_n) = \sigma$. Otherwise it is transformed to

$s(p_1, \ldots, p_n, v) \to (p'_1, \ldots, p'_m)$, where v is a fresh variable of sort σ.

Example 6 One trivial step in the derivation of Figure 3 is to add the parameter position of sort OUT for the symbol *do*. Recall that the corresponding parameters are used to accumulate the output. The corresponding transformation is represented by **Add** ⟨**Output**, *do*, OUT⟩. ♢

An application of **Add** is always defined. The operator preserves computational behaviour and it does not change the skeleton of the input rules. The other preservation properties from Subsection 2.4 do not hold, i.e. the type is changed, since a position is added, and $\mathcal{DFC}$ is not preserved. Because of the latter we have to look for a way to compensate for the violation of $\mathcal{DFC}$ which traces back to an application of **Add**.

Proposition 1 Let be $io \in$ Io, $s \in$ Name, $\sigma \in$ Sort. The transformation **Add** $\langle io, s, \sigma \rangle$ is total, projection-preserving and skeleton-preserving w.r.t. Name. It is neither type- nor $\mathcal{DFC}$-preserving. ♢

The operator **Add** is overloaded to add several positions at once, i.e.:

$$\begin{array}{rcl} \textbf{Add}\ \langle\langle io_1, s_1, \sigma_1\rangle, \ldots, \langle io_n, s_n, \sigma_n\rangle\rangle & = & \textbf{Add}\ \langle io_n, s_n, \sigma_n\rangle \\ & \circ & \cdots \\ & \circ & \textbf{Add}\ \langle io_1, s_1, \sigma_1\rangle \end{array}$$

Moreover, an auxiliary operator **Positions** _ **For** _ **Of Sort** _ : Io × $\mathcal{P}$(Name) × Sort → Position* for the construction of positions all with the same Io and Sort component, i.e.:

$$\textbf{Positions}\ io\ \textbf{For}\ \{s_1, \ldots, s_n\}\ \textbf{Of Sort}\ \sigma = \langle\langle io, s_1, \sigma\rangle, \ldots, \langle io, s_n, \sigma\rangle\rangle$$

Example 7 We continue Example 6 by adding the auxiliary positions of sort IN, which are used in Figure 3 to propagate the remaining input. The following transformation adds these positions:

Add Positions Output For $\{do, eval\}$ **Of Sort** IN
∘ **Add Positions Input For** $\{do, eval\}$ **Of Sort** IN

The intermediate result reflecting the inserted fresh parameters is shown in Figure 8. Note the difference between the intermediate result and the final form in Figure 3. All the computations dealing with appending outputs and computing the empty output still have to be inserted. Moreover, the inserted positions of sort IN are not yet connected to encode the propagation of the input. ♢

3.1.2 Inserting constant computations

The operator **Default For** _ **By** _ : Sort × Name → Trafo facilitates the elimination of undefined variables by the insertion of "constant computations", i.e. premises with no inputs and one output. Consider the transformation **Default For** σ **By** s applied to the rule r. Let be $v_1, \ldots, v_n$ all the undefined variables of sort σ in r. The premises $s \to (v_1)$, …, $s \to (v_n)$ are inserted into r.

Example 8 The transformation **Default For** VUSES **By** *nil* is useful to add the computations for the inserted parameter of sort OUT in the rules [skip] and [assign]. The intermediate result looks as follows:

$$\frac{do(\mathsf{skip}, \mathsf{ST}, \boxed{\mathsf{IN}_1}) \to (\mathsf{ST}, \boxed{\mathsf{IN}_2}, \boxed{\mathsf{OUT}_1})}{}\ [\mathsf{skip}]$$

$$\frac{\begin{array}{l} do(\mathsf{C}_1, \mathsf{ST}, \boxed{\mathsf{IN}_2}) \to (\mathsf{ST}', \boxed{\mathsf{IN}_3}, \boxed{\mathsf{OUT}_1}) \\ \wedge\ do(\mathsf{C}_2, \mathsf{ST}', \boxed{\mathsf{IN}_4}) \to (\mathsf{ST}'', \boxed{\mathsf{IN}_5}, \boxed{\mathsf{OUT}_2}) \end{array}}{do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}, \boxed{\mathsf{IN}_1}) \to (\mathsf{ST}'', \boxed{\mathsf{IN}_6}, \boxed{\mathsf{OUT}_3})}\ [\mathsf{concat}]$$

$$\frac{\begin{array}{l} eval(\mathsf{E}, \mathsf{ST}, \boxed{\mathsf{IN}_2}) \to (\mathsf{VAL}, \boxed{\mathsf{IN}_3}) \\ \wedge\ update(\mathsf{ST}, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{ST}') \end{array}}{do(\mathsf{assign}(\mathsf{ID}, \mathsf{E}), \mathsf{ST}, \boxed{\mathsf{IN}_1}) \to (\mathsf{ST}', \boxed{\mathsf{IN}_4}, \boxed{\mathsf{OUT}_1})}\ [\mathsf{assign}]$$

$$\frac{apply(\mathsf{ST}, \mathsf{ID}) \to (\mathsf{VAL})}{eval(\mathsf{var}(\mathsf{ID}), \mathsf{ST}, \boxed{\mathsf{IN}_1}) \to (\mathsf{VAL}, \boxed{\mathsf{IN}_2})}\ [\mathsf{var}]$$

Figure 8: An intermediate step from Figure 1 to Figure 3

$$\frac{\boxed{nil \to \mathsf{OUT}}}{do(\mathsf{skip}, \mathsf{ST}, \mathsf{IN}_1) \to (\mathsf{ST}, \mathsf{IN}_2, \mathsf{OUT})}\ [\mathsf{skip}]$$

$$\frac{\begin{array}{l} eval(\mathsf{E}, \mathsf{ST}, \mathsf{IN}_2) \to (\mathsf{VAL}, \mathsf{IN}_3) \\ \wedge\ update(\mathsf{ST}, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{ST}') \\ \wedge\ \boxed{nil \to \mathsf{OUT}} \end{array}}{do(\mathsf{assign}(\mathsf{ID}, \mathsf{E}), \mathsf{ST}, \mathsf{IN}_1) \to (\mathsf{ST}', \mathsf{IN}_4, \mathsf{OUT})}\ [\mathsf{assign}]$$

Note the difference to the final form in Figure 3. The data flow concerning the parameter positions of sort IN still needs to be established. That is the topic of Example 9. ♢

Proposition 2 Let be $\sigma \in$ Sort, $s \in$ Name. The transformation **Default For** σ **By** s is α-total, type-preserving, α'-skeleton-preserving w.r.t. Name$\setminus\{s\}$, projection- and $\mathcal{DFC}$-preserving, where $\forall rs \in \alpha \subseteq \mathsf{Rules} : \mathcal{DEF}(\mathcal{TYP}\mathcal{E}_{\mathsf{Rules}}(rs) \sqcup \{s :\to \sigma\})$ and $\forall rs \in \alpha' \subseteq \mathsf{Rules} : s \notin \mathcal{DS}(rs)$. ♢

Proposition 3 Let be $io \in$ Io, $add, by \in$ Name, $\sigma \in$ Sort. The $\mathcal{DFC}$-preservation for **Add** $\langle io, add, \sigma \rangle$ is recovered by **Default For** σ **By** *by*. ♢

3.1.3 Inserting copy rules

The operator **From The Left** _ : Sort → Trafo facilitates propagation by *copying systematically defining occurrences of a certain sort to undefined variables from left to right.* In attribute grammar jargon we would say that copy rules are established. Note that an application of the operator corresponds to the insertion of a potentially unknown number of copy rules. The schema is sufficient to establish certain patterns of propagation, e.g. a bucket brigade, provided the necessary positions have been added in advance. Consider the transformation **From The Left** σ applied to the rule r. Any undefined variable v of sort σ in r is replaced by the first defining variable occurrence v' of sort σ to the left of v.

Example 9 Example 8 is continued. To establish a propagation of the input from left to right (in the sense of a proof tree) the transformation **From The Left** IN is useful. As far as the rules [skip], [assign], [var] are concerned, for example, the above transformation exactly corresponds to the missing step to arrive at the final form shown in Figure 3. The rule [concat] needs some further effort concerning the combination of the outputs returned by the premises. The corresponding adaptation is discussed in Example 10. ♢

Proposition 4 Let be $\sigma \in \mathsf{Sort}$. The transformation **From The Left** σ is total, type-preserving, skeleton-preserving w.r.t. Name, projection- and $\mathcal{DFC}$-preserving. ♢

3.1.4 Pairing unused variables

The operator **Reduce _ By _** : Sort × Name → Trafo is used to pair unused variables of a certain sort σ in a dyadic computation deriving a new defining position of sort σ. The purpose of these computations is to reduce any number > 1 of unused variables of sort σ to 1. Consider the transformation **Reduce** σ **By** s applied to the rule r. Let be $v_1, \ldots, v_n$ all the unused variables of sort σ in r (in the order of their defining occurrence in r). The computations $s(v_1, v_2) \to (v_{n+1})$, $s(v_{n+1}, v_3) \to (v_{n+2})$, …, $s(v_{n+n-2}, v_n) \to (v_{n+n-1})$, are inserted into r, where the variables $v_{n+1}, \ldots, v_{n+n-1}$ are fresh variables of sort σ. Thus, v_{n+n-1} will be the only unused variable of sort σ in the output of the transformation.

Example 10 Example 9 is continued. The defining occurrences of sort OUT in rule [concat] can be combined by the transformation **Reduce** OUT **By** *append*. Thereby, the following intermediate form of the rule is obtained:

$$\frac{\begin{array}{ll} & do(\mathsf{C}_1, \mathsf{ST}, \mathsf{IN}_0) \to (\mathsf{ST}', \mathsf{IN}_1, \mathsf{OUT}_1) \\ \wedge & do(\mathsf{C}_2, \mathsf{ST}', \mathsf{IN}_1) \to (\mathsf{ST}'', \mathsf{IN}_2, \mathsf{OUT}_2) \\ \wedge & \boxed{append(\mathsf{OUT}_1, \mathsf{OUT}_2) \to (\mathsf{OUT}_3)} \end{array}}{do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}_0, \mathsf{IN}_0) \to (\mathsf{ST}'', \mathsf{IN}_2, \mathsf{OUT}_4)} \quad [\mathsf{concat}]$$

Note that the above rule is still not yet in the final form shown in Figure 3 because the variables OUT_3 and OUT_4 must be identified. This can be modelled by the transformation **From The Left** OUT. ♢

Proposition 5 Let be $\sigma \in \mathsf{Sort}$, $s \in \mathsf{Name}$. The transformation **Reduce** σ **By** s is α-total, type-preserving, α'-skeleton-preserving w.r.t. $\mathsf{Name} \setminus \{s\}$, projection- and $\mathcal{DFC}$-preserving, where $\forall rs \in \alpha \subseteq \mathsf{Rules} : \mathcal{DEF}(\mathcal{TYPE}_{\mathsf{Rules}}(rs) \sqcup \{s : \sigma \times \sigma \to \sigma\})$ and $\forall rs \in \alpha' \subseteq \mathsf{Rules} : s \notin \mathcal{DS}(rs)$. ♢

3.2 Program analysis

Program analysis obviously seems to be useful in AOP, since weaving of component and aspect code usually has to be controlled by properties of the components. The effect of the operator **Default**, for example, depends on the set $\mathcal{AO}(r) \setminus \mathcal{DO}(r)$ for a given rule r (i.e. the undefined variables in r), where the relations $\mathcal{AO}$ and $\mathcal{DO}$ to compute applied resp. defining occurrences should be considered as analyses here. As far as the paper is concerned a further simple analysis **From _ To _ In _** : $\mathcal{P}(\mathsf{Name}) \times \mathcal{P}(\mathsf{Name}) \times \mathsf{Skeleton} \to \mathcal{P}(\mathsf{Name})$ is needed. The auxiliary operator is concerned with taking the transitive closure of symbols in a skeleton based on reachability in the context-free sense. Taking such closures is an important tool because thereby program manipulations may abstract from the underlying skeleton of a target program.

Obviously, a skeleton $sk \in \mathsf{Skeleton}$ can be regarded as a CFG. Thus, it makes sense to consider the transitive closure $\Rightarrow^+_{sk}$ of the context-free direct derivation relation w.r.t. the grammar sk. **From** *from* **To** *to* **In** sk is assumed to compute the set of all symbols $s \in \mathsf{Name}$ satisfying the property $\exists f \in from, \exists t \in to : f \Rightarrow^+_{sk} s \Rightarrow^+_{sk} t$.

Example 11 Recall Example 6 and Example 7 which are meant to add the parameter positions for propagation and accumulation of inputs and outputs respectively. The positions used in the applications of **Add** are tuned towards the interpreter fragment in Figure 1. Now consider Figure 9 with the interpreter rules for *if*- and *while*-statements. The transformations from Example 6 and Example 7 cannot be adopted for this extension because the symbol *cond* must also contribute to propagation of the input and accumulation of the output. Let us paraphrase the applications of **Add** from Example 6 and Example 7 so that they are more generic:

λrs : Rules.
Let $sk = \mathcal{SKELETON}(rs, \emptyset)$ **In**
 Let $ss_{out} =$ (**From** $\{do\}$ **To** $\{do\}$ **In** sk) $\cup \{do\}$ **In**
  Let $ss_{in} =$ (**From** $\{do\}$ **To** $\{eval\}$ **In** sk) $\cup \{do, eval\}$ **In**
  (**Add Positions Output For** ss_{out} **Of Sort** OUT
   ∘ **Add Positions Output For** ss_{in} **Of Sort** IN
   ∘ **Add Positions Input For** ss_{in} **Of Sort** IN
) (rs)

Thus, it is only stated where propagation/accumulation starts and at which points access is needed. All the auxiliary symbols are derived from the skeleton of the input. ♢

$$\frac{\begin{array}{ll} & eval(\mathsf{E}, \mathsf{ST}) \to (\mathsf{VAL}) \\ \wedge & cond(\mathsf{VAL}, \mathsf{C}_1, \mathsf{C}_2, \mathsf{ST}) \to (\mathsf{ST}') \end{array}}{do(\mathsf{if}(\mathsf{E}, \mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}) \to (\mathsf{ST}')} \quad [\mathsf{if}]$$

$$\frac{do(\mathsf{if}(\mathsf{E}, \mathsf{concat}(\mathsf{C}, \mathsf{while}(\mathsf{E}, \mathsf{C})), \mathsf{skip}), \mathsf{ST}) \to (\mathsf{ST}')}{do(\mathsf{while}(\mathsf{E}, \mathsf{C}), \mathsf{ST}) \to (\mathsf{ST}')} \quad [\mathsf{while}]$$

$$\frac{do(\mathsf{C}_1, \mathsf{ST}) \to (\mathsf{ST}')}{cond(\mathsf{boolval}(\mathsf{true}), \mathsf{C}_1, \mathsf{C}_2, \mathsf{ST}) \to (\mathsf{ST}')} \quad [\mathsf{true}]$$

$$\frac{do(\mathsf{C}_2, \mathsf{ST}) \to (\mathsf{ST}')}{cond(\mathsf{boolval}(\mathsf{false}), \mathsf{C}_1, \mathsf{C}_2, \mathsf{ST}) \to (\mathsf{ST}')} \quad [\mathsf{false}]$$

Figure 9: *if*- and *while*-statements

3.3 Program composition

Two simple program compositions are adopted. $\_ \oplus \_$ corresponds to a kind of *union* operator used in many frameworks, e.g. in [BMPT94, Bro93], whereas $\_ \otimes \_$ models some kind of *pairing* (or tupling [Chi93, HITT97]) based on superimposing skeletons of the operands. Another auxiliary operator $\succ \_ \prec$ for *squeezing* target programs in the sense of the identification of positions of the same sort and mode is discussed. Squeezing is a useful companion for pairing.

The following notation is needed. $rs|_{ts}$ selects the rules from $rs \in \mathsf{Rules}$ with tags in $ts \in \mathcal{P}(\mathsf{Tag})$. $\mathcal{TAGS}(rs)$ denotes the tags of the rules rs.

Let us consider the operators in detail. $rs_1 \oplus rs_2$ composes rs_1, $rs_2 \in \mathsf{Rules}$ in the sense of textual juxtaposition provided the process results in a proper fragment in the sense of Figure 5, i.e. the tags must be pairwise distinct and the types of the operands rs_1 and rs_2 must be compatible. The following slightly more flexible form is needed. $rs_1 \oplus_{ss} rs_2$ with $ss \in \mathcal{P}(\mathsf{Name})$ denotes $rs_1 \oplus rs_2'$, where $rs_2' \in [rs_2]_{ss}$ such that $\mathcal{DEF}(\mathcal{TYPE}_{\mathsf{Rules}}(rs_1) \sqcup \mathcal{TYPE}_{\mathsf{Rules}}(rs_2'))$ holds.

Example 12 The simple interpreter in Figure 1 is obviously not compatible with the interpreter rules for I/O constructs shown in Figure 2. However, the adapted version shown in Figure 3 is

compatible with Figure 2. The interpreter rules for *if*-and *while*-constructs shown in Figure 9 are compatible with the simple interpreter in Figure 1. Thus, the following properties hold:

- $\neg\mathcal{DEF}$(Figure 1 $\oplus$ Figure 2)
- $\mathcal{DEF}$(Figure 3 $\oplus$ Figure 2)
- $\mathcal{DEF}$(Figure 1 $\oplus$ Figure 9)

$\Diamond$

Proposition 6 Let be $rs_1, rs_2, rs_3 \in \mathsf{Rules}$ and $ss, ss' \in \mathcal{P}(\mathsf{Name})$. In (1.)—(3.) it is assumed that $\mathcal{DEF}(rs_1 \oplus_{ss} rs_2)$ holds.

1. $\mathcal{SKELETON}(rs_i, ss') = \mathcal{SKELETON}(y|_{ts_i}, ss')$
2. rs_i is a projection of $y|_{ts_i}$
   where $y = rs_1 \oplus_{ss} rs_2$ $ts_i = \mathcal{TAGS}(rs_i)$, for $i = 1, 2$
3. $\mathcal{DFC}(rs_1) \wedge \mathcal{DFC}(rs_2) \Rightarrow \mathcal{DFC}(rs_1 \oplus_{ss} rs_2)$
4. $rs_1 \oplus_{ss} (rs_2 \oplus_{ss} rs_3) = (rs_1 \oplus_{ss} rs_2) \oplus_{ss} rs_3$
5. $[rs_1 \oplus_{ss} rs_2]_{ss} = [rs_2 \oplus_{ss} rs_1]_{ss}$

$\Diamond$

Now let us consider pairing. $rs_1 \otimes rs_2$ composes rs_1, $rs_2 \in \mathsf{Rules}$ by superimposing conclusions and premises of rs_1 and rs_2. The parameters of superimposed elements are concatenated. This simple form of pairing is defined if the skeletons of the operands w.r.t. Name are equal. To avoid a confusion of variables from the different operands it is assumed that at least one operand is "refreshed" by means of a renaming substitution. The following more flexible variant is needed. $rs_1 \otimes_{ss} rs_2$ with $ss \in \mathcal{P}(\mathsf{Name}) \supseteq \mathcal{DS}(rs_1) \cup \mathcal{DS}(rs_2)$ superimposes all elements e with $\pi_{\mathsf{Name}}(e) \in ss$. All the other premises are adopted preserving their relative order in rs_1 and rs_2. Note that $rs_1 \otimes_{ss} rs_2$ is defined if:

- $\mathcal{SKELETON}(rs_1, ss) = \mathcal{SKELETON}(rs_2, ss)$ and
- $\forall p_1 \in \mathcal{TYPE}_{\mathsf{Rules}}(rs_1)$, $\forall p_2 \in \mathcal{TYPE}_{\mathsf{Rules}}(rs_2)$:
  $\pi_{\mathsf{Name}}(p_1) = \pi_{\mathsf{Name}}(p_2) \Rightarrow$
  $(\pi_{\mathsf{Name}}(p_1) \in ss \vee p_1 = p_2)$,
  i.e. the types of rs_1 and rs_2 must be compatible as far as symbols which are not superimposed are concerned.

Proposition 7 Let be $rs_1, rs_2, rs_3 \in \mathsf{Rules}$, $ss \in \mathcal{P}(\mathsf{Name})$. In (1.)—(3.) it is assumed that $\mathcal{DEF}(rs_1 \otimes_{ss} rs_2)$ holds.

1. $\mathcal{SKELETON}(rs_i, ss) = \mathcal{SKELETON}(y, ss)$
2. rs_i is a projection of y
   where $y = rs_1 \otimes_{ss} rs_2$ for $i = 1, 2$.
3. $\mathcal{DFC}(rs_1) \wedge \mathcal{DFC}(rs_2) \Rightarrow \mathcal{DFC}(rs_1 \otimes_{ss} rs_2)$
4. $rs_1 \otimes_{ss} (rs_2 \otimes_{ss} rs_3) = (rs_1 \otimes_{ss} rs_2) \otimes_{ss} rs_3$
5. $[rs_1 \otimes_{ss} rs_2]_{ss} = [rs_2 \otimes_{ss} rs_1]_{ss}$

$\Diamond$

The first statement is trivial. The second statement holds because to project $rs_1 \otimes_{ss} rs_2$ to rs_1 (resp. rs_2) it is sufficient to discard the parameter positions and premises arising from rs_2 (resp. rs_1). Thereby, pairing with one operand position fixed can be regarded as a projection-preserving transformation. The third statement holds because the set of defining and applied occurrences in $rs_1 \otimes_{ss} rs_2$ is just a kind of disjoint union of the corresponding sets in rs_1 and rs_2.

Finally, the operator $\succ\_\prec$ for squeezing is regarded, where squeezing means to identify parameter positions of elements of the same sort and mode. Squeezing is performed for all the conclusions and the premises in a rule. The parameters on positions of the same sort and mode (input versus output) are used to build equations. Consider for example the input positions in $s(p_1, \ldots, p_n) \rightarrow (\ldots)$. Suppose that there are some positions $1 \leq i_1 < \cdots < i_m \leq n$ of the same sort σ, i.e. $\pi_{\mathsf{Sort}}(p_{i_1}) = \sigma$, ..., $\pi_{\mathsf{Sort}}(p_{i_m}) = \sigma$. Then the following equations are derived: $p_{i_1} = p_{i_2}, \ldots, p_{i_1} = p_{i_m}$. In the same manner equations are derived for all elements for the two modes. Thus, for a given rule we get a set of equations on parameters. The solved form of the equations is computed[7], where the resulting substitution (i.e. a most general unifier is applied to the rule. Finally, all but the first position of the same sort and mode are eliminated. The following slightly more general variant is needed. $\succ rs \prec_{ss}$ with $ss \in \mathcal{P}(\mathsf{Name})$ squeezes only those elements e in rs with $\pi_{\mathsf{Name}}(e) \in ss$.

Proposition 8 The operator $\succ\_\prec\_$ is idempotent, skeleton-preserving w.r.t. Name, projection- and $\mathcal{DFC}$-preserving. It is not type-preserving. $\Diamond$

$$\frac{emptyset \rightarrow (\mathsf{VS})}{do(\mathsf{skip}) \rightarrow (\mathsf{VS})} \quad [\mathsf{skip}]$$

$$\frac{\begin{array}{ll} & do(\mathsf{C}_1) \rightarrow (\mathsf{VS}_1) \\ \wedge & do(\mathsf{C}_2) \rightarrow (\mathsf{VS}_2) \\ \wedge & union(\mathsf{VS}_1, \mathsf{VS}_2) \rightarrow (\mathsf{VS}) \end{array}}{do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2)) \rightarrow (\mathsf{VS})} \quad [\mathsf{concat}]$$

$$\frac{\begin{array}{ll} & eval(\mathsf{E}) \rightarrow (\mathsf{VS}_1) \\ \wedge & lhs(\mathsf{ID}) \rightarrow (\mathsf{VS}_2) \\ \wedge & union(\mathsf{VS}_1, \mathsf{VS}_2) \rightarrow (\mathsf{VS}) \end{array}}{do(\mathsf{assign}(\mathsf{ID}, \mathsf{E})) \rightarrow (\mathsf{VS})} \quad [\mathsf{assign}]$$

$$\ldots$$

$$\frac{rhs(\mathsf{ID}) \rightarrow (\mathsf{VS})}{eval(\mathsf{var}(\mathsf{ID})) \rightarrow (\mathsf{VS})} \quad [\mathsf{var}]$$

$$\ldots$$

Figure 10: Accumulating variable accesses

Example 13 Consider the rules in Figure 10 covering the same skeleton as the simple interpreter in Figure 1. These rules are concerned with a kind of reflection property, that is to say with recording variable accesses. It should be assumed that the data structure used for parameters of sort VS is a pair of two lists (or multisets), where one list is used to record LHS accesses, whereas the other list is used to record RHS accesses. The accumulation of accesses is similar to the accumulation of the output. To combine Figure 1, i.e. the simple interpreter with the new functionality in Figure 10, pairing with subsequent squeezing is appropriate. Refer to Figure 11 for the result of the following composition:

$$\succ \text{Figure 1} \otimes_{\{do, eval\}} \text{Figure 10} \prec_{\{do, eval\}}$$

Note that squeezing is necessary to identify the positions related to the traversal of abstract syntactical terms. Without squeezing the rule [skip], for example, takes the following form which is in contrast to Figure 11:

$$\frac{emptyset \rightarrow (\mathsf{VS})}{do(\boxed{\mathsf{skip}, \mathsf{skip}}, \mathsf{ST}) \rightarrow (\mathsf{ST}, \mathsf{VS})} \quad [\mathsf{skip}]$$

$\Diamond$

[7] That is meant in the sense of computing most general unifiers in logic programming; refer e.g. to [NM95].

$$\frac{emptyset \to (\mathsf{VS})}{do(\mathsf{skip},\mathsf{ST}) \to (\mathsf{ST},\mathsf{VS})} \quad [\mathsf{skip}]$$

$$\frac{\begin{array}{ll} & do(\mathsf{C}_1,\mathsf{ST}) \to (\mathsf{ST}',\mathsf{VS}_1) \\ \wedge & do(\mathsf{C}_2,\mathsf{ST}') \to (\mathsf{ST}'',\mathsf{VS}_2) \\ \wedge & union(\mathsf{VS}_1,\mathsf{VS}_2) \to (\mathsf{VS}) \end{array}}{do(\mathsf{concat}(\mathsf{C}_1,\mathsf{C}_2),\mathsf{ST}) \to (\mathsf{ST}'',\mathsf{VS})} \quad [\mathsf{concat}]$$

$$\frac{\begin{array}{ll} & eval(\mathsf{E},\mathsf{ST}) \to (\mathsf{VAL},\mathsf{VS}_1) \\ \wedge & update(\mathsf{ST},\mathsf{ID},\mathsf{VAL}) \to (\mathsf{ST}') \\ \wedge & lhs(\mathsf{ID}) \to (\mathsf{VS}_2) \\ \wedge & union(\mathsf{VS}_1,\mathsf{VS}_2) \to (\mathsf{VS}) \end{array}}{do(\mathsf{assign}(\mathsf{ID},\mathsf{E}),\mathsf{ST}) \to (\mathsf{ST}',\mathsf{VS})} \quad [\mathsf{assign}]$$

$$\frac{\begin{array}{ll} & apply(\mathsf{ST},\mathsf{ID}) \to (\mathsf{VAL}) \\ \wedge & rhs(\mathsf{ID}) \to (\mathsf{VS}) \end{array}}{eval(\mathsf{var}(\mathsf{ID}),\mathsf{ST}) \to (\mathsf{VAL},\mathsf{VS})} \quad [\mathsf{var}]$$

...

Figure 11: Squeezing and pairing Figure 1 and Figure 10

4 Aspect-oriented programming

We instantiate all the central notions of AOP. Components are open declarative programs. Aspects are implemented by program transformations. One form of weaving is represented as a certain program composition.

4.1 Properties

According to the AOP terminology properties are certain decisions a program must implement. It is hard to make that term more concrete. In terms of declarative programs covered by our framework we might think of parts of the computational behaviour, certain non-functional properties such as efficiency in some sense etc.

Example 14 For the interpreter in our running example, the following properties can be isolated:

AXIOM	starting interpretation
SEQUENCE	sequencing statements
SELECTION	*if-then-else* for statements
ITERATION	iterating statements
VARIABLE	assignments and variable expressions
DATA	basic operations
READ	reading values from the input
WRITE	writing values to the output
RECORD	recording LHS/RHS variable accesses
AST	traversal of abstract syntax
EVAL	evaluation of expressions
STORE	propagation of stores
INPUT	input propagation
OUTPUT	output accumulation
ACCESS	accumulation of variable accesses

◊

A list of properties should be a proper partitioning of design decisions. However, there is no need for the properties to be atomic in some sense. It may, for example, increase modularity to subdivide *STORE* in the above example further into initialization (like in the rule [main] in Figure 2), store transformation (like in the relation *do*) and store inspection (like in the relation *eval*).

$$\frac{do(\mathsf{C})}{prog(\mathsf{C})} \quad [\mathsf{main}]$$

Implementation of *AXIOM* meeting *AST*

$$do(\mathsf{skip}) \quad [\mathsf{skip}]$$

$$\frac{\begin{array}{ll} & do(\mathsf{C}_1) \\ \wedge & do(\mathsf{C}_2) \end{array}}{do(\mathsf{concat}(\mathsf{C}_1,\mathsf{C}_2))} \quad [\mathsf{concat}]$$

Implementation of *SEQUENCE* meeting *AST*

$$\frac{\begin{array}{ll} & eval(\mathsf{E}) \to (\mathsf{VAL}) \\ \wedge & cond(\mathsf{VAL},\mathsf{C}_1,\mathsf{C}_2) \end{array}}{do(\mathsf{if}(\mathsf{E},\mathsf{C}_1,\mathsf{C}_2))} \quad [\mathsf{if}]$$

$$\frac{do(\mathsf{C}_1)}{cond(\mathsf{boolval}(\mathsf{true}),\mathsf{C}_1,\mathsf{C}_2)} \quad [\mathsf{true}]$$

$$\frac{do(\mathsf{C}_2)}{cond(\mathsf{boolval}(\mathsf{false}),\mathsf{C}_1,\mathsf{C}_2)} \quad [\mathsf{false}]$$

Implementation of *SELECTION* meeting *AST*, *EVAL*

$$\frac{do(\mathsf{if}(\mathsf{E},\mathsf{concat}(\mathsf{C},\mathsf{while}(\mathsf{E},\mathsf{C})),\mathsf{skip}))}{do(\mathsf{while}(\mathsf{E},\mathsf{C}))} \quad [\mathsf{while}]$$

Implementation of *ITERATION* meeting *AST*

$$\frac{\begin{array}{ll} & eval(\mathsf{E},\mathsf{ST}) \to (\mathsf{VAL}) \\ \wedge & update(\mathsf{ST},\mathsf{ID},\mathsf{VAL}) \to (\mathsf{ST}') \end{array}}{do(\mathsf{assign}(\mathsf{ID},\mathsf{E}),\mathsf{ST}) \to (\mathsf{ST}')} \quad [\mathsf{assign}]$$

$$\frac{apply(\mathsf{ST},\mathsf{ID}) \to (\mathsf{VAL})}{eval(\mathsf{var}(\mathsf{ID}),\mathsf{ST}) \to (\mathsf{VAL})} \quad [\mathsf{var}]$$

Implementation of *VARIABLE* meeting *AST*, *EVAL*, *STORE*

$$\frac{\begin{array}{ll} & head(\mathsf{IN}) \to (\mathsf{VAL}) \\ \wedge & tail(\mathsf{IN}) \to (\mathsf{IN}') \end{array}}{eval(\mathsf{read},\mathsf{IN}) \to (\mathsf{VAL},\mathsf{IN}')} \quad [\mathsf{read}]$$

Implementation of *READ* meeting *AST*, *EVAL*, *INPUT*

$$\frac{\begin{array}{ll} & eval(\mathsf{E}) \to (\mathsf{VAL}) \\ \wedge & list(\mathsf{VAL}) \to (\mathsf{OUT}) \end{array}}{do(\mathsf{write}(\mathsf{E})) \to (\mathsf{OUT})} \quad [\mathsf{write}]$$

Implementation of *WRITE* meeting *AST*, *EVAL*, *OUTPUT*

$$\frac{\begin{array}{ll} & eval(\mathsf{E}) \to (\mathsf{VS}_1) \\ \wedge & lhs(\mathsf{ID}) \to (\mathsf{VS}_2) \\ \wedge & union(\mathsf{VS}_1,\mathsf{VS}_2) \to (\mathsf{VS}) \end{array}}{do(\mathsf{assign}(\mathsf{ID},\mathsf{E})) \to (\mathsf{VS})} \quad [\mathsf{assign}]$$

$$\frac{rhs(\mathsf{ID}) \to (\mathsf{VS})}{eval(\mathsf{var}(\mathsf{ID})) \to (\mathsf{VS})} \quad [\mathsf{var}]$$

Implementation of *RECORD* meeting *AST*, *ACCESS*

Figure 12: Components for an interpreter

4.2 Components

Properties which can be implemented by open declarative programs, i.e. as collections of rules in the sense of the data type Rules, are regarded as components. In our running example it is obvious that all the properties *AXIOM*, *SEQUENCE*, *SELECTION*, *ITERATION*, *VARIABLE*, *DATA*,

READ, *WRITE* and *RECORD* are concerned directly with language constructs and thus they can be modelled by the corresponding interpreter rules. That is not possible for the properties *STORE*, *INPUT*, *OUTPUT* and *ACCESS*. However, components *implementing* a certain property might additionally *meet* some other properties. The component *VARIABLE*, for example, obviously has to meet the property *STORE*. To achieve modularity and thereby reusability it is necessary to minimize the properties met by components. In our running example, the rules dealing with a certain language construct should abstract from other properties, especially *STORE*, *INPUT*, *OUTPUT* and *ACCESS* whenever possible. All components but *VARIABLE*, for example, are not concerned with propagation of the store at all. Much better modularity is achieved if such components do not meet *STORE*. Thus, changing the property *STORE* (e.g. the migration to the two-level model based on environments and stores) will not affect most components. Figure 12 shows the components for our running interpreter example.[8]

4.3 Aspects

Properties which are not components are regarded as aspects. Aspects are modelled by suitable functional metaprograms the performance of which is intended to add or to adapt the computational behaviour or to modify the structure or "style" of components. Taking a component which does not yet meet a given aspect, the corresponding transformation should be sufficient to qualify the component so that the aspect is met. Note that if all components meet a certain aspect, there will be no need to specify the aspect at all. According to the simple form of weaving to be proposed below we restrict ourselves to a certain kind of transformations modelling aspects. They are of type Skeleton → Trafo and they are expected to be skeleton-preserving w.r.t. the symbols defined by the components. Furthermore, they should preserve computational behaviour, e.g. in the sense of projection-preserving transformations. The skeleton which can be accumulated from a given set components is useful to control the performance of some transformations, e.g. if closures of symbols must be computed based on the operator **From** ... **To** ... **In** .... Thus, a function implementing an aspect observes a skeleton.

Figure 13 shows the transformations associated with the aspects for our running interpreter example. Note that the aspects *AST* and *EVAL* are not associated with a transformation because these are such basic aspect that all components should meet them anyway.

4.4 Weaving

A program composition to be regarded as a kind of weaving is outlined in the sequel. There are probably several approaches to weaving based on open declarative programs as components and transformations as aspects. The most restrictive assumption about the instance of weaving described in this paper is that the transformations corresponding to the aspects are skeleton-preserving. The program composition modelling weaving is described as the operator *weaver*:

[8] The implementation of *DATA* dealing with the evaluation of basic operations (arithmetics, comparisons, etc.) is omitted.

λsk : Skeleton.
Let $read = (\textbf{From}\ \{prog\}\ \textbf{To}\ \{eval\}\ \textbf{In}\ sk) \cup \{eval\}$ **In**
Let $write = (\textbf{From}\ \{prog\}\ \textbf{To}\ \{do\}\ \textbf{In}\ sk) \cup \{do\}$ **In**
Default For ST **By** *init*
∘ **From The Left** ST
∘ **Add** (**Positions Input For** $read \cup write$ **Of Sort** ST)
∘ **Add** (**Positions Output For** $write$ **Of Sort** ST)

Aspect *STORE*

λsk : Skeleton.
Let $ss = (\textbf{From}\ \{prog\}\ \textbf{To}\ \{eval\}\ \textbf{In}\ sk) \cup \{eval\}$ **In**
From The Left IN
∘ **Add** (**Positions Input For** $ss \cup \{prog\}$ **Of Sort** IN)
∘ **Add** (**Positions Output For** ss **Of Sort** IN)

Aspect *INPUT*

λsk : Skeleton.
Let $ss = (\textbf{From}\ \{prog\}\ \textbf{To}\ \{do\}\ \textbf{In}\ sk) \cup \{prog, do\}$ **In**
Default For OUT **By** *nil*
∘ **From The Left** OUT
∘ **Reduce** OUT **By** *append*
∘ **Add** (**Positions Output For** ss **Of Sort** OUT)

Aspect *OUTPUT*

λsk : Skeleton.
Let $ss = (\textbf{From}\ \{prog\}\ \textbf{To}\ \{eval\}\ \textbf{In}\ sk) \cup \{prog, eval\}$ **In**
Default For VS **By** *emptyset*
∘ **From The Left** VS
∘ **Reduce** VS **By** *union*
∘ **Add** (**Positions Output For** ss **Of Sort** VS)

Aspect *ACCESS*

Figure 13: Aspect code for an interpreter

$$
\begin{aligned}
\mathit{weaver} &: \mathsf{Aspect}^\star \times \mathcal{P}(\mathsf{Component}) \to \mathsf{Rules} \\
\mathsf{Aspect} &= (\mathsf{Skeleton} \to \mathsf{Trafo})? \\
\mathsf{Component} &= \underbrace{\mathcal{P}(\mathcal{N})}_{\substack{\text{aspects}\\ \text{met}}} \times \underbrace{\mathcal{P}(\mathcal{N})}_{\substack{\text{irrelevant}\\ \text{aspects}}} \times \mathsf{Rules}
\end{aligned}
$$

Thus, the operator *weaver* expects two parameters, that is to say a list of aspects and a set of components. Aspects are implemented by transformations, where a maybe-type is used here for the case that no implementation can or should be provided, e.g. *AST* and *EVAL* in the running example. A component consists of some rules, the aspects met by the component and the aspects which are irrelevant for the actual component. Natural numbers are used to index the aspects. The purpose of *weaver* is to compute a combination of all the components so that all the aspects are satisfied by the result.

Definition 9 Let be $a_1, \ldots, a_n \in \mathsf{Aspect}$, $c_1, \ldots, c_m \in \mathsf{Component}$. It is assumed that $\pi_1(c_i) \cap \pi_2(c_i) = \emptyset$ and $(\pi_1(c_i) \cup \pi_2(c_i)) \subseteq \{1, \ldots, n\}$ for $i = 1, \ldots, m$. Weaving denoted by $weaver(\langle a_1, \ldots, a_n \rangle, \{c_1, \ldots, c_m\})$ is performed as follows:

1. The defined symbols ds and the skeleton sk are accumulated from the components as follows:
$$\begin{aligned} ds &= \bigcup_{i=1}^{m} \mathcal{DS}(\pi_{\mathsf{Rules}}(c_i)) \\ sk &= \bigcup_{i=1}^{m} \mathcal{SKELETON}(\pi_{\mathsf{Rules}}(c_i), ds) \end{aligned}$$
2. For each skeleton rule sh_i from sk a corresponding rule $R_i \in \mathsf{Rule}$ covering all aspects is derived as follows:
   (a) The components are filtered to obtain all the rules $r_{i,1}, \ldots, r_{i,q_i}$ with the shape sh_i. The component which contains $r_{i,j}$ is denoted by $c_{o_{i,j}}$ for $j = 1, \ldots, q_i$.
   (b) All the above rules $r_{i,1}, \ldots, r_{i,q_i}$ are paired and squeezed:
$$\succ r_{i,1} \otimes_{ds} \cdots \otimes_{ds} r_{i,q_i} \prec_{ds}$$
   (c) The intermediate result from the previous step is transformed by f_i denoting the functional composition
$$(a_{t_{i,w_i}}(sk)) \circ \cdots \circ (a_{t_{i,1}}(sk)),$$
   where the $t_{i,1}, \ldots, t_{i,w_i}$ are the indices of aspects not covered by the components, i.e. $\{1, \ldots, n\} \setminus \bigcup_{j=1}^{q_i} (\pi_1(c_{o_{i,j}}) \cup \pi_2(c_{o_{i,j}}))$.
3. The derived rules are composed:
$$\{R_1\} \oplus_{ds} \cdots \oplus_{ds} \{R_{|sk|}\}$$

♢

Note that the above composition fails if the union in step (1.) does not define a proper skeleton or if the aspects needed in step (2. (c)) are not proper transformations, i.e. $a_{t_{i,1}} =? \vee \cdots \vee a_{t_{i,w_i}} =?$. Step (3.) might also fail because of incompatible types. Note also that the aspects should be skeleton-preserving w.r.t. ss. Otherwise it does not make sense to accumulate a skeleton from the components and to observe this skeleton in step (2. (c)).

It is straightforward to represent the function *weaver* in our functional meta-programming framework.

Example 15 An interpreter covering the constructs from the components in Figure 12 and meeting the aspects in Figure 13 is derived by weaving as follows:

$$\begin{aligned} weaver \quad (\quad & \langle ?, ?, STORE, INPUT, OUTPUT, ACCESS \rangle, \\ & \{ \langle \{1\}, \{2\}, AXIOM \rangle, \\ & \ \langle \{1\}, \{2\}, SEQUENCE \rangle, \\ & \ \langle \{1,2\}, \{\}, SELECTION \rangle, \\ & \ \langle \{1\}, \{2\}, ITERATION \rangle \\ & \ \langle \{1,2,3\}, \{\}, VARIABLE \rangle, \\ & \ \langle \{1,2,4\}, \{\}, READ \rangle, \\ & \ \langle \{1,2,5\}, \{\}, WRITE \rangle, \\ & \ \langle \{1,6\}, \{\}, RECORD \rangle \\ & \} \\ ) & \end{aligned}$$

The resulting "tangled" code is shown in Figure 14. The two question marks in the above application correspond to the aspects *AST* and *EVAL*. Note also that the aspect *EVAL* is irrelevant for the components *AXIOM*, *SEQUENCE* and *ITERATION* since they are not concerned with expression evaluation. The indices correspond otherwise to the captions in Figure 12. ♢

$$\frac{init \to (\mathsf{ST}) \quad \wedge \quad do(\mathsf{C}, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})}{prog(\mathsf{C}, \mathsf{IN}) \to (\mathsf{OUT}, \mathsf{VS})} \quad [\text{main}]$$

$$\frac{nil \to (\mathsf{OUT}) \quad \wedge \quad emptyset \to (\mathsf{VS})}{do(\mathsf{skip}, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}, \mathsf{IN}, \mathsf{OUT}, \mathsf{VS})} \quad [\text{skip}]$$

$$\frac{\begin{array}{l} do(\mathsf{C}_1, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}_1, \mathsf{VS}_1) \\ \wedge \quad do(\mathsf{C}_2, \mathsf{ST}', \mathsf{IN}') \to (\mathsf{ST}'', \mathsf{IN}'', \mathsf{OUT}_2, \mathsf{VS}_2) \\ \wedge \quad append(\mathsf{OUT}_1, \mathsf{OUT}_2) \to (\mathsf{OUT}) \\ \wedge \quad union(\mathsf{VS}_1, \mathsf{VS}_2) \to (\mathsf{VS}) \end{array}}{do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}'', \mathsf{IN}'', \mathsf{OUT}, \mathsf{VS})} \quad [\text{concat}]$$

$$\frac{\begin{array}{l} eval(\mathsf{E}, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{VAL}, \mathsf{IN}', \mathsf{VS}_1) \\ \wedge \quad cond(\mathsf{VAL}, \mathsf{C}_1, \mathsf{C}_2, \mathsf{ST}, \mathsf{IN}') \to (\mathsf{ST}', \mathsf{IN}'', \mathsf{OUT}, \mathsf{VS}_2) \\ \wedge \quad union(\mathsf{VS}_1, \mathsf{VS}_2) \to (\mathsf{VS}) \end{array}}{do(\mathsf{if}(\mathsf{E}, \mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}', \mathsf{IN}'', \mathsf{OUT}, \mathsf{VS})} \quad [\text{if}]$$

$$\frac{\begin{array}{l} do(\mathsf{if}(\mathsf{E}, \mathsf{concat}(\mathsf{C}, \mathsf{while}(\mathsf{E}, \mathsf{C})), \mathsf{skip}), \mathsf{ST}, \mathsf{IN}) \\ \to (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS}) \end{array}}{do(\mathsf{while}(\mathsf{E}, \mathsf{C}), \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})} \quad [\text{while}]$$

$$\frac{\begin{array}{l} eval(\mathsf{E}, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{VAL}, \mathsf{IN}', \mathsf{VS}_1) \\ \wedge \quad update(\mathsf{ST}, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{ST}') \\ \wedge \quad lhs(\mathsf{ID}) \to (\mathsf{VS}_2) \\ \wedge \quad union(\mathsf{VS}_1, \mathsf{VS}_2) \to (\mathsf{VS}) \end{array}}{do(\mathsf{assign}(\mathsf{ID}, \mathsf{E}), \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})} \quad [\text{assign}]$$

$$\frac{eval(\mathsf{E}, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{VAL}, \mathsf{IN}', \mathsf{VS}) \quad \wedge \quad list(\mathsf{VAL}) \to (\mathsf{OUT})}{do(\mathsf{write}(\mathsf{E}), \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}, \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})} \quad [\text{write}]$$

$$\frac{do(\mathsf{C}_1, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})}{cond(\mathsf{boolval}(\mathsf{true}), \mathsf{C}_1, \mathsf{C}_2, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})} \quad [\text{true}]$$

$$\frac{do(\mathsf{C}_2, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})}{cond(\mathsf{boolval}(\mathsf{false}), \mathsf{C}_1, \mathsf{C}_2, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})} \quad [\text{false}]$$

$$\frac{apply(\mathsf{ST}, \mathsf{ID}) \to (\mathsf{VAL}) \quad \wedge \quad rhs(\mathsf{ID}) \to (\mathsf{VS})}{eval(\mathsf{var}(\mathsf{ID}), \mathsf{ST}, \mathsf{IN}) \to (\mathsf{VAL}, \mathsf{IN}, \mathsf{VS})} \quad [\text{var}]$$

$$\frac{\begin{array}{l} head(\mathsf{IN}) \to (\mathsf{VAL}) \\ \wedge \quad tail(\mathsf{IN}) \to (\mathsf{IN}') \\ \wedge \quad emptyset \to (\mathsf{VS}) \end{array}}{eval(\mathsf{read}, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{VAL}, \mathsf{IN}', \mathsf{VS})} \quad [\text{read}]$$

Figure 14: Weaving Figure 12 and Figure 13

Finally, let us state an important property of the program composition *weaver* saying that it preserves the computational behaviour of the components because each component is a projection of some rules in the result of weaving. Furthermore, a sufficient condition for the data-flow completeness of the result of weaving is given.

Proposition 9 Let be $a_1, \ldots, a_n$, $c_1, \ldots, c_m$, ds, sk as in Definition 9. y denotes $weaver(\langle a_1, \ldots, a_n \rangle, \{c_1, \ldots, c_m\})$.

1. If $a_i \neq ?$ implies $a_i(sk)$ is projection-preserving for $i = 1, \ldots, n$, and $\mathcal{DEF}(y)$, then $y|_{\mathcal{TAGS}(c_j)}$ is a projection of c_j for $j = 1, \ldots, m$;
2. If $a_i \neq ?$ implies $a_i(sk)$ is $\mathcal{DFC}$-preserving for $i = 1, \ldots, n$ and $\mathcal{DFC}(c_j)$ for $j = 1, \ldots, m$ and $\mathcal{DEF}(y)$, then $\mathcal{DFC}(y)$,

♢

Proof It is assumed that $\mathcal{DEF}(y)$ holds.

1. It is easier to observe first that every single r from some component c_k is a projection of $y|_{\mathcal{TAGS}(\{r\})}$. Since sk

is computed as the union of all skeletons of all components, there must be an i such that r will contribute to R_i according to the steps (a)—(c) in Definition 9, i.e. r will be among the $r_{i,1}, \ldots, r_{i,q_i}$ retrieved in step (a). r is a projection of R_i computed in step (b) and (c) because the expression $f_i\ (\succ r_{i,1} \otimes_{ds} \cdots \otimes_{ds} r_{i,q_i} \prec_{ds})$ can be regarded as an application of a projection-preserving transformation to r. Refer to Proposition 7 and Proposition 8 as far as pairing and squeezing are concerned. f_i is projection-preserving because it is a functional composition of projection-preserving transformations; refer to the assumption for the $a_1, \ldots, a_n$.

It remains to show that y restricted to the tags from some component c_k can be projected to the entire component c_k at once according to the requirements (1.)—(3.) in Definition 6. The relationship for the profiles stated in (1.) exists because it carries over from projecting the rules in isolation since the signature of the entire component is just the union of the signatures of all component rules and the signature of y is just the union of the signatures of all the R_i. The 1-1 correspondence of rules stated in (2.) (concerning cardinality) and (3.) (concerning tags) holds for each component rule because there is an R_i in y with the same tag as the underlying component rules because R_i is the result of applying projection-preserving transformations to the underlying rules. The substitution σ needed in (3.) carries over from projecting the rules in isolation.

2. It suffices to show that $\mathcal{DFC}$ holds for all the single rules combined in step (3.); refer to Proposition 6. Consider again the steps (a)—(c) in Definition 9 to compute each R_i. $\mathcal{DFC}$ holds for $\succ r_{i,1} \otimes \cdots \otimes r_{i,q_i} \prec$; refer to the assumption for the $c_1, \ldots, c_m$ and to Proposition 7 and Proposition 8 as far as pairing and squeezing are concerned. f_i is $\mathcal{DFC}$-preserving because it is a functional composition of $\mathcal{DFC}$-preserving transformations; refer to the assumption for the $a_1, \ldots, a_n$.

$\Diamond$

5 Concluding remarks

First, the results of this paper are concluded. Afterwards, related work is considered in some depth. Finally, a few remarks on future work are provided.

5.1 Results

We have described an instance of aspect-oriented programming. It focuses on declarative languages as far as the components are concerned. Functional meta-programs are used to implement aspects as program transformations and to perform weaving of aspects and components in the sense of a program composition. We have developed a formal framework for functional meta-programs. It is not clear at the moment how much effort is necessary for an adaptation of the approach in order to cope with non-declarative languages.

One of the primary goals of introducing aspects and considering aspect languages is to abstract away the aspects from the components. Aspect code should be independent from components and from other aspects. This goal was addressed in several respects. First, some operators such as **Default**, **From The Left** and **Reduce** abstract from concrete symbols and concrete parameter positions. The operators only deal with occurrences of variables of certain sorts. Second, in dealing with propagation, accumulation etc. the symbols contributing to the corresponding process need not to be fixed in the aspect code but they can be derived from the skeleton of all components accumulated at weaving time. On the other hand, our aspect code still contains some details about components. Dealing with propagation, for example, the nodes which need access to the propagated data must be specified. It should be possible to abstract further in that respect by accumulating more information than just the skeleton from the components. It is a subject for further research to abstract further. Different aspects can be specified by separate program transformations. That does not mean, of course, that they are independent. Our current framework does not provide any support to detect dependencies. If there is some order on the aspects it would be easy to adapt our weaving process in such a way that this order is preserved.

Another serious problem with our instance of AOP is that there is no effective means to typecheck components w.r.t. the assertions about properties implemented and met by them (refer e.g. to the captions in Figure 12). We could try to associate a kind of type constructor with each aspect. The type constructors should be derivable from the aspect code and they could be used for typechecking components.

A very attractive property of our instance of AOP is that our weaver preserves computational behaviour of the components provided that the transformations implementing aspects preserve computational behaviour.

The framework, the operator suite for program transformation, program analysis and program composition and the operator *weaver* have been implemented in $\Lambda\Delta_\Lambda$ [HLR97, Läm98] with applications in formal language definition similar to the running interpreter example.

5.2 Related work

Fradet and Südholt suggest in their recent position paper [FS98] to describe aspects as static source-to-source program transformations. Before, aspects have been usually described and implemented in an ad hoc way. Our work can be seen as an instance and refinement of this proposal because we offer a detailed framework for functional meta-programs and an operator suite facilitating the well-founded derivation of transformations implementing aspects. The proposal in [FS98] does not focus on the declarative paradigm. Their is no correspondence to using effectively sorts and modes as in our proposal. Their approach to weaving is based on fixpoint computation using program transformations as a rewriting system.

The Demeter Research Group (Karl J. Lieberherr et al.) has developed an extension of object-oriented programming, that is to say adaptive (object-oriented) programming (AP) [Lie95, PPSL96]. The Demeter method proposes *class dictionaries* for defining the structure of objects and *propagation patterns* for implementing the behaviour of the objects. Our approach is similar to AP in that transformations are independent from the actual skeleton and a reachability notion is used to establish computational behaviour schematically in concrete target programs.

Monads and monad transformers are a popular tool in

functional programming and denotational semantics [Wad92, Mog89, Esp95] to achieve extensibility. Monads rely on higher-order functions. Thus, the approach is not applicable to the representatives of the declarative paradigm addressed by our framework for functional meta-programs. A more general restriction of the monadic style which it has in common with many other concepts is that monadic programming relies on a suitable parameterization. In the terminology of AOP we had somehow to anticipate some properties to find a suitable parameterization. Note that one set of monad parameters would be in general not sufficient. Furthermore, the kind of aspects implementable by monads are restricted by the monad laws. Meuter suggests in his recent position paper to consider monads as a theoretical foundation of AOP [Meu98]. Mosses' and Watt's Action semantics [Mos96] is another approach to extensible semantics descriptions. They do not resort to higher-order features, but they rather build support for many semantic concepts such as transient, scoped, stable and permanent information into the notation; refer to [Läm98] for a detailed comparison. Thereby, such concepts need not to be coded in low-level λ- and domain-notation.

Sterling's et al. *stepwise enhancement* [Lak89, KMS96] advocates developing logic programs from skeletons and techniques. Skeletons are (in contrast to our terminology) simple logic programs with a well-understood control flow, whereas techniques are common programming practices. Applying a technique to a program yields a so-called enhancement. Our framework for functional meta-programs and our program manipulations are effective means to develop and to reason about techniques. Kirschbaum, Sterling et al. have shown in [KSJ93] that program maps—a tool similar to our projection-preserving transformations—preserve the computational behaviour of a logic program, if we assume that behaviour is manifested by the SLD computations of the program. Note that our approach is generic and that we make vital use of modes and sorts. Skeletons in the sense of our framework are not used in stepwise enhancement. Finally, there is no concept corresponding to weaving. The composition of separate enhancements of the same skeleton is considered [Jai95, KSJ93] in similarity to pairing+squeezing required for weaving components with overlapping skeletons.

Definite clause grammars (DCGs) [PW80] and some variants and extensions (e.g. the extended DCG notation in [Roy90]) support an important (Prolog) programming technique, that is to say the accumulator [SS94]. Only the access to the accumulator is specified. Otherwise, programs had to be written in a way that accumulators are modelled by means of auxiliary arguments to be chained. Such an expressive power does not provide, however, a means to adapt programs in systemic ways. The schematic adaptations, for example, according to the operators **Default**, **Reduce** and $\succ\_\prec$ are not concerned with adding and chaining arguments.

In the attribute grammar (AG) community quite a few related concepts to improve modularity have been suggested. Watt's Partitioned AGs [Wat75], for example, support decomposition and a corresponding kind of composition of AG modules which is similar to our kind of composition based on pairing and squeezing.[9] Dueck and Cormack suggest a kind of AG templates [DC90]. An instance of a template w.r.t. a CFG is obtained by a matching algorithm. One serious problem of the approach is that it is impossible to abstract sufficiently from the underlying CFG (skeleton). The AG specification language *Lido* [KW94] offers a number of concepts to describe computations abstracting from the underlying CFG. In [Läm98] we show how to simulate and to improve the above mentioned and some other approaches including object-oriented concepts by meta-programming.

[9] Actually, the term *superposition* is used in several frameworks (not only in AGs) for a similar purpose.

5.3 Future work

The class of program transformations considered in the paper focuses on data-flow aspects. We should consider aspects and corresponding operators dealing with control flow.

A projection-preserving transformation can be considered as an effective means to refine[10] a target program. Obviously, one could try to adopt a more general notion of preservation of computational behaviour. In [Läm98] we illustrate, for example, non-projection-preserving transformations to interleave premises, to reschedule some data flow and to install a new sum domain. In that case reasoning about the preservation of computational behaviour must be based on specific arguments. Note that our weaver does not rely on projection-preserving transformations.

A rather severe assumption of our weaver is that weaving is performed rule-wise and the only non-local information used in the loop body of the weaver is the skeleton to deal with reachability. Other non-local information such as component type information could be taken into consideration.

Another approach to make components compatible to each other and to allow us to implement further properties is based on "structural" transformations. A simple example concerns the implementation of optimization aspects based on folding/unfolding strategies; refer e.g. to [PP94] in the context of logic programming. Other examples are concerned with the elimination of certain forms of recursion, CPS conversion and transmutation from big-step to small-step (semantics). The relationship between such adaptations and AOP has not been investigated so far. One specific question regarding these adaptations is what are the basic roles to derive the corresponding transformations.

The scope of the framework should be extended to further representatives of the declarative paradigm, e.g. higher-order functional programming. It should also be considered if the approach can be adopted for procedural (object-oriented) programming languages.

Another issue concerns the correctness of the operator implementations. Although we represent our transformations as functional programs, it is apparently not trivial to provide rigorous proofs for all the propositions we are interested in. We want to investigate what properties can be proved automatically, e.g. by using a theorem prover. Instead of considering just functions on Rules, the derivable properties can be possibly used to establish a more powerful type system for the framework. The properties might be useful to control the weaving process.

Acknowledgement

The paper has benefitted from discussions with Isabelle Attali and Günter Riedewald. Many thanks to the anonymous referees for their constructive review. Many thanks also to Jacques Malenfant and Kenichi Asai for several advices during the preparation of the final version of the paper.

[10] (Data) refinement is usually applied in *reasoning* about programs and specifications; refer e.g. [BR94, Heh93].

References

[AOP97] *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97, Workshop Report published in Workshop Reader of the ECOOP, Finland, Sp ringer-Verlag*, June 1997.

[AOP98] *Position papers of the Aspect-Oriented Programming Workshop at ECOOP'98, http: // wwwtrese . cs . utwente . nl / aop-ecoop98*, July 1998.

[BMPT94] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 16(3):225–237, 1994.

[Boy96] Johan Boye. *Directional Types in Logic Programming*. PhD thesis, University of Linköping, 1996.

[BR94] E. Börger and D. Rosenzweig. The WAM – Definition and Compiler Correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 2, pages 20–90. North-Holland, 1994.

[Bro93] Antonio Brogi. *Program Construction in Computational Logic*. PhD thesis, University of Pisa, 1993.

[Chi93] Wei-Ngan Chin. Towards an automated tupling strategy. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Copenhagen, Denmark, 14-16 June 1993.

[DC90] G.D. Dueck and G.V. Cormack. Modular Attribute Grammars. *The Computer Journal*, 33(2):164–172, 1990.

[Esp95] David A. Espinosa. *Semantic Lego*. PhD thesis, Graduate School of Arts and Sciences, Columbia University, 1995.

[FS98] Pascal Fradet and Mario Südholt. AOP: towards a generic framework using program transformation and analysis. In AOP98 [AOP98].

[Heh93] E.C.R. Hehner. *A Practical Theory of Programming*. Springer-Verlag, 1993.

[HITT97] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data transversals. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, The Netherlands, 9–11 June 1997.

[HLR97] Jörg Harm, Ralf Lämmel, and Günter Riedewald. The Language Development Laboratory ($\Lambda\Delta\Lambda$). In Magne Haveraaen and Olaf Owe, editors, *Selected papers from the 8th Nordic Workshop on Programming Theory, December 4–6, Oslo, Norway, Research Report 248, ISBN 82-7368-163-7*, pages 77–86, May 1997.

[Jai95] Ashish Jain. Projections of Logic Programs using Symbol Mappings. In Leon Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, June 13-16, 1995, Tokyo, Japan*. MIT Press, June 1995.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lect. Notes in Comp. Sci.*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer-Verlag.

[KMS96] M. Kirschenbaum, S. Michaylov, and L.S. Sterling. Skeletons and Techniques as a Normative Approach to Program Development in Logic-Based Languages. In *Proceedings ACSC'96, Australian Computer Science Communications, 18(1)*, pages 516–524, 1996.

[KSJ93] M. Kirschenbaum, L.S. Sterling, and A. Jain. Relating logic programs via program maps. In *Annals of Mathematics and Artifical Intelligence, 8(III-IV)*, pages 229–246, 1993.

[KW94] Uwe Kastens and W.M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica 31*, pages 601–627, 1994.

[Lak89] A. Lakhotia. *A Workbench for Developing Logic Programs by Stepwise Enhancement*. PhD thesis, Case Western Reserve University, 1989.

[Läm98] Ralf Lämmel. *Functional meta-programs towards reusability in the declarative paradigm*. PhD thesis, University of Rostock, Department of Computer Science, 1998. submitted.

[Lie95] Karl J. Lieberherr. *Adaptive Object-Oriented Software — The Demeter Method*. PWS Publishing Company, 1995.

[Meu98] Wolfgang De Meuter. Monads as a theoretical foundation for AOP. In AOP98 [AOP98].

[Mog89] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, 1989.

[Mos96] Peter D. Mosses. Theory and practice of action semantics. In *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113 of *Lect. Notes in Comp. Sci.*, pages 37–61. Springer-Verlag, 1996.

[NM95] U. Nilsson and J. Maluszynski. *Logic Programming and Prolog (2 ed)*. John Wiley, 1995.

[PP94] Alberto Pettorossi and Maurizio Proietti. Transformation of Logic Programs: Foundations and Techniques. *The Journal of Logic Programming 19, 20*, pages 261–320, 1994.

[PPSL96] Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. In Hanne Riis Nielson, editor, *6th European Symposium on Programming, Linköping, Sweden, April 1996, Proceedings of ESOP'96*, volume 1058, pages 280–295. Springer-Verlag, April 1996.

[PW80] Fernando C. N. Pereira and David H. D. Warren. Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*, 13(3):231–278, 1980.

[Roy90] Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, UC Berkeley, December 1990.

[SS94] L.S. Sterling and E.Y. Shapiro. *The Art of Prolog*. MIT Press, 1994. 2nd edition.

[Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, January 1992.

[Wat75] David A. Watt. Modular Description of Programming Languages. Technical Report A-81-734, University of California, Berkeley, 1975.

[WM77] D.A. Watt and O.L. Madsen. Extended attribute grammars. Technical Report no. 10, University of Glasgow, July 1977.