Functional meta-programs towards reusability in the declarative paradigm

Dissertation
zur
Erlangung des akademischen Grades
Doktor–Ingenieur (Dr.–Ing.)
der Fakultät für Ingenieurwissenschaften
der Universität Rostock

vorgelegt von Dipl.–Inf. Ralf Lämmel
geb. am 15. Dezember 1968 in Karl–Marx–Stadt

Rostock, 6. September 1998

Referees
Prof. Dr. rer. nat. Dr.-Ing. habil. Günter Riedewald (supervisor)
Universität Rostock
Germany

Dr. habil. Isabelle Attali (external referee)
INRIA Sophia Antipolis
France

Prof. Dr. Paul Klint (external referee)
Centrum voor Wiskunde en Informatica (CWI) & Universiteit van Amsterdam
The Netherlands

Dates

Submission	6th September 1998
Defence	14th January 1999

Abstract

Functional meta-programs on declarative target programs are proposed as a means to support reuse. We approach to this goal in the following two steps:

1. A *general framework for meta-programming* is developed. It combines

    - an applicative calculus containing suitable basic data types for declarative programs and fragments obeying well-typedness and other important properties and
    - properties of target programs and meta-programs for formal reasoning, e.g. certain preservation properties of transformations.

    We assume *modes* and *types* at the target level. They are useful to improve safety of meta-programming and to control program transformation. The framework can be *instantiated* for example for natural semantics, attribute grammars, logic programming and constructive algebraic specification. Specific features of an instance can often be modelled in the general framework by a kind of normalization. *Higher-order* functions are useful to achieve *genericity* in meta-programs.

2. An *operator suite for meta-programming* is derived, where its operators model schemata of program transformation, synthesis and composition at a high level of abstraction:

    Transformation: Certain operators facilitate *adaptation* of programs, e.g. the interpolation of computations or the establishment of new sum domains.

    Synthesis: Aspects of computational behaviour can be represented as meta-programs derived from schemata supported by the operator suite, e.g. *propagation schemata*.

    Composition: Target programs can be composed for example by means of concatenation and superimposition. Target programs can be derived from target program fragments and program transformations modelling aspects of computational behaviour by means of *lifting*.

Meta-programming occasionally surpasses other approaches to reusability based on decomposition and parameterization in the common sense. The reuse of a module, for example, depends on a suitable instantiation. In contrast, our transformational approach does not rely on such parameterization, although formal reasoning is necessary to prove correctness of reuse. We demonstrate the meta-programming approach in the context of formal language definition based on natural semantics and attribute grammars. The framework and the operator suite are compared with particular approaches to reusability in the declarative paradigm, e.g. extensible semantics definitions and paradigm shifts in attribute grammars.

Keywords

Meta-Programming, Declarative Programming, Reusability, Modularity, Extensibility, Formal Language Definition, Program Transformation, Program Synthesis, Program Composition, Lifting

CR Classification: D.1.1, D.1.6, D.2.1, D.2.13, D.3.1, D.3.3, D.3.4, F.3.1, F.3.2, F.3.3, F.4.2

Zusammenfassung

Funktionale Meta-Programme über deklarativen Objektprogrammen werden zur Unterstützung von Wiederverwendbarkeit vorgeschlagen. Dieses Ziel wird in zwei Schritten angegangen:

1. Ein allgemeines Rahmenwerk zur Meta-Programmierung wird entwickelt. Es kombiniert die folgenden Bestandteile:

    - einen applikativen Kalkül mit Datentypen für deklarative Programme und Fragmente, welche die Einhaltung von Wohlgetyptheit und anderen wichtigen Eigenschaften sicherstellen, und
    - Eigenschaften von Objekt- und Meta-Programmen zur formalen Behandlung dieser, z.B. bestimmte Erhaltungseigenschaften von Transformationen.

    Wir setzen Modi und Typen auf Objektebene voraus. Sie sind hilfreich zur Erhöhung der Sicherheit in der Meta-Programmierung und zur Steuerung von Programmtransformationen. Das Rahmenwerk kann z.B. für natürliche Semantik, attributierte Grammatiken, logische Programmierung und algebraische Spezifikation instanziiert werden. Spezielle Mittel von Instanzen können oft auch in dem allgemeinen Rahmenwerk mit Hilfe einer Normalisierung modelliert werden. Funktionen höherer Ordnung sind nutzbringend, um Generizität in Meta-Programmen zu erreichen.

2. Eine Sammlung von Operationen zur Modellierung von Schemata für die Programmtransformation, -synthese und -komposition auf einer hohen Abstraktionsstufe wird abgeleitet:

    Transformation: Bestimmte Operationen unterstützen die Anpassung von Programmen, z.B. das Einschieben von Berechnungselementen oder die Herstellung von neuen Summenbereichen.

    Synthese: Berechnungsaspekte können durch Meta-Programme, welche von grundlegenderen Schemata (z.B. Propagierungsschemata) abgeleitet wurden, repräsentiert werden.

    Komposition: Objektprogramme können z.B. im Sinne einer Verkettung oder Superimposition kombiniert werden. Objektprogramme können von Objektprogrammfragmenten und Programmtransformationen, welche Berechnungsaspekte modellieren, mittels Liften abgeleitet werden.

Meta-Programmierung übertrifft in Einzelfällen andere Ansätze zur Wiederverwendbarkeit, welche auf Dekomposition und Parameterisierung im üblichen Sinne basieren. Wiederverwendung eines Modules z.B. ist nur möglich, wenn eine brauchbare Instanziierung möglich ist. Unser transformationaler Zugang hängt nicht von einer Parameterisierung in diesem Sinne ab. Die Korrektheit der Wiederverwendung bleibt aber Beweisgegenstand. Wir führen unseren Ansatz zur Meta-Programmierung im Kontext der formalen Sprachbeschreibung auf der Basis natürlicher Semantik und attributierter Grammatiken vor. Das Rahmenwerk und die Operationen werden mit Ansätzen zur Wiederverwendbarkeit im deklarativen Paradigma, z.B. erweiterbare Semantikbeschreibungen und Erweiterungen des Formalismus für attributierte Grammatiken, verglichen.

Abbreviations

ADT	abstract data type
AG	attribute grammar
AS	abstract syntax
ASM	abstract state machine
AST	abstract syntax tree
CFG	context-free grammar
EDS	extensible denotational semantics
GSF	grammar of syntactical functions
HO	higher-order ...
LHS	left-hand side
LUB	least upper bound
MGU	most general unifier
MI	multiple inheritance
OO	object-orientation
RHS	right-hand side
SI	single inheritance
WAM	Warren abstract machine
WD	well-definedness
WF	well-formedness
w.r.t.	with respect to
WT	well-typedness

Notation

Boolean	Boolean values **True** and **False**
π_i	projections for tuples/sequences
$\mathbf{In}_i^D$	injections for sums
$\mathbf{Out}_i^D$	projections for sums
$\mathbf{Is}_i^D$	test for addend
$\otimes$	domain constructor for products
$\oplus$	domain constructor for coalesced sums
$\star$	domain constructor for sequences
$\rightarrow$	function spaces / conditional
$\circ\!\!\rightarrow$	"partial" conditional
$\mathcal{N}_0$	natural numbers (with 0)
$\mathcal{P}$	power sets
$\_?$	"maybe" construction, i.e. $D? = D \oplus \{?\}_\bot$
$\circ$	functional composition $(f \circ g)x = f(g(x))$
On	functional application in meta-programs
$\bot$	bottom element in the sense of divergence
$\top$	error element
$\langle \cdots \rangle$	for sequences and tuples
$+\!\!+$	concatenation of sequences
$\bowtie$	restricted forms of $+\!\!+$

Acknowledgement

I would like to thank all the people who have helped me in several ways during all these years. First of all, thanks to my advisor Günter Riedewald for his guidance. He has given me all the support one could ask for. He was very patient and flexible in spite of my style of working: I have spent quite a while on investigating and abandoning potential topics.

Thanks to Jan Maluszyński and Ulf Nilsson for their kind hospitality when I was guesst at IDA, University Linköping in the LOGPRO group in 1995. Much of the initial orientation for my final topic was established during these days. I am also very grateful to the Programming Research Group at University of Amsterdam, CWI Amsterdam, particularly to Paul Klint and Mark van den Brand who invited me for a stay in 1996. The stay in Amsterdam was very helpful in obtaining a cleaner proposal for a PhD thesis. I had some other invaluable oppurtunities for presenting my work in a preliminary state which I want to mention here. In October 1996 I was a guest at University of Latvia, Faculty of Physics and Mathematics. Many thanks to Karlis Cerans, Vineta Arnicane, Guntis Arnicans, Janis Bicevskis, Guntis Barzdins for their interest and the discussions. I had two helpful visits at INRIA. Many thanks to Pierre Deransart and Martin Jourdan for making possible the first stay at INRIA Rocquencourt in November 1996. Many thanks to Isabelle Attali for organizing my visit to the CROAP group at INRIA Sophia-Antipolis in September 1997. I am also very grateful for her detailed review of a draft of the thesis.

Lots of thanks to my present and past colleagues here in Rostock for providing the environment for my PhD project. Beate Baum, Anke Dittmar, Jörg Harm and Uwe Lämmel deserve my warmest gratitude, since they helped me with their comments to improve my manuscript. There are also some students who contributed in some way to my work. I would like to thank Susanne Stasch and Wolfgang Lohmann.

I had several other stimulating exchanges, verbal and electronic. I want to thank Egon Börger, Jan Bosch, Mark van den Brand, David Espinosa, Uwe Kastens, Peter Knauber, Kung-Kiu Lau, Karl J. Lieberherr, Peter D. Mosses, Didier Parigot and Eelco Visser.

Special thanks go to my mother, my sister Kathrin, my girlfriend Ellen, my friends Burkhard and Michael who always believed in me.

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

Contents

List of Figures

Chapter 1

Introduction

In Section 1.1 we explain the topic of the thesis, that is to say "Funtional meta-programs towards reusability in the declarative paradigm" including a very short indication of results and related work. Afterwards, in Section 1.2 a number of examples demonstrating our approach to meta-programming facilitating reuse is demonstrated. The examples concern the adaptation of dynamic semantics definitions for simple imperative languages in the style of natural semantics. Finally, in Section 1.3 we comment on the main results of our work which are accordingly reflected by the structure of the thesis.

1.1 The topic

According to [CI84] a *meta-program* is a program about programs. To facilitate meta-programming for programs in the language L, we need a framework M (i.e. a kind of calculus, a (meta-) programming language, or an environment), the basic data objects of which include the programs and suitable fragments of L, sometimes denoted as the *target language* or the object language of M. Meta-programs take as input programs and fragments in the target language L, perform various operations on them and possibly generate modified target language programs as outputs.

The *applications* of meta-programming include source-to-source translation and application generation in software development, program transformation (optimization, specialization, deforestation, partial evaluation/deduction, etc.; refer e.g. to [BD77, Wil90, PP94, APR97]), program synthesis (refer e.g. to [DL94, Kan91, BdM97]) and program composition (refer e.g. to [Wir74, BMPT94, Bro93, FFG91, AP91]) in formal programming methodology. In this thesis, we use meta-programs to facilitate reuse of target programs. *Reusability* is a property of a programming development method where modifications and extensions in the design of a programming problem can be easily realized at the implementation level. It is also common to use the terms extensibility and adaptability for this purpose. We propose meta-programs to compose, to extend and to adapt (target) programs. We are also interested in modelling certain parts of the software as rather meta-programs than ordinary target programs. Thereby, we can obtain a more generic description of the computational behaviour.

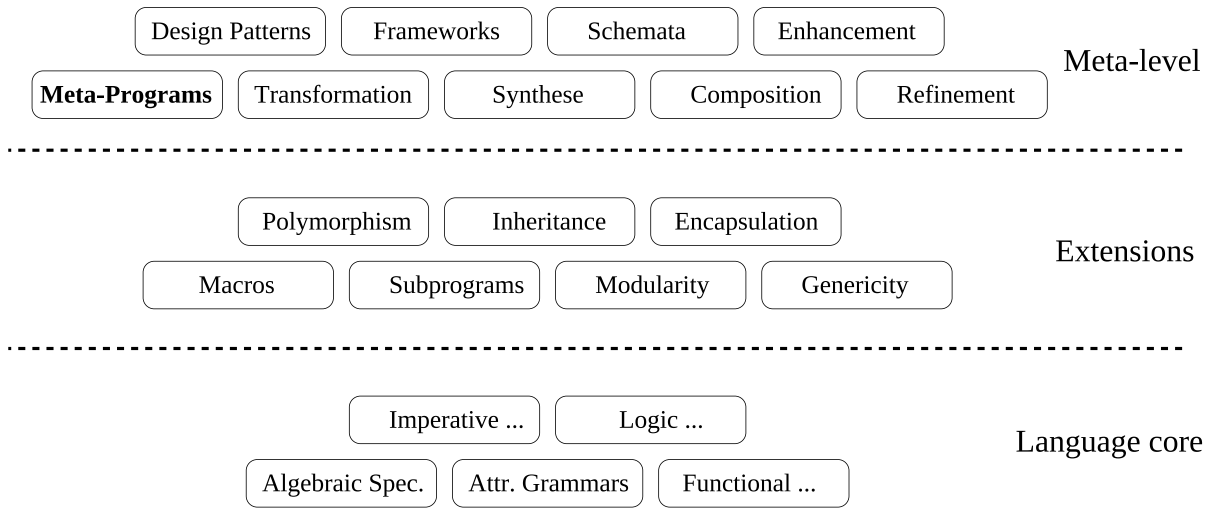


Figure 1.1: Reasoning at the meta-level

We consider meta-programming as one possible concept to improve reusability and to avoide errors and thereby to increase productivity in programming. Many other concepts are common for programming languages and specification frameworks. Some of them are possibly integrated with the underlying language, i.e. these concepts can be regarded as a kind of *extension* of the underlying language kernel, e.g. subprograms, modules, object-oriented concepts, genericity etc. in Figure 1.1. There are other concepts which are located rather at a separate level in the sense of a *meta-level*; refer again to Figure 1.1. These are concepts like refinement (e.g. in the sense of Dijkstra's method [Dij76]), design patterns [Coa95, Lar97], frameworks and program synthesis [DL94]. Meta-programming is obviously located at the meta-level as well. Note that for some concepts it depends on the point of view if they should be regarded as a language extension or as a meta-level concept. The style of adapative programming [Lie95], for example, suggests a way in which (propagation) patterns can be become an integral part of programming. Another example concerns modularity, which is not necessarily integrated with a language, but it can be the subject of meta-level reasoning like for many other approaches to program composition. Let us point out what our kind of meta-programs are meant to do. Our style of meta-programming allows to construct, deconstruct and observe target programs and type information about them. Meta-programs can represent program schemata (patterns) and program transformation schemata. Moreover, meta-programs are used to perform program compositions. We propose certain properties for formal reasoning in order to support a controlled way of composition, synthesis and transformation. Another important property of our approach is that meta-programs are executable, whereas several other meta-level concepts are rather useful for reasoning.

This thesis addresses the *declarative paradigm* as far as it concerns target languages. We think that meta-programming is a viable approach to reuse in the declarative paradigm because it is particularly suited for the review and the reconstruction of several other attempts in different specification frameworks and problem domains. In our style of meta-programming, we crucially rely on notions like many-sorted types, modes, terms, place holders, rules or equations and others as present in the declarative paradigm. To simplify the terminology, we use the term declarative (target) *language* for both, for programming languages like Prolog, Gödel and SML, and for specification formalisms like attribute grammars, natural semantics, algebraic specifications and definite clause programs. We want to achieve general results applicable for several representatives of the declarative paradigm. Thus, we try to consider an abstract form of declarative programs. Indeed, the approach can be instantiated for several existing languages or formalisms. Since *declarative* languages are the target languages in our meta-programming approach, our target programs are *declarative* programs. Note that other terms than *program* are often used in the literature for certain representatives of the declarative paradigm, e.g. the term *specification* is used for example in the context of attribute grammars and natural semantics.

The first important subject of the thesis is a *framework for meta-programming* consisting of a functional (i.e. λ-) calculus for meta-programming with built-in data types for declarative programs and fragments and some formal support to guide formal program development based on our instance of meta-programming. The data types modelling programs and fragments are defined in a way that only "correct" values can be obtained. Correctness is meant here in the sense of well-typedness and other properties. We have chosen *functional* meta-programs because higher-order functional programming provides us with a way to write generic meta-programs. Formal program development is supported by suitable properties of target programs and meta-programs, e.g. preservation properties.

The second important subject of the thesis is a *high-level operator suite for meta-programming* which is derived from the basic operators supported by the general framework. The operator suite provides us with generic schemata for composition, synthesis and transformation. The presentation of the operator suite culminates with a sophisticated composition technique called *lifting*.

Although the *achievements* of our work and the relationship to other approaches are explained in much more detail later on (see Section 1.3 and Section 5.1), we first provide a rough overview. Important properties of our approach to meta-programming are *generality*. The general framework can be instantiated for quite different representatives of the declarative paradigm. Moreover, the general framework permits us to investigate specification techniques and features as well as paradigm shifts introduced in different communities in a uniform way. We can *simulate*, for example, stepwise enhancement in logic programming [Lak89, SS94, JS94], symbol computations in attribute grammars [KW94] and remote access in attribute grammars [Kas76, Lor77, JF85, KW94, Boy96b, Boy98]. We can unbundle roles intermingled in other approaches, for example certain programming techniques in stepwise refinement can be regarded as the composition of some more elementary transformations. Our meta-programming approach crucially relies on *types* and *modes* at the target level. Thereby, safety of meta-programming is improved. Moreover, types and modes are shown to be useful to control meta-programs. The operator suite provides a solid basis for meta-programming at a high level of *abstraction*. Many approaches to reusability rely on certain forms of modularity in the sense of parameterization and decomposition, e.g. modular logic programming [HL94, Bro93, BMPT94], modules in AGs [Paa91, Bau98], specification-building operators in algebraic specification [Wir86, ST88, SST92, Wir94], or higher-order functional programs / denotational semantics parameterized by monads (or monad transformers) [Wad92, Mog89, Esp95]. We want to comment on the benefits and some limitations of reusability based on such modularity. Meta-programming-like transformations allow us to perform adaptations and extensions without depending too much on a sensible modular structure. Finally, our new composition technique *lifting* should be regarded as a major result of the thesis. At a superficial level, lifting can be compared with the monadic style in semantics [Mog89, Esp95] and functional programming [Wad92]. However, we deal with program transformation based on first-order target languages instead of monads and monad transformation in a higher-order functional setting. Our meta-programming approach is shown to be useful, for example, in the context of modular language definition based on natural semantics and attribute grammars.

1.2 Motivating examples

The purpose of this section is to present a number of examples taken from the field of formal semantics in order to demonstrate our meta-programming approach to reusability. We use specifications in the style of natural semantics as target programs. Thereby, we also provide a demonstration how our general meta-programming framework is instantiated for an actual target language (here: natural semantics). We have chosen some scenarios where given specifications must be adapted or extended. It is shown how meta-programming-like transformations can serve for that purpose. Other common approaches fail to solve such problems as below. The objective of this section is to show that the expressive power of our high-level transformation schemata and the formal degree of program manipulation provide a viable approach to reuse. A proper comparison with related work is presented in Chapter 4.

1.2.1 Preliminaries

Natural semantics [Kah87] is a popular specification formalism for static and dynamic semantics, for translations between representations and static analyses. In this thesis, natural semantics is used as one primary target language. Fragments of dynamic semantics for imperative languages in the style of natural semantics are used in numereous examples. A notational form of natural semantics similar to *RML* [Pet95, Pet94] and *Typol* [Des88, BCD^{+}88, JRG92] is used. In particular, alphanumeric identifiers are used to name propositions and a distinction between inputs and outputs in propositions is assumed.

The profiles for the relations modelling the semantics of statements and expressions for a very simple imperative language, for example, are the following:

$$\begin{aligned} \mathit{execute} &: \mathsf{STM} \times \mathsf{MEM} \rightarrow \mathsf{MEM} \\ \mathit{evaluate} &: \mathsf{EXP} \times \mathsf{MEM} \rightarrow \mathsf{VAL} \end{aligned}$$

The execution of a statement (STM) is specified by saying how the memory (MEM) is transformed, whereas the evaluation of an expression (EXP) is specified by saying how the memory is observed and what value (VAL) is returned. Refer to Figure 1.2 for some rules for these relations. Let us explain the piece of abstract syntax whose semantics is covered by the figure: The empty statement sequence is denoted by the constant (term) skip, whereas the compound statement sequence is represented by a term of the form[1] $\mathsf{concat}(\mathsf{STM}_1, \mathsf{STM}_2)$. An assignment statement with ID on the LHS and EXP on the RHS is represented by the term $\mathsf{assign}(\mathsf{ID}, \mathsf{EXP})$. Finally, a variable ID as a form of expression is represented by the term $\mathsf{var}(\mathsf{ID})$.

$$\ldots$$

$$\mathit{execute}(\mathsf{skip}, \mathsf{MEM}) \rightarrow (\mathsf{MEM}) \qquad [\mathsf{skip}]$$

$$\frac{\begin{array}{rl} & \mathit{execute}(\mathsf{STM}_1, \mathsf{MEM}_0) \rightarrow (\mathsf{MEM}_1) \\ \wedge & \mathit{execute}(\mathsf{STM}_2, \mathsf{MEM}_1) \rightarrow (\mathsf{MEM}_2) \end{array}}{\mathit{execute}(\mathsf{concat}(\mathsf{STM}_1, \mathsf{STM}_2), \mathsf{MEM}_0) \rightarrow (\mathsf{MEM}_2)} \qquad [\mathsf{concat}]$$

$$\frac{\begin{array}{rl} & \mathit{evaluate}(\mathsf{EXP}, \mathsf{MEM}_0) \rightarrow (\mathsf{VAL}) \\ \wedge & \mathit{update}(\mathsf{MEM}_0, \mathsf{ID}, \mathsf{VAL}) \rightarrow (\mathsf{MEM}_1) \end{array}}{\mathit{execute}(\mathsf{assign}(\mathsf{ID}, \mathsf{EXP}), \mathsf{MEM}_0) \rightarrow (\mathsf{MEM}_1)} \qquad [\mathsf{assign}]$$

$$\ldots$$

$$\frac{\mathit{apply}(\mathsf{MEM}, \mathsf{ID}) \rightarrow (\mathsf{VAL})}{\mathit{evaluate}(\mathsf{var}(\mathsf{ID}), \mathsf{MEM}) \rightarrow (\mathsf{VAL})} \qquad [\mathsf{var}]$$

$$\ldots$$

Figure 1.2: An interpreter fragment for a simple imperative language

Consequently, the inference rule [skip] specifies the semantics of an empty statement sequence, the rule [concat] specifies the semantics of a statement sequence and the rule [assign] specifies the execution of an assignment. Finally, the rule [var] concerns the evaluation of a variable.

1.2.2 Adapting the propagation of a data structure

As we deal with a rather simple language, it is natural that the relation *evaluate* defining the semantics of expressions only observes the memory, but it cannot modify it. Thereby, we express that side-effects do not occur during expression evaluation. It is now assumed that the language must be extended by a construct such that the evaluation of expressions may cause side-effects. The evaluation of an application of a Pascal-like function, for example, may cause side-effects due to the statement part of the function body. To reuse the interpreter program in Figure 1.2, the propagation of memories has to be adjusted. As

[1]Note the following convention for variables in target languages in this thesis: The identifier of a domain, e.g. STM, is used as the stem of variable identifiers, possibly indexed or quoted, e.g. STM_1.

far as it concerns the profiles of the relations used and defined in Figure 1.2, it is obvious that the profile of the relation *evaluate* must be extended as follows:

$$\mathit{evaluate} \quad : \quad \mathsf{EXP} \times \mathsf{MEM} \to \mathsf{VAL} \times \boxed{\mathsf{MEM}}$$

We adjust the propagation of memories in Figure 1.2 in two steps. First, a new output position of sort MEM is inserted in any proposition with the name $\mathsf{evaluate}$. This adaptation is performed by the following transformation:

$$\text{Figure 1.3} \equiv \textbf{Add } \langle \textbf{Output}, \mathsf{evaluate}, \mathsf{MEM} \rangle \textbf{ On}^{2} \text{ Figure 1.2}$$

$$\ldots$$

$$\mathit{execute}(\mathsf{skip}, \mathsf{MEM}) \to (\mathsf{MEM}) \qquad [\mathsf{skip}]$$

$$\frac{\begin{array}{rl} & \mathit{execute}(\mathsf{STM}_1, \mathsf{MEM}_0) \to (\mathsf{MEM}_1) \\ \wedge & \mathit{execute}(\mathsf{STM}_2, \mathsf{MEM}_1) \to (\mathsf{MEM}_2) \end{array}}{\mathit{execute}(\mathsf{concat}(\mathsf{STM}_1, \mathsf{STM}_2), \mathsf{MEM}_0) \to (\mathsf{MEM}_2)} \qquad [\mathsf{concat}]$$

$$\frac{\begin{array}{rl} & \mathit{evaluate}(\mathsf{EXP}, \mathsf{MEM}_0) \to (\mathsf{VAL}, \boxed{\mathsf{MEM'}}) \\ \wedge & \mathit{update}(\mathsf{MEM}_0, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{MEM}_1) \end{array}}{\mathit{execute}(\mathsf{assign}(\mathsf{ID}, \mathsf{EXP}), \mathsf{MEM}_0) \to (\mathsf{MEM}_1)} \qquad [\mathsf{assign}]$$

$$\ldots$$

$$\frac{\mathit{apply}(\mathsf{MEM}, \mathsf{ID}) \to (\mathsf{VAL})}{\mathit{evaluate}(\mathsf{var}(\mathsf{ID}), \mathsf{MEM}) \to (\mathsf{VAL}, \boxed{\mathsf{MEM'}})} \qquad [\mathsf{var}]$$

$$\ldots$$

Figure 1.3: Intermediate step from Figure 1.2 to Figure 1.4

Refer to Figure 1.3 for the intermediate result. The inserted fresh variables are boxed in Figure 1.3. Note that the rule [var] is not well-defined with regard to the data flow because of the single occurrence of MEM' on an output position of the conclusion. We do not insist on a well-defined data-flow for intermediate results. However, even intermediate results have to satisfy a number of properties including well-typedness in the sense of a many-sorted type system. In contrast to intermediate results, final results must have a well-defined data flow. This issue is implemented by making a distinction between two different types in the meta-programming type system, that is to say Rules for incomplete programs and Program for programs to be regarded as final results.

To adjust the propagation of the memory, a second step remains to be performed: The new variables must be incorporated correctly into the data flow in such a way that the result in Figure 1.4 is obtained. We do not simply speak of inserting "copy rules" to use the attribute grammar jargon, but the data flow has really to be modified and not only

[2] _ **On** _ is used for function application, i.e. f **On** $x \equiv f(x)$.

extended. Consider, for example, the rule [assign] in Figure 1.4: The memory computed by the premise with the name *evaluate* is used in the proposition with the name *update*. In contrast to that, the memory "flows" directly from the conclusion to the proposition with the name *update* in the original specification in Figure 1.2. For such problems of propagation the operator **Left To Right** is suggested to be used. It is a transformation, which, when applied to a sort σ, establishes a data flow from left to right by an identification of defining and applied occurrences[3] of sort σ in the suitable way after having refreshed all these occurrences. Thus, the propagation of memories can be adjusted by the following transformation:

$\ldots$

$$execute(\mathsf{skip}, \mathsf{MEM}) \rightarrow (\mathsf{MEM}) \qquad [\mathsf{skip}]$$

$$\frac{execute(\mathsf{STM}_1, \mathsf{MEM}_0) \rightarrow (\mathsf{MEM}_1) \quad \wedge \quad execute(\mathsf{STM}_2, \mathsf{MEM}_1) \rightarrow (\mathsf{MEM}_2)}{execute(\mathsf{concat}(\mathsf{STM}_1, \mathsf{STM}_2), \mathsf{MEM}_0) \rightarrow (\mathsf{MEM}_2)} \qquad [\mathsf{concat}]$$

$$\frac{evaluate(\mathsf{EXP}, \mathsf{MEM}_0) \rightarrow (\mathsf{VAL}, \mathsf{MEM}_1) \quad \wedge \quad update(\mathsf{MEM}_1, \mathsf{ID}, \mathsf{VAL}) \rightarrow (\mathsf{MEM}_2)}{execute(\mathsf{assign}(\mathsf{ID}, \mathsf{EXP}), \mathsf{MEM}_0) \rightarrow (\mathsf{MEM}_2)} \qquad [\mathsf{assign}]$$

$\ldots$

$$\frac{apply(\mathsf{MEM}, \mathsf{ID}) \rightarrow (\mathsf{VAL})}{evaluate(\mathsf{var}(\mathsf{ID}), \mathsf{MEM}) \rightarrow (\mathsf{VAL}, \mathsf{MEM})} \qquad [\mathsf{var}]$$

$\ldots$

Figure 1.4: An interpreter coping with side effects in expression evaluation

Figure 1.4 $\equiv$ **Left To Right** MEM **On** Figure 1.3

The operator **Left To Right** has a number of comfortable properties which make it useful for well-founded program transformation, e.g. it is total, the type of the underlying program is not changed and the skeleton of the program and well-definedness (in the sense of a correct data-flow) is preserved.

In our approach, such operators are defined in an applicative calculus supporting fragment types as basic data types. To prove the properties as mentioned above, equational reasoning (starting from the λ-expression defining an operation) can be used.

The way how transformations are formalized in the functional calculus is illustrated in Figure 1.5 which presents the calculus expression defining the operator **Left To Right**. First, the auxiliary functions *use* und *def* are declared, which are useful to replace and to refresh parameters of the given sort. The constructs[4]

[3]The input positions of the conclusion and the output positions of the premises are regarded as defining positions, whereas the complementary set corresponds to the applied positions. The variables on the corresponding positions are called occurrences.

[4]We are using mixfix notation in our functional calculus.

- **Map** $f : \tau \to \tau$ **List** $l : \tau^\star$ and
- **Fold Left** $\_ \odot \_ : \tau' \times \tau \to \tau'$ **Neutral** $e : \tau'$ **List** $\tau^\star$

are recursion/iteration schemata common in higher-order functional programming. The transformation **Left To Right** adapts each single rule by essentially refreshing and identifying variables of the given sort from left to right. There is an impure construct **New Variable** ... to generate fresh variables. A number of operators for the *deconstruction* of program fragments is used:

- **Tag Of** $\_$: $\mathsf{Rule} \to \mathsf{Tag}$,
- **Conclusion Of** $\_$: $\mathsf{Rule} \to \mathsf{Conclusion}$,
- **Premises Of** $\_$: $\mathsf{Rule} \to \mathsf{Premise}^\star$,
- **Symbol Of** $\_$: $\tau \to \mathsf{Symbol}$,
- **Parameters** $\_$ **Of** $\_$: $\mathsf{Io} \times \tau \to \mathsf{Parameter}^\star$,

where $\mathsf{Io} = \{\mathbf{Input}, \mathbf{Output}\}$, $\tau = \mathsf{Conclusion}$ or $\tau = \mathsf{Premise}$. Similarly, several *constructor* operators are exploited:

- **Rule From** $\_\ \_ \Leftarrow \_$: $\mathsf{Tag} \times \mathsf{Conclusion} \times \mathsf{Premise}^\star \to \mathsf{Rule}$,
- **Conclusion From** $\_\ \_ \longrightarrow \_$: $\mathsf{Symbol} \times \mathsf{Parameter}^\star \times \mathsf{Parameter}^\star \to \mathsf{Conclusion}$,
- **Premise From** $\_\ \_ \longrightarrow \_$: $\mathsf{Symbol} \times \mathsf{Parameter}^\star \times \mathsf{Parameter}^\star \to \mathsf{Premise}$.

This introductory example dealing with the adaptation of Figure 1.2 will be finished with some concluding remarks. The final result of the above adaptation as shown in Figure 1.4 copes with side-effects during expression evaluation. The benefit of this adaptation is that we can perform a composition of Figure 1.4 and any fragment which adheres to the same semantic model, e.g. an interpreter fragment for the evaluation of a Pascal-like function call. To "concatenate" two sets of rules is called *merging* in our work. It is facilitated by a corresponding binary operator **Merge**.

It should be pointed out that it is a big advantage to be able to specify the semantics of constructs at a level which is sufficient for the actual constructs, e.g. the evaluation of variables in Figure 1.2 does not involve side-effects. Thus, we can use the simple profile for the relation *evaluate* in Figure 1.2. The following remark should be stressed:

To be able to ignore semantic aspects is not only a matter of saving lines of code or to have a conceptionally well-structured specification, but as we cannot foresee all aspects of a specification in general—although, for interpreters of simple imperative languages we can—, it makes reuse possible per se.

1.2.3 Adding computational behaviour

Let us carry on with a slightly more complex extension. We want to add a statement of the form write(EXP) to perform an output and an expression of the form read to retrieve an input. In the semantics definition, we model the concept of outputs by accumulating output values in a corresponding sequence, whereas the remaining input is propagated to the relation *evaluate* by corresponding parameter positions. Since both, inputs and outputs, are sequences, we want to declare some straightforward polymorphic relations for

```
λ s : **Sort** .
% replace parameters of sort s by v
 Let use = λ ps : Parameter* . λ v : Variable .
  Map λ p : Parameter . Sort Of p = s → v, p List ps
 In
% refresh parameters of sort s; propagate fresh variable
  Let def = λ ps : Parameter* . λ v : Variable .
   Fold Left
    λ ⟨ps, v⟩ : Parameter* × Variable . λ p : Parameter .
     Sort Of p = s →
      Let new = New Variable Of Sort s In ⟨ps ++ ⟨new⟩, new⟩,
      ⟨ps ++ ⟨p⟩, v⟩
   Neutral ⟨⟨ ⟩, v⟩ List ps
  In
% transform each single rule
   λ rs : Rules . Map λ r : Rule .
    Let concl = Conclusion Of r In
     Let fresh = New Variable Of Sort s In
      Let ⟨conclI, v1⟩ = def On Parameters Input Of concl On fresh In
       Let ⟨prems, v2⟩ =
% iterate the premises
        Fold Left
         λ ⟨pres, vnext⟩ : Premise* × Variable . λ pre : Premise .
          Let preI = use On Parameters Input Of pre On vnext In
           Let ⟨preO, vnew⟩ = def On Parameters Output Of pre On vnext In
            ⟨pres ++ ⟨Premise From Symbol Of pre preI → preO⟩, vnew⟩
        Neutral ⟨⟨ ⟩, v1⟩ List Premises Of r
       In
        Let conclO = use On Parameters Output Of concl On v2 In
         Rule From Tag Of r Conclusion From Symbol Of concl conclI → conclO ⇐ prems
    List rs.
```

Figure 1.5: **Left To Right** _ : Sort → (Rules → Rules)

list processing required below in some fragments:

empty	:	$\rightarrow List(\tau)$	% to denote the empty list
singleton	:	$\tau \rightarrow List(\tau)$	% to transform an element into a list
append	:	$List(\tau) \times List(\tau) \rightarrow List(\tau)$	% ordinary concatenation of lists
head	:	$List(\tau) \rightarrow \tau$	% to obtain the head of a list
tail	:	$List(\tau) \rightarrow List(\tau)$	% to obtain the tail of a list
affix	:	$\tau \times List(\tau) \rightarrow List(\tau)$	% to extend a list

To achieve a kind of modular semantics, where the semantics of particular constructs is specified without too much assumptions about other design decisions which are not so relevant for the constructs, we try to specify the semantics of the new constructs in some economical way; refer to Figure 1.6 and Figure 1.7.

The reason why we call the semantics fragments "minimal" is that we abstract from

$$\frac{\begin{array}{ll} & \mathit{evaluate}(\mathsf{EXP}) \rightarrow (\mathsf{VAL}) \\ \wedge & \mathit{singleton}(\mathsf{VAL}) \rightarrow (\mathsf{OUT}) \end{array}}{\mathit{execute}(\mathsf{write}(\mathsf{EXP})) \rightarrow (\mathsf{OUT})} \qquad [\mathsf{write}]$$

Figure 1.6: A "minimal" semantics of a write-statement

$$\frac{\begin{array}{ll} & \mathit{head}(\mathsf{IN}_0) \rightarrow (\mathsf{VAL}) \\ \wedge & \mathit{tail}(\mathsf{IN}_0) \rightarrow (\mathsf{IN}_1) \end{array}}{\mathit{evaluate}(\mathsf{read}, \mathsf{IN}_0) \rightarrow (\mathsf{VAL}, \mathsf{IN}_1)} \qquad [\mathsf{read}]$$

Figure 1.7: A "minimal" semantics of a read-expression

certain details like the propagation of memories. Aiming at reusable fragments it is meaningful to abstract from the propagation of memories because there are several options for memory propagation as we have seen above in Subsection 1.2.2. It can also be the case that a two-level model consisting of an envrionment and a store must be used instead of "flat" memories. Such assumptions should not be fixed in fragments which do not rely on one or another decision. The semantics of the write-statement and the read-expression is minimal also in the sense that we ignore inputs in the rule [write] and we also ignore outputs in the rule [read]. Finally, the rule [write] resembles the basic case that statements produce outputs, but expressions do not. That is in contrast to the scenario, where expression evaluation can cause all kinds of side effects.

To reuse the given semantics fragments from Figure 1.6 and Figure 1.7 in the context of our interpreter in Figure 1.4, the corresponding rules must be qualified accordingly; refer to Figure 1.8 for the result.

$$\frac{\begin{array}{ll} & \mathit{evaluate}(\mathsf{EXP}, \boxed{\mathsf{MEM}_0}, \boxed{\mathsf{IN}_0}) \rightarrow (\mathsf{VAL}, \boxed{\mathsf{MEM}_1}, \boxed{\mathsf{IN}_1}, \boxed{\mathsf{OUT}_0}) \\ \wedge & \boxed{\mathit{affix}}(\mathsf{VAL}, \boxed{\mathsf{OUT}_0}) \rightarrow (\mathsf{OUT}_1) \end{array}}{\mathit{execute}(\mathsf{write}(\mathsf{EXP}), \boxed{\mathsf{MEM}_0}, \boxed{\mathsf{IN}_0}) \rightarrow (\boxed{\mathsf{MEM}_1}, \boxed{\mathsf{IN}_1}, \mathsf{OUT}_1)} \qquad [\mathsf{write}]$$

$$\frac{\begin{array}{ll} & \mathit{head}(\mathsf{IN}_0) \rightarrow (\mathsf{VAL}) \\ \wedge & \mathit{tail}(\mathsf{IN}_0) \rightarrow (\mathsf{IN}_1) \\ \wedge & \boxed{\mathit{empty} \rightarrow \mathsf{OUT}} \end{array}}{\mathit{evaluate}(\mathsf{read}, \boxed{\mathsf{MEM}}, \mathsf{IN}_0) \rightarrow (\mathsf{VAL}, \boxed{\mathsf{MEM}}, \mathsf{IN}_1, \boxed{\mathsf{OUT}})} \qquad [\mathsf{read}]$$

Figure 1.8: Adapted semantics of write and read

The adaptation can be described in terms of some transformations:

- Positions of sort MEM are added and the data flow for memories is established; refer to the inserted positions of sort MEM. The operators **Add** and **Left To Right**, which we have introduced in Subsection 1.2.2, are sufficient for that purpose.

- In the same way the rule [write] is transformed to contribute to the propagation of the remaining input; refer to the inserted positions of sort IN.
- Since we assume that the evaluation of expressions may cause side-effects, the relation *evaluate* also must return an output; refer to the inserted positions of sort OUT.
- For the rule [write], we must make sure that the output from the premise with the name *evaluate* is incorporated into the output produced by the statement. Thus, we perform the following transformations:
  1. **Rename Symbol** *singleton* **To** *affix*
  2. **Add** $\langle$**Input**, *affix*, OUT$\rangle$
  3. **Copy** $\langle$**Output**, *evaluate*, OUT$\rangle$ **To** $\langle$**Input**, *affix*, OUT$\rangle$

  The operator **Rename** is a straightforward operator: It serves for renaming names of propositions. The actual application from above is semantics-preserving. The operator **Copy** ... **To** ... unifies the parameters on two positions. In attribute grammars jargon, we would say that a semantic copy rule is inserted.
- Finally, it must be specified that the evaluation of a read-expression produces no output. Therefore, the position of sort OUT, which we have inserted into the rule [read], is associated with a new proposition serving as a kind of initialization. The following transformation performs the necessary adapation:

  Default For OUT **By** *empty*

  In general, the operator **Default** adds for every variable of a given sort (i.e. OUT in the example) without an associated defining occurences a new premise with the given name (i.e. *empty* in the example) and the variable as the only output position.

To conclude on the above transformations we should point out that transformations allow us to *instantiate* a specification for certain uses. The transformations we have shown so far concern the addition of parameter positions, the adaptation of the data flow, renaming and the insertion of premises.

Before we can merge our interpreter and the new (instantiated) constructs, the interpreter from Figure 1.4 must be adapted to cope with the accumulation of output and the propagation of the remaining input; refer to Figure 1.9 for the corresponding variant of the interpreter which is "compatible" to Figure 1.8 with the I/O constructs. Essentially, we add parameterization and computational behaviour in a way that the input is propagated by positions of sort IN in the sense of a bucket brigade [DC90, Ada91] or accumulator, whereas the output is "purely synthesized" based on positions of sort OUT. Let us comment on the transformations modelling the necessary adaptation:

- Positions of sort IN and OUT are inserted as visualized in Figure 1.9. The operator **Add** serves for that purpose as before. The proper data flow for the positions of sort IN is achieved by another application of **Left To Right**. The positions of sort OUT require more effort as discussed below.

$\ldots$

$$\frac{\boxed{\mathit{empty} \to \mathsf{OUT}}}{\mathit{execute}(\mathsf{skip}, \mathsf{MEM}, \boxed{\mathsf{IN}}) \to (\mathsf{MEM}, \boxed{\mathsf{IN}}, \boxed{\mathsf{OUT}})} \qquad [\mathsf{skip}]$$

$$\frac{\begin{array}{ll} & \mathit{execute}(\mathsf{STM}_1, \mathsf{MEM}_0, \boxed{\mathsf{IN}_0}) \to (\mathsf{MEM}_1, \boxed{\mathsf{IN}_1}, \boxed{\mathsf{OUT}_1}) \\ \wedge & \mathit{execute}(\mathsf{STM}_2, \mathsf{MEM}_1, \boxed{\mathsf{IN}_1}) \to (\mathsf{MEM}_2, \boxed{\mathsf{IN}_2}, \boxed{\mathsf{OUT}_2}) \\ \wedge & \boxed{\mathit{append}(\mathsf{OUT}_1, \mathsf{OUT}_2) \to (\mathsf{OUT})} \end{array}}{\mathit{execute}(\mathsf{concat}(\mathsf{STM}_1, \mathsf{STM}_2), \mathsf{MEM}_0, \boxed{\mathsf{IN}_0}) \to (\mathsf{MEM}_2, \boxed{\mathsf{IN}_2}, \boxed{\mathsf{OUT}})} \qquad [\mathsf{concat}]$$

$$\frac{\begin{array}{ll} & \mathit{evaluate}(\mathsf{EXP}, \mathsf{MEM}_0, \boxed{\mathsf{IN}_0}) \to (\mathsf{VAL}, \mathsf{MEM}_1, \boxed{\mathsf{IN}_1}, \boxed{\mathsf{OUT}}) \\ \wedge & \mathit{update}(\mathsf{MEM}_1, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{MEM}_2) \end{array}}{\mathit{execute}(\mathsf{assign}(\mathsf{ID}, \mathsf{EXP}), \mathsf{MEM}_0, \boxed{\mathsf{IN}_0}) \to (\mathsf{MEM}_2, \boxed{\mathsf{IN}_1}, \boxed{\mathsf{OUT}})} \qquad [\mathsf{assign}]$$

$\ldots$

$$\frac{\begin{array}{ll} & \mathit{apply}(\mathsf{MEM}, \mathsf{ID}) \to (\mathsf{VAL}) \\ \wedge & \boxed{\mathit{empty} \to \mathsf{OUT}} \end{array}}{\mathit{evaluate}(\mathsf{var}(\mathsf{ID}), \mathsf{MEM}, \boxed{\mathsf{IN}}) \to (\mathsf{VAL}, \mathsf{MEM}, \boxed{\mathsf{IN}}, \boxed{\mathsf{OUT}})} \qquad [\mathsf{var}]$$

$\ldots$

Figure 1.9: An interpreter coping with inputs and outputs

- In general, their can be several premises returning some output; refer e.g. to the rule [concat]. In such cases all the positions must be "composed" to a single output. Let us insert premises of the form $\mathit{append}(\mathsf{OUT}_1, \mathsf{OUT}_2) \to (\mathsf{OUT})$ to perform such a composition. There is another operator facilitating this kind of pairwise combination which is used in the following instance:

  Reduce OUT **By** *append*

- The variables on the inserted applied positions of sort OUT are not defined yet. If there is a defining position of sort OUT (note that there is at most one due to the previous step) it can be copied. Otherwise the empty output should be returned. Copying is achieved by a weaker variant of the operator **Left To Right**, that is to say **From The Left**. If there is an undefined occurrence of a variable of sort OUT, it will be unified with a defined occurrence from the left—if there is any. **From The Left** is weaker in the sense that occurrences of the corresponding sort should not be refreshed as in the case of the operator **Left To Right**. To return the empty output a corresponding premise has to be inserted based on the operator **Default** in similarity to the rule [read] in Figure 1.8.

The actual composition of the adopted interpreter fragment for basic language constructs and the I/O constructs is expressed as follows in our calculus:

Merge Figure 1.9 **And** Figure 1.8

The result is simply the concatenation of the rules from the referred figures. The composition can be described in some more detail by making explicit how the above operands were achieved. t_1 is assumed to denote the adaptations which were necessary for the interpreter in Figure 1.4 to cope with I/O, similarly, for t_2 and t_3 with respect to the I/O constructs from Figure 1.6 and Figure 1.7. These transformations have been described above. Thus, the above composition has the following more detailed description:

Merge (t_1 **On** Figure 1.4) **And** (**Merge** (t_2 **On** Figure 1.6) **And** (t_3 **On** Figure 1.7))

This example demonstrates how we can combine fragments of specifications at different layers (or levels) of the computational model (or the semantic model). Transformations like the t_i above are used to relate the levels or—to put it differently—to qualify fragments at one level for another level.

1.2.4 Further scenarios

Adapting and extending semantics specifications, there are a lot more applications for transformations. Often there are only small adaptations necessary for successful reuse, the extension of the memory propagation in Subsection 1.2.2 for example is such a rather simple adaptation. Nevertheless, without meta-programming reuse is not feasible even for such simple scenarios. Let us sketch some further scenarios:

Adding control-flow constructs When adding constructs like jumps the style of the semantics needs to be adjusted. We can use a rather transitional semantics in that case. One can define a transformation schema to adopt certain parts of a big step semantics for use in small step semantics (and vice versa).

From non-recursive abstractions to recursive abstractions It is simple to write and to understand the semantics for abstractions like procedures or functions as long as we do not cope with recursion. The variants supporting recursion are slightly more complex. Again, we can use a transformation to adapt the semantics of non-recursive abstractions to cope with recursion. It can be based on a coding technique which is common in functional programming and formal semantics, that is to say *finite unfolding*.

More general forms of LHSs in assignments A very simple language like the one in Figure 1.2 regards variable identifiers as the only form of LHSs for assignments. If we add arrays, records, pointers, or functions, assignments become more involved. There is a clean way to perform the corresponding generalization in the static semantics specification by means of transformations. Essentially, we *fold* the rule modelling the simple semantics of assignment in a way that the premises corresponding to the LHS moves to a new relation modelling the semantics of LHSs. Further forms are supported by adding rules for the new relation.

1.3 Results and structure

The results of the thesis accordingly represented by the structure of the thesis are concluded in the following subsections. Note that many technical details and some background material is contained in the Appendix chapters.

1.3.1 A general framework for meta-programming

We propose a general framework for modular and functional meta-programs on declarative target programs. It is important to notice that target programs, fragments and type information can be manipulated in meta-programs because suitable data types (Figure 1.10) are embedded into the meta-language. Let us describe and justify the actual data types and the entire framework in some detail.

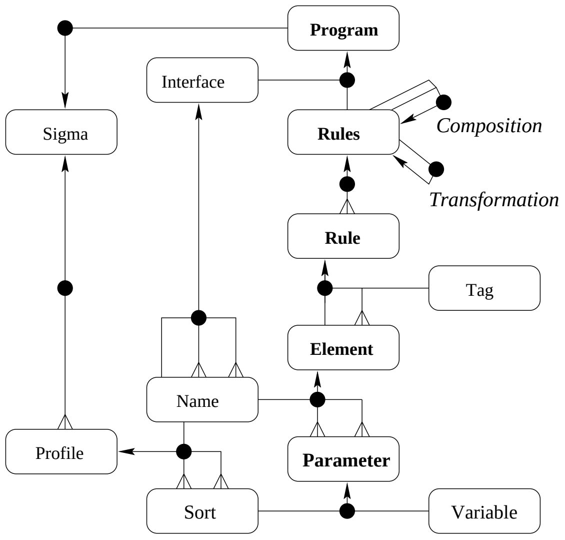


Figure 1.10: Data types modelling target programs

The data types for meta-programmming are meant to capture basic language constructs of several declarative programming languages and specification frameworks such as natural semantics (e.g. *RML*), attribute grammars (e.g. GSFs) and constructive algebraic specification. Thus, there are data types for fragments like rules, conclusions, premises etc. The data types should also take into consideration properties which are important for declarative programs, e.g. well-typedness. Actually, we can regard the data types of our general framework as an abstraction from concrete languages. Analysing concrete examples $\{F_i\}_{i \in I}$ (languages, specification frameworks) we do not only get a kind of abstract language kernel L_i (constructs + properties), but we are also interested in a characterization of M_i denoting the manipulations (paradigm shifts, features, extensions and meta-level concepts in the sense of Figure 1.1) supported by the frameworks F_i. Such an abstraction can be visualized as in Figure 1.11.

$$\begin{tikzcd}
 & F_i \arrow[dl] \arrow[dr] \arrow[d, phantom, "\text{abstraction}"] & \\
L_i & & M_i
\end{tikzcd}$$

Figure 1.11: Analysing concrete specification frameworks

Certain representatives of the declarative paradigm are not captured by our actual data types for meta-programming, e.g. denotational semantics, higher-order functional programming, "non-constructive" algebraic specification are beyond our scope. However, some reuse concepts offered by the representatives might be relevant in our discussion. In an abstract sense, our data types represent an idealized language L derived from some selection $\{L_j\}_{j \in J \subset I}$. Meta-programs on the data types for L provide the lowest level of manipulations M we propose for L in our general framework. We assume that L can be instantiated for the L_j. Technically, meta-programming is implemented by embedding the data types for meta-programming into a calculus for modular applicative programs. What we are finally looking for are obviously manipulations M' at a higher level of abstraction. The manipulations M_j of our examples F_j (e.g. modularity, remote access, schemata) are mostly at a higher level. We will try to represent such manipulations as meta-programs. Formal reasoning about target programs and meta-programs is supported by suitable properties in our general framework. Preservation properties, totality, fragment selection properties, for example, provide important ingredients for reasoning about meta-programming.

The general framework for meta-programming (i.e. the data types from Figure 1.10, the resulting calculus and properties for formal reasoning) is presented in full detail in Chapter 2. There is also shown how the language L of the general framework can be instantiated for several concrete languages L_j such as natural semantics, attribute grammars, logic programming and algebraic specification; refer to Section 2.4. Note that such an instantiation does not explain yet how to reconstruct the manipulations M_j associated with the specification frameworks F_j, e.g. modularity, remote access and schemata.

1.3.2 The operator suite for meta-programming

The general framework supports the development of modular functional meta-programs and formal reasoning about them. In our meta-programs, target programs can be constructed and deconstructed and the type information of a target program can be observed. These are the basic manipulations M in our general framework based on our idealized language L. Preservation properties and others mentioned above permits us formal reasoning about meta-programs. To approach to a higher level of abstraction in meta-programming, we develop an operator suite in the sense of a library of meta-programs. The concepts M' embodied by the operators of the suite will serve for the review and the reconstruction of existing concepts M_j.

Figure 1.12 presents the structure of the operator suite. We start from some set of

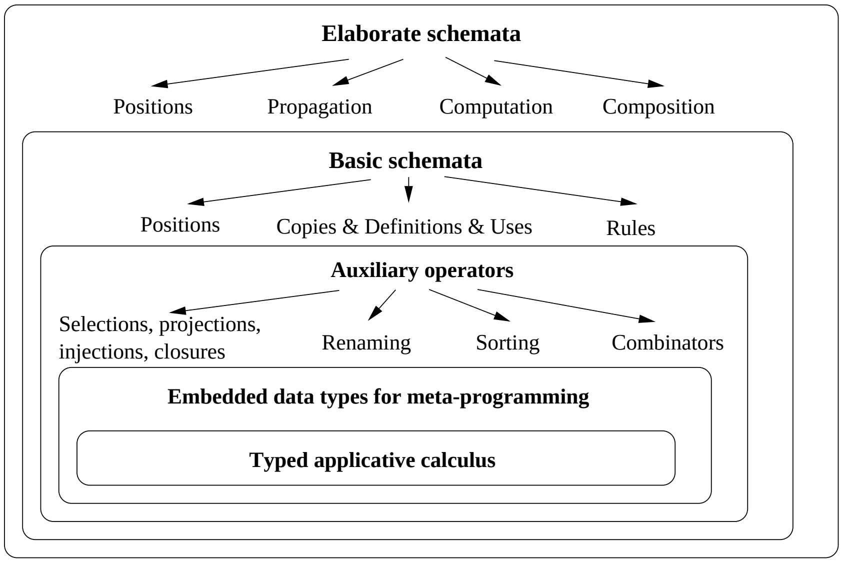

Elaborate schemata
Positions
Propagation
Computation
Composition
Basic schemata
Positions
Copies & Definitions & Uses
Rules
Auxiliary operators
Selections, projections, injections, closures
Renaming
Sorting
Combinators
Embedded data types for meta-programming
Typed applicative calculus

Figure 1.12: Layers of the operator suite

auxiliary operators. Then basic schemata capturing basic concepts of the synthesis, the adaptation and the composition of declarative target programs are defined, e.g. to add positions or simple computational elements. In the next layer, several more elaborate schemata are proposed, e.g. complex schemata to add computational behaviour or to propagate data structures. All these schemata are meant to support program composition, synthesis and transformation. The operator suite is based on a slight refinement of the idealized language L of the general framework. The actual refinement permit us to apply the suite for natural semantics and GSFs (Grammars of Syntactical Functions: parameterized context-free grammars with relational formulae on the parameters associated with the rules; a kind of attribute grammars closely related to logic programming). The suite is developed in full detail in Chapter 3. We will present several schemata which are not described elsewhere in the literature in the context of stepwise enhancement [Lak89, SS94, JS94], rule models [Hed92, KLMM93], modular attribute grammars [DC90], paradigm shifts in *Lido* [KW94], etc.

$$\begin{tikzcd}
L \arrow[d] \arrow[dr, phantom, "\text{instantiation}"] & M' \arrow[d] \\
L'_j & M'_j
\end{tikzcd}$$

Figure 1.13: Mapping the general framework to concrete specification frameworks

Based on a meta-programming-like point of view and on the actual operator suite we can reconstruct existing concepts, which have been proposed in the declarative paradigm to support reuse. Remote access, for examples, can be "compiled" by propagation schemata. In an abstract sense, we try to understand existing frameworks as instantiations $\{\langle L'_j, M'_j\rangle\}_{j\in J}$ of our enriched general meta-programming framework $\langle L, M'\rangle$; refer to Figure 1.13. We do not say that exactly $\langle L_j, M_j\rangle$ is reconstructed because the language and the manipulations need possibly to be extended, restricted or adapted. We only make a few reconstructions explicit in Chapter 4 describing related work, but it is often commented on the concepts modelled by one or another operator.

1.3.3 Composition by lifting

We propose a new composition technique *lifting* based on meta-programming; refer to Section 3.5. The starting point is to subdivide a programming problem into computational aspects. Program fragments can be located at some "level" of the complete computational model. Transformations called transformers can be used to add computational aspects. *Lifting* means to derive a complete program with the complete computational behaviour from a program skeleton (a context-free grammar or a signature), fragments at certain levels and transformers; refer to Figure 1.14.

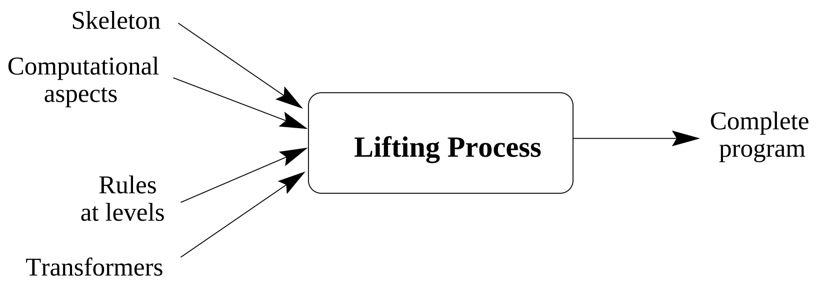


Figure 1.14: Program composition by lifting

In some sense, our notion of lifting is similar to lifting (or stratification) in modular denotational semantics based on monads and monad transformers and to the monadic style of functional programming; refer e.g. to [Esp95]. Monads are dedicated to higher-order functional specification frameworks such as higher-order functional programming and denotational semantics. In our approach, we can achieve a similar degree of extensibility by meta-programs serving as transformers, even for first-order target languages. The monadic style depends on a suitable parameterization. We indicate that our transformational approach does not require such preconditions or those inherent to other forms of modularity in Chapter 4 describing related work. It is also interesting to notice that our transformational approach to composition is similar in intent to aspect-oriented programming [KLM+97] in the sense that we also try to specify aspects of computational behaviour separatly to avoid "tangled" code.

1.3.4 Modular language definition

As the motivating examples have made clear, we concentrate on applications of the meta-programming approach in formal language definition. Therefore, attribute grammars and operational semantics (e.g. natural semantics) are used as target languages in all examples. It is demonstrated that meta-programming facilitates the specification in the following problem domains:

- semantic aspects of programming languages,
- adaptations for common syntactical and semantic variants of the described constructs and concepts and
- composition of language fragments.

We claim that the fine granularity of computational aspects we can deal with, the possibilities for composition and adaptation cannot be achieved by other prominent techniques promising reusability, particularly in AG design.

A language construction set which crucially relies on the meta-programming approach will be presented in a separate paper [LRBS]. Attribute grammars and operational semantics are used as the underlying formalisms. The construction set covers imperative languages and simple modular and object-oriented languages.

Chapter 2

The general framework

In this Chapter, we propose *functional meta-programs on declarative target programs.* To be applicable to a certain range of representatives of the declarative paradigm, the data types for meta-programming and the corresponding notions such as well-typedness are defined in a general way. Section 2.1 provides an overview of the data types, the resulting applicative calculus and the properties of target programs and meta-programs. In Section 2.2 the data types concerning programs and fragments of them are considered in more detail. Afterwards, crucial notions for dealing with declarative programs are introduced in Section 2.3, e.g. well-typedness and selection criteria for fragments. In Section 2.4, we refine the data types for meta-programming and the notions for declarative programs to cope with actual target languages. The data types for meta-programming are embedded into an applicative calculus in Section 2.5. Section 2.6 defines a number of properties of meta-programs, e.g. preservation properties.

2.1 Overview

Data type	**Explanation**	$\mathcal{WF}/\mathcal{WT}$	**Structure**
Program	complete programs	$\surd$	Rules $\otimes$ Interface
Rules	*compatible* sequences of rules	$\surd$	Rule*
Rule	rules	$\surd$	Tag $\otimes$ Conclusion $\otimes$ Premise*
Conclusion	conclusions for rules	$\surd$	Element
Premise	premises for rules	$\surd$	Element $\oplus \cdots$
Element	parameterized symbols	$\surd$	Name $\otimes$ Parameter* $\otimes$ Parameter*
Parameter	parameters	$\surd$	(Variable $\oplus \cdots$) $\otimes$ Sort
Variable	countable set of variables		
Name	symbols for elements		Id
Interface	import / export / optional axiom	$\surd$	$\mathcal{P}$(Name) $\otimes$ $\mathcal{P}$(Name) $\otimes$ Name?
Symbol	universe of symbols		Name $\oplus \cdots$
Tag	tags of rules		Id
Id	countable set of identifiers		

Figure 2.1: Data types for meta-programming (part 1/2)

The starting point for our approach to meta-programming is a collection of suitable data types for meta-programming. There are data types for programs and fragments of them; refer to Figure 2.1 for an overview. Moreover, there are auxiliary data types dealing with type information (in the sense of the target language), fragment selection, substitution and unification; refer to Figure 2.2 for an overview.

Data type	**Explanation**	$\mathcal{WF}/\mathcal{WT}$	**Structure**
Sort	sorts of positions etc.		
Profile	profiles of symbols		$\mathsf{Symbol} \oplus \mathsf{Sort}^\star \oplus \mathsf{Sort}^\star$
Sigma	signatures	$\surd$	$\mathcal{P}(\mathsf{Profile})$
Substitution	substitutions	$\surd$	$\mathcal{P}(\mathsf{Variable} \otimes \mathsf{Parameter})$
Association	associations of symbols and sorts		$\mathsf{Symbol} \otimes \mathsf{Sort}$
Io	selector fragments		$\{\mathbf{Input}, \mathbf{Output}\}$
Position	addresses of parameter positions		$\mathsf{Io} \otimes \mathsf{Symbol} \otimes \mathsf{Sort}$

Figure 2.2: Data types for meta-programming (part 2/2)

Consequently, basic operations for constructing and deconstructing fragments need to be defined. A particular property of our approach to meta-programming is that the data types for programs and fragments are restricted to elements obeying well-formedness and well-typedness; refer to the column $\mathcal{WF}/\mathcal{WT}$ in Figure 2.1 and Figure 2.2. *Well-formedness* captures simple context-sensitive properties of programs and fragments, such as that the tags of the rules are pairwise distinct. *Well-typedness* is meant in the sense of a many-sorted type system, like for many-sorted algebraic specifications or programming languages like Gödel. For complete programs we additionally require *well-definedness* capturing properties particularly important for *complete* programs such as a well-defined data flow (e.g. L-attribution or strong non-circularity for AGs, or call correction or i/o-correctness for logic programs) and a kind of reducedness property (e.g. in the context-free sense). Thereby, it is guaranteed that only proper fragments and specifications are derived in any step of a meta-program, but this also means that some applications of construction operators are not defined.

The central data types are Rule and Rules, i.e. single rules and compatible sequences of them. Transformations in the narrow sense are functions on Rules, i.e. they are of the following type:

$$\mathsf{Trafo} = \mathsf{Rules} \rightarrow \mathsf{Rules}$$

The above data types are embedded into an applicative calculus in order to support functional meta-programs; refer to Figure 2.3 for the functional programming-like constructs we assume. We prefer *functional* meta-programs instead of other possible options, because:

- the meta-programs should be *declarative* (versus imperative) to allow us simple formal reasoning about meta-programs and
- *higher-order* functions (versus first-order specification formalisms) are quite useful as meta-programs, since they provide, for example, a straightforward means to model

generic transformations.

Form	**Explanation**
_ **On** _	functional application
Let $x = e$ **In** e'	non-recursive *Let*
Letrec $x = e$ **In** e'	recursive *Let*
$\_ \circ \_$	composition $f \circ g$ **On** $x = f$ **On** $(g$ **On** $x)$
$\_ = \_$	equality (on non-functional domains)
$\_ \rightarrow \_, \_$	conditional
$\top$	an error value being an element of any type
$\_ \circ\!\!\rightarrow \_$	partial conditional; $b \circ\!\!\rightarrow e$ means $b \ \rightarrow \ e, \top$
$\langle \ldots \rangle$	construction of sequences / tuples
Head Of _, **Tail Of** _, **Nil?** _	deconstruction of sequences
$\_ +\!\!+ \_$	concatenation of sequences
Fold /**Map**	recursion schemata
_?	the "maybe" type constructor; $D? = D \oplus \{?\}_\perp$

Figure 2.3: λ-calculus-like constructs

Program transformations are very expressive. This is a statement we will comment on all through the thesis. In particular, we can perform adaptations which are not supported by common forms of modularity. On the other hand, we have to aspire to a discipline of meta-programming supporting a kind of *controlled* reuse. It is obvious that one can describe almost every adaptation by sufficiently powerful transformations, but controlled reuse means that the application of meta-programming operators is driven by their properties and by the semantics of the target programs serving as operands and results. Consequently, we discuss properties of meta-programs in some depth in Section 2.6, e.g. the well-known general semantics preservation. If we go on to define high-level operators for program transformation, synthesis and composition in the next chapter, this analysis will be helpful in characterizing particular operators. The notions for target programs from Section 2.3 such as well-typedness are crucial for our approach to *safe* meta-programming as well.

2.2 Fragments

The domains Rule, Rules, Program, Conclusion, Premise, Element, Parameter and some other auxiliary data types, e.g. Tag and Name, are defined below. For some domains we will distinguish a structural definition and the actual domain obtained as a restriction of the structural definition. The name of the domain corresponding to the structural definition is the overlined name of the actual domain, e.g. $\overline{\mathsf{Rule}}$ denotes the name of the domain of the structural definition for Rule. If no restriction is necessary, the overlined domain and the actual domain are not distinguished, e.g. for Tag. The restricted domains are usually defined by inference rules in similarity to type systems.

2.2.1 Rule

The data type Rule is an abstraction from constructs being relevant in many specification frameworks, e.g. rules in natural semantics, or definite clauses in logic programming. A rule consists of a tag, a conclusion and some premises. Tags are useful to refer to a particular rule within sequences of rules. The following structural definition is suggested:

$$\begin{aligned}
\overline{\mathsf{Rule}} &= \mathsf{Tag} \otimes \overline{\mathsf{Conclusion}} \otimes \overline{\mathsf{Premise}}^{\star} \\
\overline{\mathsf{Conclusion}} &= \overline{\mathsf{Element}} \\
\overline{\mathsf{Premise}} &= \overline{\mathsf{Element}} \oplus \cdots
\end{aligned}$$

Elements are parameterized symbols (names) in correspondence to propositions in natural semantics, literals in logic programs and grammar symbols with associated attributes in attribute grammars. Elements are considered in more detail in Subsection 2.2.4. The symbol of the conclusion is said to be *defined* by the rule. The symbols of the premises are said to be *used* by the rule. At this point, we ignore that there can be other forms of premises than elements. The domain Premise can be extended later on to cope with other forms, e.g.:

- semantic rules of an attribute grammar,
- matching constructs ($\_ = \_$) in a logic program and
- negative equations of an algebraic specification.

We should comment on the actual decision to consider a *sequence* of premises rather than an (unordered) set of premises. If the body of a definite clause is read simply as a conjunction, there will be no necessity for maintaining the order among the literals in the conjunction. For many instances, however, the actual order of the premises is significant or at least pragmatically useful:

- The order is significant for the RHS of a context-free grammar rule. Consequently, sequences of elements need to be considered for attribute grammars.
- Several specification formalisms or well-modedness conditions require certain data flow properties, e.g. *RML* [Pet95, Pet94], *ASF(+SDF)* [Kli93], call-correctness and i/o-correctness in logic programming [Boy96a], which depend on an actual order of premises.
- The order of the premises is often understood as a description of control flow and data flow, Sterling's et al. notion of a skeleton in [KMS96], for example, captures (logic) programs with a well-understood computational behaviour (including control-flow). Indeed, we will assume that the relative order of the premises possibly contributes to control-flow and/or data-flow.
- Finally, premises can be addressed by their position in the sequence.

Note also that the order of premises is possibly relevant for certain evaluation strategies or for incremental evaluation; refer e.g. to [AC90, ACG92] in the context of natural semantics.

Proper values of Rule are characterized as follows:

$$\frac{\begin{array}{ll} & \overline{r} \in \overline{\mathsf{Rule}} \text{ is a triple of the form } \langle t, \overline{e}_0, \langle \overline{e}_1, \ldots, \overline{e}_n \rangle \rangle, n \geq 0 \\ \wedge & \overline{e}_i \in \mathsf{Element} \text{ for } i = 0, \ldots, n \\ \wedge & \mathcal{WF}_{\mathsf{Rule}}(\overline{r}) \\ \wedge & \exists\, \Sigma : \mathcal{WT}_{\mathsf{Rule}}(\Sigma, \overline{r}) \end{array}}{\overline{r} \in \mathsf{Rule}} \quad [\mathsf{Rule}]$$

Well-formedness ($\mathcal{WF}$) is discussed in Subsection 2.3.1, whereas well-typedness ($\mathcal{WT}$) of target programs and fragments is the subject of Subsection 2.3.2. Signatures Σ are expected to associate names (symbols) with many-sorted directional types (or profiles).

The construction of a rule r from a tag t, a conclusion e and a sequence of premises $e^\star$ is expressed in the following mixfix notation:

$$\textbf{Rule From } t\ e\ \Leftarrow\ e^\star$$

It should be pointed out that operations for the construction of fragments are usually partial.

It is simple to define basic operations for the deconstruction of rules:

$$\begin{array}{rcl} \textbf{Tag Of \_} & : & \mathsf{Rule} \to \mathsf{Tag} \\ \textbf{Conclusion Of \_} & : & \mathsf{Rule} \to \mathsf{Conclusion} \\ \textbf{Premises Of \_} & : & \mathsf{Rule} \to \mathsf{Premise}^\star \end{array}$$

$$\textbf{Tag Of Rule From } t\ e\ \Leftarrow\ e^\star \to t \qquad [\textbf{Tag Of}]$$

$$\textbf{Conclusion Of Rule From } t\ e\ \Leftarrow\ e^\star \to e \qquad [\textbf{Conclusion Of}]$$

$$\textbf{Premises Of Rule From } t\ e\ \Leftarrow\ e^\star \to e^\star \qquad [\textbf{Premises Of}]$$

2.2.2 Rules

The data type Rules models certain restricted sequences of rules. Thus, obviously the following structural definition can be assumed:

$$\overline{\mathsf{Rules}} = \overline{\mathsf{Rule}}^\star$$

Elements of Rules are restricted in the sense that they have to satisfy well-formedness and the types of the single rules must be compatible to each other. Consequently, proper values of Rules are characterized as follows:

$$\frac{\begin{array}{ll} & \overline{rs} \in \overline{\mathsf{Rules}} \text{ is a sequence of the form } \langle \overline{r}_1, \overline{r}_2, \ldots, \overline{r}_n \rangle, n \geq 0 \\ \wedge & \overline{r}_i \in \mathsf{Rule} \text{ for } i = 1, \ldots, n \\ \wedge & \mathcal{WF}_{\mathsf{Rules}}(\overline{rs}) \\ \wedge & \exists\, \Sigma : (\mathcal{WT}_{\mathsf{Rule}}(\Sigma, \overline{r}_i) \text{ for } i = 1, \ldots, n) \end{array}}{\overline{rs} \in \mathsf{Rules}} \quad [\mathsf{Rules}]$$

The construction of an element $rs \in \mathsf{Rules}$ is expressed in the following way:

$$\textbf{Rules From \_} \quad : \quad \mathsf{Rule}^\star \to \mathsf{Rules}$$

Concatenation on Rules is denoted by $\_ \bowtie \_ : \mathsf{Rules} \times \mathsf{Rules} \to \mathsf{Rules}$.

$$\frac{\overline{rs}_1 +\!\!+ \overline{rs}_2 \in \mathsf{Rules}}{\textbf{Rules From } \overline{rs}_1 \bowtie \textbf{Rules From } \overline{rs}_2 \to \overline{rs}_1 +\!\!+ \overline{rs}_2} \qquad [\_ \bowtie \_ \text{ for } \mathsf{Rules}]$$

Here $\_ +\!\!+ \_$ denotes ordinary concatenation of sequences.

It is straightforward to define basic operations **Nil?**, **Head Of** and **Tail Of** for the deconstruction of elements of Rules in similarity to the iteration on sequences; refer to Section B.1.

2.2.3 Program

A program $p \in \mathsf{Program}$ is an even more restricted sequence of rules as constrained in the data type Rules together with a kind of interface. The structural definition of programs is the following:

$$\begin{array}{rcl} \overline{\mathsf{Program}} & = & \overline{\mathsf{Rules}} \otimes \overline{\mathsf{Interface}} \\ \overline{\mathsf{Interface}} & = & \mathcal{P}(\mathsf{Name}) \otimes \mathcal{P}(\mathsf{Name}) \otimes \mathsf{Name?} \end{array}$$

An *interface* for some rules defines the imported symbols, the exported symbols and an optional axiom in the context-free sense. The imported symbols should be a subset of the required symbols which correspond to all symbols used but not defined in the rules. The exported symbols should be a subset of the defined symbols. Finally, if an axiom is given it has to be an exported symbol.

$$\frac{\begin{array}{ll} & ss_{import} \in \mathcal{P}_{finite}(\mathsf{Name}), ss_{export} \in \mathcal{P}_{finite}(\mathsf{Name}), a \in \mathsf{Name?} \\ \wedge & ss_{import} \cap ss_{export} = \emptyset \\ \wedge & a \neq ? \Rightarrow a \in ss_{export} \end{array}}{\langle ss_{import}, ss_{export}, a \rangle \in \mathsf{Interface}} \qquad [\mathsf{Interface}]$$

Well-formedness and well-typedness of programs are assumed to result from the properties of the data type Rules. Any proper program p has to satisfy well-definedness. Indeed, well-definedness is the distinguishing property of Program and Rules. To check that the rules "implement" the interface is part of the well-definedness property; refer to Subsection 2.3.3.

$$\frac{\begin{array}{ll} & \overline{rs} \in \mathsf{Rules} \\ \wedge & \overline{i} \in \mathsf{Interface} \\ \wedge & \mathcal{WD}_{\mathsf{Program}}(\overline{rs}, \overline{i}) \end{array}}{\langle \overline{rs}, \overline{i} \rangle \in \mathsf{Program}} \qquad [\mathsf{Program}]$$

To construct an interface and to lift some rules $rs \in \mathsf{Rules}$ (w.r.t. an interface) onto $\mathsf{Program}$, the following operations can be used:

$$\begin{array}{rcl}
\textbf{Interface From} \downarrow \_ \uparrow \_ & : & \mathcal{P}(\mathsf{Name}) \otimes \mathcal{P}(\mathsf{Name}) \to \mathsf{Interface} \\
\textbf{Interface From} \downarrow \_ \uparrow \_ \textbf{ Axiom Is } \_ & : & \mathcal{P}(\mathsf{Name}) \otimes \mathcal{P}(\mathsf{Name}) \otimes \mathsf{Name} \to \mathsf{Interface} \\
\textbf{Program From } \_ \textbf{ With Interface } \_ & : & \mathsf{Rules} \times \mathsf{Interface} \to \mathsf{Program}
\end{array}$$

These operations are partial like many other operations for constructing fragments. The rule [Program] models that a sequence of rules can only be considered as a proper program, if well-definedness holds.

2.2.4 Element

Again, the data type *Element* is an abstraction from constructs being relevant in many specification frameworks, e.g. conclusions or premises in natural semantics, atoms or literals in logic programming, conclusions and conditions (with free variables on one side) in algebraic specification.

An element consists of a *name* (or a *symbol*), some *inputs* and some *outputs*. Consequently, the following structural definition can be given:

$$\overline{\mathsf{Element}} = \mathsf{Name} \otimes \overline{\mathsf{Parameter}}^{\star} \otimes \overline{\mathsf{Parameter}}^{\star}$$

Proper values of Element are characterized as follows:

$$\frac{\begin{array}{ll}
 & \overline{e} \in \overline{\mathsf{Element}} \text{ is of the form } \langle n, \langle \overline{p}_1, \ldots, \overline{p}_m \rangle, \langle \overline{p}_{m+1}, \ldots, \overline{p}_k \rangle \rangle, m \geq 0, k \geq m \\
\wedge & \overline{p}_i \in \mathsf{Parameter} \text{ for } i = 1, \ldots, k \\
\wedge & \mathcal{WF}_{\mathsf{Element}}(\overline{e}) \\
\wedge & \exists\, \Gamma, \Sigma : \mathcal{WT}_{\mathsf{Element}}(\Gamma, \Sigma, \overline{e})
\end{array}}{\overline{e} \in \mathsf{Element}} \quad [\mathsf{Element}]$$

Contexts Γ associate variables (parameters) with sorts. For a given rule, it must be possible to associate each variable with a single sort.

The construction of an element e is expressed in the following mixfix notation:

$$\textbf{Element From } n\ p^{\star}_{\downarrow} \to p^{\star}_{\uparrow}$$

The basic operations for the deconstruction of elements are as follows:

$$\begin{array}{rcl}
\textbf{Name Of } \_ & : & \mathsf{Element} \to \mathsf{Name} \\
\textbf{Parameters } \_ \textbf{ Of } \_ & : & \mathsf{Io} \times \mathsf{Element} \to \mathsf{Parameter}^{\star}
\end{array}$$

Here the domain Io is defined as follows: $\mathsf{Io} = \{\mathbf{Input}, \mathbf{Output}\}$.

$$\textbf{Name Of Element From } n\ p_{\downarrow}^{\star} \to p_{\uparrow}^{\star} \to n \qquad [\textbf{Name Of}]$$

$$\textbf{Parameters Input Of Element From } n\ p_{\downarrow}^{\star} \to p_{\uparrow}^{\star} \to p_{\downarrow}^{\star} \qquad [\textbf{Parameters}.1]$$

$$\textbf{Parameters Output Of Element From } n\ p_{\downarrow}^{\star} \to p_{\uparrow}^{\star} \to p_{\uparrow}^{\star} \qquad [\textbf{Parameters}.2]$$

Some minor remarks concerning the domain Name are necessary. We assume that both, tags and names for elements, are based on the same underlying set of identifiers. This property can be facilitated to turn tags into symbols and vice versa. That possibility can be used, for example, in order to add computations to rules based on tags of the rules, or to switch from a signature to a program skeleton. Consequently, we assume coercions for both directions:

$$\begin{aligned} \textbf{Tag From} &: \mathsf{Name} \to \mathsf{Tag} \\ \textbf{Name From} &: \mathsf{Tag} \to \mathsf{Name} \end{aligned}$$

Finally, the generation of fresh names should be supported, for example, in order to be able to specify the compilation-oriented semantics of the operators for modularity in [Bro93, BMPT94]. We assume an operation $\mathcal{NEW}_{\mathsf{Name}} : \mathcal{P}(\mathsf{Name}) \to \mathsf{Name}$. $\mathcal{NEW}_{\mathsf{Name}}(N) \mapsto n$ means: n is a name not mentioned in N. There is the pragmatic problem of the proper accumulation and propagation of the set of names N being in use in a meta-program. This problem is addressed in Subsection 2.5.3.

2.2.5 Parameter

The data type $\mathsf{Parameter}$ is an abstraction for entities like terms (in natural semantics, logic programming, and algebraic specification), attributes (in attribute grammars) etc. We will have at least variables as a form of parameters. Thus, the initial structural definition is the following:

$$\overline{\mathsf{Parameter}} = (\mathsf{Variable} \oplus \cdots) \otimes \mathsf{Sort},$$

where $\mathsf{Variable}$ is a countable set of *variables*. As shown in the equation defining $\overline{\mathsf{Parameter}}$, a parameter is always expected to be associated with a sort, i.e. the type of the parameter.

Parameters are required to satisfy the properties of well-formedness and well-typedness. Consequently, proper values of $\mathsf{Parameter}$ are characterized as follows:

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

The partial function $\mathcal{TYPE}_{\mathsf{Parameter}} : \mathsf{Context} \times \mathsf{Sigma} \times \overline{\mathsf{Parameter}} \to \mathsf{Sort}$ associates a potential parameter with its sort; refer to Subsection 2.3.2 for details. Since we should be able to generate variables, there is a need for an operation $\mathcal{NEW}_{\mathsf{Variable}} : \mathcal{P}(\mathsf{Variable}) \times \mathsf{Sort} \rightarrow \mathsf{Variable}$, where $\mathcal{NEW}_{\mathsf{Variable}}(V, \sigma) \mapsto v$ means: v is a variable not mentioned in V. The sort position of $\mathcal{NEW}_{\mathsf{Variable}}$ supports the concept of meta-variables, i.e. each variable has an associated sort, that is to say σ for the above v. Similar to the generation of names, there is the pragmatic problem with generating fresh variables concerning the proper accumulation and propagation of the set of variables V being in use in a meta-program. We will return to this issue in Subsection 2.5.3 when the data types are embedded into an applicative calculus.

The construction of a parameter of a certain sort σ from a variable v is expressed as follows:

$$\textbf{Variable From } v \textbf{ Of Sort } \sigma$$

However, if we assume *meta*-variables, the application of the construction operator can be omitted in the sense of an implicit coercion.

To deconstruct parameters, the following operations are useful.

$$\begin{aligned} \textbf{Sort Of } \_ &: \mathsf{Parameter} \rightarrow \mathsf{Sort} \\ \textbf{Variable}? \ \_ &: \mathsf{Parameter} \rightarrow \mathsf{Boolean} \\ \textbf{Variable Of } \_ &: \mathsf{Parameter} \rightarrow \mathsf{Variable} \end{aligned}$$

$$\frac{\pi_{\mathsf{Sort}}(p) \rightarrow \sigma}{\textbf{Sort Of } p \rightarrow \sigma} \qquad [\textbf{Sort Of }]$$

$$\frac{\textbf{Is}_{\mathsf{Variable}}(\pi_1(p)) \rightarrow b}{\textbf{Variable}? \ p \rightarrow b} \qquad [\textbf{Variable}?]$$

$$\textbf{Variable Of Variable From } v \textbf{ Of Sort } \sigma \rightarrow v \qquad [\textbf{Variable Of}]$$

Finally, we declare an operation selecting all variables contained within a given sequence of parameters. It is frequently needed for computing closures of variables. Its specification is straightforward; refer to Section B.2.

$$\textbf{Variables In } \_ \ : \ \mathsf{Parameter}^{\star} \rightarrow \mathcal{P}(\mathsf{Variable})$$

2.3 Notions for target programs

2.3.1 Well-formedness

The data types above have been defined in such a way that a necessary precondition for the well-formedness of a compound fragment is the well-formedness of its components. To satisfy well-formedness a particular requirement for a program is the uniqueness of tags.

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

Instances of the framework can add other requirements such as:

- normal form properties of rules (as common for AGs),
- only one defining occurrence of a variable (linearity),
- pattern criteria for functional equations (e.g. non-overlapping patterns).

2.3.2 Well-typedness

Any fragment such as a rule, an element and a parameter can be associated with the type information relevant for the symbols occurring in the fragment. Let Sort be the data type of sorts. At this point, there are only symbols in the sense of Name contributing to the definition of Element. In several instances other kinds of symbols are necessary, e.g. term constructors (data type constructors) in natural semantics or function symbols (functors) in logic programming. Thus, a corresponding sum domain is established:

$$\mathsf{Symbol} \quad = \quad \mathsf{Name} \oplus \cdots$$

Every $s \in \mathsf{Symbol}$ shall be associated with a directional type based on a many-sorted type system in similarity to [Boy96a]. The notation

$$s : \sigma_1 \times \cdots \times \sigma_n \to \sigma_{n+1} \times \cdots \times \sigma_m,$$

where $s \in \mathsf{Symbol}$, $\sigma_i \in \mathsf{Sort}$, $i = 1, \ldots, m$, is used for profiles modelled by the domain Profile. $1, \ldots, n$ are regarded as the *input positions* of s, whereas $n+1, \ldots, m$ are regarded as the *output positions* of s. Note the following special cases:

- s has no input positions at all, i.e. $n = 0$, or
- s has no output positions at all, i.e. $m = n$, or
- s has no positions at all, i.e. $m = 0 = n$.

The construction of the above profile is expressed as follows:

$$\textbf{Profile From } s \ \langle \sigma_1, \ldots, \sigma_n \rangle \to \langle \sigma_{n+1}, \ldots, \sigma_m \rangle$$

The data types Profile of profiles and Sigma of signatures are defined as follows:

$$\begin{array}{rcl} \mathsf{Profile} & = & \mathsf{Symbol} \otimes \mathsf{Sort}^\star \otimes \mathsf{Sort}^\star \\ \mathsf{Sigma} & = & \mathcal{SIGMA}(\mathcal{P}(\mathsf{Profile}) \otimes \cdots) \end{array}$$

$\mathcal{SIGMA}$ restricts elements of $(\mathcal{P}(\mathsf{Profile}) \otimes \cdots)$ to proper signatures $\Sigma \in \mathsf{Sigma}$. Some common restrictions can be indicated as follows:

1. uniqueness of the profiles for any symbol, i.e. the following definition of Sigma is assumed:
$$\mathsf{Sigma} = \mathsf{Symbol} \rightarrow (\mathsf{Sort}^\star \ \otimes \ \mathsf{Sort}^\star),$$
2. function symbols have exactly one output position, as for example in logic programming.

It is assumed in the sequel that $\mathcal{SIGMA}$ satisfies at least the first condition, that is to say, symbols are not overloaded. The actual structure of Sigma possibly has to be adapted or extended in some instances, e.g. if sorts are defined by domain equations, this information will have to be maintained within a signature.

The following operations on Sigma are required:

$$\begin{array}{rcll} \_ \sqcup \_ & : & \mathsf{Sigma} \times \mathsf{Sigma} \rightarrow \mathsf{Sigma} & \text{% least upper bound (LUB)} \\ \_ \leq \_ & : & \mathsf{Sigma} \times \mathsf{Sigma} \rightarrow \{\mathbf{True}, \mathbf{False}\} & \text{% subtype relationship} \end{array}$$

$\phi_1 \cdots \phi_k$ denotes a proper signature if $\{\phi_1\} \ \sqcup \ \cdots \ \sqcup \ \{\phi_k\}$ is defined.

Example 2.3.1
For the names defined and used in our introductory interpreter example in Figure 1.2 the following profiles (directional many-sorted types) are appropriate:

$$\begin{array}{rcl} \mathit{execute} & : & \mathsf{STM} \times \mathsf{MEM} \rightarrow \mathsf{MEM} \\ \mathit{evaluate} & : & \mathsf{EXP} \times \mathsf{MEM} \rightarrow \mathsf{VAL} \\ \mathit{update} & : & \mathsf{MEM} \times \mathsf{ID} \times \mathsf{VAL} \rightarrow \mathsf{MEM} \\ \mathit{apply} & : & \mathsf{MEM} \times \mathsf{ID} \rightarrow \mathsf{VAL} \end{array}$$

Note that we also use the following term constructors in the referred specification:

$$\begin{array}{rcl} \mathsf{skip} & : & \rightarrow \mathsf{STM} \\ \mathsf{concat} & : & \mathsf{STM} \times \mathsf{STM} \rightarrow \mathsf{STM} \\ \mathsf{assign} & : & \mathsf{ID} \times \mathsf{EXP} \rightarrow \mathsf{STM} \\ \mathsf{var} & : & \mathsf{ID} \rightarrow \mathsf{EXP} \end{array}$$

We will see later on that another kind of symbols needs to be added to the definition of the domain Symbol because of terms. Any way, in some or another way we must qualify profiles by the corresponding kind of symbol. Above, we use different typesets for names and constructors.

$\diamondsuit$

Here are the basic operations for the deconstruction of profiles:

$$\begin{array}{rcl} \mathbf{Symbol\ Of}\ \_ & : & \mathsf{Profile} \rightarrow \mathsf{Symbol} \\ \mathbf{Sorts}\ \_\ \mathbf{Of}\ \_ & : & \mathsf{Io} \times \mathsf{Profile} \rightarrow \mathsf{Sort}^\star \end{array}$$

Symbol Of Profile From $s\ \sigma^\star_\downarrow \rightarrow \sigma^\star_\uparrow \rightarrow s$ [**Symbol Of**]

Sorts Input Of Profile From $s\ \sigma^\star_\downarrow \rightarrow \sigma^\star_\uparrow \rightarrow \sigma^\star_\downarrow$ [**Sorts Input**]

Sorts Output Of Profile From $s\ \sigma^\star_\downarrow \rightarrow \sigma^\star_\uparrow \rightarrow \sigma^\star_\uparrow$ [**Sorts Output**]

Iteration on signatures[1] is based on operators similar to the operations for deconstructing sequences:

$$\begin{array}{rcl}
\textbf{Head Of }\_ & : & \mathsf{Sigma} \rightarrow \mathsf{Profile} \\
\textbf{Tail Of }\_ & : & \mathsf{Sigma} \rightarrow \mathsf{Sigma} \\
\textbf{Nil?}\_ & : & \mathsf{Sigma} \rightarrow \mathsf{Boolean}
\end{array}$$

Remember that the data types Program, Rules, Rule, Element and Parameter contain *well-typed* elements only. The general framework assumes type checking / inference relations for these data types. The following rules are the basis:

$$\frac{\begin{array}{ll} & \textbf{Is}_{\mathsf{Variable}}(\pi_1(\overline{p})) \rightarrow \textbf{True} \\ \wedge & \textbf{Out}_{\mathsf{Variable}}(\pi_1(\overline{p})) \rightarrow v \\ \wedge & \pi_{\mathsf{Sort}}(\overline{p}) \rightarrow \sigma \\ \wedge & (v : \sigma) \in \Gamma \end{array}}{\mathcal{TYPE}_{\mathsf{Parameter}}(\Gamma, \Sigma, \overline{p}) \rightarrow \sigma} \quad \text{[type of a variable]}$$

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

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

We should also define the type of a program. For every $rs \in$ Rules its *type* Σ is denoted by **Sigma Of** rs.

$$\frac{\begin{array}{ll} & \mathcal{WT}_{\mathsf{Rule}}(\Sigma, r_i) \text{ for } i = 1, \ldots, n \\ \wedge & \Sigma \text{ is minimal,} \\ & \text{i.e. } \forall\ \Sigma' \neq \Sigma : \mathcal{WT}_{\mathsf{Rule}}(\Sigma', r_i) \text{ for } i = 1, \ldots, n \Rightarrow |\Sigma| \leq |\Sigma'| \end{array}}{\textbf{Sigma Of Rules From } \langle r_1, \ldots, r_n \rangle \rightarrow \Sigma} \quad [\textbf{Sigma Of }\_]$$

Note that a minimal Σ is assumed because the type of a fragment should not contain useless profiles. It is assumed that the operator **Sigma Of** is overloaded to be applicable to other fragment types as well.

[1]We assume a suitable representation for Sigma based on sequences, e.g. $(\mathsf{Symbol} \otimes \mathsf{Sort}^\star \otimes \mathsf{Sort}^\star)^\star$.

Example 2.3.2
The profiles shown in Example 2.3.1 represent exactly **Sigma Of** Figure 1.2. For all the variants of the interpreter from Section 1.2, $\mathcal{WT}$ holds because we cannot compute programs or fragements which are not well-typed. ♢

2.3.3 Well-definedness

A program is said to be *well-defined* if it satisfies certain criteria particularly important for complete programs. Here are some examples:

1. variables are bound from left to right and they are not used before they are bound (refer e.g. to *RML* [Pet95, Pet94]—a variant of natural semantics),
2. reducedness of the underlying context-free grammar of an AG,
3. non-circularity of an attribute grammar,
4. absence of free λ-variables in functional equations,
5. call- or i/o-correctness in logic programs with directional types [Boy96a].

To approach to an initial form of well-definedness in our framework we need some further notions. Parameter positions in a rule are divided into *defining* and *applied positions.*

Definition 2.3.1
Let be $r \in \mathsf{Rule}$. The input positions of the conclusion of r and the output positions of the premises of r are called *defining* positions of r. A position in r which is not a defining position, is called an *applied* position of r. ♢

The idea behind these terms 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 extended attribute grammars [WM77]. There are other terms used for this purpose, e.g. imported and exported positions in directional typing [Boy96a]. In attribute grammars, notions like *used* and *defined attribute occurrences* are defined. The latter terms are tuned towards named attributes rather than a position-oriented framework as in our case.

Now it is straightforward to define the sets **Ao In** r and **Do In** r for a given rule r which represent the applied and the defining (variable) occurrences of r, i.e. all the variables occurring on applied or defining positions respectively. Refer to Section B.3 for the inference rules defining the corresponding relations.

The following definition of well-definedness is assumed. A requirement for rules in the context of directional types of symbols is that every applied occurrence of a variable is justified by a defining occurrence of the same variable. This data flow criterion is modelled by the rule $\mathcal{DF}.1$ below. Moreover, a kind of *conformance* should be satisfied between the rules and the interface of a program; refer to the rule $\mathcal{IMPLEMENTS}$.

$$\frac{\begin{array}{ll} & \mathcal{DF}(\overline{rs}, \overline{i}) \\ \wedge & \mathcal{IMPLEMENTS}(\overline{rs}, \overline{i}) \end{array}}{\mathcal{WD}_{\mathsf{Program}}(\langle \overline{rs}, \overline{i} \rangle)} \qquad [\mathcal{WD}_{\mathsf{Program}}]$$

$$\frac{\textbf{Ao In}\ \overline{r}_j \subseteq \textbf{Do In}\ \overline{r}_j \text{ for } j = 1,\ldots,n}{\mathcal{DF}(\langle \overline{r}_1,\ldots,\overline{r}_n\rangle, \overline{i})} \quad [\mathcal{DF}.1]$$

$$\frac{\begin{array}{ll} & ss_{export} \text{ are defined symbols in } \overline{rs} \\ \wedge & ss_{import} \text{ are not defined symbols in } \overline{rs} \\ \wedge & \overline{rs} \text{ is reduced in the context-free sense w.r.t. } ss_{import}, ss_{export}, a \end{array}}{\mathcal{IMPLEMENTS}(\overline{rs}, \langle ss_{import}, ss_{export}, a\rangle)} \quad [\mathcal{IMPLEMENTS}]$$

We can speak about *reducedness* in the context-free sense, if we regard the imported names of the interface as "terminals", whereas the exported names are regarded as "nonterminals". If the interface specifies an axiom, reducedness can be checked in its full extent. Otherwise, we can only check that all required names (i.e. names which are used but not defined) are listed as imported names.

A very simple and common data flow criterion is obtained by restricting the dependencies of variables in the elements of a rule from left to right; refer to Section B.4 for details.

Well-definedness is only required for complete programs, because transformations can often be defined in a more convenient way, if intermediate results do not have to satisfy the properties modelled by well-definedness. The process of establishing a propagation pattern along certain symbols of a set of rules, for example, can be divided into two phases. The first phase adds new parameter positions, whereas the second phase establishes a certain data flow based on the new positions. The intermediate result will not be well-defined.

Example 2.3.3
Recall our introductory interpreter example of Subsection 1.2.2. The orginal interpreter in Figure 1.2 and the final adaptation in Figure 1.4 coping with side effects during expression evaluation are well-defined, i.e. $\mathcal{WD}$ holds, whereas the intermediate result in Figure 1.3 with the new output position of sort MEM is not well-defined because of the variable MEM' which only occurs on an applied position of the rule [var], but not on a defining position, i.e. $\mathcal{WD}$ does not hold due to $[\mathcal{DF}.1]$. $\diamondsuit$

The final result of a meta-program must be well-defined. The following definition defines the term of *undefined variables* in a rule. We might also say that these variables are not defined. These variables are exactly those variables which violate well-definedness and thus meta-programs have to focus on them. Dually, we can also speak of *unused variables* corresponding to useless variable occurrences in a target program. Unused variables are not regarded as a violation of well-definedness but still they are useful to control meta-programs.

Definition 2.3.2
Let be $r \in \mathsf{Rule}$. **Ao In** $r \setminus$ **Do In** r denotes the set of all *undefined variables* in r. Dually, **Do In** $r \setminus$ **Ao In** r denotes the set of all *unused variables* in r. $\diamondsuit$

Note that there can be several applied occurrences of an undefined variable in a rule. There are two basic ways in which an occurrence of an undefined variable v can be eliminated in a rule r.

- The variable v is replaced in the corresponding occurrence by another variable, more generally by a parameter not referring to v.
- A defining occurrence of v can be created, most likely by the insertion of a new premise with v on an output position.

Both approaches model somehow the addition of a semantic rule in the AG jargon, where the first approach corresponds to the addition of semantic copy rules. Unused variables can obviously be eliminated in a dual manner. Let us declare some useful terms. If a premise p is inserted in order to eliminate an undefined variable v, then p is also called *definition* (of v). Dually, if a premise p is inserted in order to eliminate an unused variable v, then p is also called *use* (of v). Definitions are basically constant computations, i.e. premises with zero input positions and one output position. Uses are basically unary conditions, i.e. premises with one input position and zero output positions. Finally, if a parameter on an applied position is replaced by a variable with a defining occurrence, more generally by a parameter without undefined variables, the resulting parameter is called a *copy*.

2.3.4 Substitution and unification

The notions *substitution* and *unification* are well-established in the declarative paradigm. Usually they are used to describe the meaning of programs or to explain the syntactical proof derivation. In the context of meta-programming, we need these concepts at the *meta-level* to perform "symbolic" substitution and unification in meta-programs. Note that additional requirements for our kind of substitution and unification arise from our well-typedness constraints.

In a formal sense, a substitution is a mapping from variables to parameters. As it is common practice, we use a rather syntactic definition based on pairs of variables and parameters:

$$\overline{\mathsf{Substitution}} = \mathcal{P}(\mathsf{Variable} \otimes \overline{\mathsf{Parameter}})$$

$\langle v, p\rangle$[2] means that v has to be substituted by p in parameters, elements and others. *Proper* substitutions, i.e. elements of $\mathsf{Substitution}$, are such sets $\{\langle v_1, p_1\rangle, \ldots, \langle v_n, p_n\rangle\}$, where $n \geq 0$, $v_i \neq p_i$ for $i = 1, \ldots, n$ and $v_i \neq v_j$ for $i \neq j$. For several instances, e.g. natural semantics, compound parameters in the sense of terms need to be considered. Then the sort of any nested occurrence of a variable v_i in the parameters p_j must not be in conflict with the sort of p_i, and the LUB of the type information associated with the p_j has to exist in order to retain well-typedness. Consequently, the concatenation of substitutions is restricted. Let θ and σ be substitutions. $\theta \bowtie \sigma = \theta \cup \sigma$ provided $\theta \cup \sigma$ denotes a proper substitution.

Substitutions can be applied to parameters. The application of substitutions can straightforwardly be generalized for other syntactical domains, such as elements and rules.

[2]The notation v/p is used quite often in the literature instead of $\langle v, p\rangle$.

$$\begin{array}{rcl}
\textbf{Substitute \_ In Parameter \_} & : & \mathsf{Substitution} \times \mathsf{Parameter} \rightarrow \mathsf{Parameter} \\
\textbf{Substitute \_ In Element \_} & : & \mathsf{Substitution} \times \mathsf{Element} \rightarrow \mathsf{Element} \\
\textbf{Substitute \_ In Rule \_} & : & \mathsf{Substitution} \times \mathsf{Rule} \rightarrow \mathsf{Rule}
\end{array}$$

We give only the inference rules for the application of substitutions to parameters. Remember that there are only variables as parameters in the general framework.

$$\frac{\begin{array}{rl} & \textbf{Variable?}\ p \rightarrow \textbf{True} \\ \wedge & \textbf{Variable Of}\ p = v \\ \wedge & \textbf{Sort Of}\ p = \textbf{Sort Of}\ p' \end{array}}{\textbf{Substitute}\ \{\ldots, \langle v, p' \rangle, \ldots\}\ \textbf{In Parameter}\ p \rightarrow p'} \qquad [\mathcal{SUBST}.1]$$

$$\frac{\begin{array}{rl} & \textbf{Variable?}\ p \rightarrow \textbf{True} \\ \wedge & \textbf{Variable Of}\ p \neq v_i \text{ for } i = 1, \ldots, n \end{array}}{\textbf{Substitute}\ \{\langle v_1, p_1 \rangle, \ldots, \langle v_n, p_n \rangle\}\ \textbf{In Parameter}\ p \rightarrow p} \qquad [\mathcal{SUBST}.2]$$

The application of a substitution has to be restricted to retain well-typedness as expressed by the above inference rules. For notational convenience we may write $e\ [v/p]$ for the application of a substitution $\{\langle v, p \rangle\}$ to a syntactical entity e.

Another important operation on substitutions is *composition* coinciding with function composition. Given two substitutions $\theta = \{\langle v_1, p_1 \rangle, \ldots, \langle v_n, p_n \rangle\}$ and $\sigma = \{\langle v_{n+1}, p_{n+1} \rangle, \ldots, \langle v_m, p_m \rangle\}$, their composition is denoted by $\sigma \circ \theta$, and it is obtained from the set $\{\langle v_1, p_1\ \sigma \rangle, \ldots, \langle v_n, p_n\ \sigma \rangle, \langle v_{n+1}, p_{n+1} \rangle, \ldots, \langle v_m, p_m \rangle\}$ by removing all $\langle v_i, p_i\ \sigma \rangle$ with $v_i = p_i\ \sigma$ for $i = 1, \ldots, n$ and by removing those $\langle v_j, p_j \rangle$ for which $v_j \in \{v_1, \ldots, v_n\}$ for $j = n+1, \ldots, m$.

The concept of substitution permits us to introduce another concept, that is *unification* of parameters, similarly to logic programming; refer e.g. to [NM95]. The computation of the most general unifier of equations $\{\langle t_1, t'_1 \rangle, \ldots, \langle t_n, t'_n \rangle\}$ is based on deriving an equivalent set of equations in solved form according to Robinson's algorithm. As usual, a set of equations is in *solved form* if the LHS of every equation is a variable, and the variables do not occur in the parameters on the RHSs. Two sets of equations are *equivalent* if they have the same sets of unifiers. Thus, we need an auxiliary domain $\overline{\mathsf{Equations}} = \mathcal{P}(\overline{\mathsf{Parameter}} \otimes \overline{\mathsf{Parameter}})$. Proper sets of equations have to be restricted to retain well-typedness similarly to Substitution.

As far as unification is concerned, the following relations are needed:

$$\begin{array}{rcl}
\textbf{Unify Parameters \_ And \_} & : & \mathsf{Parameter} \times \mathsf{Parameter} \rightarrow \mathsf{Substitution} \\
\mathcal{SOLVE}(\_) & : & \mathsf{Equations} \rightarrow \mathsf{Substitution}
\end{array}$$

$$\frac{\mathcal{SOLVE}(\{\langle p, p'\rangle\}) \to \theta}{\textbf{Unify Parameters } p \textbf{ And } p' \to \theta} \qquad [\mathcal{MGU}.1]$$

There should be possibly also a relation to test if a unifier exists at all. Refer to Section B.5 for details of the relation $\mathcal{SOLVE}$, which computes the solved form of some equations. Unification is easily generalized for elements. Unfication of elements (with the same underlying profile) corresponds to the unification of their parameters in the same positions. Thus the following definition is appropriate:

$$\textbf{Unify Element \_ And \_} \quad : \quad \mathsf{Element} \times \mathsf{Element} \to \mathsf{Substitution}$$

$$\frac{\mathcal{SOLVE}(\{\langle p_1^{\downarrow}, p'^{\downarrow}_1\rangle, \ldots, \langle p_m^{\downarrow}, p'^{\downarrow}_m\rangle, \langle p_1^{\uparrow}, p'^{\uparrow}_1\rangle, \ldots, \langle p_k^{\uparrow}, p'^{\uparrow}_k\rangle\}) \to \theta}{\begin{array}{rl} \textbf{Unify Elements} & \textbf{Element From } n \ \langle p_1^{\downarrow}, \ldots, p_m^{\downarrow}\rangle \to \langle p_1^{\uparrow}, \ldots, p_k^{\uparrow}\rangle \\ \textbf{And} & \textbf{Element From } n \ \langle p'^{\downarrow}_1, \ldots, p'^{\downarrow}_m\rangle \to \langle p'^{\uparrow}_1, \ldots, p'^{\uparrow}_k\rangle \\ \to & \theta \end{array}} \qquad [\mathcal{MGU}.2]$$

Subsitution is useful in meta-programs, for example, for establishing a certain data flow by the unification of variables in a suitable way. Unification is useful in meta-programs, for example, for unfolding, i.e. a certain premise is unfolded according to a rule defining the premise's symbol.

2.3.5 Addressing fragments

Meta-programs frequently need to address (select) certain fragments in programs, namely:

- a rule in a sequence of rules,
- a premise in a sequence of premises,
- a (parameter) position of a conclusion / a premise,
- a defining or an applied position.

The purpose of this subsection is to comment on such fragment selections in more detail. Values intended to address fragments are called selectors in the sequel. Usually a selector is expected to select uniquely a fragment. However, in some cases it is acceptable that a selector matches with several fragments or even with no fragment at all.

Rules can easily be selected based on tags. For certain instances of the framework a selection based on parameter patterns (for the input positions of the conclusions) makes sense as well, e.g. for recursive function definitions and constructive algebraic specifications.

Premises of rules can be addressed, in general, by indices. There are the following objections for using this addressing method:

1. It is not readable.
2. It depends on the order of the premises and on the absence or presence respectively of premises possibly not relevant for the actual selection.

3. For certain forms of premises, e.g. semantic rules in an AG, there is no natural order because the order is semantically meaningless.

Consequently, names of premises should be used for selection. If a name is actually used to select a certain premise, we must ensure that the name can be used as a unique selector.

Definition 2.3.3
Let be $s \in \mathsf{Name}$, $r \in \mathsf{Rule}$. We say, s is a *unique selector for a premise* of r if $|\{i \mid \textbf{Name Of } e_i = s \text{ with } i = 1, \ldots, n\}| = 1$, where **Premises Of** $r = e_1, \ldots, e_n$. ♢

Definition 2.3.3 can be easily generalized to cope with other forms of premises.

Example 2.3.4
Let us consider again the introductory interpreter example in Figure 1.2. All the names *execute*, *evaluate*, *update* and *apply* can be used as unique selectors for premises, except for *execute* in the rule [concat] because there are two matching premises. ♢

A rule violating the property of premise selection can be folded in a way that new auxiliary symbols are used instead of the non-unique symbol. Thereby, unique selection is achieved. Another possibility is the augmentation of premises with selectors as in several specification languages.

Now addressing positions in premises and conclusions is regarded. Since we use a position-oriented framework, symbols have many-sorted profiles. That is suitable for logic programming, logical grammars, GSFs, algebraic specifications and natural semantics. In contrast to such frameworks, attribute grammars in the Knuthian style are based on attributes instead of (sorts associated with) positions. The attributes have pairwise disjoint names for a given grammar symbol. Thus, the names allow attributes to be addressed uniquely. In a position-oriented framework, (indices of) positions must be applied, in general, as unique selectors. There are two points to be criticised with regard to this method based on indices:

1. It is not readable.
2. It crucially depends on the order of the positions.

The latter point is critical because transformations using this poor addressing method cannot be applied to programs with a different order in the profiles. We will even not necessarily realize that the wrong positions are addressed. Consequently, we look for an addressing method based on sorts rather than indices.[3]

The following definition captures the necessary and sufficient condition for a symbol so that its positions can be uniquely addressed based on sorts.

Definition 2.3.4
Let be $s \in \mathsf{Symbol}, rs \in \mathsf{Rules}$. s is *uniquely sorted* in rs, if $\sigma_i \neq \sigma_j$ for $i, j = 1, \ldots, n$ and for $i, j = n+1, \ldots, m$, where $i \neq j$ and $s : \sigma_1 \times \cdots \times \sigma_n \to \sigma_{n+1} \times \cdots \times \sigma_m$ is the profile of s according to the type of rs. ♢

[3]Instead of using sorts, we also could adopt the structural definitions of profiles and/or elements to cope with *key* parameters.

Example 2.3.5
In the interpreter in Figure 1.2 all symbols are uniquely sorted. Refer to Example 2.3.1 for the profiles of the symbols. In contrast to that, the symbol *append*, for example, is not uniquely sorted in the interpreter in Figure 1.9 coping with I/O. $\diamondsuit$

If a symbol is not used in some rules at all it is not uniquely sorted by definition. Note that it is usually sufficient to consider the uniqueness of a particular sort σ w.r.t. a symbol s when σ is used as a selector among the input or output position of sort σ of s. The following definition explains how positions are selected.

Definition 2.3.5
Let be $io \in \mathsf{Io}$, $s \in \mathsf{Symbol}$, $\sigma \in \mathsf{Sort}$, $rs \in \mathsf{Rules}$. We say, the triple $\langle io, s, \sigma\rangle \in \mathsf{Position}$ is a *unique selector for a position* in rs, if

- $|\{i|\sigma_i = \sigma \text{ with } i = 1, \ldots, n\}| = 1$, for $io =$ **Input**,
- $|\{i|\sigma_i = \sigma \text{ with } i = n+1, \ldots, m\}| = 1$, for $io =$ **Output**,

where the profile of s in rs is $s : \sigma_1 \times \cdots \times \sigma_n \to \sigma_{n+1} \times \cdots \times \sigma_m$. $\diamondsuit$

Example 2.3.6
We continue Example 2.3.5. The symbol *append* is not uniquely sorted in Figure 1.9, because $\langle$**Input**$, append, \mathsf{OUT}\rangle$ is not a unique selector for a position. $\diamondsuit$

As far as meta-programs are concerned, it must be ensured that the result of a transformation is not defined if improper selectors are involved, i.e. strictness of meta-programs with regard to failures arising from selections is to be preferred.

If a meta-program has to incorporate computational elements, e.g. definitions for applied positions or uses for defining positions if it has to copy parameters from defining to applied positions, the unique selection of defining and applied positions is crucial.

Definition 2.3.6
Let be $io \in \mathsf{Io}$, $s \in \mathsf{Symbol}$, $\sigma \in \mathsf{Sort}$, $r \in \mathsf{Rule}$.
We say, the triple $\langle io, s, \sigma\rangle$ is a *unique selector for a defining position* in r, if $\langle io, s, \sigma\rangle$ is a unique selector for a position in r and

- for $io =$ **Input**: **Name Of Conclusion Of** $r = s$,
- for $io =$ **Output**: s is a unique selector for a premise of r.

Dually, we say, the triple $\langle io, s, \sigma\rangle$ is a *unique selector for an applied position* in r, if $\langle io, s, \sigma\rangle$ is a unique selector for a position in r and

- for $io =$ **Input**: s is a unique selector for a premise of r,
- for $io =$ **Output**: **Name Of Conclusion Of** $r = s$.

$\diamondsuit$

Example 2.3.7
We comment again on the interpreter from Figure 1.2. $\langle$**Input**, *execute*, STM$\rangle$ is a unique selector of a defining position in the rule [concat], whereas it is not a unique selector for an applied position because there are two matching premises. $\diamondsuit$

The property of unique selectors for applied and defining positions is important if several positions have to be selected simultaneously, e.g. if a condition of arity > 1 is added to a rule: Consider selectors $p_1, \ldots, p_n$ for the positions of the condition. If some selector turns out to be an improper selector in the sense that no matching position can be determined, the whole condition cannot be assembled. However, even *uniqueness* is needed because which of the matching positions should be selected otherwise.

On the other hand, there is no need to insist on unique selectors for defining and applied positions in general, e.g. if it should be ensured that single positions are used by unary conditions or they are defined by constant computations. If the selection is not unique, the corresponding number of computational elements can be added. The case of no matching position can be accepted in such a way that no computational element is added. Indeed, the operator suite offers operators adhering to that style; refer e.g. to the operators **Define** and **Use** presented in Subsection 3.3.2

Finally, we want to mention another opportunity for selecting rules, premises and parameters, that is to say a selection based on pattern matching. In [DC90, Ada91] it is argued that pattern matching is useful for the simultaneous selection of entities, e.g. for the association of semantic rules with syntactical rules, where the underlying rule pattern of the semantic rules must be matched with an actual syntactical rule. In contrast to that, we are interested here in more atomic selections.

2.4 Instances

We *instantiate* the general framework for certain target languages, e.g. natural semantics, attribute grammars, logic programming and algebraic specification. New forms of constructs, e.g compound parameters or refinements of the notions well-definedness, well-typedness etc. need to be taken into consideration. A selection of the features discussed in this Section will be used in the next Chapter as the underlying instance for establishing an operator suite for meta-programming. It should be clear that certain extensions, e.g. function symbols and thus terms for logic programs, permit us to consider specific meta-operations. Nevertheless, we try to understand the instances below as "conservative" extensions of the general framework, that is to say we want to indicate how specific features, e.g. terms, can be simulated in the general framework. We take advantage of research concerning the relation between different representatives of the declarative paradigm; refer e.g. to [CFZ82a, CFZ82b, PW80, DM85, DM93, AFZ88, Att89, AC90, AP94]. Some arguments also arise from common programming practice, e.g. the simulation of products as targets for profiles in algebraic specification by means of dedicated tupling constructors.

2.4.1 Natural Semantics

The data type $\overline{\mathsf{Rule}}$ obviously coincides with rules in the sense of natural semantics. The data type $\overline{\mathsf{Parameter}}$ has to be extended to cope with compound parameters, i.e. terms. Terms (compound parameters) are needed for the representation of abstract syntax trees, semantic objects, environments, contexts, and other data. In *RML* [Pet95, Pet94], for example, SML-like algebraic data types are used for that purpose. Well-formedness possibly concerns the proper distinction between names of relations and data type constructors. Our type system as outlined in the general framework is very much the same like in *RML*, although we had to put extra effort to achieve polymorphism. Thus, the refinement of $\mathcal{WT}$ is straightforward. Concerning well-definedness, we adopt the philosophy of *RML*, where it is defined exactly when variables are bound. Since, *RML* executes premises from left to right, it also binds variables in that order. If we assume, that the inputs of a premise must be known before it is executed, we will obtain a data-flow criterion similar to L-attribution in AGs.[4] First, we present the refinement of the structural definition of the data types:

$$\begin{aligned}
\overline{\mathsf{Parameter}} &= (\mathsf{Variable} \oplus \overline{\mathsf{Term}}) \otimes \mathsf{Sort} \\
\overline{\mathsf{Term}} &= \mathsf{Constructor} \otimes \overline{\mathsf{Parameter}}^{\star} \\
\mathsf{Constructor} &= \mathsf{Id} \\
\mathsf{Symbol} &= \mathsf{Name} \oplus \mathsf{Constructor}
\end{aligned}$$

$\mathcal{SIGMA}$ has to restrict the targets of data type constructors to single sorts (in contrast to arbitrary finite sequences). The definition of operations for the construction and deconstruction of compound parameters is straightforward. A compound parameter consisting of a constructor f and parameters $p^{\star}$ is constructed as follows:

$$\textbf{Term From } f\ p^{\star} \textbf{ Of Sort } \sigma$$

Here σ denotes the sort of the resulting term, i.e. f is supposed to be a constructor with a profile $f : \sigma_1 \times \cdots \sigma_n \to \sigma$, where the σ_i are supposed to match with the types of the parameters $p^{\star}$. Deconstruction of terms and tests for different kinds of symbols is facilitated by the following operations:

$$\begin{aligned}
\textbf{Term}?\ \_ &: \mathsf{Parameter} \to \mathsf{Boolean} \\
\textbf{Constructor Of} &: \mathsf{Parameter} \to \mathsf{Constructor} \\
\textbf{Subterms Of} &: \mathsf{Parameter} \to \mathsf{Parameter}^{\star} \\
\textbf{Name}?\ \_ &: \mathsf{Symbol} \to \mathsf{Boolean} \\
\textbf{Constructor}?\ \_ &: \mathsf{Symbol} \to \mathsf{Boolean}
\end{aligned}$$

[4] If *unknowns*, which play an important role in some applications of natural semantics, are taken into consideration, a more general approach must be followed to.

Refer to Section B.6 for the straightforward inference rules. Moreover, coercions should be assumed to coerce constructors into tags and names (**Tag From**, **Name From**) and vice versa (**Constructor From**).

Types of compound parameters are determined as follows:

$$\frac{\begin{array}{ll} & \mathbf{Is}_{\overline{\mathsf{Term}}}(\pi_1(\overline{p})) \to \mathbf{True} \\ \wedge & \mathbf{Out}_{\overline{\mathsf{Term}}}(\pi_1(\overline{p})) \to \langle f, \langle \overline{p}_1, \ldots, \overline{p}_n \rangle \rangle \\ \wedge & \pi_{\mathsf{Sort}}(\overline{p}) \to \sigma \\ \wedge & f : \sigma_1 \times \cdots \times \sigma_n \to \sigma \in \Sigma \\ \wedge & \mathcal{TYPE}_{\mathsf{Parameter}}(\Gamma, \Sigma, \overline{p}_i) \to \sigma_i \text{ for } i = 1, \ldots, n \end{array}}{\mathcal{TYPE}_{\mathsf{Parameter}}(\Gamma, \Sigma, \overline{p}) \to \sigma} \quad \text{[type of a compound parameter]}$$

Section B.6 also generalizes the application of substitutions to parameters for compound parameters. Unification has to be generalized by implementing Robinson's unification algorithm by the relation $\mathcal{SOLVE}$ taking into consideration additional constraints due to well-typedness; refer also to Subsection 2.3.4 and Section B.5.

$\ldots$

$$\frac{\boxed{out_{\mathbf{skip}}(\mathsf{STM})}}{execute(\boxed{\mathsf{STM}}, \mathsf{MEM}) \to (\mathsf{MEM})} \quad \text{[skip]}$$

$$\frac{\begin{array}{ll} & \boxed{out_{\mathbf{concat}}(\mathsf{STM}) \to (\mathsf{STM}_1, \mathsf{STM}_2)} \\ \wedge & execute(\mathsf{STM}_1, \mathsf{MEM}_0) \to (\mathsf{MEM}_1) \\ \wedge & execute(\mathsf{STM}_2, \mathsf{MEM}_1) \to (\mathsf{MEM}_2) \end{array}}{execute(\boxed{\mathsf{STM}}, \mathsf{MEM}_0) \to (\mathsf{MEM}_2)} \quad \text{[concat]}$$

$$\frac{\begin{array}{ll} & \boxed{out_{\mathbf{assign}}(\mathsf{STM}) \to (\mathsf{ID}, \mathsf{EXP})} \\ \wedge & evaluate(\mathsf{EXP}, \mathsf{MEM}_0) \to (\mathsf{VAL}) \\ \wedge & update(\mathsf{MEM}_0, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{MEM}_1) \end{array}}{execute(\boxed{\mathsf{STM}}, \mathsf{MEM}_0) \to (\mathsf{MEM}_1)} \quad \text{[assign]}$$

$\ldots$

$$\frac{\begin{array}{ll} & \boxed{out_{\mathbf{var}}(\mathsf{EXP}) \to (\mathsf{ID})} \\ \wedge & apply(\mathsf{MEM}, \mathsf{ID}) \to (\mathsf{VAL}) \end{array}}{evaluate(\boxed{\mathsf{EXP}}, \mathsf{MEM}) \to (\mathsf{VAL})} \quad \text{[var]}$$

$\ldots$

Figure 2.4: Figure 1.2 in the pure framework

We should make clear that this instance is a modest extension of the general framework. We can represent any target program of this instance in the general framework in the following way. Applications of data type constructors can be regarded as special premises modelling term construction (for applied positions) or deconstruction (for defining positions); refer for example to Figure 2.4 showing the "pure" variant of the natural semantics from Figure 1.2. It should be pointed out that the transformation from natural semantics to the pure framework (and vice versa) can be specified in the instance of natural semantics itself. All transformations which are applicable for the general framework are applicable to natural semantics without further adaptation. It is also very comfortable that we can deal with data type constructors in much the same way as with premises. There is no need for additional tools.

Let us conclude this subsection with remarks on certain features of particular variants of natural semantics. In *Typol* [Des88, BCD$^+$88, JRG92] subjects and predicates are regarded as different forms of premises. Actually, predicates are called rather computations than premises. It is a convention which does not add expressive power. Nevertheless, an adaptation of the calculus with two different forms of premises will be demonstrated for the instance of Grammars of Syntactical Functions; refer to Subsection 2.4.5 because there is more convincing argument for such a distinction. Finally, if unknowns are to be used, there are two options: Unknowns are declared as in *RML* or there is no explicit declaration as in *Typol*. To declare an unknown can be considered as a simple form of a premise. Well-definedness has to be adjusted accordingly if unknowns are used.

2.4.2 Logic Programming

It is obvious that logic programming with sorts and modes can be regarded as an instance of the general framework. The most important extension, i.e. terms, can be performed in the same way as for the instance of natural semantics. By the way, the similarity of inference rules and definite clauses is the basis for the translation of natural semantics into Prolog rules providing an option for executing the executable specification formalism *Typol* for natural semantics as integrated in the Centaur system [Des88, BCD$^+$88, JRG92].

The data type $\overline{\mathsf{Rule}}$ obviously coincides with definite clauses. The data type $\overline{\mathsf{Parameter}}$ has to be extended to cope with terms. Well-formedness possibly concerns the proper distinction between predicate and function symbols. Our type system as outlined in the general framework can be regarded as a many-sorted type system of a Gödel-like [HL94] logic programming language. Thus, the refinement of $\mathcal{WT}$ is straightforward. In logic programming, there are notions like "well-modedness", which are appropriate as an instance of $\mathcal{WD}$, refer e.g. to call- or i/o-correctness in [Boy96a].

To consider elements and not other forms of premises corresponds to *pure* logical programs. Several forms of premises can be handled like $\mathsf{Element}$, e.g. negative literals, matching constructs and computations according to predefined (impure) predicates (e.g. for dealing with numbers or atoms). Computations will be considered in more detail for the instance of Grammars of Syntactical Functions; refer to Subsection 2.4.5.

2.4.3 Algebraic Specification

The general framework can easily be instantiated for algebraic specification [HL89, LEW96, LNC91, Mos97]. As a matter of fact, we are mainly concerned with *constructive* specifications (refer e.g. to [HL89]), i.e. the LHS of a term equation (or rewrite rule) must have a non-constructor function symbol as an outermost symbol applied to terms without non-constructor function symbols. The corresponding distinction between non-constructor operations and constructors and the restricted structure of term equations fit with the general framework; refer to Figure 2.5 for constructive term equations describing the dynamic semantics in much the same way as the interpreter in the style of natural semantics in Figure 1.2 does.

$$\begin{array}{lll}
\ldots & & \\
\mathit{execute}(\mathsf{skip}, \mathsf{MEM}) = \mathsf{MEM} & & [\mathrm{skip}] \\
\mathit{execute}(\mathsf{concat}(\mathsf{STM}_1, \mathsf{STM}_2), \mathsf{MEM}) = \mathit{execute}(& \mathsf{STM}_2, & \\
 & \mathit{execute}(\mathsf{STM}_1, \mathsf{MEM}) & [\mathrm{concat}] \\
) & & \\
\mathit{execute}(\mathsf{assign}(\mathsf{ID}, \mathsf{EXP}), \mathsf{MEM}) = \mathit{update}(& \mathsf{MEM}, \mathsf{ID}, & \\
 & \mathit{evaluate}(\mathsf{EXP}, \mathsf{MEM}) & [\mathrm{assign}] \\
) & & \\
\ldots & & \\
\mathit{evaluate}(\mathsf{var}(\mathsf{ID}), \mathsf{MEM}) = \mathit{apply}(\mathsf{MEM}, \mathsf{ID}) & & [\mathrm{var}] \\
\ldots & &
\end{array}$$

Figure 2.5: An algebraic specification for the interpreter from Figure 1.2

We first have to look for the counterpart for rules in algebraic specification. Obviously, term equations and the data type Rule have to be related to each other. There is no direct correspondence because the RHS of a constructive term equation is simply a term and it is not some sequence of premises as in the general framework. To adopt the general framework for algebraic specifications, we can use a normalization procedure as follows: RHSs are *flattened* by taking the sequence of applications of non-constructor operations in the order of nesting adhering to the *call-by-value* evaluation strategy.

Consequently, LHSs of constructive term equations and conclusions of the data type Rule coincide. RHSs of term equations are simplified to obtain a sequence of premises corresponding to applications with at most one non-constructor operation. Terms are not a problem at all, since we can deal with them in the same way as for instance of natural semantics; refer to Subsection 2.4.1. Indeed, by flattening the algebraic specification in Figure 2.5 we obtain a variant literally equivalent to the interpreter in Figure 1.2 in the style of natural semantics.

A minor problem with algebraic specifications with respect to the data types of the general framework is that an operation has exactly a *single* sort as its target in algebraic specification, whereas the general framework promotes any arity (including 0). This flexibility concerning output positions is crucial for our approach. There are two solutions to this problem:

- Extending the result of a non-constructor operation is considered as the extension of a dedicated constructor profile where this constructor is used as a kind of tupling constructor for the result of the operation. This approach corresponds to the programming practice in algebraic specification where auxiliary sorts for compound function results are introduced.

- We allow non-constructor operations to have Cartesian products as their targets. Finally, such a specification can be transformed into a usual algebraic specification by associating non-constructor operations with proper products as targets with dedicated constructors in the sense of the first solution. For notational convenience, we can assume that (), (_), (_, _), ... denote these (overloaded) tupling constructors.

The normalization of term equations induces an order of premises and thereby an order of introduced variables from left to right. Thereby, well-definedness is straightforward to define.

$$\frac{\mathit{evaluate}(\mathsf{EXP}, \mathsf{MEM}) = \mathsf{true}}{\mathit{execute}(\mathsf{if}(\mathsf{EXP}, \mathsf{STM}_1, \mathsf{STM}_2), \mathsf{MEM}) = \mathit{execute}(\mathsf{STM}_1, \mathsf{MEM})} \quad [\mathsf{if.true}]$$

$$\frac{\mathit{evaluate}(\mathsf{EXP}, \mathsf{MEM}) = \mathsf{false}}{\mathit{execute}(\mathsf{if}(\mathsf{EXP}, \mathsf{STM}_1, \mathsf{STM}_2), \mathsf{MEM}) = \mathit{execute}(\mathsf{STM}_2, \mathsf{MEM})} \quad [\mathsf{if.false}]$$

Figure 2.6: Conditional equations defining the dynamic semantics of the if-construct

Conditional rewrite rules as for example in *ASF*(+*SDF*) [Kli93] can also be reduced to rules in the sense of the general framework. For a proper conditional rewrite rule it must be satisfied that non-constructor operation symbols must not occur on a variable-introducing side because such a side cannot be reduced anyway. To represent conditional rewrite rules in the general framework, not only the RHSs of the conclusions need to be flattened as for an ordinary algebraic specification, but the conditions must be flattened as well.

Consider Figure 2.6 for the conditional rewrite rules describing the interpretation of an if-statement. The flattened variant is shown in Figure 2.7.

$$\frac{\begin{array}{ll} & \mathit{evaluate}(\mathsf{EXP}, \mathsf{MEM}_0) \to (\mathsf{true}) \\ \wedge & \mathit{execute}(\mathsf{STM}_1, \mathsf{MEM}_0) \to (\mathsf{MEM}_1) \end{array}}{\mathit{execute}(\mathsf{if}(\mathsf{EXP}, \mathsf{STM}_1, \mathsf{STM}_2), \mathsf{MEM}_0) \to (\mathsf{MEM}_1)} \quad [\mathsf{if.true}]$$

$$\frac{\begin{array}{ll} & \mathit{evaluate}(\mathsf{EXP}, \mathsf{MEM}_0) \to (\mathsf{false}) \\ \wedge & \mathit{execute}(\mathsf{STM}_2, \mathsf{MEM}_0) \to (\mathsf{MEM}_1) \end{array}}{\mathit{execute}(\mathsf{if}(\mathsf{EXP}, \mathsf{STM}_1, \mathsf{STM}_2), \mathsf{MEM}_0) \to (\mathsf{MEM}_1)} \quad [\mathsf{if.false}]$$

Figure 2.7: "Pure" variant of Figure 2.6

An extension for negative equations as in *ASF*(+*SDF*) is straightforward. The basic form of premise corresponds to positive equations, whereas another form of premise is needed for negative equations.

$$\begin{aligned}
\overline{\mathsf{Premise}} &= \overline{\mathsf{Positive}} \oplus \overline{\mathsf{Negative}} \\
\overline{\mathsf{Positive}} &= \overline{\mathsf{Element}} \\
\overline{\mathsf{Negative}} &= \overline{\mathsf{Element}}
\end{aligned}$$

Construction and deconstruction of negative conditions are quite similar to the case of positive conditions.

2.4.4 Functional programs

Obviously, first-order functions can be regarded as an instance in much the same way as algebraic specifications, but it should be pointed out that the normalization can be regarded as a semantics-preserving transformation of first-order functions, whereas the normalization of algebraic specifications requires them to be mapped to the data types for meta-programming. Considering Figure 2.5 as a functional program, the corresponding normalized functional program is shown in Figure 2.8.

```
...
execute(skip, MEM) = MEM                                    [skip]

execute(concat(STM_1, STM_2), MEM_0) =
 Let MEM_1 = execute(STM_1, MEM_0) In
  Let MEM_2 = execute(STM_2, MEM_1) In                      [concat]
   MEM_2

execute(assign(ID, EXP), MEM_0) =
 Let VAL = evaluate(EXP, MEM_1) In
  Let MEM_1 = update(MEM_0, ID, VAL) In                     [assign]
   MEM_1
 ...
evaluate(var(ID, MEM) =
 Let VAL = apply(MEM, ID) In                                [var]
  VAL
 ...
```

Figure 2.8: Normalized functional program obtained from Figure 2.5

Higher-order functional programming requires a substantial adaptation of the framework which is a subject of our current work.

2.4.5 Grammars of Syntactical Functions

Here we comment on an instance for Grammars of Syntactical Functions (GSFs) [Rie91, RMD83, Rie72, Rie79] which are a kind of attribute grammars closely related to logic programs; refer to Section A.3 for a short presentation of GSFs. GSFs are also similar to the more recent formalism RAG [CD84, DM85, DM93]. Ordinary Knuthian AGs [Alb91] must be treated differently; refer to Subsection 2.4.6.

A GSF consists of

- a *GSF schema* corresponding to a set of so-called *GSF rules*, which can be regarded as parameterized context-free rules with relational formulae on the parameters and
- a *GSF interpretation* corresponding to carriers for the parameters and relations for the interpretation of the relational symbols.

We are rather interested in GSF rules than GSF interpretations. AGs (including GSF schemata) are usually *open* in the sense that specifications use semantic function *symbols* (corresponding to relational symbols in GSF rules). The actual interpretation of the symbols is beyond the scope of the AG formalism.

Figure 2.9 shows a typical application of AGs, that is, a frontend specification for an imperative language. The specification is intended to model type checking and the construction of an abstract syntactical representation. Actually, the language[5] is the same as in the interpreter examples, e.g. in Figure 1.2. The parameters of sort ST model the symbol table to be propagated for type checking. The parameters of sort T are bound to types of variables and expressions. The parameters of sort EXP and STM should be regarded as placeholders for abstract syntactical representations for expressions and statements respectively. The relational formulae with the prefix $\&_{ast}$ model construction of abstract syntactical representations, whereas the relational formulae with the prefix $\&_{static}$ concern type checking. Note that the actual interpretation of the relational symbols is beyond the scope of this specification.

Let us consider GSFs as an instance of the general framework. The type $\overline{\mathsf{Rule}}$ coincides with GSF rules and the type $\overline{\mathsf{Premise}}$ has to be extended to cope with relational formulae. There are no special problems with well-formedness and well-typedness. Well-definedness can be regarded as non-circularity + reducedness. As far as structural definitions are concerned the following refinement of the data types is assumed:

$$\begin{array}{rcl}
\overline{\mathsf{Premise}} & = & \overline{\mathsf{Element}} \oplus \overline{\mathsf{Computation}} \\
\overline{\mathsf{Computation}} & = & \mathsf{Operation} \otimes \overline{\mathsf{Parameter}}^{\star} \otimes \overline{\mathsf{Parameter}}^{\star} \\
\mathsf{Operation} & = & \mathsf{Prefix} \otimes \mathsf{Id} \\
\mathsf{Symbol} & = & \mathsf{Name} \oplus \mathsf{Operation}
\end{array}$$

The data type $\mathsf{Computation}$ models relational formulae, whereas $\mathsf{Operation}$ models symbols used in relational formulae (for short: relational symbols) in similarity to names of elements. In earlier presentations of the GSF formalism [Rie91, RMD83, Rie72, Rie79] two classes of

[5]Note that the underlying context-free grammar reflects a rather abstract syntax. Refer, for example, to Figure 3.31 for the concrete syntax of the *if*-construct obtained by a certain refinement of the rule [if].

$$\begin{array}{lll}
\mathit{program} \to (\mathsf{PRO}) & : & \&_{static}\ \mathit{init} \to (\mathsf{ST}_0), \\
 & & \mathit{declarations}(\mathsf{ST}_0) \to (\mathsf{ST}_1), \\
 & & \mathit{statements}(\mathsf{ST}_1) \to (\mathsf{STM}), \\
 & & \&_{ast}\ \mathit{prog}(\mathsf{STM}) \to (\mathsf{PRO}). \qquad [\mathsf{prog}] \\
\mathit{declarations}(\mathsf{ST}_0) \to (\mathsf{ST}_2) & : & \mathit{declaration}(\mathsf{ST}_0) \to (\mathsf{ST}_1), \\
 & & \mathit{declarations}(\mathsf{ST}_1) \to (\mathsf{ST}_2). \qquad [\mathsf{decs}] \\
\mathit{declarations}(\mathsf{ST}) \to (\mathsf{ST}). & & \qquad [\mathsf{nodecs}] \\
\mathit{declaration}(\mathsf{ST}_0) \to (\mathsf{ST}_1) & : & \mathit{identifier} \to (\mathsf{ID}), \\
 & & \mathit{type} \to (\mathsf{T}), \\
 & & \&_{static}\ \mathit{add}(\mathsf{ST}_0, \mathsf{ID}, \mathsf{T}) \to (\mathsf{ST}_1). \qquad [\mathsf{dec}] \\
\ldots & & \\
\mathit{statements}(\mathsf{ST}) \to (\mathsf{STM}) & : & \mathit{statement}(\mathsf{ST}) \to (\mathsf{STM}_1), \\
 & & \mathit{statements}(\mathsf{ST}) \to (\mathsf{STM}_2), \\
 & & \&_{ast}\ \mathit{concat}(\mathsf{STM}_1, \mathsf{STM}_2) \to (\mathsf{STM}). \qquad [\mathsf{concat}] \\
\mathit{statements}(\mathsf{ST}) \to (\mathsf{STM}) & : & \&_{ast}\ \mathit{skip} \to (\mathsf{STM}). \qquad [\mathsf{skip}] \\
\mathit{statement}(\mathsf{ST}) \to (\mathsf{STM}) & : & \mathit{identifier} \to (\mathsf{ID}), \\
 & & \&_{static}\ \mathit{lookup}(\mathsf{ST}, \mathsf{ID}) \to (\mathsf{T}_{LHS}), \\
 & & \mathit{expression}(\mathsf{ST}) \to (\mathsf{T}_{RHS}, \mathsf{EXP}), \\
 & & \&_{static}\ \mathit{assignable}(\mathsf{T}_{LHS}, \mathsf{T}_{RHS}), \\
 & & \&_{ast}\ \mathit{assign}(\mathsf{ID}, \mathsf{EXP}) \to \mathsf{STM}. \qquad [\mathsf{assign}] \\
\mathit{statement}(\mathsf{ST}) \to (\mathsf{STM}) & : & \mathit{expression}(\mathsf{ST}) \to (\mathsf{T}, \mathsf{EXP}), \\
 & & \&_{static}\ \mathit{isBool}(\mathsf{T}), \\
 & & \mathit{statements}(\mathsf{ST}) \to (\mathsf{STM}_1), \\
 & & \mathit{statements}(\mathsf{ST}) \to (\mathsf{STM}_2), \\
 & & \&_{ast}\ \mathit{if}(\mathsf{EXP}, \mathsf{STM}_1, \mathsf{STM}_2) \to \mathsf{STM}. \qquad [\mathsf{if}] \\
\ldots & & \\
\mathit{expression}(\mathsf{ST}) \to (\mathsf{T}, \mathsf{EXP}) & : & \mathit{identifier} \to (\mathsf{ID}), \\
 & & \&_{static}\ \mathit{lookup}(\mathsf{ST}, \mathsf{ID}) \to (\mathsf{T}), \\
 & & \&_{ast}\ \mathit{var}(\mathsf{ID}) \to \mathsf{EXP}. \qquad [\mathsf{var}] \\
\ldots & &
\end{array}$$

Figure 2.9: A frontend for a simple imperative language

relational symbols were distinguished for pragmatic reasons. Here it is assumed that there can be an arbitrary number of classes of relational symbols like in $\Lambda\Delta_\Lambda$ [HLR97, LRH96, RL93, Rie92], where relational symbols are prefixed by a kind of module qualifier. Here Prefix is some countable set of prefixes. To avoid confusion with other kinds of identifiers, we will use "&" followed by ordinary identifiers to denote prefixes. "&" denotes the empty prefix which is used in examples if there is no need for different prefixes. Since the kind of symbol is the only difference between Element and Computation, a single constructor for both kinds of premises can be used:

$$\textbf{Premise From } \_\ \_ \to \_ \quad : \quad \mathsf{Symbol} \times \mathsf{Parameter}^\star \times \mathsf{Parameter}^\star \to \mathsf{Premise}$$

To retain a simple notation we assume implicit coercions between Element and Premise. Concerning the deconstruction of premises, some relations in similarity to the deconstruction of elements are needed. Moreover relations to test for different kinds of premises and symbols are needed:

$$
\begin{aligned}
\textbf{Element}?\ \_ &: \mathsf{Premise} \rightarrow \mathsf{Boolean}\\
\textbf{Computation}?\ \_ &: \mathsf{Premise} \rightarrow \mathsf{Boolean}\\
\textbf{Symbol Of}\ \_ &: \mathsf{Premise} \rightarrow \mathsf{Symbol}\\
\textbf{Parameters}\ \_\ \textbf{Of}\ \_ &: \mathsf{Io} \times \mathsf{Premise} \rightarrow \mathsf{Parameter}^{\star}\\
\textbf{Name}?\ \_ &: \mathsf{Symbol} \rightarrow \mathsf{Boolean}\\
\textbf{Operation}?\ \_ &: \mathsf{Symbol} \rightarrow \mathsf{Boolean}
\end{aligned}
$$

Refer to Section B.7 for the corresponding inference rules. For completeness, an operation for constructing relational symbols from tags (similar to **Name From**) is assumed. Moreover, a generalized variant of the operation **Tag From** to coerce a name into a tag coping with symbols instead of names is assumed. Finally, an operation for generating fresh relational symbols is introduced.

$$
\begin{aligned}
\textbf{Operation From}\ \_\ \_ &: \mathsf{Prefix} \times \mathsf{Tag} \rightarrow \mathsf{Operation}\\
\textbf{Tag From}\ \_ &: \mathsf{Symbol} \rightarrow \mathsf{Tag}\\
\mathcal{NEW}_{\mathsf{Operation}} &: \mathcal{P}(\mathsf{Operation}) \rightarrow \mathsf{Operation}
\end{aligned}
$$

It follows from the above declarations that we consider sequences of parameterized grammar symbols and relational symbols as proper RHSs of GSF rules. In contrast, taking the purely declarative point of view, GSF rules are parameterized context-free rules with computations associated with the parameters. The order of the relational formulae among each other and also relative to the parameterized grammar symbols is declaratively meaningless. However, the actual order can be used to control meta-programs. Refer also to Subsection 2.2.1 for the reasons why we prefer sequences of premises instead of sets of premises.

It is interesting to notice that a distinction between grammar symbols and relational symbols and the notion of the underlying context-free grammar of a GSF schema leads us directly to a corresponding notion of composition. GSF schemata with the same underlying context-free grammar can be composed rule-wise by superimposing grammar symbols, concatenating parameters of superimposed grammar symbols and taking over relational formulae. This technique which we call *superimposition* is presented more in detail in §3.3.3.1; refer also to [Läm97] where we have suggested a variant of this technique in the context of logic programming.

2.4.6 Knuthian Attribute Grammars

We continue the discussion on instantiating the general framework for attribute grammars by commenting on the *Knuthian* style of attribute grammars. (Knuthian) AGs require some encoding, that is, semantic rules have to be modelled by another form of premises. We describe the instance for Knuthian AGs by defining a transformation from Knuthian AGs to GSF schemata. The transformation is performed for every syntactical rule of the AG as follows:

- Inherited (synthesized) attributes correspond to the input (output) positions of grammar symbols. For every attribute of every symbol in a syntactical rule, we introduce a corresponding variable and use it as a parameter on the corresponding parameter position.
- Semantic copy rules can be compiled by identifying parameters accordingly.
- A semantic equation e of the form $a_{e,0} := f_e(a_{e,1}, \ldots, a_{e,n_e})$ is transformed into a corresponding relational formula & $f_e(v_{e,1}, \ldots, v_{e,n_e}) \to v_{e,0}$, where the $v_{e,i}$ are the variables associated with the attributes $a_{e,i}$.
- The parameterized grammar symbols and the derived relational formulae representing the semantic equations are combined in a rule according to the data type Rule.

Refer to Section A.3 for an example of a Knuthian AG and the associated GSF schema.

2.5 Completion to an applicative calculus

The complete calculus is obtained by augmenting a typed λ-calculus with the data types for meta-programming. Specification features for dealing with compound domains and with error specification are added. Reusability of meta-programs is supported by modularity particularly at the meta-level. Altogether, we obtain an applicative calculus for typed, modular and strict functional meta-programs.

2.5.1 Simple λ-calculus-like constructs

Typed λ-abstraction is denoted as usual by $\lambda x : \tau \, . \, e$. Functional application is denoted by f **On** p, whereas $f \circ g$ denotes functional composition, where

$$f \circ g \ \textbf{On}\ x = f \ \textbf{On}\ (g \ \textbf{On}\ x).$$

The conditional is denoted by $b \to e_1, e_2$, where the compound expression evaluates to the value of e_1 (resp. e_2) if the condition b can be evaluated to **True** (resp. **False**). The conditional is the only construct which is not strict w.r.t. $\perp$ (i.e. divergence).

Let $x = e$ **In** e' binds x to e during the evaluation of e'. Free occurrences of x in e are not bound to e. **Letrec** $f : \tau = e$ **In** e' binds f being of type τ to e during the evaluation of e'. Free occurrences of f in e refer to e as well. The **Letrec**-construct is the only possibility to express general recursion. However, the iteration constructs introduced below are strongly recommended because termination definitely holds for them.

Divergence, i.e. non-terminating evaluations are denoted by $\perp$. We say that *exp is defined* if the value of *exp* is not equal to $\perp$. Whenever variables are quantified, $\perp$ is not considered as a proper value, i.e. $\forall rs \in \mathsf{Rules}$ does actually mean $\forall rs \in (\mathsf{Rules} \setminus \{\perp\})$. Note that both of these directives need to be updated in Subsection 2.5.2.

2.5.2 Error specification

Simple features for error specification are required for two reasons. First, if we embed the data types for meta-programming into the applicative calculus, we need a standard way for representing the undefined value, since several basic operations are partial. Second, meta-programs (transformations, analyses, etc.) are quite often partial, since the parameters have to satisfy certain preconditions. Thus, a specification feature is needed to propagate an error. Errors should be handled strictly, i.e. once an error occurred during the evaluation of some part of an expression, the entire evaluation must fail.

Consequently, a special error element $\top$ (pronounced as *top*[6]) is assumed to be an element of any type. Instead of including $\top$ as a construct, the partial conditional is added:

$$b \circ\!\!\rightarrow e$$

The above expression is evaluated as follows. The value of the expression is the value of e, if the value of b is **True**. Otherwise, the value of the expression is $\top$. Thus, the exact meaning of $b \circ\!\!\rightarrow e$ can be understood as $b \ \rightarrow \ e, \top$. Note that the conditional is also not strict w.r.t. $\top$.

The notion of definedness needs to be updated as follows: *exp is defined* if its value is neither $\top$ nor $\perp$. Whenever variables are quantified, $\top$ is not considered as a proper value either, i.e. $\forall rs \in \mathsf{Rules}$ does actually mean $\forall rs \in (\mathsf{Rules} \setminus \{\perp, \top\})$.

Note that the above approach to error specification in meta-programs is quite minimal. We can say a transformation fails if it returns $\top$. Unfortunately, it can mean almost anything, if an expression is evaluated to $\top$. It can mean, for example, that

- the construction of a fragment failed or
- well-definedness of the final target program does not hold or
- some selection made in the meta-program was not correct or
- at some stage well-typedness was not satisfied.

Errors are notified by the partial conditional. A more realistic calculus should probably support a more sensible error notification, e.g. meaningful error messages. It is important that our simple approach to error specification ensures strictness, that is to say failures cannot be "overlooked".

[6]in the sense of a top element in a complete lattice; refer e.g. to [Sto77, page 81]

2.5.3 Embedding data types for meta-programming

All the basic data types for meta-programming become proper types of the calculus. Thus, the names of these types $\mathsf{Program}$, Rules, etc. can be regarded as basic type expressions, whereas the operations on the data types can be regarded as predefined functions in the calculus. Instead of considering partial functions in the calculus, the value of the application of $f(v_1, \ldots, v_n)$ is assumed to be $\top$, if the application of the basic operation f is not defined.

The operations $\mathcal{NEW}_{\mathsf{Variable}}$ $\mathcal{NEW}_{\mathsf{Name}}$ and $\mathcal{NEW}_{\mathsf{Operation}}$ for the generation of fresh variables and names are incorporated in the resulting calculus in another way. We assume impure variants (similarly to reference allocation in SML [MTH90]):

$$\begin{array}{rcl}
\textbf{New Variable Of Sort \_} & : & \mathsf{Sort} \to \mathsf{Variable} \\
\textbf{New Name} & : & \to \mathsf{Name} \\
\textbf{New Operation} & : & \to \mathsf{Operation}
\end{array}$$

Ordinary sequences on Rule are not permitted. We declare that $\langle r_1, \ldots, r_n \rangle$ is an abbreviation of **Rules From** $\langle r_1, \ldots, r_n \rangle$. Thereby, coercions from $\mathsf{Rule}^\star$ to Rules can be omitted. The type Rules is coerced to $\mathsf{Program}$ at the top level of a meta-program. Without further declarations it is assumed that all required symbols are imported, all defined symbols are exported and no axiom is defined. There are clauses to override these defaults, e.g. an **Axiom Is** clause to define an axiom.

2.5.4 Domain constructors

Domain constructors for products, domains of sequences, power sets and domains with optional values are added.

2.5.4.1 Tuples

Let τ_1, ..., τ_n be proper type expressions. $\tau_1 \otimes \cdots \otimes \tau_n$ denotes the type of all tupels with the i-th projection of type τ_i for $i = 1, \ldots, n$. The expression $\langle exp_1, \ \ldots, \ exp_n \rangle$ denotes the common construction of tuples from expressions exp_1, ..., exp_n. The **Let**-construct is generalized to cope with tuples, i.e. **Let** $\langle x_1, \ldots, x_n \rangle = e$ **In** e' binds the projections of e to the x_i in e'. Typed λ-abstraction is generalized as well, i.e. the projections of a tuple can be bound to several λ-variables in the following way: $\lambda \langle x_1, \ldots, x_n \rangle : \tau_1 \otimes \cdots \otimes \tau_n . e$.

2.5.4.2 Sequences

Let τ be a proper type expression. $\tau^\star$ denotes the type of all sequences of elements of type τ. The expression $\langle exp_1, \ \ldots, \ exp_n \rangle$ denotes the common construction of sequences from expressions $exp_1 : \tau$, ..., $exp_n : \tau$. The empty sequence is denoted by $\langle\rangle$. Figure 2.10 enumerates all the basic operations on sequences, whereas Figure 2.11 establishes some recursion / iteration schemata well-known in higher-order functional programming. It is assummed that these schemata are applicable to Rules, Sigma and $\mathsf{Substitution}$ as well.

Profile			**Explanation**
Head Of _	:	$\tau^\star \to \tau$	head of a sequence
Tail Of _	:	$\tau^\star \to \tau^\star$	tail of a sequence
Nil? _	:	$\tau^\star \to$ Boolean	test for the empty sequence
_ ++ _	:	$\tau^\star \times \tau^\star \to \tau^\star$	concatenation of sequences
# _	:	$\tau^\star \to \mathcal{N}_0$	length of the sequence
Reverse _	:	$\tau^\star \to \tau^\star$	to reverse a sequence

Figure 2.10: Operations on sequences

Map f **List** $\langle x_1, x_2, \ldots, x_{n-1}, x_n \rangle$ $\Vert$ $\langle f(x_1), f(x_2), \ldots, f(x_{n-1}), f(x_n) \rangle$
Fold Left $\odot$ **Neutral** e **List** $\langle x_1, x_2, \ldots, x_{n-1}, x_n \rangle$ $\Vert$ $(\cdots((e \odot x_1) \odot x_2) \odot \cdots \odot x_{n-1}) \odot x_n$
Fold Right $\odot$ **Neutral** e **List** $\langle x_1, x_2, \ldots, x_{n-1}, x_n \rangle$ $\Vert$ $x_1 \odot (x_2 \odot \cdots \odot (x_{n-1} \odot (x_n \odot e)) \cdots)$

Figure 2.11: Iteration on sequences

2.5.4.3 Sets

Let τ be a type expression. Then $\mathcal{P}(\tau)$ denotes the type of subsets of τ. Here τ must be a non-functional type because we need equality on τ for obvious technical reasons. The empty set is denoted by $\emptyset$. Let be $exp_1 : \tau$, ..., $exp_n : \tau$. The expression $\{exp_1, \ldots, exp_n\}$ denotes the set of values of the expressions exp_i for $i = 1, \ldots, n$.

Each set of type $\mathcal{P}(\tau)$ is a sequence of type $\tau^\star$, i.e. all operations on sequences directly apply to sets as well. In particular, for iteration on sets, it is important to know that the order in $\{exp_1, \ldots, exp_n\}$ is transferred into the resulting set. Figure 2.12 enumerates all the additional basic operations on sets.

Each sequence $\langle v_1, \ldots, v_n \rangle$ of type $\tau^\star$ is automatically coerced to the corresponding set $\{v_1\} \cup \cdots \cup \{v_n\}$ of type $\mathcal{P}(\tau)$ if it serves as an actual parameter for a formal parameter of type $\mathcal{P}(\tau)$.

Profile			**Explanation**
_ $\cup$ _	:	$\mathcal{P}(\tau) \times \mathcal{P}(\tau) \to \mathcal{P}(\tau)$	union of sets
_ $\cap$ _	:	$\mathcal{P}(\tau) \times \mathcal{P}(\tau) \to \mathcal{P}(\tau)$	intersection of sets
_ $\setminus$ _	:	$\mathcal{P}(\tau) \times \mathcal{P}(\tau) \to \mathcal{P}(\tau)$	difference of sets
_ $\subseteq$ _	:	$\mathcal{P}(\tau) \times \mathcal{P}(\tau) \to$ Boolean	test of proper subset
_ $\in$ _	:	$\tau \times \mathcal{P}(\tau) \to$ Boolean	membership test

Figure 2.12: Operations on sets

Another iteration construct is frequently needed:

$$\begin{array}{c}\textbf{Map Union } f \textbf{ List } \langle x_1, x_2, \ldots, x_{n-1}, x_n\rangle \\ \| \\ f(x_1) \cup f(x_2) \cup \ldots \cup f(x_{n-1}) \cup f(x_n)\end{array}$$

Note also that all the iteration constructs are written as **Map _ Set _**, **Fold Left _ Neutral _ Set _**, **Fold Right _ Neutral _ Set _** and **Map Union _ Set _** when it should be pointed out that iteration is performed on a set instead of a list.

2.5.4.4 The *Maybe* type constructor

For every type τ, there is also the *maybe type* $\tau?$. Every element of τ is an element of $\tau?$. Additionally, a special element ? is added to $\tau?$.

Maybe types are useful in meta-programming for:

- optional arguments, where functions can observe ? providing an indication of a missing argument,
- return values, where ? is returned as an indication of a missing meaningful result. To return $\top$ instead of ? is not appropriate, since we cannot test for $\top$ due to strictness.

Example 2.5.1
Consider the following expression:

```
λ s : Symbol . λ t : Sigma .
 Fold Left
  λ p0 : Profile? . λ p : Profile . Name Of p = s → p, p0
 Neutral ? List t.
```

It defines the operator **Profile Of _ In _** : Symbol × Sigma → Profile? looking up the profile of a symbol s in a given signature t. There are scenarios in meta-programs where a symbol does not need to have a type at all in a given signature. The lookup should not fail, since the whole surrounding application would fail because of strictness. Thus, the neutral element ? will be returned if no profile for s is contained in t. ♢

2.5.5 Modules

To facilitate meta-programming a certain module concept should be supported. Here a module is regarded simply as a separate specification unit with its own semantics, i.e. in a technical sense a kind of module allowing for separate compilation. Although information hiding is a central notion in many module systems, it is almost ignored as far as it concerns this work.

There are two kinds of modules:

- modules at the target-level and

- modules at the meta-level.

Modularity at the target-level is useful for obvious reasons, even without considering meta-programming at all. For meta-programming, target-level modules provide the central operands on which transformations and compositions are performed. The execution of a meta-program results in a new target-level program. Modules at the meta-level are useful in order to implement reusable program transformations and to represent central parts of a problem rather at the meta-level (in a more abstract manner) than at the target-level.

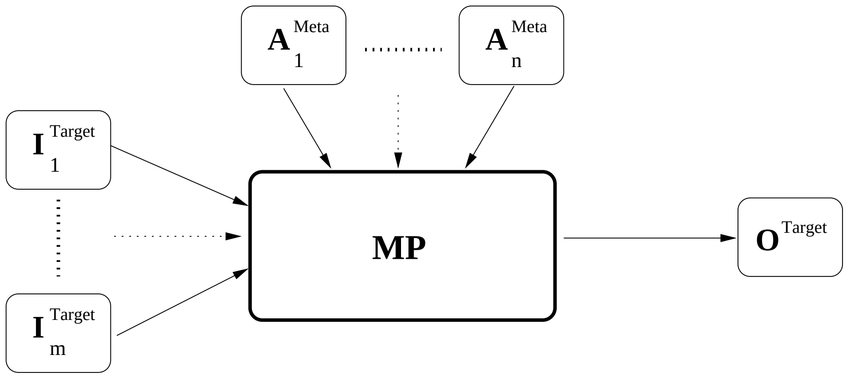

$\mathbf{A}_1^{\text{Meta}}$
$\mathbf{A}_n^{\text{Meta}}$
$\mathbf{I}_1^{\text{Target}}$
$\mathbf{I}_m^{\text{Target}}$
MP
$\mathbf{O}^{\text{Target}}$

Figure 2.13: Modular meta-programming

Figure 2.13 shows the scenario of modular meta-programming. There is a central meta-program **MP** which is applied to target-level modules $\mathbf{I}_i^{Target}$ serving as inputs for a program transformation / composition. The execution of **MP** produces the target-level module $\mathbf{O}^{Target}$ as output. The meta-level modules $\mathbf{A}_j^{Meta}$ are assumed to provide reusable program transformations, generic fragments etc.

To support this kind of modularity, module identifiers are permitted as a form of expression in meta-programs. If the module identifier refers to a target program, the expression is of type Program. If the module identifier refers to a meta-program, the type of the meta-program is the type of the underlying expression. It is also useful to support a kind of abstraction in meta-program modules so that all the abstractions can be "imported" in another program. Without such a facility, a meta-program module only "exports" a single expression.

Example 2.5.2
Recall our attempt to modular semantics description as outlined in the introduction; refer to Section 1.2. There we have seen that target program and meta-program modules are very useful during the composition of semantics descriptions fragments. Here we want to comment on a more complex problem. Consider, for example, a simple imperative language with simple statement forms for assignment, selection (*if*), iteration (*while*), and compound statements (statement sequence), I/O, basic data types.

We can isolate target program modules (specifying the semantics of some constructs) like the following:

- variables as expressions and in assignments,
- *if*-statements,
- *while*-statements,
- statement sequences,
- I/O constructs,
- the overall structure of a program and the declaration part and
- constants, basic operations, simple type expressions.

We assume that the above modules abstract from any semantic issue which is not relevant for the actual construct, i.e. they are "minimal" in the sense of Section 1.2. Most modules, for example, abstract from I/O. Many modules abstract from the actual memory model, i.e. whether a flat model or a two-level model is used and whether side-effects might possibly be involved in expression evaluation etc.

We need transformations to adapt the above fragments accordingly. The actual memory model can be manifested by a corresponding transformation, for example. More in detail, the following meta-program modules are involved:

- the transformation to establish memory propagation,
- the transformation to propagate the remaining input,
- the transformation to accumulate the output.

The composition of the semantics description can be represented by a meta-program applying the above transformations to the corresponding target program modules and merging the intermediate results. We will comment in more detail on such a composition in Section 3.5 on lifting. Refer also to Section D.1 for the complete source code concerning the composition of a frontend specification and an interpreter definition (dynamic semantics) for a language like the one above. $\diamondsuit$

2.6 Properties of meta-programs

By characterizing transformations, we are looking for classes of transformations satisfying certain useful or important properties. We can be interested, for example, in the question whether a transformation is total or we can ask whether a transformation preserves certain properties of the input program such as the type or the "skeleton" obtained from rules by abstracting from parameterization.

2.6.1 Skeletons and their preservation

An important property of many meta-programs is skeleton preservation, where the notion of a skeleton wants to grasp the overall structure of some rules $rs \in \mathsf{Rules}$ abstracting from computational behaviour, similar to the underlying CFG of an AG. The purpose of this subsection is to formalize skeletons and to define skeleton preservation.

Two further data types are needed, that is to say $\mathsf{Skeleton}$ for an abstraction from Rules and Shape for an abstraction from Rule, with the following structural definitions:

$$\begin{aligned} \overline{\mathsf{Skeleton}} &= \mathsf{Shape}^\star \\ \mathsf{Shape} &= \mathsf{Tag} \otimes \mathsf{Name} \otimes \mathsf{Name}^\star \end{aligned}$$

There are no constraints on Shape concerning $\mathcal{WF}$, $\mathcal{WT}$ and $\mathcal{WD}$. The only constraint for $\mathsf{Skeleton}$ concerns $\mathcal{WF}$, namely tags have to be unique. The *skeleton* of some rules $rs \in \mathsf{Rules}$ is obtained by discarding the parameterization and all premises which are not elements in rs; refer to Definition 2.6.1.

Definition 2.6.1
Consider the following definition of a function **Skeleton Of _** : $\mathsf{Rules} \rightarrow \mathsf{Skeleton}$:

```
λ rs : Rules .
 Map
  λ r : Rule .
   ⟨Tag Of r, Name Of Conclusion Of r,
    Fold Left
     λ rhs : Name* . λ e : Element . rhs ++ ( Element? e → ⟨Name Of e⟩, ⟨ ⟩)
    Neutral ⟨ ⟩ List Premises Of r
   ⟩
 List rs.
```

Let be $rs \in \mathsf{Rules}$, sk is the *skeleton* of rs, if $sk =$ **Skeleton Of** rs. ♢

As Definition 2.6.1 points out, other premises than elements (data type $\mathsf{Element}$) are not included into the skeleton. To point out this role of elements as form of premises, we sometimes use the term *skeleton elements*. Recall that there are other forms of premises with the same structure as $\mathsf{Element}$, e.g. computations (relational formulae) in GSF schemata. The notion of a skeleton becomes a more vital abstraction device if there are other forms of premises than (skeleton) elements.

Example 2.6.1
Consider Figure 2.14 showing the skeleton of the GSF schema from Figure 2.9 serving as a frontend specification for a simple imperative language. The skeleton can be regarded as the underlying context-free grammar of the GSF schema. ♢

Definition 2.6.2
A transformation $t \in \mathsf{Trafo}$ is *skeleton-preserving* if $\forall rs \in \mathsf{Rules}$:

$$rs' \text{ is defined } \Rightarrow \textbf{Skeleton Of } rs = \textbf{Skeleton Of } rs',$$

where $rs' = t$ **On** rs. ♢

Consider the following Proposition 2.6.1 as a sort of an example.

```
program      : declarations, statements.                          [prog]
declarations : declaration, declarations.                         [decs]
declarations : .                                                  [nodecs]
declaration  : identifier, type.                                  [dec]
 ...
statements   : statement, statements.                             [concat]
statements   : .                                                  [skip]
statement    : identifier, expression.                            [assign]
statement    : expression, statements, statements.                [if]
 ...
expression   : identifier.                                        [var]
 ...
```

Figure 2.14: Skeleton of the frontend specification from Figure 2.9

Proposition 2.6.1
$\forall \sigma \in \mathsf{Sort}$ the transformation **Left To Right** σ (refer to Figure 1.5) is skeleton-preserving.
◊

Proofs of such statements can be based on a simple equational reasoning in our framework. Proposition 2.6.1 can be proved by showing that the structure of the definition of a skeleton in Definition 2.6.1 (i.e. the recursion schema) is contained in the definition of the operation **Left To Right** (refer to Figure 1.5) and that the differences are invariant for the resulting skeleton what can be derived from simple properties of element construction and deconstruction.

2.6.2 Totality

Transformations are potentially partial because of the possibility that an application of a basic operation or a partial conditional fails. However, we can show that some transformations are total according to the following definition.

Definition 2.6.3
A transformation $t \in \mathsf{Trafo}$ is:

1. *total* if t **On** rs is defined $\forall rs \in \mathsf{Rules}$,
2. *α-total* if t **On** rs is defined $\forall rs \in \alpha$, where $\alpha \subseteq \mathsf{Rules}$.

◊

Certain properties often do not hold for all $rs \in \mathsf{Rules}$, but only for a restricted subset α. In Definition 2.6.3, for example, *α-total* transformations were introduced, i.e. transformations whose result is defined at least for all $rs \in \alpha$. In the sequel, we sometimes consider α-properties. If the α is omitted, it means that the property holds for all $rs \in \mathsf{Rules}$. Obviously, a property which holds for all $rs \in \mathsf{Rules}$ is more comfortable to use. For an α-property, we always have to make sure that a given rs belongs to α to derive the property. This makes indeed sense for α-totality, for example because the α simply specifies where the transformation is defined (i.e. applicable).

2.6.3 Preservation and recovery of well-definedness

Definition 2.6.4
A transformation $t \in \mathsf{Trafo}$ is *α-$\mathcal{WD}$-preserving* if $\forall rs \in \alpha \subseteq \mathsf{Rules}$:

$$\mathcal{WD}_{\mathsf{Program}}(rs) \wedge rs' \text{ is defined} \Rightarrow \mathcal{WD}_{\mathsf{Program}}(rs'),$$

where $rs' = t$ **On** rs. $\diamondsuit$

Note that there is no sense in defining α-$\mathcal{WF}$-preserving or α-$\mathcal{WT}$-preserving transformations, because results of transformations are well-formed and well-typed by definition. Thus, these preservation properties are somehow captured by α-totality.

α-$\mathcal{WD}$-preservation for $\alpha \subset \mathsf{Rules}$ is not very instructive in many cases because often we are not interested in the α whose elements can be transformed such that $\mathcal{WD}$ is preserved, but we rather look for a suitable description how $\mathcal{WD}$ can be restored. Adding an input position for a symbol s, for example, input positions in premises with s as symbol will not be defined, but the violation of $\mathcal{WD}$ is restricted to these positions and it could be eliminated in a straightforward manner. The idea of recovery of $\mathcal{WD}$ is captured by the following definition; refer to Proposition 3.3.1 for an application of this concept.

Definition 2.6.5
A transformation $t \in \mathsf{Trafo}$ is *α-$\mathcal{WD}$-recoverable* by another transformation $t' \in \mathsf{Trafo}$ if $\forall rs \in \alpha \subseteq \mathsf{Rules}$:

$$\mathcal{WD}_{\mathsf{Program}}(rs) \wedge rs' \text{ is defined} \Rightarrow \mathcal{WD}_{\mathsf{Program}}(rs'),$$

where $rs' = (t' \circ t)$ **On** rs. $\diamondsuit$

2.6.4 Type preservation

Definition 2.6.6
A transformation $t \in \mathsf{Trafo}$ is:

1. *α-type-preserving* if **Sigma Of** $rs \sqcup$ **Sigma Of** rs' is defined;
2. *α-type-monoton increasing* if **Sigma Of** $rs \leq$ **Sigma Of** rs';
3. *α-type-monoton decreasing* if **Sigma Of** $rs \geq$ **Sigma Of** rs';
4. *α-strongly type-preserving* if it is α-type-monoton increasing and decreasing

$\forall rs \in \alpha \subseteq \mathsf{Rules}$ such that rs' is defined, where $rs' = t$ **On** rs. $\diamondsuit$

Proposition 2.6.2
$\forall \sigma \in \mathsf{Sort}$ the transformation **Left To Right** σ (refer to Figure 1.5) is strongly type-preserving.

$\diamondsuit$

2.6.5 Type extension

By taking a transformational approach to program synthesis, many transformations are likely to change the type of the input program, i.e. none of the criteria in Definition 2.6.6 applies, but we are still looking for useful restrictions for the behaviour of transformations. A type-extending transformation is a modest generalization of type-preserving transformation.

Definition 2.6.7
Let be $t \in \mathsf{Trafo}$. $\forall rs \in \alpha \subseteq \mathsf{Rules}$ such that rs' is defined, where $rs' = t$ **On** rs. t is a *type-extending* transformation if the following property holds:

$\forall p' \in$ **Sigma Of** $rs' : \exists p \in$ **Sigma Of** rs :
  p is a projection of p', i.e.
  if $p' = s\ \sigma_1^{\downarrow} \times \cdots \times \sigma_n^{\downarrow} \to \sigma_1^{\uparrow} \times \cdots \sigma_m^{\uparrow}$,
   then $\exists in_1, \ldots, in_q, out_1, \ldots, out_r$:
    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
    $p = s\ \sigma_{in_1}^{\downarrow} \times \cdots \times \sigma_{in_q}^{\downarrow} \to \sigma_{out_1}^{\uparrow} \times \cdots \times \sigma_{out_r}^{\uparrow}$

$\diamondsuit$

The property of type-extension is particularly useful in combination with uniquely sorted symbols. Moreover, by adding skeleton preservation, a quite disciplined transformational style is achieved.

2.6.6 Projections

A projection rs' of rules rs is obtained by deleting some premises and projecting the parameterization of the conclusions and the remaining premises so that the result is equal to rs'.

Definition 2.6.8
Given $rs, rs' \in \mathsf{Rules}$, rs' is a *projection* of rs if

1. $\forall p \in$ **Sigma Of** rs:
   $\exists p' \in$ **Sigma Of** rs' : **Symbol Of** $p =$ **Symbol Of** $p' \Rightarrow$
    p' is a projection of p (refer to Definition 2.6.7).

2. Every rule $[t]\ e_0 \Leftarrow e_1, \ldots, e_n$ in rs can be transformed into a corresponding rule $[t]\ \Pi^{\mathsf{Conclusion}}(e_0) \Leftarrow \Pi^{\mathsf{Premise}}(e_{w_1}), \ldots, \Pi^{\mathsf{Premise}}(e_{w_u})$, where $1 \leq w_1 < \cdots < w_u \leq n$

and $\Pi^{\mathsf{Conclusion}} : \mathsf{Conclusion} \to \mathsf{Conclusion}$ and $\Pi^{\mathsf{Premise}} : \mathsf{Premise} \to \mathsf{Premise}$ are the functions projecting parameters of conclusions and premises according to (1.), so that the resulting rules are equal to rs'.

◇

If rs' and rs have the same skeleton in common, then rs can be regarded as an extension of rs' preserving not only the computational behaviour of rs' but also its skeleton. Note that premises, which are not skeleton elements, can still be deleted by such a projection. Transformations the input program of which is always a projection of the output program (or vice versa), are very attractive transformations. They are more disciplined transformations than type-extending transformations.

```
program  :  &_static init → (ST_0),                                   [prog]
            declarations(ST_0) → (ST_1),
            statements(ST_1).
...
statements(ST)  :  statement(ST),                                     [concat]
                   statements(ST).
statements(ST)  :  .                                                  [skip]
statement(ST)  :  identifier → (ID),                                  [assign]
                  &_static lookup(ST, ID) → (T_LHS),
                  expression(ST) → (T_RHS),
                  &_static assignable(T_LHS, T_RHS).
statement(ST)  :  expression(ST) → (T),                               [if]
                  &_static isBool(T),
                  statements(ST),
                  statements(ST).
...
expression(ST) → (T)  :  identifier → (ID),                           [var]
                         &_static lookup(ST, ID) → (T).
...
```

Figure 2.15: A projection (static semantics) of the specification from Figure 2.9

Example 2.6.2
Figure 2.15 and Figure 2.16 show two different projections of the frontend specification from Figure 2.9. The first projection contains all the parameterization and computational elements which are relevant for the specification of static semantics, whereas the second projection is only concerned with AST construction. Note that the parameterizations of both projections are not "disjoint" because *identifier*'s terminal attribute of sort ID is needed for both, static semantics and AST construction. ◇

The following example should demonstrate how projections are useful to characterize transformations.

$$\begin{array}{lll}
\mathit{program} \to (\mathsf{PRO}) : & \mathit{declarations}, & [\mathsf{prog}] \\
& \mathit{statements} \to (\mathsf{STM}), & \\
& \&_{ast}\ \mathit{prog}(\mathsf{STM}) \to (\mathsf{PRO}). & \\
\ldots & & \\
\mathit{statements} \to (\mathsf{STM}) : & \mathit{statement} \to (\mathsf{STM}_1), & [\mathsf{concat}] \\
& \mathit{statements} \to (\mathsf{STM}_2), & \\
& \&_{ast}\ \mathit{concat}(\mathsf{STM}_1, \mathsf{STM}_2) \to (\mathsf{STM}). & \\
\mathit{statements} \to (\mathsf{STM}) : & \&_{ast}\ \mathit{skip} \to (\mathsf{STM}). & [\mathsf{skip}] \\
\mathit{statement} \to (\mathsf{STM}) : & \mathit{identifier} \to (\mathsf{ID}), & [\mathsf{assign}] \\
& \mathit{expression}(\mathsf{ST}) \to (\mathsf{EXP}), & \\
& \&_{ast}\ \mathit{assign}(\mathsf{ID}, \mathsf{EXP}) \to (\mathsf{STM}). & \\
\mathit{statement} \to (\mathsf{STM}) : & \mathit{expression} \to (\mathsf{EXP}), & [\mathsf{if}] \\
& \mathit{statements}(\mathsf{STM}_1), & \\
& \mathit{statements}(\mathsf{STM}_2), & \\
& \&_{ast}\ \mathit{if}(\mathsf{EXP}, \mathsf{STM}_1, \mathsf{STM}_2) \to (\mathsf{STM}). & \\
\ldots & & \\
\mathit{expression} \to (\mathsf{EXP}) : & \mathit{identifier} \to (\mathsf{ID}), & [\mathsf{var}] \\
& \&_{ast}\ \mathit{var}(\mathsf{ID}) \to (\mathsf{EXP}). & \\
\ldots & &
\end{array}$$

Figure 2.16: Another projection (AST construction) of the specification from Figure 2.9

Example 2.6.3
For the operator **Left To Right** (refer to Figure 1.5), projections support the characterization of the following instructive property. Let be $rs \in \mathsf{Rules}$, $\sigma \in \mathsf{Sort}$. $rs_{/\sigma}$ denotes the projection of rs, where all but the parameter positions of sort σ have been removed, whereas $rs_{/\overline{\sigma}}$ denotes the complementary projection, where all the parameter positions of sort σ have been removed. Given $\sigma \in \mathsf{Sort}, rs \in \mathsf{Rules}, rs' =$ **Left To Right** σ **On** rs, the following properties can be stated:

1. $rs_{/\overline{\sigma}} = rs'_{/\overline{\sigma}}$
2. $rs'_{/\sigma}$ represents the propagation of a data structure from left to right.

$\diamondsuit$

Definition 2.6.9
Given a transformation $t \in \mathsf{Trafo}$, t is

1. *α-extending*, if rs is a projection of rs';
2. *α-contracting*, if rs' is a projection of rs

$\forall rs \in \alpha \subseteq \mathsf{Rules}$ such that rs' is defined, where $rs' = t$ **On** rs. $\diamondsuit$

Proposition 2.6.3
$\forall \sigma \in \mathsf{Sort}$ the transformation **Left To Right** σ is not extending. $\diamondsuit$

2.6.7 Identity

α-totality corresponds to the property where a transformation can only be applied to certain rules. There is a somewhat related property of α-identity concerning the question for which rules a transformation behaves like the identity function.

Definition 2.6.10
A transformation $t \in \mathsf{Trafo}$ is an *α-identity* if $\forall rs \in \alpha \subseteq \mathsf{Rules} : \; rs = t \; \mathbf{On} \; rs$. $\diamondsuit$

α-total transformations makes sense because we can have transformations which are only defined if certain preconditions hold. Dually, α-identity makes sense because we can consider the fact that a transformation t behaves like the identity function on $rs \in \alpha$ as an indication for the property that rs already captures the intended effect of t. We can think of applications, where transformations are actually written in this style, i.e. they should test if their "intended effect" is not manifest yet. If it is manifest, they should silently behave like the identity function. A rather weak characterization of transformations written in this style is given by idempotence.

Definition 2.6.11
A transformation $t \in \mathsf{Trafo}$ is *α-idempotent* if $\forall rs \in \alpha \subseteq \mathsf{Rules} : \; t \; \mathbf{On} \; rs = t \circ t \; \mathbf{On} \; rs$. $\diamondsuit$

Proposition 2.6.4
$\forall \sigma \in \mathsf{Sort}$ the transformation **Left To Right** σ (refer to Figure 1.5) is idempotent. $\diamondsuit$

It is a weak characterization because two subsequent applications of t are not likely to occur. We rather would prefer that transformations behave like the identity function after a single application, even if transformations from a certain class were applied to the intermediate result. The property of being an α-identity can be generalized in the same way.

Definition 2.6.12
Let be $\beta \subseteq \mathsf{Trafo}$.
t is an *α-identity closed under β*, if $\forall t' \in \beta, rs \in \alpha \subseteq \mathsf{Rules} : \; t' \; \mathbf{On} \; rs = t \circ t' \; \mathbf{On} \; rs$.
t is *α-idempotent closed under β*, if $\forall t' \in \beta, rs \in \alpha \subseteq \mathsf{Rules} : \; t' \circ t \; \mathbf{On} \; rs = t \circ t' \circ t \; \mathbf{On} \; rs$.
$\diamondsuit$

Proposition 2.6.5
Let be $\sigma \in \mathsf{Sort}$. The transformation **Left To Right** σ (refer to Figure 1.5) is idempotent closed under all extending transformations (refer to Definition 2.6.9) which do not establish new positions of sort σ. $\diamondsuit$

2.6.8 Structure of transformations

The way a transformation is defined, that is, the structure of the expression, can give some hints of other properties of transformation or it can be instructive for proving properties. Let us consider one example of a structural restriction for transformations. A *local* transformation transforms a sequence of rules rule-wise without "observing" the other rules.

Definition 2.6.13
A transformation $t \in \mathsf{Trafo}$ is *local* if the following property holds:

$$\exists t' \in (\mathsf{Rule} \to \mathsf{Rule}) : \forall rs \in \mathsf{Rules} : t \ \mathbf{On}\ rs = \langle t' \ \mathbf{On}\ r_1, \ldots, t' \ \mathbf{On}\ r_n \rangle,$$

where $rs = \langle r_1, \ldots, r_n \rangle$. t' is called the *rule transformation* of t. ♢

Proposition 2.6.6
For a transformation $t \in \mathsf{Trafo}$ to be *local* it is a sufficient condition if t is defined by a λ-expression of the following form:

$$\lambda rs : \mathsf{Rules}.\ \mathbf{Map}\ t'\ \mathbf{List}\ rs,$$

where rs is not a free variable in t'. Moreover, then t' is the *rule transformation* of t. ♢

2.6.9 Discussion

Transformations are mostly expected to preserve the computational behaviour of a specification. To deal with this requirement in detail, we had to define what the semantics of a specification (at the target-level) is, or how we expect the computational behaviour is to be manifested. We will not consider this topic here in detail, since it is rather difficult to define these notions in the general framework.

Kirschbaum, Sterling et al. have shown in [KSJ93] that program maps—a tool similar to our extending transformations—preserve the computational behaviour of a logic program, if we assume that behaviour is manifested by the SLD computations of the program. Obviously, not all interesting transformations are extending, refer e.g. to Proposition 2.6.3 concerning the operator **Left To Right**.

A common requirement for transformations is general semantics preservation. We try to indicate how this requirement can be stated in the general framework. Given rules $rs \in \mathsf{Rules}$, carriers D_i for the sorts σ_i in **Sigma Of** rs, the semantics of rs is defined by a function with the following profile:

$$[\![\cdot]\!] : \mathsf{Rules} \to (\mathsf{Name} \to \mathcal{U})$$

Here $\mathcal{U}$ denotes some suitable universal domain; refer e.g. to [SHLG94]. The details of the definition of $[\![\cdot]\!]$ depend on the actual framework. More in detail, a name n with profile

$$n : \sigma_1 \times \cdots \times \sigma_m \to \sigma_{m+1} \times \cdots \times \sigma_k$$

is associated with a semantics from the following domain:

$$\mathcal{E}((\mathcal{D}_{\sigma_1} \times \cdots \times \mathcal{D}_{\sigma_m}), (\mathcal{D}_{\sigma_{m+1}} \times \cdots \times \mathcal{D}_{\sigma_k}))$$

Here $\mathcal{E}(D, D')$ denotes some domain construction on D and D'. For an instance of the framework with deterministic semantics, $\mathcal{E}$ will probably correspond to $\_ \to \_$, for example. More modular approaches are possible, e.g. [Bro93, BMPT94] where the semantics of logic programs is based on the intermediate consequence operator. That is particularly useful since we consider potentially incomplete programs.

In the narrow sense, the application of a transformation t to $rs \in \mathsf{Rules}$ is semantics-preserving if $rs' = t$ **On** rs is defined and $[\![rs]\!] = [\![rs']\!]$.

We can speak of an α-semantics-preserving transformation if the above condition holds for all $rs \in \alpha \subseteq \mathsf{Rules}$. Usually, we do not insist on the property that the result of the transformation is defined. In the broader, but still agreeable sense, semantics preservation can be defined modulo some adaptation of $[\![rs]\!]$ and/or $[\![rs']\!]$. Recall for example projections introduced in Subsection 2.6.6. It is also possible to consider a kind of projection on $\mathcal{U}$, i.e. a semantic variant of an contracting transformation which has to be regarded as a syntactical device. In general terms, operations on $\mathcal{U}$ can be used to express the semantics of the output of a transformation as a *refinement* of the semantics of the input. We have not investigated yet that issue more in depth, although there are several approaches in the context of refinement and correctness which should provide a good starting point, e.g. [BR94, BS98, TWW81, Heh93]. Another problem is that there are several transformations, which are inherently not semantics-preserving in any obvious sense or only due to very specific arguments, e.g. **Left To Right**. The question how to cope with such transformations should be studied in future. Since we do not only deal with synthesis and composition but also adaptation, semantics preservation does not seem to be appropriate in all cases.

Chapter 3

The operator suite

In this Chapter, we present *an operator suite for meta-programming on declarative programs.* Besides the general framework which has to be regarded as a basis for this Chapter, the suite is a further major result of the thesis. Different layers of the suite are presented in Section 3.2 - Section 3.4. The presentation culminates with Section 3.5 describing the sophisticated composition technique *lifting.*

3.1 Overview

The operator suite models *schemata for program composition, synthesis and transformation.* The corresponding operators are specified in an instance of the general framework supporting both, natural semantics (refer to Subsection 2.4.1) and GSF schemata (see Subsection 2.4.5). Thereby, our instance supports terms as a kind of compound parameters and there is a distinction between skeleton elements and computational elements (computations for short).

The GSF schema from Figure 2.9 specifying the frontend of a language processor for a simple imperative language will serve as a running example. It will be shown how certain aspects of type checking, AST construction can be synthesized and combined and how intermediate variants can be reused in some cases.

Refer to Figure 1.12 for an illustrative presentation of the structure of the operator suite. We start with Section 3.2 presenting a set of *auxiliary operators* allowing more advanced operators to be specified in a more comprehensive way. Section 3.3 continues with schemata modelling *basic concepts* of synthesis, adaptation and composition. Afterwards, Section 3.4 introduces several more *elaborate schemata* on top of the basic concepts. This chapter culminates with Section 3.5 describing *lifting* which is a new and powerful composition technique. Lifting facilitates the derivation of complete programs from transformations and program fragments. Lifting substantially simplifies the problem of finding a proper structure during nested program composition, synthesis and transformation.

The actual definition of several operators of the suite is included in the text flow if the definition is regarded as instructive or the formal details are required for a discussion of the properties of the operators. The remaining specifications are presented in Appendix C. Note also that some showcases are collected in Appendix D. The actual set of operators including their actual definition is far from being optimal, complete and orthogonal. The operator suite is regarded as a subject of further research. The suite ran through some iterations, where early versions have been covered by [LR96, LR97] and a more recent version has been thoroughly evaluated in [Sta97].

3.2 Auxiliary operators

There is a number of auxiliary operators which can be reused frequently during program transformation and for the definition of basic and elaborate schemata as described later in this Chapter. Regarding the layers of the operator suite as presented in Figure 1.12, the auxiliary operators correspond to the layer on top of the applicative calculus with the embedded basic data types for meta-programming.

First, simple selections, projections, injections and closures over target program fragments are presented in Subsection 3.2.1. Second, a group of renaming operators is considered in Subsection 3.2.2. Third, in Subsection 3.2.3 the simple problem to arrange a sequence of rules according to a given sequence of tags regarded as a reference is addressed. Finally, certain combinators on transformations are discussed in Figure 3.2.4.

3.2.1 Selections, projections, injections and closures

Figure 3.1 enumerates operators for the selection of rules. **Select Tags** *ts* **On** *rs* selects all rules in *rs* with tags in *ts*. **Select Symbols** *ss* **On** *rs* selects all rules in *rs* defining the symbols in *ss*, i.e. the rules with a symbol in *ss* in the conclusion. **Forget Tags** *ts* **On** *rs* selects the complementary set of **Select Tags** *ts* **On** *rs*. **Forget Symbols** *ss* **On** *rs* selects the complementary set of **Select Symbols** *ss* **On** *rs*.

$$\begin{array}{rcl}
\textbf{Select Tags}\ \_ & : & \mathcal{P}(\mathsf{Tag}) \to \mathsf{Trafo} \\
\textbf{Select Symbols}\ \_ & : & \mathcal{P}(\mathsf{Symbol}) \to \mathsf{Trafo} \\
\textbf{Forget Tags}\ \_ & : & \mathcal{P}(\mathsf{Tag}) \to \mathsf{Trafo} \\
\textbf{Forget Symbols}\ \_ & : & \mathcal{P}(\mathsf{Symbol}) \to \mathsf{Trafo}
\end{array}$$

Figure 3.1: Selection of rules

Figure 3.2 enumerates operators for the selection of symbols, that is, for the defined and used symbols in a given set of rules, for the symbols either prefixed or unprefixed (possibly restricted to a certain prefix) in a given set of symbols and for the symbols associated in a given sequence of associations.

Figure 3.3 enumerates operators for the selection of either all tags or the tags of rules defining certain symbols.

Figure 3.4 enumerates operators for the selection of parameters and variables of a certain sort.

Symbols In _	:	$\mathsf{Rules} \rightarrow \mathcal{P}(\mathsf{Symbol})$
Symbols Defined In _	:	$\mathsf{Rules} \rightarrow \mathcal{P}(\mathsf{Symbol})$
Symbols Used In _	:	$\mathsf{Rules} \rightarrow \mathcal{P}(\mathsf{Symbol})$
Unprefixed In _	:	$\mathcal{P}(\mathsf{Symbol}) \rightarrow \mathcal{P}(\mathsf{Symbol})$
Prefixed In _	:	$\mathcal{P}(\mathsf{Symbol}) \rightarrow \mathcal{P}(\mathsf{Symbol})$
Prefixed By _ In _	:	$\mathsf{Prefix} \times \mathcal{P}(\mathsf{Symbol}) \rightarrow \mathcal{P}(\mathsf{Symbol})$
Symbols Associated In _	:	$\mathsf{Association}^{\star} \rightarrow \mathcal{P}(\mathsf{Symbol})$

Figure 3.2: Selection of symbols

Tags In _	:	$\mathsf{Rules} \rightarrow \mathcal{P}(\mathsf{Tag})$
Tags For _ In _	:	$\mathcal{P}(\mathsf{Symbol}) \times \mathsf{Rules} \rightarrow \mathcal{P}(\mathsf{Tag})$

Figure 3.3: Selection of tags

Parameters Of Sort _ In _	:	$\mathsf{Sort} \times \mathcal{P}(\mathsf{Parameter}) \rightarrow \mathcal{P}(\mathsf{Parameter})$
Variables Of Sort _ In _	:	$\mathsf{Sort} \times \mathcal{P}(\mathsf{Variable}) \rightarrow \mathcal{P}(\mathsf{Variable})$

Figure 3.4: Selection of parameters / variables

There is an auxiliary operator

Positions _ For _ Of Sort _ : $\mathsf{Io} \times \mathcal{P}(\mathsf{Symbol}) \times \mathsf{Sort} \rightarrow \mathsf{Position}^{\star}$

for the construction of a sequence of selectors (for positions) addressing either input or output positions (first parameter) of certain symbols (second parameter) of a certain sort (third parameter).

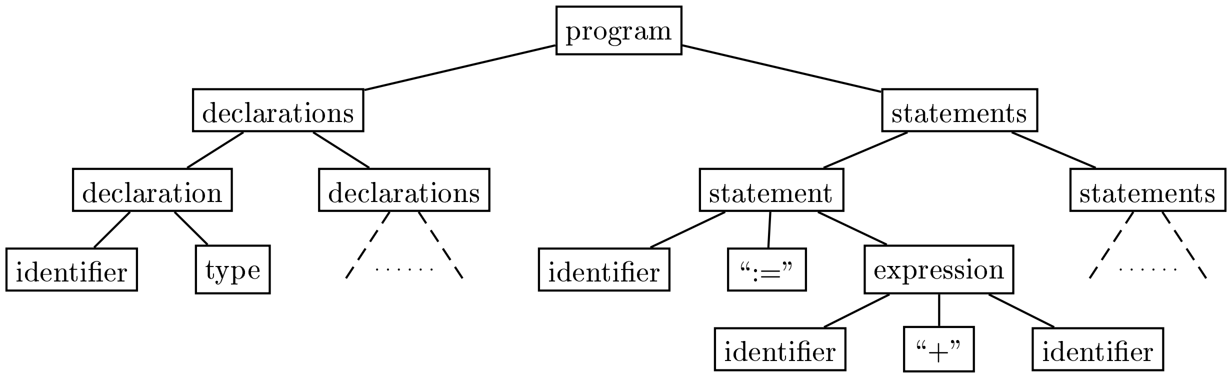
For selecting symbols to participate in a propagation, often closures in the sense of reachability similar to context-free grammars have to be computed; refer to Figure 3.5 for the auxiliary operators supporting the computation of such closures. Refer also to Figure 3.6 for some examples for the corresponding closures. The closures facilitate, for example, the definition of propagation schemata; refer to Subsection 3.4.2.

3.2.2 Renaming

Renaming all kinds of entities should be possible in a meta-program. The corresponding group of operators for renaming tags (**Rename Tag**), symbols (**Rename Symbol**), prefixes (**Rename Prefix**) and sorts (**Rename Sort**) is listed in Figure 3.7. The scope for renaming symbols can be restricted to conclusions (**Rename Conclusion**) and premises (**Rename Premise**). Renaming a sort can be restricted to some parameter positions (**Rename Positions**). The specifications of all these renaming operators correspond to traversals. It should be pointed out that the term *renaming* is meant here in very broad

Derivable From _ In _	:	$\mathcal{P}$(Symbol) × Skeleton → $\mathcal{P}$(Symbol)
Derivable To _ In _	:	$\mathcal{P}$(Symbol) × Skeleton → $\mathcal{P}$(Symbol)
From _ To _ In _	:	$\mathcal{P}$(Symbol) × $\mathcal{P}$(Symbol) × Skeleton → $\mathcal{P}$(Symbol)

Figure 3.5: Computation of closures concerning reachability



statements is derivable to (**Derivable To** ...) *statement*, *identifier*, *expression* and to *statements* itself. *identifier* is derivable from (**Derivable From** ...) *program*, *declarations*, *declaration*, *statements*, *statement* and *expression*. The symbols *statements* and *statement* occur on paths between *program* and *expression* (**From** ... **To** ...).

Figure 3.6: Examples for reachability

sense. It is possible, for example, to identify two entities (e.g. two symbols or two sorts) by the above operators. Identification is usually not permitted when renaming is regarded as restricted form of substitution, but in the case of meta-programming it is desirable, e.g. for the instantation of open programs.

Rename	**Symbol _ To _**	:	Symbol × Symbol → Trafo
Rename	**Conclusion _ To _**	:	Name × Name → Trafo
Rename	**Premise _ To _**	:	Symbol × Symbol → Trafo
Rename	**Tag _ To _**	:	Tag × Tag → Trafo
Rename	**Prefix _ To _**	:	Prefix × Prefix → Trafo
Rename	**Sort _ To _**	:	Sort × Sort → Trafo
Rename	**Positions _ To _**	:	$\mathcal{P}$(Position) × Sort → Trafo

Figure 3.7: Forms of renaming

Let us comment us slightly more in detail on the traversals implementing the rename operators. To rename a symbol, e.g. a name used in elements, is straightforward. To rename a sort σ to another sort σ', a traversal of the rules down to the parameters must be performed. Any parameter of sort σ is annotated during reconstruction with the sort σ'. Since the sort of a (meta-) variable v cannot be changed, a fresh variable v' of the new sort once for v in a rule need to be generated. v is then replaced by v' allover the rule.

Finally, the operator **Rename Positions** renaming sorts in certain parameter positions is discussed. The operator is useful to perform certain kinds of data refinements in target programs, where the sorts of parameter positions need to be unified or distinguished, the latter, for example, when new sum domains need to be established. Renaming positions is performed as follows. Variables of matching positions are replaced by variables of the new sort. Since the variables occurring on a certain matching position will in general occur on some other positions, too, all these positions should be renamed as well. Indeed, to force a consistent renaming, we must list all positions which have to be renamed simultaneously, i.e. if a variable occurs on a matching position, it must not occur on a position which is not explicitly listed for renaming. Renaming positions preserves semantics in the following sense.

Proposition 3.2.1
Let be $d_1, \ldots, d_n \in \mathsf{Io}$, $s_1, \ldots, s_n \in \mathsf{Symbol}$, $\sigma, \sigma' \in \mathsf{Sort}$. Concerning the interpretation used for the semantics definition (refer to Subsection 2.6.9) we assume that the carriers of σ and σ' are unified. Then the transformation

$$\textbf{Rename Positions } \{\langle d_1, s_1, \sigma\rangle, \ldots, \langle d_n, s_n, \sigma\rangle\} \textbf{ To } \sigma'$$

is semantics-preserving. $\diamondsuit$

In §3.4.3.5 on interpolating precomputations and others we will give an example where renaming positions is useful in establishing a new sum domain in a given specification; refer to Example 3.4.6.

3.2.3 Sorting

In this subsection a trivial operator **Order By** _ : $\mathcal{P}(\mathsf{Tag}) \rightarrow \mathsf{Trafo}$ for arranging sequences of rules in a certain way is presented. Such a transformation is useful, for example, to preserve the order of the rules, which serve as an input for a transformation, in the output of the transformation. There are possibly other operators doing some kind of sorting or "pretty printing" which could be useful in an operator suite for meta-programming. We will not go into detail, but we want to mention a further simple example. Results of transformations which are presented to the user should contain meaningful variable names. Auxiliary operators could be useful to *preserve variable identifiers* provided by the user and to *renumber variable indices* if appropriate.

```
λ by : P(Tag) . λ rs : Rules .
 ( Fold Left
    λ rest : Rules . λ t : Tag . rest ⋈ (Select Tags {t} On rs)
   Neutral ⟨ ⟩ Set by
 )
   ⋈ (Forget Tags by On rs).
```

Figure 3.8: **Order By** _ : $\mathcal{P}(\mathsf{Tag}) \rightarrow \mathsf{Trafo}$

Order By *ts* **On** *rs* arranges all rules in *rs* so that the relative order of the tags in *ts* is preserved, and all rules with tags not covered by *ts* are appended to the end preserving the original order in *rs*. The order of the rules can be relevant in some instances, e.g. it is operationally relevant for logic programs with the depth-first proof search rule. Moreover, the preservation of the order usually contributes to readability.

Operations should be specified in such a way that the order of rules is preserved. For rule-wise transformations based on the **Fold** or **Map** recursion schemata this property is often achieved without further effort. However, for some operators it is necessary or more convenient to rearrange the result by an application of **Order By**.

3.2.4 Combinators

We present certain combinators on transformations. The corresponding operators compute from a given transformation another transformation. First, the operator **Replace** is suggested (§3.2.4.1). It supports so-called element substitution. For short, the operator takes some description of how conclusions and premises have to be transformed and derives a complete transformation performing all the smaller transformations in a systematic way at once. Afterwards, we introduce operators to support selective transformation (§3.2.4.2) and incremental construction of premises (§3.2.4.3).

3.2.4.1 Element substitution

We introduce a general schema for so-called element substitution. The search for such higher-order schemata is an interesting problem because these schemata allow more concrete schemata to be defined at a higher level of abstraction. In particular, some properties of schemata can be analysed at a more general level.

We need two auxiliary types:

$$\begin{array}{rcl} \mathsf{LhsSubstitution} & = & \mathsf{Conclusion} \to (\mathsf{Conclusion} \otimes \mathsf{Premise}^\star \otimes \mathsf{Premise}^\star \otimes \mathsf{Substitution}) \\ \mathsf{RhsSubstitution} & = & \mathsf{Premise} \to (\mathsf{Premise}^\star \otimes \mathsf{Substitution}) \end{array}$$

These types are intended to specify how conclusions and premises can be replaced. The combinator **Replace** can be used to define several transformation schemata which adapt elements in a systematic way as we will see below. **Replace** t_l t_r applied to a rule

$$[t]\ e_0 \Leftarrow e_1, \ldots, e_n$$

returns the following rule:

$$[t]\ \theta_n \circ \cdots \circ \theta_1(e_0') \Leftarrow e_0^\star, \theta_n \circ \cdots \circ \theta_2 \circ \theta_0(e_1^\star), \ldots, \theta_{n-1} \circ \cdots \circ \theta_1 \circ \theta_0(e_n^\star), e_0^{\star\prime},$$

where $t_l(e_0) \mapsto \langle e_0{}', e_0^\star, e_0^{\star\prime}, \theta_0 \rangle$ and $t_r(e_q) \mapsto \langle e_q^\star, \theta_q \rangle$ for $q = 1, \ldots, n$.

The identity for replacements on LHSs, denoted by *lhsIdentity*, is defined by the expression $\lambda e : \mathsf{Element}.\langle\ e, \langle\rangle, \langle\rangle, \langle\rangle\ \rangle$. The identity for replacements of premises, denoted by *rhsIdentity*, is defined by the expression $\lambda e : \mathsf{Element}.\langle\ \langle e\rangle, \langle\rangle\ \rangle$. As defined above, all the single substitutions are just composed. Another probably more general approach would be to accumulate a set of equations and then to use the substitution corresponding to its solved form.

In some applications of **Replace**, e.g. to add a parameter position for a certain symbol, substitution is not involved. However, for other applications, substitution is necessary, for example, for the contraction of parameter positions, the substitution has to unify contracted positions. In many applications, a specific LHS / RHS substitution will behave for many elements like the identity function.

3.2.4.2 Selective transformation

Functions on Rules must often be restricted. If a computation is inserted into a certain rule r, for example, the corresponding transformation has to be restricted to r. A group of operators **Selecting / Forgetting** are offered for that purpose; refer to Figure 3.9.

Selecting Symbols _ Do _	:	$\mathcal{P}(\mathsf{Symbol}) \times \mathsf{Trafo} \to \mathsf{Trafo}$
Selecting Tags _ Do _	:	$\mathcal{P}(\mathsf{Tag}) \times \mathsf{Trafo} \to \mathsf{Trafo}$
Forgetting Symbols _ Do _	:	$\mathcal{P}(\mathsf{Symbol}) \times \mathsf{Trafo} \to \mathsf{Trafo}$
Forgetting Tags _ Do _	:	$\mathcal{P}(\mathsf{Tag}) \times \mathsf{Trafo} \to \mathsf{Trafo}$

Figure 3.9: Selective transformation

Selecting Tags $\langle t_1, \ldots, t_n\rangle$ **Do** *trafo* **On** *rs* transforms *rs* in the following steps:

1. The rules with tags $t_1, \ldots, t_n$ are selected in *rs*.
2. *trafo* is applied to the result of (1.).
3. The result of (2.) and all rules which were not selected in (1.) are concatenated.

```
λ ts : P(Tag) . λ trafo : Trafo . λ rs : Rules .
 Order By Tags In rs
  On ((trafo On (Select Tags ts On rs)) ⋈ (Forget Tags ts On rs)).
```

Figure 3.10: **Selecting Tags _ Do _** : $\mathcal{P}(\mathsf{Tag}) \times \mathsf{Trafo} \to \mathsf{Trafo}$

The transformation *trafo* has to be type-preserving and it must not return rules with tags overlapping with the complementary set of rules selected in (1.), if the definedness of the whole transformation, in general, is required to follow from the definedness of (2.). The simple definition of **Selecting Tags** is shown in Figure 3.10. Other forms of restricted transformations can be expressed based on **Selecting Tags**, in the same manner as **Select Tags** is sufficient to express **Select Symbols**, **Forget Tags** and **Select Symbols**:

- **Selecting Symbols**: restriction to rules defining certain symbols,
- **Forgetting Tags**: restriction to the complementary set of **Selecting Tags**,
- **Forgetting Symbols**: restriction to the complementary set of **Selecting Symbols**

3.2.4.3 Temporary invisibility of symbols

Finally, the operator **Hiding** is proposed. **Hiding** *s* **Do** *t* **On** *rs* makes the symbol *s* invisible in *rs* during the performance of the transformation *t* applied to *rs*; refer to Figure 3.11 for the formal definition. Hiding is based here on renaming the symbol *s* to a fresh symbol which is definitely not in use. An extra service added to the actual operator **Hiding** is that the profile of the hidden symbol is possibly permuted (refer to Subsection 3.3.1 for the permutation of positions) to coincide with the profile according to the existing use before the renaming is undone.

```
λ sym : Symbol . λ t : Trafo . λ rs : Rules .
 Let profOld = Profile Of sym In Sigma Of rs In
  Let fresh = Name? sym → New Name, New Operation In
    Rename Symbol fresh To sym
  ∘ (λ rs : Rules .
% if the hidden symbol is not present, no permutation will be necessary
        profOld = ? →
         rs,
         Let profNew = Profile Of sym In Sigma Of rs In
          profNew = ? →
           rs,
% if the profiles are equal, no permutation is necessary
          (Sorts Input Of profOld = Sorts Input Of profNew) And
          (Sorts Output Of profOld = Sorts Output Of profNew) →
           rs,
% try to permute the new computational elememts according to the original profile
           Permute profOld On rs
     )
  ∘ t
  ∘ Rename Symbol sym To fresh
   On rs.
```

Figure 3.11: **Hiding _ Do _** : Symbol × Trafo → Trafo

Hiding turned out to be necessary, for example, for the *incremental construction of premises*. Essentially, the operator **Hiding** enables us to insert and parameterize premises (and conclusions) in several steps without conflicting with existing uses of the underlying symbol in the given program. Note that without further effort, stepwise parameterization is not possible due to the type system because a symbol must have a consistent type all over a target program at any time. On the other hand, stepwise parameterization is necessary with regard to orthogonality of operators and granularity of adaptation. Besides type conflicts, there is another problem with "naive" stepwise construction of computational elements: While constructing computational elements in n steps, it is not straightforward how to avoid that existing elements with the same underlying symbol are not effected in steps 2, ..., $n-1$. Here step 1 is assumed to perform the initial construction of the computation. Sometimes, such interferences can be avoided by "forgetting" (refer ro §3.2.4.2) some rules during the transformation, but not in general.

The permutation of the parameterization is useful for stepwise construction of computational elements because there are usually dependencies on the order of the steps which influence the order of the parameterization. Thereby, the order of parameters might differ between the new elements and existing uses.

3.3 Basic schemata

According to Figure 1.12 we want to describe a number of schemata modelling basic concepts during program synthesis, adaptation and composition. If we think for example of the "incremental" development of an attribute grammar or its adaptation by means of metaprogramming, certain basic concepts are evident. More elaborate schemata in the sense of strategies are explored in the subsequent section. We suggest the following classification for basic schemata:

- schemata dealing primary with *positions* (or parameterization), e.g. the operator **Add** to add a parameter position; refer to Subsection 3.3.1;
- schemata concerning copies, definitions and uses, e.g. the operator **Define** to define a parameter position by a constant computation; refer to Subsection 3.3.2;
- schemata acting primary at the rules level, e.g. operators facilitating program transformation in the sense of folding and unfolding.

3.3.1 Positions

Semantic aspects of an AG specification or a natural semantics description are roughly represented by the profiles of the underlying symbols. The traversal, the propagation and the synthesis of data structures can be associated with corresponding parameter positions of certain symbols. The specification of type checking, for example, requires an output position for the grammar symbol for expressions, since a type position has to be synthesized. Consequently, the suite should offer corresponding operators. There are basic operators for adding (**Add**), removing (**Sub**), contracting (**Contract**) and permuting (**Permute**) parameter positions; refer to Figure 3.12 for the profiles of the operators.

$$
\begin{array}{rcl}
\textbf{Add}\ \_ & : & \mathsf{Position} \rightarrow \mathsf{Trafo} \\
\textbf{Sub}\ \_ & : & \mathsf{Position} \rightarrow \mathsf{Trafo} \\
\textbf{Contract}\ \_ & : & \mathsf{Position} \rightarrow \mathsf{Trafo} \\
\textbf{Permute}\ \_ & : & \mathsf{Profile} \rightarrow \mathsf{Trafo}
\end{array}
$$

Figure 3.12: Basic schemata for positions

The operator **Sub** for the removal of parameter positions can be regarded as the opposite of **Add**. Contraction of parameterization as facilitated by the operator **Contract** needs to be performed during program composition if two operands have some part of the parameterization in common. More precisely, the operator **Contract** is suitable to unify all (input or output) parameters of a symbol s which are of the same sort σ. It is not so obvious if contraction is a basic concept like the addition of positions. Possibly, we could argue that contraction can be regarded as a combination of unification and removal of parameter positions. Contraction will be demonstrated in §3.3.3.1, where the composition of specifications with the same underlying skeleton is discussed. Permutation of parameters is a very simple operation. It is obviously needed in a position-oriented framework. The operator **Permute** is based rather on a profile than indices of positions. Describing the permutation of a symbol's parameterization by means of the intended profile is more readable, but we must insist on uniquely sorted symbols. Subsection 3.4.1 will show some elaborate schemata dealing with positions. Recall that there is also a form of renaming which can be used to change the sorts of some parameter positions (**Rename Positions**); refer to Subsection 3.2.2 on renaming for this issue.

The above schemata can be expressed in terms of the parameterized transformation schema for element substitution introduced in §3.2.4.1 because the schemata can be regarded as homogeneous transformations of parameter lists of elements; refer to Section C.2.

Let us consider the operator **Add** in more detail. **Add** $\langle io, s, \sigma\rangle$ adds an input position ($io =$ **Input**) or an output position ($io =$ **Output**) respectively to any element with the symbol s by inserting fresh variables of the sort σ.

Example 3.3.1
The operator **Add** is used to add *several* parameter positions suitable to eventually propagate a symbol table. We start from the following grammar fragment:

$$\textit{program} : \textit{declarations}, \textit{statements}. \qquad [\mathsf{prog}]$$

The following transformation is applied to the above fragment:

$$\begin{array}{ll} & \textbf{Add}\ \langle \textbf{Input}, \textit{declarations}, \mathsf{ST}\rangle \\ \circ & \textbf{Add}\ \langle \textbf{Output}, \textit{declarations}, \mathsf{ST}\rangle \\ \circ & \textbf{Add}\ \langle \textbf{Input}, \textit{statements}, \mathsf{ST}\rangle \end{array}$$

The result of the transformation is as follows:

$$\begin{array}{lll} \textit{program} & : & \textit{declarations}(\mathsf{ST}_0) \rightarrow (\mathsf{ST}_1), \qquad [\mathsf{prog}] \\ & & \textit{statements}(\mathsf{ST}_2). \end{array}$$

◇

3.3.2 Copies & Definitions & Uses

To *incrementally develop* the computational behaviour of a specification, there is a need for unifying parameters and adding computational elements including conditions. To *adapt* a specification we have to be able to perform the oppositional transformation, i.e. to liquidate unification of parameter positions or to remove computational elements. During the synthesis of an attribute grammar, for example, attributes (in our framework: parameter positions) are added in a first step. This is modelled by the operator **Add**. In a next step usually the new attributes (positions) are "defined" in the sense of definitions and copies corresponding to the insertion of semantic rules including semantic copy rules. During adaptation, semantic rules possibly have to removed or replaced. The corresponding set of basic operators is shown in Figure 3.13.

Copy _ **To** _	:	$\mathsf{Position} \times \mathsf{Position} \rightarrow \mathsf{Trafo}$
Define _ **By** _	:	$\mathsf{Position} \times \mathsf{Symbol} \rightarrow \mathsf{Trafo}$
Use _ **By** _	:	$\mathsf{Position} \times \mathsf{Symbol} \rightarrow \mathsf{Trafo}$
Undefined! _	:	$\mathsf{Position} \rightarrow \mathsf{Trafo}$
Unused! _	:	$\mathsf{Position} \rightarrow \mathsf{Trafo}$
Purge _	:	$\mathsf{Symbol} \rightarrow \mathsf{Trafo}$

Figure 3.13: Basic computation schemata

Several more elaborate schemata for adding computational behaviour are discussed in Section 3.4.3.

3.3.2.1 Copies

The simplest way to eliminate an undefined variable occurrence at a position pos_u is to copy a parameter from a defining position pos_d to the position pos_u, what is expressed by **Copy** pos_d **To** pos_u. Applying the transformation to a rule r, pos_d has to be a unique selector of a defining position in r, whereas pos_u has to be a unique selector for an applied position in r.

To restrict the data flow to copying from defining to applied positions has been adopted from attribute grammars, where semantic rules always define synthesized attributes of the LHS and inherited attributes of the RHS (i.e. applied positions in our terminology) and the normal form property of attribute grammars says that only inherited attributes of the LHS and synthesized attributes of the RHS can be used in the semantic rules (i.e. defining positions in our terminology). The same assumption will be used whenever computational elements (e.g. definitions and uses) are inserted, i.e. the input (resp. output) positions have to be unified with defining (resp. applied) positions.

Example 3.3.2
We continue Example 3.3.1 by adding a copy rule in order to inherit the symbol table synthesized in the declaration part to the statement part. Applying the following transformation to Example 3.3.1

$$\textbf{Copy}\ \langle \textbf{Output}, \mathit{declarations}, \mathsf{ST}\rangle\ \textbf{To}\ \langle \textbf{Input}, \mathit{statements}, \mathsf{ST}\rangle$$

we obtain the following output:

$$\begin{array}{lll} \mathit{program} & : & \mathit{declarations}(\mathsf{ST}_0) \rightarrow (\mathsf{ST}_1), \qquad [\mathsf{prog}] \\ & & \mathit{statements}(\mathsf{ST}_1). \end{array}$$

◊

3.3.2.2 Adding definitions

The operator **Define** can be used to insert definitions. Consider the transformation **Define** *pos* **By** s applied to the rule r. For every undefined parameter p matching with *pos* a new premise of the form $s \to p$ is inserted into r. This constant computation is intended to compute the corresponding parameter p.

Example 3.3.3
We continue Example 3.3.1 and Example 3.3.2. The initialization of the symbol table is modelled by the following transformation:

$$\textbf{Define } \langle \textbf{Input}, \mathit{declarations}, \mathsf{ST} \rangle \textbf{ By } \&_{static}\ \mathit{init}$$

The corresponding output is as follows:

$$\begin{array}{lcll} \mathit{program} & : & \&_{static}\ \mathit{init} \to (\mathsf{ST}_0), & [\mathsf{prog}] \\ & & \mathit{declarations}(\mathsf{ST}_0) \to (\mathsf{ST}_1), & \\ & & \mathit{statements}(\mathsf{ST}_1). & \end{array}$$

$\diamondsuit$

The operator **Define** is also applicable if the given selector is not a unique selector for an undefined position because it usually makes sense to insert the corresponding constant computation for *all* matching positions. Note that by adding input positions to the symbol used for the constant computation and providing definitions to these new positions, the constant computations can be extended to computations of any arity.

Proposition 3.3.1
$\forall d \in \mathsf{Io}\ \forall s, s' \in \mathsf{Symbol}, \forall \sigma \in \mathsf{Sort}$, the transformation **Add** $\langle d, s, \sigma \rangle$ is $\mathcal{WD}$-recoverable by the transformation **Define** $\langle d, s, \sigma \rangle$ **By** s'. $\diamondsuit$

3.3.2.3 Adding uses

Uses of defining positions can be forced by the operator **Use**. Consider the transformation **Use** *pos* **By** s applied to the rule r. For every parameter p on a defining parameter position matching with *pos* a new premise of the form $s(p)$ (i.e. a unary condition) is inserted into r. For pragmatic reasons the opertor **Use** is not completely dual to the operator **Define** in the sense that the focus is on all relevant defining positions and not only on unused variables (on defining positions). It is not very common to consider multiple definitions for a variable, whereas multiple uses are very common. Actually, there should be operators dual to **Define** and **Use**, but they have not been required so far.

3.3.2.4 Establishing undefined and unused variables

The operators for inserting copy rules (**Copy**) or computational elements serving as definitions (**Define**) or uses (**Use**) of positions are central to program synthesis. However, to adapt a program, the effect of such operators has possibly to be nullified.

For an application of the **Undefined!** operator, every matching applied position is refreshed, i.e. a fresh variable of the appropriate sort is inserted. This behaviour is obviously sufficient to discard the effect of a copy rule. Suppose that a computation served as definition of the matching position. Then the computation will possibly be useless.

The operator **Unused!** causes every matching defining position to be refreshed. Suppose that a (unary) condition served as use of the matching position. Then the condition will possibly be useless.

3.3.2.5 Removing computations

The operator **Purge** supports the removal of computations. To retain orthogonality w.r.t. **Undefined!** and **Unused!** a computation with the symbol s is removed in rs by **Purge** s **On** rs if all of its input parameters are undefined and all of its output parameters are unused. Obviously, the effect of **Define** (resp. **Use**) can be nullified by **Undefined!** (resp. **Unused!**) and a subsequent application of **Purge**.

3.3.3 Rules

Here we define some basic operators acting at the rules level. The profiles of the corresponding operator are shown in Figure 3.14. First, the concept of superimposition (**Superimpose**) is introduced. Afterwards, program transformations supporting folding and unfolding are added to the operator suite (**Fold** and **Unfold**).

Superimpose _ **And** _	:	Rules × Rules → Rules
Fold _ **By** _ **Into** _	:	Tag × Name?* × Tag → Trafo
Unfold _ **By** _ **Into** _	:	Tag × Tag?* × Tag? → Trafo

Figure 3.14: Basic schemata for rules

3.3.3.1 Superimposition

The operator $\_ \bowtie \_$ provides the most obvious form of composition for two sequences of rules, that is to say the concatenation of the rules. Another form of composition will be discussed in the sequel. This form models *superimposition* of rules synchronized by skeletons. Thereby, the parameterization and the computational behaviour of two sequences of rules can be combined in one specification in the sense of tupling.

More in detail, **Superimpose** rs_1 **And** rs_2 superimposes the skeletons of the operands, concatenates the parameters of superimposed symbols and takes over computational elements.

Example 3.3.4
We compose the complete frontend specification from Figure 2.9 by means of superimposition. Figure 2.15 and Figure 2.16 contain two projections which can be superimposed to

obtain the complete specification. The first projection models static semantics, whereas the second projection specifies AST construction.

$$\text{Figure 2.9} \equiv \textbf{Contract}\ \langle \textbf{Output}, \textit{identifier}, \mathsf{ID}\rangle\ \textbf{On}$$
$$\textbf{Superimpose}\ \text{Figure 2.15}\ \textbf{And}\ \text{Figure 2.16}$$

An application of **Contract** is involved in the above composition, because the parameterization of the symbol *identifier*, which both operands have in common, has to be unified.

◊

Superimposition is defined iff the operands are based on the same skeleton. Refer to Section C.5 for the formal definition. Example 3.3.4 demonstrates how to handle the case of a common part of parameterization which has to be unified, although it is a trivial example, where the common part corresponds to "terminal" attribution. However, the approach of using **Contract** (refer to Section 3.3.1 for its presentation) also works for more complex parameterizations. Thereby, we support a kind of modularity similar to Watt's Partitioned AGs [Wat75]. Moreover, a reconstruction of *composition* in the sense of stepwise enhancement [Lak89, SS94, JS94] is achieved.

3.3.3.2 Folding

Folding and unfolding is well-known in semantics-preserving program transformation; refer e.g. to [PP94]. We suggest two operators **Fold** and **Unfold** which are particularly suitable for structural transformations in meta-programs; refer to Figure 3.14 for the profiles and refer to Section C.6 for the formal definitions.

Fold t **By** $\langle s?_1, \ldots, s?_n\rangle$ **Into** t' with $\exists k$ such that $s?_1 = ?, \ldots, s?_{k-1} = ?$, $s?_k \neq ?$, $s?_{k+1} = ?, \ldots, s?_n = ?$, folds the rule with the tag t in the following manner:

- The $s?_i$ are matched with the skeleton symbols of the premises of t, where ? matches with any single element and $s?_k$ matches with any non-empty sequence of elements.
- The elements $e^\star$ covered by $s?_k$ are replaced by an element e with symbol $s?_k$ and the undefined variables in $e^\star$ as inputs and the defining occurrences in from $e^\star$ with applied occurrences outside of $e^\star$ as outputs.
- Moreover, the rule $[t']\ e \Leftarrow e^\star$ is added.

3.3.3.3 Unfolding

Unfold offers the operation reverse to **Fold**, i.e. some skeleton elements e_1, …, e_m among the premises of a rule t are replaced by the premises of some rules t_1, …, t_m with the same symbols in the conclusion as the e_i. If a premise (note that only skeleton element are counted) of t matches with ?, the corresponding premise will be taken over unchanged, i.e. it will not be unfolded.

$$\begin{array}{lcll}
\textit{statement}(\mathsf{ST}) \to (\mathsf{STM}) & : & \textit{expression}(\mathsf{ST}) \to (\mathsf{T}, \mathsf{EXP}), & [\mathsf{if}] \\
 & & \&_{static}\ \textit{isBool}(\mathsf{T}), & \\
 & & \textit{statements}(\mathsf{ST}) \to (\mathsf{STM}_1), & \\
 & & \boxed{\textit{else}(\mathsf{ST}) \to (\mathsf{STM}_2)}, & \\
 & & \&_{ast}\ \textit{if}(\mathsf{EXP}, \mathsf{STM}_1, \mathsf{STM}_2) \to \mathsf{STM}. & \\
\boxed{\textit{else}(\mathsf{ST}) \to (\mathsf{STM})} & : & \textit{statements}(\mathsf{ST}) \to (\mathsf{STM}). & [\mathsf{else}] \\
\boxed{\textit{else}(\mathsf{ST}) \to (\mathsf{STM})} & : & \boxed{\&_{ast}\ \textit{skip} \to (\mathsf{STM})}. & [\mathsf{noelse}]
\end{array}$$

Figure 3.15: An optional if-construct obtained by fold/unfold

Example 3.3.5
Let us adapt the frontend specification from Figure 2.9 in such a way that an optional else-path is supported. Figure 3.15 shows how the changed GSF rules have to look like. The actual transformation can be described as follows:

$$\begin{array}{rcl}
\text{Figure 3.15} & \equiv & \textbf{Unfold}\ [\mathsf{else}]\ \textbf{By}\ \langle[\mathsf{skip}]\rangle\ \textbf{Into}\ [\mathsf{noelse}] \\
 & \circ & \textbf{Fold}\ [\mathsf{if}]\ \textbf{By}\ \langle ?, ?, \mathsf{else}\rangle\ \textbf{Into}\ [\mathsf{else}] \\
 & & \textbf{On}\ \text{Figure 2.9}
\end{array}$$

♢

3.4 Elaborate schemata

It was the intention of the previous section to present schemata corresponding to *basic* concepts in program synthesis, adaptation and composition. In contrast to that, the schemata of this Section are rather thought as *strategies*. Some of the schemata presented below are thought directly as elaborations of some basic schema. Other schemata are obtained by a more involved derivation combining several basic aspects.

First, some schemata extending our tool set for dealing with *positions* (or parameterization) are discussed in Subsection 3.4.1. Second, *propagation* schemata are investigated in Subsection 3.4.2. We carry on with certain strategies to establish *computational* behaviour in Subsection 3.4.3. Finally, *composition* schemata are discussed in Subsection 3.4.4.

3.4.1 Positions

The elaborations of schemata for positions introduced in Subsection 3.3.1 are straightforward. The operators **Add**, **Sub** and **Contract** are generalized to cope with several positions at once. The operator **Ensure** facilitates the conditional addition of a position depending on the fact if a position of the corresponding sort has not yet been added before. **Ensure** $\langle\langle io, s, \sigma\rangle\rangle$ **On** rs adds the position if and only if s has not an io-position of sort σ in rs. Several elaborate schemata defined below use the operator **Ensure** to ensure the existence of a certain parameterization as a kind of precondition. The operator **Project** combines permutation (**Permute**) and removal (**Sub**) of parameter positions. For projection we assume uniquely sorted symbols.

Add _	:	Position* $\rightarrow$ Trafo
Sub _	:	Position* $\rightarrow$ Trafo
Contract _	:	Position* $\rightarrow$ Trafo
Ensure _	:	Position* $\rightarrow$ Trafo
Project _	:	Profile $\rightarrow$ Trafo

Figure 3.16: More elaborate schemata for positions

3.4.2 Propagation

Many aspects of a declarative program concern the propagation of data. Thus, it is obvious that the operator suite should provide corresponding support; refer to Figure 3.17 for an enumeration of corresponding operators.

From The Left _	:	Sort $\rightarrow$ Trafo
Left To Right _	:	Sort $\rightarrow$ Trafo
Inherit _ **From** _ **To** _	:	Sort $\times$ $\mathcal{P}$(Symbol) $\times$ $\mathcal{P}$(Symbol) $\rightarrow$ (Skeleton $\rightarrow$ Trafo)
Accumulate _ **From** _ **To** _	:	Sort $\times$ $\mathcal{P}$(Symbol) $\times$ $\mathcal{P}$(Symbol) $\rightarrow$ (Skeleton $\rightarrow$ Trafo)
Remote _ **From** _	:	Sort $\times$ $\mathcal{P}$(Symbol) $\rightarrow$ Trafo

Figure 3.17: Schemata for propagation

3.4.2.1 Left-to-right dependencies

The operator **From The Left** _ : Sort $\rightarrow$ Trafo. facilitates propagation by *copying systematically defining occurrences of a certain sort to undefined variables from left to right.* As long as we consider variables as the only form of parameters, every single copy rule could be expressed by means of the operator **Copy**, but **From The Left** is independent from parameter positions and a single application of the operator corresponds to a potentially unlimited number of copy rules. The schema is sufficient to establish a computational behaviour suitable to encode pervasive inheritance or a bucket brigade [Ada91] or any mixture of them provided the necessary positions have been added in advance. All of the propagation schemata considered below make use of the operator **From The Left**.

Example 3.4.1
The propagation of a symbol table can be specified by an application of **From The Left** as follows:

$$
\begin{array}{lcl}
\text{Figure 3.18} & \equiv & \textbf{From The Left}\ \mathsf{ST} \\
 & \circ & \textbf{Add}\ \langle\langle\textbf{Input}, \textit{declarations}, \mathsf{ST}\rangle, \langle\textbf{Input}, \textit{declaration}, \mathsf{ST}\rangle, \\
 & & \quad \langle\textbf{Output}, \textit{declarations}, \mathsf{ST}\rangle, \langle\textbf{Output}, \textit{declaration}, \mathsf{ST}\rangle, \\
 & & \quad \langle\textbf{Input}, \textit{statements}, \mathsf{ST}\rangle, \langle\textbf{Input}, \textit{statement}, \mathsf{ST}\rangle, \\
 & & \quad \langle\textbf{Input}, \textit{expression}, \mathsf{ST}\rangle\rangle \\
 & \textbf{On} & \text{Figure 2.14}
\end{array}
$$

◊

program	:	*declarations*(ST_0) → (ST_1), *statements*(ST_1).	[prog]
declarations(ST_0) → (ST_2)	:	*declaration*(ST_0) → (ST_1), *declarations*(ST_1) → (ST_2).	[decs]
declarations(ST) → (ST)	:	.	[nodecs]
...			
statements(ST)	:	*statement*(ST), *statements*(ST).	[concat]
statements(ST)	:	.	[skip]
...			

Figure 3.18: Symbol table propagation

We suggest the operator **Left To Right** σ as a slight elaboration of **From The Left**. Copying is performed based on *fresh* positions rather than the original positions. Thereby, additional symbols and positions can be incorporated in an existing propagation path. It seems to be impossible to express such an adaptation within other common frameworks, particularly [KW94], [Kos91] and [KLMM93] (rule models). Refer to the motivating example in Subsection 1.2.2.

All the following propagation schemata have in common, that **Left To Right** (which in turn is conceptionally based on **From The Left**[1]) or **From The Left** are used in order to establish the proper data flow. The schemata differ in the way how symbols participating in the propagation are selected and how the mode of propagation for every symbol, i.e. inheritance or accumulation, is defined.

3.4.2.2 Inheritance

The operator **Inherit** propagates a data structure according to pervasive inheritance [Ada91]. Two sets of symbols *from* and *to* are required. The parameter *from* enumerates the symbols where the propagation should start. Often this is a singleton set. The

[1]The definition of **Left To Right** shown in Figure 1.5 does not refer to **From The Left**, but we are working on a reformulation of the propagation schemata to express this relationship.

parameter *to* corresponds to the symbols which require reading access to the propagated data structure of a certain sort. The data structure is copied along input positions of all symbols in the closure (concerning reachability) between *from* and *to* including *to*. Refer to Figure 3.19 for the definition of the operator.

```
λ s : Sort . λ from : $\mathcal{P}$(Symbol) . λ to : $\mathcal{P}$(Symbol) .
λ sk : Skeleton . λ rs : Rules .
 Let closure = (From from To to In sk) ∪ to In
    Left To Right s
  ∘ Ensure Positions Input For closure Of Sort s
     On rs.
```

Figure 3.19: **Inherit _ From _ To _** : Sort × $\mathcal{P}$(Symbol) × $\mathcal{P}$(Symbol) → (Skeleton → Trafo)

3.4.2.3 Accumulation

The operator **Accumulate** is quite similar to the previous one, but the symbols in *to* require reading *and* writing access to the data structure. Thus, for all relevant symbols an input and an output position is added. The data flow achieved by the subsequent application of the **Left To Right** ensures that an accumulator is simulated. Refer to Figure 3.20 for the definition of the operator. The corresponding propagation pattern is also called *bucket brigade* [Ada91].

```
λ s : Sort . λ from : $\mathcal{P}$(Symbol) . λ to : $\mathcal{P}$(Symbol) .
λ sk : Skeleton . λ rs : Rules .
 Let closure = (From from To to In sk) ∪ to In
    Left To Right s
  ∘ Ensure (Positions Output For closure Of Sort s)
  ∘ Ensure (Positions Input For closure Of Sort s)
     On rs.
```

Figure 3.20: **Accumulate _ From _ To _** : Sort × $\mathcal{P}$(Symbol) × $\mathcal{P}$(Symbol) → (Skeleton → Trafo)

3.4.2.4 Remote access

A set of symbols *read* of defined symbols in rules with undefined occurrences of a given sort is derived assuming that these symbols need reading access to the data structure. Dually, a set of symbols *write* of defined symbols in rules with unused occurrences of the given sort is derived assuming that these symbols update or synthesize the data structure. Input and output positions are added to symbols accordingly based on closures between a parameter *from* as above and the derived sets *read* and *write*; refer to Figure 3.21.

The operator **Remote** promotes a style of specification similar to remote access [KW94, JF85, Boy96b, Boy98].

```
$\lambda$ s : **Sort** . $\lambda$ from : $\mathcal{P}$(**Name**) . $\lambda$ rs : **Rules** .
 **Let** undefined =
  **Map Union** $\lambda$ r : **Rule** .
   **Variables Of Sort** s **In** (**Ao In** r $\setminus$ **Do In** r) $= \emptyset \rightarrow$
    $\emptyset$,
    {**Name Of Conclusion Of** r}
  **List** rs
 **In**
  **Let** unused =
   **Map Union** $\lambda$ r : **Rule** .
    **Variables Of Sort** s **In** (**Do In** r $\setminus$ **Ao In** r) $= \emptyset \rightarrow$
     $\emptyset$,
     {**Name Of Conclusion Of** r}
   **List** rs
  **In**
   **Let** sk = **Skeleton Of** rs **In**
    **Let** read = (**From** from **To** undefined **In** sk) $\cup$ undefined **In**
     **Let** write = (**From** from **To** unused **In** sk) $\cup$ unused **In**
        **From The Left** s
      $\circ$ **Add** (**Positions Input For** read **Of Sort** s)
      $\circ$ **Add** (**Positions Output For** write **Of Sort** s)
       **On** rs.
```

Figure 3.21: **Remote** _ **From** _ : Sort $\times$ $\mathcal{P}$(Symbol) $\rightarrow$ Trafo

Example 3.4.2
Consider the rules in Figure 3.22 as variants of the corresponding rules in the frontend specification in Figure 2.9. The symbol table is *used* in computations, but the actual propagation is not specified. The resulting attribute grammar has to be considered as non-well-defined because of the uses undefined ST-positions. Moreover, there are unused ST-positions, e.g. in [dec]. However, the propagation of the symbol table can be established by the following application of the operator **Remote**:

$$\text{Figure 2.9} \quad \equiv \quad \textbf{Remote}\ \mathsf{ST}\ \textbf{From}\ \{\textit{program}\}\ \textbf{On}\ \text{Figure 3.22}$$

$\diamondsuit$

3.4.3 Computations

In Subsection 3.3.2 basic operators concerning the addition of “copy rules”, constant computations serving as definitions and unary conditions serving as uses were introduced. In this subsection, elaborations for these schemata, i.e. to insert *arbitrary* computational elements (**Compute** and **Condition**), are presented. Afterwards, the operator **Default**, which is more flexible in some cases than the operator **Define**, is introduced. Then, three advanced schemata for establishing computational behaviour (**Relate**, **Reduce** and **Precompute**) are presented. Finally, the relationship between computational elements and term construction (**Construction** etc.) is investigated.

$$\begin{array}{lll}
\textit{program} \rightarrow (\mathsf{PRO}) \ \vdots & \&_{static}\ \textit{init} \rightarrow (\boxed{\mathsf{ST}}), & [\mathsf{prog}] \\
& \textit{declarations} & \\
& \textit{statements} \rightarrow (\mathsf{PRO}), & \\
& \&_{ast}\ \textit{prog}(\mathsf{STM}) \rightarrow \mathsf{PRO}. & \\
\ldots & & \\
\textit{declaration} \ \vdots & \textit{identifier} \rightarrow (\mathsf{ID}), & [\mathsf{dec}] \\
& \textit{type} \rightarrow (\mathsf{T}), & \\
& \&_{static}\ \textit{add}(\boxed{\mathsf{ST}}, \mathsf{ID}, \mathsf{T}) \rightarrow (\boxed{\mathsf{ST'}}). & \\
\ldots & & \\
\textit{statement} \rightarrow (\mathsf{STM}) \ \vdots & \textit{identifier} \rightarrow (\mathsf{ID}), & [\mathsf{assign}] \\
& \&_{static}\ \textit{lookup}(\boxed{\mathsf{ST}}, \mathsf{ID}) \rightarrow (\mathsf{T}_{\mathsf{LHS}}), & \\
& \textit{expression} \rightarrow (\mathsf{T}_{\mathsf{RHS}}, \mathsf{EXP}), & \\
& \&_{static}\ \textit{assignable}(\mathsf{T}_{\mathsf{LHS}}, \mathsf{T}_{\mathsf{RHS}}), & \\
& \&_{ast}\ \textit{assign}(\mathsf{ID}, \mathsf{EXP}) \rightarrow \mathsf{STM}. & \\
\ldots & &
\end{array}$$

Figure 3.22: A frontend specification before making the remote access explicit

Compute _ _ → _	:	Symbol × Position* → Position* → Trafo
Condition _ _	:	Symbol × Position* → Trafo
Default For _ **By** _	:	Sort × Symbol → Trafo
Relate _ _ _	:	Io × Association* × Prefix → Trafo
Reduce _ **By** _	:	Sort × Symbol → Trafo
Precompute _ **By** _	:	Association × Symbol → Trafo
Construction _	:	Prefix → Trafo
Deconstruction _	:	Prefix → Trafo
Construction^{-1} _ _	:	$\mathcal{P}$(Sort) × Prefix → Trafo
Deconstruction^{-1} _ _	:	$\mathcal{P}$(Sort) × Prefix → Trafo

Figure 3.23: Elaborate computation schemata

3.4.3.1 Nontrivial computations

The basic operators **Define** and **Use** are limited to *constant computations* for providing definitions and *unary conditions* for providing uses. We generalize them so that we can cope with computations with any number of arguments and results.

Consider the transformation

$$\textbf{Compute}\ s\ \langle pos_1, \ldots, pos_n \rangle \rightarrow \langle pos_{n+1}, \ldots, pos_m \rangle$$

applied to the rule r. pos_1, ..., pos_n must be unique selectors for defining positions in r, whereas pos_{n+1}, ..., pos_m must be unique selectors for applied positions in r. Then a computational element

$$s(p_1, \ldots, p_n) \rightarrow (p_{n+1}, \ldots, p_m)$$

is inserted into r, where the p_i are the parameters corresponding to the selected parameter positions in r.

Although the **Compute** operator generalizes **Define** and **Use** in the sense that several positions can be used and defined simultaneously in a single computational element, only computations based on *unique* selectors can be modelled.

3.4.3.2 Defaults for providing definitions

A slight generalization of the operator **Define** is offered by the operator **Default**. Variable occurrences to be defined are not found by matching a position, but variables are rather found by the property to be of a certain sort σ. As for the operator **Define**, computations are only inserted for undefined variables. **Default For** σ **By** s applied to the rule r inserts a constant computation of the form $s \to v$ into r for each undefined variable v of sort σ.

Example 3.4.3
Another specification of the transformation required in Example 3.3.3 based on the operator **Default** instead of the operator **Define** is provided:

$$\textbf{Default For } \mathsf{ST} \textbf{ By } \&_{static}\ init$$

The result is the same as in Example 3.3.3. However, the approach based on the operator **Default** is slightly more abstract because it is assumed that any "undefined" variable occurrence of sort ST should be associated with a defining occurrence. There is not a dependency on particular positions of grammar symbols any longer. $\diamondsuit$

As a condition is a computational element without output positions, we can define the operator **Condition** as follows:

$$\textbf{Condition}\ s\ \langle pos_1, \ldots, pos_n \rangle = \textbf{Compute}\ s\ \langle pos_1, \ldots, pos_n \rangle \to \langle\rangle$$

3.4.3.3 Compositional computations

Now the operator **Relate** to be regarded as a high-level schema for adding computational behaviour will be discussed. Many specification problems are of a compositional nature, e.g. semantics definition, AST traversal, traversal of data structures, code generation, translations, i.e. for all rules, an output position of the defined symbol is computed from certain output positions of the premises (**Relate Output**) or input positions of premises are computed from an input position of the conclusion (**Relate Input**) respectively.

$$\textbf{Relate}\ io\ \langle\langle s_1, \sigma_1 \rangle, \ldots, \langle s_m, \sigma_m \rangle\rangle\ pfx$$

can be characterized as follows: Let us consider a rule r of the form as usual $[t]\ e_0 \Leftarrow e_1, \ldots, e_n$. Let be $pos^\star = \langle io, s_1, \sigma_1 \rangle$, ..., $\langle io, s_m, \sigma_m \rangle$. Let lhs_1, ..., lhs_k be the parameters on positions of the conclusion matching the positions $pos^\star$ respecting the order of the parameters in the conclusion (usually $k = 1$). Let rhs_1, …, rhs_q be the parameters on positions of the premises matching the positions $pos^\star$ respecting the order of the premises and the parameters in each single premise. If $k > 0$ and/or $q > 0$, then a new computational element e is inserted into r. The result of the transformation is denoted by r' below:

- $io =$ **Input**: $e =$ **Premise From** $s\ (lhs_1, \ldots, lhs_k) \rightarrow (rhs_1, \ldots, rhs_q)$, $r' = [t]\ e_0 \Leftarrow e, e_1, \ldots, e_n$,
- $io =$ **Output**: $e =$ **Premise From** $s\ (rhs_1, \ldots, rhs_q) \rightarrow (lhs_1, \ldots, lhs_k)$, $r' = [t]\ e_0 \Leftarrow e_1, \ldots, e_n, e$,

where $s =$ **Operation From** $pfx\ t$ (refer to Subsection 2.4.5). s should be a prefixed symbol because computational behaviour should be added and the skeleton should be retained. The position, where the new computational element e is inserted into the original sequence premises is indicated in order to point out the parameter dependencies; refer to Subsection C.3.5 for the actual definition of the operator.

Example 3.4.4
The following transformation shows that the operator **Relate** is useful to add computational behaviour modelling the (inherently compositional) AST construction in a frontend specification. We start from the skeleton (i.e. the underlying context-free grammar) of a simple imperative language; refer to Figure 2.14. Note that the result of the below transformation is equivalent to the projection in Figure 2.16 which models exactly the aspect of AST construction contained in the complete frontend specification originally introduced in Figure 2.9.

$$\begin{array}{llll}
\text{Figure 2.16} & \equiv & \textbf{Relate} & \textbf{Output} \\
 & & & \langle\ \langle \mathit{program}, \mathsf{PRO}\rangle, \langle \mathit{statements}, \mathsf{STM}\rangle, \langle \mathit{statement}, \mathsf{STM}\rangle, \\
 & & & \quad \langle \mathit{expression}, \mathsf{EXP}\rangle, \langle \mathit{identifier}, \mathsf{ID}\rangle \\
 & & & \rangle \\
 & & & \&_{ast} \\
 & & \textbf{On}\ \text{Figure 2.14} &
\end{array}$$

◇

3.4.3.4 Combining unused parameters

Similarly to the operator **From The Left**, which inserts copy rules to identify defining and applied occurrences of a certain sort σ from the left to the right, the operator **Reduce** 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 following computations are inserted into r:

$$s(v_1, v_2) \rightarrow v_{n+1}, s(v_{n+1}, v_3) \rightarrow v_{n+2}, \ldots, s(v_{n+n-2}, v_n) \rightarrow v_{n+n-1},$$

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 result of the transformation.

Example 3.4.5
Assume that all identifiers used in the statement part should be accumulated in order to detect superfluous variables. Attributes with sets of identifiers as associated type (sort IDS below) have to be synthesized for that purpose and for compound syntactical constructs, the accumulation can be performed by taking the union of the sets of identifiers (relational symbol $\&_{ids}$ *union*) accumulated for the subconstructs.

$$\begin{array}{ll} & \textbf{Default For } \mathsf{IDS} \textbf{ By } \&_{ids}\ \mathit{empty} \\ \circ & \textbf{From The Left } \mathsf{IDS} \\ \circ & \textbf{Reduce } \mathsf{IDS} \textbf{ By } \&_{ids}\ \mathit{union} \\ \circ & \textbf{Add } \langle\langle \textbf{Output}, \mathit{statements}, \mathsf{IDS}\rangle, \langle \textbf{Output}, \mathit{statement}, \mathsf{IDS}\rangle, \langle \textbf{Output}, \mathit{expression}, \mathsf{IDS}\rangle\rangle \\ \textbf{On} & \end{array}$$

$$\begin{array}{lll} \mathit{statements} & : \ \mathit{statement}, \mathit{statements}. & [\mathsf{concat}] \\ \mathit{statements} & : \ . & [\mathsf{skip}] \\ \mathit{statement} & : \ \mathit{expression}, \mathit{statements}, \mathit{statements}. & [\mathsf{if}] \\ \ldots & & \end{array}$$

$$\rightsquigarrow$$

$$\begin{array}{lll} \mathit{statements} \to (\mathsf{IDS}_3) & : \ \mathit{statement} \to (\mathsf{IDS}_1), & [\mathsf{concat}] \\ & \quad \mathit{statements} \to (\mathsf{IDS}_2), & \\ & \quad \&_{ids}\ \mathit{union}(\mathsf{IDS}_1, \mathsf{IDS}_2) \to (\mathsf{IDS}_3). & \\ \mathit{statements} \to (\mathsf{IDS}) & : \ \&_{ids}\ \mathit{empty} \to (\mathsf{IDS}). & [\mathsf{skip}] \\ \mathit{statement} \to (\mathsf{IDS}_5) & : \ \mathit{expression} \to (\mathsf{IDS}_1), & [\mathsf{if}] \\ & \quad \mathit{statements} \to (\mathsf{IDS}_2), & \\ & \quad \mathit{statements} \to (\mathsf{IDS}_3), & \\ & \quad \&_{ids}\ \mathit{union}(\mathsf{IDS}_1, \mathsf{IDS}_2) \to (\mathsf{IDS}_4), & \\ & \quad \&_{ids}\ \mathit{union}(\mathsf{IDS}_4, \mathsf{IDS}_3) \to (\mathsf{IDS}_5). & \\ \ldots & & \end{array}$$

◇

3.4.3.5 Interpolating computational elements

There are several forms of *inserting computational elements into rules* in order to adapt parameters of certain sorts or certain parameter positions. We use the term *interpolation* for that purpose to point out that premises are not only inserted but the data flow of the given rule is adapted as well.

There are several possibilities for interpolation. The operator **Precompute**, for example, models the insertion of precomputations for input positions of premises. Consider a premise of the following form:

$$s\ (\ldots, p, \ldots) \to (\ldots),$$

```
$\lambda$ ⟨sym, sort⟩ : **Association** . $\lambda$ by : **Symbol** .
 **Replace**
  lhsIdentity
  ( rhsForSymbol
     **On** sym
     **On** $\lambda$ e : **Premise** .
% accumulate precomputations and modified input parameters
      **Let** ⟨precomputations, psI⟩ =
       **Fold Left**
        $\lambda$ ⟨es, ps⟩ : **Premise*** × **Parameter***. $\lambda$ p : **Parameter** .
         **Sort Of** p = sort →
          **Let** fresh = **New Variable Of Sort** sort **In**
           ⟨es ++ ⟨**Premise From** by ⟨p⟩ → ⟨fresh⟩⟩, ps ++ ⟨fresh⟩⟩,
          ⟨es, ps ++ ⟨p⟩⟩
       **Neutral** ⟨⟨ ⟩, ⟨ ⟩⟩ **List Parameters Input Of** e
      **In**
% construct result of RHS substitution
       ⟨precomputations ++ ⟨**Premise From** sym psI → **Parameters Output Of** e⟩,
        ⟨ ⟩
       ⟩
  ).
```

Figure 3.24: **Precompute _ By _** : Association × Symbol → Trafo

where p is of sort σ. To insert a unary precomputation with the symbol *by* intended to adapt p, means to substitute the above premise by the following two premises

$$by\ (p) \to (v), s\ (\ldots, v, \ldots) \to (\ldots),$$

where v is a fresh variable of sort σ. The corresponding transformation is forced by the following application of the operator **Precompute**; refer to Figure 3.24 for the formal definition:

$$\textbf{Precompute}\ \langle s, \sigma \rangle\ \textbf{By}\ by.$$

Such an adaptation is useful whenever the parameter p cannot directly be used by s. The inserted computation is expected to adapt the parameter accordingly. Another approach would be to adjust the definition of s—provided it is accessible—by an operator dual to **Precompute**.

Example 3.4.6

In our running example there are only simple variable declarations so far; refer to the frontend specification in Figure 2.9. If we want to cope with constants, procedures, type definitions etc., the symbol table access becomes slightly more involved. We cannot simply associate identifiers with types any longer. We need more information classifying the actual symbol table entry. That is a typical situation, where injections and projections for a sum domain coming into being need to be inserted. Refer to Figure 3.25 for the new variants of rules dealing with symbol table entries. The following transformation can be applied to derive the new variant for [dec] Figure 3.25 from the original frontend specification in Figure 2.9.

$$\begin{array}{ll} & \textbf{Rename Positions } \{\langle \textbf{Output}, \&_{static}\ \mathit{var2entry}, \mathsf{T}\rangle, \langle \textbf{Input}, \&_{static}\ \mathit{add}, \mathsf{T}\rangle\} \textbf{ To } \mathsf{ENTRY} \\ \circ & \textbf{Precompute } \langle \textbf{Input}, \&_{static}\ \mathit{add}, \mathsf{T}\rangle \textbf{ By } \&_{static}\ \mathit{var2entry} \end{array}$$

The application of the operator **Precompute** inserts a computation

$$\&_{static}\ \mathit{var2entry}(\mathsf{T}) \rightarrow (\mathsf{T}'),$$

whereas the new sort ENTRY is established by renaming some parameter positions subsequently. The new variant of the rule [var] can be derived in a dual manner. ♢

$$\begin{array}{lcll} \mathit{declaration}(\mathsf{ST}) \rightarrow (\mathsf{ST}') & : & \mathit{identifier} \rightarrow (\mathsf{ID}), & [\mathsf{dec}] \\ & & \mathit{type} \rightarrow (\mathsf{T}), & \\ & & \boxed{\&_{static}\ \mathit{var2entry}(\mathsf{T}) \rightarrow (\mathsf{ENTRY})}, & \\ & & \&_{static}\ \mathit{add}(\mathsf{ST}, \mathsf{ID}, \mathsf{ENTRY}) \rightarrow (\mathsf{ST}'). & \\ \mathit{expression}(\mathsf{ST}) \rightarrow (\mathsf{T}, \mathsf{EXP}) & : & \mathit{identifier} \rightarrow (\mathsf{ID}), & [\mathsf{var}] \\ & & \&_{static}\ \mathit{lookup}(\mathsf{ST}, \mathsf{ID}) \rightarrow (\mathsf{ENTRY}), & \\ & & \boxed{\&_{static}\ \mathit{entry2var}(\mathsf{ENTRY}) \rightarrow (\mathsf{T})}, & \\ & & \&_{ast}\ \mathit{var}(\mathsf{ID}) \rightarrow \mathsf{EXP}. & \end{array}$$

Figure 3.25: Symbol table access coping with more than one kind of entries

The operator **Precompute** obeys some comfortable properties concluded in Proposition 3.4.1. They can be easily shown based on the definition of **Precompute** via **Replace**.

Proposition 3.4.1
$\forall s \in \mathsf{Name}, by \in \mathsf{Operation}, \sigma \in \mathsf{Sort}$: **Precompute** $\langle s, \sigma\rangle$ **By** *by* is:

- $\mathcal{WD}$-preserving,
- type-monoton increasing (increasing because *by* is possibly added),
- skeleton-preserving,
- $\mathit{compatible}_{by}$-total for $\mathit{compatible}_{by} \subset \mathsf{Rules}$ such that $\forall rs \in \mathit{compatible}_{by}$: **Sigma Of** $rs \sqcup (by : \sigma \rightarrow \sigma)$ is defined.

♢

Similarly to the insertion of *pre*computations for premises, *post*computations can be supposed. The insertion of computations for conclusions makes sense as well. The corresponding details are omitted.

We also want to comment on the relation of the operator **Precompute** to semantics preservation. The operator is not extending (refer to Definition 2.6.9). Thus, our simple syntactical criterion for disciplined transformations is not applicable. However, the operator is semantics-preserving by a specific but simple argument: If we substitute the premises inserted by **Precompute** by identity, we obtain the original rules. Thus, if the inserted premises behave like an identity for all previous applications, semantics preservation holds.

Finally, we want to comment on the orthogonality of the operator suite from a specific point of view. The usability of the operator **Hiding** for the incremental construction of premises is discussed in the following example.

Example 3.4.7
Let us assume, we want to insert a precomputation $s(p, q) \to (p')$ for a certain position *pos*. q should be regarded as an auxiliary parameter. The corresponding transformation consists of the following steps:

1. The basic precomputation $s(p) \to (p')$ is inserted with the operator **Precompute**.
2. The parameter position for q is added using the operator **Add**.
3. The auxiliary parameter has to be defined, e.g. with the operator **Define**.

A problem arises in step (1.), if there are already uses of s. Such existing uses are probably *binary* computations in contrast to the inserted *unary* computation. The intermediate result would not be defined because well-typedness would not hold.

We could try to invent a kind of precomputation operator, which simultaneously adds auxiliary positions to the precomputation. However, there are many other possible scenarios of stepwise construction of premises. Thus, it is impractical to support all such scenarios by special operators of the operator suite. We prefer to be able to unbundle roles. The operator **Hiding** provides our our generic solution for the problem of the incremental construction of premises. In the above example we simply have to hide s during the performance of the three steps. $\diamondsuit$

3.4.3.6 Terms versus computational elements

In Subsection 2.4.1 we have shown how terms in the sense of a form of compound parameters can be understood as a rather modest extension of the general framework, where we consider variables as the only form of parameters. Applications of term constructors can be turned into premises and vice versa. The advantage of such a relationship is that operations which are applicable to premises are thereby immediately useful for terms, too.

The operators **Construction**, **Deconstruction** $\_ \; : \; \mathsf{Prefix} \to \mathsf{Trafo}$ turn computational elements into terms. More precisely, the operator **Construction** (resp. **Deconstruction**) transforms the given rules by interpreting all computational elements with a given prefix as term constructors (resp. *de*constructors). The operators **Construction**$^{-1}$, **Deconstruction**$^{-1}$ $\_ \; \_ \; : \; \mathcal{P}(\mathsf{Sort}) \times \mathsf{Prefix} \to \mathsf{Trafo}$ work in the opposite direction, i.e. terms are turned into computational elements. The operator **Construction**$^{-1}$ (resp. **Deconstruction**$^{-1}$) replaces terms on applied (resp. defining) positions of the given sorts by auxiliary variables and inserts computational elements with the same shape using the term constructor together with a given prefix as relational symbol.

$$\mathit{program} \rightarrow (\mathit{prog}(\mathsf{STM})) \ \vdots\ \mathit{declarations},\ \mathit{statements} \rightarrow (\mathsf{STM}). \qquad [\mathsf{prog}]$$

...

$$\mathit{statements} \rightarrow (\mathit{concat}(\mathsf{STM}_1, \mathsf{STM}_2)) \ \vdots\ \mathit{statement} \rightarrow (\mathsf{STM}_1),\ \mathit{statements} \rightarrow (\mathsf{STM}_2). \qquad [\mathsf{concat}]$$

$$\mathit{statements} \rightarrow (\mathit{skip}) \ \vdots\ . \qquad [\mathsf{skip}]$$

$$\mathit{statement} \rightarrow (\mathit{assign}(\mathsf{ID}, \mathsf{EXP})) \ \vdots\ \mathit{identifier} \rightarrow (\mathsf{ID}),\ \mathit{expression}(\mathsf{ST}) \rightarrow (\mathsf{EXP}). \qquad [\mathsf{assign}]$$

$$\mathit{statement} \rightarrow (\mathit{if}(\mathsf{EXP}, \mathsf{STM}_1, \mathsf{STM}_2)) \ \vdots\ \mathit{expression} \rightarrow (\mathsf{EXP}),\ \mathit{statements}(\mathsf{STM}_1),\ \mathit{statements}(\mathsf{STM}_2). \qquad [\mathsf{if}]$$

...

$$\mathit{expression} \rightarrow (\mathit{var}(\mathsf{ID})) \ \vdots\ \mathit{identifier} \rightarrow (\mathsf{ID}). \qquad [\mathsf{var}]$$

...

Figure 3.26: Figure 2.16 with term construction made explicit

Example 3.4.8
Consider the result from Example 3.4.4. The actual result was shown in Figure 2.16. There are several relational formulae prefixed by $\&_{ast}$. They model AST construction. Let us "unfold" this interpretation by making the term construction explicit:

$$\text{Figure 3.26} \ \equiv\ \textbf{Construction}\ \&_{ast}\ \textbf{On}\ \text{Figure 2.16}$$

Refer to Figure 3.26 for the result. Obviously, the corresponding computational elements are discarded, but terms are substituted for the variables on their output positions.
$\diamondsuit$

The operator **Replace**, i.e. the schema for element substitution, can naturally be instrumented for both directions, that is, for turning terms into computational elements and vice versa. We only consider the direction of turning computational elements into terms. The other direction can be implemented in a dual manner. Computational elements which are intended to model term construction must have the following form:

$$s(p_1, \ldots, p_n) \rightarrow p$$

Making term construction explicit means to discard the element and to substitute the parameter p by the term $s(p_1, \ldots, p_n)$. There is a further precondition: The parameter on the output position must be a variable. Otherwise the basic concept of substitution (mapping variables to parameters) is not applicable. Figure 3.27 presents a function mapping a symbol s to an element of $\mathsf{RhsSubstitution}$ specifying how elements based on s are rewritten as term constructors. Now it is straigthforward to define the operator **Construction** from above; similarly for the other operators.

λ e : **Premise** .
Let in = **Parameters Input Of** e **In**
Let $\langle$out$\rangle$ = **Parameters Output Of** e **In**
$\langle\langle\ \rangle$, $\langle\langle$**Variable Of** out, **Term From Constructor From Operation Of** e in **Of Sort Sort Of** out$\rangle\rangle\rangle$.

Figure 3.27: Replacing computations by term construction

3.4.4 Composition

We have seen already some forms of composition, namely concatenation of rules (_ $\bowtie$ _) and superimposition (**Superimpose**). In this Subsection, we present some elaborate composition schemata; refer to Figure 3.28 for the profiles of the corresponding operators. First, a more flexible form of rule concatenation is presented (**Merge**). Second, a kind of composition facilitating the replacement of rules by other variants with the same tag is discussed (**Override**). The corresponding operator combines concatenation and selection. Afterwards, the rather simple problem of inserting keywords into rules (**Concretize**), which can be regarded as another kind of superimposition, is considered. Finally, an operator facilitating the derivation of chain rules in the sense of attribute grammars is suggested (**Chain**). Lifting (**Lift**) is the subject of a separate section; refer to Section 3.5.

Merge _ **And** _	:	Rules $\times$ Rules $\to$ Rules
Override _ **By** _	:	Rules $\times$ Rules $\to$ Rules
Concretize By _	:	(Tag $\times$ String?)* $\to$ Trafo
Chain Rule _ _ $\Leftarrow$ _	:	Tag $\times$ Symbol $\times$ Symbol $\to$ Trafo
Lift _	:	((Skeleton $\to$ Trafo)* $\otimes$ Rules)* $\to$ Rules

Figure 3.28: Elaborate schemata for composition

3.4.4.1 A relaxed form of rule concatenation

For a concatenation $rs_1 \bowtie rs_2$ to be defined means that for all symbols which have rs_1 and rs_2 in common, the corresponding profiles are equal (i.e. the LUB exists). For uniquely sorted symbols the requirement for equal profiles could be weakened by saying that there must exist a unique permutation to make the profiles equal. Thereby, rs_2 can be made compatible to rs_1 by permuting the parameterization of elements in rs_2 accordingly. The actual combination now can be performed with _ $\bowtie$ _. This additional service is provided by the operator **Merge**. Note that the property of unique sortedness is only required for symbols with different profiles in the operands. Figure 3.29 presents the specification of the operator **Merge**.

Example 3.4.9
The following two rules cannot be combined by _ $\bowtie$ _, because the profiles of *expression* in the two rules are not equal.

$$\begin{array}{lll} \mathit{expression}(\mathsf{ST}) \to \boxed{(\mathsf{T}, \mathsf{EXP})} & : & \mathit{identifier} \to (\mathsf{ID}), \\ & & \&_{\mathit{static}}\ \mathit{lookup}(\mathsf{ST}, \mathsf{ID}) \to (\mathsf{T}), \\ & & \&_{\mathit{ast}}\ \mathit{var}(\mathsf{ID}) \to \mathsf{EXP}. \end{array} \quad [\mathsf{var}]$$

$$\begin{array}{lll} \mathit{statement}(\mathsf{ST}) \to (\mathsf{STM}) & : & \mathit{identifier} \to (\mathsf{ID}), \\ & & \&_{\mathit{static}}\ \mathit{lookup}(\mathsf{ST}, \mathsf{ID}) \to (\mathsf{T}_{LHS}), \\ & & \mathit{expression}(\mathsf{ST}) \to \boxed{(\mathsf{EXP}, \mathsf{T}_{RHS})}, \\ & & \&_{\mathit{static}}\ \mathit{assignable}(\mathsf{T}_{LHS}, \mathsf{T}_{RHS}), \\ & & \&_{\mathit{ast}}\ \mathit{assign}(\mathsf{ID}, \mathsf{EXP}) \to \mathsf{STM}. \end{array} \quad [\mathsf{assign}]$$

However, since there is a unique permutation to make the profiles equal, concatenation based on the operator **Merge** is possible. The profile of *expression* is taken over from the first rule. ♢

```
λ rs1 : Rules . λ rs2 : Rules .
 rs1 ⋈
  Let t2 = Sigma Of rs2 In
   Fold Left
    λ rs0 : Rules . λ p1 : Profile .
     Let p2 = Profile Of Symbol Of p1 In t2 In
     (p2 = ?) →
      rs0,
      (Sorts Input Of p1 = Sorts Input Of p2) And
      (Sorts Output Of p1 = Sorts Output Of p2) →
       rs0,
       Permute p1 On rs0
   Neutral rs2 List Sigma Of rs1.
```

Figure 3.29: **Merge** _ **And** _ : Rules × Rules → Rules

There is an important advantage of using **Merge** instead of _ ⋈ _. The parameterization has often a different order for two operands to be composed because the corresponding aspects of computational behaviour have been possibly established in different orders. The operands cannot be concatenated, but they can be merged.

Example 3.4.10
Consider two sets of rules rs_1 and rs_2 and two parts of computational behaviour a and b. There are transformations t_a and t_b intended to add the computational behaviour of a and b, respectively. Now assume, that rs_1 only reflects a, whereas rs_2 only reflects b.

$$(t_b\ \mathbf{On}\ rs_1) \bowtie (t_a\ \mathbf{On}\ rs_2)$$

will not be defined in general, but

$$\mathbf{Merge}\ (t_b\ \mathbf{On}\ rs_1)\ \mathbf{And}\ (t_a\ \mathbf{On}\ rs_2).$$

♢

3.4.4.2 Overriding rules

A crucial problem concerning reuse is the possibility to override parts of a program. We have mentioned the operators **Undefined!**, **Unused!** and **Purge** which can be used in order to override copies and computations. Another kind of overriding, which can be regarded as a combination of two sets of rules, is discussed in the sequel.

It is assumed that some rules in a given set of rules rs_1 should be replaced by other variants contained in another set of rules rs_2. A pair of matching rules can be found by different strategies, e.g.:

1. equality of the tags,
2. equality of the skeletons,
3. existence of an unifier for the input positions of the conclusions, especially a renaming.

We only consider the first approach because it is applicable to the general framework and not only to some suitable instances as the third approach. The second approach is also generally applicable, but it has not been proved to be useful so far.

Override rs_1 **By** rs_2 replaces rules in rs_1 with tags also occurring in rs_2 by the corresponding variants in rs_2. Moreover, all the rules in rs_1 and rs_2 with tags which do not occur in both operands, are taken over to the result; refer to Figure 3.30. A more orthogonal definition w.r.t. $\_ \bowtie \_$ would ensure that rs_2 does not contain rules without a counterpart in rs_1. However, this requirement is not comfortable because then the concatenating aspect and the overriding aspect had always to be separated during composition. There is another possible option for overriding: It has to be decided if the skeleton of the rs_2 should respect the skeleton of corresponding rules in rs_1 or not. In any case, the relative order of the rules in rs_1 should be preserved; refer to Subsection 3.2.3 for sorting rules. Note that the existence of the LUB of the types of rs_1 and rs_2 is a sufficient but not necessary condition for the existence of the result of overriding.

```
λ rs1 : Rules . λ rs2 : Rules .
 Order By Tags In rs1 On Merge (Forget Tags (Tags In rs2) On rs1) And rs2.
```

Figure 3.30: **Override _ By _** : Rules × Rules → Rules

As long as there is a proper transformational relationship between two stages of a specification, overriding should not be applied because it is a rather drastic operation.

3.4.4.3 Inserting keywords

We comment on a rather trivial, but nevertheless necessary composition, that is to say the insertion of keywords. This composition can be regarded as a kind of superimposition. Keywords are assumed as a kind of premises. The corresponding operator for insertion of keywords has the following profile:

$$\textbf{Concretize By } \_ : (\mathsf{Tag} \times \mathsf{String?})^\star \rightarrow \mathsf{Trafo}$$

The parameter can be regarded as a sequence of patterns of rules. It is assumed that a question mark is superimposed with a skeleton element, whereas a proper string denotes a keyword to be inserted.

$$
\begin{array}{lcll}
\mathit{statement}(\mathsf{ST}) \to (\mathsf{STM}) & : & \text{“If”}, \mathit{expression}(\mathsf{ST}) \to (\mathsf{T}, \mathsf{EXP}), & [\mathsf{if}] \\
 & & \&_{static}\ \mathit{isBool}(\mathsf{T}), & \\
 & & \text{“Then”}, \mathit{statements}(\mathsf{ST}) \to (\mathsf{STM}_1), & \\
 & & \mathit{else}(\mathsf{ST}) \to (\mathsf{STM}_2), & \\
 & & \text{“End-If”}, & \\
 & & \&_{ast}\ \mathit{if}(\mathsf{EXP}, \mathsf{STM}_1, \mathsf{STM}_2) \to \mathsf{STM}. & \\
\mathit{else}(\mathsf{ST}) \to (\mathsf{STM}) & : & \text{“Else”}, \mathit{statements}(\mathsf{ST}) \to (\mathsf{STM}). & [\mathsf{else}] \\
\mathit{else}(\mathsf{ST}) \to (\mathsf{STM}) & : & \&_{ast}\ \mathit{skip} \to (\mathsf{STM}). & [\mathsf{noelse}]
\end{array}
$$

Figure 3.31: An optional if-construct (concrete syntax)

Example 3.4.11
Let us transform the rules concerning the if-statement with the optional else-path from Figure 3.15 to reflect a rather concrete syntax by inserting keywords. The result is shown in Figure 3.31.

$$
\begin{array}{lcll}
\text{Figure 3.31} & \equiv & \textbf{Concretize By} & \langle \quad \langle [\mathit{if}], \langle \text{“If”}, ?, \text{“Then”}, ?, ?, \text{“End-If”} \rangle \rangle, \\
 & & & \qquad \langle [\mathit{else}], \langle \text{“Else”}, ? \rangle \rangle \\
 & & & \rangle \\
 & & \textbf{On}\ \text{Figure 3.15} &
\end{array}
$$

◇

3.4.4.4 Chain rules

Context-free grammars (or skeletons) in practice contain chain rules to improve readability. In AGs, chain rules are often necessary to distinguish entities of the same structure. Introducing chain rules frequently is required during structural adaptations. The operator **Chain Rule** adds a chain rule to a given program.

Chain Rule t $lhs \Leftarrow rhs$ **On** rs has the following effect. A new rule is added to rs, where t is taken as the tag, i.e. rs must not contain a rule tagged by t and the name lhs is used to build the conclusion, whereas the name rhs is used to build the only premise. The conclusion and the premise are parameterized with the same fresh variables based on the profiles of lhs and rhs in rs. At least one of the symbols must have a profile in rs. If both have a profile, the profiles must be equal.

Example 3.4.12
Assume that a structural adaptation shall be applied to the frontend specification Figure 2.9 in order to distinguish basic expressions and compound expressions, because we want to

deal with priorities by layers of expressions as common in top-down parsing. Then a chain rule modelling "Every expression enclosed in brackets is a basic expression as well." is useful:

$$\begin{array}{lll} \mathit{basic\_expression}(\mathsf{ST}) \rightarrow (\mathsf{T},\mathsf{EXP}) & : & \text{“(“}, \qquad\qquad\qquad [\mathsf{brackets}] \\ & & \mathit{expression}(\mathsf{ST}) \rightarrow (\mathsf{T},\mathsf{EXP}), \\ & & \text{“)“}. \end{array}$$

We can establish the above chain rule by the following transformation:

$$\begin{array}{ll} & \textbf{Concretize By } \langle\langle[\mathsf{brackets}],\langle\text{“(“},?,\text{“)''}\rangle\rangle\rangle \\ \circ & \textbf{Chain Rule } [\mathsf{brackets}]\ \mathit{basic\_expression} \ \Leftarrow\ \mathit{expression} \end{array}$$

Note that the above adaptation can be performed for any profile of *expression*. In contrast, if the above rule was specified directly, a certain profile would be assumed. ♢

3.5 Composition by lifting

Lifting is a kind of composition of program fragments (more precisely rules) and program transformations modelling computational behaviour. There can be several "packets" of rules to be lifted. Each of them covers certain computational aspects, where the remaining aspects are expected to be established by the corresponding transformations. Concatenation, superimposition and overriding are involved in the complete process of lifting as more basic schemata of composition.

In Subsection 3.5.1 a detailed but informal and abstract view on lifting is outlined, before a certain variant of lifting is formalized as the operator **Lift** in Subsection 3.5.2. The whole section is partially based on our previous work presented in [Läm97, LR97]. As there are several options for instantiating the notion of lifting and there are some ideas how to go beyond the variant incorporated into the operator suite, we close the section with a discussion in Subsection 3.5.3.

3.5.1 Notions

We need a number of basic notions which are suitable to derive finally the notion of lifting.

Complete program In program composition (or in general in program development) we are interested in complete programs, that is a program is required to solve a certain task or to perform certain computations. According to our examples and the used formalisms, complete programs are type checkers, program simplifiers, interpreters, etc.

Computational aspect Programs are assumed to be semantically structured according to computational aspects. To manage the complexity of a program, it is crucial to identify these aspects.

Example 3.5.1
For the frontend specification in Figure 2.9 we can think, for example, of the following atomar aspects:

1. Terminal attribution for identifiers
2. Representation of type expressions
3. Accumulation of the symbol table entries in the declaration part
4. Initialization of the symbol table
5. Propagation of the symbol table in the statements part
6. Representation of operator symbols
7. Type synthesis for expressions
8. Context conditions for statements
9. Compositional computations for AST construction

◇

The idea is to represent computational aspects by program transformations what will be clarified below when the notion *transformer* is introduced. Note that in semantics, particularly in (modular) denotational semantics, a computational aspect is often referred to as a *semantic aspect*.

Level of the computational model Considering all possible sets of computational aspects we obtain a space of levels, a lattice-like structure. Among these compositions there are probably some which are conceptionally particular important because they correspond to subproblems of the complete program. The space of levels corresponds to the first dimension of a complete program in our approach.

Example 3.5.2
For the frontend with the aspects given in Example 3.5.1, the following meaningful levels can be identified:

- The static semantics corresponds to the level composed from the aspects (1.), ..., (8.). Note that (7.) and (8.) can be regarded as the primary aspects, whereas the remaining aspects are needed to specify type checking of expressions and context conditions of statements for some forms.
- The actual AST construction is located at the level consisting of the computational aspects (1.), (6.) and (9.). Note that the aspects (1.) and (6.) are secondary in the sense that they supply some parameter positions contributing to the ASTs. The actual compositional AST construction is modelled by (9.).

◇

Note that in semantics, particularly in (modular) denotational semantics, a level is often referred to as a *level* (layer) *of the semantic model.* We use the term *computational model* to denote the computational level of the complete program.

Skeleton Recall that the space of computational levels corresponds to the first (a more semantical) dimension for complete programs. The second (a more structural) dimension is provided by the power set of skeleton rules of the complete program. Remember that a skeleton is simply a set of non-parameterized rules; refer to Figure 2.14 for the skeleton of our running example. One can think of a context-free grammar, a signature, or a data type description as well. The connection between skeletons and computational aspects / levels is the assumption, that the complete program is based on a certain skeleton which is constant for intermediate stages of composing and synthesizing the computational behaviour of the complete program.

Rules at levels One can speak of rules at certain levels according to the computational aspects covered by them as manifested by the computational behaviour of the rules. Note that skeleton rules are at the empty level of computational aspects, whereas the complete program is at the level of the computational model.

$$\begin{array}{lll}
\mathit{declaration}(\mathsf{ST}_0) \rightarrow (\mathsf{ST}_1) \;:\; & \mathit{identifier} \rightarrow (\mathsf{ID}), & [\mathsf{dec}] \\
& \mathit{type} \rightarrow (\mathsf{T}), & \\
& \&_{static}\ \mathit{add}(\mathsf{ST}_0, \mathsf{ID}, \mathsf{T}) \rightarrow (\mathsf{ST}_1). & \\[6pt]
\mathit{statement}(\mathsf{ST}) \;:\; & \mathit{identifier} \rightarrow (\mathsf{ID}), & [\mathsf{assign}] \\
& \&_{static}\ \mathit{lookup}(\mathsf{ST}, \mathsf{ID}) \rightarrow (\mathsf{T}_{LHS}), & \\
& \mathit{expression}(\mathsf{ST}) \rightarrow (\mathsf{T}_{RHS}), & \\
& \&_{static}\ \mathit{assignable}(\mathsf{T}_{LHS}, \mathsf{T}_{RHS}). & \\[6pt]
\mathit{statement} \;:\; & \mathit{expression} \rightarrow (\mathsf{T}), & [\mathsf{if}] \\
& \&_{static}\ \mathit{isBool}(\mathsf{T}), & \\
& \mathit{statements}, & \\
& \mathit{statements}. & \\[6pt]
\mathit{expression}(\mathsf{ST}) \rightarrow (\mathsf{T}) \;:\; & \mathit{identifier} \rightarrow (\mathsf{ID}), & [\mathsf{var}] \\
& \&_{static}\ \mathit{lookup}(\mathsf{ST}, \mathsf{ID}) \rightarrow (\mathsf{T}). &
\end{array}$$

Figure 3.32: Some rules at certain levels of the computational model

Example 3.5.3
Consider, for example, the rules in Figure 3.32. We explain the level of the rule [if] with regard to the aspects in Example 3.5.1. Aspect (8.), i.e. context conditions of statements, has to be instantiated for *if*-statements as follows: The expression serving as a condition of the *if*-statement must be of the Boolean type. Therefore, [if] is at the level composed from (8.) and (7.) because we need the synthesized attribute for the type of an expression in order to specify the context condition. $\diamondsuit$

Irrelevance and contribution The advantage of using rules at certain levels is that we can go along our two dimensions addressing parts of the complete problem with the focus on certain skeleton rules. Usually we can abstract from details. The corresponding aspects are called *irrelevant* (w.r.t. the part of the complete problem and some skeleton rules).

Example 3.5.4
Aspects (1.), (2.), (3.), (4.), (5.), (6.) and (9.) are irrelevant for the context condition of an *if*-statement. Note that there are statements whose context conditions possibly require other aspects. To specify the usual context condition for assignment, for example, the aspects (1.) and (5.) are needed as well. ♢

An aspect is said to *contribute* to the computational behaviour of a certain skeleton rule, if the corresponding rule of the complete program has some computational behaviour which can be associated with the aspect. Note that irrelevance w.r.t. the complete computational model can be regarded as the opposite of contribution.

Example 3.5.5
Although aspects (1.), (2.), (3.), (4.), (5.), (6.) and (9.) are irrelevant for the context condition of an *if*-statement, some of these aspects contribute to the computational behaviour of [if], namely (5.) and (9.) because the rule contributes to the propagation of the symbol table and AST construction needs also to be performed; refer to the rule [if] in Figure 2.9. ♢

Superimposition and contraction Given two rules based on the same skeleton rule which are intended to describe different parts of the computational behaviour for the skeleton rule, they can be composed by superimposition in the sense of the corresponding operator **Superimpose**; refer to §3.3.3.1. A contraction of the parameterization can be necessary, if the rules have some assumption about the parameterization in common; refer to the operator **Contract** in Subsection 3.3.1.

Completeness and consistency Let us consider the completeness and consistency of a collection C of sets of rules at certain levels with regard to some skeleton and some computational model in the following way: For each skeleton rule, the corresponding rules in C cover exactly all aspects contributing to the computational model for this skeleton rule. Note that this characterization does not take program transformations into consideration yet.

Transformer relating levels Rules at certain levels are one possible way to represent computational behaviour. We suggest program transformations adding parameterization, computational elements and adapting computational behaviour as another form. A *transformer* is a program transformation t intended to model an aspect a. Given a rule r at a certain level (i.e. a skeleton rule in the trivial case) we can add the aspect a to r by applying t on r. For some skeleton rules, t will not be required because there is possibly a corresponding rule at a certain level covering a. Transformers should be skeleton-preserving and semantics-preserving. Completeness can be relaxed by saying for each skeleton rule a contributing aspect must be either covered by a rule in C or there must be a suitable transformer. Note also that a transformer sometimes needs to be restricted as far as it concerns the level of rules which the transformer is applicable to.

Example 3.5.6
Some aspects from Example 3.5.1 can be modelled by transformers as follows:

- (3.): **Accumulate** ST **From** $\{program\}$ **To** $\{declaration\}$
- (4.): **Default For** ST **By** $\&_{static}$ *init*
- (5.): **Inherit** ST **From** $\{program\}$ **To** $\{expression\}$
- (9.): **Relate Output** $\langle\langle program, \mathsf{PRO}\rangle, \langle statements, \mathsf{STM}\rangle, \langle statement, \mathsf{STM}\rangle, \langle expression, \mathsf{EXP}\rangle, \langle identifier, \mathsf{ID}\rangle, \ldots\rangle\ \&_{ast}$

Note (4.) should not be applied to a rule at a level which does not contain (3.), where (3.) is an aspect contributing to the rule. $\diamondsuit$

Lifting is the process of deriving a complete program from:

1. a skeleton,
2. a computational model,
3. a collection of sets of rules at certain levels,
4. a set of transformers.

Possibly, the skeleton and the computational model can be regarded as implicitly described by the other two ingredients.

Example 3.5.7
The frontend specification in Figure 2.9 is obtained by lifting the rules at certain levels from Figure 3.32 using the transformers in Example 3.5.6. Note that we need less rules at levels than final GSF rules. Note also that the rules at the levels are not so complex, since they abstract from irrelevant aspects. $\diamondsuit$

This abstract characterization of lifting can be put in concrete form in different ways including the possibility to enrich the process by features helpful for program composition or adaptation.

3.5.2 A concrete form

Now we want to present a concrete form of lifting which proved to be useful in our work concerning the composition of language processors from reusable fragments [LRBS]. The operator suite supports that form by the operator **Lift**; refer to Figure 3.33 for the definition.

The input of the operator **Lift** is a sequence $\langle p_1, \ldots, p_n\rangle$ of so-called *parts*, whereas the output is a set of rules. Each part has the following structure:

$$\underbrace{(\mathsf{Skeleton} \to \mathsf{Trafo})^\star}_{\substack{\text{transformers}\\ \text{to be applied to the rules}}} \quad\otimes\quad \underbrace{\mathsf{Rules}}_{\substack{\text{rules}\\ \text{at a level}}}$$

```
$\lambda$ parts : ((**Skeleton** $\rightarrow$ **Trafo**)* $\times$ **Rules**)* .
% compute skeleton and override
 **Let** $\langle$sk, overridden$\rangle$ =
  **Fold Right**
   $\lambda$ $\langle$ts, rs$\rangle$ : (**Skeleton** $\rightarrow$ **Trafo**)* $\times$ **Rules** .
   $\lambda$ $\langle$skSofar, partsSofar$\rangle$ : **Skeleton** $\times$ ((**Skeleton** $\rightarrow$ **Trafo**)* $\times$ **Rules**)* .
    **Let** tags = **Map** $\lambda$ $\langle$t, l, r$\rangle$ : **Shape** . t **List** skSofar **In**
     **Let** rest = **Forget Tags** tags **On** rs **In**
      $\langle$(**Skeleton Of** rest) $\bowtie$ skSofar, $\langle\langle$ts, rest$\rangle\rangle$ ++ partsSofar$\rangle$
  **Neutral** $\langle\langle\ \rangle, \langle\ \rangle\rangle$ **List** parts
 **In**
% iterate the overridden parts
  **Fold Left**
   $\lambda$ sofar : **Rules** . $\lambda$ $\langle$ts, rs$\rangle$ : (**Skeleton** $\rightarrow$ **Trafo**)* $\times$ **Rules** .
    **Order By Tags In** sofar **On**
     **Merge** sofar
     **And**

% apply transformers to rules of this part
      **Fold Left**
       $\lambda$ rsSofar : **Rules** . $\lambda$ t : **Skeleton** $\rightarrow$ **Trafo** .
        t **On** sk **On** rsSofar
      **Neutral** rs **List** ts
  **Neutral** $\langle\ \rangle$ **List** overridden.
```

Figure 3.33: **Lift** _ : ((Skeleton $\rightarrow$ Trafo)* $\otimes$ Rules)* $\rightarrow$ Rules

The second projection can be regarded as a set of rules at a certain level, whereas the first projection enumerates the transformers, each of the type Skeleton $\rightarrow$ Trafo to be applied to the rules in order to establish the complete computational model for these rules. The type of transformers reflects that a transformer is assumed to observe the skeleton of the complete program, what is important, for example, for any kind of propagation. Besides the skeleton parameter, a transformer is simply a function on Rules.

The actual lifting is performed with regard to the skeleton of the rules of all parts. The computational model is implicitly defined by the parts. Note that there can be aspects without a corresponding transformer if the rules of all parts are at levels already containing this aspect. This case is even very common because there are often aspects which are so central that all fragments cover these aspects, e.g. type synthesis for expressions in a frontend definition or evaluation of expressions in an interpreter definition.

Besides the formal definition of the operator **Lift** in Figure 3.33, an informal explanation is provided here as well. Consider an application of **Lift** of the following form:

$$\textbf{Lift}\ \langle p_1, \ldots, p_n \rangle.$$

Each part p_i consists of transformers $\langle t_{i,1}, \ldots, t_{i,m_i} \rangle$ and of some rules rs_i. From all the rs_i a skeleton can be obtained. The basic strategy is the concatenation of the skeletons of the parts. The operator **Lift** follows a more general approach in the sense that overriding is integrated in lifting, i.e. a set of rules rs_i can override rules of any rs_j with $j < i$ based on tags as usual. Thus, the skeleton is accumulated from backwards (**Fold Right** ...) and only those rules of the skeleton of rs_i are incorporated which have a tag not occurring in the skeleton accumulated so far. Simultaneously, the parts are minimized to fade out of those rules which are to be overridden. Now the composition of the complete programs is performed as an iteration on minimized parts, where the rules of the actual part are lifted by the associated transformers observing the accumulated skeleton.

Example 3.5.8
We explain the lifting process for the frontend specification in Figure 2.9. We use (3.), (4.), (5.) and (9.) to denote the transformers from Example 3.5.6 associated with computational aspects from Example 3.5.1.

$$
\begin{array}{lllll}
\textbf{Lift} & \langle & & & \\
 & & \langle & \langle (3.), (4.), (5.), (9.) \rangle, & \text{Figure 2.14} \quad \rangle, \\
 & & \langle & \langle \rangle, & \text{Figure 3.32.}[\mathsf{dec}] \quad \rangle, \\
 & & \langle & \langle (9.) \rangle, & \text{Figure 3.32.}[\mathsf{assign}] \quad \rangle, \\
 & & \langle & \langle (5.), (9.) \rangle, & \text{Figure 3.32.}[\mathsf{if}] \quad \rangle, \\
 & & \langle & \langle (9.) \rangle, & \text{Figure 3.32.}[\mathsf{var}] \quad \rangle, \ldots \\
 & \rangle & & &
\end{array}
$$

Note that Figure 2.14 refers to the *complete* skeleton of the frontend specification. It would be slightly more precise to fade out rules with tags occurring in subsequent parts, but that is not necessary because the operator **Lift** performs such a minimization anyway. ♢

As pointed out above, transformers need to be skeleton-preserving. If structural adaptations have to be performed on the operands of lifting, they must already be manifest in rules in the parts. Thus, lifting will be based on the modified structure.

3.5.3 Discussion

The operator **Lift** does not consider parts with overlapping skeleton rules. For more complex computational models, the approach outlined in the abstract view might be preferred, i.e. there can be any number of rules with the same underlying skeleton. These rules, which usually are concerned with different computational aspects, must first be combined by superimposition. Contraction may be necessary to identify parameterization due to common assumptions about the computational aspects. Afterwards, the remaining aspects can be added by transformers. In contrast to the operator **Lift**, each set of rules has to be associated with the aspects rather *covered* by the rules in the sense of a static type information than *to be applied* to the rules. Moreover overriding has to be realized explicitly, e.g. by applying the operator **Forget** to the rules before giving them as arguments to the lifting process.

Let us formalize the approach of having several rules at levels per skeleton rule. The input has the following structure:

$$\overbrace{(\mathsf{Skeleton} \rightarrow \mathsf{Trafo})^\star}^{\text{transformers}} \otimes \overbrace{(\underbrace{\mathcal{P}(\mathcal{N})}_{\substack{\text{to index}\\ \text{transformers}}} \otimes \underbrace{\mathsf{Rules}}_{\substack{\text{rules}\\ \text{at a level}}})^\star}^{\text{parts}}$$

Aspects with an associated transformer are enumerated by the first projection. The second projection somehow corresponds to the parts from above, but here rules are associated with indices indexing the transformers from first projection saying which of those aspects are already covered by the rules of the parts. That is in contrast to the form of the parts assumed for the operator **Lift**, where each part enumerates the aspects *to be applied*.

Example 3.5.9
We associate the rules in Figure 3.32 with the corresponding aspects:

[dec] : (1.), (2.), (3.)
[assign] : (1.), (5.), (7.), (8.)
[if] : (7.), (8.)
[var] : (1.), (5.), (7.)

♢

Let $\langle\langle t_1, \ldots, t_n\rangle, \langle p_1, \ldots, p_m\rangle\rangle$ be an input for the lifting process with the skeleton sk, each p_i has the following structure: $\langle\{n_{i,1}, \ldots, n_{i,k_i}\}, rs_i\rangle$. To consider an input as valid means the following:

1. $1 \leq n_{i,j} \leq n$ for $i = 1, \ldots, m, \; j = 1, \ldots, k_i$
2. $\nexists i, j : 1 \leq i, j \leq m, \; i \neq j \; \wedge$
   $\textbf{Tags In } rs_i \cap \textbf{Tags In } rs_j \neq \emptyset \wedge \{n_{i,1}, \ldots, n_{i,k_i}\} \cap \{n_{j,1}, \ldots, n_{j,k_j}\} \neq \emptyset$

The first condition ensures that indices in the parts are valid indices in the list of aspects (transformers). Note that due to the structure of the parts, no aspect is referred to more than once in a part anyway. The second condition concerns the problem when rules of parts with an overlapping skeleton have aspects in common. Such an ambiguity must possibly be regarded as inconsistency because there are possibly multiple (contradictory) definitions for some parameter positions. However, there is a solution to this problem explained below.

Lifting in the more general form starts from a skeleton sk and it is performed for each single skeleton rule as follows:

1. For every single skeleton rule we first lookup all corresponding rules in the parts. These rules are denoted by r_1, ..., r_q.
2. r_1, ..., r_q are superimposed:

$$\textbf{Superimpose } r_1 \textbf{ And } (\textbf{Superimpose } r_2 \textbf{ And } (\cdots \textbf{ And } r_q)\cdots)$$

3. The result of the superimposition is transformed by:

$$(t_{d_w} \textbf{ On } sk) \circ \cdots \circ (t_{d_1} \textbf{ On } sk),$$

   where the $d_1, \ldots, d_w$ are the indices of aspects not covered by $r_1, \ldots, r_q$.
4. The separately lifted rules are merged.

There is another option in steps (2.) and (3.): The composed transformation mentioned in (3.) could also be applied to the corresponding skeleton rule and the result will be superimposed with the superimposition from the previous step. However, this option ignores that transformers are likely to adapt rules at certain levels, i.e. that some aspects must be present before a certain transformer can be applied. Consequently we should even assume an order of applying transformers in step (3.). For simplicity, the following strategy is assumed:

- Transformers are applied as late as possible, i.e. first the rules are superimposed and the transformers are applied to that intermediate result. This issue is reflected in (2.) and (3.)
- The order of the transformers in the first projection of the input is regarded as a reference and step (3.) can easily be adjusted to preserve that order by adding the requirement $d_1 < d_2 < \ldots < d_w$.

The second condition for an input to be valid can be relaxed. Suppose two parts with an overlapping skeleton have aspects in common. A rule from the intersection of the skeletons can still be superimposed as described in the second step of lifting above, but the parameterization according to the common aspects must be unified. That is easy to perform by contraction in the superimposed intermediate result. There are a number of problems with this approach.

- When defining positions are contracted, variables with multiple defining occurrences are obtained. That is not always acceptable, although it makes sense for some target languages.
- It is not obvious how to determine the parameter positions to be contracted. In general, declarations about the parameterization associated with aspects would be needed. It also must be assumed that every part enumerates *exhaustively* the aspects covered by the rules. On the other hand there is a pragmatic strategy if unique sortedness for skeleton elements is assumed. After each superimposition a contraction can be performed so that unique sortedness is recovered.

The discussion of lifting is concluded by pointing out some remaining problems making clear that this topic is worth to be considered further:

Well-definedness A rule is regarded as well-defined if all applied occurrences can be associated with at least one or exactly one defining occurrence. It is not clear if well-definedness of the rules in the parts should be required? If the adaptation and the initialization of a data structure, for example, are regarded as separate aspects, non-well-defined rules make sense because the fragment dealing with adaptation can rely on a separate initialization. Mostly, well-definedness is useful.

Type checking It is easy to observe that lifting (especially the operator **Lift**) as described above is partial, even if the transformers are total. During superimposition and concatenation type conflicts can occur. It is not so obvious how to approach to a kind of type checking for the input of lifting. One approach is to enumerate all aspects *exhaustively* and to describe the contribution of each aspect to the parameterization (e.g. in terms of parameter positions) and to the computational behaviour (e.g. in terms of profiles of relational symbols). It could be checked then if rules from the parts are well-typed. It would remain to prove that the transformers actually adhere to the declared contribution.

Rules versus transformers The operator **Lift** supports only one rule at some level per skeleton rule. The remaining computational behaviour must be added by transformers. In principle, this is always possible due to the expressive power of the calculus for transformations. One extreme for the style of a transformer is that it adds computational behaviour following a completely uniform schema. Another extreme is that it modifies only a certain rule or it describes a case distinction on the rules, where rules at levels are possibly more appropriate. At this point it is not clear how to decompose complex programs in terms of rules at levels and transformers.

Overriding The operator **Lift** incorporates smoothly overriding of rules into the process of lifting. It is not obvious how to perform such an amalgamation for other approaches to lifting. Besides overriding rules, overriding computational behaviour is possibly useful, too.

Chapter 4

Related work

We want to understand how reuse is facilitated in other specification frameworks and particular problem domains (e.g. formal semantics specification). For some of the manipulations provided in other frameworks and domains we want to attempt a reconstruction based on our meta-programming-like point of view. The benefit of such a reconstruction is that the underlying concepts are made available for other instances of our framework. For some approaches we are able to identify particular weaknesses and limitations.

First, the scope of related work covered by this chapter is explained in Section 4.1. Second, paradigm shifts in attribute grammars are compared with our meta-programming approach in Section 4.2. Some of the paradigm shifts can be simulated in our framework. Third, sophisticated approaches to reusability in semantics are discussed in Section 4.3. The most promising attempts in semantics are not directly applicable to the target languages in our work. Nevertheless, we will try to identify the limitations of the corresponding attempts and to make some use of the corresponding concepts in our context. Finally, several approaches belonging to the field of formal program development are outlined in Section 4.4.

4.1 Scope

When I started my research presented in the thesis in early 1995, I was interested in compiler compilers, particularly based on attribute grammars and formal semantics, particularly, denotational semantics. The very rough goal I had in mind was to provide support for reuse based on operations on (attribute grammar and/or semantics) specifications. Reuse is too often based on "text editing". My operations should facilitate a formal way of reuse. Moreover, reuse should be executable in contrast to several other meta-level approaches, e.g. refinement. Consequently, I have dedicated two sections on improvements of the basic attribute grammars paradigm (with the emphasis on any kind of modularity) (Section 4.2) and on extensibility in semantics (Section 4.3).

Even at the beginning of my research I was aware of modularity concepts in programming languages, including declarative programming languages. Modularity in the common sense essentially supports programming in the large by decomposition and parametricity. This overall approach emphasizes *design for reuse in advance*. In principle, I agree to the suitability of that premise, but I wanted to look beyond the border of this restriction to reuse. What kind of reusability can be achieved by adaptations based on a transformational point of view?

As far as I can see, there are two major problems with modularity in the common sense:

- An insufficient decomposition and parameterization makes reuse impossible. Thus, the decisions about the actual decomposition and the parameters including the assumptions about the parameters are very critical. On the contrary, transformations can adapt, in principle, “any” given program. In particular, a transformation may even install a parameterization in a given input program. Thus, the long term goal of my study is to show that transformations may improve reusability.
- Another problem concerns the overhead for establishing a sufficient decomposition and parameterization and for realizing proper instantiations. I want to consider several powerful techniques, e.g. monads [Wad92] or object-orientation in functional programming [SA97], as “coding techniques”. Again, transformations might be more appropriate in some cases, since the properties for their applicability and their effect are easier to understand.

Because of these limitations, common approaches to modularity will be commented on in this chapter only to a limited extent. As my project proceeded, I became more familiar with methods of formal program development, such as program transformation, program synthesis, program refinement, mainly in the context of logic programming. Section 4.4 reports on related work in this area.

The scope of the related work chapter covers compiler construction, extensibility in semantics, program transformation and refinement, operations on specifications and some more almost unrelated fields. It was my intention to cover such a wide spectrum, although my results could possibly be stated for one or another particular community. With that commitment to such a wide spectrum, some related work will not be commented on in depth, including the following approaches:

- Meta-programming uses the meta-level to define classes of target programs. In higher-order functional programming certain operators like *map* / *foldl* / *foldr* are used in conjunction with polymorphism to describe classes of algorithms. Shapes and polytypism [JC94, JJ96, JJ97, Jeu95] lead us even a step further in the degree of abstraction. The idea of *map*, for example, can be applied to any algebraic type such as trees and matrices. Thereby, we obtain a generic *map* when applied to a data structure of a certain *shape* returns a data structure of the same shape.

- Representing a whole class of computations on a particular data structure by means of suitable higher-order predicates has been suggested for example by L. Naish [Nai96]. Essentially, Naish argues for a higher order approach to programming in Prolog based on similar techniques widely used in functional programming. That approach depends on impure features of Prolog. A similar but more abstract and formal approach to higher-order predicates is taken by J.F. Nilsson and A. Hamfelt [HN95, HN96, NH95]. We should mention another paper [NS97] by Naish and Sterling, where they apply higher-order logic programming in Prolog for a kind of higher-order reconstruction of stepwise enhancement which is described in some detail in Subsection 4.4.2.

- 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 [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 that of Demeter in that transformations are independent from the actual skeleton and how computational behaviour (including propagation based on the notion of reachability) can be established in concrete target programs.

- Aspect-oriented programming [KLM+97] is a very recent programming technique which claims to support the separation and composition of aspects (design decisions and others). Thereby, it can be avoided that "tangled" code arises from the fact that certain design decisions cross-cut the system's basic functionality. The technique is based on a very general view on procedural programming (including object-oriented programming), where special language support is added for the development of aspect code. [KLM+97] introduces the central notions *component* and *aspect* as follows: *With respect to a system and its implementation using a general procedure-based language, a property that must be implemented is:*

  - *a component if it can be cleanly encapsulated in a procedure, a method, an object, or an API; components tend to be units of the system's functional decomposition,*
  - *an aspect, if it cannot be cleanly encapsulated in such a way; aspects tend to be properties that effect the performance or semantics of the components in systemic ways.*

  Although aspect-oriented programming so far has been formulated in the imperative paradigm, a distinction between components and aspects is similar in intent to our notions of skeleton rules and computational aspects as proposed in Section 3.5 on lifting. The actual choice of an aspect language, i.e. the language used for the description of aspects, depends on the nature of the aspects. One example given in [KLM+97] brings us very closely to meta-programming: An aspect dealing with optimization is expected to operate on the data flow graph of a component program. Furthermore, the component programs and the aspect code are compiled into a complete program based on a technique called *weaving* which again—at a superficial level—corresponds to our lifting. The main difference between the two approaches is, that we are concerned with declarative programs and that we have a very detailed methodology for meta-programming and lifting instead of a rather abstract proposal for aspects and weaving.

- There are various further language extensions whose expressive power should be compared with our meta-programming approach, e.g. multi-stage programming suitable for expressing staged computations explicitly [NN92, TS97], and mixins in object-oriented programming [DS96, Bra92, BL92].

4.2 Extension of the AG formalism

There are several surveys on (extensions of) the attribute grammar formalism, e.g. [Bau98, Ada91, KW94, Boy96b, Paa95]. We also want to refer to Parigot's complete bibliography on attribute grammars[1] and Attribute Grammar Page[2]. For reasons of economy, we will comment here only on some specific paradigm shifts, namely:

- object-orientation (Subsection 4.2.1),
- remote access (Subsection 4.2.2),
- symbol computations (Subsection 4.2.3),
- coupling (Subsection 4.2.4),
- patterns (Subsection 4.2.5),
- actual features of AG systems using *FNC-2* as an example (Subsection 4.2.6).

We regard several other approaches as beyond our scope, in particular the style of object-orientation in Koskimies' et al. system Tools [Kos91], the non-declarative features of Hedin's Door AGs [Hed91, Hed92], the "unification" of syntax and semantics in Swierstra's and Vogt's Higher-Order AGs [SV91]. For a survey on approaches with the emphasis on modularity we recommend Baum's thesis [Bau98]. Simpler forms of modularity are for example provided by different instances of hierarchical/functional decomposition of AGs. Watt's partitioned AGs [Wat75] and Ganzinger's signature morphisms [Gan83] can be regarded as sophisticated approaches to modularity.

4.2.1 Object-orientation

4.2.1.1 Motivation

There are different approaches to incorporate object-oriented notions into attribute grammars. [Kos91] provides a survey on this subject. Refer also to [Paa95]. Besides the pragmatic aims to shorten the notation and to improve readability, there are essentially the following motivations for such extensions:

1. A benefit of object orientation is that it supports reusing existing code. Computational behaviour can be specified somewhere in the class hierarchy. The behaviour

[1]http://www-rocq.inria.fr/oscar/www/fnc2/AGabstract.html
[2]http://www-rocq.inria.fr/oscar/www/fnc2/attribute-grammar-people.html

is then inherited to descendant classes, where it can possibly be adapted. This motivation arises from the view in Smalltalk, for example.

2. Inheritance can also be used to structure domain-specific frameworks. Thereby, application-oriented software is supported. This motivation arises from the view in Simula, for example.
3. The AG formalism is an open formalism and not a complete specification language. It is common to look for specification language features in order to improve the pragmatic properties of AG specification.
4. Object-oriented notions like state of an object and message passing can be used to extend the AG formalism with explicit dynamic capabilities.

We will ignore the forth point completely in the following consideration, because the kind of dynamic capabilities goes beyond our purely declarative framework. There are other approaches to the extension by dynamic capabilities, e.g. Dynamic Attribute Grammars [PRJD96a, PRJD96b], which are more appropriate in our context.

The remainder of this subsection will deal with object-orientation based on object-oriented context-free grammars. Note that Section 4.2.3 reports on paradigm shifts in *Lido* based on another kind of inheritance which is almost independent from the underlying CFG.

4.2.1.2 Object-oriented context-free grammars

An object-oriented view of attribute grammars can be based on an object-oriented view of the underlying concept, i.e. CFGs; refer to Figure 4.1. Thus, chain productions $A \to B$ can be regarded as the definition of a class system (refer to Section A.4 for technical details), whereas a production $A \to B_1 \dots B_n$ can be regarded as structural description, i.e. an object of class A has attributes of static classes B_1, ..., B_n. To be sensible from the object-oriented point of view, all alternatives for a given nonterminal A are either chain productions or there is only one production giving a structural specification.

OO	**CFG**
class	nonterminal
object	a (sub)word derived from a nonterminal
structural specification of an object	production
superclass/subclass relation	chain productions

Figure 4.1: Object-oriented notions for CFGs

From a syntactic point of view reduced CFGs are common. However, from the point of view of object-oriented AGs, it is useful to allow nonterminals n_i which are not reachable from the axiom of the CFG. These n_i model semantic base classes. Behaviour can thus be inherited to a nonterminal (class) n by a chain production $n_i \to n$. Consequently, the class system of an object-oriented AG is mainly obtained by chain productions of the underlying syntax description possibly extended by chain productions with non-reachable symbols on the LHS corresponding to semantic classes. Note that CFGs with multiple inheritance cannot effectively be used for object-oriented AGs as explained in more detail below.

Refer to Section A.5 for a formal definition of object-oriented CFGs including examples.

4.2.1.3 Attribute inheritance and default values

An ordinary AG associates a set of synthesized and inherited attributes with each symbol. Each syntactic rule must be associated with semantic rules defining the synthesized attributes of the symbol on the LHS and the inherited attributes of the symbols on the RHS. In order to avoid confusion concerning the meaning of the term inherited attribute, we adhere to the Mjølner/Orm terminology to use the term *ancestral* attribute instead. In the following inheritance is only used in the sense of object-orientation.

Essentially, object-orientation for AGs is an extension of the basic AG paradigm by inheritance of attributes and semantic rules, where the underlying CFG must obey single inheritance. Inheritance of attributes is not very effective, especially if we take into consideration that the existence of the corresponding chain productions including the auxiliary nonterminals (to have exactly one structural specification per nonterminal) is almost a consequence of the required form of object-oriented CFGs. Without any further extensions (such as rule models discussed below) inheritance of semantic rules does not give not much expressive power. The RHSs of semantic rules associated with a nonterminal n to be regarded as a superclass can only depend on ancestral attributes of n itself. The following example taken from [KLMM93] shall illustrate the concept of inheritance of semantic rules.

Example 4.2.1
There are sometimes proper defaults for synthesized attributes in the sense that only a few subclasses have to define a different value, i.e. the inheritance of the default value is useful. Consider, for example, the following extension of the class Exp modelling any kind of expressions. Checking contextual constraints, we need to detect expressions which are proper forms for LHSs of assignments. Thus, a synthesized attribute hasLeftValue is suitable for that purpose. For several forms of expressions, e.g. constants, monadic and dyadic arithmetic expression, the following default formalized in the notation of Mjølner/Orm is correct:

```
addto Exp
{
  syn hasLeftValue : Boolean;
  hasLeftValue := false;
}
```

By the way, using **addto** construct of *Mjølner/Orm*, attributes and corresponding semantic rules can be added. Thereby, the semantics decomposition of a specification similar to phases in *OLGA* of *FNC-2* [JP91, JP90, Par88, JPJ$^+$90] is supported. However, note that this feature should not be regarded as an object-oriented feature because the extension is not coupled with inheritance. ◇

Refer to Section A.6 for some more samples of object-oriented AGs.

4.2.1.4 Models of semantic rules

Without adding further concepts like, for example, rule models, I claim that object-orientation in AGs does not improve modularity significantly. A collective equation [Hed92, p. 82], [KLMM93, p. 472], or a rule model

$$\begin{array}{l}\textbf{for all sons}(x)\ \textbf{in}\ \mathit{class}\\ \quad\textbf{son}(x).a_1 := f(a_2, \ldots, a_n)\end{array}$$

defines the value of an inherited attribute a_1 for all son nodes of a given *class*. The concept of collective equations provides one possibility for defining general behaviour at suitable levels of generalization with regard to the class hierarchy. A rule model is not dedicated to a certain syntactical rule. This flexibility is possible because a rule model does not depend on the exact number and the types of sons.

Example 4.2.2
Let us consider a part of the static semantics of a block-structured language. We are actually concerned with the symbol table propagation. The symbol table information has to be spread practically throughout the whole AST in order to reach all identifier references. In the basic paradigm of AGs, corresponding attributes have to be declared for all relevant symbols and copy rules have to be inserted in order to code the actual propagation. The following specification (a variant adopted from [Kos91]) uses a rule model as default for the normal propagation.

```
<Node> ::= Abstract

<Root> : <Node> ::= Abstract
  Loc rootST : SymbolTable;
  stROOT := init;
  for all sons(x) in Descendant son(x).st := stROOT;

<Descendant> : <Node> ::= Abstract
  Anc st : SymbolTable;
  for all sons(x) in Descendant son(x).st := st;

<Program> : <Root> ::= {<mainBlock : BeginBlock>}

<BeginBlock> : <Descendant> ::= {<declPart : DeclList> & <stmtPart : StmtList>}
  Loc stLOCAL : SymbolTable;
  stLOCAL := declPart.stASSEMBLED;
  stmtPart.st := stLOCAL;
```

The class Node models general (abstract) nodes in the whole AST. We assume that grammar symbols either inherit from Root or Descendant, both being subclasses of Node. Program is the start symbol of a concrete grammar. BeginBlock models nested blocks consisting of declarations and statements. For that production, the propogation has to be overridden. Note that the actual accumulation of symbol table entries in the declarations part is not modelled yet by the above specification. ♢

4.2.1.5 Discussion

There are some problems with object-oriented AGs (based on object-oriented CFGs) besides the need for adhering to a certain style of CFGs:

Insufficient support for propagation Example 4.2.2 demonstrates how the *propagation* downwards in an AST can be specified. The concept of rule models defining ancestral attributes is crucial for that purpose. However, this concept is not sufficient to describe the *accumulation* of a data structure, i.e. the symbol table in a declaration part, for example. To define such computational behaviour in a compact way, we had to be able to define how attributes are copied on the RHS (not only from the LHS to the RHS), and how synthesized attributes of the LHS are computed. Since rule models are not applicable in this case, we can only use concrete semantic rules. Consequently, the computational behaviour cannot be described in a way abstracting from the underlying CFG. This shortcoming is overcome in our transformational approach because the propagation and computation schemata provide more expressive power than rule models; refer also to Section D.3 for an example generalizing Example 4.2.2. Note that certain paradigm shifts of *Lido* provide a means for that problem, too.

Missing concepts for adaptation There is a notion of overriding semantic rules. More in detail, semantic rules and rule models (in *Mjølner/Orm*) can be overridden by semantic rules, but we cannot override given semantic rules by a rule model. This is a minor technical point. There is another problem due to lack of expressive power: For several adaptations of the computational behaviour we have in mind, there is no way to express them, e.g. the insertion of pre-/post- computations for certain semantic rules, the extension of a propagation.

Another serious lack of adaptability concerns structural specifications. They cannot be overridden. Once a nonterminal has been specified by a structural description, it is subject to inheritance no longer. This anomaly is not much improved by *case*-classes in *Mjølner/Orm*, because the inherited syntactical structure can be insufficient and the applicability of the concept crucially relies on the proper introduction of *case*-classes during the initial design process.

Consequently, object-oriented AGs facilitate design of AGs, but adaptation is only addressed to a lower extent.

Relationship to object-oriented programming languages The outlined approach to add object-oriented concepts to the AG formalism omits several notions typical for object-oriented programming languages. Attributes do not describe a modifiable state of an object. There are extensions of that view, e.g. in the system Tools [Kos91], but then the declarative nature is not preserved. Thus, we consider that property rather as an advantage. Nevertheless, this problem indicates crucial differences between object-oriented (imperative) programming languages and declarative formalisms.

Semantic rules correspond to methods in an acceptable manner: Due to the locality principle of AGs, all the attributes of a syntactic rule to be defined are known. For each of them there must be a semantic rule (a "method for definition"). It is sometimes suggested to regard the method selection in object-oriented AGs as late binding, e.g. in [Hed89]. That point of view seems to be artifical because even in the *basic* AG paradigm each production *locally* defines how synthesized attributes of the LHS and ancestral attributes of the RHS are computed.

It would be interesting to see if the constructs *super* and *self* present in object-oriented programming languages were useful in the context of AGs.

To sum up, the primary notion added to AGs, when speaking of object-oriented AGs, is *inheritance*. To obtain some expressive power, rule models or other concepts must be added. Although rule models *rely* on inheritance, they provide rather yet another concept than some inherently object-oriented concept.

Restrictions to retain well-formedness Dealing with attribute inheritance, some extra effort is necessary to retain well-formedness of the underlying AG. An AG is well-formed if each syntactical rule is associated with semantic rules defining synthesized attributes of the LHS and ancestral attributes of the RHS. Moreover, each root nonterminal, i.e. the syntactical start symbol and/or the semantic base classes, must not have ancestral attributes, because it would not be possible to define them.

For CFGs obeying single inheritance, this property can be checked. The following restriction permits us to check that all ancestral attributes of the RHS are defined in a reasonable way: Descendants of the nonterminals on the RHS must not declare new ancestral attributes. It is correct that this restriction does not introduce practical problems, as stated in [Hed89] because we can always move the declaration of ancestral attributes upwards in the class hierarchy. However, it is a formal artifact as well as a contradiction to the object-oriented point of view, that adaptations should not effect existing classes.

Incompatibility of multiple inheritance and attribute inheritance Allowing CFGs to define a class system with multiple inheritance, severe restrictions are needed on the attribute declarations to ensure well-formedness. Thus, multiple inheritance can not be used de facto. This a serious problem because it is by no means obvious that one superclass per nonterminal is sufficient to factor out the common behaviour.

Following our transformational approach, the above problem does not exist because an arbitrary number of parts of the computational behaviour can be added by subsequent transformations. For each of these steps a different closure of symbols can be used. The limited form of inheritance based on grammar symbols can be simulated by suitable reachability closures in our approach. We can take other collections of symbols as well.

4.2.2 Remote access

The basic formalism of attribute grammars imposes the principle of locality. Attributes referred to in the semantic rules associated with a syntactical rule r must be attributes of the symbols in r. If a computation depends on a non-local attribute, auxiliary attributes for symbols on the path and suitable semantic copy rules have to be added to propagate the attributes along the tree. To avoid this explicit propagation, constructs for remote (attribute) access have been suggested by Kastens [Kas76] and Lorho [Lor77], for example. A comprehensive presentation of the subject has been published by Kastens and Waite in [KW94], where some of the examples and comments have been taken from. There, the following three forms of remote access are distinguished:

1. A computation depends on an attribute to be found *walking up* the tree from the current node.
2. A computation is a *combination* of certain attributes in the *subtree* rooted in the current node.
3. A computation *updates an invariant* for some iterative computation visiting nodes in (depth-first) left-to-right order.

Note that these concepts are "static" in spite of the above explanation, i.e. the corresponding attributes are known at AG compile time. Furthermore, we want to mention, that the second form somehow combines the aspect of remote access *and* the use of all the accessed attributes in computations. We will present some examples for these patterns of remote access and we will discuss the corresponding simulation based on our approach.

In the first example we want to compute the static nesting depth of a block. The main program block has nesting depth 0; refer to the rule [program]. Let us assume that a block is one form of statement, then the depth of a nested block is obtained from the increment of the depth of its ascendant block. To access the depth of the ascendant block, it has to be transmitted to the nonterminal *statement* by means of auxiliary attributes and semantic copy rules. To avoid this coding, the *Including* ... construct can be used to find the first instance of the specified attribute by walking up the tree.

$$\begin{array}{llcl}
[\mathsf{program}] & \mathit{root} & ::= & \mathit{block} \\
 & \mathit{block}.\mathsf{DEPTH} & = & \mathit{zero} \\
 & & & \\
[\mathsf{inner}] & \mathit{statement} & ::= & \mathit{block} \\
 & \mathit{block}.\mathsf{DEPTH} & = & \mathit{inc}(\textbf{Including}\ \mathit{block}.\mathsf{DEPTH})
\end{array}$$

There are some possibilities to simulate this kind of remote access. Let us sketch one scenario where we start from the following non-well-defined GSF schema:

$$\mathit{root}(\ldots)\ :\ \&\ \mathit{zero} \to \mathsf{DEPTH}, \mathit{block}(\ldots, \mathsf{DEPTH}, \ldots). \qquad [\mathsf{program}]$$

$$\mathit{statement}(\ldots)\ :\ \&\ \mathit{inc}(\mathsf{DEPTH}) \to \mathsf{DEPTH}', \mathit{block}(\ldots, \mathsf{DEPTH}', \ldots). \qquad [\mathsf{inner}]$$

In rule [inner] there is an undefined variable DEPTH. To derive a well-defined specification with the proper propagation and update of nesting depths, the following transformation can be applied:

$$\textbf{Remote}\ \mathsf{DEPTH}\ \textbf{From}\ \{\mathit{root}\}$$

We only have to point out that the nesting depths are propagated starting at *root*. We can even omit the initialization of the nesting depth as provided by the semantic rule for [program] because it can be represented by the following transformation:

$$\textbf{Default For}\ \mathsf{DEPTH}\ \textbf{By}\ \&\ \mathit{zero}$$

To illustrate the second pattern of remote access, that is to say attributes in descendant nodes are combined in a certain way, the problem of determining undeclared and useless variables is addressed[3]. Variable identifiers are accumulated separately in the declaration and the statement part:

$$\begin{array}{llcl}
[\mathsf{blockrule}] & \mathit{block} & ::= & \mathit{declaration\_part}\,\mathit{statement\_part} \\
 & \mathit{block.undeclared} & = & \mathit{statement\_part}.\mathsf{IDS} \setminus \mathit{declaration\_part}.\mathsf{IDS} \\
 & \mathit{block.useless} & = & \mathit{declaration\_part}.\mathsf{IDS} \setminus \mathit{statement\_part}.\mathsf{IDS} \\
 & & & \\
[\mathsf{dp}] & \mathit{declaration\_part} & ::= & \mathit{declaration\_list} \\
 & \mathit{declaration\_part}.\mathsf{IDS} & = & \textbf{Constituents}\ \mathit{variable}.\mathsf{ID} \\
 & & & \quad \textbf{With}\ (\mathsf{IDS}, \_\cup\_, \{\_\}, \emptyset) \\
 & & & \\
[\mathsf{sp}] & \mathit{statement\_part} & ::= & \mathit{statement\_list} \\
 & \mathit{statement\_part}.\mathsf{IDS} & = & \textbf{Constituents}\ \mathit{variable}.\mathsf{ID} \\
 & & & \quad \textbf{With}\ (\mathsf{IDS}, \_\cup\_, \{\_\}, \emptyset)
\end{array}$$

In general, the *Constituents* ... *With* ... construct is defined as follows. Let s be a grammar symbol, σ, σ' sorts (attribute names), *union*, *unit* and *zero* are semantic function symbols with the profiles $\mathit{union} : \sigma' \times \sigma' \to \sigma'$, $\mathit{unit} : \sigma \to \sigma'$, $\mathit{zero} :\to \sigma'$. Let $v_1, \ldots, v_n$ be the instances of $s.\sigma$ found in the descendant nodes of the current node.

$$\textbf{Constituents}\ s.\sigma\ \textbf{With}\ (\sigma', \mathit{union}, \mathit{unit}, \mathit{zero})$$

denotes the following computation:

- $n = 0$: *zero*
- $n > 0$: $\mathit{union}(\mathit{unit}(v_1), \mathit{union}(\mathit{unit}(v_2), \mathit{union}(\cdots, v_n)))$

In the above example, the parameters are instantiated as follows:

- σ': IDS denoting the sort of sets of identifiers, i.e. $\mathsf{IDS} = \mathcal{P}(\mathsf{ID})$,
- *union*: $\_\cup\_$, i.e. the union on sets,

[3]We prefer to use a different example than the pedagogical (?) example presented in [KW94].

- *unit*: $\{\_\}$, i.e. the singleton set construction,
- *zero*: $\emptyset$, i.e. the empty set.

Obviously, two aspects are intermingled in the *Constituents* ... *With* ... construct, that is to say the access of attributes in the subtree—which is somehow dual to the access of attributes found by walking up the tree—and the combination of the potentially unknown number of attribute instances. It is the unknown number that requires the higher-order behaviour in the sense of *fold* recursion schemata, but the actual way of computing the combination is not really a concept inherent to remote access. All the aspects of the *Constituents* ... *With* ... construct can be unbundled in a corresponding definition based on our operator suite; refer to Section D.4. We use the operator **Reduce** for the combination of multiple attributes in a rule and the propagation schemata are useful to propagate the composed value.

Let us comment on the third pattern of remote access, that is to say *chains*. A chain relates computations in left-to-right depth-first order within certain subtrees. A chain may propagate values or specify dependencies in that order. To support remote access for chains means that we specify only computations which compute a new chain value, whereas the actual propagation is not specified. The accumulation of symbol table entries serves as an example:

$$
\begin{array}{lll}
[\mathsf{dp}] & \mathit{declaration\_part} & ::= \mathit{declaration\_list} \\
 & \textbf{Chainstart} & \mathit{declaration\_list}.\mathsf{ST} = \mathit{init} \\
 & & \\
[\mathsf{decs}] & \mathit{declaration\_list} & ::= \mathit{declaration}\ \mathit{declaration\_list} \\
 & & \\
[\mathsf{dec}] & \mathit{declaration} & ::= \mathit{variable}\ \text{“:”}\ \mathit{type} \\
 & \mathit{declaration}.\mathsf{ST} & \ = \ \mathit{add}(\mathit{declaration}.\mathsf{ST}, \mathit{variable}.\mathsf{ID}, \mathit{type}.\mathsf{T})
\end{array}
$$

It is obvious that chains can be simulated using the operator suite, because all the propagation schemata in our operator suite are based on left-to-right propagation. To specify only the computations which compute a new value corresponds to the style proposed for the operator **Remote**.

Boyland describes in [Boy96b, Boy98] *collection attributes* as a way to combine disparate definitions of an attribute. The declaration of a collection attribute states an initial value and a combining function. In contrast to that, the above approach describes the “collection” (i.e. combination) as part of the actual computation in terms of the *Constituents*-construct. In similarity to Kastens and Waite, Boyland also strongly links collection attributes and remote access, although his understanding of remote access is different. He proposes a paradigm shift such that objects with fields may be created. References to such objects may be transmitted as ordinary attributes. The fields can be read and written via the reference attributes. In [Boy98] Boyland analyses the resulting direct non-local dependencies and he shows how to render these dependencies in classical terms. Essentially, the fields of an object must be scheduled in a way that classical dependencies based on control attributes are sufficient. The *Lido* specification formalism [Kas91, KW94] in the system Eli [GHL+92] supports side-effects in a related way where dependencies between computations can be forced by the *Depends* ... *On* ... construct. In contrast to that, Boyland derives such dependencies by an analysis. Hedin's door attribute grammars [Hed92, Hed91, Hed94] leave all responsibility for scheduling to hand-written code.

4.2.3 Symbol computations

In the previous subsection we have described forms of remote access with emphasis on the concepts as provided by *Lido*—the AG specification language of *Eli*. In this subsection we want to comment on further paradigm shifts of *Lido*, that is to say *symbol computations* and *inheritance*. Besides rule models, symbol computations are another concept to specify semantic rules (i.e. computations) abstracting from the underlying context-free grammar. We should point out that the form of inheritance in *Lido* is quite different from the inheritance which we have characterized in Section 4.2.1 on object-oriented AGs. It is a matter of terminology if *Lido* should be called an *object-oriented* AG specification language.

Following our meta-programming approach, it is straightforward to define transformations which insert computations (including conditions) and copy parameters. Thus, computational behaviour can obviously be described independently from a skeleton. By turning the sorts and the symbols, which are used to address parameter positions etc., into parameters of the transformation, such descriptions of computational behaviour become reusable. Thereby, *symbol computations* in the sense of [KW94] can be presented as applications of operators like **Define** and **Use**. Applying such transformations to some rules, a specific computational behaviour is inserted. More elaborate symbol computations usually have to make use of remote access. Again, the simulation in our meta-programming approach is straightforward. The corresponding transformations simply make use of the corresponding propagation schemata.

Consider the following fragment of an AG. It specifies how the block nesting depth is initialized for the axiom of the AG and how it is adapted for blocks as a form of statements and for procedure bodies, where the new depth is obtained by incrementing the current depth in both cases.

$$\begin{array}{llll}
[\mathsf{program}] & \mathit{root} & ::= & \mathit{block} \\
 & \mathit{block}.\mathsf{DEPTH} & = & \mathit{zero} \\
 & & & \\
[\mathsf{inner}] & \mathit{statement} & ::= & \mathit{block} \\
 & \mathit{block}.\mathsf{DEPTH} & = & \mathit{inc}(\mathbf{Including}\ \mathit{block}.\mathsf{DEPTH}) \\
 & & & \\
[\mathsf{procbody}] & \mathit{body} & ::= & \mathit{block} \\
 & \mathit{block}.\mathsf{DEPTH} & = & \mathit{inc}(\mathbf{Including}\ \mathit{block}.\mathsf{DEPTH})
\end{array}$$

Symbol computations make it possible to associate computations rather with symbols than with rules. Thus, a more reusable formulation of the computational behaviour associated with the rules [*inner*] and [*procbody*] is expressed as follows:

$$\textbf{Symbol}\ \ \mathit{block}\ \ :\ \ \textbf{Inh}.\mathsf{DEPTH}\ \ =\ \ \mathit{inc}(\textbf{Including}\ \mathit{block}.\mathsf{DEPTH})$$

Instead of concrete grammar symbols, abstract symbols can be used. Grammar symbols can inherit from the abstract symbols by a separate declaration.

$$\begin{array}{lll} \textbf{Symbol} & \mathit{contour} & : \textbf{Inh}.\mathsf{DEPTH} = \mathit{inc}(\textbf{Including}\ \mathit{contour}.\mathsf{DEPTH}) \\ \textbf{Symbol} & \mathit{block} & : \textbf{Inherits}\ \mathit{contour} \end{array}$$

Multiple inheritance is possible. Furthermore, symbol computations can be overridden by concrete computations associated with rules. The fact that symbol computations are really independent of the symbols used in a particular language definition depends very often on the use of remote access. The above abstract symbol computation can be represented as a transformation as follows:

$$\begin{array}{l} \lambda \mathit{contour} : \mathsf{Name}.\ \lambda \mathit{from} : \mathsf{Name}.\ \textbf{Hiding}\ \&\ \mathit{inc}\ \textbf{Do}\ (\\ \quad \textbf{Remote}\ \mathsf{DEPTH}\ \textbf{From}\ \{\mathit{from}\} \\ \circ\ \textbf{Add}\ \langle\langle \mathsf{Input}, \&\ \mathit{inc}, \mathsf{DEPTH}\rangle\rangle \\ \circ\ \textbf{Define}\ \langle \mathsf{Input}, \mathit{contour}, \mathsf{DEPTH}\rangle\ \textbf{By}\ \&\ \mathit{inc} \\ \circ\ \textbf{Ensure}\ \langle\langle \mathsf{Input}, \mathit{contour}, \mathsf{DEPTH}\rangle\rangle) \end{array}$$

The parameter *from* is needed to establish the remote access. Note that it is possible to use a more compact form for certain symbol computations by introducing auxiliary schemata. The above symbol computation, for example, suggest the following pattern: A unary computation is added to define an input position of a grammar symbol, where the input of the computation is obtained by remote access.

4.2.4 Coupling

Attribute coupled grammars (ACGs) have been proposed by Ganzinger & Giegerich for designing phase-oriented AG specifications; refer e.g. to [Gie88]. Two AGs are coupled via the underlying CFG of the second AG, i.e. the CFG can be thought of to define an intermediate language. A special root attribute of the former AG is synthesized by constructing a word of the intermediate language by exploiting productions of the second CFG as constructors.

Coupling is not a proper extension to the AG paradigm. It is rather a programming technique. The benifit of coupling is that a problem can be specified in separate phases which can be combined into a single specification under certain circumstances based on *descriptional composition.* Thereby, the construction and the traversal of intermediate data structures can be avoided. The relationships between descriptional composition and deforestation have been studied by Correnson, Duris, Jourdan, Parigot and Roussel [DPRJ96, DPRJ97, CDPR98] emphasizing the benefits and the point of view of descriptional composition.

The question is whether the degree of reusability achieved by coupling is sufficient. This is certainly not the case because phase-like decomposition is only a very simple means of modularity. The mapping described by a component AG of a ACG cannot be modified, but only surrounded by further phases. Bellec, Jourdan, Parigot, Roussel extend the concept of descriptional composition with the intent to improve modularity in AG specification [LJPR93, RPJ94]. Particularly, they suggest to *derive* the coupling from simple associations between the grammar symbols of two grammars rather than to specify the coupling. Another achievement is separate compilation (i.e. separate evaluator construction). Separate compilation is an aspect of modularity which we almost ignored in our work, since our emphasis is on expressive power facilitating reuse.

Farrow's et al. Composable Attribute Grammars (CAGs) [FMY92] can be understood as a generalization of ACGs. A so-called glue AG may construct phrases of so-called component AGs by using productions as constructors. Terminals may have input and output attributes in order to allow bidirectional data flow between glue and components. Essentially, CAGs generalize ACGs because of the output attributes for terminals. Following [FMY92], the expressive power of output attributes can be gained alternatively by synthesizing a single complexly-structured root attribute. [KW94] reports a number of problems concerning reusability of CAGs.

4.2.5 Patterns

Dueck's and Cormack's MAGs (Modular Attribute Grammars) [DC90] are based on (production) patterns and (attribution) templates. A *pattern* is similar to a context-free rule. Whereas a context-free rule contains only vocabulary symbols, a MAG pattern contains variable symbols, which match *any* vocabulary symbol, quoted symbols which match one vocabulary symbol and ellipses, which match zero or more vocabulary symbols. A *template* is a semantic rule on the variable or quoted symbols.

Attribution of a CFG with regard to a set of MAGs is done in terms of syntactical matching controlled by semantic constraints, i.e. a production pattern matches a context-free rule. The attributes defined in the corresponding semantic rules and these semantic rules themselves are only added, if the attributes used in the semantic rules can be synthesized due to other semantic rules, and if the defined attributes are used somewhere else.

Example 4.2.3
The following two MAGs define the attribution schema for a bucket brigade. The module *env* describes how the data structure is propagated down the derivation tree, whereas the module *def* describes how the data structure is passed up the tree. We assume that the start symbol in the grammar is *goal*.

module *env*

1 $'goal \to A \ldots$

$A.env = 0$

2 $A \to B \ldots$

$B.env = A.env$

3 $A \to \ldots B\ C \ldots$

$C.env = B.def$

module *def*

4 $A \to \ldots B \ldots$

$B.def = B.env$

5 $A \to \ldots B$

$A.def = B.def$

6 $A \to$

$A.def = A.env$

$\diamondsuit$

The way in which an attribution is added to a CFG is primarily controlled by syntax. That is not most appropriate to obtain an abstract definition of aspects of attribution. Indeed, [KW94] reports problems in instrumenting that kind of matching for the design process, e.g. auxiliary attributes have to be added to trigger matching in the desired way. The *Constituents*-construct (see Subsection 4.2.2), for example, cannot generally be simulated by MAGs. Our transformational approach is more flexible than the use of patterns and templates. Example 4.2.3 can be simulated by propagation schemata of our operator suite.

Adams reports in his thesis [Ada91] on an approach similar to MAGs.

4.2.6 FNC-2

There are many compiler compilers with support for attribute grammars, e.g. *FNC-2* [JP91, JP90, Par88, JPJ+90], *Eli* [GHL+92] and *Cocktail* [GE90]. Such systems use mostly a certain instance of the attribute grammar paradigm with some particular specification features. The most interesting concepts underlying *Lido*—the attribute grammar formalism of *Eli*—have been explained in Subsection 4.2.2 and Subsection 4.2.3 (i.e. remote access, symbol computations and inheritance). The concept of object-oriented attribute grammars as used in the *Ag* specification language in *Cocktail* has been discussed in Subsection 4.2.1. Besides that, *Ag* supports a rather simple straightforward module concept which need not to be considered here.[4] *FNC-2* offers a number of descriptional tools supporting reuse. There are features arising from the system architecture of *FNC-2* and there are other features more closely related to *OLGA*—the attribute grammar description language of *FNC-2*.

Now let us consider *FNC-2*'s features relevant for reusability in more detail.

Passes A large application can be split into a sequence of passes where each pass takes as input the intermediate representation produced by a previous one and as output and transforms it into another intermediate representation to be fed to the next pass. The passes are usually described by AGs or other specifications following the tree-to-tree mapping paradigm. If a pass is described by an AG, it is either

- a side-effect AG, where the output tree is the same as the input tree except that it carries different attributes, or

[4]The module concept is similar to the concept of phases mentioned below for the *FNC-2* system.

- a functional AG having zero, one or more output trees, generally different from the input tree.

FNC-2 supports merging of side-effect AGs as well as descriptional composition of functional AGs, i.e. coupling.

AAS The intermediate representations are called attributed abstract syntaxes (AAS) which can be regarded as grammars extended with attribute declarations. The specification of AASs and AGs is done separately.

Declaration and definition modules Regarding *OLGA* as a general-purpose applicative language, it supports the notion of modules, in which a set of related objects (type, functions, constants and exceptions) can be defined. Similar to Modula-2, a module is split into two compilation units, a declaration module declaring the objects visible from outside and a definition module in which the actual implementation of visible and non-visible objects is given. Objects can be opaque and modules can be parameterized.

Phases An AG can be divided into phases to be regarded as blocks with local declarations and import clauses. A phase is likely to contain the semantic rules for some aspect of the complete AG. A phase is a pure decomposition construct, i.e. it is not an extension of AGs.

Productions Productions are also regarded as blocks. This is at least useful for the consideration of values which are local to the production. These values, which may depend on attributes of the production, are usually referred to as local attributes.

Attribute classes The automatic generation of semantic copy rules is a rather well-known technique to define attribute occurrences more implicitly. *FNC-2* also supports the generation of non-copy rules based on the concept of attribute classes [Le 89, Le 93]. An attribute class consists of

- sets of attribute occurrences and
- associated templates to specify the semantic rules which define these occurrences.

A template specifies

- the productions to which the template will be applied to and
- the actual semantic rules.

If some attribute occurrence is not defined explicitly, it will be tried to match the corresponding production with the syntactic part of some template, and—under certain not so straightforward circumstances—the semantic rules of the template are used to define the occurrence. The concept of attribute classes is similar to symbol computations in *Lido*; refer to Subsection 4.2.3 as far as it concerns the expressive power.

Consequently, *FNC-2* supports modular specification in a sophisticated manner, i.e. passes can be used at the top level of a compound application. Phases support semantics decomposition similar to Watt's Partitioned Attribute Grammars [Wat75]. Attribute classes support semantics decomposition as well, but the actual details of syntactical and semantic constraints to find default rules are not so apparent. Last but not least, the module concept of *OLGA* as a general-purpose applicative language makes it possible to use ADTs in the design of an AG.

Even with these powerful modularity concepts, *FNC-2* fails to solve some problems we can solve with our operator suite, mostly because composition is supported rather than adaptation. In particular, semantic rules and thereby computational behaviour cannot be adapted. Such an adaptation would be useful, for example, to establish a different propagation or to insert a precomputation. Syntactical rules cannot be overridden, folded and unfolded, but that is necessary for structural adaptations. Some problems can be handled in *FNC-2* by fairly simple textual adaptions of specifications or by introducing a different pass. The situation could be slightly improved if phases and attribute classes could be separately checked. However, in general, a modular specification will fail to be reusable, if the assumed structure and the supported parameterization is not sufficient for a certain application.

Bellec, Jourdan, Parigot and Roussel have done some work on improving modularity based on descriptional composition [LJPR93, Le 93, RPJ94, Rou94]. They suggest, for example, to *derive* an attribute grammar specifying the translation from one grammar to the other from certain associations between the non-terminals and terminals of the grammar. Thereby, one can deal with ACGs (refer to Subsection 4.2.4) more modular. The corresponding concepts will possibly be added to the implementation of the *FNC-2* system.

4.3 Semantics

Many researchers have worked on reusability (compositionality, modularity, extensibility) of semantics specifications, refer e.g. to [Mos83, Mos88, Mog89, Mog91, SJ94, Mos92, BL92, Bra92, CF94, BR94, Hud96, Mos96, LH96, WH97, BS98]. In this section we comment on some of these attempts. We also want to compare our meta-programming approach with some attempts in the semantics community. Such a comparison must appear somehow artifical because the most promising attempts are usually based on styles and notations which are beyond our general framework, e.g. denotational semantics, action semantics and abstract state machines. On the other hand, this situation makes clear that we cannot adopt existing (partial) solutions to achieve reusability for representatives of our framework, e.g. natural semantics, attribute grammars etc. The solutions suggested in the framework of denotational semantics (or higher-order functional programming), for example, heavily rely on the higher-order nature of the specifications.

This section is structured as follows. Subsection 4.3.1 recalls some well-known problems regarding the extensibility of (denotational) semantics. Afterwards, we consider possible improvements of mainly extensibility, but also other pragmatic properties. First, Mosses' and Watt's action semantics are reviewed in Subsection 4.3.2. Second, the use of monads in semantics (and functional programming) is the subject of Subsection 4.3.3. Third, Cartwright's and Felleisen's extensible denotational semantics are presented in Subsection 4.3.4. Finally, the notions of conservative extension and (successive) refinement for abstract statement machines (evolving algebras) are regarded in Subsection 4.3.5.

4.3.1 Motivation

In denotational (and operational) semantics adding an unforeseen construct to a language may require a reformulation of the entire description because denotational descriptions crucially depend on the domains used in the profiles of the semantic functions which have to be adapted for new constructs. This problem becomes a serious hindrance when developing descriptions of larger languages. It also prevents the reuse of parts of a denotational description when describing a related language.

We want to present examples for problems with the extensibility of denotational semantics. We start with the profile of the semantic function for statements of a rather simple language:

$$[\![\cdot]\!]_{\mathsf{STM}} : \mathsf{STM} \to \mathsf{MEM} \to \mathsf{MEM}$$

The semantic meaning of a statement sequence, for example, is obtained by the normal composition:

$$\begin{aligned} [\![S_1; S_2]\!]_{\mathsf{STM}} &= [\![S_2]\!]_{\mathsf{STM}} \circ [\![S_1]\!]_{\mathsf{STM}} \\ &= \lambda m.[\![S_2]\!]_{\mathsf{STM}} \ ([\![S_1]\!]_{\mathsf{STM}} \ m) \end{aligned}$$

The way λ-notation is used for specifying semantic entities depends strongly on the details of domain definitions. If errors during statement execution are taken into consideration, not only the above profile will change, but any intermediate meaning must be handled differently. The new version of the semantic function will be based on the following profile:

$$[\![\cdot]\!]_{\mathsf{STM}} : \mathsf{STM} \to \mathsf{MEM} \to (\mathsf{MEM} \oplus \{error\}_\perp)$$

The above semantic equation is reformulated as follows:

$$[\![S_1; S_2]\!]_{\mathsf{STM}} = [\![S_1]\!]_{\mathsf{STM}} \ \mathit{then} \ [\![S_2]\!]_{\mathsf{STM}},$$

where $\_ \ \mathit{then} \ \_ \ : (D \to (D' \oplus \{error\}_\perp)) \times (D' \to (D'' \oplus \{error\}_\perp)) \to (D \to (D'' \oplus \{error\}_\perp))$ corresponds to strict (w.r.t. *error*) composition and it is defined as follows:

$$f \ \mathit{then} \ g \ x = \begin{cases} error, & \text{if } \mathbf{Is}_{\{error\}_\perp}(f \ x) = \mathbf{True} \\ g \ (f \ x), & \text{if } \mathbf{Is}_{D'}(f \ x) = \mathbf{True} \end{cases}$$

For languages with sharing, i.e. with pointers or call-by-reference parameter passing, the flat memory model is insufficient. An environment binding identifiers to denotable values, e.g. locations of a store and a store associating locations with storable values must be distinguished. Consequently, the profile of the semantic function becomes as follows:

$$[\![\cdot]\!]_{\mathsf{STM}} : \mathsf{STM} \to \mathsf{ENV} \to \mathsf{STORE} \to (\mathsf{STORE}\ \oplus \{error\}_\perp)$$

Note also that all semantic equations need to be reformulated to adhere to the new style of variable lookup and modification and to propagate environments and stores accordingly.

An even more fundamental change is required when jumps are added because a migration from the direct style to the continuation style has to be performed. The profile of the semantic function for statements becomes as follows:

$$[\![\cdot]\!]_{\mathsf{STM}} : \mathsf{STM} \to \mathsf{ENV} \to \mathsf{CONT} \to \mathsf{CONT},$$

where $\mathsf{CONT} = \mathsf{STORE} \to (\mathsf{STORE}\ \oplus \{error\}_\perp)$. The semantics of statements sequences, for example, changes because the composition of meanings has essentially to be reversed compared to the direct style:

$$[\![S_1; S_2]\!]_{\mathsf{STM}}\ e\ c = [\![S_1]\!]_{\mathsf{STM}}\ e\ ([\![S_2]\!]_{\mathsf{STM}}\ e\ c)$$

Similar serious problems arise when we generalize to power domains when adding non-determinism. If we anticipated all these changes, we could start with the more complex domains, but that would be unreasonable as well as notionally burdensome. Note that although the above examples are tuned towards denotational semantics, similar problems arise for operational semantics descriptions, e.g. in the style of SOS, or natural semantics, i.e. domains, profiles and data flow becomes inappropriate if a language extension must be performed.

4.3.2 Action semantics

Action semantics [Mos92, Mos96] is a framework for the formal description of programming languages. Its main advantage over other frameworks is the inherent extensibility and modifiability of action semantics descriptions (ASDs), ensuring that extensions and changes to the described language require only proportionate changes to its descriptions. Another purely pragmatic problem addressed by action semantics is the difficulty of recovering fundamental concepts, such as order of execution or scopes for bindings, from their denotational semantics description. The concepts are rather encoded in higher-order functions on domains. In action semantics, there is support for several concepts such as transient, scoped, stable and permanent information built into the notation.

The overall structure of an ASD is similar to a denotational semantics description:

- a context-free grammar defines the abstract syntax,
- semantic equations are used to give inductive definitions of compositional semantic functions mapping abstract-syntax trees to semantic entities.

In contrast to denotational semantics, the main kind of semantic entities is *actions*. Semantic entities are specified by the so-called *action notation* in contrast to λ-notation in denotational semantics. Actions are essentially computational entities. The performance of an action directly represents information processing behaviour and reflects the gradual, step-wise nature of computation. There are subsidiary kinds of semantic entities, that is to say *data* and *yielders*. Items of data are (in contrast to actions) essentially static, mathematical entities, representing pieces of information, e.g. particular numbers. A yielder represents an unevaluated item of data the value of which depends on the current information. Action semantics is intended as a framework for semantics description. To approach this goal, the action notation supports a reasonable number of concepts for semantics description directly.

Example 4.3.1
Let us consider a semantic equation modelling the semantics of an identifier as an expression (in the sense of a constant or a variable) as common for imperative languages:

evaluate I = give the number bound to I or
give the number stored in the cell bound to I.

_ *or* _ is an action combinator to choose between alternative actions. If one or another operand is bound to fail—as in the example—the choice is deterministic. In the first option I corresponds to a constant, whereas in the second option I corresponds to a variable, with an associated cell (according to scoped information) and with a stored value in the cell (according to stable information). The yielder *the d bound to Y* evaluates to the current binding for the particular token Y, provided that it is of sort d. The yielder *the d stored in Y* is a similar yielder to access stable information. The primitive action *give Y* completes and gives the data yielded by evaluating the yielder Y. Thereby, transient information is produced. ◇

The action notation can be specialized according to particular semantics description, i.e. certain domains are instantiated as appropriate for the actual language, e.g. a simple language declares values like numbers and booleans as *storable* values, whereas memory cells and values are *bindable* values to cope with variables and constants.

A performance of an action, which may be part of an enclosing action, either *completes*, *escapes*, *fails*, or *diverges*. An action may be nondeterministic having different possible performances. An action performance processes information. There are different kinds of information giving rise to so-called *facets* of actions. The information may be classified according to how far it tends to be propagated, as follows:

- *transient*: tuples of data, corresponding to intermediate results;
- *scoped*: bindings of tokens to data, corresponding to symbol tables;
- *stable*: data stored in cells, corresponding to values assigned to variables;
- *permanent*: data communicated between distributed actions.

Making extensions and changes to an ASD generally affects only those parts of the description dealing directly with the constructs involved. This property depends on two crucial features of action notation:

- Each combinator is defined *universally* on actions, in contrast with function composition in λ-notation, for example, which requires exact matching of types between the composed functions.
- There is no mention of the presence or absense of any particular kind of information processing, except where creation or inspection of this information is required. For instance, stored information is referred to only in semantic equations dealing with program variables.

It is obvious that action semantics succeeds to support extensibility in semantics definition. However, this requires the use of a special notation based on quite a few action primitives and combinators. It is hard to compare such a semantics framework with our meta-programming approach which is not tuned towards semantics description. Our approach emphasizes the synthesis, transformation and combination of (first-order) specifications. We try to give some concluding remarks on the relationship between both approaches:

1. Extensibility of ASDs relies on the above mentioned features of action notation, i.e. a *particular specification language* is used. In contrast to that, extensibility in our approach arises from the modifiability of specifications through *meta*-programs.

2. Since we do not perform any specific extension of the considered (first-order) target specification formalisms, we are not able to make semantic concepts explicit in the manner as action semantics does. On the other hand, some schemata of the operator suite can be related to semantic concepts built into the action notation. The propagation of storable and scoped information, for example, is facilitated by propagation schemata. The access to and the production of transient information can be modelled by the computation schemata in various ways.

3. Although action notation can be extended, there are some basic assumptions which are tuned towards semantics definition. Inheritance and accumulation of data, for example, is supported by the declarative facet (scoped information) and the imperative facet (stable information). There is no obvious way to propagate several different data structures in different ways.

4. Without programming at the meta-level certain adaptations and compositions cannot be performed per se, precomputations, for example, cannot be interpolated. In more general terms, semantic equations cannot be adapted at all.

5. Action notation satisfies several algebraic laws. However, the intended interpretation of an ASD is based on an operational semantics (SOS) for action notation. In this respect, action semantics can be regarded as a *higher level of semantics description* compared to operational (denotational) semantics. Our operator suite rather reflects possible manipulations on programs of certain target languages. Several of

these manipulations can be regarded as abstractions from programming practice. In this respect, meta-programming can be regarded as a *higher level of programming* compared to the underlying target language.

4.3.3 Monads and monad transformers

Moggi proposed to use monads to structure denotational semantics [Mog89]. A monad is essentially an (endo-) functor with an additional structure (certain natural transformations) in the categorical sense. We will stick here to the simpler view based on the terminology of functional programming and we will use Haskell-like notation for our examples. In functional programming, a monad is a type constructor together with some polymorphic functions characterized below. Wadler [Wad92] popularized Moggi's ideas in the functional programming community by showing that many frequently used type constructors together with common combinators are actually monads and that interpreters for a great variety of language concepts, for example, can be designed in a modular fashion if the equations adhere to the monadic style. Espinosa developed Semantic Lego [Esp95]—a Scheme-based system for the composition of modular interpreters exploiting monads and monad transformers. Some other contributions to modular interpretation based on monads are [SJ94, LHJ95].

The basic idea of the monadic style of programming is to consider a function of type $\tau \rightarrow \tau'$ rather as a function of type $\tau \rightarrow M\ \tau'$, where M is a type constructor. Extensibility is achieved by instantiating M as appropriate. M can, for example, add state transformation to τ'. For a given type τ, elements of τ are called *values* and elements of $M\ \tau$ are called *computations*, according to the terminology of Moggi. Besides the type constructor M, we need two polymorphic functions:

$$\begin{aligned}\mathsf{unit}_M &:: \tau \rightarrow M\ \tau\\ \mathsf{bind}_M &:: M\ \tau \rightarrow (\tau \rightarrow M\ \tau') \rightarrow M\ \tau'\end{aligned}$$

unit_M is a generalization of the identity function. $\mathsf{unit}_M\ x$ takes $x \in \tau$ to the corresponding representation in $M\ \tau$. bind_M is a generalization of the functional application in a monad. bind_M takes a value $x \in M\ \tau$ and a function on τ (but not $M\ \tau$). bind_M is usually written in infix notation.

A *monad* is a triple $\langle M, \mathsf{unit}_M, \mathsf{bind}_M\rangle$, where the functions satisfy the following laws:

- bind_M is associative.
- bind_M has unit_M as left and right identity.

Figure 4.2 lists some simple monad definitions. $\mathcal{M}_I$ is the identity monad. $\mathcal{M}_S$ is the monad for state transformation. $\mathcal{M}_E$ is the environment monad.

We want to characterize the approach to modular interpreters and the monadic style of higher-order functional programming using interpreter examples in a Haskell-like notation as in [Wad92]. The domains according to the core of an interpreter for a functional language are presented in Figure 4.3. An interpreter in the monadic style is presented in Figure 4.4.

Monad	type $M\ \tau =$	$\mathsf{unit}_M\ v =$	$c\ \mathsf{bind}_M\ f =$
$\mathcal{M}_I$	a	v	$f\ c$
$\mathcal{M}_S$	$\mathsf{State} \to (a, \mathsf{State})$	$\lambda s.(v, s)$	$\lambda s_0.\text{let } (v, s_1) = c\ s_0 \text{ in } f\ v\ s_1$
$\mathcal{M}_E$	$\mathsf{Env} \to a$	$\lambda e.v$	$\lambda e.\text{let } v = c\ e \text{ in } f\ v\ e$

Figure 4.2: Some monads

$$\begin{array}{lll}
\text{type} & \mathsf{Name} & = \text{String} \\
\text{data} & \mathsf{Exp} & = \mathsf{Var\ Name} \mid \mathsf{Lambda\ Name\ Exp} \mid \mathsf{Apply\ Exp\ Exp} \\
 & & \mid\ \mathsf{Const}\ \text{Int} \mid \mathsf{Dyadic\ Exp\ Exp\ Dsym} \\
\text{data} & \mathsf{Dsym} & = \mathsf{Plus} \mid \ldots \\
\text{data} & \mathsf{Value} & = \mathsf{Wrong} \mid \mathsf{Num}\ \text{Int} \mid \mathsf{Fun}\ (\mathsf{Value} \to M\ \mathsf{Value}) \\
\text{type} & \mathsf{Env} & = [(\mathsf{Name}, \mathsf{Value})]
\end{array}$$

Figure 4.3: Signature for an interpreter of a pure functional language

Nested function applications are flattened in terms of a sequence of applications of bind_M. Values are coerced to computations by unit_M.

$$\begin{array}{lcl}
ie & :: & \mathsf{Exp} \to \mathsf{Env} \to M\ \mathsf{Value} \\
ie\ (\mathsf{Const}\ n)\ \rho & = & \mathsf{unit}_M\ (\mathsf{Num}\ n) \\
ie\ (\mathsf{Var}\ i)\ \rho & = & \mathsf{lookup}_{\mathsf{Env}}\ \rho\ i \\
ie\ (\mathsf{Lambda}\ i\ e)\ \rho & = & \mathsf{unit}_M\ (\mathsf{Fun}\ (\lambda x.ie\ e\ ((i, x) : \rho))) \\
ie\ (\mathsf{Dyadic}\ e_1\ e_2\ \mathsf{ds})\ \rho & = & (ie\ e_1\ \rho)\ \mathsf{bind}_M\ (\lambda v_1.(ie\ e_2\ \rho)\ \mathsf{bind}_M\ (\lambda v_2.\mathsf{comp}\ v_1\ v_2\ \mathsf{ds})) \\
ie\ (\mathsf{Apply}\ e_1\ e_2)\ \rho & = & (ie\ e_1\ \rho)\ \mathsf{bind}_M\ (\lambda v_1.(ie\ e_2\ \rho)\ \mathsf{bind}_M\ (\lambda v_2.\mathsf{apply}\ v_1\ v_2)) \\
 & & \\
\mathsf{lookup}_{\mathsf{Env}} & :: & \mathsf{Env} \to \mathsf{Name} \to M\ \mathsf{Value} \\
\mathsf{lookup}_{\mathsf{Env}}\ [\,]\ i & = & \mathsf{unit}_M\ \mathsf{Wrong} \\
\mathsf{lookup}_{\mathsf{Env}}\ ((j, v) : \rho)\ i & = & \text{if } i == j \text{ then } \mathsf{unit}_M\ v \text{ else } \mathsf{lookup}_{\mathsf{Env}}\ \rho\ i \\
 & & \\
\mathsf{comp} & :: & \mathsf{Value} \to \mathsf{Value} \to \mathsf{Dsym} \to M\ \mathsf{Value} \\
\mathsf{comp}\ (\mathsf{Num}\ n_1)\ (\mathsf{Num}\ n_2)\ \mathsf{Plus} & = & \mathsf{unit}_M\ (\mathsf{Num}\ (n_1 + n_2)) \\
\ldots & & \\
\mathsf{comp}\ v_1\ v_2\ \mathsf{ds} & = & \mathsf{unit}_M\ \mathsf{Wrong} \\
 & & \\
\mathsf{apply} & :: & \mathsf{Value} \to \mathsf{Value} \to M\ \mathsf{Value} \\
\mathsf{apply}\ (\mathsf{Fun}\ f)\ x & = & f\ x \\
\mathsf{apply}\ f\ x & = & \mathsf{unit}_M\ \mathsf{Wrong}
\end{array}$$

Figure 4.4: Interpretation in a monad (call-by-value)

Some first remarks should be made. The monadic style is burdensome because of all the applications of the polymorphic functions unit_M and bind_M. Moreover, it must be decided which functions return values and which return computations. It can be assumed that *all* functions return computations, but this is possibly an overspecification. Note also that there can be different layers of computations. If we think of, for example, state transformation in an interpreter, the central interpreter function will possibly transform the state, whereas an auxiliary function will not. If we think of error handling, several functions will possibly produce error messages, not only the central interpreter function. As with all parameterization techniques, such problems cannot be avoided completely. Eventually, different monads for different parts are needed. Such an adaptation cannot be performed. In the case of statement-oriented imperative languages, the evaluation of expressions either involves side-effects or it does not. In the absence of side-effects the monad used for evaluation of expressions should not represent state transformation but rather propagation of a constant state.

The standard call-by-value interpreter can be derived from Figure 4.4 by the following substitution. M, unit_M and bind_M is substituted by the identity monad $\mathcal{M}_I$. It is now assumed that the interpreter should be extended with language constructs for reference cells; refer to Figure 4.5 showing the signature part. The following new constructs have to be established:

```
type  Loc    =  Int
data  Exp    =  ... | Ref | Set Exp Exp | Deref
data  Value  =  ... | Loc
type  State  =  [(Loc, Value)]
```

Figure 4.5: Extension for reference cells (signature part)

- Ref intended for the allocation of a cell,
- Set e_1 e_2 for the update of a cell, where e_1 is computed to a cell, whereas e_2 is computed to the value to be stored and
- Deref e for dereferencing the cell computed from e.

The monad parameters have to be substituted by the monad for state transformation. We ommit the straighforward equations for defining the interpretation of the new constructs.

Let us comment on another extension of the intial interpreter. Instead of using the error value Wrong, proper error messages will be returned. A distinction between successful values and error messages can be modelled by a monad as in Figure 4.6. One advantage of using the monadic style for the more realistic kind of error handling is that the strict behaviour can be ensured, i.e. once an error occurred, the evaluation of the entire expression fails.

```
type  Partial a  =   Ok a | Fail String
      _ ⇀ _      ::  Partial a → (a → Partial b) → Partial b
      Fail s ⇀ f =   Fail s
      Ok x ⇀ f   =   f x
```

Figure 4.6: The error monad

The substitution of the monad parameters by the error monad is not sufficient yet because the possibility of producing error messages is not used at all. There are still equations returning the accidentally "successful" value Wrong. We cannot anticipate all such changes and the monadic style fails to provide a solution required here. What is needed is that the equations concerning error handling are *replaced*[5]; refer to Figure 4.7 for the suitable equations.

$\mathsf{lookup}_{\mathsf{Env}}$ [ ] i	=	Fail "variable not bound"
comp v_1 v_2 ds	=	Fail "type error in basic operation"
apply f x	=	Fail "illegal application"

Figure 4.7: Variants of equations making use of error messages

All in all, the monadic style is an elegant parameterization technique giving support for modular programming and specification. The style crucially relies on the possibility that extensions can be expressed by a suitable actual parameterization of the monad parameters. There are extensions or in other words adaptations, which cannot be expressed in this way. The technique is well-studied for interpreters of programming languages, since the entire space of features can be anticipated.

It is interesting to notice that meta-programming provides some oppurtunities to improve the usability of the monadic style:

- It is a rather simple transformation to establish the monadic style in a given program. Thereby, it is not necessary any longer to code in the monadic style all the time. More significantly, the decision which functions return values and which return computations can be delayed. Moreover, different sets of monad parameters can be distinguished.
- We can perform adaptations which are beyond the parametricity provided by the monadic style, e.g. to override equations, or to insert precomputations.

```
ie                              ::  Exp → Value
ie (Const n)                    =   Num n
ie (Dyadic e1 e2 ds)            =   comp (ie e1) (ie e2) ds
ie (Apply e1 e2)                =   apply (ie e1) (ie e2)

comp                            ::  Value → Value → Dsym → Value
comp (Num n1) (Num n2) Plus     =   Num (n1 + n2)
...
comp v1 v2 ds                   =   Wrong

apply                           ::  Value → Value → Value
apply (Fun f) x                 =   f x
apply f x                       =   Wrong
```

Figure 4.8: Constructs at the *Value*-level

[5][Wad92] points out such an adaptation, but there, the only option is text-editing.

- It is not always the obvious choice to hide semantic aspects in a monad. However, to achieve modularity based on the monadic style, we must put all aspects into the monad. Meta-programming provides other possibilities to install semantic aspects. Consider, for example, the interpreter module in Figure 4.8 which concerns the same functional language as in Figure 4.4, but only the constructs at the *Value*-level being the most basic level of the semantic model for interpretation. The module is not written in the monadic style. In contrast to Figure 4.4, environments are not propagated. Using a suitable propagation schema (e.g. the operator **Inherit**) we can add environment propagation; refer to Figure 4.9 for the result. Wadler's examples in [Wad92] do not consider environments as part of the composed monad, i.e. environments are propagated explicitly as in Figure 4.9. Espinosa's Semantic Lego [Esp95] points out a separate environment level. Both approaches are problematic (without meta-programming). In the first approach we cannot achieve modularity because constructs at the *Value*-level must be described with the irrelevant propagation of environments, i.e. a module like Figure 4.8 could not be reused. Following the second approach, a remarkable overspecification can be recognized because environment propagation is restricted to only a few functions and not to all interpreter functions, e.g. the functions apply and comp do not contribute to the environment propagation.

$$
\begin{array}{lcl}
ie & :: & \mathsf{Exp}\ \boxed{\rightarrow \mathsf{Env}} \rightarrow \mathsf{Value} \\
ie\ (\mathsf{Const}\ n)\ \boxed{\rho} & = & \mathsf{Num}\ n \\
ie\ (\mathsf{Dyadic}\ e_1\ e_2\ \mathsf{ds})\ \boxed{\rho} & = & \mathsf{comp}\ (ie\ e_1\ \boxed{\rho})\ (ie\ e_2\ \boxed{\rho})\ \mathsf{ds} \\
ie\ (\mathsf{Apply}\ e_1\ e_2)\ \boxed{\rho} & = & \mathsf{apply}\ (ie\ e_1\ \boxed{\rho})\ (ie\ e_2\ \boxed{\rho}) \\
 & & \\
\mathsf{comp} & :: & \mathsf{Value} \rightarrow \mathsf{Value} \rightarrow \mathsf{Dsym} \rightarrow \mathsf{Value} \\
\ldots & & \\
 & & \\
\mathsf{apply} & :: & \mathsf{Value} \rightarrow \mathsf{Value} \rightarrow \mathsf{Value} \\
\ldots & &
\end{array}
$$

Figure 4.9: Figure 4.8 with added environment propagation

Finally, we want to comment on the superficial correspondence of monads (or monad transformation) and program transformation. For that purpose we explain in more detail Espinosa's approach to modular interpreters [Esp95] based on lifting ignoring the similar approach based on stratification.

For a functional language as in Figure 4.4 with the extension for reference cells in Figure 4.5 the semantic model can be characterized by the following type A:

$$A = \mathsf{Env} \rightarrow \mathsf{State} \rightarrow (\mathsf{Value} \times \mathsf{State})$$

Modularity is possible because most language constructs operate primarily at a single "level" of the above type. The following levels can be distinguished:

$$A \;\; = \;\; \mathsf{Env} \rightarrow \mathsf{State} \rightarrow (\mathsf{Value} \times \mathsf{State})$$

$$\begin{array}{rcl} S & = & \mathsf{State} \to (\mathsf{Value} \times \mathsf{State}) \\ V & = & \mathsf{Value} \end{array}$$

These levels are related to each other in the sense that A captures S and V, whereas S captures V. More technically, S is related to V by the monad $\mathcal{M}_S$ (Figure 4.2) and in turn A is related to S by the monad $\mathcal{M}_E$. Figure 4.8, for example, shows all constructs at level V. The constructs for reference cells are at the level S. The level A is sufficient for all constructs. Actually, we would like to have a level covering values and environments:

$$E = \mathsf{Env} \to \mathsf{Value}$$

This level would be optimal for the semantics of λ-variables (constructs Var Name and Lambda Name Exp). However, we cannot include this level into the tower of levels because there is no monad relating E and A, i.e. we cannot reuse modules at the level E. Thereby, environment constructs cannot be described in a completely modular way.

To reuse modules at some levels, lifting is performed. In our examples we can lift through V, S and A. Lifting means here, that a monad is used to lift functions at a lower level to an upper level. Thus, we can combine the result of lifting with a module which is defined at the upper level anyway. This process can be repeated as often as necessary. To lift a function f with a certain profile according to a monad $\langle M, \mathsf{unit}_M, \mathsf{bind}_M \rangle$ can be described by lifting operators. Consider, for example, a function f with one parameter p which is untouched by the lifting process and another parameter c to be lifted, i.e. f has the following profile:

$$f : X \times A \to A$$

The resulting function f' has the following profile:

$$f' : X \times B \to B$$

The monad relates A and B. We assume, that the result of functions is lifted in all cases. The corresponding lifting operator which is suitable to lift f to f' can be described by the following λ-expression:

$$\lambda f.\lambda p.\lambda c.c \ \mathsf{bind}_M \ \lambda v.\mathsf{unit}_M \ f(p, v)$$

The above problem with the level E is related to general problem that monads do not compose. More precisely, there is no general constructive way to compose a monad from two other monads such that the features of both monads are combined; refer to [JD93] for a proof and some methods for composition in particular cases. Thereby, modularity based on monads is limited. Moggi's way out of this dilemma (and Espinosa's reminds us in this respect) is to use monad transformers, which is a next step of abstraction. Generally speaking, a monad transformer is a function on monads. The monad transformers $\mathcal{T}_E(M)(T)$ to add environment propagation or $\mathcal{T}_S(M)(T)$ to add state transformation to a monad M applied to a type T can be defined as follows when only the effect to the type is shown:

$$\begin{array}{rcl} \mathcal{T}_E(M)(T) & = & \mathsf{Env} \to M(T) \\ \mathcal{T}_S(M)(T) & = & \mathsf{State} \to M(T \times \mathsf{State}) \end{array}$$

Again, lifting operators can be defined. As modules are parameterized by monad transformers, it is possible, for example, to define the environment constructs completely modular and to pass the state transformation monad transformer $\mathcal{T}_S$ to that module as a parameter in order to arive at A.

Let us compare the monadic style and our meta-programming approach.

- To "lift" a target program from one level to another is performed by a program transformation (e.g. transformers in the sense of Section 3.5 on lifting) in our approach. Thus, the problem of finding a suitable monad (transformer) corresponds to finding a suitable program transformation. Regarding the running example in a first-order setting, we should be able to deal with modules at all the levels V, S, E, A.
- An important difference is that the reuse of modules in the monadic style crucially relies on the previous parameterization of a module by a monad (transformer). Moreover, the monadic style is per se only applicable to settings with order higher-order functions, e.g. functional programming and denotational semantics. Following our approach, we de not rely on an explicit parameterization and we provide a solution for even first-order settings.
- The correspondence of monads and program transformations is obviously superficial because monads and monad transformers are higher-order objects in the underlying formalism, i.e. monads are a means for modularity within the language, whereas transformations are objects from the meta-level.
- Lifting in our sense (refer to Section 3.5) corresponds to the complete process of lifting in the monadic sense, where several modules are lifted (in the sense of monads) through several levels.

The monadic style relies on proper design for reuse in advance. Programs have to be parameterized. Reuse corresponds to passing monads or to explicit lifting. Program transformations emphasize adaptation of programs.

4.3.4 Extensible denotational semantics

Cartwright and Felleisen present in [CF94] an approach to extensible denotational semantics specifications. Actually, they introduce a new format for denotational language specifications, the so-called *extended direct semantics* (EDS), that accommodates orthogonal extensions of a language without changing the denotations of existing phrases. The authors demonstrate the method by a stepwise definition of a powerful dialect of Scheme. The method also supports the construction of interpreters for complete languages by composing interpreters for language fragments. Many of the subsequent explanations and the examples have been taken from [CF94].

The suggested schema crucially relies on a distinction between a *complete program* and a *nested program phrase*. A complete program is thought of as an agent that interacts with the outside world, e.g. a file system and that effects global *resources*, e.g. the store. A *central authority* administers these resources. The meaning of a program phrase is a *computation*, which may be a *value* or an *effect*. If it is an effect, it is propagated to a central authority. The propagation process adds a function in the sense of a *handle* to the effect package such that the central authority can resume the suspended calculation. An "administrator" function modelling the central authority performs the actions specified by effects. Actions can examine and modify resources, or may simply abort execution. Once the action has been performed, the administrator extracts the handle portion of the effect and invokes it, if necessary, in similarity to the continuation passing style. Casting a language extension into the framework requires the specification of four components:

- the new syntactic constructors,
- the extension of the domains for values, resources and actions,
- new clauses of the meaning function for program phrases and
- new clauses of the administrator function.

EDS are extensible because for several semantic concepts, such as error handling, continuations, stores, the profile of meaning function for program phrases and previous semantic equations have not to be modified. Extensions effect only the central authority which must adapted to perform the new actions according to the language extension. A special composition operator for meanings ensures that all effects are always passed to the central authority. Some more technical details about EDS and an example are concluded in Section A.8.

The approach of EDS is very much tuned towards (dynamic) semantics description similar to action semantics. This is in contrast to the monadic style and to our meta-programming approach. In particular, a distinction between complete programs and nested program phrases and the overall design of the semantic framework only applies to dynamic semantics descriptions (of certain languages). Besides extensibility, the primary achievement is that extensions do not imply changes to the denotations of program phrases. That does not hold for the monadic style. EDS is a small framework compared to action semantics which is huge specification language. Technically, EDS is rather a programming *style* or a style of denotational semantics than an extension of a specification language or a new language per se. The style of EDS corresponds again to a kind of parameterization. It is assumed that profiles of functions and the structure of domains do not need to be modified and that is sufficient to extend domains in a certain way to take new actions (effect messages) into consideration.

4.3.5 Extension and refinement of abstract state machines

Gurevich's Abstract State Machines (ASMs), previously called Evolving Algebras [Gur95], provide an operational semantics approach. It is a good intuition to understand an ASM as "pseudo-code over abstract data". For the purpose of our work we will concentrate on the application of ASMs for modelling semantics and implementations of programming languages, although the formalism is also applicable for modelling architectures, protocols and control software etc. Very roughly, to specify an abstract state machine, an algebra to start with and rules describing function updates need to be characterized. Thus, algebras correspond to states, whereas the update rules which are performed simultaneously correspond to the transition relation in the sense of operational semantics.

It is a particular feature of the ASM approach that an ASM can be tailored to an arbitrary abstraction level (in contrast to Turing machines and other approaches to operational semantics). If different abstract levels are needed, one can even have a hierarchy of ASMs. In [BR94], for example, Egon Börger and Dean Rosenzweig develop a hierarchy of ASMs by means of *successive refinement* in order to reconstruct the WAM [War93] from a more abstract ASM for Prolog. Another interesting example showing that the ASM approach supports modularity and extensibility is the modular Java semantics [BS98] by Egon Börger and Wolfram Schulte where they factor out sublanguages by isolating orthogonal language features, namely imperative, procedural, object-oriented, exception handling and concurrency features. Starting from the imperative kernel language all the other features can be added in successive steps. The resulting ASMs build up a sequence of models, where each model is a *conservative extension* of its predecessor.

Let us first characterize the notion of refinement following [BR94]. Afterwards the stronger notion of conservative extension is outlined. Finally, we comment on the kind of extensibility and modularity provided by ASMs with regard to our approach.

In a refinement step a more "concrete" ASM B is constructed and it is related to a more "abstract" ASM A. For a proper refinement we are seeking for a $\mathcal{F}$ mapping states $\mathcal{B}$ of B to states $\mathcal{F}(\mathcal{B})$ of A, and rule sequences R of B to rule sequences $\mathcal{F}(R)$ of A, so that the following diagram commutes:

$$\begin{tikzcd}
A \arrow[r, "\mathcal{F}(R)"] & A' \\
B \arrow[u, "\mathcal{F}"] \arrow[r, "R"'] & B' \arrow[u, "\mathcal{F}"']
\end{tikzcd}$$

Refer to [BR94] for details including notions like correctness, completeness and operational equivalence. Let us mention some kinds of adaptations to be performed during refinement. One possibility is to place assumptions on certain members of the signature of the more abstract ASM which are "implemented" in the more concrete ASM. Another kind of adaptation concerns rules. They can be replaced. New rules can be added. It is possibly also necessary to adapt the signature of the given ASM making the definition of the above $\mathcal{F}$ more involved.

A conservative extension is a special kind of refinement, where each run of B, which only depends on A's signature, can be transformed canonically into a run of A. Egon Börger and colleagues work on the rigorous definition of this concept and they plan to publish proofs for the conservative extensions presented in [BS98].

Comparing ASMs with our meta-programming approach, we first should state that ASMs are beyond the scope of our target languages. A more interesting question is how the kind of modularity and extensibility achieved by refinement can be compared with our results. ASMs are executable (under certain conditions), but note that the notion of refinement only facilitates the proof of a certain relationship between ASMs. There is no useful *effective* method so far to make a more abstract ASM more concrete. Thus, in a narrow sense the ASM approach does not facilitate modular composition or performance of an extension. It rather provides a proof technique to realize that one ASM is a refinement / conservative extension of another one. In our approach we are interested in effective methods for program composition and adaptation. We are *computing* target programs.

4.4 Program development

4.4.1 Stepwise refinement

Much of the work on formal methods for the development of correct programs is based on Dijkstra's work on the weakest precondition calculus. Back, for example, developed a refinement calculus [BvW98] providing a unified framework for stepwise refinement, program transformation and program synthesis for imperative programming. There are some works on refinement of logic programs. [KT93, Trc93] is based on partial deduction (PD) originating from partial evaluation in functional programming. The primary field of application for PD is program optimization and specialization, but it turned out that it is quite suitable for stepwise refinement based on a transformational approach. Most of the following definitions and explanations are taken from [Trc93].

Definition 4.4.1
Let S and S' be programs. S is correctly refined by S', denoted by S **ref** S', if S' satisfies any specification that S does, i.e. S **sat** $R \Rightarrow S'$ **sat** R for any R in the set of specifications.
$\diamondsuit$

Here **sat** denotes the satisfaction relation. It follows from the definition that **ref** is a preorder, i.e. **ref** is reflexive and transitive. Constructs for combining programs into larger ones must be monotonic w.r.t. **ref**, for subprograms for example, a subprogram T in a program $S[T]$ can always be replaced by its refining program T'.

Let us mention several operators for refinement in logic programming; refer to Section A.9 for details. *unfold* allows an atom in the body of a clause c to be replaced by a conjunction of atoms. *fold* is inverse to *unfold*. It abbreviates a conjunction of atoms. *prune* and *add* delete or add a clause in a program. On clause level, *thin* and *fatten* delete or add an atom in the body of a clause. *restrict* selects a subprogram.

However, the operators for refinement are neither intended nor sufficient to facilitate meta-programming:

- There are no operators on the atom level (and at the term level either). A large set of schemata, e.g. **Replace**, **Left To Right**, cannot be specified. What is needed are basic operations for constructing and deconstructing programs.

- Except for the simple application of the concept of a call-graph in *restrict*, no global considerations are involved as for our reachability operators building the basis for propagation schemata.
- The form of the operators *unfold*, *fold*, *thin* and *fatten* is not suitable for meta-programming at all because writing a meta-program we cannot regard the actual atoms of programs as assumed.
- Based on *fatten*, computations can be inserted, but again one would require total knowledge of the variables of the rule under consideration. Transformation cannot be stated here in a way abstracting from the actual program. Moreover, the insertion of computations cannot be combined with changing the parameters of the original premises as necessary for the insertion of precomputations, for example.

The obvious advantage of a stepwise refinement approach during program development is the straightforward support for correctness of derivation. The operators preserve *refinement equivalence* if certain applicability conditions are satisfied; refer to Section A.9 for details. The operations are suitable for *reasoning* about refinements of programs and about partial deduction. The actual set of operations is not useful for general program synthesis, transformation and composition. Moreover, refinement *equivalence* is too restrictive in several cases during program adaptation.

In general, refinement has been studied much more exhaustively for the imperative paradigm [BvW98, Heh93]. There is certain direction in refinement called *data refinement* or (data transformation [Heh93]) which possibly could be adopted for our framework to characterize properties of certain program transformations in a systematic manner.

4.4.2 Stepwise enhancement

Stepwise enhancement [Lak89, SS94, JS94] developed by Sterling et al. is a program development methodology. The methodology suggests to develop Prolog programs systematically from two classes of standard components. *Skeletons* are simple Prolog programs with a well-understood control flow. *Techniques* are standard Prolog programming practices.

Example 4.4.1
This example is taken from [NS97]. The following two programs are skeletons for traversing binary trees with values only at the leaf nodes.

The following program does a complete traversal of the tree.

```
is_tree(leaf(X)).
is_tree(tree(L, R)) :- is_tree(L), is_tree(R).
```

In contrast to that, the following program traverses a single branch of the tree.

```
branch(leaf(X)).
branch(tree(L, R)) :- branch(L).
branch(tree(L, R)) :- branch(R).
```

Note that the first program can be regarded as a type definition of trees. ♢

Standard examples for skeletons are traversals of recursive data structures. Techniques capture basic Prolog programming practices, such as building a data structure of performing calculations in recursive code. A technique interleaves some additional computation around the control flow of a skeleton. More syntactically, techniques may rename predicates, add arguments to predicates, add goals to clauses and/or add clauses to programs. Unlike skeletons, techniques are not programs but can be conceived as a family of operations that can be applied to a program to produce a program. Obviously, this characterization brings us very close to our meta-programming methodology.

Example 4.4.2
We will give two examples of applying the so-called *calculate* technique to the `is_tree` predicate given in Example 4.4.1 (again adopted from [NS97]). The *calculate* technique computes a value. An extra argument is added to the defining predicate of the skeleton for the computed value and an extra goal for an arithmetic calculation is added to the body of each recursive clause.

The following program computes the product of the value of the leaves of the tree. Note the predicate `is_tree` has been renamed.

```
prod_leaves(leaf(X), X).
prod_leaves(tree(L, R), Z)
 :- prod_leaves(L, X), prod_leaves(R, Y), Z is X * Y.
```

Similarly, the following program computes the sum of the value of the leaves of the tree. The only difference is the choice of names and the extra goal.

```
sum_leaves(leaf(X), X).
sum_leaves(tree(L, R), Z)
 :- sum_leaves(L, X), sum_leaves(R, Y), Z is X + Y.
```

♢

A technique applied to a skeleton is said to yield an *enhancement*. An enhancement which preserves the computational behaviour of the skeleton is called an *extension*. Two enhancements of the same skeleton share computational behaviour and they can be combined into a single program by *composition*. Obviously, we can also consider the combination of two techniques.

We try to present a comparison of stepwise enhancement and our methodology based on meta-programming:

- Stepwise enhancement is dedicated to Prolog programming. Indeed, the kind of computations and syntactical manipulations considered are really tuned towards Prolog. Recently, Kirschbaum et al. [KMS96] discussed that stepwise enhancement is equally applicable to other logic programming languages. Our approach provides a general framework which can be instantiated for quite different specification formalisms.

- Stepwise enhancement does not consider modes or types. The use of directional types is a crucial factor in our approach. Types (sorts) are needed for the selection of parameters, for example. Moreover, programs are required to be well-typed corresponding to a safety feature for program construction. Modes are needed for data flow criteria. A transformation, for example, which should provide definitions for undefined variables must use modes.

- The emphasis in stepwise enhancement is on the identification of useful skeletons and techniques. Another issue is correctness of program construction [SJK93, JKS94], which means that properties of components are retained in a composed program. Here the notions of composition and extension as well as program maps are central [Jai95, KSJ93]. In our work, the emphasis is on the actual calculus for meta-programming, i.e. on the machinery to *define* techniques in the sense of stepwise enhancement. Nevertheless, our operator suite attempts to capture programming practices as well.

- We unbundle several roles of useful program transformations by our schemata for parameterization, computation, etc. Thereby, we have a kind of a basis for deriving useful techniques. Properties of transformations are analysed in some depth including properties beyond the scope of the settings of stepwise enhancement, e.g. totality, idempotence.

- The concept of *composition* (of enhancements) is similar in intent to our operation for superimposition. However, there are some technical differences. First, following our approach the same skeleton (including names) is assumed for both operands of superimposition, whereas in stepwise enhancement, renaming is considered as part of composition. Second, in our approach skeleton elements and computations are strictly distinguished from each other arising from the origin in attribute grammars. A more conceptional difference arises from the possibility in our approach to contract parameters.

- The process of producing an enhancement (an extension), i.e. the the application of a technique to a skeleton, is quite similar to the application of an (extending) transformation to some rules. The extension and the skeleton can be related to each other by a *symbol mapping* studied, for example, in [Jai95, KSJ93]. Our projections are similar to the concept of symbol mappings.

- A concept like lifting (refer to Section 3.5) is not considered at all in stepwise enhancement because lifting is rather related to program composition.

4.4.3 Generic fragments and transformations

Generic fragments (or schemata, templates, cliches etc.) are used in program synthesis, whereas generic transformations (or transformation schemata) are used in program transformation. In both fields there are other tools than such schemata which are however beyond the scope of this work. For a survey on program transformation in logic programming refer to [PP94]. Program schemata (refer to [Dev90] for an early reference, refer e.g. to [FLO97] for some enumeration of recent work) have been introduced in logic programming in the context of program synthesis [DL94] with the motivation of reusability.

$$
\boxed{
\begin{array}{lcll}
r(\mathsf{X}) \to (\mathsf{Y}) & : & \mathit{isMinimal}(\mathsf{X}), & [\mathsf{minimal}] \\
 & & \mathit{solve}(\mathsf{X}) \to (\mathsf{Y}). & \\
 & & & \\
r(\mathsf{X}) \to (\mathsf{Y}) & : & \mathit{isNonminimal}(\mathsf{X}), & [\mathsf{nonminimal}] \\
 & & \mathit{decompose}(\mathsf{X}) \to (Z, X_1, X_2), & \\
 & & r(\mathsf{X}_1) \to (\mathsf{Y}_1), & \\
 & & r(\mathsf{X}_2) \to (\mathsf{Y}_2), & \\
 & & \mathit{compose}(\mathsf{Z}, Y_1, Y_2) \to Y. &
\end{array}
}
$$

Figure 4.10: A generic fragment for the *divide-and-conquer schema*

Consider, for example, the rules in Figure 4.10 defining the divide-and-conquer schema as useful for logic programming. We refer to [Smi85], where the synthesis of divide-and-conquer algorithms is considered in the field of functional programming. As far as meta-programming is concerned, we can regard such a generic fragment t as a function f_t of the following form:

$$f_t : \mathsf{Symbol}^\star \times \mathsf{Symbol}^\star \times \mathsf{Sort}^\star \to \mathsf{Rules}$$

$f_t(\mathit{defined}, \mathit{required}, \mathit{sorts})$ is intended to derive a concrete specification fragment from the generic fragment, where *defined* are the actual symbols to be defined by the template (r in Figure 4.10), *required* are the actual symbols required in the schema (*isMinimal*, *solve*, ... in Figure 4.10) and *sorts* enumerates the sorts to be used in the schema (X, Y, Z in Figure 4.10). f_t can be derived from t by a simple transformation; refer to Section D.2 for the function corresponding to Figure 4.10.

Consequently, program schemata can be represented as such functions, whereas program transformation schemata can be regarded as parameterized meta-programs. In both cases instantiation is simply functional application. Our meta-programming framework and the actual operator suite provide a detailed framework for reusable and executable descriptions of program (transformation) schemata.

4.4.4 Specification-building operators

Several approaches to modularity have been formalized in terms of operators on specifications. There are for example formal operators to model import and export constructs, e.g. the operations union, intersection and encapsulation with a compositional semantics supporting modularity in logic programming [Bro93, BMPT94].

In this subsection, we want to consider a sophisticated approach to modularity in algebraic specification based on so-called specification-building operators [Wir86, ST88, SST92, Wir94]. The following characterization has been taken from [SST92] to a great extent.

Algebraic specification is used to model (software) systems as algebras. The simplest possible way to give a specification of a system is to present a (very long, unstructured and hence unmanageable) list of axioms over a given signature. Thereby, the properties can be described which have to be satisfied by the system. Specification languages allow specifications to be built in a structured manner using a predefined set of specification-building operations. Consequently, Σ-specifications are considered instead of Σ-sentences. A Σ-specification SP is expected to determine a class $[\![SP]\!] \in \mathcal{P}(Alg(\Sigma))$ of Σ-algebras, the models of SP. SP is consistent if $[\![SP]\!] \neq \emptyset$.

Let us mention some typical operators; refer to Section A.10 for formal details.

- **impose** Φ **On** SP to impose (further) axioms Φ on a specification SP,
- **derive from** SP **by** σ and **translate** SP **by** σ to apply signature morphisms in various ways,
- $\_ \cup \_$ and $\_ + \_$ to combine two specifications,
- **minimal** SP ... to consider minimal algebras only,
- **iso** $-$ **close** SP to take the closure under isomorphism.

At the semantic level, specification-building operations are functions mapping classes of algebras to classes of algebras. Such operations may also be regarded as functions mapping specifications to specifications, the operator **impose**, for example, syntactically merges two sets of axioms.

We provide a comparison of the algebraic approach and our meta-programming approach to reusability. The arguments, which are raised here, apply accordingly to several other operator suites, e.g. those in [Bro93, BMPT94]:

- The specification building operators support programming in the large. Sets of axioms and the associated class of algebras are the main subjects under consideration. This is in contrast to our approach, as we can operate on any fragment of a specification, not only on rules but also on parameterized symbols and on parameters. Higher-order functional programs are used to *compute* specifications, signatures and fragments of them.
- Operators like **derive** and **translate** are abstract forms of well-established concepts for modularity, mainly parameterization (with renaming involved).
- Several operators are only meaningful as far as the associated models are concerned. They cannot be regarded as functions from sets of axioms to sets of axioms, e.g. the operators **iso** $-$ **close** and **minimal**.
- Indeed, the algebraic approach supposes model-theoretic operators (or in other words semantics-oriented operators in [Bro93, BMPT94]), whereas we take a rather syntactical approach, although we insist on certain preservation properties.

Chapter 5

Concluding remarks

First, the main achievements of the thesis are summarized in Section 5.1. Second, the implementation of the framework and the operator suite for meta-programming is outlined in Section 5.2. We also comment on first experiences with this implementation. Finally, topics for future work are indicated in Section 5.3.

5.1 Achievements

The results of the thesis have been discussed in an abstract style in Section 1.3. In this Section, we point out some particular contributions of our work reported in the thesis.

1. There are several suggestions for frameworks for meta-programming, e.g. the approach supported in the logic programming language Gödel [HL94]. An important contribution of our work is its *generality* and its *high level of abstraction*. We can deal with reusability in attribute grammars and logic programming etc. in much the same abstract way. Generality is achieved by the identification of some common target language kernel; refer to Chapter 2. Abstraction is essentially achieved by different layers of operators for meta-programming; refer to Figure 1.12.

2. A particular emphasis of our meta-programming approach is to create a fully-*typed* framework, which is in contrast to several meta-level approaches in the Prolog context and also in contrast to the AsFix approach [Kli94]—to mention an approach in the context of algebraic specification. To take into consideration target types in meta-programs obviously improves safety of meta-programming. More interestingly, we have shown how types can effectively be exploited to control meta-programs, e.g. for addressing parameter positions in target programs.

3. Actually, we are not only concerned with types, but also with modes. *Modes* are as useful for safety of meta-programs and for the control of meta-programs as types are. The usefulness of modes has been recognized in the attribute grammar community as we can see in several works on related paradigm shifts such as Dueck's and Cormack's

modular attribute grammars [DC90] and *Lido* [KW94]. On the other hand, modes have been ignored in other related attempts, e.g. stepwise enhancement [Lak89, SS94, JS94] in logic programming. Our schemata for computations and propagation heavily rely on modes showing the general usefulness of them for other instances such as natural semantics, logic programming and algebraic specification.

4. Semantics preservation is an important notion for reasoning about program transformation. We have indicated several other (and in some contexts more useful) preservation properties of meta programs (Section 2.6), e.g. extending transformations or recovery of well-definedness and fragment selection properties and others for target programs (Section 2.3). In contrast to other attempts such as stepwise refinement [BvW98, KT93, Trc93], we compiled an operator suite suitable for rather meta-*programming* than *formal reasoning*.

5. Program schemata and program transformation schemata have been extensively investigated, for example, in the field of logic programming; refer e.g. to the surveys on program synthesis [DL94] and program transformation [PP94] in logic programming and the LOPSTR proceedings [Fuc97, Gal97, Pro96]. Our suite "unbundles" roles which are used in program transformation aiming, for example, at optimization, program refinement, program composition, program synthesis and programming techniques used for example in stepwise enhancement.

6. As a consequence of generality and abstraction, we can provide a *reconstruction of existing attempts*. Concepts introduced for one target language, can be adopted for other languages. First, such a reconstruction provides an abstract rigorous definition of the concept. Second, it may drastically improve the pragmatics of target languages, where the extracted concept has not been considered so far. We considered, for example, the reconstruction of remote access specific to attribute grammars, stepwise enhancement specific to logic programming. Now these concepts can be applied in natural semantics and algebraic specification as well.

7. There are some unique schemata for transformation and composition:

   - superimposition where contraction is involved; refer to Example 3.3.4;
   - left-to-right propagation where a given propagation is extended in the sense that the previous data flow is "rescheduled"; refer to the introductory example of Subsection 1.2.2;
   - simultaneous renaming of sorts of parameter positions; refer to Subsection 3.2.2;
   - interpolation of computational elements; refer to §3.4.3.5;
   - hiding symbols for the incremental construction of premises; refer to §3.2.4.3;
   - lifting as introduced as higher-order composition on transformations and rules; refer to Section 3.5.

5.2 Implementation

The general framework from Chapter 2 instantiated for natural semantics and GSFs has been implemented in $^{\Lambda}\Delta_{\Lambda}$ [HLR97, LRH96, RL93, Rie92]. A superset of the operator suite presented in Chapter 3 has been specified in the functional calculus provided by the implementation of the instantiated framework. Thereby, we can exploit the meta-programming approach for formal specification—especially language definition—in the specification framework of $^{\Lambda}\Delta_{\Lambda}$.

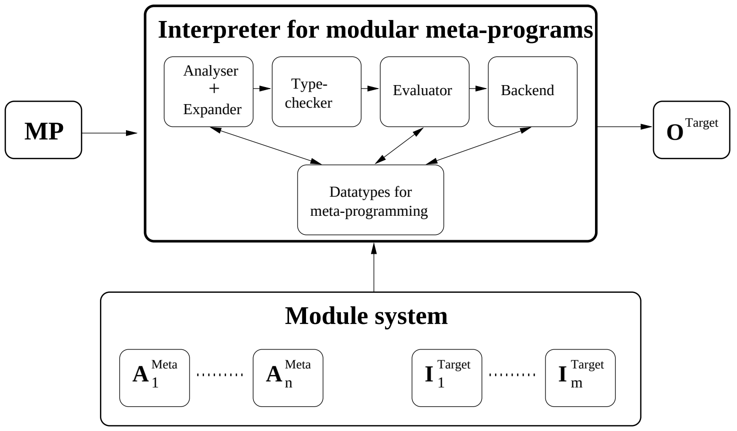


Figure 5.1: Interpretation of modular meta-programs in $^{\Lambda}\Delta_{\Lambda}$

Figure 5.1 shows the overall structure of the implementation. A meta-program **MP** is interpreted as follows. First, **MP** is analysed to obtain an intermediate representation. **MP** possibly refers to target-level modules $\mathbf{I}_i^{Target}$ or auxiliary meta-level modules $\mathbf{A}_j^{Meta}$; refer to Subsection 2.5.5 for modular meta-programming. These modules are obtained from the $^{\Lambda}\Delta_{\Lambda}$ module system and expanded in the intermediate representation. A natural semantics is used to check static semantics of the intermediate representation. Note that a meta-program can be considered as a functional program. Thus, there are no special problems with type checking. If type checking is successful, the actual interpretation or evaluation is performed which is specified by a recursive function definition in the style of denotational semantics. Again, the evaluator is specified in a standard way as common for the semantics of a functional programming language. The evaluator makes use of an ADT for the meta-programming data types introduced in Section 2.1. The ADT is obtained by all the axioms and inference rules shown in Chapter 2 including them for special features related to the instances natural semantics and GSFs. A successful evaluation of **MP** returns the abstract representation of a target program which is passed to a backend as common for $^{\Lambda}\Delta_{\Lambda}$ specification formalisms. The backend writes the target program back to the module systems, keeps track of the dependencies between modules in the module system to support make features, pretty-prints type information and target code and generates executable Prolog code from the target program according to $^{\Lambda}\Delta_{\Lambda}$'s implementation strategy.

The current implementation has some shortcomings we should comment on.

1. The meta-programming interpreter is *very* slow. Doing a composition like the language composition in Section D.1 takes several minutes on a SUN Ultra 5. The main reason for that is that the interpreter is specified in $^{\Lambda}\Delta_{\Lambda}$'s specification framework and $^{\Lambda}\Delta_{\Lambda}$'s implementation strategy is useful for prototyping but not for efficient language implementation. Since the entire operator suite is implemented in the interpreted calculus, huge environments holding all the operator definitions are passed around during interpretation. That results in an unacceptable access efficiency due to the naive environment implementation. The efficiency of interpretation could be improved drastically by an implementation of the functional calculus based on a state of the art implementation of a functional programming language like SML or Haskell.
2. There are some conflicts between the actual $^{\Lambda}\Delta_{\Lambda}$ specification framework and the ideal meta-programming framework developed in Chapter 2. There are for example some $^{\Lambda}\Delta_{\Lambda}$ constructs not covered by the implemented meta-programming framework. There are different representations used in the extended $^{\Lambda}\Delta_{\Lambda}$ system. E.g. GSFs and types have been represented in another way in the previous system compared to what is sensible for meta-programming. Altogether, the actual $^{\Lambda}\Delta_{\Lambda}$ system should be reconstructed to support meta-programming in a clean way without redundancy so that an orthogonal specification framework is achieved.
3. There is no support for finding type errors and debugging at the meta-level.
4. The current implementation of the framework is monolithic in the sense that actually the instance of the framework is specified, i.e. the basic framework and the instantiation is not separated from each other. We would like approach to a modular approach to instantiation.

In spite of these limitations, we can conclude some positive remarks on the actual integration of $^{\Lambda}\Delta_{\Lambda}$ and our meta-programming approach. The expressive power of meta-programming allows us to decompose, compose and adapt specifications in many ways which were not possible before in $^{\Lambda}\Delta_{\Lambda}$. We can go strictly beyond the scope of modular specification as supported by $^{\Lambda}\Delta_{\Lambda}$'s specification formalism ***PRA*** [HLR97, LRH96]. The modular language definition discussed in Section D.1, for example, requires the reusable specification of semantic aspects, the composition technique lifting and structural adaptations. There is no other system to the best of our knowledge which supports such a modular definition.

5.3 Future work

Further areas should be investigated in future.

1. We have tried to outline possible notions of preservation and other properties. We should search for further properties and we should try to develop a more complete programming methodology. The relation between the properties and the real programming practice should be analysed in more detail. An advantage of our approach

to reuse is that the meta-programs are *executable*. A weakness is that many properties of operands and results of transformations can only be ensured by separate proofs. We would like to cover more properties, in the meta language itself in the sense of a kind of type checking.

2. It is current limitation of the operator suite that the computation and propagation schemata are only applicable to instances with a well-definedness notion meeting L-attribution. In general terms, the state of the operator suite concerning completeness, orthogonality and simplicity can certainly be improved.

3. The meta-programming approach together with the actual operator suite requires case studies. We would like to demonstrate that the additional expressive power gained by meta-programming really improves reusability in a practical context. I am working together with coauthors N.v. Bac and G. Riedewald on a language construction set [LRBS], that is to say a library with specification fragments for language design supporting the derivation of prototype interpreters.

4. To actually *write* meta-programs is only one possible application of our work. For many applications, a program manipulation system can be more effective. The challenge of a work on providing such tool support arises from the fact that in existing systems like *Translog* [Bru95] and *Spes* [ABFQ92] essentially fold/unfold-based strategies are considered. A program manipulation system should not only support the application of transformation rules and strategies. It should also guide the user in showing dependencies and conflicts or incomplete aspects of a target program. We possibly can adopt some concepts from Attali's, Pascual's and Roudet's environment for program transformation based on the rule-based language *TrfL* for program transformations [APR97].

5. We should investigate concrete specification frameworks and systems in order to find out if they are useful for the implementation of our meta-programming approach and if they could take advantage from some concepts supplied by the framework and the operator suite. We regard compiler compilers such as *Cocktail* [GE90] or *FNC-2* [JP91, JP90, Par88, JPJ+90] and specification environments such as *Centaur* [BCD+88] or *ASF+SDF* [Kli93] as some possible candidates.

6. The general framework is tuned towards first-order specification formalism with a monomorphic type system. Due to the popularity of polymorphic higher-order functional programming, e.g. Haskell [Has97, Tho96] and SML [MTH90], we would like to see how a similar approach can be taken for such programming languages. Then we need to model, for example, the following notions: anonymous functions, polymorphism, curried functions, type construction. Type constructors and associated combinators tend to be vital parts of functional programs. Therefore, a corresponding meta-programming approach must address type constructors which is out of the scope of the current framework.

Appendix A

Background

A.1 Domain notation

We use the following domain constructors:

- $\mathsf{Boolean} = \{\mathbf{True}, \mathbf{False}\}_\perp$,
- $\_ \otimes \_$ for products,
- $\_ \oplus \_$ for coalesced sums,
- $\_^\star$ for sequences,
- $\_ \rightarrow \_$ for function spaces,
- $\mathcal{P}(\_)$ for power sets,
- $\_?$ for the maybe construction, i.e. $D? = D \oplus \{?\}_\perp$.

π_i^D denotes the i-th projection in $D = D_1 \otimes \cdots \otimes D_n$. For $d \in D = D'^\star$, $\pi_i^D(d)$ evaluates to the element indexed by i in d. $\mathbf{In}_i^D$ denotes the i-th injection, $\mathbf{Out}_i^D$ denotes the i-th projection, $\mathbf{Is}_i^D$ denotes the test for D_i in $D = D_1 \oplus \cdots \oplus D_n$. The i in π_i^D, $\mathbf{In}_i^D$, $\mathbf{Out}_i^D$, $\mathbf{Is}_i^D$ is replaced by D_i if the D_i are distinguishable. The D in superscripts is omitted if it can be derived from the context. Sometimes we also use another notation for constructing sum domains which facilitates pattern matching. $D = \mathit{injection}_1(D_1) \oplus \cdots \oplus \mathit{injection}_n(D_n)$, where the $\mathit{injection}_i$ denote user-defined names for the injections.

A.2 Inference rules

Rules of a natural semantics [Kah87] define a logic and are used as proof-theoretic tool to prove theorems within that logic, building proof trees in a recursive top-down strategy applying axioms and rules and involving unification. This process is non-deterministic, i.e. there can be several proof trees for the same fact.

A prominent example of an executable specification formalism for natural semantics is *Typol* [Des88] as integrated in the Centaur system [BCD$^+$88, JRG92]. One option to execute *Typol* is provided by a translation of the inference rules into Prolog rules taking advantage of the similarity of inference rules and definite clauses.

The style of natural semantics is very suitable for defining static semantics (or type checking) and dynamic semantics of languages. The general idea of a semantic definition in natural semantics is to provide axioms and rules characterizing semantic properties of language constructs. Thus, a semantic definition coincides with a logic and reasoning about the language means proving theorems within that logic. Proofs are done using structural induction on abstract syntax patterns. The initial goal to prove contains a complete abstract syntax term. The corresponding proof tree can be bigger than the given abstract syntax term and even infinite. That is the reason for natural semantics to be suitable for the description of dynamic semantics.

Let us consider the *Typol* formalism slightly more in detail. Inference rules indicate how a conclusion $I_0 \vdash T_0 : S_0$ may be deduced from certain premises $I_i \vdash T_i : S_i$ for $i = 1, \ldots, n$. The I_j are called inherited positions, the T_j are abstract syntax patterns and the S_j are called synthesized positions. A *Typol* rule is of the following form:

$$\frac{I_1 \vdash T_1 : S_1, \ldots, I_n \vdash T_n : S_n}{I_0 \vdash T_0 : S_0}$$

Besides premises, the numerator can also contain predicates for auxiliary computations of the form:

$$\mathit{pred}(\alpha_1, \ldots, \alpha_l \rightarrow \beta_1, \ldots, \beta_m)$$

Inherited and synthesized positions are (tuples of) variables. The set of *input positions of a rule* is composed from I_0, S_1, …, S_n, β_1, …, β_m, whereas the complementary set corresponds to the set of *output positions*. Roughly, input positions are computed by the outer context and they are used in the rule to compute the output positions which are then transmitted to the outer context.

For specifying any kind of judgements in this thesis we use a notational variant of inference rules similar to *RML* [Pet95, Pet94] suggested for natural semantics specifications.

- Alphanumeric identifiers are used to name propositions. Subscripts and superscripts are not parameters, but they qualify the name of the proposition (for readability or to avoid overloading).
- Arguments and results are distinguished. If there are any results, they are separated from the arguments by $\rightarrow$.
- Premises are read from left to right. Actually, arguments of a premise are required to occur somewhere before on a result position of another premise or as an argument of the conclusion.

We do not make use of unknowns.

A.3 Grammars of Syntactical Functions

GSFs (Grammars of Syntactical Functions, [Rie91, RMD83, Rie72, Rie79]) are a kind of attribute grammars closely related to logic programs, and logical grammars. The GSF formalism has been derived from two-level grammars during 1971–1972 with the aim to obtain an executable and more readable form of two-level grammars. GSFs are also similar to the more recent formalism RAG [CD84, DM85, DM93].

A GSF consists of

- *a GSF schema*, i.e. a set of GSF rules, that is to say a parameterized context-free grammar with relational formulae associated with the rules and
- *a GSF interpretation* providing carriers for the parameters and interpretations for relational symbols in the relational formulae.

Concerning formal language definition, a GSF schema defines the syntax and the rough structure of the semantics of a language. The GSF interpretation refines the GSF schema.

Definition A.3.1
A *GSF schema* is a tuple $GS = \langle B, R, V, \alpha, PP \rangle$, where $B = \langle N, T, P, s \rangle$ is a reduced context-free grammar (N set of *nonterminals*, T set of *terminals*, P set of *production rules*, $s \in N$ *start symbol*)—the *basic grammar* of the GSF—, R and V are finite sets of *relational symbols* and *variables*, respectively. PP is a finite set of *production rule patterns*, each of the form

$$\begin{aligned} & f_0(P_{f_0,1}, \ldots, P_{f_0,\alpha(f_0)}) : \\ & \quad f_1(P_{f_1,1}, \ldots, P_{f_1,\alpha(f_1)}), \ldots, f_n(P_{f_n,1}, \ldots, P_{f_n,\alpha(f_n)}), \\ & \quad h_1(P_{h_1,1}, \ldots, P_{h_1,\alpha(h_1)}), \ldots, h_m(P_{h_m,1}, \ldots, P_{h_m,\alpha(h_m)}). \end{aligned} \tag{A.1}$$

where $f_0 \in N, f_1, \ldots, f_n \in N \cup T,\ h_1, \ldots, h_m \in R$,
$P_{f_0,1}, \ldots, P_{h_m,\alpha(h_m)} \in V$ and
$f_0 \to f_1 \ldots f_n \in P$.

N, T and R are pairwise disjoint. The *arity* α maps each symbol (element of $N \cup T \cup R$) into $\mathcal{N}_0$ (number of parameters of a function). Each $s(P_1, \ldots, P_{\alpha(s)})$ occurring on the left-hand side of some production rule pattern is a *start element* of the GSF. ◇

We also use the term GSF rule instead of production rule pattern. Variables are the only kind of parameters so far. It is possible to extend the basic formalism to cope with constants, tupels, terms and sequences. We instrument a special notation, where the relational symbols are marked by the symbol &. Thereby, special declarations of grammar symbols and relational symbols are not required. GSF rules are usually labelled by a tag.

Example A.3.1
Consider the following GSF rules modelling syntax, static semantics and AST construction for sequences of assignments as in an imperative programming language.

$$\begin{array}{lcll}
\textit{statements}(\mathsf{ST},\mathsf{STM}) & \colon & \textit{statement}(\mathsf{ST},\mathsf{STM}_1), & [\mathsf{concat}] \\
 & & \textit{statements}(\mathsf{ST},\mathsf{STM}_2), & \\
 & & \&\ \textit{concat}(\mathsf{STM}_1,\mathsf{STM}_2,\mathsf{STM}). & \\
\textit{statements}(\mathsf{ST},\mathsf{STM}) & \colon & \&\ \textit{skip}(\mathsf{STM}). & [\mathsf{skip}] \\
\textit{statement}(\mathsf{ST},\mathsf{STM}) & \colon & \textit{identifier}(\mathsf{ID}), & [\mathsf{assign}] \\
 & & \&\ \textit{lookup}(\mathsf{ST},\mathsf{ID},\mathsf{T}_{LHS}), & \\
 & & \textit{expression}(\mathsf{ST},\mathsf{T}_{RHS},\mathsf{EXP}), & \\
 & & \&\ \textit{assignable}(\mathsf{T}_{LHS},\mathsf{T}_{RHS}), & \\
 & & \&\ \textit{assign}(\mathsf{ID},\mathsf{EXP},\mathsf{STM}). &
\end{array}$$

statements, *statement*, *expression* $\in N$, *identifier* $\in T$, *concat*, *skip*, *lookup*, *assignable*, *assign* $\in R$, ST, STM, STM_1, STM_2, T_{LHS}, T_{RHS}, ID, $\mathsf{EXP} \in V$. The parameters concern symbol table propagation (ST), types of identifiers and expressions (T_{LHS}, T_{RHS}), terminal attribution for identifiers (ID), abstract representations of expressions and statements (STM, STM_1, STM_2, EXP). The relational formula & *lookup*(ST, ID, T_{LHS}) models a symbol table lookup to retrieve the type associated with the variable identifier. The relational formula & *assignable*($\mathsf{T}_{LHS}, \mathsf{T}_{RHS}$) models a test if types of LHS and RHS of an assignment are compatible. All the other relational formulae are concerned with the construction of ASTs. ◇

In many applications it is comfortable to distinguish different groups of relational symbols. Thus, we use a form $\&_p$ to prefix relational formulae. p can be regarded as a kind of qualifier in the sense of a module name, e.g. in the above example it makes sense to consider one group of relational formulae related to static semantics, whereas another group concerns constructions of ASTs.

A GSF interpretation defines the domains of the parameter positions and assigns relations between these domains to the relational symbols.

Definition A.3.2
Let $GS = \langle B, R, V, \alpha, PP \rangle$ be a GSF schema. A *GSF interpretation* for GS is a tuple $IP = \langle \mathcal{D}, \delta_p, \delta_V, \rho \rangle$, where

- $\mathcal{D}$ is a family of domains,
- δ_p is a function assigning to the i-th parameter position of a symbol f a domain $\delta_p(f, i) \in \mathcal{D}$,
- δ_V is a function assigning to each variable $v \in V$ a domain $\delta_V(v) \in \mathcal{D}$,
- ρ is a function associating with each element $f \in R$ an $\alpha(f)$-ary relation $\rho(f) \subseteq \delta_p(f, 1) \times \cdots \times \delta_p(f, \alpha(f))$.

For all production rule patterns $p \in PP$ and all elements $f(P_1, \ldots, P_{\alpha(f)})$ occurring in p with $P_1, \ldots, P_{\alpha(f)} \in V$ the following conditions have to be satisfied for $i = 1, \ldots, \alpha(f)$: $P_i \in V \Rightarrow \delta_V(P_i) = \delta_p(f, i)$. ◇

Example A.3.2
We give an interpretation for the above GSF schema. $\mathcal{ST}$ is the domain of symbol tables, $\mathcal{T}$ is the domain of types (i.e. type expressions), $\mathcal{I}$ is the domain of identifiers. $\mathcal{E}$, $\mathcal{C}$ are the domains of abstract representations of expressions and statements respectively. The following tables define δ_p and δ_V:

f	$\alpha(f)$	$\delta_p(f,1)$	$\delta_p(f,2)$	$\delta_p(f,3)$
statements	2	$\mathcal{ST}$	$\mathcal{C}$	
statement	2	$\mathcal{ST}$	$\mathcal{C}$	
expression	3	$\mathcal{ST}$	$\mathcal{T}$	$\mathcal{E}$
identifier	1	$\mathcal{I}$		
skip	1	$\mathcal{C}$		
concat	3	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{C}$
lookup	3	$\mathcal{ST}$	$\mathcal{I}$	$\mathcal{T}$
assignable	2	$\mathcal{T}$	$\mathcal{T}$	
assign	3	$\mathcal{I}$	$\mathcal{E}$	$\mathcal{C}$

v	$\delta_V(v)$
ST	$\mathcal{ST}$
STM	$\mathcal{C}$
STM$_1$	$\mathcal{C}$
STM$_2$	$\mathcal{C}$
T$_{LHS}$	$\mathcal{T}$
T$_{RHS}$	$\mathcal{T}$
ID	$\mathcal{I}$
EXP	$\mathcal{E}$

Let us assume the following definition of the domain of symbol tables.

$$\mathcal{ST} = \mathcal{I} \to \mathcal{T}$$

The relations associated with the relational symbols are defined as follows:

$$\begin{aligned}
\rho(\mathit{skip}) &= \{\underline{\mathit{skip}}\} \\
\rho(\mathit{concat}) &= \{(c_1,c_2,c_3)|c_i \in \mathcal{C},\ \text{for } i=1,2,3, c_3 = \underline{\mathit{concat}}(c_1,c_2)\} \\
\rho(\mathit{lookup}) &= \{(st,i,t)|st \in \mathcal{ST}, i \in \mathcal{I}, t \in \mathcal{T}, t = st(i)\} \\
\rho(\mathit{assignable}) &= \{(t,t)|t \in \mathcal{T}\} \\
\rho(\mathit{assign}) &= \{(i,e,c)|i \in \mathcal{I}, e \in \mathcal{E}, c \in \mathcal{C}, c = \underline{\mathit{assign}}(i,e)\}
\end{aligned}$$

skip, *concat* and *assign* are interpreted as term constructors in the sense of a term algebra. Thus, the domain $\mathcal{C}$ is regarded as a domain of terms, where $\underline{\mathit{skip}}$, $\underline{\mathit{concat}}$ and $\underline{\mathit{assign}}$ are the corresponding term constructors. The interpretation for *lookup* is suitable to lookup the type of a variable in a symbol table. The interpretation of *assignable* is fixed in a way that types of the LHS and the RHS in an assignment must be *equal*. ♢

Definition A.3.3
A *GSF* is a pair $G = \langle GS, IP \rangle$, where GS is a GSF schema and IP is a GSF interpretation for GS. ♢

To generate a word by means of context-free derivation, first the production rule patterns have to be turned into context-free production rules. Each variable occurring in a production rule pattern is consistently substituted by a value from its corresponding domain. This substitution process is controlled by the relations occurring in the production rule pattern.

Definition A.3.4
Let $G = \langle GS, IP \rangle$ be a GSF with $GS = \langle B, R, V, \alpha, PP \rangle$ and $IP = \langle \mathcal{D}, \delta_p, \delta_V, \rho \rangle$.

$$\mathcal{I}(f) = \{f(d_1, \ldots, d_{\alpha(f)}) | d_i \in \delta_p(f, i), i = 1, \ldots, \alpha(f)\}$$

is called the set of *instances* of the symbol f [1].

$$\mathcal{I}(A) = \bigcup_{f \in A} \mathcal{I}(f) \text{ where } A \subseteq N \cup T \cup R$$

$F_0 \to F_1 \ldots F_n$ is a *context-free production rule derived from* $p \in PP$ of form (A.1) if

$$\begin{array}{l} f_0(d_{f_0,1}, \ldots, d_{f_0,\alpha(f_0)}) : \\ \quad f_1(d_{f_1,1}, \ldots, d_{f_1,\alpha(f_1)}), \ldots, f_n(d_{f_n,1}, \ldots, d_{f_n,\alpha(f_n)}), \\ \quad h_1(d_{h_1,1}, \ldots, d_{h_1,\alpha(h_1)}), \ldots, h_m(d_{h_m,1}, \ldots, d_{h_m,\alpha(h_m)}). \end{array} \tag{A.2}$$

- is the result of the consistent substitution of each variable $v \in V$ occurring in p by a value from $\delta_V(v)$,
- $\langle d_{h_i,1}, \ldots, d_{h_i,\alpha(h_i)} \rangle \in \rho(h_i)$ for $i = 1, \ldots, m$ and
- $F_0, \ldots, F_n$ are the instances $f_0(d_{f_0,1}, \ldots, d_{f_0,\alpha(f_0)}), \ldots, f_n(d_{f_n,1}, \ldots, d_{f_n,\alpha(f_n)})$ of the symbols $f_0, \ldots, f_n$.

$\mathcal{I}_{cf}(p)$ is the set of context-free production rules derived from $p \in PP$; $\mathcal{I}_{cf}(PP)$ is the set of context-free production rules derived from the production rule patterns in PP. ♢

Definition A.3.5
Let $G = \langle GS, IP \rangle$ be a GSF with $GS = \langle B, R, V, \alpha, PP \rangle$ and $IP = \langle \mathcal{D}, \delta_p, \delta_V, \rho \rangle$. The binary relation $\Rightarrow$ on $\mathcal{I}(N \cup T)^*$ is defined as follows:

$$uFw \Rightarrow uvw \iff F \to v \in \mathcal{I}_{cf}(PP)$$

where $uw \in \mathcal{I}(N \cup T)^*$. The relation $\Rightarrow^+$ is the transitive closure of $\Rightarrow$, $\Rightarrow^*$ is the reflexive closure of $\Rightarrow^+$. Let be s the start symbol of the basic grammar of GS, $t^* \in \mathcal{I}(T)^*$. t^* is a *word* generated by G, if

$$\exists F_0 \in \mathcal{I}(s) : F_0 \Rightarrow^* t^*.$$

$\mathcal{L}(G)$ denotes the *language* generated by the GSF G. It contains all the words generated by G. ♢

Note that $L(G) \subseteq L(B)$, i.e. the language generated by the GSF is usually some subset of the language generated by the underlying context-free grammar.

Usually, there is an infinite number of derived production rule patterns. Thus, for the analysis of a given string of terminal symbols it is impractical to use a context-free parsing technique with the derived context-free production rules. There are two general

[1] In the case of $\alpha(f) = 0$ the parentheses are omitted, thus $\mathcal{I}(f) = \{f\}$

approaches to solve this problem. The first approach parses the program according to the basic context-free grammar and then calculates a variable substitution satisfying all the corresponding relations. The second approach incorporates the variable substitution and the evaluation of the relations into the parsing process based on a directed evaluation schema. Thereby, semantics-directed parsing (or parsing directed by attribute values) can be modelled.

The second approach is more general because in a technical sense it permits parsing to depend on attribute values. The approach is formalized in the sequel. We start by refining the definition of a GSF schema by dividing parameter positions into input and output positions and requiring a certain data flow criterion.

Definition A.3.6
Let $GS = \langle B, R, V, \alpha, PP \rangle$ be a GSF schema, β is a function defining *directions* (also called *modes*) for GS, if $\beta(f,i) \in \{\downarrow, \uparrow\}$ for $f \in (N \cup T \cup R)$, $i = 1, \ldots, \alpha(f)$. If $\beta(f,i) = \downarrow$, i is an *input position* of f, whereas for $\beta(f,i) = \uparrow$, i is an *output position.* ♢

In the Knuthian terminology, input and output positions of terminals and nonterminals can be called inherited and synthesized positions respectively.

Definition A.3.7
Let $GS = \langle B, R, V, \alpha, PP \rangle$ be a GSF schema, β is a function defining *directions* for GS. Suppose that for each $f \in (N \cup T \cup R)$, the partitioning of parameter positions by β is written as follows: $i(f,1)$, …, $i(f,i_f)$ are the input positions of f, whereas $o(f,1)$, …, $o(f,o_f)$ are the output positions of f.

Each occurrence of a variable in a production rule pattern r of the form (A.1) is classified either as *applied* or *defining occurrence.* The sets A_r of applied occurrences and D_r of defining occurrences are defined as follows:

$$\begin{aligned} A_r &= (\bigcup_{k=1}^{n} \{P_{f_k,i(f_k,1)}, \ldots, P_{f_k,i(f_k,i_{f_k})}\}) \cup \{P_{f_0,o(f_0,1)}, \ldots, P_{f_0,o(f_0,o_{f_0})}\} \\ D_r &= \{P_{f_0,i(f_0,1)}, \ldots, P_{f_0,i(f_0,i_{f_0})}\} \cup (\bigcup_{k=1}^{n} \{P_{f_k,o(f_k,1)}, \ldots, P_{f_k,o(f_k,o_{f_k})}\}) \end{aligned}$$

♢

Applied occurrences are expected to be “computed” in terms of defining positions. These terms are used in much the same way in extended attribute grammars [WM77]. In attribute grammars, notions like *used* and *defined attribute occurrences* are defined. These terms are tuned towards named attributes rather than a position-oriented framework as in our case.

Definition A.3.8
Let $GS = \langle B, R, V, \alpha, PP \rangle$ be a GSF schema, β is a function defining *directions* for GS. GS is a *β-directed GSF schema*, if the following property holds $A_r \subseteq D_r$, i.e. each variable

occurring on an input position in the rule body or an output position in the rule head occurs on an output position in the rule body or an input position in the rule head. $\diamondsuit$

The notion of a β-directed GSF schema can be used for a deterministic evaluation strategy in the following manner. If the interpretations of relational symbols allow output positions to be computed effectively from the input positions, the application of the interpretations is delayed until all input positions have been computed.

Example A.3.3
The GSF schema from Example A.3.1 is a β-directed GSF schema with regard to the following β.

f	$\beta(f,1)$	$\beta(f,2)$	$\beta(f,3)$	$i(f,1)$, …, $i(f,i_f)$	$o(f,1)$, …, $o(f,o_f)$
statements	$\downarrow$	$\uparrow$		1	2
statement	$\downarrow$	$\uparrow$		1	2
expression	$\downarrow$	$\uparrow$	$\uparrow$	1	2,3
identifier	$\uparrow$				1
lookup	$\downarrow$	$\downarrow$	$\uparrow$	1,2	3
assignable	$\downarrow$	$\downarrow$		1,2	
skip	$\uparrow$				1
concat	$\downarrow$	$\downarrow$	$\uparrow$	1,2	3
assign	$\downarrow$	$\downarrow$	$\uparrow$	1,2	3

$\diamondsuit$

The following definition captures what is meant by "output positions of a parameterized relational symbol can effectively be computed from the input positions by the interpretation of the relational symbol".

Definition A.3.9
A *GSF* $G = \langle GS, IP \rangle$ is β-directed GSF, if GS is a β-GSF schema, and the following property holds for the GSF interpretation IP: For each $r \in R$, $\rho(r)$ can be described by a $\rho^{\rightarrow}(r)$ as follows:

$$\begin{array}{l}\forall d_1^{\downarrow} \in \delta_p(r, i(r,1)), \ldots, d_{i_r}^{\downarrow} \in \delta_p(r, i(r,i_r)) : \\ \rho^{\rightarrow}(r)\langle d_1^{\downarrow}, \ldots, d_{i_r}^{\downarrow} \rangle = \langle d_{1,1}^{\uparrow}, \ldots, d_{1,o_r}^{\uparrow} \rangle, \langle d_{2,1}^{\uparrow}, \ldots, d_{2,o_r}^{\uparrow} \rangle, \ldots \text{ such that} \\ \langle d_1, \ldots, d_{\alpha(r)} \rangle \in \rho(r) \text{ with } d_{i(r,1)} = d_1^{\downarrow}, \ldots, d_{i(r,i_r)} = d_{i_r}^{\downarrow} \Leftrightarrow \\ \exists k : d_{o(r,1)} = d_{k,1}^{\uparrow}, \ldots, d_{o(r,o_r)} = d_{k,o_r}^{\uparrow}\end{array}$$

$\diamondsuit$

An even more restricted variant of ρ should be mentioned, that is to say the output positions of a parameterized relational symbol are *uniquely* defined for given input parameters. That is similar to the interpretation of semantic function symbols in an ordinary AG by *functions.*

This kind of GSF schemata has been suggested in [RL89] as *functional* GSFs. A slightly more general variant of ρ requires a *finite set* of possible output substitutions. This form of GSFs has been suggested in [Har97] (executable GSFs). The above definition copes with an arbitrary number of output substitutions.

Instead of defining β, a notational variant for production rule patterns is suggested. Assuming for each $f \in (N \cup T \cup R)$ a $\gamma(f)$ such that $\beta(f,1) =\downarrow$, ..., $\beta(f,\gamma(f)) =\downarrow$, $\beta(f,\gamma(f)+1) =\uparrow$, ..., $\beta(f,\alpha(f)) =\uparrow$, then the $\gamma(f)$ and thereby β can be obtained if the production rule patterns are of the following form:

$$\begin{aligned}
&f_0(P_{f_0,1},\ldots,P_{f_0,\gamma(f_0)}) \to (P_{f_0,\gamma(f_0)+1},\ldots,P_{f_0,\alpha(f_0)}) : \\
&\quad f_1(P_{f_1,1},\ldots,P_{f_1,\gamma(f_1)}) \to (P_{f_1,\gamma(f_1)+1},\ldots,P_{f_1,\alpha(f_1)}),\ldots, \\
&\quad f_n(P_{f_n,1},\ldots,P_{f_n,\gamma(f_n)}) \to (P_{f_n,\gamma(f_n)+1},\ldots,P_{f_n,\alpha(f_n)}), \\
&\quad h_1(P_{h_1,1},\ldots,P_{h_1,\gamma(h_1)}) \to (P_{h_1,\gamma(h_1)+1},\ldots,P_{h_1,\alpha(h_1)}),\ldots, \\
&\quad h_m(P_{h_m,1},\ldots,P_{h_m,\gamma(h_m)}) \to (P_{h_m,\gamma(h_m)+1},\ldots,P_{h_m,\alpha(h_m)}).
\end{aligned} \tag{A.3}$$

Thus, the $\to$ is used to separate input and output positions.

Example A.3.4
Example A.3.1 is rewritten in the arrow notation. Moreover, we also use different prefixes to point out the different groups of relational symbols.

$$\begin{aligned}
&\mathit{statements}(\mathsf{ST}) \to (\mathsf{STM}) \;:\; \mathit{statement}(\mathsf{ST}) \to (\mathsf{STM}_1), && [\mathsf{concat}]\\
&\qquad \mathit{statements}(\mathsf{ST}) \to (\mathsf{STM}_2),\\
&\qquad \&_{ast}\ \mathit{concat}(\mathsf{STM}_1,\mathsf{STM}_2) \to (\mathsf{STM}).\\
&\mathit{statements}(\mathsf{ST}) \to (\mathsf{STM}) \;:\; \&_{ast}\ \mathit{skip} \to (\mathsf{STM}). && [\mathsf{skip}]\\
&\mathit{statement}(\mathsf{ST}) \to (\mathsf{STM}) \;:\; \mathit{identifier} \to (\mathsf{ID}), && [\mathsf{assign}]\\
&\qquad \mathit{expression}(\mathsf{ST}) \to (\mathsf{T}_{RHS},\mathsf{EXP}),\\
&\qquad \&_{static}\ \mathit{lookup}(\mathsf{ST},\mathsf{ID}) \to (\mathsf{T}_{LHS}),\\
&\qquad \&_{static}\ \mathit{assignable}(\mathsf{T}_{LHS},\mathsf{T}_{RHS}),\\
&\qquad \&_{ast}\ \mathit{assign}(\mathsf{ID},\mathsf{EXP}) \to (\mathsf{STM}).
\end{aligned}$$

◇

There are various choices for restricting the data flow according to evaluation strategies. A simple example are L-attributed AGs, where attibute evaluation coincides with a single left-ro-right traversal of the syntax tree.

Definition A.3.10
Let $GS = \langle B, R, V, \alpha, PP\rangle$ be a GSF schema, β is a function defining *directions* for GS. GS is L-attributed, if there is some permutation g_1, ..., g_q, $q = n + m$ of f_1, ..., f_n, h_1, ..., h_m such that the relative order of f_1, ..., f_n is preserved and the following properties hold:

$$\begin{aligned}
\{P_{g_s,i(g_s,1)},\ldots,P_{g_s,i(g_s,i_{g_s})}\} \subseteq & \bigcup_{k=1}^{s-1}\{P_{g_k,o(g_k,1)},\ldots,P_{g_k,o(g_k,o_{g_k})}\} \text{ where } s=1,\ldots,q \\
\cup \;& \{P_{f_0,i(f_0,1)},\ldots,P_{f_0,i(f_0,i_{f_0})}\} \\
\{P_{f_0,o(f_0,1)},\ldots,P_{f_0,o(f_0,o_{f_0})}\} \subseteq & \bigcup_{k=1}^{q}\{P_{g_k,o(g_k,1)},\ldots,P_{g_k,o(g_k,o_{g_k})}\} \\
\cup \;& \{P_{f_0,i(f_0,1)},\ldots,P_{f_0,i(f_0,i_{f_0})}\}
\end{aligned}$$

◇

Using L-attributed GSF schemata, another notation of GSF rules making L-attribution more explicit is straightforward. Parameterized grammar symbols and relational symbols are no longer grouped, but parameterized relational symbols are inserted in such positions that a proper permutation in the sense of Definition A.3.10 is made explicit. Such GSF schemata are *call-correct* in similarity to call-correctness of logic programs with modes (w.r.t. the simple Prolog computation rule for example) [Boy96a]. A formal definition is omitted. The data-flow in a call-correct rule is visualized in Figure A.1.

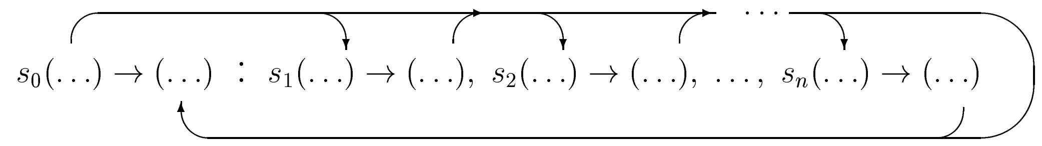

$s_0(\ldots) \to (\ldots) \;:\; s_1(\ldots) \to (\ldots),\; s_2(\ldots) \to (\ldots),\; \ldots,\; s_n(\ldots) \to (\ldots)$
...

Figure A.1: Dependencies from the left to the right

Besides directions, the usability of types will be discussed below. So far typing is only considered at the level of the GSF interpretation. Parameter positions and variables are associated with domains. We can also refine the notion of a GSF schema to cope with types, e.g. many-sorted types.

Definition A.3.11
A *many-sorted GSF schema* is a tuple $GS = \langle B, R, V, D, \tau, \sigma, PP\rangle$, where D is a set of *sorts* (also called names of domains), the function $\tau : (N \cup T \cup R) \to D^*$ associates a *type* (also called *profile*) with every function name, the function $\sigma : V \to D$ associates a *sort* (also called *type*) with every parameter, $\langle B, R, V, \alpha, PP\rangle$ is a a GSF schema such that $\alpha(f) = |\tau(f)|$ for all $f \in (N \cup T \cup R)$, for all $p \in PP$ of form (A.1), for all $f \in \{f_0, f_1, \ldots, f_n, h_1, \ldots, h_m\}$ the following soundness condition must be satisfied:

$$\tau(f) = \langle \sigma(P_{f,1}), \ldots, \sigma(P_{f,\alpha(f)})\rangle$$

◇

GSF interpretations should be restricted for many-sorted GSF schemata in the same way as Σ-algebras are certain algebras satisfying Σ. We omit the corresponding definition.

Obviously, types and directions may be combined. Other concepts can be integrated with the basic GSF formalism as well. The *GS* specification formalism [LRH96] of $^{\Lambda}\Delta_{\Lambda}$, for example, supports a combination of many-sorted types, directions, tagged rules and optional prefixes (qualifier) for relational formulae, constants and terms as further forms of parameters and predefined relational symbols for the support of basic data types. There are further possible extensions concerning overloading, polymorphism or higher-order features.

From a practical point of view it is not so convenient to declare explicitly profiles of symbols and sorts of variables (functions τ and σ). Especially, it can be decided if a suitable σ exists for a given τ. On the other hand, we can also assume a kind of naming discipline for variables, where the stems of the variables are supposed to represent sorts. Actually, this kind of discipline is assumed in the specification framework of $^{\Lambda}\Delta_{\Lambda}$. Thus, neither τ nor σ need to be defined explicitly.

Example A.3.5
Let us derive the signature associated with the directed GSF schema in Example A.3.4. We use the stems of the variables as sorts.

$$\begin{array}{rcl}
\textit{statements} & : & \mathsf{ST} \to \mathsf{STM} \\
\textit{statement} & : & \mathsf{ST} \to \mathsf{STM} \\
\textit{expression} & : & \mathsf{ST} \to \mathsf{T} \times \mathsf{EXP} \\
\textit{identifier} & : & \to \mathsf{ID} \\
\&_{ast}\ \textit{skip} & : & \to \mathsf{STM} \\
\&_{ast}\ \textit{concat} & : & \mathsf{STM} \times \mathsf{STM} \to \mathsf{STM} \\
\&_{static}\ \textit{lookup} & : & \mathsf{ST} \times \mathsf{ID} \to \mathsf{T} \\
\&_{static}\ \textit{assignable} & : & \mathsf{T} \times \mathsf{T} \\
\&_{ast}\ \textit{assign} & : & \mathsf{ID} \times \mathsf{EXP} \to \mathsf{STM}
\end{array}$$

$\diamondsuit$

To conclude the presentation of the GSF formalism, we want to reconstruct below a famous example of Knuth's paper on Attribute Grammars [Knu68]. The example concerns the computation of the decimal value of binary numbers.

Example A.3.6
The underlying context-free grammar of the below GSF schema describes binary numbers $l_1.l_2$, i.e. l_1 and l_2 are sequences of 0's and 1's. The attribution models the computation of the (decimal) value of a binary number. Attributes of the sort *VAL* are computed as the value of (a part of) a binary number. Attributes of the sort *LEN* accumulate the length of the parts l_1 and l_2. Attributes of the sort *SCALE* are inherited to point out the valency of a position. There are the following GSF rules. Note that the same example is shown in some standard notation in Example A.3.7.

$$z \rightarrow (VAL) \;:\; l(SCALE_1) \rightarrow (LEN_1, VAL_1),\ \text{“.”},\ l(SCALE_2) \rightarrow (LEN_2, VAL_2),\ \&\ add(VAL_1, VAL_2) \rightarrow (VAL),\ \&\ zero_{SCALE} \rightarrow (SCALE_1),\ \&\ neg(LEN_2) \rightarrow (SCALE_2). \quad [p_1]$$

$$z \rightarrow (VAL) \;:\; l(SCALE) \rightarrow (LEN, VAL),\ \&\ zero_{SCALE} \rightarrow (SCALE). \quad [p_2]$$

$$l(SCALE) \rightarrow (LEN', VAL_0) \;:\; l(SCALE') \rightarrow (LEN, VAL_1),\ b(SCALE) \rightarrow (VAL_2),\ \&\ add(VAL_1, VAL_2) \rightarrow (VAL_0),\ \&\ inc_{LEN}(LEN) \rightarrow (LEN'),\ \&\ inc_{SCALE}(SCALE) \rightarrow (SCALE'). \quad [p_3]$$

$$l(SCALE) \rightarrow (LEN, VAL) \;:\; b(SCALE) \rightarrow (VAL),\ \&\ one \rightarrow (LEN). \quad [p_4]$$

$$b(SCALE) \rightarrow (VAL) \;:\; \text{“1”},\ \&\ power(SCALE) \rightarrow (VAL). \quad [p_5]$$

$$b(SCALE) \rightarrow (VAL) \;:\; \text{“0”},\ \&\ zero_{VAL} \rightarrow (VAL). \quad [p_6]$$

The GSF interpretation can be described as follows. First, we associate the sorts with some suitable domains:

$$\begin{aligned} VAL &= \mathcal{Q} \text{ (rational numbers)} \\ LEN &= \mathcal{N}_0 \text{ (natural numbers including zero)} \\ SCALE &= \mathcal{Z} \text{ (integer numbers)} \end{aligned}$$

Second, the relational symbols are associated with suitable relations:

$$\begin{aligned} \langle v_1, v_2, v_3 \rangle \in add &\Leftrightarrow v_3 = v_1 + v_2 \\ s \in zero_{SCALE} &\Leftrightarrow s = 0 \\ \langle l, s \rangle \in neg &\Leftrightarrow s = -l \\ \langle l, l' \rangle \in inc_{LEN} &\Leftrightarrow l' = l + 1 \\ \langle s, s' \rangle \in inc_{SCALE} &\Leftrightarrow s' = s + 1 \\ l \in one &\Leftrightarrow l = 1 \\ \langle s, v \rangle \in power &\Leftrightarrow v = 2^s \\ v \in zero_{VAL} &\Leftrightarrow v = 0 \end{aligned}$$

◇

Example A.3.7
The below AG specification developed in the common AG notation is intended to be equivalent to the GSF from Example A.3.6.

$$
\begin{array}{lll}
[p_1] & : & z \to l_1.l_2 \\
 & & \begin{array}{lcl} z.VAL & := & l_1.VAL + l_2.VAL \\ l_1.SCALE & := & 0 \\ l_2.SCALE & := & -l_2.LEN \end{array} \\
[p_2] & : & z \to l \\
 & & \begin{array}{lcl} z.VAL & := & l_1.VAL \\ l.SCALE & := & 0 \end{array} \\
[p_3] & : & l_0 \to l_1\ b \\
 & & \begin{array}{lcl} l_0.VAL & := & l_1.VAL + b.VAL \\ l_0.LEN & := & l_1.LEN + 1 \\ l_1.SCALE & := & l_0.SCALE + 1 \\ b.SCALE & := & l_0.SCALE \end{array} \\
[p_4] & : & l \to b \\
 & & \begin{array}{lcl} l.VAL & := & b.VAL \\ l.LEN & := & 0 \\ b.SCALE & := & l.SCALE \end{array} \\
[p_5] & : & b \to 1 \\
 & & \begin{array}{lcl} b.VAL & := & 2^{b.SCALE} \end{array} \\
[p_6] & : & b \to 0 \\
 & & \begin{array}{lcl} b.VAL & := & 0 \end{array}
\end{array}
$$

◊

A.4 Object-oriented class systems

We establish some basic notions for dealing with class hierarchies and inheritance. These notions are needed for object-oriented attribute grammars. $B \to A$ reads as A is a subclass of B or B is a superclass of A. $(\mathcal{C}, \to)$ denotes a class system, where $\mathcal{C}$ is a set of classes. $\to^+$ denotes the transitive closure of $\to$, whereas $\to^\star$ denotes the reflexive, transitive closure of $\to$. $C \to^+ A$ reads as A is a descendant class of C or C is an ancestor class of A.

$(\mathcal{C}, \to)$ is a cycle-free class system if $A \to^+ A$ holds for no A. Without any further restrictions, multiple inheritance (MI) is captured. Single inheritance (SI) puts the following restriction on the class system: $B \to A \Rightarrow \nexists C \neq B$ with $C \to A$, i.e. $\to$ corresponds to a forest.

A.5 Object-oriented context-free grammars

We define various forms of object-oriented context-free grammars [Kos91].

Definition A.5.1
Let G be a CFG, i.e. a quadrupel $\langle N, T, s, P\rangle$. N is the set of nonterminals, whereas T is the set of terminals. N and T are finite sets such $N \cup T = \emptyset$. $s \in N$ is the start symbol of G. P is a finite set of context-free productions.

1. A CFG is *pseudo-reduced*, if for all nonterminals A either $s \Rightarrow^\star uAv \Rightarrow^\star w$, or $A \Rightarrow^+ B$ and $s \Rightarrow^\star uBv \Rightarrow^\star w$, where $u, v \in (T \cup N)^\star$ and $w \in T^\star$.

2. A pseudo-reduced, cycle-free CFG is *MI-structured*, if for each $A \in N$ (1) or (2) holds:

   (1) $|\{p \in P \mid p = (A \to v \text{ for some } v\}| = 1$ and $(A \to v) \in P$ implies $v \in (T \cup N)^\star$

   (2) $(A \to v) \in P$ implies $v \in N$

3. An MI-structured CFG is *SI-structured*, if $(A \to B) \in P$ and $(C \to B) \in P$ implies $A = C$.

4. An MI(SI)-structured CFG is *strongly MI(SI)-structured* if it is reduced.

$\diamondsuit$

Example A.5.1
The following rules are part of a CFG for the syntax of a simple imperative language.

$$\begin{array}{rcl}
\mathit{Statement} & \to & \mathit{Identifier} \text{ “ := ” } \mathit{Expression} \\
\mathit{Statement} & \to & \mathit{Identifier} \text{ “(”} \mathit{Expression} \text{“)”} \\
\mathit{Statement} & \to & \text{“While” } \mathit{Expression} \text{ “Do” } \mathit{Statement} \\
 & \ldots &
\end{array}$$

To obey the above mentioned properties for context-free grammars to be sensible from the object-oriented point of view, the rules have to be transformed as follows:

$$\begin{array}{rcl}
\mathit{Statement} & \to & \mathit{AssignStatement} \\
\mathit{Statement} & \to & \mathit{ProcedureCall} \\
\mathit{Statement} & \to & \mathit{WhileStatement} \\
\mathit{AssignStatement} & \to & \mathit{Identifier} \text{ “ := ” } \mathit{Expression} \\
\mathit{ProcedureCall} & \to & \mathit{Identifier} \text{ “(”} \mathit{Expression} \text{“)”} \\
\mathit{WhileStatement} & \to & \text{“While” } \mathit{Expression} \text{ “Do” } \mathit{Statement} \\
 & \ldots &
\end{array}$$

$\diamondsuit$

A.6 Object-oriented attribute grammars

Here we want to present some examples of object-oriented AGs. The computation of some forms of expressions serve as a running example.

Example A.6.1
The computation of expressions is specified in the *Ag* notation of *Cocktail*.

$$\begin{array}{lcll}
\mathit{Expr} & = & [\mathit{Value} : \mathit{INTEGER}] & \{\mathit{Value} := 0\} \\
< & & & \\
\ \mathit{Add} & = & \mathit{Lop} : \mathit{Expr} \text{ “ + ” } \mathit{Rop} : \mathit{Expr} & \{\mathit{Value} := \mathit{Lop} : \mathit{Value}\ +\ \mathit{Rop} : \mathit{Value}\}. \\
\ \mathit{Sub} & = & \mathit{Lop} : \mathit{Expr} \text{ “ − ” } \mathit{Rop} : \mathit{Expr} & \{\mathit{Value} := \mathit{Lop} : \mathit{Value}\ -\ \mathit{Rop} : \mathit{Value}\}. \\
\ \mathit{Const} & = & \mathit{Integer} & \{\mathit{Value} := \mathit{Integer} : \mathit{Value}\}. \\
>. & & & \\
\mathit{Integer} & : & [\mathit{Value} : \mathit{INTEGER}]. &
\end{array}$$

The CFG is SI-structured due to the nested notation of subclasses. Attribute declarations are enclosed in square brackets (the role of either an ancestral or synthesized attribute is derived from the context), whereas semantic rules are enclosed in braces. For completeness, the underlying CFG is shown:

$$\begin{array}{lcl}
\mathit{Expr} & \rightarrow & \mathit{Add} \mid \mathit{Sub} \mid \mathit{Const} \\
\mathit{Add} & \rightarrow & \mathit{Expr} \text{ “ + ” } \mathit{Expr} \\
\mathit{Sub} & \rightarrow & \mathit{Expr} \text{ “ − ” } \mathit{Expr} \\
\mathit{Const} & \rightarrow & \mathit{Integer}
\end{array}$$

The use of attribute inheritance is obvious. The synthesized attribute *Value* is declared for *Expr*, only. It is inherited to *Add*, *Sub*, *Const*. That is not an impressive result because these nonterminals have been introduced to adhere to the style of object-oriented context-free grammar. Inheritance of semantic rules is shown only in the sense of a pedagogical example: The computed value of an expression is 0 by default. This definition has to be overridden in all concrete subclasses. ◇

It is essential for object-orientation in CFGs to distinguish nonterminals defined by chain productions and nonterminals defined by a structural specification. For completeness, we mention the *Mjølner/Orm* terminology for class definitions using a finer granularity:

- *Abstract classes* correspond to superclass nonterminals.
- *Structured classes* correspond to nonterminals with a structural specification.
- *Case classes* are a special feature supporting the inheritance of syntactic patterns.

Structured classes are further divided into

- *construction* classes specified by a sequence of components,
- *list* classes for lists of components of the same kind and
- *lexeme* classes for lexical items.

Example A.6.2
Example A.6.1 is rewritten in the style of *Mjølner/Orm*.

$<Expr>$ ::= **Abstract**
  Syn *Value* : *integer*;
  Value := 0;
$<Const>$: $<Expr>$ ::= **Lexeme**
  Value := ... integer value of the constant ...;
$<BinOp>$: $<Expr>$ ::= $\{<left : Expr>$ & $<right : Expr>\}$
$<Add>$: $<BinOp>$ ::= **Case**
  Value := $left.Value + right.Value$;
$<Sub>$: $<BinOp>$ ::= **Case**
  Value := $left.Value - right.Value$;

Using case classes we obtain a CFG which slightly differs from that in Example A.6.1. Case classe are useful to point out the common structure of binary addition and subtraction.

$$\begin{array}{lcl} Expr & = & BinOp \mid Const \\ BinOp & = & Add \mid Sub \\ Add & = & Expr\ Expr \\ Sub & = & Expr\ Expr \end{array}$$

◇

A.7 Action semantics

The specification of a simple imperative language SIMPL is presented below in order to complete the discussion of action semantics in Subsection 4.3.2. The specification of SIMPL has been taken from [Mos96].

```
module: Abstract Syntax. grammar:

 (*)  Stmt   =  [[Id ":=" Expr]]
              |  [["if" Expr "then" Stmts "else" Stmts]]
              |  [["while" Expr "do" Stmts]].
 (*)  Stmts  =  <Stmt <";" Stmt >*>.
 (*)  Expr   =  Num | Id | [[ Expr Op Expr ]].
 (*)  Op     =  "+" | "/=".
 (*)  Num    =  [[digit+]].
 (*)  Id     =  [[letter (letter|digit)*]].

endgrammar. closed. endmodule: Abstract Syntax.
```

Figure A.2: Abstract syntax of the SIMPL language

- Figure A.2 defines the abstract syntax of SIMPL.

- Figure A.3 provides the equations defining the semantics of SIMPL.
- Figure A.4 specializes the action notation for the action semantics of SIMPL.

```
module: Semantic Functions.  needs: Abstract Syntax, Semantic Entities.
introduces: execute_, evaluate_, the result of_.
variables: I:Id; N:Num; E,E1,E2:Expr; O:Op; S:Stmt; S1, S2:Stmts.

 (*)   execute_ :: Stmts -> action[completing|diverging|storing].
 [1:]  execute [[I ":=" E]] = evaluate E then
         store the given number in the cell bound to I.
 [2:]  execute [["if" E "then" S1 "else" S2]] = evaluate E then (
         ( check the given truth-value and then execute S1) or
         ( check not the given truth-value and then execute S2)).
 [3:]  execute [["while" E "do" S1]] = unfolding ( evaluate E then (
         ( check the given truth-value and then execute S1 and then unfold) or
         ( check not the given truth-value))).
 [4:]  execute <S ";" S2> = execute S and then execute S2.

 (*)   evaluate_ :: Expr -> action[giving a value].
 [5:]  evaluate N = give decimal N.
 [6:]  evaluate I = give the number bound to I or
         give the number stored in the cell bound to I.
 [7:]  evaluate [[E1 O E2]] = (evaluate E1 and evaluate E2)
         then give the result of O.

 (*)   the result of_ :: Op -> yielder[of a value] [using given (value, value)].
 [8:]  the result of "+" = the number yielded by the sum of
         (the given number#1, the given number#2).
 [9:]  the result of "/=" = not (the given value#1 is the given value#2).
endmodule: Semantic Functions.
```

Figure A.3: Action semantics of the SIMPL language

```
module: Semantic Entities.  includes: Action Notation.
introduces: value, number.

 (*)   token      =    string.
 (*)   bindable   =    cell | number.
 (*)   storable   =    number.
 (*)   value      =    number | truth-value.
 (*)   number     =<   integer.
endmodule: Semantic Entities.
```

Figure A.4: Specializing action notation for SIMPL semantics

A.8 Extensible denotational semantics

In Subsection 4.3.4 we commented on the style of extensible denotational semantics [CF94]. Some more details concerning this style of semantics are provided in this section. First let us consider the semantic framework. The meaning functions P of a complete program and $\mathcal{M}$ of a program phrase have the following profiles:

$$\begin{array}{rcl} \mathcal{P} & : & \mathit{Prog} \to ((V \oplus E) \otimes R) \\ \mathcal{M} & : & \mathit{Expr} \to \mathit{Env} \to C \end{array}$$

Thus, the interpretation of a program returns either a value (V) or an error (E) paired with the final resources (R). In general, the domain of value V is a sum of domains constructed from V and the domain of computations C. This is indicated by the following notation:

$$V = \Sigma_1(V, C)$$

For Pure Scheme, V contains numbers, Boolean values, functions from values to computations and $\perp$ as the denotation of the diverging expression. The domain E of errors can be assumed as follows:

$$E = \{\mathit{err}\}_\perp$$

The domain of resources R can be regarded as product of domains constructed from V, C and others as indicated by the following notation:

$$R = \Pi_1(V, C, \ldots)$$

Environments are functions from variables to values. The domain of denotations for phrases consists of two disjoint pieces: the sub-domain of value denotations V and the sub-domain of effect messages (effects):

$$C = \mathit{inV}(V) \oplus \overbrace{\mathit{inFX}(\underbrace{(V \to C)}_{\text{handles}} \otimes \underbrace{A}_{\text{actions}})}^{\text{effect messages}}$$

The action component A is a sum of domains built from V and C.

$$A = \mathit{inE}(E) \oplus \Sigma_2(V, C)$$

Here the error action is included into A as the most basic effect. The meaning of a complete program is specified as follows:

$$\mathcal{P}[\![P]\!] = \mathit{admin}(\mathcal{M}[\![P]\!]\perp, r_0),$$

where $\perp$ denotes the empty environment and r_0 denotes the initial resources. The basic definition of the administrator consists of the following clauses:

$$\begin{array}{rcl} \mathit{admin} & : & C \times R \to ((C \oplus E) \otimes R) \\ \mathit{admin}(\perp, r) & = & \perp \\ \mathit{admin}(\mathit{inV}(v), r) & = & \langle v, r \rangle \\ \mathit{admin}(\mathit{inFX}(k, \mathit{inE}(\mathit{err})), r) & = & \langle \mathit{err}, r \rangle \end{array}$$

The first clause concerns converging programs. The second clause describes normal termination, that is a value has been computed. The third clause is applied if the evaluation fails due to an error. Adding actions for extensions, new clauses will become relevant. They define how actions are processed accessing the resources and that the handle possibly is invoked to continue the computation.

Semantic domains

$$V \quad = \quad inN(\mathcal{N}_\bot) \oplus inB(\{true, false\}_\bot) \oplus inP((V \to C))$$

Semantic functions

$$\begin{array}{rcl}
\mathcal{M}[\![n]\!]\rho & = & inV(inN(n)) \\
\dots & & \\
\mathcal{M}[\![x]\!]\rho & = & inV(\rho(x)) \\
\mathcal{M}[\![\lambda x.e]\!]\rho & = & inV(inP(\lambda d : V.\mathcal{M}[\![e]\!]\rho[x/d])) \\
\mathcal{M}[\![e_1 e_2]\!]\rho & = & handler(\mathcal{M}[\![e_1]\!]\rho) \\
& & \quad (\lambda f : V.handler(\mathcal{M}[\![e_2]\!]\rho) \\
& & \quad\quad (\lambda a : V.\text{case } f \text{ of} \\
& & \quad\quad\quad [inP(g) \Rightarrow g(a)] \\
& & \quad\quad\quad [g \Rightarrow inAC(inE(err))]) \\
\dots & &
\end{array}$$

Figure A.5: An extended direct semantics of Pure Scheme

We start with two basic constructs Ω denoting the diverging expressions and *err* to signal an error. Their denotations are defined as follows:

$$\begin{array}{rcl}
\mathcal{M}[\![\Omega]\!]\rho & = & inV(\bot) \\
\mathcal{M}[\![err]\!]\rho & = & inAC(inE(err))
\end{array}$$

Here $inAC(a)$ is an abbreviation for $inFX(inV, a)$.

Sub-phrases of complex phrases are evaluated via recursive calls to the interpreter. Since the result of such a recursive call is a computation, it is necessary to inspect the tag of the result. If it is a plain value, the value component can be consumed locally. If it is an effect, however, it must be propagated to the central administrator. To deal with this situation uniformly, the function *handler* mapping a computation and the consumer of its eventual value to computations is introduced:

$$\begin{array}{rcl}
handler & : & C \to \overbrace{(V \to C)}^{\text{consumer}} \to C \\
handler(\bot)f & = & \bot \\
handler(inV(v))f & = & f(v) \\
handler(inFX(k,p))f & = & inFX(\lambda v : V.handler(k(v))f), p)
\end{array}$$

Essentially, the definition makes sure that an effect message is propagated from *handler* to *handler* until it eventually reaches the administrator.

Refer to Figure A.5 for the semantics of Pure Scheme. To add, for example, reference cells to Pure Scheme, essentially three new actions have to be introduced:

- $\mathit{inRef}(V)$, which represents an allocation message;
- $\mathit{inSet}(L, V)$, which represents an update message;
- $\mathit{inDer}(L)$, which represents a dereferencing message.

Here L denotes the domain of locations. The meaning function $\mathcal{M}$ is straightforwardly extended for the corresponding constructs which essentially delegate the interpretation to the administrator (i.e. the function *admin*). The adminstrator also has to be extended to perform the new actions on the resources (the state). Constructs for programming with first-class continuation objects are added in [CF94] in the same orthogonal manner. The equations of the meaning function $\mathcal{M}$ need never to be changed.

A.9 Stepwise refinement

In Subsection 4.4.1 we commented on stepwise refinement in logic programming. In this section, some more details are provided. Most of the following definitions and explanations are taken from [Trc93].

The operators for stepwise refinement of logic programs are defined as follows:

unfold Let P be a program, $c : A \leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n$ a clause in P. Let $c_j, 1 \leq j \leq m$ be all the clauses in P where there exists $\theta_j = \mathit{mgu}(B^j, A_i)$, $c_j : B^j \leftarrow B^j_1, \ldots, B^j_h$.
Define $c'_j : (A \leftarrow A_1, \ldots, A_{i-1}, B^j_1, \ldots, B^j_h, A_{i+1}, \ldots, A_n)\theta_j$.

Then $\mathit{unfold}(P, c, A_i) = (P - \{c\}) \cup \{c'_j | 1 \leq j \leq m\}$.

fold Let P be a program, $c : A \leftarrow B_1, \ldots, B_i, A_1, \ldots, A_k, B_{i+1}, \ldots, B_n$ and $d : B \leftarrow A'_1, \ldots, A'_k$, $k \geq 1$, be clauses in P. Let $\theta = \mathit{mgu}((A_1, \ldots, A_k), (A'_1, \ldots, A'_k))$.
Define $c' : A \leftarrow B_1, \ldots, B_i, B\theta, B_{i+1}, \ldots, B_n$.

Then $\mathit{fold}(P, c, (A_1, \ldots, A_k)) = (P - \{c\}) \cup \{c'\}$.

prune Let P be a program, c a clause in P.

Then $\mathit{prune}(P, c) = P - \{c\}$.

add Let P be a program, c a clause in P.

Then $\mathit{add}(P, c) = P \cup \{c\}$.

thin Let $c : A \leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n$ be a clause.

Then $\mathit{thin}(c, A_i) = A \leftarrow A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n$.

fatten Let $c : A \leftarrow A_1, \ldots, A_n$ be a clause and B an atom.

Then $\mathit{fatten}(c, B) = A \leftarrow A_1, \ldots, A_i, B, A_{i+1}, \ldots, A_n$.

restrict Let P be a program. Let $p \in \mathit{preds}(P)$. Let $Q = \bigcup \mathit{def}_P(q)$ for all q such that $\mathit{def}_P(q) \cap P(p) = \emptyset$.

Then $\mathit{restrict}(P, p) = (P - Q)$.

Here $\mathit{preds}(P)$ denotes all predicate symbols in P, $\mathit{def}_P(q)$ all clauses of P defining q, i.e. the clauses with q as the predicate symbol of the head. $P(p)$ is the call-graph for the predicate symbol p. It is the set of clauses obtained from P starting with $\mathit{def}_P(p)$ and adding recursively all clauses defining predicate symbols occurring in bodies of the computed closure.

The operators preserve refinement equivalence if the following *applicability conditions* are satisfied:

unfold A single unfolding step is always correct because all MGUs of the atom and the heads of other clauses are considered. If a clause is added in the next step, this refinement is not necessarily equivalent for the inverse order of steps.

fold Abbreviating a part of the body of a clause is correct, if the abbreviated clauses can be unfolded to obtain the original clause again.

prune Pruning a clause is applicable if either the clause is redundant or if it cannot be used to derive an answer, i.e. the body of the clause can never be proven.

add A clause can be added, if it is implied already by the program or if it cannot be used to derive an answer.

restrict The call-graphs of the original and the restricted program are equal.

A.10 Specification-building operators

In Subsection 4.4.4 we commented on specification-building operators in algebraic specification. In this section, some more details are provided. The following details are mostly taken from [SST92]. We assume that the reader is familar with standard notions of algebraic specification, particularly:

- algebraic many-sorted signatures, usually denoted by Σ, Σ',
- algebraic signature morphism $\sigma : \Sigma \to \Sigma'$,
- Σ-algebra, class $\mathit{Alg}(\Sigma)$ of Σ-algebras,
- Σ-homomorphism and Σ-isomorphism,
- Σ-equation, first-order Σ-sentence,
- satisfaction relation between Σ-algebras and Σ-sentences,

- the reduct functor $\_|_\sigma : Alg(\Sigma) \to Alg(\Sigma')$ for a signature morphism $\sigma : \Sigma \to \Sigma'$.

A Σ-specification SP is expected to determine a class $[\![SP]\!] \in \mathcal{P}(Alg(\Sigma))$ of Σ-algebras, the models of SP. SP is consistent if $SP \neq \emptyset$. There are the following specification-building operators:

- If Σ is a signature, then Σ is a Σ-specification with the semantics:
$$[\![\Sigma]\!] = Alg(\Sigma)$$
- If SP is Σ-specification and Φ is a set of Σ-sentences, then **impose** Φ **On** SP is a Σ-specification with the semantics:
$$[\![\textbf{impose } \Phi \textbf{ On } SP]\!] = \{A \in [\![SP]\!] | A \models \Phi\}$$
- If SP is a Σ-specification and $\sigma : \Sigma' \to \Sigma$ is a signature morphism, then **derive from** SP **by** σ is a Σ'-specification with the semantics:
$$[\![\textbf{derive from } SP \textbf{ by } \sigma]\!] = \{A|_\sigma | A \in [\![SP]\!]\}$$
- If SP is a Σ-specification and $\sigma : \Sigma \to \Sigma'$ is a signature morphism, then **translate** SP **by** σ is a Σ'-specification with the semantics:
$$[\![\textbf{translate } SP \textbf{ by } \sigma]\!] = \{A' \in Alg(\Sigma') | A'|_\sigma \in [\![SP]\!]\}$$
- If SP and SP' are Σ-specifications, then $SP \cup SP'$ is a Σ-specification with the semantics:
$$[\![SP \cup SP']\!] = [\![SP]\!] \cap [\![SP']\!]$$
- If SP is a Σ-specification and $\sigma : \Sigma \to \Sigma'$ is a signature morphism, then **minimal** SP **w.r.t.** σ is a Σ-specification with the semantics:
$$[\![\textbf{minimal } SP \textbf{ w.r.t. } \sigma]\!] = \{A \in [\![SP]\!] | A \text{ is minimal in } Alg(\Sigma) \text{ w.r.t. } \sigma\},$$
where a Σ-algebra is minimal w.r.t. σ if it has no non-trivial subalgebra with an isomorphic σ-reduct.
- If SP is a Σ-specification, then **iso − close** SP is a Σ-specification with the semantics:
$$[\![\textbf{iso} - \textbf{close } SP]\!] = \{A \in Alg(\Sigma) | A \text{ is isomorphic to } B \text{ for some } B \in [\![SP]\!]\}$$
- If SP is a Σ-specification and $\sigma : \Sigma \to \Sigma'$ is a signature morphism and Φ' is a set of Σ'-sentences, then **abstract** SP **w.r.t.** Φ' **via** σ is a Σ-specification with the semantics:
$$[\![\textbf{abstract } SP \textbf{ w.r.t. } \Phi' \textbf{ via } \sigma]\!] = \{A \in Alg(\Sigma) | A \equiv^\sigma_{\Phi'} B \text{ for some } B \in [\![SP]\!]\},$$
where $A \equiv^\sigma_{\Phi'} B$ means that A is observationally equivalent to B w.r.t. Φ' via σ.

- If SP is a Σ-specification and If SP' is a Σ'-specification, then $SP + SP'$ is a $(\Sigma \cup \Sigma')$-specification with the semantics:

  $$[\![SP + SP']\!] = \{A \in Alg(\Sigma \cup \Sigma') | A|_\Sigma \in [\![SP]\!] \text{ and } A|_{\Sigma'} \in [\![SP']\!]\}$$

  This is expressible using $\_ \cup \_$ and **translate** as defined above.

- If SP is a Σ-specification, S is a set of sort names, Ω is a set of ranked operation names such that adding S and Ω to Σ yields a well-formed signature Σ' and Φ' is a set of Σ'-sentences, then **enrich** SP **by sorts** S **opns** Ω **axioms** Φ' is a Σ'-specification with the semantics:

  $$\begin{aligned}&[\![\textbf{enrich } SP \textbf{ by sorts } S \textbf{ opns } \Omega \textbf{ axioms } \Phi']\!] = \\ &\quad \{A \in Alg(\Sigma') | A|_\Sigma \in [\![SP]\!] \text{ and } A \models \Phi'\}\end{aligned}$$

  This is expressible using **translate** and **impose** as defined above.

- If SP is a Σ-specification and S is a set of sort names, then **reachable** SP **on** S is a Σ-specification with the semantics:

  $$[\![\textbf{reachable } SP \textbf{ on } S]\!] = \{A \in [\![SP]\!] | A \text{ is generated on } S\},$$

  where A is said to be generated on S if it has no proper subalgebra having the same carriers of sorts not in S. This is expressible using **minimal** as defined above.

Appendix B

Technical details of the framework

B.1 Deconstruction of sequences of rules

$$\begin{aligned}\textbf{Head Of }\_ &: \textsf{Rules} \to \textsf{Rule}\\ \textbf{Tail Of }\_ &: \textsf{Rules} \to \textsf{Rules}\\ \textbf{Nil? }\_ &: \textsf{Rules} \to \textsf{Boolean}\end{aligned}$$

$$\textbf{Head Of Rules From } \langle r_1, \ldots, r_n\rangle \to r_1 \qquad [\textbf{Head Of}]$$

$$\textbf{Tail Of Rules From } \langle r_1, r_2, \ldots, r_n\rangle \to \langle r_2, \ldots, r_n\rangle \qquad [\textbf{Tail Of}]$$

$$\textbf{Nil? Rules From } \langle\rangle \to \textbf{True} \qquad [\textbf{Nil?}.1]$$

$$\frac{n \geq 1}{\textbf{Nil? Rules From } \langle r_1, \ldots, r_n\rangle \to \textbf{False}} \qquad [\textbf{Nil?}.2]$$

B.2 Selection of variables

The relations for the selection of variables in elements are defined below. In meta-programs, we refer to $\mathcal{VARS}_{\textsf{Parameter}^\star}$ by **Variables In** _.

$$\mathcal{VARS}_{\textsf{Parameter}}(\textbf{Variable From } v \textbf{ Of Sort } \sigma) \Rightarrow \{v\} \qquad [\mathcal{VARS}.1]$$

$$\frac{\begin{array}{ll} & \mathcal{VARS}_{\textsf{Parameter}}(\overline{p}_1) \Rightarrow V_1\\ \wedge & \ldots\\ \wedge & \mathcal{VARS}_{\textsf{Parameter}}(\overline{p}_m) \Rightarrow V_m\end{array}}{\mathcal{VARS}_{\textsf{Parameter}^\star}(\langle \overline{p}_1, \ldots, \overline{p}_m\rangle) \Rightarrow V_1 \cup \cdots \cup V_m} \qquad [\mathcal{VARS}.2]$$

$$\frac{\mathcal{VARS}_{\textsf{Parameter}^\star}(\langle \overline{p}^{\downarrow}_1, \ldots, \overline{p}^{\downarrow}_m\rangle) \Rightarrow V^{\downarrow}}{\mathcal{VARS}^{\downarrow}_{\textsf{Element}}(\langle n, \langle \overline{p}^{\downarrow}_1, \ldots, \overline{p}^{\downarrow}_m\rangle, \langle \overline{p}^{\uparrow}_1, \ldots, \overline{p}^{\uparrow}_k\rangle\rangle) \Rightarrow V^{\downarrow}} \qquad [\mathcal{VARS}.3]$$

$$\frac{\mathcal{VARS}_{\mathsf{Parameter}^\star}(\langle \overline{p}_1^{\uparrow}, \ldots, \overline{p}_k^{\uparrow}\rangle) \Rightarrow V^{\uparrow}}{\mathcal{VARS}^{\uparrow}_{\mathsf{Element}}(\langle n, \langle \overline{p}_1^{\downarrow}, \ldots, \overline{p}_m^{\downarrow}\rangle, \langle \overline{p}_1^{\uparrow}, \ldots, \overline{p}_k^{\uparrow}\rangle\rangle) \Rightarrow V^{\uparrow}} \quad [\mathcal{VARS}.4]$$

$$\frac{\begin{array}{ll} & \mathcal{VARS}^{\downarrow}_{\mathsf{Element}}(\overline{e}) \Rightarrow V^{\downarrow} \\ \wedge & \mathcal{VARS}^{\uparrow}_{\mathsf{Element}}(\overline{e}) \Rightarrow V^{\uparrow} \end{array}}{\mathcal{VARS}_{\mathsf{Element}}(\overline{e}) \Rightarrow V^{\downarrow} \cup V^{\uparrow}} \quad [\mathcal{VARS}.5]$$

B.3 Applied and defining occurrences

$$\frac{\begin{array}{ll} & \mathcal{VARS}^{\downarrow}_{\mathsf{Element}}(\overline{e}_0) \Rightarrow V_0 \\ \wedge & \mathcal{VARS}^{\uparrow}_{\mathsf{Element}}(\overline{e}_1) \Rightarrow V_1 \\ \wedge & \ldots \\ \wedge & \mathcal{VARS}^{\uparrow}_{\mathsf{Element}}(\overline{e}_n) \Rightarrow V_n \end{array}}{\textbf{Do In } \langle t, \overline{e}_0, \langle \overline{e}_1, \ldots, \overline{e}_n\rangle\rangle \Rightarrow V_0 \cup V_1 \cdots \cup V_n} \quad [\mathcal{DO}]$$

$$\frac{\begin{array}{ll} & \mathcal{VARS}^{\uparrow}_{\mathsf{Element}}(\overline{e}_0) \Rightarrow V_0 \\ \wedge & \mathcal{VARS}^{\downarrow}_{\mathsf{Element}}(\overline{e}_1) \Rightarrow V_1 \\ \wedge & \ldots \\ \wedge & \mathcal{VARS}^{\downarrow}_{\mathsf{Element}}(\overline{e}_n) \Rightarrow V_n \end{array}}{\textbf{Ao In } \langle t, \overline{e}_0, \langle \overline{e}_1, \ldots, \overline{e}_n\rangle\rangle \Rightarrow V_0 \cup V_1 \cdots \cup V_n} \quad [\mathcal{AO}]$$

B.4 Left-to-right dependencies ($\mathcal{WD}$)

To work out a common restriction of well-definedness ($\mathcal{WD}$), left-to-right dependencies are formalized here. Thereby, we obtain a very simple data flow criterion which can be understood, for example, as the property of L-attribution [Alb91] for attribute grammars or call-correctness [Boy96a] for logic programs with directional types. To force this property, the rule $\mathcal{DF}.1$ (refer to Subsection 2.3.3) has to be rejected in favour of the rule $\mathcal{DF}.2$ given below.

$$\frac{\mathcal{L}2\mathcal{R}(\overline{r}_j) \text{ for } j = 1, \ldots, n}{\mathcal{DF}(\langle \overline{r}_1, \ldots, \overline{r}_n\rangle, \overline{i})} \quad [\mathcal{DF}.2]$$

$$\frac{\begin{array}{ll} & \mathcal{VARS}^{\downarrow}_{\mathsf{Element}}(\overline{e}_1) \subseteq \mathcal{VARS}^{\downarrow}_{\mathsf{Element}}(\overline{e}_0) \\ \wedge & \mathcal{VARS}^{\downarrow}_{\mathsf{Element}}(\overline{e}_2) \subseteq \mathcal{VARS}^{\downarrow}_{\mathsf{Element}}(\overline{e}_0) \cup \mathcal{VARS}^{\uparrow}_{\mathsf{Element}}(\overline{e}_1) \\ \wedge & \ldots \\ \wedge & \mathcal{VARS}^{\downarrow}_{\mathsf{Element}}(\overline{e}_n) \subseteq \mathcal{VARS}^{\downarrow}_{\mathsf{Element}}(\overline{e}_0) \cup \bigcup_{i=1}^{n-1} \mathcal{VARS}^{\uparrow}_{\mathsf{Element}}(\overline{e}_i) \\ \wedge & \mathcal{VARS}^{\uparrow}_{\mathsf{Element}}(\overline{e}_0) \subseteq \mathcal{VARS}^{\downarrow}_{\mathsf{Element}}(\overline{e}_0) \cup \bigcup_{i=1}^{n} \mathcal{VARS}^{\uparrow}_{\mathsf{Element}}(\overline{e}_i) \end{array}}{\mathcal{L}2\mathcal{R}(\langle t, \overline{e}_0, \langle \overline{e}_1, \ldots, \overline{e}_n\rangle\rangle)} \quad [\mathcal{L}2\mathcal{R}]$$

B.5 Basic unification

The following definition of $\mathcal{SOLVE}$ only copes with variables as parameters. If instances of the general framework with compound parameters (terms) are considered, the complete Robinson's unification algorithm need to be instrumented; refer e.g. to [NM95, p. 39] for a suitable presentation.

$$\frac{\mathcal{E} \text{ is in solved form}}{\mathcal{SOLVE}(\mathcal{E}) \to \mathcal{E}} \qquad [\mathcal{SOLVE}.1]$$

$$\frac{\begin{array}{ll} & \exists p \in \mathsf{Parameter}: \\ & \langle p, p\rangle \in \mathcal{E} \\ \wedge & \mathcal{SOLVE}(\mathcal{E} \setminus \{\langle p,p\rangle\}) \to \theta \end{array}}{\mathcal{SOLVE}(\mathcal{E}) \to \theta} \qquad [\mathcal{SOLVE}.2]$$

$$\frac{\begin{array}{ll} & \exists p, p' \in \mathsf{Parameter}: p \neq p' \\ \wedge & \langle p, p'\rangle \in \mathcal{E} \\ \wedge & \textbf{Variable Of } p \to v \\ \wedge & p \text{ occurs in } \mathcal{E} \setminus \{\langle p,p'\rangle\} \\ \wedge & \mathcal{SOLVE}(\ ((\mathcal{E} \setminus \{\langle p,p'\rangle\})\ [v/p']) \cup \{\langle p,p'\rangle\}\ ) \to \theta \end{array}}{\mathcal{SOLVE}(\mathcal{E}) \to \theta} \qquad [\mathcal{SOLVE}.3]$$

B.6 Terms

$$\frac{\textbf{Is}_{\mathsf{Term}}(\pi_1(p)) \to b}{\textbf{Term? } p \to b} \qquad [\textbf{Term?}]$$

$$\textbf{Constructor Of Term From } f\ p^\star \textbf{ Of Sort } \sigma \to f \qquad [\textbf{Constructor Of}]$$

$$\textbf{Subterms Of Term From } f\ p^\star \textbf{ Of Sort } \sigma \to p^\star \qquad [\textbf{Subterms Of}]$$

$$\frac{\textbf{Is}_{\mathsf{Name}}(s) \to b}{\textbf{Name? } s \to b} \qquad [\textbf{Name?}]$$

$$\frac{\textbf{Is}_{\mathsf{Constructor}}(s) \to b}{\textbf{Constructor? } s \to b} \qquad [\textbf{Constructor?}]$$

$$\frac{\begin{array}{ll} & \textbf{Term? } p \to \textbf{True} \\ \wedge & \textbf{Constructor Of } p \to f \\ \wedge & \textbf{Subterms Of } p \to \langle p_1, \ldots, p_n\rangle \\ \wedge & \textbf{Sort Of } p \to \sigma \\ \wedge & \textbf{Substitute } \theta \textbf{ In Parameter } p_i \to p'_i \text{ for } i = 1, \ldots, n \\ \wedge & \textbf{Term From } f\ \langle p'_1, \ldots, p'_n\rangle \textbf{ Of Sort } \sigma \to p' \end{array}}{\textbf{Substitute } \theta \textbf{ In Parameter } p \to p'} \qquad [\mathcal{SUBST}.3]$$

B.7 Computational elements

$$\frac{\mathbf{Is}_{\overline{\mathsf{Element}}}(pre) \to b}{\mathbf{Element?}\ pre \to b} \qquad [\mathbf{Element?}]$$

$$\frac{\mathbf{Is}_{\overline{\mathsf{Computation}}}(pre) \to b}{\mathbf{Computation?}\ pre \to b} \qquad [\mathbf{Computation?}]$$

$$\mathbf{Symbol\ Of\ Premise\ From}\ s\ p_{\downarrow}^{\star} \to p_{\uparrow}^{\star} \to s \qquad [\mathbf{Symbol\ Of}]$$

$$\mathbf{Parameters\ Input\ Of\ Premise\ From}\ s\ p_{\downarrow}^{\star} \to p_{\uparrow}^{\star} \to p_{\downarrow}^{\star} \qquad [\mathbf{Parameters}.3]$$

$$\mathbf{Parameters\ Output\ Of\ Premise\ From}\ s\ p_{\downarrow}^{\star} \to p_{\uparrow}^{\star} \to p_{\uparrow}^{\star} \qquad [\mathbf{Parameters}.4]$$

$$\frac{\mathbf{Is}_{\overline{\mathsf{Name}}}(s) \to b}{\mathbf{Name?}\ s \to s} \qquad [\mathbf{Name?}]$$

$$\frac{\mathbf{Is}_{\overline{\mathsf{Operation}}}(s) \to b}{\mathbf{Operation?}\ s \to s} \qquad [\mathbf{Operation?}]$$

Appendix C

Remainder of the operator suite

C.1 More auxiliary operators

We need to define some other auxiliary operators used elsewhere.

C.1.1 Transformations on fragments

In this subsection, trivial transformations on fragment types are presented. Many of them can be regarded as lifting operators to apply transformations to more complex fragment types. The transformations are useful for the definition of several more elaborate operators, e.g. for applications of the operator **Replace**.

% identity for substituting conclusions
lhsIdentity :
 λ e : **Conclusion** . ⟨e, ⟨ ⟩, ⟨ ⟩, ⟨ ⟩⟩.

% identity for substituting premises
rhsIdentity :
 λ e : **Premise** . ⟨⟨e⟩, ⟨ ⟩⟩.

% substitute conclusions with a certain symbol, only
lhsForSymbol :
 λ s : **Symbol** . λtLhs : **LhsSubstitution** . λ e : **Conclusion** .
  s = **Symbol Of** e → tLhs **On** e, lhsIdentity **On** e.

% substitute premises with a certain symbol, only
rhsForSymbol :
 λ s : **Symbol** . λtRhs : **RhsSubstitution** . λ e : **Premise** .
  s = **Symbol Of** e → tRhs **On** e, rhsIdentity **On** e.

% coerce transformation on elements to LhsSubstitution
tE2tLhs :
 λ tE : **Element** → **Element** . λ e : **Conclusion** .
  ⟨tE **On** e, ⟨ ⟩, ⟨ ⟩, ⟨ ⟩⟩.

% coerce transformation on premises to RhsSubstitution
tP2tRhs :
 λ tP : **Premise** → **Premise** . λ e : **Premise** .
  ⟨⟨tP **On** e⟩, ⟨ ⟩⟩.

```
% transform elements with a certain symbol, only
tEforSymbol :
 λ s : Symbol . λ tE : Element → Element . λ e : Element .
  Symbol Of e = s → tE On e, e.

% transform premises with a certain symbol, only
tPforSymbol :
 λ s : Symbol . λ tP : Premise → Premise . λ e : Premise .
  Symbol Of e = s → tP On e, e.

% coerce Element → (Element x Substitution) to LhsSubstitution
tEandSubst2tLhs :
 λ tEandSubst : Element → (Element × Substitution) . λ e0 : Conclusion .
  Let ⟨e1, subst⟩ = tEandSubst On e0 In ⟨e1, ⟨ ⟩, ⟨ ⟩, subst⟩.

% coerce Premise → (Premise x Substitution) to RhsSubstitution
tPandSubst2tRhs :
 λ tPandSubst : Premise → (Premise × Substitution) . λ e0 : Premise .
  Let ⟨e1, subst⟩ = tPandSubst On e0 In ⟨⟨e1⟩, subst⟩.

% coerce Parameter* → (Parameter* x Substitution) to LhsSubstitution
tPsAndSubst2tLhs :
 λ tPsAndSubst : Parameter* → (Parameter* × Substitution) .
 λ io : Io .
 λ e0 : Element .
  Let ⟨ps, subst⟩ = tPsAndSubst On Parameters io Of e0 In
   io = Input →
    ⟨Conclusion From Symbol Of e0 ps → Parameters Output Of e0, ⟨ ⟩, ⟨ ⟩, subst⟩,
    ⟨Conclusion From Symbol Of e0 Parameters Input Of e0 → ps, ⟨ ⟩, ⟨ ⟩, subst⟩.

% coerce Parameter* → (Parameter* x Substitution) to RhsSubstitution
tPsAndSubst2tRhs :
 λ tPsAndSubst : Parameter* → (Parameter* × Substitution) .
 λ io : Io .
 λ e0 : Premise .
  Let ⟨ps, subst⟩ = tPsAndSubst On Parameters io Of e0 In
   io = Input →
    ⟨⟨Premise From Symbol Of e0 ps → Parameters Output Of e0⟩, subst⟩,
    ⟨⟨Premise From Symbol Of e0 Parameters Input Of e0 → ps⟩, subst⟩.

% coerce transformation on Parameter* to Element
tPs2tE :
 λ io : Io . λ tPs : Parameter* → Parameter* . λ e : Element .
  Let f = λ select : Io .
   Let ps = Parameters select Of e In
    select = io → tPs On ps, ps
  In
   Element From (Symbol Of e) (f On Input) → (f On Output).
```

```
% coerce transformation on Parameter* to Premise
tPs2tP :
 λ io : **Io** . λ tPs : **Parameter*** → **Parameter*** . λ e : **Premise** .
  **Let** f = λ select : **Io** .
   **Let** ps = **Parameters** select **Of** e **In**
    select = io → tPs **On** ps, ps
  **In**
   **Premise From** (**Symbol Of** e) (f **On Input**) → (f **On Output**).
```

C.1.2 Inserting premises into rules

Given a rule and a premise, there are various possibilities how to define the target position of the premise. One way is to specify the exact position by an index. Another one is to leave it unspecified, what is suitable if we assume that the actual position of computational elements is meaningless. We prefer to adhere to the more restricted well-definedness property, where an applied occurrence of a variable must not occur *before* a defining occurrence. There are two extremes, that is to say to insert the premise either as early as possible (*insertEarly*) or as late as possible (*insertLate*). For pragmatic reasons, the first extreme is suitable for premises that do not have an output position at all.

Insert _ Into _ : Premise × Rule → (Rule → Rule)

```
 λ e : **Premise** .
  **Nil? Parameters Output Of** e →
   insertEarly **On** e,
   insertLate **On** e.

insertEarly :
 λ e : **Premise** . λ r : **Rule** .
  **Let** required = **Variables In Parameters Input Of** e **In**
   **Letrec** early : **Premise*** → $\mathcal{P}$(**Variable**) → **Premise*** =
    λ es : **Premise*** . λ vs : $\mathcal{P}$(**Variable**) .
     required ⊆ vs →
      ⟨e⟩ ++ es,
      **Let** skip = **Head Of** es **In**
       ⟨skip⟩ ++ (
        early
         **On Tail Of** es
         **On** (vs ∪ **Variables In Parameters Output Of** skip)
       )
   **In**
    **Rule From Tag Of** r **Conclusion Of** r ⇐ (
      early
       **On Premises Of** r
       **On Variables In Parameters Input Of Conclusion Of** r
     ).
```

```
insertLate :
 λ e : **Premise** . λ r : **Rule** .
  Let defined = Variables In Parameters Output Of e In
   Letrec late : Premise* → Premise* = λ es : Premise* .
    Nil? es →
     ⟨e⟩,
     Let head = Head Of es In
      Variables In Parameters Input Of head ∩ defined = ∅ →
      ⟨head⟩ ++ (late On Tail Of es),
      ⟨e⟩ ++ es
   In
    Rule From Tag Of r Conclusion Of r ⇐ (late On Premises Of r).
```

C.1.3 Skipping computations in a sequence of premises

In our instance of the calculus skeleton elements and computations are distinguished. In several cases, iterations on the skeleton elements (excluding computations) must be performed, e.g. during folding and unfolding. The following function skips the heading computations in a sequence of premises. Thus, a caller of the function receives a pair $\langle a, b\rangle$, where a are the skipped computations, whereas b is the remaining sequence of premises starting with a skeleton element if there is any left.

```
skipComputations :
 λ es : Premise* .
  Letrec skipComputationsSlave : (Premise* × Premise*) → (Premise* × Premise*) =
   λ sofar : Premise* × Premise* .
    Let ⟨skipped, todo⟩ = sofar In
     todo = ⟨ ⟩ →
      sofar,
      Let spot = Head Of todo In
       Computation? spot →
        skipComputationsSlave On ⟨skipped ++ ⟨spot⟩, Tail Of todo⟩,
        sofar
  In
   skipComputationsSlave On ⟨⟨ ⟩, es⟩.
```

C.1.4 Selection of parameters

The selection of parameters on defining and applied occurrences follows a common schema which is presented below:

```
selectOccurrences :
 λ ctrl : Io . λ r : Rule . λ ⟨io, sy, so⟩ : Position .
 % select parameters of corresponding sort
  Let selectPs = λ ps : Parameter* .
   Parameters Of Sort so In ps
  In
   io = ctrl →
```

```
% defining (applied) occurrences on input (output) positions are found on LHS
   Symbol Of Conclusion Of r = sy → selectPs On Parameters io Of Conclusion Of r, ⟨ ⟩,
% defining (applied) occurrences on output (input) positions are found on RHS
   Fold Left λ sofar : Parameter* . λ e : Premise .
    sofar ++ (Symbol Of e = sy → selectPs On Parameters io Of e, ⟨ ⟩)
   Neutral ⟨ ⟩ List Premises Of r.
```

The selection of a defining occurrence can now be performed by applying the above abstraction to **Input**, i.e.:

```
selectDos :
 selectOccurrences On Input.
```

Dually, the selection of an applied occurrence can now be performed by applying the above abstraction to **Output**, i.e.:

```
selectAos :
 selectOccurrences On Output.
```

In some cases we need *unique* occurrences. Such a restriction can be easily obtained from the above schema *selectOccurrences*.

```
selectUniqueOccurrence :
 λ io : Io . λ r : Rule . λ pos : Position .
  Let {p} = selectOccurrences On io On r On pos In p.
```

The selection of unique defining (applied) occurrences is denoted by *selectDo* (*selectAo*).

C.2 Parameterization schemata

C.2.1 Addition, removal, contraction

We present the details of the following parameterization schemata:

$$\begin{aligned} \textbf{Add}\ \_ &: \mathsf{Position}^\star \to \mathsf{Trafo} \\ \textbf{Ensure}\ \_ &: \mathsf{Position}^\star \to \mathsf{Trafo} \\ \textbf{Sub}\ \_ &: \mathsf{Position}^\star \to \mathsf{Trafo} \\ \textbf{Contract}\ \_ &: \mathsf{Position}^\star \to \mathsf{Trafo} \end{aligned}$$

These operators can be specified following the schema of element substitution, i.e. using the operator **Replace**. Actually, the essential behaviour of these operators can be stated as a transformation of the following profile:

$$\mathsf{Parameter}^\star \to (\mathsf{Parameter}^\star \otimes \mathsf{Substitution})$$

To lift such transformations to LHS / RHS substitutions, and to iterate on the sequences of associations, the following more elaborate variant of **Replace** is assumed:

```
replacePositions :
 λ tPs : Sort → Parameter* → (Parameter* × Substitution) .
 λ poss : Position* .
 λ rs : Rules .
  Fold Left
   λ sofar : Rules . λ ⟨io, sy, so⟩ : Position .
    Replace
     (lhsForSymbol On sy On (tPsAndSubst2tLhs On (tPs On so) On io))
     (rhsForSymbol On sy On (tPsAndSubst2tRhs On (tPs On so) On io))
    On sofar
  Neutral rs List poss.
```

Now the operators **Add**, **Sub** and **Contract** can be directly implemented by applying *replacePositions* to the following expressions:

for **Add**:

```
λ sort : Sort . λ ps : Parameter* .
⟨ps ++ ⟨New Variable Of Sort sort⟩, ⟨ ⟩⟩.
```

for **Sub**:

```
λ sort : Sort . λ ps : Parameter* .
⟨Fold Left λ sofar : Parameter* . λ p : Parameter .
  sofar ++ ( (Sort Of p) = sort → ⟨ ⟩, ⟨p⟩ )
 Neutral ⟨ ⟩ List ps,
 ⟨ ⟩
⟩.
```

for **Contract**:

```
λ sort : Sort . λ ps : Parameter* .
⟨ps \ Tail Of Parameters Of Sort sort In ps,
 Let keep = Head Of Parameters Of Sort sort In ps In
  Fold Left
   λ subst : Substitution . λ p : Parameter .
    subst ⋈ Unify Parameters keep And p
  Neutral ⟨ ⟩
  List Tail Of Parameters Of Sort sort In ps
⟩.
```

C.2.2 Conditional addition

Since the operator **Ensure** is intended to add only those positions which do not exist yet, it can be expressed via **Add** if we first reject the associations corresponding to existing positions by a simple traversal of the type of the input rules.

```
 λ poss : **Position*** . λ rs : **Rules** .
  Let possToBeAdded =
   Let sigma = Sigma Of rs In
    Fold Left λ sofar : Position* . λ ⟨io, sy, so⟩ : Position .
     Let prof = Profile Of sy In sigma In
      prof = ? →
       sofar,
       so ∈ Sorts io Of prof →
        sofar,
        sofar ++ ⟨⟨io, sy, so⟩⟩
    Neutral ⟨ ⟩ List poss
  In
   Add possToBeAdded On rs.
```

C.2.3 Permutation

Permute _ : Profile → Trafo

```
% permute parameters according to sorts
Let tPs = λ ps : Parameter* . λ ss : Sort* .
 # ps = # ss ∘→
  Map λ sort : Sort . Let ⟨p⟩ = Parameters Of Sort sort In ps In p List ss
In

 λ prof : Profile .
% permutation of a conclusion
  Let tE = tEforSymbol On Symbol Of prof On
   (λ e : Element .
     Element From Symbol Of e
        (tPs On Parameters Input Of e On Sorts Input Of prof)
     → (tPs On Parameters Output Of e On Sorts Output Of prof)
   )
  In
% permutation of a premise
   Let tP = tPforSymbol On Symbol Of prof On
    (λ e : Premise .
      Premise From Symbol Of e
         (tPs On Parameters Input Of e On Sorts Input Of prof)
      → (tPs On Parameters Output Of e On Sorts Output Of prof)
    )
   In
    Replace (tE2tLhs On tE) (tP2tRhs On tP).
```

C.3 Computation schemata

C.3.1 Copies

Copy _ To _ : Position × Position → Trafo

```
$\lambda$ from : **Position** . $\lambda$ to : **Position** .
 **Let** $\langle$fio, fsy, fso$\rangle$ = from **In**
  **Let** $\langle$tio, tsy, tso$\rangle$ = to **In**
   fso = tso $\circ\!\!\rightarrow$
    ($\lambda$ rs : **Rules** .
      **Map** $\lambda$ r : **Rule** .
       **Let** pF = selectUniqueDo **On** r **On** from **In**
        **Let** pT = selectUniqueAo **On** r **On** to **In**
         $\neg$ **Variables In** {pT} $\cap$ (**Ao In** r $\setminus$ **Do In** r) = $\emptyset$ $\circ\!\!\rightarrow$
          **Substitute Unify Parameters** pF **And** pT **In Rule** r
      **List** rs
    )
  $\circ$ **Ensure** $\langle$from, to$\rangle$.
```

C.3.2 Constant computations

Define _ By _ : Position × Symbol → Trafo

```
$\lambda$ pos : **Position** . $\lambda$ by : **Symbol** .
 ($\lambda$ rs : **Rules** .
   **Map** $\lambda$ r0 : **Rule** .
    **Fold Left** $\lambda$ r : **Rule** . $\lambda$ p : **Parameter** .
     **Insert Premise From** by $\langle\ \rangle \rightarrow \langle$p$\rangle$ **Into** r
    **Neutral** r0
    **List** (**Variables In** (selectAos **On** r0 **On** pos) $\setminus$ **Do In** r0)
   **List** rs
 ).
```

Default For _ By _ : Sort × Symbol → Trafo

```
$\lambda$ so : **Sort** . $\lambda$ by : **Symbol** . $\lambda$ rs : **Rules** .
 **Map** $\lambda$ r0 : **Rule** .
  **Fold Left** $\lambda$ r : **Rule** . $\lambda$ v : **Variable** .
   **Insert Premise From** by $\langle\ \rangle \rightarrow \langle$v$\rangle$ **Into** r
  **Neutral** r0
  **List Variables Of Sort** so **In** (**Ao In** r0 $\setminus$ **Do In** r0)
 **List** rs.
```

C.3.3 Unary conditions

Use _ By _ : Position × Symbol → Trafo

```
$\lambda$ pos : **Position** . $\lambda$ by : **Symbol** .
  ($\lambda$ rs : **Rules** .
    **Map** $\lambda$ r0 : **Rule** .
     **Fold Left** $\lambda$ r : **Rule** . $\lambda$ p : **Parameter** .
      **Insert Premise From** by $\langle$p$\rangle \rightarrow \langle\ \rangle$ **Into** r
     **Neutral** r0
     **List Variables In** (selectDos **On** r0 **On** pos)
    **List** rs
```

C.3.4 Nontrivial computations

Compute $\_\ \_ \rightarrow \_$: $\mathsf{Symbol} \times \mathsf{Position}^\star \rightarrow \mathsf{Position}^\star \rightarrow \mathsf{Trafo}$

```
   λ sy : Symbol . λ possI : Position* . λ possO : Position* .
     (λ rs : Rules .
       Map λ r : Rule .
   % compute inputs of premise
         Let psI =
          Map λ pos : Position .
           Let p = selectUniqueDo On r On pos In p
          List possI
         In
   % compute outputs of premise
          Let psO =
           Map λ pos : Position .
            Let p = selectUniqueAo On r On pos In p
           List possO
          In
   % ensure that the outputs are not yet defined
           Variables In psO ∩ Do In r = ∅ ∘→
   % insert premise
            Insert Premise From sy psI → psO Into r
       List rs
     )
   ∘ Ensure possO
   ∘ Ensure possI.
```

C.3.5 Compositional computations

Relate $\_\ \_\ \_$: $\mathsf{Io} \times \mathsf{Association}^\star \times \mathsf{Prefix} \rightarrow \mathsf{Trafo}$

```
λ io : Io . λ as : Association* . λ pfx : Prefix .
% select parameters according to associations
 Let parasToRelate = λ e : Element .
  Fold Right
   λ p : Parameter . λ ps : Parameter* .
    ( Fold Right
       λ ⟨sym, sort⟩ : ⟨Symbol, Sort⟩ . λ ps : Parameter* .
        (((Symbol Of e) = sym) ∩ ((Sort Of p) = sort) → ⟨p⟩, ⟨ ⟩) ++ ps
      Neutral ⟨ ⟩ List as
    ) ++ ps
  Neutral ⟨ ⟩ List Parameters io Of e
 In
  (λ rs : Rules . Map λ r : Rule .
% accumulate relevant positions on LHS and RHS
    Let lhs = parasToRelate On Conclusion Of r In
     Let rhs =
      Fold Right λ e : Premise . λ ps : Parameter* .
       (Element? e → (parasToRelate On e), ⟨ ⟩) ++ ps
      Neutral ⟨ ⟩ List Premises Of r
     In
      Let s = Operation From pfx Tag Of r In
       io = Output →
% outputs from RHS are used to DEFINE outputs on LHS
        Let defocc = Variables In lhs In
         (¬ defocc = ∅) And (defocc ⊆ (Ao In r \ Do In r)) →
          Let e = Premise From s rhs → lhs In
           Rule From Tag Of r Conclusion Of r ⇐ (Premises Of r ++ ⟨e⟩),
          r,
% inputs from LHS are used to DEFINE inputs on RHS
        Let useocc = Variables In lhs In
         Let defocc = Variables In rhs In
          (¬ useocc = ∅) And (defocc ⊆ (Ao In r \ Do In r)) →
           Let e = Premise From s lhs → rhs In
            Rule From Tag Of r Conclusion Of r ⇐ (⟨e⟩ ++ Premises Of r),
           r
    List rs) ∘ Ensure Map λ ⟨sy, so⟩ : Association . ⟨io, sy, so⟩ List as.
```

C.3.6 Combining unused parameters

Reduce _ By _ : Sort × Symbol → Trafo

```
  λ so : Sort . λ by : Symbol . λ rs : Rules .
   Map
    λ r0 : Rule .
     Let vs = Variables Of Sort so In (Do In r0 \ Ao In r0) In
      vs = ∅ →
       r0,
       Let ⟨r1, unused⟩ =
        Fold Left λ ⟨r, v1⟩ : Rule × Variable . λ v2 : Variable .
         Let new = New Variable Of Sort so In
          ⟨Insert Premise From by ⟨v1, v2⟩ → ⟨new⟩ Into r, new⟩
        Neutral ⟨r0, Head Of vs⟩
        List Tail Of vs
       In r1
   List rs.
```

C.4 Reachability

Derivable From _ In _ : $\mathcal{P}$(Symbol) × Skeleton → $\mathcal{P}$(Symbol)

```
$\lambda$ from : $\mathcal{P}$(**Symbol**) . $\lambda$ sk : **Skeleton** .
 **Letrec** f : $\mathcal{P}$(**Symbol**) $\rightarrow$ $\mathcal{P}$(**Symbol**) =
  $\lambda$ ss0 : $\mathcal{P}$(**Symbol**) .
   **Let** ss1 =
    ss0 $\cup$
      **Fold Right**
        $\lambda$ $\langle$t, l, r$\rangle$ : **Shape** . $\lambda$ syms : $\mathcal{P}$(**Symbol**) .
          l $\in$ (from $\cup$ syms) $\rightarrow$ syms $\cup$ r, syms
      **Neutral** ss0 **List** sk
   **In** (ss0 $\subseteq$ ss1) **And** (ss1 $\subseteq$ ss0) $\rightarrow$ ss0, f **On** ss1
 **In** f **On** $\emptyset$.
```

Derivable To _ In _ : $\mathcal{P}$(Symbol) $\times$ Skeleton $\rightarrow$ $\mathcal{P}$(Symbol)

```
$\lambda$ to : $\mathcal{P}$(**Symbol**) . $\lambda$ sk : **Skeleton** .
 **Letrec** f : $\mathcal{P}$(**Symbol**) $\rightarrow$ $\mathcal{P}$(**Symbol**) =
  $\lambda$ ss0 : $\mathcal{P}$(**Symbol**) .
   **Let** ss1 =
    ss0 $\cup$
      **Fold Right**
        $\lambda$ $\langle$t, l, r$\rangle$ : **Shape** . $\lambda$ syms : $\mathcal{P}$(**Symbol**) .
          $\neg$ ((to $\cup$ syms) $\cap$ r) = $\emptyset$ $\rightarrow$ syms $\cup$ {l}, syms
      **Neutral** ss0 **List** sk
   **In** (ss0 $\subseteq$ ss1) **And** (ss1 $\subseteq$ ss0) $\rightarrow$ ss0, f **On** ss1
 **In** f **On** $\emptyset$.
```

From _ To _ In _ : $\mathcal{P}$(Symbol) $\times$ $\mathcal{P}$(Symbol) $\times$ Skeleton $\rightarrow$ $\mathcal{P}$(Symbol)

```
$\lambda$ from : $\mathcal{P}$(**Symbol**) . $\lambda$ to : $\mathcal{P}$(**Symbol**) . $\lambda$ sk : **Skeleton** .
 **Derivable From** from **In** sk $\cap$ **Derivable To** to **In** sk.
```

C.5 Superimposition

Superimpose _ And _ : Rules $\times$ Rules $\rightarrow$ Rules

```
**Let** superimposeEs = $\lambda$ e1 : **Element** . $\lambda$ e2 : **Element** .
 **Element From Symbol Of** e1
  (**Parameters Input Of** e1 ++ **Parameters Input Of** e2) $\rightarrow$
  (**Parameters Output Of** e1 ++ **Parameters Output Of** e2)
**In**
 $\lambda$ rs1 : **Rules** . $\lambda$ rs2 : **Rules** .
*% ensure soundness of superimposition*
  **Skeleton Of** rs1 = **Skeleton Of** (**Order By Tags In** rs1 **On** rs2) $\circ\!\!\rightarrow$
*% iterate the rules*
   **Map** $\lambda$ r1 : **Rule** .
    **Let** $\langle$r2$\rangle$ = **Select Tags** {**Tag Of** r1} **On** rs2 **In**
```

```
% ensure disjoint variables
     Let r2fresh = refreshRule On r2 In
      Rule From Tag Of r1
% superimpose LHSs
       superimposeEs On Conclusion Of r1 On Conclusion Of r2fresh
        ⇐
          Let ⟨rhs, rest⟩ =
% iterate RHSs for superimposition
           Fold Left
            λ ⟨sofar, r2a⟩ : Premise* × Premise* . λ e : Premise .
             Computation? e →
              ⟨sofar ++ ⟨e⟩, r2a⟩,
              Let ⟨skipped, r2b⟩ = skipComputations On r2a In
               ⟨sofar ++ skipped ++ ⟨superimposeEs On e On Head Of r2b⟩, Tail Of r2b⟩
           Neutral ⟨⟨ ⟩, Premises Of r2fresh⟩ List Premises Of r1
          In rhs ++ rest
   List rs1.
```

C.6 Folding & unfolding

Fold _ By _ Into _ : Tag × Symbol?* × Tag → Trafo

```
λ from : Tag . λ syms : Symbol?* . λ to : Tag . λ rs : Rules .
 Let ⟨r⟩ = Select Tags {from} On rs In
%
% remove heading "?"
%
  Letrec affix
   : ⟨Premise*, Premise*, Symbol?*⟩ → ⟨Premise*, Premise*, Symbol?*⟩ =
   λ ⟨es1, es2, ss⟩ : ⟨Premise*, Premise*, Symbol?*⟩ .
    Let ⟨es3, es5⟩ = skipComputations On es2 In
     Let es4 = es1 ++ es3 In
      Head Of ss = ? →
       affix On ⟨ es4 ++ ⟨Head Of es5⟩, Tail Of es5, Tail Of ss ⟩,
       ⟨es4, es5, ss⟩
  In
%
% split the rule in:
%  before: premises matching heading "?"
%  after: premises matching trailing "?"
%  match: premises matching the skeleton symbol
%
   Let ⟨before, rest, tail⟩ = affix On ⟨⟨ ⟩, Premises Of r, syms⟩ In
    Let ⟨ai, mi, ss⟩ = affix On ⟨⟨ ⟩, Reverse rest, Reverse tail⟩ In
     Let after = Reverse ai In
      Let match = Reverse mi In
       (¬ match = ⟨ ⟩) ∘→ Let ⟨s⟩ = ss In
```

```
% acccumulate variables on input or output positions
        Let ios = λ io : Io .
         Fold Left
          λ vs : P(Variable) . λ e : Element .
           vs ∪ Variables In Parameters io Of e
         Neutral ∅ List match
        In
% construct bridge element for fold; construct rules
         Let ins = ios On Input In
          Let outs = ios On Output In
           Let new = Element From s (ins \ outs) → (outs \ ins) In
            (   Forget Tags {from} On rs)
             ⋈ ⟨Rule From Tag Of r Conclusion Of r ⇐ (before ++ ⟨new⟩ ++ after)⟩
             ⋈ ⟨Rule From to new ⇐ match⟩.
```

Unfold _ By _ Into _ : Tag × Tag?* × Tag? → Trafo

```
λ from : Tag . λ ts : Tag* . λ to : Tag? . λ rs : Rules .
 Let ⟨r⟩ = Select Tags {from} On rs In
  Let ⟨lhs, rhs⟩ =
   Letrec pumpRec
    : Tag* → Conclusion → Premise* → Premise* → ⟨Conclusion, Premise*⟩ =
    λ ts : Tag* . λ lhs : Conclusion . λ done : Premise* . λ rest : Premise* .
     ts = ⟨ ⟩ →
      ⟨lhs, done ++ rest⟩,
      Let spot = Head Of rest In
% skip computations
       Computation? spot →
        pumpRec On ts On lhs On (done ++ ⟨spot⟩) On Tail Of rest,
        Head Of ts = ? →
% do not expand premise because of tag "?"
         pumpRec On Tail Of ts On lhs On (done ++ ⟨Head Of rest⟩) On Tail Of rest,
% expand premise according to tag
        Let ⟨r⟩ = Select Tags {Head Of ts} On rs In
         Let fresh = (refreshRule On r) In
          Let s = unifyElements On Head Of rest On Conclusion Of fresh In
           pumpRec
            On Tail Of ts
            On (substituteInElement On s On lhs)
            On (substituteInElements On s On (done ++ Premises Of fresh))
            On (substituteInElements On s On Tail Of rest)
   In
    pumpRec On ts On Conclusion Of r On ⟨ ⟩ On Premises Of r
  In
% modify input rule or add a copy with a new tag
   to = ? →
    Forget Tags {from} On rs ⋈ ⟨Rule From Tag Of r lhs ⇐ rhs⟩,
    rs ⋈ ⟨Rule From to lhs ⇐ rhs⟩.
```

C.7 Deriving chain rules

Chain Rule _ _ $\Leftarrow$ _ : Tag × Symbol × Symbol → Trafo

```
λ tag : **Tag** . λ lhs : **Symbol** . λ rhs : **Symbol** . λ rs : **Rules** .
 rs ⋈
 ⟨
   Let sorts2ps = λ sorts : Sort* .
    Map λ sort : Sort . New Variable Of Sort sort List sorts
   In
    Let prof =
     Let lhsProf = Profile Of lhs In Sigma Of rs In
      Let rhsProf = Profile Of rhs In Sigma Of rs In
       lhsProf = ? →
        ¬ rhsProf = ? ∘→ rhsProf,
        rhsProf = ? →
         lhsProf,
         (Sorts Input Of lhsProf = Sorts Input Of rhsProf) And
         (Sorts Output Of lhsProf = Sorts Output Of rhsProf) ∘→ rhsProf
    In
     Let psI = sorts2ps On Sorts Input Of prof In
      Let psO = sorts2ps On Sorts Output Of prof In
       Rule From tag
        Conclusion From lhs psI → psO
         ⇐ ⟨Premise From rhs psI → psO⟩
 ⟩.
```

Appendix D

A collection of meta-programs

The purpose of this Appendix Chapter is to provide some showcases for nontrivial meta-programs demonstrating the expressive power of the calculus.

D.1 Composition of a simple language definition

A language definition is composed from modules specifying language constructs. More technically, we compose an interpreter definition consisting essentially of a frontend part and a separate dynamic semantics. We consider a very simple imperative programming language with the fundamental imperative constructs (assignment, selection, iteration, sequence, input, output) and only basic data types for integer and Boolean values. The complete example has been checked with $^{\Lambda}\Delta_{\Lambda}$ [HLR97, LRH96, RL93, Rie92].

D.1.1 The structure of the interpreter definition

The *PRA* [LRH96] specification below makes the structure of the interpreter definition explicit. Interpretation consists of two phases. First, the concrete input is analysed, context conditions are checked and an abstract syntactical representation is constructed. Second, the intermediate representation is "interpreted" according to the dynamic semantics.

lcs/examples/basic/main.pra

```
% modular composition of a specification for analysing source programs
frontend : Interpret In
            Refine ./analyser By lib/scanner/trivial
             &static By (Interpret In ./static &st By lcs/adts/simpleSt)
             &ast    As Constructor.
% modular composition of a specification interpreting abstract programs
dynamic : Interpret In ./dynamic
           &bops   By lcs/adts/bops
           &memory By lcs/adts/memory.
```

```
% actual execution of the language processor
With SOURCE, IN Return OUT Do
 % executing the frontend specification in a reading scope
 Reading ./examples/SOURCE.basic Do
  Run frontend → PROG
 End Reading;
 % applying the interpreter specification to the abstract representation
 Run dynamic(PROG, IN) → OUT;
End.
```

Below we show how the main components of the interpreter definition, that is to say the dynamic semantics (*./dynamic*; refer to Subsection D.1.2), the GSF schema of the frontend (*./analyser*; refer to Subsection D.1.3 and Subsection D.1.5), the auxiliary predicates facilitating type checking in the above GSF schema (*./static*; refer to Subsection D.1.4), are composed from atomar specification units. Some rather auxiliary modules are presented in Subsection D.1.6.

D.1.2 Composition of the dynamic semantics

We are seeking for a natural semantics specification for the dynamic semantics of the sample language. The final semantics description is as follows.

lcs/examples/basic/dynamic.pp

```
[prog] &memory init → MEM_4,
       initOUT → OUT_4,
       execute(STM, MEM_4, IN_3, OUT_4) → (MEM_5, IN_5, OUT_5)
       ---------------------------------------------------
       program(prog(STM), IN_3) → OUT_5
[initOUT] initOUT → []\OUT
[assign] evaluate(EXP, MEM, IN_12) → (VAL, IN_15),
         &memory update(MEM, ID, VAL) → MEM'
         ----------------------------------------
         execute(assign(ID, EXP), MEM, IN_12, OUT_8) → (MEM', IN_15, OUT_8)
[skip] execute(skip, MEM_14, IN_26, OUT_18) → (MEM_14, IN_26, OUT_18)
[concat] execute(STM_1, MEM_16, IN_28, OUT_20) → (MEM_19, IN_31, OUT_23),
         execute(STM_2, MEM_19, IN_31, OUT_23) → (MEM_21, IN_33, OUT_25)
         ---------------------------------------------------------
         execute(concat(STM_1, STM_2), MEM_16, IN_28, OUT_20) → (MEM_21, IN_33, OUT_25)
[if] evaluate(EXP, MEM_47, IN_48) → (VAL, IN_51),
     cond(VAL, STM_1, STM_2, MEM_47, IN_51, OUT_38) → (MEM_51, IN_53, OUT_41)
     ------------------------------------------------------------------
     execute(if(EXP, STM_1, STM_2), MEM_47, IN_48, OUT_38) → (MEM_51, IN_53, OUT_41)
[while] concat(STM, while(EXP, STM)) = STM_unfold,
        execute(if(EXP, STM_unfold, skip), MEM_68, IN_66, OUT_54) → (MEM_71, IN_69, OUT_57)
        ---------------------------------------------------------------------------
        execute(while(EXP, STM), MEM_68, IN_66, OUT_54) → (MEM_71, IN_69, OUT_57)
```

```
[output] evaluate(EXP, MEM_95, IN_106) → (VAL, IN_109),
         addOUT(OUT, VAL) → OUT'
         ------------------------------------------
         execute(output(EXP), MEM_95, IN_106, OUT) → (MEM_95, IN_109, OUT')
[var] &memory lookup(MEM, ID) → VAL
      -----------------------------
      evaluate(var(ID), MEM, IN_16) → (VAL, IN_16)
[int] evaluate(const(intC(INT)), MEM_80, IN_86) → (intV(INT), IN_86)
[true] evaluate(const(boolC(true)), MEM_81, IN_88) → (boolV(True\BOOL), IN_88)
[false] evaluate(const(boolC(false)), MEM_82, IN_90) → (boolV(False\BOOL), IN_90)
[monadic] evaluate(EXP, MEM_83, IN_92) → (VAL, IN_95),
          &bops evaluateMonadic(MOS, VAL) → VAL'
          ----------------------------------------
          evaluate(monadic(MOS, EXP), MEM_83, IN_92) → (VAL', IN_95)
[dyadic] evaluate(EXP, MEM_85, IN_96) → (VAL, IN_99),
         evaluate(EXP', MEM_85, IN_99) → (VAL', IN_101),
         &bops evaluateDyadic(DOS, VAL, VAL') → VAL''
         ---------------------------------------------
         evaluate(dyadic(EXP, DOS, EXP'), MEM_85, IN_96) → (VAL'', IN_101)
[input] inputType(T, VAL)
        -----------------
        evaluate(input(T), MEM_89, [VAL|VAL*]\IN) → (VAL, VAL*\IN)
[then] execute(STM, MEM_52, IN_54, OUT_42) → (MEM_55, IN_57, OUT_45)
       -----------------------------------------------------
       cond(boolV(True\BOOL), STM, STM_3, MEM_52, IN_54, OUT_42) → (MEM_55, IN_57, OUT_45)
[else] execute(STM, MEM_56, IN_58, OUT_46) → (MEM_59, IN_61, OUT_49)
       -----------------------------------------------------
       cond(boolV(False\BOOL), STM_4, STM, MEM_56, IN_58, OUT_46) → (MEM_59, IN_61, OUT_49)
[addOUT] OUT ++ [VAL]\OUT → OUT'
         ------------------------
         addOUT(OUT, VAL) → OUT'
[inputInt] VAL = intV(INT_0)
           ----------------
           inputType(intT, VAL)
[inputBool] VAL = boolV(BOOL_0)
            ------------------
            inputType(boolT, VAL)
```

The composition is performed by means of lifting.

lcs/examples/basic/dynamic.sgc

% aspects of computational behaviour
mem : lcs/transformers/memory.
in : lcs/transformers/input.
out : lcs/transformers/output.
Inference Rules
Axiom Is program

```
Lift
 ⟨
  ⟨⟨mem, in, out⟩, lcs/fragments/program/dynamic⟩,
  ⟨⟨in, out⟩, lcs/fragments/variable/dynamic⟩,
  ⟨⟨mem, in, out⟩, lcs/fragments/compound/natural/dynamic⟩,
  ⟨⟨mem, in, out⟩, lcs/fragments/selection/if/deterministic/dynamic⟩,
  ⟨⟨mem, in, out⟩, lcs/fragments/iteration/while/dynamic⟩,
  ⟨⟨mem, in⟩, lcs/fragments/type/dynamic⟩,
  ⟨⟨mem⟩, lcs/fragments/input/dynamic⟩,
  ⟨⟨mem, in⟩, lcs/fragments/output/dynamic⟩
 ⟩
```

There are three semantic aspects, that is to say memory propagation including initialization (*mem*), inputs (*in*) and output (*out*). The corresponding transformers are shown below.

lcs/transformers/memory.fra

```
λ sk : Skeleton .
   Default For MEM By &memory init
 ∘ (Inherit MEM From {program} To {evaluate} On sk)
 ∘ (Accumulate MEM From {program} To {execute} On sk)
```

lcs/transformers/input.fra

```
λ sk : Skeleton .
 Let closure = (From {program} To {evaluate} In sk) ∪ {evaluate} In
   Left To Right IN
 ∘ Ensure Positions Output For closure Of Sort IN
 ∘ Ensure Positions Input For closure ∪ {program} Of Sort IN
```

lcs/transformers/output.fra

```
λ sk : Skeleton .
 Let closure = (From {program} To {execute} In sk) ∪ {execute} In
    Default For OUT By initOUT
  ∘ Left To Right OUT
  ∘ Ensure Positions Output For closure ∪ {program} Of Sort OUT
  ∘ Ensure Positions Input For closure Of Sort OUT
```

Now we present the modular semantics of the underlying language constructs. Usually, we give a short $\Lambda\Delta\Lambda$ interface description for the abstract syntax (a module name ending with *as.if*) and a fragment of natural semantics (a module name ending with *dynamic.ir*) to be regarded as rules at some level in the terminology of lifting.

Declarations are not regarded as relevant for the semantics definition. Thus, the following abstract syntax for entire programs is appropriate.

lcs/fragments/program/as.if

```
PROG = prog(STM)
```

To interprete a program, means to execute the statements.

lcs/fragments/program/dynamic.ir

```
% abstract syntax
Include ./as
% semantic functions
program: PROG
execute: STM

[prog] execute(STM)
       -----------------
       program(prog(STM))
```

The concept of the imperative variable provides a building block for imperative languages. There are two important constructs, that is to say assignment and variables (as expressions) with the following abstract syntactical representation.

lcs/fragments/variable/as.if

```
STM = assign(ID, EXP)
EXP = var(ID)
```

The semantics of the above constructs is easily defined. To evaluate a variable identifier, the memory is observed. To execute an assignment, the memory is updated. The relational symbols prefixed by $\&_{memory}$ are concerned with memory access. The module *lcs/adts/memory.ir* providing the corresponding interpretations is presented in Subsection D.1.6.

lcs/fragments/variable/dynamic.ir

```
% abstract syntax
Include ./as
% semantic functions
execute: STM × MEM → MEM
evaluate: EXP × MEM → VAL

% assignment
[assign] evaluate(EXP, MEM) → VAL,
         &memory update(MEM, ID, VAL) → MEM'
         ------------------------------------
         execute(assign(ID, EXP), MEM) → MEM'
% variables as expressions
[var] &memory lookup(MEM, ID) → VAL
      -----------------------------
      evaluate(var(ID), MEM) → VAL
```

Statements can be composed in the sense of statement sequences. The following piece of abstract syntax introduces the empty statement and the compound statement.

lcs/fragments/compound/as.if

```
STM = skip + concat(STM, STM)
```

The execution of the empty statement is modelled by a simple axiom, whereas the execution of a compound statement means sequenced execution.

lcs/fragments/compound/natural/dynamic.ir

```
% abstract syntax
Include ../as
% semantic functions
execute: STM × MEM → MEM

% semantics of the empty statement (sequence)
[skip] execute(skip, MEM) → MEM
% semantics of a sequence of statements
[concat] execute(STM₁, MEM) → MEM',
         execute(STM₂, MEM') → MEM''
         -----------------------------------------
         execute(concat(STM₁, STM₂), MEM) → MEM''
```

If-statements are well-known representatives of the class of statements serving for selection. We assume the following abstract syntactical representation.

lcs/fragments/selection/if/as.if

```
STM = if(EXP, STM, STM)
```

The semantics of an *if*-statement is defined below in a deterministic style, i.e. first the condition is evaluated and then an auxiliary relation *cond* is used to execute either the *then*-part or the *else*-part depending on the value of the condition.

lcs/fragments/selection/if/deterministic/dynamic.ir

```
% abstract syntax
Include ../as
% values
VAL = boolV(BOOL)
BOOL = Boolean
% semantic functions
execute: STM
evaluate: EXP → VAL
cond: VAL × STM × STM
```

```
% first evaluate condition, then branch on value
[if] evaluate(EXP) → VAL,
     cond(VAL, STM_1, STM_2)
     ----------------------------
     execute(if(EXP, STM_1, STM_2))

% execute the Then-path of an If-statement
[then] execute(STM)
       ----------------------------
       cond(boolV(True), STM, _STM)
% execute the Else-path of an If-statement
[else] execute(STM)
       -----------------------------
       cond(boolV(False), _STM, STM)
```

While-loops are well-known representatives of the class of statements serving for iteration. We assume the following abstract syntactical representation.

lcs/fragments/iteration/while/as.if

```
STM = skip + concat(STM, STM) + if(EXP, STM, STM) + while(EXP, STM)
```

The semantics of a *while*-statement is defined below by a kind of unfolding, i.e. the semantics is expressed in terms of an *if*-statement.

lcs/fragments/iteration/while/dynamic.ir

```
% abstract syntax
Include ./as
% semantic functions
execute: STM

[while] concat(STM, while(EXP, STM)) = STM_unfold,
        execute(if(EXP, STM_unfold, skip))
        ----------------------------------------
        execute(while(EXP, STM))
```

To cope with simple forms of expressions according to basic data types (constants, monadic and dyadic expressions), the following piece of abstract syntax is needed.

lcs/fragments/type/as.if

```
EXP  = const(C) + monadic(MOS, EXP) + dyadic(EXP, DOS, EXP)
C    = intC(INT) + boolC(BC)
INT  = Integer
BC   = true + false
```

The evaluation of all the above kinds of constants, monadic and dyadic expressions is shown below.

lcs/fragments/type/dynamic.ir

```
% abstract syntax
Include ./as
% values
VAL  = intV(INT) + boolV(BOOL)
INT  = Integer
BOOL = Boolean
% semantic functions
evaluate: EXP → VAL

[int] evaluate(const(intC(INT))) → intV(INT)
[true] evaluate(const(boolC(true))) → boolV(True)
[false] evaluate(const(boolC(false))) → boolV(False)
[monadic] evaluate(EXP) → VAL,
          &bops evaluateMonadic(MOS, VAL) → VAL'
          ---------------------------------------
          evaluate(monadic(MOS, EXP)) → VAL'
[dyadic] evaluate(EXP) → VAL,
         evaluate(EXP') → VAL',
         &bops evaluateDyadic(DOS, VAL, VAL') → VAL''
         ---------------------------------------------
         evaluate(dyadic(EXP, DOS, EXP')) → VAL''
```

The application of corresponding basic operations is modelled by the premises prefixed by $\&_{bops}$. The module *lcs/adts/bops.ir* providing the corresponding interpretations is presented in Subsection D.1.6.

To consume an input value is regarded as a form of an expression. Thus, the following abstract representation for an input construct is appropriate.

lcs/fragments/input/as.if

```
EXP = input(T)
T   = intT + boolT
```

To evaluate an input expression means to consume the head of the propagated input, where the head is regarded at the same time as the value of the expression.

lcs/fragments/input/dynamic.ir

```
% abstract syntax
Include ./as
% semantic domains
VAL  = intV(INT) + boolV(BOOL)
INT  = Integer
BOOL = Boolean
IN   = VAL*
% semantic/auxiliary functions
evaluate: EXP × IN → VAL × IN
inputType: T × VAL
```

```
[input] inputType(T, VAL)
        --------------------------------------------
        evaluate(input(T), [VAL|VAL*]) → (VAL, VAL*)

[inputInt] VAL = intV(_INT) ⇒ inputType(intT, VAL)

[inputBool] VAL = boolV(_BOOL) ⇒ inputType(boolT, VAL)
```

Producing output is regarded as a side-effect similar to an assignment. The following abstract syntactical representation is suggested.

lcs/fragments/output/as.if

```
STM = output(EXP)
```

The semantic meaning of an output statement is modelled as follows. The value of the expression to be written is appended with the propagated output.

lcs/fragments/output/dynamic.ir

```
% abstract syntax
Include ./as

% semantic domain
VAL  = intV(INT) + boolV(BOOL)
INT  = Integer
BOOL = Boolean
OUT  = VAL*

% semantic functions
evaluate: EXP → VAL
execute: STM × OUT → OUT

[output] evaluate(EXP) → VAL,
         addOUT(OUT, VAL) → OUT'
         ---------------------------------
         execute(output(EXP), OUT) → OUT'

% return the empty output
initOUT: → OUT

[initOUT] initOUT → []

% extend the output consumed already by a value
addOUT: OUT × VAL → OUT

[addOUT] OUT ++ [VAL]\OUT → OUT'
         ------------------------
         addOUT(OUT, VAL) → OUT'
```

D.1.3 Composition of the frontend

We are seeking for a GSF schema defining the syntax, static semantics and the construction of an abstract syntactical representation for our sample language. The final GSF schema looks as follows.

lcs/examples/basic/structure.pp

```
[prog] program → PROG_0
        :
          &static initST → ST_41,
          declarations(ST_41) → ST_42,
          statements(ST_42) → STM_5,
          &ast prog(STM_5) → PROG_0.
[decs] declarations(ST_44) → ST_49
        :
          declaration(ST_44) → ST_47,
          declarations(ST_47) → ST_49.
[nodecs] declarations(ST_50) → ST_50 : .
[concat] statements(ST_58) → STM_6
          :
            statement(ST_58) → STM_12,
            statements(ST_58) → STM_7,
            &ast concat(STM_12, STM_7) → STM_6.
[skip] statements(ST_61) → STM_8
        :
          &ast skip → STM_8.
[vdec] declaration(ST) → ST'
        :
          id → ID,
          type → T,
          &static addVar(ST, ID, T) → ST'.
[if] statement(ST_62) → STM_13
      :
        expression(ST_62) → (T, EXP_6),
        &static isBoolType(T),
        statements(ST_62) → STM_10,
        statements(ST_62) → STM_9,
        &ast if(EXP_6, STM_10, STM_9) → STM_13.
[while] statement(ST_66) → STM_14
         :
           expression(ST_66) → (T, EXP_7),
           &static isBoolType(T),
           statements(ST_66) → STM_11,
           &ast while(EXP_7, STM_11) → STM_14.
[output] statement(ST_70) → STM_15
          :
            expression(ST_70) → (T, EXP_9),
            &static isOutputType(T),
            &ast output(EXP_9) → STM_15.
```

```
[assign] statement(ST) → STM_{16}
          :
           id → ID,
           &static isVar(ST, ID) → T,
           expression(ST) → (T', EXP_{11}),
           &static assignable(T, T'),
           &ast assign(ID, EXP_{11}) → STM_{16}.
[intT] type → intT : .
[boolT] type → boolT : .
[const] expression(ST_{52}) → (T, EXP_{0})
         :
          constant → (T, C_{0}),
          &ast const(C_{0}) → EXP_{0}.
[monadic] expression(ST_{53}) → (T, EXP_{1})
           :
            mos → MOS,
            expression(ST_{53}) → (T', EXP_{2}),
            &static profileMonadic(MOS, T') → T,
            &ast monadic(MOS, EXP_{2}) → EXP_{1}.
[dyadic] expression(ST_{55}) → (T, EXP_{3})
          :
           expression(ST_{55}) → (T_{1}, EXP_{5}),
           dos → DOS,
           expression(ST_{55}) → (T_{2}, EXP_{4}),
           &static profileDyadic(DOS, T_{1}, T_{2}) → T,
           &ast dyadic(EXP_{5}, DOS, EXP_{4}) → EXP_{3}.
[input] expression(ST_{69}) → (T, EXP_{8})
         :
          type → T,
          &static isInputType(T),
          &ast input(T) → EXP_{8}.
[var] expression(ST) → (T, EXP_{10})
       :
        id → ID,
        &static isVar(ST, ID) → T,
        &ast var(ID) → EXP_{10}.
[boolC] constant → (boolT, C_{1})
         :
          boolean → BC_{0},
          &ast boolC(BC_{0}) → C_{1}.
[intC] constant → (intT, C_{2})
        :
         nat → INT_{0},
         &ast intC(INT_{0}) → C_{2}.
[neg] mos → neg : .
[not] mos → not : .
[plus] dos → plus : .
[minus] dos → minus : .
```

```
[times] dos → times : .
[div] dos → div : .
[eq] dos → eq : .
[neq] dos → neq : .
[gt] dos → gt : .
[lt] dos → lt : .
[ge] dos → ge : .
[le] dos → le : .
[and] dos → and : .
[or] dos → or : .
[true] boolean → BC₁
        :
          &ast true → BC₁.
[false] boolean → BC₂
          :
            &ast false → BC₂.
```

We use two prefixes for different kinds of relational formulae. The prefix $\&_{ast}$ refers to AST construction, whereas the prefix $\&_{static}$ qualifies relational formulae modelling static semantics. The interpretation of the first kind of relational symbols is simply term construction, whereas the relational formulae dealing with static semantics are interpreted by the relations discussed in Subsection D.1.4.

The underlying context-free grammar of the above GSF schema specifies a rather abstract syntax. The more or less trivial adaptation to cope with a more concrete syntax is the subject of Subsection D.1.5.

The composition of the above GSF schema is performed by means of lifting.

lcs/examples/basic/structure.sgc

```
% aspects of computational behaviour
st : lcs/transformers/simpleSt.
as : lcs/transformers/ast On ./dynamic On Output On {declarations} On &ast.

Gsf Scheme
Axiom Is program

Lift
 ⟨
  ⟨⟨st, as⟩,    lcs/fragments/program/structure
              ⋈ lcs/fragments/declarations/structure
              ⋈ lcs/fragments/type/structure
              ⋈ lcs/fragments/compound/structure
              ⋈ lcs/fragments/selection/if/structure
              ⋈ lcs/fragments/iteration/while/structure
              ⋈ lcs/fragments/input/structure
              ⋈ lcs/fragments/output/structure
  ⟩,
  ⟨⟨as⟩, lcs/fragments/variable/structure⟩
 ⟩
```

There are two computational aspects covered by transformers. The aspect *st* deals with the initialization and the propagation of the symbol table, whereas the aspect *as* deals with the construction of an abstract syntactical representation. Remember that ASTs need to be synthesized, because they are interpreted in the second phase of the interpreter, i.e. by the dynamic semantics. The definition of the transformer for the aspect *st* is shown below.

lcs/transformers/simpleSt.fra

λ sk : **Skeleton** .
  (./defaultSt **On** sk)
 ∘ (./inheritSt **On** sk)
 ∘ (./accumulateSt **On** sk)

To have a more modular definition of the above transformer, the following three components were identified.

lcs/transformers/defaultSt.fra

λ sk : **Skeleton** . **Default For** ST **By** &static initST

lcs/transformers/inheritSt.fra

λ sk : **Skeleton** . **Inherit** ST **From** {program} **To** {expression} **On** sk

lcs/transformers/accumulateSt.fra

λ sk : **Skeleton** . **Accumulate** ST **From** {program} **To** {declaration} **On** sk

The transformer for the aspect *as* is not presented here, because the actual definition is not language-specific. Its definition is considered in some depth in Section D.7, because it is interesting on its own. We only want to comment on the underlying generic approach. The transformer *as* could be defined in terms of a **Relate Output** ... transformation, but we also can compute such a transformation by unifying the skeleton to be lifted and the signature of a "reference" specification, where the dynamic semantics specifiation serves for this purpose here. Technically, term constructors and sorts in the signature are unified with tags and symbols in the skeleton.

Now we present the rules (at some level) from which the above GSF schema has been composed. We start with the overall structure of programs.

lcs/fragments/program/structure.gs

```
[prog] program : declarations, statements.
```

Sequences of declarations are defined as follows.

lcs/fragments/declarations/structure.gs

```
[decs] declarations : declaration, declarations.
[nodecs] declarations : .
```

The rules concerning type expressions, constants, monadic and dyadic expressions according to basic data types are the following.

lcs/fragments/type/structure.gs

```
[intT] type → intT : .
[boolT] type → boolT : .
[const] expression → T
         :
           constant → T.
[monadic] expression → T
           :
             mos → MOS,
             expression → T',
             &static profileMonadic(MOS, T') → T.
[dyadic] expression → T
          :
            expression → T1,
            dos → DOS,
            expression → T2,
            &static profileDyadic(DOS, T1, T2) → T.

[boolC] constant → boolT : boolean.
[intC] constant → intT : nat.
[true] boolean : .
[false] boolean : .
[neg] mos → neg : .
[not] mos → not : .
[plus] dos → plus : .
[minus] dos → minus : .
[times] dos → times : .
[div] dos → div : .
[eq] dos → eq : .
[neq] dos → neq : .
[gt] dos → gt : .
[lt] dos → lt : .
[ge] dos → ge : .
[le] dos → le : .
[and] dos → and : .
[or] dos → or : .
```

Statement sequences are specified by the following rules.

lcs/fragments/compound/structure.gs

```
[concat] statements
          :
            statement,
            statements.
[skip] statements : .
```

The concept of an imperative variable effects several syntactical classes. Variable declarations, assignments and variables as expressions need to be specified.

lcs/fragments/variable/structure.gs

```
[vdec] declaration(ST) → (ST')
        :
          id → ID,
          type → T,
          &static addVar(ST, ID, T) → ST'.
[var] expression(ST) → T
       :
         id → ID,
         &static isVar(ST, ID) → T.
[assign] statement(ST)
          :
            id → ID,
            &static isVar(ST, ID) → T,
            expression(ST) → T',
            &static assignable(T, T').
```

If- and *while*-statements are specified below. As far as static semantics is concerned, we want to ensure that conditions are Boolean expressions.

lcs/fragments/selection/if/structure.gs

```
[if] statement
      :
        expression → T,
        &static isBoolType(T),
        statements,
        statements.
```

lcs/fragments/iteration/while/structure.gs

```
[while] statement
         :
           expression → T,
           &static isBoolType(T),
           statements.
```

Input expressions and output statements are specified below. Possibly, the types legal for input or output need to be restricted.

lcs/fragments/input/structure.gs

```
[input] expression → T
          :
            type → T,
            &static isInputType(T).
```

lcs/fragments/output/structure.gs

```
[output] statement
          :
            expression → T,
            &static isOutputType(T).
```

D.1.4 Auxiliary relations for the static semantics

We develop the module providing interpretations for relational symbols prefixed by $\&_{static}$ in the GSF schema above.

lcs/examples/basic/static.pp

```
[initST] &st init → ST
         --------------
         initST → ST
[addVar] &st add(ST, ID, varEntry(T)) → ST'
         -----------------------------------
         addVar(ST, ID, T) → ST'
[isVar] &st lookup(ST, ID) → varEntry(T)
        ---------------------------------
        isVar(ST, ID) → T
[assignable] equalTypes(T_lhs, T_rhs)
             ----------------------
             assignable(T_lhs, T_rhs)
[isIntType] isIntType(intT)
[isBoolType] isBoolType(boolT)
[profile1] profileMonadic(neg, intT) → intT
[profile2] profileMonadic(not, boolT) → boolT
[profile3] profileDyadic(plus, intT, intT) → intT
[profile4] profileDyadic(minus, intT, intT) → intT
[profile5] profileDyadic(times, intT, intT) → intT
[profile6] profileDyadic(div, intT, intT) → intT
[profile7] profileDyadic(eq, intT, intT) → boolT
[profile8] profileDyadic(neq, intT, intT) → boolT
```

```
[profile9] profileDyadic(lt, intT, intT) → boolT
[profileA] profileDyadic(gt, intT, intT) → boolT
[profileB] profileDyadic(ge, intT, intT) → boolT
[profileC] profileDyadic(le, intT, intT) → boolT
[profileD] profileDyadic(eq, boolT, boolT) → boolT
[profileE] profileDyadic(neq, boolT, boolT) → boolT
[profileF] profileDyadic(lt, boolT, boolT) → boolT
[profileG] profileDyadic(gt, boolT, boolT) → boolT
[profileH] profileDyadic(ge, boolT, boolT) → boolT
[profileI] profileDyadic(le, boolT, boolT) → boolT
[profileJ] profileDyadic(and, boolT, boolT) → boolT
[profileK] profileDyadic(or, boolT, boolT) → boolT
[tEqT] equalTypes(T, T)
[inputInt] isIntType(T)
           ------------
           isInputType(T)
[inputBool] isBoolType(T)
            -------------
            isInputType(T)
[outputInt] isIntType(T)
            ------------
            isOutputType(T)
[outputBool] isBoolType(T)
             -------------
             isOutputType(T)
```

The above specification is obtained by a simple concatenation of some rules.

lcs/examples/basic/static.sgc

Inference Rules

lcs/fragments/program/static
⋈ lcs/fragments/variable/static
⋈ lcs/fragments/type/static
⋈ lcs/fragments/input/static
⋈ lcs/fragments/output/static

The rules below are related to entire programs, constructs for variables, basic data types or I/O constructs. The following modules are used in the above composition.

lcs/fragments/program/static.ir

```
% return the empty symbol table
initST: → ST
[initST] &st init → ST
         --------------
         initST → ST
```

lcs/fragments/variable/static.ir

```
% symbol table entries
INFO = varEntry(T)

% add an entry for a variable to a symbol table
addVar: ST × ID × T → ST
[addVar] &st add(ST, ID, varEntry(T)) → ST'
         ----------------------------------
         addVar(ST, ID, T) → ST'

% lookup an entry for a variable in a symbol table
[isVar] &st lookup(ST, ID) → varEntry(T)
        ---------------------------------
        isVar(ST, ID) → T

% check two types to be compatible for assignment
assignable: T × T
[assignable] equalTypes(T_lhs, T_rhs)
             ----------------------
             assignable(T_lhs, T_rhs)
```

lcs/fragments/type/static.ir

```
T = intT + boolT
MOS = neg + not
DOS = plus + minus + times + div
    + eq + neq + gt + lt + ge + le
    + and + or
isIntType: T
isBoolType: T
profileMonadic: MOS × T → T
profileDyadic: DOS × T × T → T

% test for integer/boolean type
[isIntType] isIntType(intT)
[isBoolType] isBoolType(boolT)

% compute result type for unary operators
[profile1] profileMonadic(neg, intT) → intT
[profile2] profileMonadic(not, boolT) → boolT
```

```
% compute result type for binary operators
[profile3] profileDyadic (plus, intT, intT) → intT
[profile4] profileDyadic (minus, intT, intT) → intT
[profile5] profileDyadic (times, intT, intT) → intT
[profile6] profileDyadic (div, intT, intT) → intT
[profile7] profileDyadic (eq, intT, intT) → boolT
[profile8] profileDyadic (neq, intT, intT) → boolT
[profile9] profileDyadic (lt, intT, intT) → boolT
[profileA] profileDyadic (gt, intT, intT) → boolT
[profileB] profileDyadic (ge, intT, intT) → boolT
[profileC] profileDyadic (le, intT, intT) → boolT
[profileD] profileDyadic (eq, boolT, boolT) → boolT
[profileE] profileDyadic (neq, boolT, boolT) → boolT
[profileF] profileDyadic (lt, boolT, boolT) → boolT
[profileG] profileDyadic (gt, boolT, boolT) → boolT
[profileH] profileDyadic (ge, boolT, boolT) → boolT
[profileI] profileDyadic (le, boolT, boolT) → boolT
[profileJ] profileDyadic (and, boolT, boolT) → boolT
[profileK] profileDyadic (or, boolT, boolT) → boolT

% equivalence of types
[tEqT] equalTypes(T, @T)
```

lcs/fragments/input/static.ir

```
isInputType: T
[inputInt] isIntType(T) ⇒ isInputType(T)
[inputBool] isBoolType(T) ⇒ isInputType(T)
```

lcs/fragments/output/static.ir

```
[outputInt] isIntType(T) ⇒ isOutputType(T)
[outputBool] isBoolType(T) ⇒ isOutputType(T)
```

D.1.5 The frontend coping with concrete syntax

It is shown how the GSF schema from Subsection D.1.3 can be adapted to cope with a rather concrete syntax. The transformational approach which was taken here is rather pragmatic. We refer to [KW96] for a rather disciplined alternative. There, an approach is presented which simplifies the design of the grammars representing concrete and abstract syntax as well as the mapping between them.

First, the final GSF schema is shown.

lcs/examples/basic/analyser.pp

```
[prog] program → PROG$_0$
        :
          &static initST → ST$_{41}$,
          declarations(ST$_{41}$) → ST$_{42}$,
          “Begin”,
          statements(ST$_{42}$) → STM$_5$,
          “End”,
          “.”,
          &ast prog(STM$_5$) → PROG$_0$.
[decs] declarations(ST$_{44}$) → ST$_{49}$
        :
          declaration(ST$_{44}$) → ST$_{47}$,
          “;”,
          declarations(ST$_{47}$) → ST$_{49}$.
[nodecs] declarations(ST$_{50}$) → ST$_{50}$ : .
[concat] statements(ST$_{58}$) → STM$_6$
          :
            statement(ST$_{58}$) → STM$_{12}$,
            “;”,
            statements(ST$_{58}$) → STM$_7$,
            &ast concat(STM$_{12}$, STM$_7$) → STM$_6$.
[skip] statements(ST$_{61}$) → STM$_8$
        :
          &ast skip → STM$_8$.
[vdec] declaration(ST) → ST’
        :
          “Var”,
          id → ID,
          “:”,
          type → T,
          &static addVar(ST, ID, T) → ST’.
[while] statement(ST$_{66}$) → STM$_{14}$
         :
           “While”,
           expression(ST$_{66}$) → (T, EXP$_7$),
           “Do”,
           &static isBoolType(T),
           statements(ST$_{66}$) → STM$_{11}$,
           “End”,
           &ast while(EXP$_7$, STM$_{11}$) → STM$_{14}$.
[output] statement(ST$_{70}$) → STM$_{15}$
          :
            “Output”,
            expression(ST$_{70}$) → (T, EXP$_9$),
            &static isOutputType(T),
            &ast output(EXP$_9$) → STM$_{15}$.
```

```
[assign] statement(ST) → $STM_{16}$
          :
            id → ID,
            ":=",
            &static isVar(ST, ID) → T,
            expression(ST) → (T', $EXP_{11}$),
            &static assignable(T, T'),
            &ast assign(ID, $EXP_{11}$) → $STM_{16}$.
[if] statement($ST_{62}$) → $STM_{13}$
      :
        "If",
        expression($ST_{62}$) → (T, $EXP_6$),
        "Then",
        &static isBoolType(T),
        statements($ST_{62}$) → $STM_{10}$,
        else($ST_{62}$) → $STM_9$,
        "End",
        &ast if($EXP_6$, $STM_{10}$, $STM_9$) → $STM_{13}$.
[intT] type → intT
        :
          "Integer".
[boolT] type → boolT
         :
          "Boolean".
[const] expression($ST_{52}$) → (T, $EXP_0$)
         :
          constant → (T, $C_0$),
          &ast const($C_0$) → $EXP_0$.
[monadic] expression($ST_{53}$) → (T, $EXP_1$)
           :
            mos → MOS,
            expression($ST_{53}$) → (T', $EXP_2$),
            &static profileMonadic(MOS, T') → T,
            &ast monadic(MOS, $EXP_2$) → $EXP_1$.
[dyadic] expression($ST_{55}$) → (T, $EXP_3$)
          :
            "(",
            expression($ST_{55}$) → ($T_1$, $EXP_5$),
            dos → DOS,
            expression($ST_{55}$) → ($T_2$, $EXP_4$),
            ")",
            &static profileDyadic(DOS, $T_1$, $T_2$) → T,
            &ast dyadic($EXP_5$, DOS, $EXP_4$) → $EXP_3$.
[input] expression($ST_{69}$) → (T, $EXP_8$)
         :
          "Input",
          type → T,
          &static isInputType(T),
          &ast input(T) → $EXP_8$.
```

```
[var] expression(ST) → (T, EXP_{10})
       :
         id → ID,
         &static isVar(ST, ID) → T,
         &ast var(ID) → EXP_{10}.
[else] else(ST_{62}) → STM_{9}
        :
          “Else”,
          statements(ST_{62}) → STM_{9}.
[noelse] else(ST_{72}) → STM_{18}
          :
            &ast skip → STM_{18}.
[boolC] constant → (boolT, C_{1})
         :
           boolean → BC_{0},
           &ast boolC(BC_{0}) → C_{1}.
[intC] constant → (intT, C_{2})
        :
          nat → INT_{0},
          &ast intC(INT_{0}) → C_{2}.
[neg] mos → neg
       :
         “−”.
[not] mos → not
       :
         “Not”.
[plus] dos → plus
        :
          “+”.
[minus] dos → minus
         :
           “−”.
[times] dos → times
         :
           “∗”.
[div] dos → div
       :
         “Div”.
[eq] dos → eq
      :
        “=”.
[neq] dos → neq
       :
         “≠”.
[gt] dos → gt
      :
        “>”.
```

```
[lt] dos → lt
      :
        "<".
[ge] dos → ge
      :
        "≥".
[le] dos → le
      :
        "≤".
[and] dos → and
       :
         "And".
[or] dos → or
      :
        "Or".
[true] boolean → BC₁
        :
          "True",
          &ast true → BC₁.
[false] boolean → BC₂
         :
           "False",
           &ast false → BC₂.
```

The above GSF schema is obtained by transforming the GSF schema from Subsection D.1.3 (*./structure*). Essentially, certain keywords and separators are inserted (refer to the application of the operator **Concretize By**) and the rule for *if*-statement is adapted to cope with an optional *else*-part.

lcs/examples/basic/analyser.sgc

```
Gsf Scheme
Axiom Is program

Concretize By
 (  lcs/fragments/program/concrete
  ++ lcs/fragments/declarations/concrete
  ++ lcs/fragments/type/concrete
  ++ lcs/fragments/compound/concrete
  ++ lcs/fragments/selection/if/optional/concrete
  ++ lcs/fragments/iteration/while/concrete
  ++ lcs/fragments/variable/concrete
  ++ lcs/fragments/input/concrete
  ++ lcs/fragments/output/concrete
 )
On
 ( lcs/tools/dyadicInBrackets
 ∘ (lcs/tools/ifOptional On [skip])
   On ./structure
 )
```

First, all the trivial fragments used in the application of the operator **Concretize By** are presented.

lcs/fragments/program/concrete.fra

```
⟨⟨[prog], ⟨?, “Begin”, ?, “End”, “.”⟩⟩⟩
```

lcs/fragments/declarations/concrete.fra

```
⟨⟨[decs], ⟨?, “;”, ?⟩⟩⟩
```

lcs/fragments/type/concrete.fra

```
⟨
  ⟨[intT], ⟨“Integer”⟩⟩,
  ⟨[boolT], ⟨“Boolean”⟩⟩,
  ⟨[true], ⟨“True”⟩⟩,
  ⟨[false], ⟨“False”⟩⟩,
  ⟨[neg], ⟨“\”⟩⟩,
  ⟨[not], ⟨“Not”⟩⟩,
  ⟨[plus], ⟨“+”⟩⟩,
  ⟨[minus], ⟨“\”⟩⟩,
  ⟨[times], ⟨“*”⟩⟩,
  ⟨[div], ⟨“Div”⟩⟩,
  ⟨[eq], ⟨“=”⟩⟩,
  ⟨[neq], ⟨“⟨ ⟩”⟩⟩,
  ⟨[gt], ⟨“⟩”⟩⟩,
  ⟨[lt], ⟨“⟨”⟩⟩,
  ⟨[ge], ⟨“≥”⟩⟩,
  ⟨[le], ⟨“≤”⟩⟩,
  ⟨[and], ⟨“And”⟩⟩,
  ⟨[or], ⟨“Or”⟩⟩
⟩
```

lcs/fragments/compound/concrete.fra

```
⟨⟨[concat], ⟨?, “;”, ?⟩⟩⟩
```

lcs/fragments/selection/if/optional/concrete.fra

```
⟨
 ⟨[if], ⟨“If”, ?, “Then”, ?, ?, “End”⟩⟩,
 ⟨[else], ⟨“Else”, ?⟩⟩
⟩
```

lcs/fragments/iteration/while/concrete.fra

$\langle\langle$[while], $\langle$"*While*", ?, "*Do*", ?, "*End*"$\rangle\rangle$

lcs/fragments/variable/concrete.fra

```
⟨
  ⟨[vdec], ⟨"Var", ?, ":", ?⟩⟩,
  ⟨[assign], ⟨?, ":=", ?⟩⟩
⟩
```

lcs/fragments/input/concrete.fra

$\langle\langle$[input], $\langle$"*Input*", ?$\rangle\rangle$

lcs/fragments/output/concrete.fra

$\langle\langle$[output], $\langle$"*Output*", ?$\rangle\rangle$

The following transformation adapts the rule for dyadic expressions to force enclosing brackets. Thereby, priorities of operation symbols become irrelevant.

lcs/tools/dyadicInBrackets.fra

Concretize By $\langle\langle$[dyadic], $\langle$"*(*", ?, ?, ?, "*)*"$\rangle\rangle$

The following transformation installs an optional *else*-part for *if*-statements.

lcs/tools/ifOptional.fra

```
λ skip : Tag .
  Unfold [else] By ⟨skip⟩ Into [noelse]
∘ Fold [if] By ⟨?, ?, else⟩ Into [else]
```

Note that the above approach based on folding and unfolding has been described in Example 3.3.5.

D.1.6 Auxiliary modules

For completeness, all the auxiliary modules used in the composition of the sample language are included below. First, a suitable scanner definition is shown (*lib/scanner/trivial.lg*). Second, a simple symbol table management module is presented (*lcs/adts/simpleSt.ir*). Third, the application of basic operations (of the sample language) is specified in terms of basic operations of $^{\Lambda}\Delta_{\Lambda}$. Finally, an ADT for memories is included (*lcs/adts/memory.ir*).

lib/scanner/trivial.lg

```
Sets
 letter         = 'A' .. 'Z' | 'a' .. 'z'.
 digit          = '0' .. '9'.
 but_eoln       = Any − Eoln.
 but_star       = Any − '*'.
 but_div_star   = Any − "/*".
Classes
 spaces     = (Space | Tab | Eoln)+.
 id         = letter (letter | digit)* : lib/conv chars2identifier.
 nat        = digit+                   : lib/conv chars2integer.
 end        = Eof.
 comment    = '/' '*' ( but_star | '*'+ but_div_star )* '*'+ '/'
            | '%' but_eoln*.
Switches
 Skip spaces.
 Skip comment.
```

lcs/adts/simpleSt.ir

```
ST    = ENTRY*
ENTRY = <ID, INFO>

% return the empty symbol table

init: → ST

init → []

% add an entry

add: ST × ID × INFO → ST

[add] ¬ ENTRY* = _ ++ [<@ID, _>] ++ _
      -------------------------------------------
      add(ENTRY*, ID, INFO) → [<ID, INFO>|ENTRY*]

% lookup an entry

lookup: ST × ID → INFO

[lookup] lookup(_ ++ [<ID, INFO>] ++ _, @ID) → INFO
```

lcs/adts/bops.ir

```
INT  = Integer
BOOL = Boolean
VAL  = intV(INT) + boolV(BOOL)
MOS  = neg + not
DOS  = plus + minus + times + div
     + eq + neq + gt + lt + ge + le
     + and + or

evaluateMonadic: MOS × VAL → VAL
evaluateDyadic: DOS × VAL × VAL → VAL
```

```
[neg] −INT → INT'
      ---------------------------------------------
      evaluateMonadic(neg, intV(INT)) → intV(INT')

[not1] evaluateMonadic(not, boolV(True)) → boolV(False)
[not2] evaluateMonadic(not, boolV(False)) → boolV(True)

[plus]  INT1 + INT2 → INT
        --------------------------------------------------------
        evaluateDyadic(plus, intV(INT1), intV(INT2)) → intV(INT)

[minus] INT1 − INT2 → INT
        ---------------------------------------------------------
        evaluateDyadic(minus, intV(INT1), intV(INT2)) → intV(INT)

[times] INT1 * INT2 → INT
        ---------------------------------------------------------
        evaluateDyadic(times, intV(INT1), intV(INT2)) → intV(INT)

[div]   INT1 // INT2 → INT
        -------------------------------------------------------
        evaluateDyadic(div, intV(INT1), intV(INT2)) → intV(INT)

[and] ? BOOL ← BOOL1 ∧ BOOL2
      -------------------------------------------------------------
      evaluateDyadic(and, boolV(BOOL1), boolV(BOOL2)) → boolV(BOOL)

[or] ? BOOL ← BOOL1 ∨ BOOL2
     -------------------------------------------------------------
     evaluateDyadic(or, boolV(BOOL1), boolV(BOOL2)) → boolV(BOOL)

[eq1] evaluateDyadic(eq, VAL, @VAL) → boolV(True)
[eq1] VAL =/= VAL' ⇒ evaluateDyadic(eq, VAL, VAL') → boolV(False)

[neq] evaluateDyadic(eq, VAL1, VAL2) → VAL,
      evaluateMonadic(not, VAL) → VAL'
      ---------------------------------------
      evaluateDyadic(neq, VAL1, VAL2) → VAL'

[lt1] evaluateDyadic(lt, boolV(False), boolV(True)) → boolV(True)
[lt2] evaluateDyadic(lt, boolV(False), boolV(False)) → boolV(False)
[lt3] evaluateDyadic(lt, boolV(True), boolV(True)) → boolV(False)

[lt4] ? BOOL ← INT1 < INT2
      -------------------------------------------------------
      evaluateDyadic(lt, intV(INT1), intV(INT2)) → boolV(BOOL)

[gt1] evaluateDyadic(gt, boolV(False), boolV(True)) → boolV(False)
[gt2] evaluateDyadic(gt, boolV(False), boolV(False)) → boolV(False)
[gt3] evaluateDyadic(gt, boolV(True), boolV(True)) → boolV(True)

[gt4] ? BOOL ← INT1 > INT2
      -------------------------------------------------------
      evaluateDyadic(gt, intV(INT1), intV(INT2)) → boolV(BOOL)

[ge] evaluateDyadic(lt, VAL1, VAL2) → VAL,
     evaluateMonadic(not, VAL) → VAL'
     ---------------------------------------
     evaluateDyadic(ge, VAL1, VAL2) → VAL'
```

```
[le] evaluateDyadic(gt, VAL₁, VAL₂) → VAL,
     evaluateMonadic(not, VAL) → VAL'
     -------------------------------------
     evaluateDyadic(le, VAL₁, VAL₂) → VAL'
```

lcs/adts/memory.ir

```
% memories binding identifiers to values
MEM = <ID, VAL>*
% return the empty memory
init: → MEM
[init] init → []
% update the memory
update: MEM × ID × VAL → MEM
[update1] update([], ID, VAL) → [<ID, VAL>]
[update2] update([<ID, _VAL>|MEM], @ID, VAL) → [<ID, VAL>|MEM]
[update3] ID ≠ ID',
          update(MEM, ID', VAL') → MEM'
          ----------------------------------------------------
          update([<ID, VAL>|MEM], ID', VAL') → [<ID, VAL>|MEM']
% lookup the memory
lookup: MEM × ID → VAL
[lookup] lookup(_MEM ++ [<ID, VAL>] ++ _MEM, @ID) → VAL
```

D.2 The divide-and-conquer schema

The program transformation below can be regarded as a representation of the divide-and-conquer schema; refer also to Subsection 4.4.3.

$\lambda\langle r\rangle$: Symbol*.
$\lambda\langle \mathit{isMinimal}, \mathit{solve}, \mathit{isNonminimal}, \mathit{decompose}, \mathit{compose}\rangle$: Symbol*.
$\lambda\langle x, y, z\rangle$: Sort*.
Let $x_0 =$ **New Variable Of Sort** x **In**
Let $y_0 =$ **New Variable Of Sort** y **In**
Let $x_1 =$ **New Variable Of Sort** x **In**
Let $y_1 =$ **New Variable Of Sort** y **In**
Let $x_2 =$ **New Variable Of Sort** x **In**
Let $y_2 =$ **New Variable Of Sort** y **In**
Let $z_0 =$ **New Variable Of Sort** z **In** $\langle$
  Rule From [minimal] $r\ \langle \mathsf{x}\rangle \to \langle \mathsf{y}\rangle \Leftarrow \langle\ \mathit{isMinimal}\ \langle \mathsf{x}\rangle \to \langle\rangle, \mathit{solve}\ \langle \mathsf{x}\rangle \to \langle \mathsf{y}\rangle\ \rangle$,
  Rule From [nonminimal] $r\ \langle \mathsf{x}\rangle \to \langle \mathsf{y}\rangle \Leftarrow \langle\ \ \mathit{isNonminimal}\ \langle \mathsf{x}\rangle \to \langle\rangle$,
    $\mathit{decompose}\ \langle \mathsf{x}\rangle \to \langle z, x_1, x_2\rangle$,
    $r\ \langle \mathsf{x}_1\rangle \to \langle \mathsf{y}_1\rangle$,
    $r\ \langle \mathsf{x}_2\rangle \to \langle \mathsf{y}_2\rangle$,
    $\mathit{compose}\ \langle \mathsf{z}, y_1, y_2\rangle \to \langle y\rangle\ \rangle$
  $\rangle$

D.3 Symbol tables in a block-structured language

The following transformation describes the accumulation of a symbol table in the declaration part, the pervasive inheritance in the statement part and its proper initialization for a block-structured language. Thus, it generalizes Example 4.2.2.

```
λ sk : Skeleton .
  Default For ST By &static initST
∘ Left To Right ST
∘ (Let write = From {block} To {declaration} In sk In
    Let read = From {block} To {declaration, expression} In sk In
     Ensure (
      Positions Output For (write ∪ {declaration}) \ {block} Of Sort ST ++
      Positions Input For read ∪ {block, declaration, expression} Of Sort ST
     )
  )
```

The transformation is based on the following assumptions: The nonterminal *expression* models expressions, whereas *declaration* models declarations. Both, program blocks and any other kind of nested blocks (e.g. as a part of a function or a procedure declaration), are modelled by the nonterminal *block*. A block consists of a declaration and a statement part. All symbols on paths between *block* and *declaration* / *expression* including the symbols *block*, *declaration* and *expression* need at least reading access to the symbol table (**Positions Input**). Since, the symbol table is accumulated in the declarations part, some more symbols need writing process as well (**Positions Output**). We have to take care that *block* does not synthesize a symbol table ($\ldots \setminus \{block\}$) because the symbol table entries of a nested block should not be visible in the enclosing block.

D.4 The *Constituents* ... *With* ... construct

We derive a schema useful to establish a computational behaviour simulating the *Constituents* ... *With* ... construct for remote access discussed in Subsection 3.4.2. The profile of the schema takes the following form:

$$\textbf{Constituents } \_.\_ \textbf{ With } \_, \_, \_, \_ \textbf{ For } \_ \textbf{ In } \_ \;:$$
$$\mathsf{Symbol} \times \mathsf{Sort} \times \mathsf{Sort} \times \mathsf{Symbol} \times \mathsf{Symbol} \times \mathsf{Symbol} \times \mathsf{Symbol} \times \mathcal{P}(\mathsf{Tag}) \rightarrow \mathsf{Trafo}$$

Consider the following application of the schema:

$$\textbf{Constituents } s.\sigma \textbf{ With } \sigma', union, unit, zero \textbf{ For } for \textbf{ In } in,$$

The schema can be subdivided into several transformations to be performed subsequently:

1. For any occurrence of a parameter p of sort σ on an output position of s a computation $unit(p) \rightarrow (p')$, where p' is a fresh variable of sort σ', will be included.
2. All symbols on paths between *for* and (including) the symbols defined by rules using s get attached an output position of sort σ'.

3. All unused occurrences of parameters p_1, …, p_n of sort σ' have to be composed by a sequence of computations

$$union(p_1, p_2) \to p'_1, \ldots, union(p'_{n-2}, p_n) \to p'_{n-1}$$

where p'_1, …, p'_{n-1} are fresh variables of sort σ'.
4. The added output positions (refer to step (2.)) are defined by taking the most recent definition from the left.
5. The remaining undefined occurrences p of sort σ' are defined by inserting a computation *zero* $\to$ p assuming *zero* as a left unit of *union*.

The actual specification is the following:

```
λ rsym : Symbol .
λ rsort : Sort .
λ aux : Sort .
λ union : Symbol .
λ unit : Symbol .
λ zero : Symbol .
λ for : Symbol .
λ in : 𝒫(Tag) .
λ rs : Rules .
 Let sk = Skeleton Of rs In
  Let cl = (From {for} To {rsym} In sk) ∪ {for} In
% 5. insert constant computations
    Default For aux By zero
% 4. copy accumulated value to the LHS
  ∘ From The Left aux
% 3. combine defining occurrences
  ∘ Forgetting Tags in Do
    Reduce aux By union
% 2. add positions for synthesis
  ∘ Add Positions Output For cl Of Sort aux
% 1. add unary computations
  ∘ Selecting Symbols cl Do
    Hiding unit Do (
        Add ⟨⟨Output, unit, aux⟩⟩
      ∘ Use ⟨Output, rsym, rsort⟩ By unit
     )
  On rs.
```

To give an example, we start with a GSF schema for a part of an imperative language with terminal attribution for identifiers. For simplicity, no other attribution is considered here.

```
[prog]    program : declarations, statements.
[decs]  declarations : declaration, declarations.
[nodec] declarations : .
[dec]   declaration : type, identifier → ID.
```

```
[int]   type : .
[bool]  type : .
[concat] statements : statement, statements.
[skip]   statements : .
[assign]  statement : variable, expression.
[var]     expression : variable.
[varid]   variable : identifier → ID.
```

Similarly to the example in Subsection 3.4.2, all occurrences of identifiers should be accumulated separately for the declaration and the statement part. This is modelled by the following application of the operator **Constituents**:

Let t = λ s : **Symbol** .
 Constituents identifier.ID
  With IDS, &ids union, &ids unit, &ids zero
  For s
  In {[prog]}
In (t **On** declarations) ∘ (t **On** statements).

The result of the transformation is the following:

```
[prog] program
        :
          declarations → IDS_17,
          statements → IDS_3.
[decs] declarations → IDS_21
        :
          declaration → IDS_15,
          declarations → IDS_19,
          &ids union(IDS_15, IDS_19) → IDS_21.
[nodec] declarations → IDS_20
         :
           &ids zero → IDS_20.
[concat] statements → IDS_12
          :
            statement → IDS_1,
            statements → IDS_5,
            &ids union(IDS_1, IDS_5) → IDS_12.
[skip] statements → IDS_6
        :
          &ids zero → IDS_6.
[dec] declaration → IDS_14
       :
         type,
         identifier → ID,
         &ids unit(ID) → IDS_14.
[assign] statement → IDS_13
          :
            variable → IDS_7,
            expression → IDS_10,
            &ids union(IDS_7, IDS_10) → IDS_13.
```

```
[int] type : .
[bool] type : .
[varid] variable → IDS_0
         :
           identifier → ID,
           &ids unit(ID) → IDS_0.
[var] expression → IDS_8
        :
          variable → IDS_8.
```

The rule [prog] could be extended to make use of the derived sets of identifiers, e.g. to check that all declared identifiers are also used.

D.5 Elimination of tail recursion

We demonstrate a simple elimination procedure for tail recursion. It is assumed that the rules for the dedicated symbol *sym* describe a traversal of a data structure of certain sort *sort*. Tail-recursive calls of *sym* are then eliminated by returning the parameter of sort as a new output of the conclusion.

Consider, for example, the following inference rules of a big step semantics. There are tail calls in [while], [then] and [else].

```
[assign] evaluate(EXP, MEM) → VAL,
         update(MEM, ID, VAL) → MEM'
         -------------------------------------
         execute(assign(ID, EXP), MEM) → MEM'
% first evaluate condition, then branch on value
[if] evaluate(EXP, MEM) → VAL,
     cond(VAL, STM_1, STM_2, MEM) → MEM'
     ----------------------------------------
     execute(if(EXP, STM_1, STM_2), MEM) → MEM'
[while] concat(STM, while(EXP, STM)) = STM',
        if(EXP, STM', skip) = STM'',
        execute(STM'', MEM) → MEM'
        -------------------------------------
        execute(while(EXP, STM), MEM) → MEM'
% execute the Then-path of an If-statement
[then] execute(STM, MEM) → MEM'
       ----------------------------------------
       cond(boolV(True), STM, _STM, MEM) → MEM'
% execute the Else-path of an If-statement
[else] execute(STM, MEM) → MEM'
       -----------------------------------------
       cond(boolV(False), _STM, STM, MEM) → MEM'
```

The following variant is the result of the transformation eliminating the tail calls. The parameter *skip* is used for rules without tail-recursive calls. Concerning styles of semantics definition, we moved from a big step semantics (natural semantics) to a small step semantics (transitional semantics).

```
[assign] evaluate(EXP, MEM) → VAL,
         update(MEM, ID, VAL) → MEM'
         ---------------------------
         execute(assign(ID, EXP), MEM) → (MEM', skip)
[if] evaluate(EXP, MEM) → VAL,
     cond(VAL, STM1, STM2, MEM) → (MEM', STM5)
     ----------------------------------------
     execute(if(EXP, STM1, STM2), MEM) → (MEM', STM5)
[while] concat(STM, while(EXP, STM)) = STM',
        if(EXP, STM', skip) = STM''
        ---------------------------------
        execute(while(EXP, STM), MEM) → (MEM, STM'')
[then] cond(boolV(True\BOOL), STM, STM3, MEM) → (MEM, STM)
[else] cond(boolV(False\BOOL), STM4, STM, MEM) → (MEM, STM)
```

There are certain assumptions for the elimination procedure which are skipped here. The definition of the transformation is omitted here as well because of its extent.

D.6 Establishing CPS

We demonstrate a transformation which is suitable to establish the continuation passing style for the rules of a dedicated symbol *sym*. The continuations are of a certain sort *sort*. There is a functor *skip* for the empty continuation and another functor *concat* for the combination of continuations. To illustrate this transformation, we continue the example of the previous section by transforming the transitional semantics into a semantics in the continuation passing style.

```
[assign] evaluate(EXP, MEM) → VAL,
         update(MEM, ID, VAL) → MEM'
         ---------------------------
         execute(assign(ID, EXP), MEM, STM6) → (MEM', STM6)
[if] evaluate(EXP, MEM) → VAL,
     cond(VAL, STM1, STM2, MEM) → (MEM', STM5)
     ----------------------------------------
     execute(if(EXP, STM1, STM2), MEM, STM7) → (MEM', concat(STM5, STM7))
[while] concat(STM, while(EXP, STM)) = STM',
        if(EXP, STM', skip) = STM''
        ---------------------------------
        execute(while(EXP, STM), MEM, STM8) → (MEM, concat(STM'', STM8))
[then] cond(boolV(True\BOOL), STM, STM3, MEM) → (MEM, STM)
[else] cond(boolV(False\BOOL), STM4, STM, MEM) → (MEM, STM)
```

There are certain assumptions for the applicability of the transformation which are skipped here. The definition of the transformation is omitted here as well because of its extent.

D.7 Coupling

Modular specifications consisting of several phases implicitly describe certain central data structures more than once. In a language processor, for example, the abstract syntax is described twice, once by the "frontend" performing AST construction among other subtask and once by the dynamic semantics definition performing essentially a traversal of the given AST.

In this section we want to introduce a transformation which scans a given specification for term constructors and then tries to match the skeleton of another specification with these constructors. The justification for such a matching process is the well-known correspondence between signatures (term constructors) and context-free grammars (skeletons).

Let us first consider an example. The following rule defines the static semantics of assignment statement of a simple imperative language.

```
[assign] statement(ST)
          :
            id → ID,
            &static isVar(ST, ID) → T,
            expression(ST) → T',
            &static assignable(T, T').
```

The following rule defines the dynamic semantics of the assignment statements. The abstract syntax is implicitly covered, since a semantics definition is essentially a traversal of the abstract syntax.

```
execute: STM × MEM → MEM
evaluate: EXP × MEM → VAL
[assign] evaluate(EXP, MEM) → VAL,
         update(MEM, ID, VAL) → MEM'
         ------------------------------------
         execute(assign(ID, EXP), MEM) → MEM'
```

To couple the two phases, the above GSF rules should construct terms according to the term constructors in the semantics definition. We first show the result we are interested in.

```
[assign] statement(ST) → STM0
          :
            id → ID,
            &static isVar(ST, ID) → T,
            expression(ST) → (T', EXP0),
            &static assignable(T, T'),
            &ast assign(ID, EXP0) → STM0.
```

The result can be obtained from the previous GSF rule which does not cover AST construction by the transformation below. The meta-program takes two specifications as input. Furthermore, a number of nonterminals which do not contribute to abstract syntax can be enumerated. The transformation first looks for associations of skeleton symbols and sorts by matching the constructor profiles with skeleton rules based on conformance of constructor symbol and tag. The accumulated associations are then used in a simple application of the operator **Relate** for adding compositional parameterization to construct (as in the example) or deconstruct terms accordingly.

lcs/transformers/ast.fra

```
% add an association preserving mapping condition
Let extendMap = λ as : 𝒫(Association) . λ sym : Symbol . λ sort : Sort .
 Let sofar =
  Fold Left
   λ sort0 : Sort? . λ ⟨sym1, sort1⟩ : Association .
    sym = sym1 → sort1, sort0
  Neutral ? List as
 In sofar = ? → as ++ ⟨⟨sym, sort⟩⟩, sofar = sort ∘→ as
In
λ at : Rules . λ io : Io . λ ignore : 𝒫(Symbol) . λ pfx : Prefix .
λ sk : Skeleton .
% accumulate all constructors (auxiliary function)
 Let constructors = constructorsInRules On at In
  Let associations =
% iterate signature with constructors
   Fold Left
    λ as : 𝒫(Association) . λ prof : Profile .
     ¬ Symbol Of prof ∈ constructors →
      as,
      Let tag = Tag From Symbol Of prof In
       Let maybe =
        Fold Left λ maybe : (Name × Name*)? . λ ⟨t, l, r⟩ : Shape .
         t = tag → ⟨l, r⟩, maybe
        Neutral ? List sk
       In
        maybe = ? →
% no matching rule found for current constructor profile
         as,
         Let ⟨sym, syms⟩ = maybe In
          sym ∈ ignore →
% LHS symbol to be ignored ⇒ ignore rule altogether
           as,
           Let ⟨target⟩ = Sorts Output Of prof In
            Let ⟨sorts, as⟩ =
% match LHS / RHS of rule with target / source of constructor
             Fold Left
              λ ⟨rest, sofar⟩ : ⟨Sort*, 𝒫(Association)⟩ . λ sym : Name .
               sym ∈ ignore →
                ⟨rest, sofar⟩,
                ⟨Tail Of rest, extendMap On sofar On sym On Head Of rest⟩
             Neutral ⟨Sorts Input Of prof, (extendMap On as On sym On target)⟩
             List syms
            In sorts = ⟨ ⟩ ∘→ as
   Neutral ∅ List Sigma Of at
  In
% attach compositional computational behaviour
   Selecting Symbols Symbols Associated In associations
    Do Relate io associations pfx
```

Bibliography

[ABFQ92] Francis Alexandre, Khadel Bsaies, Jean Pierre Finance, and Alain Quere. Spes: A System for Logic Program Transformation. In A. Voronkov, editor, *Logic Programming and Automated Reasoning, LPAR'92*, volume 624 of *LNCS*, pages 445–447. Springer-Verlag, 1992.

[AC90] Isabelle Attali and Jacques Chazarain. Functional evaluation of strongly non-circular typol specifications. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *LNCS*, pages 157–176. Springer-Verlag, September 1990. Paris.

[ACG92] Isabelle Attali, Jacques Chazarain, and Serge Gilette. Incremental Evaluation of Natural Semantics Specifications. In M. Bruynooghe and M.Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 87–99. Springer-Verlag, New York–Heidelberg–Berlin, August 1992.

[Ada91] Stephen Robert Adams. *Modular Grammars for Programming Language Prototyping.* PhD thesis, University of Southampton, Faculty of Engineering, Department of Electronics and Computer Science, March 1991.

[AFZ88] I. Attali and P. Franchi-Zannettacci. Unification-free execution of TYPOL programs by Semantic Attribute Evaluation. In *Fifth International Conference Symposium on Logic Programming, Seattle*, pages 166–177. Cambridge MIT Press, August 1988.

[Alb91] H. Alblas. Introduction to attribute grammars. In Alblas and Melichar [AM91], pages 1–15.

[AM91] Henk Alblas and Bořivoj Melichar, editors. *Attribute grammars, Applications and Systems, Proceedings of the In ternational Summer School SAGA, Prague, Czechoslovakia*, volume 545 of *LNCS*. Springer-Verlag, June 1991.

[AP91] Martín Abadi and Gordon D. Plotkin. A logical view of composition and refinement. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 323–332, Orlando, Florida, January 1991.

[AP94] I. Attali and D. Parigot. Integrating natural semantics and attribute grammars: the minotaur system. Research Report no. 2339, Inria, September 1994.

[APR97] Isabelle Attali, Valrie Pascual, and Christophe Roudet. A language and an integrated environment for program transformations. Rapport de recherche 3313, INRIA, December 1997.

[Att89] I. Attali. Compiling TYPOL with Attribute Grammars. In Deransart et al. [DLM89], pages 252–272.

[Bau98] Beate Baum. *Modularisierung attributierter Grammatiken*. PhD thesis, Department of Comp. Sc., University Rostock, 1998.

[BCD$^+$88] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of SIGSOFT'88, Boston, USA*, 1988.

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BdM97] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[BL92] Gilad Bracha and Gary Lindstrom. Modularity Meets Inheritance. In *Proceedings of the IEEE International Conference on Computer Languages*, April 1992.

[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.

[Boy96a] Johan Boye. *Directional Types in Logic Programming*. PhD thesis, University of Linköping, 1996.

[Boy96b] John Tang Boyland. *Descriptional Composition of Compiler Components*. PhD thesis, University of California, Berkeley, September 1996. Available as technical report UCB//CSD-96-916.

[Boy98] John Tang Boyland. Analyzing Direct Non-local Dependencies. In Kai Koskimies, editor, *Compiler Construction, 7th International Conference, CC'98*, volume 1383 of *LNCS*, pages 31–49. Springer-Verlag, April 1998.

[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.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, The University of Utah, Department of Computer Science, March 1992.

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

[Bru95] J.J. Brunekreef. Translog, an Interactive Tool for Transformation of Logic Programs. Technical Report P9512, University of Amsterdam, Programming Research Group, December 1995.

[BS98] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer-Verlag, 1998.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, April 1998.

[CD84] Bruno Courcelle and Pierre Deransart. Proofs of Partial Correctness for Attribute Grammars with Application to Recursive Procedures and Logic Programming. Technical Report RR 332, INRIA Rocquencourt, 1984.

[CDPR98] L. Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Symbolic composition. Technical Report 3348, INRIA, January 1998.

[CF94] Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software: International Symposium*, volume 789, pages 244–272. Springer-Verlag, April 1994.

[CFZ82a] Bruno Courcelle and Paul Franchi-Zannettacci. Attribute Grammars and Recursive Program Schemes I. *Theoretical Computer Science*, 17(2):163–191, February 1982.

[CFZ82b] Bruno Courcelle and Paul Franchi-Zannettacci. Attribute Grammars and Recursive Program Schemes II. *Theoretical Computer Science*, 17(3):235–257, March 1982.

[CI84] Robert D. Cameron and M. Robert Ito. Grammar-Based Definition of Metaprogramming Systems. *ACM Transactions on Programming Languages and Systems*, 6(1):20–54, 1984.

[Coa95] P. Coad. *Object Models: Strategies, Patterns and Applications.* Prentice Hall, 1995.

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

[Des88] T. Despeyroux. Typol: A formalism to implement natural semantics. Technical report 94, INRIA, March 1988.

[Dev90] Y. Deville. *Logic Programming: Systematic Program Development.* Addison Wesley, 1990.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall International, 1976.

[DL94] Yves Deville and Kung-Kiu Lau. Logic Program Synthesis. *The Journal of Logic Programming 19*, pages 321–350, 1994.

[DLM89] P. Deransart, B. Lorho, and J. Maluszyński, editors. *Programming Languages Implementation and Logic Programming, Proceedings of the International Workshop PLILP '88, Orleans, France*, number 348 in LNCS. Springer-Verlag, May 1989.

[DM85] Pierre Deransart and Jan Małuszyński. Relating Logic Programs and Attribute Grammars. *Journal of Logic Programming*, 2(2):119–155, 1985.

[DM93] Pierre Deransart and Jan Maluszyński. *A Grammatical View of Logic Programming.* The MIT Press, 1993.

[DPRJ96] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Attribute grammars and folds: Generic control operators. Rapport de recherche 2957, INRIA, August 1996.

[DPRJ97] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Structure-directed genericity in functional programming and attribute grammars. Rapport de Recherche 3105, INRIA, February 1997.

[DS96] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, Philadelphia, Pennsylvania, 24–26 May 1996.

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

[FFG91] Limor Fix, Nissim Francez, and Orna Grumberg. Program composition and modular verification. In Javier Leach Albert, Burkhard Monien, and Mario Rodríguez-Artalejo, editors, *Automata, Languages and Programming, 18th International Colloquium*, volume 510 of *LNCS*, pages 93–114, Madrid, Spain, 8–12 July 1991. Springer-Verlag.

[FLO97] P. Flener, K.-K. Lau, and M. Ornaghi. On Correct Program Schemas. In Fuchs [Fuc97]. Report CW 253, Katholieke Universiteit Leuven, Department Of Computing Science.

[FMY92] R. Farrow, T.J. Marlowe, and D.M. Yellin. Composable Attribute Grammars. In *Proceedings of 19th ACM Symposium on Principles of Programming Languages (Albuquerque, NM)*, pages 223–234, January 1992.

[Fuc97] Norbert E. Fuchs, editor. *Proceedings LOPSTR'97, Leuven, Belgium. July 10–12, 1997*, 1997. Report CW 253, Katholieke Universiteit Leuven, Department Of Computing Science.

[Gal97] John (John P.) Gallagher, editor. *Logic program synthesis and transformation: 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 28–30, 1996: proceedings*, volume 1207 of *LNCS*, New York, NY, USA, 1997. Springer-Verlag Inc.

[Gan83] Harald Ganzinger. Increasing Modularity and Language Independency in Automatically Generated Compilers, 1983.

[GE90] Josef Grosch and Helmut Emmelmann. A Tool Box for Compiler Construction. In *Proceedings of CC'90*, 1990.

[GHL$^+$92] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM 35*, pages 121–131, February 1992.

[Gie88] R. Giegerich. Composition and Evaluation of Attribute Coupled Grammars. *Acta Informatica 25*, pages 355–423, 1988.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Har97] Jörg Harm. Automatic Test Program Generation from Formal Language Spe cifications. In RIB [RIB97]. 24 pages, to appear.

[Has97] Haskell 1.4—A Non-strict, Purely Functional Language, April 1997. Yale University, University of St. Andrews.

[Hed89] Görel Hedin. An object-oriented notation for attribute grammars. In S. Cook, editor, *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP'89)*, BCS Workshop Series, pages 329–345. Cambridge University Press, July 1989.

[Hed91] Görel Hedin. Incremental static-semantics analysis for object-oriented languages using door attribute grammars. In Alblas and Melichar [AM91], pages 374–379.

[Hed92] Görel Hedin. *Incremental Semantic Analysis.* Ph.D. thesis, Lund University, Lund, Sweden, 1992. LUTEDX/(TECS-1003)/1-276/(1992).

[Hed94] Görel Hedin. An Overview of Door Attribute Grammars. In P.A. Fritzson, editor, *Proceedings of Compiler Construction CC'94, 5th International Conference, CC'94, Edinburgh, U.K.*, number 786 in LNCS, pages 31–51, 1994.

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

[HL89] Ivo Van Horebeek and Johan Lewi. *Algebraic Specifications in Software Engineering.* Springer-Verlag, 1989.

[HL94] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

[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.

[HN95] A. Hamfelt and J.F. Nilsson. Towards a Logic Programming Methodology based on Higher-order Predicates. 23 pages, 1995.

[HN96] A. Hamfelt and J.F. Nilsson. Declarative Logic Programming with Primitive Recursive Relations on Lists. In P. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming, MIT Press*, pages 230–243, 1996.

[Hud96] Paul Hudak. Building Domain-Specific Embedded Languages, December 1996.

[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.

[JC94] C.B. Jay and J.R.B. Cockett. Shapely types and shape polymorphism. In Donald Sannella, editor, *Proceedings Programming Languages and Systems-ESOP'94*, volume 788 of *LNCS*, pages 302–316. Springer-Verlag, 1994.

[JD93] Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Yale University, December 1993.

[Jeu95] J. Jeuring. Polytypic pattern matching. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 238–248, 1995.

[JF85] Gregory F. Johnson and Charles N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 141–151, New Orleans, Louisiana, January 1985.

[JJ96] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming, Second International School*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.

[JJ97] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[JKS94] Ashish Jain, Marc Kirschenbaum, and Leon Sterling. Constructing provably correct logic programs. Technical Report CES-94-04, Department of Computer Engineering and Science, Case Western Reserve University, March 1994.

[JP90] Martin Jourdan and Didier Parigot. Application Development with the FNC-2 Attribute Grammar System. In Dieter Hammer and Michael Albinus, editors, *Compiler Compilers '90*, volume 477 of *LNCS*, pages 11–25. Springer-Verlag, Schwerin, 1990.

[JP91] Martin Jourdan and Didier Parigot. Internals and Externals of the FNC-2 Attribute Grammar System. In Alblas and Melichar [AM91], pages 485–504.

[JPJ$^+$90] Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Conf. on Programming Languages Design and Implementation*, pages 209–222, White Plains, NY, June 1990. Published as *ACM SIGPLAN Notices*, 25(6).

[JRG92] Ian Jacobs and Laurence Rideau-Gallot. A Centaur Tutorial. Rapport de recherche 2881, INRIA Sophia-Antipolis, July 1992.

[JS94] Ashish Jain and Leon Sterling. A methodology for program construction by stepwise structural enhancement. Technical Report CES-94-10, Department of Computer Engineering and Science, Case Western Reserve University, June 1994.

[Kah87] Gilles Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39, Passau, Germany, 19–21 February 1987. Springer-Verlag.

[Kan91] Max I. Kanovich. Efficient program synthesis: Semantics, logic, complexity. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526, pages 615–632. Springer-Verlag, September 1991.

[Kas76] Uwe Kastens. Ein Übersetzer-erzeugendes System auf der Basis Attributierter Grammatiken. interner Bericht 10, Fakultät für Informatik, University Karlsruhe, September 1976.

[Kas91] Uwe Kastens. Attribute Grammars in a Compiler Construction Environment. In Alblas and Melichar [AM91], pages 380–400.

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

[Kli94] Paul Klint. Writing meta-level specifications in ASF+SDF. Draft, November 1994.

[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 *LNCS*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer-Verlag.

[KLMM93] J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, and B. Magnusson, editors. *Object-Oriented Environments: The Mjølner Approach.* Prentice Hall, 1993.

[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.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2:127–145, 1968. Corrections in 5:95-96, 1971.

[Kos91] Kai Koskimies. Object Orientation in Attribute Grammars. In Alblas and Melichar [AM91], pages 297–329.

[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.

[KT93] J. Komorowski and S. Trcek. Towards Refinement of Definite Logic Programs. In *ERCIM Workshop on Development and Transformation of Logic Programs, France*, 1993.

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

[KW96] Basim M. Kadhim and William M. Waite. Maptool — Supporting Modular Syntax Development. In Tibor Gyimóthy, editor, *Compiler Construction, 6th International Conference, CC'96*, volume 1060 of *LNCS*, pages 268–280. Springer-Verlag, April 1996.

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

[Läm97] Ralf Lämmel. Composition based on Meta-Programming. In Antonio Brogi and Patricia Hill, editors, *Proceedings of LOCOS'97, LOGIC-BASED COMPOSITION OF SOFTWARE, Post Conference Workshop for the International Conference on Logic Programming, Leuven, Belgium, July 8-11th, 1997*, pages 49–58, July 1997.

[Lar97] Craig Larman. *Applying UML and Patterns.* Prentice Hall, 1997.

[Le 89] Carole Le Bellec. Spécification de règles sémantiques manquantes. rapport de DEA, Dépt. d'Informatique, University d'Orléans, September 1989.

[Le 93] Carole Le Bellec. *La généricité et les grammaires attribuées.* PhD thesis, Dépt. d'Informatique, University d'Orléans, 1993.

[LEW96] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types.* Wiley and Teubner, 1996.

[LH96] S. Liang and P. Hudak. Modular Denotational Semantics for Compiler Construction. In Nielson [Nie96], pages 219–234.

[LHJ95] Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995.

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

[LJPR93] Carole Le Bellec, Martin Jourdan, Didier Parigot, and Gilles Roussel. Specification and Implementation of Grammar Coupling Using Attribute Grammars. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '93)*, volume 714 of *LNCS*, pages 123–136, Tallinn, August 1993. Springer-Verlag.

[LNC91] Algebraic Methods II: Theory, Tools and Applications. In Jan A. Bergstra and Loe M.G. Feijs, editors, *Algebraic Methods II: Theory, Tools and Applications*, volume 490. Springer-Verlag, 1991.

[Lor77] Bernard Lorho. Semantic attributes processing in the system DELTA. In A. Ershov and Cornelius H. A. Koster., editors, *Methods of Algorithmic Language Implementation*, volume 47 of *LNCS*, pages 21–40. Springer-Verlag, 1977.

[LR96] Ralf Lämmel and Günter Riedewald. A calculus for modular and extensible language definition. April 1996. Proceedings (Technical Report) of ALEL Workshop at CC'96, Linköping, Sweden, April 26, 1996.

[LR97] Ralf Lämmel and Günter Riedewald. Operations on fragments of formal language definitions towards semantic extensibility. In RIB [RIB97]. 19 pages.

[LRBS] Ralf Lämmel, Günter Riedewald, Nguyen Van Bac, and Susanne Stasch. A language construction set. in preparation.

[LRH96] Ralf Lämmel, Günter Riedewald, and Jörg Harm. Specification formalisms in $^{\Lambda}\Delta_{\Lambda}$. Preprint CS-08-96, University of Rostock, Department of Computer Science, December 1996. 100 pages.

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

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[Mos83] Peter D. Mosses. Abstract semantic algebras! In *Formal Description of Programming Concepts II, Proc. IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982*, pages 45–71. North-Holland, 1983.

[Mos88] Peter D. Mosses. Action semantics. *Cubus*, 1(4):9–13, 1988. Published by Dansk Datamatik Center, Lyngby, Denmark.

[Mos92] Peter D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[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 *LNCS*, pages 37–61. Springer-Verlag, 1996.

[Mos97] Peter D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In *TAPSOFT'97*, volume 1214. Springer-Verlag, 1997.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Nai96] Lee Naish. Higher Order Logic Programming in Prolog. In *Proc. Workshop on Multi-Paradigm Logic Programming, JICSLP'96, Bonn*, 1996.

[NH95] J.F. Nilsson and A. Hamfelt. Constructing Logic Programs with Higher Order Predicates. In M. Alpuente and M. Sessa, editors, *Proceedings of GULP-PRODE'95, the Joint Conference on Declarative Programming 1995, Universita' Degli Studi di Salerno, Salerno*, pages 307–312, 1995.

[Nie96] Hanne Riis Nielson, editor. *6th European Symposium on Programming, Linköping, Sweden, April 1996, Proceedings of ESOP'96*, volume 1058. Springer-Verlag, April 1996.

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

[NN92] F. Nielson and H. R. Nielson. *Two-Level Functional Languages.* Cambridge Tracts in Theoretical Computer Science **vol. 34**. Cambridge University Press, 1992.

[NS97] Lee Naish and Leon Sterling. A Higher Order Reconstruction of Stepwise Enhancement. In Fuchs [Fuc97]. Report CW 253, Katholieke Universiteit Leuven, Department Of Computing Science.

[Paa91] Jukka Paakki. *Paradigms for Attribute-grammar-based Language Implementation.* Ph.D. thesis, Department of Comp. Sc., University of Helsinki, February 1991.

[Paa95] Jukka Paakki. Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[Par88] Didier Parigot. *Transformations, Évaluation Incrmentale et Optimisations des Grammaires Attribus: Le Systme FNC-2.* PhD thesis, Universit de Paris-Sud, Orsay, 1988.

[Pet94] Mikael Pettersson. RML – a new language and implementation for natural semantics. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming, PLILP'94*, volume 844 of *LNCS*, pages 117–131. Springer-Verlag, 1994.

[Pet95] Mikael Pettersson. *Compiling Natural Semantics.* PhD thesis, Department of Computer and Information Science, Linköping University, December 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 Nielson [Nie96], pages 280–295.

[PRJD96a] Didier Parigot, Gilles Roussel, Martin Jourdan, and Etienne Duris. Dynamic Attribute Grammars. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140 of *LNCS*, pages 122–136, Aachen, September 1996. Springer-Verlag.

[PRJD96b] Didier Parigot, Gilles Roussel, Martin Jourdan, and Etienne Duris. Dynamic Attribute Grammars. Rapport de recherche 2881, INRIA, May 1996. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-2881.ps.gz`.

[Pro96] Maurizio Proietti, editor. *Logic program synthesis and transformation: 5th International Workshop, LOPSTR'95, Utrecht, The Netherlands, September 20–22, 1995: proceedings*, volume 1048 of *LNCS*, New York, NY, USA, 1996. Springer-Verlag Inc.

[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.

[RIB97] *Rostocker Informatik-Berichte*, volume 20. Universität Rostock, 1997.

[Rie72] Günter Riedewald. *Syntaktische Analyse von ALGOL68-Programmen.* Dissertation A, Universität Rostock, Sektion Mathematik, 1972.

[Rie79] Günter Riedewald. *Compilerkonstruktion und Grammatiken syntaktischer Funktionen.* Dissertation B, Rechenzentrum der Universität Rostock, 1979.

[Rie91] Günter Riedewald. Prototyping by Using an Attribute Grammar as a Logic Program. In Alblas and Melichar [AM91], pages 401–437.

[Rie92] Günter Riedewald. The LDL – Language Development Laboratory. In U. Kastens and P. Pfahler, editors, *Compiler Construction, 4th International Conference, CC'92, Paderborn, Germany*, number 641 in LNCS, pages 88–94, October 1992.

[RL89] Günter Riedewald and Uwe Lämmel. Using an attribute grammar as a logic program. In Deransart et al. [DLM89], pages 161–179.

[RL93] Günter Riedewald and Ralf Lämmel. Provable correctness of prototype interpreters in LDL. Preprint CS-09-93, University of Rostock, Department of Computer Science, 1993.

[RMD83] G. Riedewald, J. Maluszyński, and P. Dembinski. *Formale Beschreibung von Programmiersprachen, Eine Einführung in die Semantik.* Oldenbourg-Verlag, München, Wien and Akademie-Verlag, Berlin, 1983.

[Rou94] Gilles Roussel. *Algorithmes de base pour la modularit et la rutilisabilit des grammaires attribues.* PhD thesis, Département d'Informatique, Université de Paris 6, March 1994.

[RPJ94] Gilles Roussel, Didier Parigot, and Martin Jourdan. Coupling Evaluators for Attribute Coupled Grammars. In Peter A. Fritzson, editor, *5th Int. Conf. on Compiler Construction (CC' 94)*, volume 786 of *LNCS*, pages 52–67, Edinburgh, April 1994. Springer-Verlag.

[SA97] Wolfram Schulte and Klaus Achatz. Functional Object-oriented Programming with Object-Gofer. In Herbert Kuchen, editor, *Proceedings Arbeitstagung Programmiersprachen, Aachen, 22.-23. September, 1997, GI-Jahrestagung'97, 12 pages*, September 1997. 9 pages.

[SHLG94] Viggo Stoltenberg-Hansen, Ingrid Lindström, and Edward R. Griffor. *Mathematical Theory of Domains.* Number 22 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.

[SJ94] Guy L. Steele Jr. Building interpreters by composing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 472–492, Portland, Oregon, January 1994.

[SJK93] Leon Sterling, Ashish Jain, and Marc Kirschenbaum. Composition based on skeletons and techniques. In *ILPS '93 post conference workshop on Methodologies for Composing Logic Programs, Vancouver*, October 1993.

[Smi85] D.R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

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

[SST92] Donald Sannella, Stefan Sokolowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29(8):689–736, 1992.

[ST88] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76(2/3):165–210, February/March 1988.

[Sta97] Susanne Stasch. Fallstudie für Sprachdefinitionen aus wiederverwendbaren Bausteinen auf der Basis des Semantic Grammar Calculus. Master's thesis, University of Rostock, Department of Computer Science, 1997.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[SV91] Doaitse Swierstra and Harald Vogt. Higher Order Attribute Grammars. In Alblas and Melichar [AM91], pages 256–296.

[Tho96] Simon Thompson. *Haskell, The Craft of Functional Programming.* Addison-Wesley, 1996.

[Trc93] S. Trcek. A contribution to refinement of logic programs. Studienarbeit, 1993.

[TS97] Walid Taha and Tim Sheard. Multi-Stage Programming with Explicit Annotations. In *PEPM '97, Amsterdam, June 1997*, 1997.

[TWW81] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. More an advice on structuring compilers and proving them correct. *Theoretical Computer Science*, 15:223–249, 1981.

[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.

[War93] D.H.D Warren. An Abstract Prolog Instruction Set. Technical report, 1993. Technical Note 309, Artifical Intelligence Center, SRI International.

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

[WH97] Keith Wansbrough and John Hamer. A modular monadic action semantics. In *Conference on Domain-Specific Languages*, pages 157–170. The USENIX Association, 1997.

[Wil90] Reinhard Wilhelm. Tree transformations, functional languages, and attribute grammars. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*, pages 116–129. Springer-Verlag, New York–Heidelberg–Berlin, September 1990. Paris.

[Wir74] Niklaus Wirth. On the composition of well-structured programs. *ACM Computing Surveys*, 6(4):247–259, December 1974.

[Wir86] Martin Wirsing. Structured algebraic specifications: A kernel language. *Theoretical Computer Science*, 42(2):123–249, August 1986.

[Wir94] M. Wirsing. Algebraic specifiation languages: An overview. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification, 10th Workshop on Specification of Abstract Data Types, Joint with the 5th COMPASS Workshop, S. Margherita, Italy, May/June 1994, Selected Papers*, volume 906 of *LNCS*, pages 81–115. Springer-Verlag, 1994.

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

Index