Evolution of Rule-Based Programs

Ralf Lämmel

Free University
Faculty of Sciences
Department of Information Management and Software Engineering
De Boelelaan 1081a
NL-1081 HV Amsterdam
The Netherlands

Centrum voor Wiskunde en Informatica
Department of Software Engineering
Kruislaan 413
NL-1098 SJ Amsterdam
The Netherlands

Abstract

The term *rule-based program* is meant to include definite clause programs, SOS specifications, attribute grammars, and conditional rewrite systems. These setups are widely used for the executable specification or implementation of language-based tools, e.g., interpreters, translators, type checkers, program analysers, and program transformations.

We provide a pragmatic, transformation-based approach for expressing and tracking changes in rule-based programs in the course of program evolution. To this end, we design an operator suite for the transformation of rule-based programs. The operators facilitate steps for clean-up, refactoring, and enhancement. We use SOS-based interpreter examples to illustrate evolution of rule-based programs. We use logic programming to execute the examples, while the relevant evolution operators are made available as logic meta-programs.

Keywords: Software Evolution, Program Transformation, Rule-Based Programming, Metaprogramming, Natural Semantics, Structural Operational Semantics, Logic Programming, Attribute Grammars, Constructive Algebraic Specification, Refactoring, Extensibility, Reuse

Email address: `ralf@cs.vu.nl` (Ralf Lämmel).
URL: `http://www.cs.vu.nl/~ralf` (Ralf Lämmel).

Last modified on 17 November 2004

Contents

1	Introduction	3
2	Illustrative evolution scenarios	6
2.1	SIMPLE — an expression language	7
2.2	IMPURE = SIMPLE + imperative constructs	8
2.3	PURE = SIMPLE + let-bindings	9
2.4	FUN = PURE + functions	10
2.5	STEP = PURE but in small-step style	12
2.6	OO = PURE + object-oriented constructs	14
2.7	MCI = OO + method-call interception	16
3	Evolution-enabled rule-based programs	19
3.1	Syntax of rule-based programs	19
3.2	The link between sorts and evolution	22
3.3	The link between modes and evolution	25
3.4	Semantics-based evolution relations	27
3.5	Contribution problems in programs	29
4	Primitive transformation operators	32
4.1	The category *abstraction*	34
4.2	The category *reference*	37
4.3	The category *parameter*	38
4.4	The category *aggregate*	39
4.5	The category *composite*	42
5	Derived transformation operators	44
5.1	Adding definitions and uses	44
5.2	The threading technique	46
5.3	Bottom-up computation of data	48
5.4	Refreshment of data flow	51
6	Related work	52
6.1	Program calculation	53
6.2	Stepwise refinement	53
6.3	Stepwise enhancement	54
6.4	Extensibility vs. evolution	56
7	Concluding remarks	58

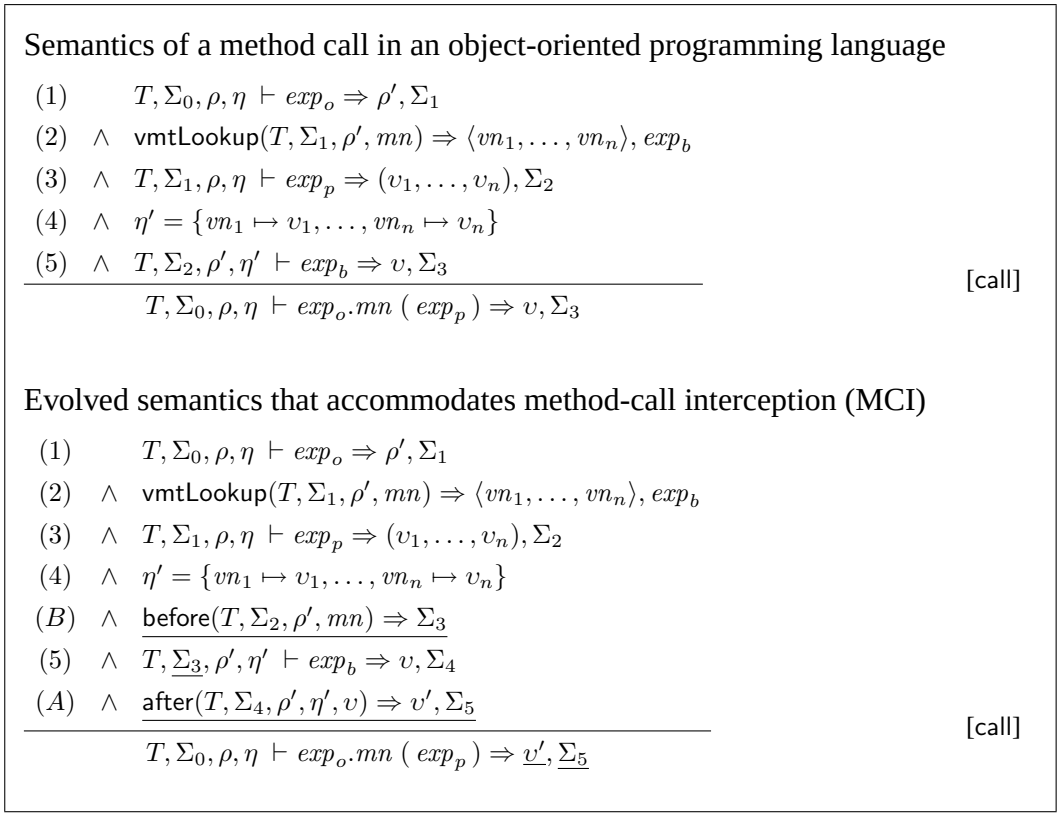

Fig. 1. *Evolution of an operational semantics for an object-oriented programming language*

1 Introduction

The research context of this paper is *software evolution* for declarative programming and executable specification. More specifically, we are concerned with the following kinds of programs or executable specifications:

- Logic programs, most notably definite clause programs.
- SOS specifications (small step and big step style).
- Attribute grammars (in Knuth's sense, but also other notations).
- Constructive algebraic specifications and conditional rewrite systems.
- First-order functional programs.

We use the term *rule-based programs* to refer to such programs or specifications.

An evolution scenario Software evolution is a widely established field in mainstream programming and software engineering. In the present paper, we adopt software evolution to rule-based programming. In general, software evolution is understood to mean change in properties (or attributes) of software. In Fig. 1, an example of changing a rule-based program is illustrated, while employing Natural Semantics [Kah87]. The example deals with the evolution of an operational semantics for

an object-oriented programming language. We only show the rule for the meaning of method calls. [1] The evolution scenario is about facilitating *method-call interception* (MCI [Läm02]). That is, the execution of a method call can potentially involve extra functionality to be executed before and after the actual method body; see the underlined premises in the lower part of the figure. This extra functionality is also called *before and after advice* in aspect-oriented programming [EFB01, KLM+97]. The details of the semantical rules are not so important, but the example is rather meant to illustrate some sorts of changes.

Evolutionary transformations In the present paper, we use transformations for changing rule-based programs. In the example, we need to perform transformations to add the underlined premises, and to rearrange the data flow for the object store Σ_3, ..., Σ_5. We also need to set up the data flow for the preliminary result of a method call, which is potentially revised by the after advice; see the variables v and v'. The following transformation sequence tracks these changes:

- inject(..., [call]) (Inject extra premises before(...) and after(...))
- refresh(Σ, [call]) (Disconnect positions for stores Σ)
- refresh(v, [call]) (Disconnect positions for values v)
- thread(Σ, [call]) (Establish threading-like data flow for stores)
- thread(v, [call]) (Establish threading-like data flow for values)

This sequence derives the evolved rule [call] in the lower part of the figure from the original rule [call] in the upper part. The evolution operators inject, refresh and thread are employed to complete the full evolution scenario. We will explain the operators in detail later. A rule-based programmer can write down such transformation sequences as an operationalisation and documentation of a change. One can also think of interactive transformations, where such transformation sequences are produced in the background. In the present paper, we view evolutionary transformations as meta-programs. In fact, the paper is backed up by a Prolog-based meta-programming framework — the *R*ule *E*volution *K*it (*REK* [REK04]). *REK* offers a list of evolution operators, and it comes with examples that illustrate the approach within the domain of language interpretation.

A discipline of editing We view evolutionary transformations as a tool to express the programmer's intentions about changing programs in a disciplined and convenient manner. We aim at an operator suite that captures many typical sorts of changes that programmers otherwise perform using text or structure editors. The

[1] Specification details: The judgement for expression evaluation takes the form $T, \Sigma, \rho, \eta \vdash exp \Rightarrow v, \Sigma'$. Here, T is the method table, Σ is the input store, Σ' is the output store, ρ is the object reference denoted by "**this**", η is the environment for method arguments and let-bindings, and v is the result of expression evaluation. The rule defines the semantics of method calls as follows. In premise (1), the callee for the method call is determined. In premise (2), the virtual method-table lookup is performed. In premise (3), the arguments are evaluated. In premise (4), the environment for the method execution is constructed. In premise (5), the method body is performed.

added value of using transformations is that the changes are tracked, and that one can reason about the changes by referring to the transformation properties of the involved operators. We will deal with a number transformation properties as opposed to simply semantics preservation. These properties help assessing the direction of evolution and the degree of preservation. Let us consider a very simple editing scenario, where a program B could be derived from a program A by removing some rules, and adding some other rules. This is a very syntactical formulation because the roles of the removed and added rules are not yet identified. For instance, adding a rule can serve quite different purposes:

- An added rule can extend an existing definition by a new case.
- An added rule can provide a so-far lacking definition.
- An added rule can provide a so-far lacking base case.
- An added rule could introduce a completely new definition.

Evolutionary transformations can readily distinguish these situations. So even if all of the above scenarios amount to the same syntactical operation of adding a rule, there can be different evolution operators that use preconditions to restrict themselves to the respective scenario.

Evolution in context There are two approaches that are complementary:

- "*Don't transform! Be extensible!*": Assuming a well-understood design space for a problem domain, one can try to reduce evolution to extensibility. A showcase is the domain of programming language semantics, where extensibility means that the specification of a given language construct should ideally be *definitive*, i.e., no revisions are required when new language constructs are added [Mog91, Mos92, CF94, LHJ95, Mos02]. Extensibility rests on powerful forms of parameterisation, abstraction, and modular composition as, for example, in Modular SOS [Mos99]. By contrast, our approach applies to basic rule-based notations, and it acts as a disciplined replacement for editing, which includes general program revision and refactoring in addition to merely program extension.
- "*Prove your transformations!*": Transformational program development [BD77, Par90, PP96, Bd97] advocates the use of transformations for the derivation of (efficient) implementations from (high-level) specifications. This approach relies on semantics-preserving transformations, maybe modulo a refinement relation, and it is normally coupled with a mathematical modus operandi. Our approach is more pragmatic. We advocate tracking more general program changes on the basis of an editing-like intuition for clean-up, refactoring, and enhancement. We do not insist on semantics preservation for several reasons. Firstly, this would rule out evolution scenarios other than refactoring. Secondly, compound changes often involve intermediate results that do not meet semantics preservation. Thirdly, side conditions for semantics preservation are often not amenable to automatic checking. Evolutionary transformations involve checkable preconditions, but proofs of any form are not requested from programmers.

Road-map of the paper

- In Sec. 2, we work through a number of evolution scenarios using interpreter examples. We take advantage of the Prolog-based implementation of our approach as supported by *REK* — the *R*ule *E*volution *K*it [REK04].
- In Sec. 3, we provide a characterisation of rule-based programs, which takes evolution into account. Most notably, evolution relations are identified that are meant to measure the distance between an original and an evolved program.
- In Sec. 4, we define primitive operators for evolutionary transformations. This development is based on an abstract classification of syntactical categories and their corresponding evolution operators.
- In Sec. 5, we define some typical derived operators. These are schemes of program adaptation that iterate over a program in a certain way, and that use simple analyses to drive the adaptation.
- In Sec. 6, we discuss related work. This discussion will relate the evolutionary approach to complementary notions, namely modular SOS, transformational program development, stepwise enhancement, and stepwise refinement.
- In Sec. 7, the paper is concluded.

2 Illustrative evolution scenarios

To explore evolution scenarios, we will now work through a number of interpreter examples. The examples cover different facets of evolution: enhancement, refactoring, and revision. We have chosen language interpretation because this is a standard benchmark for methods of reuse [Mos92, CF94, LHJ95, Mos02]. All the specifications to come are executable, tested Prolog programs in virtue of *REK* — the *R*ule *E*volution *K*it for Prolog [REK04]. In fact, these Prolog programs can be viewed as (encodings of) SOS rules on the basis of widely established mappings from SOS to Prolog [Des88, Pet94, SK95]. The evolution scenarios are operationalised by means of logic meta-programming. That is, we describe the evolution of an interpreter in terms of applications of meta-program operators.

We will begin with an interpreter for a trivial expression language, which is sufficient to explain some notational issues. We will then provide pure, impure, functional, object-oriented and aspect-oriented extensions of the interpreter. The last interpreter will indeed reconstruct the introductory example from Fig. 1. The evolution scenarios will require meta-program operators such as the following:

- Add argument positions to existing predicates.
- Wrap argument positions as terms or tuples.
- Establish an environment- or state-like data flow.
- Inject additional literals into existing clauses.

- Revise data flow in existing clauses.
- Turn existing type aliases into proper new datatypes.
- Perform some forms of normalisation.
- Migrate from big-step to small-step style of SOS.

2.1 SIMPLE — an expression language

There are three constructs in the simple language: constants, binary (arithmetic) expressions, and if-expressions. Using *REK*'s notation for algebraic datatypes, the corresponding abstract syntax is declared as follows: [2]

```
:- data exp = const(number)       % Constants
            | binary(exp,op,exp)  % Binary expressions
            | if(exp,exp,exp).    % If-expressions
```

We omit the definition of the datatype `op`, which comprises the normal binary operators on number types. The evaluation of an expression is meant to return a number. (The datatype `number` is a basic datatype of Prolog; it comprises `integer` and `float`.) For clarity, we define a corresponding type alias `val` for results of expression evaluation:

```
:- alias val = number.
```

The predicate for expression evaluation is of the following type in *REK*: [3]

```
:- profile evaluate(+exp,-val).
```

The following clauses define the interpreter:

```
evaluate(const(Number),Number).

evaluate(binary(Exp0,Op,Exp1),Val2) :-
   evaluate(Exp0,Val0),
   evaluate(Exp1,Val1),
   apply_bop(Op,Val0,Val1,Val2).

evaluate(if(Exp0,Exp1,_Exp2),Val1) :-
   evaluate(Exp0,Val0),
   is_not_zero(Val0),
   evaluate(Exp1,Val1).
```

[2] Notation: Datatypes are declared by Prolog directives starting with "`:- data ...`". The various constructor declarations of a datatype are separated by "|". We refer to [MO84, AL94, BM97, Jef02] for background on type-checking logic programs. *REK*'s notation is adopted from [LR01].

[3] Notation: Predicate types are prefixed by "`:- profile ...`", and each argument position can be associated with a sort and a mode. Again, this notation is adopted from [LR01]. The modes "+" and "-" denote input resp. output positions. The intuition is that predicates compute output positions from input positions. We refer to [AL94, BM97, Jef02] for some background on modes in logic programming.

For example, the second clause defines that binary expressions are evaluated by evaluating both operands and applying the operator to the two obtained values. We omit the definitions of trivial helper predicates for testing for "0" (i.e., `is_zero/1` and `is_not_zero/1`), and for applying binary operators (i.e., `apply_bop/4`).

2.2 IMPURE = SIMPLE + imperative constructs

We will now add some imperative constructs to SIMPLE, namely:

```
:- data exp = var(varid)        % Read a variable
            | assign(varid,exp) % Assign to a variable
            | concat(exp,exp)   % Sequential composition
            | while(exp,exp).   % While-loops

:- alias varid = atom.
```

These new constructor declarations are simply added to the previous expression forms. Eventually we need to add a store to `evaluate/2` so that we can model imperative variables. However, the last two constructs given above do not *directly* interact with stores. Hence, we just add the two corresponding clauses to SIMPLE's predicate `evaluate/2`:

```
evaluate(concat(Exp1,Exp2),Val) :-
  evaluate(Exp1,_Val),
  evaluate(Exp2,Val).

evaluate(while(Exp1,Exp2),Val1) :-
  evaluate(if(Exp1,concat(Exp2,while(Exp1,Exp2)),const(0)),Val1).
```

These two clauses and SIMPLE's clauses have to be lifted to a store-aware semantic model. Stores are lists of variable identifier-value pairs. We define a corresponding type alias:

```
:- alias store = [(varid,val)].
```

We are now in the position to extend the predicate for expression evaluation by adding positions of type `store`. In fact, we need both an input and an output position. This is achieved by the following two transformations:

```
:- add(+store,evaluate).
:- add(-store,evaluate).
```

These are Prolog directives that invoke a meta-programming operator `add/2` for adding parameters to predicates. All such operators directly operate on Prolog's database. The type of the predicate for expression evaluation has grown as follows:

```
:- profile evaluate(+exp,+store,-val,-store).
```

For instance, the clause for sequential composition has evolved as follows:

```
evaluate(concat(Exp0,Exp1),_Store0,Val1,_Store5) :-
  evaluate(Exp0,_Store1,_Val0,_Store2),
  evaluate(Exp1,_Store3,Val1,_Store4).
```

That is, there are various new variables of type `store`. We still need to unify these variables in a way to encode state passing. [4] This is achieved by the application of another operator:

```
:- thread(store).
```

The name `thread` hints at the data flow that is wired up with this operator. That is, the store enters a clause via the input position of the head, and then it is threaded through the body, and finally it leaves the clause via the output position of the head. For instance, the clause for sequential composition has further evolved as follows:

```
evaluate(concat(Exp0,Exp1),Store0,Val1,Store2) :-
  evaluate(Exp0,Store0,_Val0,Store1),
  evaluate(Exp1,Store1,Val1,Store2).
```

This is the normal semantics of sequential composition in an imperative language with a store. As a last step, it remains to add the constructs for variable access:

```
evaluate(var(Varid),Store,Val,Store) :-
  lookup(Store,Varid,Val).

evaluate(assign(Varid,Exp),Store0,Val,Store2) :-
  evaluate(Exp,Store0,Val,Store1),
  modify(Store1,Varid,Val,Store2).
```

That is, we simply add two more clauses relying on the fact that the semantic model for expression evaluation has been elaborated previously to include stores. For brevity, we omit the straightforward specifications of the predicates `lookup/3` for looking up values from finite maps, and `modify/4` for modifying finite maps in a point. This completes our first evolution scenario.

2.3 PURE = SIMPLE + let-bindings

Let us also consider a PURE extension of SIMPLE, where we add let-bindings instead of imperative variables. These are the corresponding constructs:

[4] Terminology: State passing is a common term in the context of functional programming [Wad92]. Such state passing is also called accumulation in logic programming [SS94], or a chain or a bucket brigade in the case of attribute grammars [Ada91, KW94].

```
:- data exp = var(varid)           % Read a variable
            | let(varid,exp,exp).  % Let-expressions

:- alias varid = atom.
```

The semantics of lets employs environments according to the following type alias:

```
:- alias env = [(varid,val)].
```

The following transformations establish environments in SIMPLE's clauses:

```
:- add(+env,evaluate).
:- thread(env).
```

That is, an input position of type `env` is added, and we again employ the `thread/1` operator for wiring up the data flow for the added parameters. This time, environment passing is achieved because there is only an input position of the relevant type. For instance, the interpretation of binary expressions evolves as follows:

```
evaluate(binary(Exp0,Op,Exp1),Env,Val2) :-
  evaluate(Exp0,Env,Val0),
  evaluate(Exp1,Env,Val1),
  apply_bop(Op,Val0,Val1,Val2).
```

That is, the environment `Env` enters the clause via the new parameter in the head, which is then passed on to some premises in the body. The PURE extension of SIMPLE is completed by adding the clauses for variables and let-expressions:

```
evaluate(var(Varid),Env,Val) :-
  lookup(Env,Varid,Val).

evaluate(let(Varid,Exp0,Exp1),Env0,Val1) :-
  evaluate(Exp0,Env0,Val0),
  modify(Env0,Varid,Val0,Env1),
  evaluate(Exp1,Env1,Val1).
```

The PURE and IMPURE extensions showcase the overall mode of extension when using evolutionary transformations: clauses for new constructs are added once the preexisting clauses were lifted to the required semantic model. The PURE and the IMPURE extension could be combined to provide both pure variables and impure locations as in a SML-like language [MTH90].

2.4 FUN = PURE + functions

We will now enhance PURE to become a simple functional language in the sense of an applied λ-calculus. As far as abstract syntax is concerned, this necessitates the following extension:

```
:- data exp = lambda(varid,exp)  % Lambda-abstractions
            | apply(exp,exp).    % Function applications
```

As far as the semantic domains are concerned, we need to enhance the structure of values that are obtained by expression evaluation. This is because the result of evaluating a λ-abstraction is a function closure. While we considered the domain `val` to coincide with `number` so far, we now have to turn it into a proper datatype. This is expressed by the following transformation:

```
:- newtype(val,num).
```

At the type level, this implies that the original type alias `val` becomes a proper datatype declaration as if it were declared as follows:

```
:- data val = num(number).
```

The application of the `newtype/2` operator also implies that all clauses evolve such that each and every parameter of type `val` is wrapped by the functor `num`. For instance, the clauses for constants and let-expressions evolve as follows:

```
evaluate(const(Number),_Env,num(Number)).

evaluate(let(Varid,Exp0,Exp1),Env0,num(Number1)) :-
  evaluate(Exp0,Env0,num(Number0)),
  modify(Env0,Varid,num(Number0),Env1),
  evaluate(Exp1,Env1,num(Number1)).
```

Notice the added occurrences of `num/1`. The fact for constants is perfectly appropriate because it is concerned with the `num`-case of `val`. That is, the abstract syntactical term `const(Number)` evaluates to the value `num(Number)`. However, the clause for let-expressions is overly specific in the view of the planned extension. That is, the various occurrences of injections and projections via the functor `num/1` preclude the application of this rule to expressions with results other than numbers. The following transformation resolves this issue:

```
:- relax(val).
```

For instance, it affects the clause for let-expressions as follows:

```
evaluate(let(Varid,Exp0,Exp1),Env0,Val1) :-
  evaluate(Exp0,Env0,Val0),
  modify(Env0,Varid,Val0,Env1),
  evaluate(Exp1,Env1,Val1).
```

The terms on the result positions are now all plain variables as opposed to the earlier terms that were of the form `num(...)`. This reflects that a let-expression

does not constrain the result types for the evaluation of its ingredients. The operator `relax/1` supports a certain form of anti-unification [Plo70, Rey70, LMM88]. Namely, terms of the given type are replaced by variables. The operator checks that there is precisely one functor for the given type because this guarantees that the occurrences of the functor do not properly constrain the term at hand anyway.

We are now in the position to elaborate the datatype `val` such that function closures are accommodated, while we repeat the preexisting functor `num` for clarity:

```
:- data val = num(number)          % Numbers as values
            | fun(env,varid,exp).  % Function closures
```

We are now done with all preparations so that we can add the clauses that interpret λ-abstractions and function applications:

```
evaluate(lambda(Var,Exp),Env,fun(Env,Var,Exp)).

evaluate(apply(Exp1,Exp2),Env0,Val0) :-
  evaluate(Exp1,Env0,fun(Env1,Var,Exp0)),
  evaluate(Exp2,Env0,Val2),
  modify(Env1,Var,Val2,Env2),
  evaluate(Exp0,Env2,Val0).
```

We note that the previous examples PURE and IMPURE emphasised evolution in the argument dimension including data flow, while the example FUN demonstrated evolution in the datatype dimension including term matching and building.

2.5 STEP = PURE but in small-step style

The interpreter examples that we saw so far are all given in big-step style (i.e., Natural Semantics [Kah87]). In principle, one might favour different styles, namely big-step vs. small-step vs. mixed style; see [Plo81, Mos02] for a discussion. For instance, it is commonplace to describe the semantics of language constructs for concurrency in small-step style. A benefit of a transformation framework like *REK* is that there is no need for a commitment to this or that style, but — subject to reasonable preconditions — one can go back and forth between big-step and small-step style; one can also mix these styles on a per-predicate basis. The detailed discussion of such style conversions is beyond the scope of this paper, but we will illustrate the evolution of PURE's big-step semantics into small-step style.

The first provision is to enable the value-added syntax trees of a small-step semantics. To this end, we add a constructor `val2exp` to potentially embed evaluation results into the expression domain, i.e., `val2exp` goes from `val` to `exp`:

```
:- data exp = val2exp(val).
```

We now need to change the profile of `evaluate/3` such that the output position of type `val` becomes of type `exp`. Such an adapted type is a prerequisite for small steps, which relate two expressions — as opposed to big steps, which relate an expression and an ultimate evaluation result. Changing the predicate's type alone would not be valid, but we also need to wrap the affected terms using `val2exp`. This is accomplished by the following transformation:

```
:- othertype(evaluate,val2exp).
```

The operator `othertype` is a cousin of `newtype`, which we used in the construction of FUN. Both operators wrap a position by a unary functor. In the case of `othertype`, the relevant functor is type-changing. Here is the changed type of `evaluate`, and we also show the evolved clause for let-expressions, which involves several occurrences of the `val2exp` wrapper:

```
:- profile evaluate(+exp,+env,-exp).

evaluate(let(Varid,Exp0,Exp1),Env0,val2exp(Val1)) :-
  evaluate(Exp0,Env0,val2exp(Val0)),
  modify(Env0,Varid,Val0,Env1),
  evaluate(Exp1,Env1,val2exp(Val1)).
```

The type of `evaluate/3` is now readily prepared to make the transition from big-step to small-step style. This transition is issued as follows:

```
:- big2small(evaluate,exp).
```

The operator `big2small` takes the predicate to be transformed as well as the type on which induction is performed. The operator performs a type-preserving transformation for the given predicate while splitting up clauses such that small steps are performed. Here are the three small-step clauses for let-expressions, which were derived from the big-step clause:

```
evaluate( let(Varid,Exp0,Exp1), Env
        , let(Varid,Exp2,Exp1)
        )
 :- evaluate(Exp0,Env,Exp2).

evaluate( let(Varid,val2exp(Val),Exp0) ,Env0
        , let(Varid,val2exp(Val),Exp1)
        )
 :- modify(Env0,Varid,Val,Env1), evaluate(Exp0,Env1,Exp1).

evaluate( let(Varid,val2exp(Val0),val2exp(Val1)), Env0
        , val2exp(Val1)
        )
 :- modify(Env0,Varid,Val0,_Env1).
```

The inner workings of this operator are somewhat involved, [5] but the overall scheme can be summarised as follows. There are as many small-step rules as there are premises in the big-step rule plus an extra rule for the final step. The rule 'per premise' performs one step for the subterm at hand. The last step computes the final result from the reduced subterms. This applies to the above rules for let-expressions as follows. The first rule progresses with the expression `Exp0` in `let(...,Exp0,...)`. The second rule progresses with the expression `Exp1` in `let(...,...,Exp1)`. The last rule returns the value obtained from the latter expression as the result of the let-expression.

We can recover a big-step relation by providing a predicate that takes the transitive closure of the small-step relation. The following predicate `evaluate_big/3` computes this closure. For clarity, we also rename the predicate `evaluate/3` to `evaluate_small/3`:

```
:- rename(pred(evaluate/3),pred(evaluate_small/3)).

evaluate_big(Exp0,Env,Val) :-
  evaluate_small(Exp0,Env,Exp1), % Do one step!
  ( evaluate_big(Exp1,Env,Val)   % Do more steps if possible!
  ; Exp1 = val2exp(Val)          % All steps done; value found.
  ).
```

We note that this example goes beyond the earlier examples in that it does not simply elaborate predicate profiles, or data flow, or datatype structure, but it rather mediates between different styles.

2.6 OO = PURE + object-oriented constructs

We pick up PURE again to extend it this time with object-oriented constructs. To see what will come, we provide the abstract syntax of the new expression forms:

```
:- data exp = this                    % The active object
            | get(exp,field)          % Reading field access
            | set(exp,field,exp)      % Writing field access
            | new(class)              % Object construction
            | call(exp,meth,[exp]).   % Method calls

:- alias class = atom.                % Class ids
:- alias field = atom.                % Field ids
:- alias meth  = atom.                % Method ids
```

We begin with a number of very simple transformations. That is, some predicates are renamed to avoid name clashes, and the type for expression evaluation is turned

[5] The `big2small` operator relies on a number of non-trivial assumptions. Most notably, the data flow is searched for state or environment passing. The obtained information is then used to control the order of steps. Also, notational variations on structural induction have to be harmonised. Furthermore, the operator performs normalisations, e.g., for the removal of double steps.

into a proper datatype:

```
:- rename(alias(env),alias(argenv)).
:- rename(pred(lookup/3),pred(lookupArgenv/3)).
:- rename(pred(modify/4),pred(modifyArgenv/4)).
:- newtype(val,num).
:- relax(val).
```

The semantic model of the forthcoming OO language requires the following extension of type `val`, and some additional domains for method tables, object stores, and other common ingredients [BSG95, ACE96, FKF99]:

```
:- data  val   = num(number)  % Numbers as results
               | ref(ref).    % Object references as results

:- alias ref   = integer.
:- alias this  = maybe(ref).                % The active object
:- alias vmt   = [(class,meth,[varid],exp)]. % Method tables
:- alias store = [(ref,class,obj)].         % Object stores
:- alias obj   = [(field,val)].             % Field-value pairs
```

We lift the PURE semantics to the OO level using transformations for threading akin to those illustrated for the PURE and IMPURE extensions. That is, we need to add input and output positions, and we need to establish environment and state passing:

```
:- add(+vmt,evaluate).
:- add(+this,evaluate).
:- thread(vmt).
:- thread(this).
:- add(+store,evaluate).
:- add(-store,evaluate).
:- thread(store).
```

For clarity, this is the current type of the judgement for expression evaluation:

```
:- profile evaluate(+exp,+vmt,+argenv,+this,+store,-val,-store).
```

One could argue that this judgement starts to become somewhat complex because of the many argument positions. This calls for some refactoring, which will be accomplished below indeed.

At this point, we can add clauses for the interpretation of object-oriented constructs. Field access and and object construction is omitted for brevity. The following two rules cover the constructs `this` and `call`(...):

```
evaluate(this,_Argenv,_Vmt,just(Ref),Store,ref(Ref),Store).

evaluate(call(Exp0,Meth,Exps),Argenv0,Vmt,This,Store0,Val,Store3)
 :-
    evaluate(Exp0,Argenv0,Vmt,This,Store0,ref(Ref),Store1),  %1
    evaluatelist(Exps,Argenv0,Vmt,This,Store1,Vals,Store2),  %2
    lookupVmt(Vmt,Store2,Ref,Meth,Varids,Exp1),              %3
    zip(Varids,Vals,Argenv1),                               %4
    evaluate(Exp1,Argenv1,Vmt,just(Ref),Store2,Val,Store3).  %5
```

The fact for `this` simply returns the active object as maintained as a dedicated argument of the judgement. The clause for method calls models steps as follows:

(1) The receiver `Ref` of the call is determined by expression evaluation.
(2) The actual arguments `Exps` are evaluated resulting in a list `Vals` of values.
(3) The body and the formal arguments of the called method are looked up.
(4) Actual arguments are bound to argument names by zipping.
(5) The body of the called method is evaluated in the right environment.

2.7 MCI = OO + method-call interception

We will now operationalise the introductory example. That is, we elaborate the OO interpreter to integrate constructs for the interception of method calls. The central language construct serves for superimposition of extra functionality onto selected method calls. Here is the abstract syntax:

```
:- data exp = superimpose(
           exp   % Extra functionality (aka advice code)
         , jpq   % Join-point qualifier
         , exp   % Object whose invocations to be intercepted
         , meth  % Method whose invocations to be intercepted
         ).

:- data jpq = before | after.
```

That is, a superimposition expression adds functionality (say, advice) to a method of a given object using a qualifier `before` vs. `after`. Advice code can typically interact with all ingredients of the join point (such as caller, callee, arguments, result). For instance, in `after` advice, one can refer to the preliminary result using the construct `getResult`.

```
:- data exp = getResult.  % Read the call's result
```

The MCI-enabled interpreter will need to maintain a registry for advice, and an environment for join-point interaction. To this end, the following semantic domains are added:

```
:- alias mci = [ ( jpq   % Join-point qualifier
                 , ref   % Object whose invocations to intercept
                 , meth  % Method whose invocations to intercept
                 , exp   % Expression to be executed
                 , this  % Registering object
                 )
               ].

:- alias resenv = maybe(val).  % Preliminary return value if any
```

Each entry in the `mci` registry characterises a join point for executing extra functionality. An entry also maintains the `this` reference of the superimposing site so that the advice code can be executed in this environment.

The registry and the join-point environment are threaded as follows:

```
:- add(+mci,evaluate), add(-mci,evaluate).
:- add(+mci,evaluatelist), add(-mci,evaluatelist).
:- thread(mci).
:- add(+resenv,evaluate).
:- add(+resenv,evaluatelist).
:- thread(resenv).
```

Before we continue with the MCI extension, we should refactor the predicate for expression evaluation, which has meanwhile 10 positions. We can group several positions in a natural manner. Namely. we can group environment- and state-like arguments. This refactoring is accomplished as follows:

```
:- rename(alias(store),alias(objstore)).
:- alias env   = (argenv,vmt,this,resenv).
:- alias store = (objstore,mci).
:- group(env).
:- group(store).
```

The operator `group/1` turns the positions corresponding to the components of a tuple type into a single position. That is, multiple predicate positions are wrapped by tuple construction. In the example, the object store and the MCI registry are grouped as a store-like product, and the four components `argenv`, `vmt`, `this` and `resenv` are grouped as an environment-like product. As a result, we are back to a very simple type for expression evaluation:

```
:- profile evaluate(+exp,+env,+store,-val,-store).
```

We are now in the position to accommodate the semantics of method-call interception. Most notably, this requires an adaptation of the existing clause for interpreting method calls. That is, we have to instrument the clause such that additional premises model the execution of relevant advice code before or after the execution of the method body. This is accomplished by the following transformation:

```
:- inject((
    evaluate(call(_,Meth,_),_,_,_,_) :-
      _,
      _,
      lookupVmt(_,_,_,_,_,Exp),
      _,
      { before(Ref,Meth,Argenv,Vmt,Store0,Store0) },
      evaluate(Exp,(Argenv,Vmt,just(Ref),_),Store0,Val,Store1),
      { after(Ref,Meth,Argenv,Vmt,Val,Store1,Val,Store1) }
)).
```

The `inject/1` operator takes the 'sketch' of a clause with additional premises surrounded by $\{\ldots\}$. By sketch we mean that the clause does not need to be written precisely as in the database, but underscores and variables can abbreviate premises and terms. This serves convenient and tolerant recording of clauses that need an adaptation. More formally, the preexisting clause must be an *instance* of the sketch. The above example illustrates that the new premises `before(...)` and `after(...)` can pick up variables of the preexisting premises. This is useful to connect the new premises to data flow of the original clause. For brevity, we omit the specification of `before/6` and `after/8`, which simply iterate over the MCI registry to execute advice code that is registered for the method call at hand.

Following this injection, we have to complete the data flow in the clause such that threading of stores and preliminary results goes through the new premises. To this end, we use an operator `refresh/2` to disconnect all positions of a given sort in the clause for method calls, and then, we perform threading for this refreshed clause:

```
:- refresh(store,evaluate(call)).
:- refresh(val,evaluate(call)).
:- thread(store,evaluate(call)).
:- thread(val,evaluate(call)).
```

These operator applications employ an addressing mechanism where the relevant clause is specified as `evaluate(call)`. Here, `evaluate` denotes the relevant predicate, and `call` is the functor selecting among the discriminated cases. Refreshment means that the variables of the relevant sort are replaced by fresh ones — for each occurrence a fresh variable is chosen as opposed to consistent renaming.

The semantics of `superimpose(...)` and `getResult` are now straightforward:

```
evaluate( superimpose(Exp0,Jpq,Exp1,Meth)
        , Env,Store1,ref(Ref),Store3
        ) :-
  evaluate(Exp1,Env,Store1,ref(Ref),Store2),
  Env    = (_Argenv,_Vmt,This,_Resenv),
  Entry  = (Jpq,Ref,Meth,Exp0,This),
  Store2 = (ObjStore,Mci),
  Store3 = (ObjStore,[Entry|Mci]).

evaluate(getResult,(_,_,_,just(Val)),Store,Val,Store).
```

For `superimpose(...)`, we assemble an entry from all the operands of superimposition, and we add it to the MCI registry. For `getResult`, we simply look up the dedicated component from the environment.

This completes our aspect-oriented interpreter extension. We note that this extension goes beyond plain semantics preservation. Most notably, premises were injected into clauses to further constraint them, and the threading-like data flow in clauses was refreshed such that extra premises could participate in the threads.

3 Evolution-enabled rule-based programs

We will now characterise rule-based programs — their syntax, typing, and semantics. While we aim at an evolutionary approach that is meaningful for a variety of rule-based notations, the following characterisation will be biased towards definite clause programs. This is unavoidable in the view of a technical discussion. Nevertheless, we will comment on other rule-based notations on several occasions. Our definition of rule-based programs will emphasise evolutionary concerns. For instance, typing is arranged such that types are readily useful in driving evolutionary transformations. Also, semantical issues of evolving programs are discussed. Furthermore, we will define various *evolution relations* on rule-based programs, which capture directions of making progress in the course of evolution.

3.1 Syntax of rule-based programs

In Fig. 2, we define a core syntax for rule-based programs, which deliberately resembles the syntax of (typed) definite clause programs. We postpone discussing semantical issues until Sec. 3.4. A rule-based program s is a collection of rules and types for the occurring symbols. We use the notation $\Phi(s)$ or $\Psi(s)$ to extract the type contexts ϕ and ψ for predicates and functors from a rule-based program $s = \phi; \psi \rhd \ldots$. We use s to refer (ambiguously) to both a program and its collection of rules. For simplicity, we treat collections of rules as sets, ignoring the fact that order is often relevant for the operational semantics of rule-based notations. A rule takes the form of an implication, prefixed by a type context ω for variables. The assertion preceding "$\Leftarrow$" is called the conclusion (or the head of the rule), and the assertions following "$\Leftarrow$" are called premises (which are said to form the body of the rule). We write the empty body as $true$. Assertions are atomic formulae that apply predicate symbols to terms, which in turn are built from variables and functors. We denote the definition of a predicate p in a program s by $s_{/p}$, which is the collection of all rules whose head applies p. This notation is also used for sets of predicate symbols such as in $s_{/\{p_1,\ldots,p_n\}}$, which is the same as $s_{/p_1} \cup \cdots \cup s_{/p_n}$. We use $preds(s)$ to denote the set of all predicate symbols that are applied in any assertion in s — not counting type declarations for predicates in this case.

$$
\begin{array}{lll}
S & := \Phi; \Psi \rhd R \ldots R & \text{(Rule-based programs } s\text{)} \\
R & := \Omega \rhd A \Leftarrow A \wedge \cdots \wedge A & \text{(Rules } r\text{)} \\
A & := P(T, \ldots, T) & \text{(Assertions } a\text{)} \\
T & := V \mid F \mid F(T, \ldots, T) & \text{(Terms } t\text{)} \\
\Phi & := \emptyset \mid P : M\,\Sigma \times \cdots \times M\,\Sigma \mid \Phi \cup \Phi & \text{(Type contexts } \phi \text{ for predicates)} \\
\Psi & := \emptyset \mid F : \Sigma \times \cdots \times \Sigma \rightarrow \Sigma \mid \Psi \cup \Psi & \text{(Type contexts } \psi \text{ for functors)} \\
\Omega & := \emptyset \mid V : \Sigma \mid \Omega \cup \Omega & \text{(Type contexts } \omega \text{ for variables)} \\
M & := + \mid - \mid ? & \text{(Modes } \mu\text{)} \\
\Sigma & & \text{(Sorts } \sigma\text{)} \\
P & & \text{(Predicate symbols } p\text{)} \\
F & & \text{(Functors } f\text{)} \\
V & & \text{(Variables } v\text{)}
\end{array}
$$

Fig. 2. Syntax of rule-based programs

Types — why? The fact that the syntax of rule-based programs involves type contexts in the definitions of S (programs) and R (rules) seems to suggest that we require explicit declarations of types for predicates, functors, and variables. While inference of some type information might be acceptable, we indeed require explicit typing for the purpose of this article. As we will clarify below, we do not require types for the reason of strong typing, but we are rather interested in using type information for controlling meta-programming operators for evolution. This will be discussed in detail in Sec. 3.2 and Sec. 3.3.

Term-like data — why? It is debatable whether 'data notation' should be part of the essence of rule-based programs. The diversity of data notations in rule-based programming suggests to avoid commitment to a specific notation. Also, not every rule-based notation is readily bundled with data notation. For example, attribute grammars as defined by Knuth [Knu68] only fix the specification of context-free structure and attribute dependencies while assuming that an interpretation of attribute domains and function symbols will be supplied at a different stage. We insist on having data notation in our core syntax because many meaningful evolution scenarios tend to involve data notation. Our choice for a term-like data notation is in alignment with logic programs, SOS, and term rewriting.

Coverage of rule-based notations We discuss some classes of rule-based notations and how they match with the identified core syntax.

- The core syntax resembles definite clause programs quite faithfully. Types were added, but this is widely established in logic programming by now [MO84, AL94, BM97, LR01, Jef02]. Negative literals as in general logic programs could

be added to our syntax, but we omit them here for simplicity. While the semantics of negation is reasonably understood [vRS91], our discussion of evolution would become unnecessarily complicated. Our implementation *REK* [REK04] covers negation, and several other extensions.

- An SOS specification with its first-order configurations can be directly encoded in the core syntax according to mappings of SOS to logic programs [Des88, SK95]. Again, one could want to add negative premises (see [AFV01] for a discussion), but we omit them for simplicity. Our notation is akin to Pettersson's typed Relational Meta-Language (RML [Pet94]), which is a programming language for Natural Semantics. The core syntax misses RML's basic datatypes, but our implementation *REK* [REK04] offers Prolog's basic types.
- A rewrite system can be turned into a rule-based program in a meaningful way if data constructors (i.e., functors in our terminology) and other function symbols (i.e., predicate symbols in our terminology) are readily separated. This is the case for constructive algebraic specifications and first-order functional programs. Such rewrite systems can be normalised to our core syntax. Essentially, for a given rewrite rule, one needs to flatten nested applications of ordinary function symbols as series of premises. (This is folklore.) For example, consider the following rewrite system for computing factorial:

$$\begin{aligned} fac(zero) &\rightarrow succ(zero) \\ fac(succ(x)) &\rightarrow mult(succ(x), fac(x)) \end{aligned}$$

  Here we assume that x is a variable, while $zero$ and $succ$ are constructors for Naturals. All the other symbols are ordinary function symbols. The rules can be flattened as follows:

$$\begin{aligned} fac(zero, succ(zero)) &\Leftarrow true \\ fac(succ(x), z) &\Leftarrow fac(x, y) \wedge mult(succ(x), y, z) \end{aligned}$$

  That is, fac and $mult$ became predicates, where the last position is for the result of the original function. New variables y and z are used to name the intermediate results originating from flattening.
- Various grammar-based formalisms, most notably, attribute grammars can be mapped to our syntax. To this end, one will normally consider different kinds of predicate symbols: terminals, nonterminals, and relational symbols that do not serve for word derivation, but only for computations and conditions. Again, we take advantage of the well-established link between grammar-based formalisms and definite clause programs [PW80, Abr84, DM85, RL89, DM93, LR01].

Syntax-based evolution relations Even at a syntactical level, we can start to identify relations that characterise the distance between two rule-based programs, i.e., the original program and the evolved one. The most obvious relation is *structural equality*. The programs s and s' are structurally equal, denoted by $s = s'$,

if their representation according to Fig. 2 is the same modulo renaming of variables. Renaming of functors and predicate symbols should not be done implicitly, but we can define a dedicated relation for *structural equality modulo renaming* of functors or predicate symbols. Another way of relaxing structural equality is to restrict the claim to a given set X of predicate symbols. Such restricted claims are useful during evolution to clarify what parts of the evolving program are (structurally) preserved. For instance, operating on a focus can be characterised in this manner. Technically, the programs s and s' are structurally equal *restricted to* X, denoted by $s =_{/X} s'$, if $s_{/X} = s'_{/X}$. Here, we assume that $X \subseteq preds(s) \cup preds(s')$. This can be inverted such that X enumerates exceptions as opposed to structurally equal predicates. This is denoted as $s =_{/\overline{X}} s'$, which holds iff $s =_{/Y} s'$ with $Y = (preds(s) \cup preds(s')) \setminus X$. Equality claims can also be restricted to rules rather than predicates. One can also think of more involved relations at the structural level. Most notably, one can consider projections to relate two programs, namely projection of parameters in assertions as well as projection of assertions in rule bodies [PS90b, KSJ93, Jai95, Läm99b, LRL00]. Note that defining such structural relations would already amount to defining transformations, which is not our intention at this stage. We will later see that such structural projections can be characterised very concisely as relations at the semantic level.

3.2 The link between sorts and evolution

We now start defining the static semantics of rule-based programs. Eventually, we will identify notions of well-sortedness, well-modedness, and others. The present section will focus on well-sortedness, by which we mean that predicate symbols, functors and variables are used in accordance to the declared sorts. We will later add well-modedness, which is about mode information (cf. '+', '-', '?'). While the definition of the static semantics is largely straightforward, we will emphasise evolutionary concerns related to static semantics.

Well-sortedness is defined in Fig. 3 using SOS. The given rules resemble a straightforward, many-sorted type system for definite clause programs. We also assume that well-sortedness includes well-formedness of type contexts, but this is omitted in the figure for brevity. It is routine work to make common extensions for basic datatypes, polymorphism, and higher-order predicates. Our implementation *REK* [REK04] provides these extensions.

Holy well-sortedness We require well-sortedness for all rule-based programs encountered during evolution. In particular, evolution operators are supposed to only produce well-sorted results if any. Also, when composing transformations from primitive operators, not even an intermediate, non-observable result is allowed to be ill sorted. We will later see that violating static properties other than well-sortedness can be acceptable for intermediate results encountered during evolution, but well-sortedness is a basic requirement that must be meet at all times.

Well-sortedness of rule-based programs $\boxed{\mathcal{WS}(s)}$

$$\frac{\mathcal{WS}(\phi,\psi,r_1)\ \wedge\ \cdots\ \wedge\ \mathcal{WS}(\phi,\psi,r_n)}{\mathcal{WS}(\phi;\psi \rhd r_1 \ldots r_n)} \quad [\mathcal{WS}.1]$$

Well-sortedness of rules $\boxed{\mathcal{WS}(\phi,\psi,r)}$

$$\frac{\mathcal{WS}(\phi,\psi,\omega,a_0)\ \wedge\ \mathcal{WS}(\phi,\psi,\omega,a_1)\ \wedge\ \cdots\ \wedge\ \mathcal{WS}(\phi,\psi,\omega,a_n)}{\mathcal{WS}(\phi,\psi,\omega \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_n)} \quad [\mathcal{WS}.2]$$

Well-sortedness of assertions $\boxed{\mathcal{WS}(\phi,\psi,\omega,a)}$

$$\frac{\begin{array}{c} p : \mu_1\,\sigma_1 \times \cdots \times \mu_n\,\sigma_n\ \in\ \phi \\ \wedge\ \mathcal{WS}(\phi,\psi,\omega,t_1,\sigma_1)\ \wedge\ \cdots\ \wedge\ \mathcal{WS}(\phi,\psi,\omega,t_n,\sigma_n) \end{array}}{\mathcal{WS}(\phi,\psi,\omega,p(t_1,\ldots,t_n))} \quad [\mathcal{WS}.3]$$

Well-sortedness of terms $\boxed{\mathcal{WS}(\phi,\psi,\omega,t,\sigma)}$

$$\frac{v : \sigma\ \in\ \omega}{\mathcal{WS}(\phi,\psi,\omega,v,\sigma)} \quad [\mathcal{WS}.4]$$

$$\frac{f : \rightarrow \sigma\ \in\ \psi}{\mathcal{WS}(\phi,\psi,\omega,f,\sigma)} \quad [\mathcal{WS}.5]$$

$$\frac{\begin{array}{c} f : \sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma_0\ \in\ \psi \\ \wedge\ \mathcal{WS}(\phi,\psi,\omega,t_1,\sigma_1)\ \wedge\ \cdots\ \wedge\ \mathcal{WS}(\phi,\psi,\omega,t_n,\sigma_n) \end{array}}{\mathcal{WS}(\phi,\psi,\omega,f(t_1,\ldots,t_n),\sigma_0)} \quad [\mathcal{WS}.6]$$

Fig. 3. Well-sortedness of rule-based programs

Type-driven meta-programming The many-sorted type system is not just meant for type safety. Our evolutionary approach employs sorts (and modes) as abstract means to identify relevant program fragments. For instance, recall the operator `newtype/2` from Sec. 2.4. An application of it, as in `newtype(val,num)`, collectively addresses all terms of sort `val`, which can occur as arguments of assertions or as subterms. Addressing all these locations without reference to a sort, say by enumerating positions, is clearly less adequate. As another example, consider state passing. The positions for state passing are again identified by means of sorts (cf. the operator `thread/1` in Sec. 2.2). This example also illustrates the role of modes in addressing predicate positions because threading distinguishes input and output positions of the same sort. The use of sorts (and modes) as addressing mechanisms suggests that we need other type-related properties in addition to just well-sortedness. For instance:

- The couple of a predicate p, a sort σ, and a mode m is said to *uniquely select a position* in a given program s, if p has exactly one position of mode m and sort σ in s. This property states that we can use documentary sorts (and modes) to address positions rather than the indexes of the positions themselves.
- If a predicate meets unique position selection for all of its positions then we achieved what we call *position neutrality*. This is a desirable property because it indeed allows us to abstract from the order of arguments. We can rather identify arguments using their sorts (and modes). One can even define structural equality *modulo reordering* positions of position-neutral predicates. [6]
- One can also define properties that are biased towards certain evolution operators. These properties can then often directly be used as preconditions of the operators. A prominent example is *threading fitness* (cf. the operator `thread/1` in Sec. 2.2). That is, a given rule is said to be fit for threading of sort σ if all of its assertions carry at most one input position of sort σ, and an optional output position of sort σ. Without this property, threading would have to be defined somewhat arbitrarily.

Type-based evolution relations We say that s and s' are of the same type, denoted by $s \doteq s'$, if $\Phi(s) = \Phi(s')$ and $\Psi(s) = \Psi(s')$. Type equality is clearly a much more liberal relation than structural equality. Nevertheless, knowing that some or all types are preserved by a transformation is useful information. There are now again several ways to restrict type equality:

- We can split the relation "$\doteq$". Equality of predicate types is denoted by "$\doteq_\Phi$", and equality of functor types is denoted by "$\doteq_\Psi$". We might further restrict claims about type equality by the means that we discussed for structural equality. That is, we use $s \doteq_{/X} s'$ to state type equality for all predicate symbols and functors in X, alike for the dual situation "$\doteq_{/\overline{X}}$".
- We might face the situation that s' simply involves more types than s, or vice versa. Hence, we also consider a relation $s \lessdot s'$ in case $\Phi(s) \subset \Phi(s')$ and $\Psi(s) \subset \Psi(s')$. This relation is liberal in that it allows for adding predicate symbols or functors, but s' does not revise types from s.
- Generally, the union and intersection operations on relations are conveniently used to form more relations. For example, the union of "$\doteq$" and "$\lessdot$" naturally results in "$\leq$". Also, the intersection of "$\doteq_\Phi$" and "$\lessdot_\Psi$" characterises progress at the data level alone, i.e., functors are added, while the types of predicates are completely preserved.
- We use "$\widehat{\doteq}$" to denote shared type equality. This relation expresses that s and s' do not disagree on types of shared functors and predicate symbols.

[6] Attribute grammars in the normal Knuthian notation are position-neutral by definition — if we view the attribute names as sorts. In *REK* [REK04], we support type aliases to make fine type distinctions so that position neutrality can be accommodated.

Well-modedness of rule-based programs $\boxed{\mathcal{WM}(s)}$

$$\frac{\mathcal{WM}(\phi, r_1) \ \wedge \ \cdots \ \wedge \ \mathcal{WM}(\phi, r_n)}{\mathcal{WM}(\phi; \psi \rhd r_1 \ldots r_n)} \quad [\mathcal{WM}.1]$$

Well-modedness of rules $\boxed{\mathcal{WM}(r)}$

$$\frac{\mathcal{USED}(\phi, r) \subseteq \mathcal{DEFINED}(\phi, r)}{\mathcal{WM}(\phi, r)} \quad [\mathcal{WM}.2]$$

Variables on defining, using, input, output positions

$$\begin{aligned}
\mathcal{USED}(\phi, \omega \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_n) &= \mathcal{IO}(\phi, a_1, +) \cup \cdots \cup \mathcal{IO}(\phi, a_n, +) \cup \mathcal{IO}(\phi, a_0, -) \\
\mathcal{DEFINED}(\phi, \omega \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_n) &= \mathcal{IO}(\phi, a_0, +) \cup \mathcal{IO}(\phi, a_1, -) \cup \cdots \cup \mathcal{IO}(\phi, a_n, -) \\
\mathcal{IO}(\phi, s(t_1, \ldots, t_n), \mu) &= \mathcal{IO}'(t_1, \mu_1, \mu) \cup \cdots \cup \mathcal{IO}'(t_n, \mu_n, \mu) \\
&\quad \text{where } p : \mu_1 \, \sigma_1 \times \cdots \times \mu_n \, \sigma_n \ \in \ \phi \\
\mathcal{IO}'(t, \mu, \mu) &= \mathcal{VARS}(t) \\
\mathcal{IO}'(t, ?, \mu) &= \mathcal{VARS}(t) \\
\mathcal{IO}'(t, +, -) &= \emptyset \\
\mathcal{IO}'(t, -, +) &= \emptyset
\end{aligned}$$

Fig. 4. Well-modedness of rule-based programs ($\mathcal{VARS}(t)$ denotes the variables in a term)

3.3 The link between modes and evolution

Next to well-sortedness, we consider well-modedness, i.e., the occurrences of variables in rules meet some data-flow criteria that are in accordance with the available mode information. In declarative programming, it is not uncommon to summarise well-sortedness and well-modedness as well-typedness or type correctness. We identify these separate notions because they interact with evolution differently. That is, we are willing to abandon well-modedness for intermediate results, while well-sortedness is considered holy. We will justify this liberalism below.

Rule-based programming languages favour different notions of well-modedness, but in our experience the following definition corresponds to a least upper bound of what is normally required. We provide modes "$+$" (input) and "$-$" (output) for predicate positions. The intuition is that predicates compute output positions from input positions. On the basis of these modes, we also categorise positions in rules. Namely, there are using and defining positions:

Using positions	Defining positions
Input positions of any premise	Input positions of the conclusion
Output positions of the conclusion	Output positions of any premise

We say that a rule-based program is *well moded* if for all rules it holds that each variable on a using position also occurs somewhere on a defining position. This is formalised in Fig. 4.

Modes in logic programming Modes provide a means to describe the intended ways of calling predicates and the user's intention of how the program behaves when called as prescribed [AL94, Boy96, BM97, Jef02]. The modes "$+$" and "$-$" can be interpreted to mean ground or instantiated arguments when calling ("$+$") vs. on return ("$-$"); see call-correctness in [Boy96, BM97]. When we assume Prolog's standard computation rule, then call-correctness requires left-to-right instantiation in the body, which is a stronger well-modedness criterion than the one defined in Fig. 4. Other forms of call-correctness apply when mode-directed evaluation is performed, e.g., in the sense of constraint or delay mechanisms. One might even require well-modedness to include static resolution of the occur check. The above formalisation of well-modedness abstracts from the operational order of premises; it is a relaxed form of I/O correctness in [Boy96, BM97].

Well-modedness elsewhere Left-to-right data flow among the conditions is required in ASF (Algebraic Specification Formalism [BHK89]) — a programming language for conditional term rewriting. That is, variables are bound in the order of the conditions (say, premises), and only one side of a condition is allowed to involve unbound variables. In RML (Relational Meta-Language [Pet94]) — a programming language for Natural Semantics, the same kind of left-to-right data flow is required for the premises modulo the notion of 'unknowns'. That is, a variable can also get bound (to an unknown value) by a dedicated rule annotation. Our definition of well-modedness agrees with the notion of a well-formed attribute grammar, which requires that the semantic equations associated to a given production must define each synthesised attribute on the left-hand side and each inherited attribute on the right-hand side [Knu68, WM77]. Furthermore, non-circularity is often required for the attribute dependencies. More restricted schemes of attribute dependencies exist. For instance, left-to-right data flow is known under the name L-attribution.

Loose well-modedness Our evolutionary approach admits undefined and unused variables — at least for intermediate results arising during evolution. The sets of undefined and unused variables are computed from the sets of using and defining occurrences as follows:

$$\begin{aligned}\mathcal{UNDEFINED}(\phi, r) &= \mathcal{USED}(\phi, r) \setminus \mathcal{DEFINED}(\phi, r)\\ \mathcal{UNUSED}(\phi, r) &= \mathcal{DEFINED}(\phi, r) \setminus \mathcal{USED}(\phi, r)\end{aligned}$$

Used variables lacking definitions violate well-modedness. Defined variables lacking uses are not in conflict with the defined well-modedness notion, but they could still indicate problems related to name confusion or an incompletely defined data

flow.[7] We do not ban undefined and unused variables because they serve a purpose during evolution:

- When adding new positions to predicates, then distinct fresh variables are chosen for all the relevant locations in conclusions and premises. This will obviously create undefined and unused variables. This is desirable because the variables can still be constrained to participate in different schemes of data flow. In Sec. 2, we demonstrated this by adding positions for stores, environments and others using the operator `add/2`.
- If the data flow in a rule is considered inappropriate, then evolutionary transformations can essentially perform the inverse of substitution such that variable distinctions are realised. This will create undefined and unused variables, which are in turn available for establishing a revised scheme of data flow. In Sec. 2.7, we demonstrated this by refreshing positions for stores using the operator `refresh/2`.

Mode-based evolution relations We need relations that state, in some sense, how two programs relate with regard to the amount of undefined or unused variables. Given a rule-based program s, we denote the sets of sorts for which predicate positions with undefined or unused variables exist by $undefinedness(s)$ and $unusedness(s)$. Then, s and s' are undefinedness-wise equal, denoted by $s \overset{\text{UNDEF}}{=} s'$, if $undefinedness(s) = undefinedness(s')$. Likewise, the programs s and s' are unusedness-wise equal, denoted by $s \overset{\text{UNUSED}}{=} s'$, if $unusedness(s) = unusedness(s')$. If an evolution step adds or removes undefined or unused variables, we will rather need relations "$\overset{\text{UNDEF}}{<}$", "$\overset{\text{UNDEF}}{>}$", etc. We can also provide some forms of restricted claims, e.g., "$\overset{\text{UNDEF}}{=}{}_{/X}$" expresses that undefinedness-wise equality is restricted to the sorts X. In Sec. 5, we will derive some high-level operators that are actually driven by (positions with) undefined and unused variables. That is, such operators iterate over predicate positions in rules and perform basic adaptations depending on the status of a position to contain undefined or unused variables. This can be viewed as *undefinedness- or unusedness-driven transformation*, which is a sophistication of type-driven transformation.

3.4 Semantics-based evolution relations

The most disciplined evolution scenarios are such that the original program and the adapted program have the same semantic meaning. This can be relaxed in a number of ways, while still offering meaningful constraints for evolution. We will now compile a list of corresponding semantics-based relations on rule-based programs, while we consider the standard fixpoint semantics of logic programs [VK76,

[7] Unused variables are somewhat similar to Prolog's singletons. Prolog systems tend to produce 'singleton warnings', in which case the programmer is encouraged to rename singletons to "_" so that the revealed status of an irrelevant binding is pointed out explicitly.

Llo87, Apt90] as our reference semantics. This is based on our working assumption that most rule-based notations can be mapped quite directly to definite clause programs, or at least to more general logic programs with extensions such as cut and negation — for which accordingly enhanced semantics exist [vRS91, And03].

The semantics of a program s, denoted by $[\![s]\!]$, is the set of all ground consequences that can be derived from $ground$(s), which comprises the set of all ground and well-sorted instances of the rules of s. This semantics is expressed as the following fixpoint of the intermediate consequence operator T_s:

$$[\![s]\!] = \bigcup_{n=0,1,\ldots} T_s^n(\emptyset)$$

The operator T_s is defined as a function on ground assertions as usual:

$$a_0 \in T_s(\mathcal{I}) \Leftrightarrow \exists a_1, \ldots, a_n.\ a_0 \Leftarrow a_1, \ldots, a_n \in ground(s) \wedge \{a_1, \ldots, a_n\} \subseteq \mathcal{I}$$

(The first layer $T_s(\emptyset)$ contributes all ground, well-sorted instances of facts in s. The second layer $T_s^2(\emptyset)$ contributes all immediate consequences derivable from $T_s(\emptyset)$, i.e., the heads of $ground$(s) with premises in $T_s(\emptyset)$. And so on ...)

Set-theoretic, semantics-based evolution relations The rule-based programs s and s' are semantically equal, denoted by $s \cong s'$, if $[\![s]\!] = [\![s']\!]$. We require $s \doteq s'$ before we wonder whether $s \cong s'$ holds. Otherwise we were comparing apples and oranges. There are several ways to relax and complement equality:

- Because meanings are sets we can immediately use the subset relation in forming semantics-based relations. That is, s is semantically smaller (or greater resp.) than s', denoted by $s \tilde{<} s'$ (or $s \tilde{>} s'$ resp.), if $[\![s]\!] \subset [\![s']\!]$ (or $[\![s]\!] \supset [\![s']\!]$ resp.). Assuming that operational and declarative semantics are aligned, these relations model that the adapted program s' produces more answers (or fewer answers resp.) than the original program s.
- Semantical equality can be restricted in all the ways that we have previously encountered at the structural level, or the type level. For instance:
  - $\cong_{/X}$: We only consider ground assertions for certain predicates X.
  - $\cong_{/\overline{X}}$: We consider ground assertions for all but certain predicates X.
  - $\widetilde{\cong}$: We consider ground assertions for shared predicate symbols only.

These various means help us to quantify the parts of the program that were not changed more arbitrarily. We can also hide helper predicates in this way.

Equality modulo renaming We already mentioned that structural equality can be relaxed to hold modulo a renaming ρ. This is clearly also sensible for semantic equality (and type equality). A renaming is specified by mapping predicate symbols p_i and functors f_j to fresh, distinct symbols such as in $p_1 \mapsto p'_1, \ldots, p_m \mapsto p'_m$,

$f_1 \mapsto f_1'$, ..., $f_n \mapsto f_n'$. Given such a mapping ρ, we overload ρ to also denote a function on ground assertions. Then, we can define that s and s' are semantically equal modulo renaming ρ, denoted by $s \cong_\rho s'$, if $[\![s']\!] = \{\rho(a) \mid a \in [\![s]\!]\}$. Note that this definition is more convenient than structural equality modulo renaming because the latter would amount already to the definition of a (simple) structural transformation on rules.

Relations modulo mappings Equality modulo renaming is just a concrete instance of the more general concept that semantic meanings are related modulo some mapping. That is, s is semantically greater than s' modulo a surjective mapping m, denoted by $s \tilde{\geq}_m s'$, if $[\![s']\!] = \{m(a) \mid a \in [\![s]\!]\}$. That is, $[\![s]\!]$ is mapped onto $[\![s']\!]$ in a many-to-one manner. This mapping is a very versatile provision to relate semantic meanings. For instance, the mapping can project away some arguments. There is also the inverse situation, where the roles of s and s' are reversed in the definition, i.e., s is semantically smaller than s' modulo a surjective mapping m, denoted by $s \tilde{\leq}_m s'$. If m is even a bijection, then we have a proper equality property. We say that s and s' are semantically equal modulo (forward) bijection m, denoted by $s \cong_m s'$, if the bijection m goes from $[\![s]\!]$ to $[\![s']\!]$.

Renaming is an example for semantical equality modulo bijection. An example of "semantically greater" modulo surjection is when we relate s and s' by a kind of projection to reflect that the predicates in s carry extra arguments compared to the corresponding predicates in s'. To this end, m is defined to transform ground assertions in $[\![s]\!]$ such that the extra arguments are omitted. This sort of projection can easily be specified as follows. Given a rule-based program s, for each predicate symbol $p : \mu_1\, \sigma_1 \times \cdots \mu_n\, \sigma_n$, the specification of the projection comprises a corresponding index list, which is of the form $p : \langle k_1, \ldots, k_q \rangle$ with $1 \leq k_1 < \cdots < k_q \leq n$.

3.5 Contribution problems in programs

Reasoning about semantics preservation or relaxations thereof is complicated by the fact that evolving programs can potentially involve what we call contribution problems. Given a program s, these kinds of problems can be distinguished:

- A predicate symbol p that is used (or at least its type is declared) in s lacks a definition, which implies that it will be non-productive in a trivial sense, i.e., the set of assertions for p in $[\![s]\!]$, denoted by $[\![s]\!]_{/p}$, is the empty set.
- A given rule lacks a semantical contribution to s, if there is no ground instance of the rule's head that would be a consequence of $\bigcup_{i=0,1,\ldots,n} T_s^i(\emptyset)$ for some n. The rule could as well be removed without affecting the semantic meaning of s.
- Any use of non-productive predicates within the body of a rule immediately implies that this rule must be non-contributing. By transitive closure, this can

lead to empty semantic meanings for many or all predicates.

- When all rules for p lack a contribution, e.g., when p lacks a valid base case, then p is non-productive. No matter how 'meaningful' the structural description of p, it could as well be removed without affecting the semantic meaning of s.
- Contribution problems also concern sorts and functors. There can be undefined sorts, non-productive functors, or sorts without a base case. Rules with argument positions of non-contributing sorts, are necessarily non-contributing.
- Lack of contribution also concerns unreachability of predicates with regard to a set of distinguished root predicates. We say that p' is *reachable* from p if p' is used in the body of a rule defining p, or if it used in a rule defining another predicate p'' for which it holds that p'' is reachable from p.[8] Sorts can be unreachable as well — simply because they are not used by any (reachable) predicate.

These problems lead to somewhat uninformative semantic meanings. That is, meanings of predicates tend to be empty all too easily, and rules that are maybe meaningful on their own tend to be unrepresented just because of non-contributing predicates that are used. Consider the following definite clause program:

```
p(h(X),Z) :- q(X,Y), r(Y,Z), s(Z).
q(a,b).
r(c,d).
```

The predicate `s` is not defined, and hence trivially non-productive. The single rule for `p` is non-contributing because `s` is non-productive, but also because the conjunction `q(X,Y)`, `r(Y,Z)` is not feasible for the given definitions of `q` and `r`.

Inevitability of contribution problems We want to substantiate that contribution problems are to be expected. Depending on the rule-based language at hand, contribution problems are not fully amenable to automatic detection. For instance, non-productivity of a predicate in a definite clause program is generally undecidable. As a counterexample, attribute grammars without conditions can be checked, in principle, to be fully productive. No matter whether contribution problems are uncovered by the programmer or by a static check, the necessary code adaptations call for evolutionary transformations. To reason about these transformations calls for a semantic model that is meaningful for programs *with* contribution problems.

Furthermore, the stepwise nature of evolutionary transformations can naturally involve intermediate results with contribution problems. For instance, this is the case when clean-up steps precede enhancement steps:

[8] Contribution problems for rule-based programs can be seen as a generalisation of reducedness problems for context-free grammars. Recall: a context-free grammar is reduced if all nonterminals are both reachable (from the start symbol) and productive. A nonterminal is productive, if it generates strings (be it the empty string).

- Replacement of an inappropriate predicate consists of discarding the definition, and adding the appropriate one. The intermediate result is likely to gain in both productivity and reachability problems. The continuous transformation of the inappropriate definition into the appropriate one is often not tractable.
- When the base case of a predicate or sort is considered inappropriate, then the base case will be removed prior to adding a new case. Keeping the inappropriate base case until the alternative, appropriate base case has been added does not seem to be aligned with the intuition of programmers.

Consequently, while reachability and, even more, productivity is desirable for perfect programs, we cannot insist on these properties for evolving or emerging programs. We rather would like to improve the semantic model for rule-based programs such that contribution problems do not spoil the intuitive expectations from relations on semantic meanings. This is comparable to the challenge of making the semantics of logic programs *with negation* compliant with common sense [vRS91].

Relevant evolution relations Before we discuss adjustment of the semantic model, we indicate that productivity and reachability again naturally imply corresponding evolution relations. The programs s and s' are productivity-wise equal, denoted by $s \overset{\text{PROD}}{=} s'$, if for all $p \in preds(s) \cup preds(s')$ it holds that $[\![s]\!]_{/p} = \emptyset \Leftrightarrow [\![s']\!]_{/p} = \emptyset$. Likewise, we can define productivity-wise smaller, i.e., in p', more predicates are productive, and greater alike. We can also quantify equality and others by enumerating predicate symbols, rules, sorts, and functors explicitly. Finally, we can also define reachability-wise equality, etc.

We will now consider a *program completion*, which potentially increases the number of contributing rules and productive predicates. Given a program s, the completed program is denoted by $s{\uparrow}$. Unless stated otherwise, we will assume in the sequel that claims about semantics preservation refer to completed programs.

Program completion The overall idea of completion is that predicates are given 'more room to succeed'. Thereby, non-contributing rules can be potentially turned into contributing ones. The extra room for the rule's premises allows the rule to succeed where it previously could only fail.[9] A basic scheme for completion is the following. For each sort σ in the original program, we assume new functors $c_\sigma : \rightarrow \sigma$, and $f_\sigma : \sigma \rightarrow \sigma$. These functors suffice to construct an infinite number of new terms of sort σ. Furthermore, we assume a fresh predicate p_σ that is defined as follows:

[9] The proposed method is similar to the method in [vRS91], where programs with unsafe negation are given room to fail. The proposed method can be seen as a generalisation of our method described in [Läm01], where non-productive nonterminals of a context-free grammar are made productive.

$$\emptyset \quad \rhd \quad p_\sigma(c_\sigma) \quad \Leftarrow \mathit{true}$$
$$\{v : \sigma\} \quad \rhd \quad p_\sigma(f_\sigma(v)) \Leftarrow p_\sigma(v)$$

The extension of this predicate is precisely the set of all the new terms of sort σ. Finally, for each predicate symbol p in the original program, we add a rule of the following form:

$$\{v_1 : \sigma_1, \ldots, v_n : \sigma_n\} \quad \rhd \quad p(v_1, \ldots, v_n) \Leftarrow p_{\sigma_1}(v_1) \wedge \cdots \wedge p_{\sigma_n}(v_n)$$

Here, the arity of p is n, and the sorts of its arguments are σ_1, ..., σ_n. This completion provides each predicate with a guaranteed base case whose extension is an unbounded number of assertions involving only new terms. Recall the earlier example with predicates `p`, `q`, `r`, and `s`. The completed program recovers productivity; in particular, the rule for `p` becomes contributing.

The proposed completion is rather conservative. One can easily construct examples of rule-based programs where non-contributing rules remain all too easily. The main limitation is that premises with non-variable terms cannot possibly succeed in the room provided by the extra base cases with their distinguished terms. This limitation can be remedied by a simple folding transformation such that all premises with non-variable terms are extracted as auxiliary predicates, which again are given extra room during completion.

To conclude, completion ensures that more or even all rules are represented when deriving consequences. Hence, the semantic meanings of predicates change whenever rules are added or removed — as one would expect intuitively. This makes reasoning about comparison relations on semantic meanings more informative — as opposed to a situation, where we would compare empty semantic meanings.

4 Primitive transformation operators

We will now systematically develop a suite of primitive operators for the transformation of rule-based programs in the course of evolution. The operators are of a syntactical nature because of the assumed editing-like intuition. The operators are constrained by preconditions that can be automatically checked. We also provide transformation properties for all the operators, which can be viewed as correctness criteria for an implementation. The definition of the operators themselves is given in a semi-formal style here. Before we start developing the various operators, we will first discuss the principles underlying the design of the suite.

Primitive vs. derived operators We aim at primitives that are (ideally) sufficient to represent any kind of evolutionary transformation either directly or by means of composition. There are more detailed requirements:

- The primitives inhabit orthogonal (i.e., non-overlapping) roles.
- The primitives meet informative preservation properties.
- One can specify the inverse of each transformation.
- The primitives operate in scopes that are as restricted as possible.
- Transformation including checking conditions must be fully automated.

A typical example of a primitive is adding a new argument position to a predicate. A typical example of a derived operator is threading because this operation can be taken apart quite naturally to be performed as a series of primitive unification steps, which can be determined on a per-rule basis.

Well-sortedness and well-modedness preservation A transformation $\Delta : S \to S$ is well-sortedness-preserving if for all $s \in S$ with $\mathcal{WS}(s)$ such that $\Delta(s)$ is defined it holds that $\mathcal{WS}(\Delta(s))$. Applying the same scheme, we can also define well-modedness preservation. From here on, we generally assume that transformations are only applied to well-sorted inputs and that they always produce well-sorted outputs. That is, well-sortedness is an invariant for transformation. We can think of this assumption as an implicit precondition about well-sortedness, and an implicit postcondition about well-sortedness, both attached to each transformation operator. By contrast, well-modedness preservation is not an absolute requirement.

Transformation properties The previously defined evolution relations can be turned into transformation properties very directly. That is, given a binary relation $\circ$, we construct a corresponding preservation property $\chi_\circ$ as follows: A transformation $\Delta : S \to S$ obeys property $\chi_\circ$, if for all $s \in S$ such that $\Delta(s)$ is defined it holds that $s \circ \Delta(s)$. The following transformation properties are all obtained in this systematic manner:

- Structure preservation (use relation "$=$").
- Type preservation (use relation "$\doteq$").
- Φ- and Ψ-preservation (use relations "$\overset{\Phi}{=}$" and "$\overset{\Psi}{=}$").
- Undefinedness/unusedness preservation (use relations "$\overset{\text{UNDEF}}{=}$" and "$\overset{\text{UNUSED}}{=}$").
- Semantics preservation (use relation "$\simeq$").
- Productivity preservation (use relation "$\overset{\text{PROD}}{=}$")
- Limited forms of preservation (use notation "$\cdots/_X$").
- Preservation with exceptions (use notation "$\cdots/_{\overline{X}}$").
- Forms of shared preservation (use relations "$\widehat{=}$", "$\widehat{\doteq}$", "$\widehat{\simeq}$", ...).
- Strictly increasing and decreasing semantics (use relations "$\widetilde{<}$" and "$\widetilde{>}$").
- Increasing and decreasing semantics (use relations "$\widetilde{\leq}$" and "$\widetilde{\geq}$").
- Alike for the various other increasing and decreasing properties.

An abstract syntactical categorisation Rather than suggesting operators that refer directly to the structure of rule-based programs, we would like to provide a less language-dependent principle for defining evolution primitives. To this end,

we have reviewed other operator suites for program transformation — not just for rule-based programs, but also for syntax definitions, object-oriented languages, or higher-order functional languages [PP96, Opd92, Fow99, LRL00, Läm01, KL03]. As a result, we propose an abstract categorisation of syntactical structures:

- Abstraction: named forms of abstraction including their application.
- Reference: entities that serve as references in scopes.
- Parameter: entities that can be specialised and generalised.
- Aggregate: entities that can aggregate parts or alternatives.
- Composite: aggregates that allow for grouping.

For each of these categories, there are dedicated operators, as we will discuss below. In the following, we will relate the specific syntactical domains of rule-based programs to the abstract categories. We note that this categorisation is meant to be meaningful for most programming languages, but this will not be substantiated in the present paper.

4.1 The category *abstraction*

By abstraction we mean some form of declaration that allows for repeated use on the basis of an associated application form. In the case of rule-based programs, the foremost abstraction form is a group of rules that defines a predicate. These are the operators for the category *abstraction*:

- eliminate: Safe removal of an unused abstraction.
- introduce: Insertion of a new abstraction.
- duplicate: Duplicate an abstraction.
- unfold: Replace an application of an abstraction by its body.
- fold: Reverse the effect of unfolding.
- move: Move an abstraction from one scope to the other. [10]

Type abstractions Types in rule-based programs also provide an instance of the category abstraction. We have encountered folding for type aliases in Sec. 2.7 as part of a scenario where we grouped argument positions. That is, several predicate positions were turned into a single position of a suitable product type. Folding was involved in so far that the tuple type for the new predicate position was replaced by a corresponding type alias. For brevity, we omit the trivial definitions of the operators eliminate, introduce, duplicate, unfold, and fold. We refer to [KL03] for an operator suite that covers datatype-biased transformations.

Predicate abstractions We will now define all operators for predicates. Folding and unfolding is well-established for predicates (and functions) in the field of

[10] The operator move will not be instantiated because there are no scopes in our rule-based notation.

program calculation [BD77, PP96].

We use the following meta-variables in the sequel:

- s and s' — the rule-based programs before and after transformation.
- s and s' are overloaded to also denote the plain rules without the type contexts.
- $\Phi(s)$, $\Psi(s)$, $\Phi(s')$, $\Psi(s')$ denote the type contexts.
- We omit the definition of $\Phi(s')$ for $\Psi(s')$ for type-preserving transformations.

- eliminate(p) (Eliminate an unused predicate definition)
  - · Preconditions:
    - $s_{/p} \neq \emptyset$, i.e., p is defined.
    - $p \notin preds(s \setminus s_{/p})$, i.e., p does not occur in the rest of the program.
  - · Transformation: $s' = s \setminus s_{/p}$
  - · Properties:
    - Type preservation. [11]
    - Shared semantics preservation. [12]
    - Well-modedness preservation.
    - Undefinedness/unusedness preservation except for p. [13]

- introduce$(\{r_1, \ldots, r_n\})$ (Introduce a new predicate; add its definition)
  - · Preconditions:
    - $r_1, \ldots, r_n$ all refer to a common predicate p in the conclusion.
    - $p \notin preds(s)$, i.e., p does not occur in the rules.
    - $\mathcal{WS}(\Phi(s), \Psi(s), r_i)$ for $i = 1, \ldots, n$, i.e., the rules for p are well sorted. [14]
  - · Transformation: $s' = s \cup \{r_1, \ldots, r_n\}$
  - · Properties:
    - Type preservation.
    - Shared semantics preservation.
    - Well-modedness preservation subject to well-modedness of $\{r_1, \ldots, r_n\}$.
    - Undefinedness/unusedness preservation except for p.

- duplicate(p, p') (Duplicate a predicate definition)
  - · Preconditions:
    - $s_{/p} \neq \emptyset$, i.e., p is defined.
    - $p, p' : \mu_1\, \sigma_1 \times \cdots \times \mu_n\, \sigma_n \in \Phi(s)$ for some n, μ_i, and σ_i.

[11] The type of p can be eliminated by the operator eliminate for 'types as abstractions'.
[12] Semantics preservation generally implies productivity preservation.
[13] The eliminated rules could be the source of sorts with undefined/unused variables.
[14] Before introducing a predicate in terms of its rules, the predicate's *type* must be introduced.

- $p' \notin preds(s)$, i.e., p' is not defined, and not even used.

· Transformation: $s' = s \cup \{\omega \rhd g(a_0) \Leftarrow a_1 \wedge \cdots \wedge a_m | \, \omega \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_m \in s_{/p}\}$ where $g(p(t_1, \ldots, t_n)) = p'(t_1, \ldots, t_n)$

· Properties:
- Type preservation.
- Shared semantics preservation.
- Undefinedness/unusedness preservation. [15]

• unfold(r, k) (Unfold a predicate in the position of some premise)

· Preconditions:
- $r \in s$
- k-th premise of r refers to some p of arity n.

· Transformation: $s' = s \setminus \{r\} \cup \{r_1, \ldots, r_n\}$, where the r_i are rules obtained from r by all possible replacements as follows: The k-th premise $p(t_1, \ldots, t_n)$ in r is replaced by the body of some rule r' defining p while unifying the head of r' with $p(t_1, \ldots, t_n)$.

· Properties:
- Type preservation.
- Semantics preservation. [16]
- Well-modedness preservation. [17]

• fold(r, i, j, p) (Fold some premises that constitute a body of a predicate)

· Preconditions:
- $1 \leq i \leq j$
- $r \in s$, number of premises of r not smaller than j.
- $s_{/p} = \{\omega \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_{j-i+1}\}$ for some ω, a_0, …, a_{j-i+1}.

· Transformation: $s' = s \setminus \{r\} \cup \{r'\}$ where r' is obtained from r by unifying the i-th to j-th premises with a_1 to a_{j-1+1}, and replacing the premises by the instance of a_0. [18]

· Properties:
- Type preservation.
- Semantics preservation.
- Well-modedness preservation.

---

[15] Undefinedness preservation generally implies well-modedness preservation.

[16] To be precise, unfolding is semantics-decreasing for the basic completion procedure that we have described in Sec. 3.5. This is caused by the fact that the guaranteed base case for the unfolded predicate does not contribute to the rules resulting from unfolding. This is an intended property of the semantic model. In the case of folding, this implies an increase of semantics.

[17] The stronger property of undefinedness/unusedness preservation requires subtle side conditions.

[18] For detailed side conditions for fold/unfold transformations see [PP96]. We should also not that unfold and fold are not defined in a completely symmetric manner. In particular, the operator fold only affects a single rule, while the operator unfold inlines p in all possible ways.

4.2 The category *reference*

By reference we mean all kinds of names. We distinguish the categories *abstraction* and *reference* because one can think of references that do not come with an associated, named form of abstraction. In rule-based programs, there are references to predicates, sorts, and functors. One could also argue that variables in rules are like references, but the category *parameter* deals with variables in a more appropriate manner, as we will see. These are the operators that are associated with references:

- rename: Consistent replacement of references.
- redirect: Replace a reference such that it refers elsewhere.

Predicate references The operator redirect allows us to adapt a premise $p(\ldots)$ such that it refers to a different predicate p'. Let us assume that p' was previously obtained by duplication. Then, redirection will be semantics-preserving. By such point-wise redirection, a subsequent evolution of p' will only affect premises that were redirected to it.

- redirect(r, k, p, p') (Redirect a premise to a different predicate)
  - · Preconditions:
    - $r = \omega \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_n \in s$ for some ω, n, and a_i.
    - $1 \leq k \leq n$; the assertion a_k refers to p.
    - $p, p' : \mu_1\, \sigma_1 \times \cdots \times \mu_m\, \sigma_m \ \in \Phi(s)$ for some m, μ_i, and σ_i.
  - · Transformation: $s' = s \setminus \{r\} \cup \{\omega \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge g(a_k) \wedge \cdots \wedge a_n\}$ where $g(p(t_1, \ldots, t_m)) = p'(t_1, \ldots, t_m)$
  - · Properties:
    - Type preservation.
    - Semantics-preserving if p' is a duplicate of p.
    - Undefinedness/unusedness preservation.

Strictly speaking, we do not need to specify a rename operator for predicates because rename(p, p') can be derived in three steps: (1) duplicate p as p', (2) redirect p to p', and (3) eliminate p and its type.

Functor references Likewise, redirection and renaming can be provided for functors. When addressing functor occurrences for the purpose of redirection or otherwise, we need to employ a kind of path expression since we might need to address arbitrarily nested subterms.

4.3 The category *parameter*

By parameter we mean constructs that allow for some form of specialisation or generalisation. As for rule-based programs, terms together with normal substitution provide a form of parameterisation. That is, each variable is seen as a potential parameter that can be instantiated (say, specialised). Each non-variable term is seen as a potential target of generalisation. [19] There are two associated operators:

- specialise: Instantiate parameters.
- generalise: The inverse of specialise.

The operator for *threading*, which we illustrated several times in Sec. 2, is defined in terms of specialisation, i.e., substitution. The operator for *relaxation*, which we also illustrated in Sec. 2, is defined in terms of generalisation; recall the elimination of unnecessary matches and builds.

- $\mathsf{specialise}(r, \theta)$ (Apply a substitution to a rule)
  - · Preconditions:
    - $r \in s$; the head of r refers to some p.
    - $\mathsf{vars}(\theta) \subseteq \mathsf{vars}(r)$.
    - θ is well sorted.
  - · Transformation: $s' = s \setminus \{r\} \cup \{\theta(r)\}$
  - · Properties:
    - Type preservation.
    - Semantics-decreasing. [20]
    - Undefinedness-/unusedness-decreasing.

- $\mathsf{generalise}(r', \theta)$ (Replace a rule by a more general rule)
  - · Preconditions:
    - $\theta(r') \in s$; the head of r' refers to some p.
    - $\mathsf{vars}(\theta) \subseteq \mathsf{vars}(r')$.
    - θ is well sorted.
  - · Transformation: $s' = s \setminus \{\theta(r')\} \cup \{r'\}$
  - · Properties:
    - Type preservation.
    - Semantics-increasing.
    - Undefinedness-/unusedness-increasing.

---

[19] An example of parameterisation that is not related to rule-based programs is specialising or generalising class constraints in an object-oriented context [PS90a, TS00].
[20] Decrease (increase resp.) of semantics generally implies decrease (increase resp.) of productivity.

The status of these operators to be semantics-decreasing or semantics-increasing resp. follows directly from the fact that there are less or more ground instances of rules to be considered by the immediate consequence operator; cf. Sec. 3.4.

4.4 The category *aggregate*

By aggregate we mean syntactical forms that potentially comprise several components. For example, rules in a rule-based program can be viewed as components in the sense of alternatives. The various premises of a rule can be viewed as components in the sense of parts. The arguments in an assertion are like parts, too. These examples illustrate that there are two kinds of components, namely alternatives and parts. Accordingly, we can also identify two kinds of aggregates, namely unions and products. The evolution operators in this category serve for the addition and the removal of components. When the order of the components matters, an operator for reordering is needed as well. That is:

- add: Add a component to the aggregate.
- remove: Remove a component from the aggregate.
- permute: Permute the various components in the aggregate.

We used an operator for adding predicate positions all-through Sec. 2.

Predicate definitions as union-like aggregates We define operators to add rules to a program, and to remove rules from a program. We omit permutation of rules.

- add(r) (Add a rule)
  - Preconditions:
    - $r \notin s$; head of r refers to some p.
    - $p \in preds(s)$
    - $\mathcal{WS}(\Phi(s), \Psi(s), r)$, i.e., the new rule is well sorted.
  - Transformation: $s' = s \cup \{r\}$
  - Properties:
    - Type-preserving.
    - Semantics-increasing.
    - Undefinedness-/unusedness-preserving except for p.

- remove(r) (Remove a rule)
  - Preconditions: $r \in s$; head of r refers to some p.
  - Transformation: $s' = s \setminus \{r\}$
  - Properties:
    - Type-preserving.

- Semantics-decreasing.
- Undefinedness-/unusedness-preserving except for p.

Datatype declarations as union-like aggregates Likewise, aggregation operators can be accomplished for functors. Then, the add operator adds a functor to a sort. The remove operator removes a functor, to which end the functor must not be in use anymore in the rules. Permutation can be ignored since the order of functors in datatype declarations is usually irrelevant. For brevity, we omit the definitions of add and remove. Adding a functor is semantics-increasing because the functor allows for additional ground terms of the relevant sort; dually for removal of functors. The operators clearly meet undefinedness and unusedness preservation because rules are not modified at all.

Predicate type declarations as product-like aggregates Likewise, aggregation operators can be accomplished for predicate types, i.e., type declarations for a predicate can be added or removed. Addition is subject to the precondition that the predicate is not yet covered by the type context. Removal of the type of p is only feasible if p does not occur in the given program s, i.e., $p \notin preds(s)$. Both operators are obviously very much well behaved. Assuming a setup where predicate types are inferred, these transformations will be performed implicitly.

Bodies as product-like aggregates Bodies of rules can be viewed as aggregates for premises. We will now define the corresponding operators. For simplicity, we assume that the operators add and remove operate on the last premise of a given body. One can clearly apply permutation to become more flexible (specification omitted). We refer back to Sec. 2.7, where we illustrated addition of premises by means of an operator for *injection*. This is clearly a derived operator because its adds several premises at once in a convenient manner.

- $\mathsf{add}(\omega_+, a, r)$ (Add a premise to a rule)
  - · Preconditions:
    - $r = \omega \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_n \in s$ for some ω, n and a_i.
    - $\mathcal{WS}(\Phi(s), \Psi(s), \omega_+, a)$, i.e., the new premise a is well sorted.
    - $\omega \cup \omega_+$ is well formed.
  - · Transformation: $s' = s \setminus \{r\} \cup \{\omega \cup \omega_+ \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_n \wedge a\}$
  - · Properties:
    - Type-preserving.
    - Semantics-decreasing.

- Undefinedness/unusedness decreasing subject to side conditions. [21]

- $\mathsf{remove}(r)$ (Remove a premise from a rule)
  - · Preconditions: $r = \omega \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_n \in s$ for some ω, n, and a_i.
  - · Transformation: $s' = s \setminus \{r\} \cup \{\omega' \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_{n-1}\}$
    where ω' is obtained from ω by removing variables that only occurred in a_n.
  - · Properties:
    - Type-preserving.
    - Semantics-increasing.
    - Undefinedness/unusedness properties subject to subtle side conditions.

Predicate positions as product-like aggregates The operator add for adding predicate positions is a very frequent step of preparing enhancements. Then, dually, removal of predicate positions relates to the clean-up mode of evolution. We also consider an operator for permutation because *position neutrality* of predicate is not a strict requirement (cf. Sec. 3.2). In fact, permutation is clearly a refactoring.

- $\mathsf{add}(p, \mu, \sigma)$ (Add a predicate position)
  - · Preconditions: $p : \mu_1\, \sigma_1 \times \cdots \times \mu_n\, \sigma_n \ \in \Phi(s)$ for some n, μ_i, and σ_i.
  - · Transformation:
    $s' = \{\omega' \rhd g(a_0) \Leftarrow g(a_1) \wedge \cdots \wedge g(a_m) \mid \omega \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_m \in s\}$
    where ω' is ω extended by fresh variables that are added by g, and
    $$g(p_0(t_1, \ldots, t_m)) = \begin{cases} p_0(t_1, \ldots, t_m, v), & \text{for } p_0 = p, m = n, v \text{ is a fresh variable} \\ p_0(t_1, \ldots, t_m), & \text{for } p_0 \neq p \end{cases}$$
  - · Type context: $\Phi(s') = \Phi(s) \setminus \{p : \mu_1\, \sigma_1 \times \cdots \times \mu_n\, \sigma_n\} \cup \{p : \mu_1\, \sigma_1 \times \cdots \times \mu_n\, \sigma_n \times \mu\, \sigma\}$
    $\Psi(s') = \Psi(s)$
  - · Properties:
    - Type-preserving except for p.
    - Semantics-increasing modulo surjection (project away position $n + 1$).
    - Undefinedness-/unusedness-increasing for σ.
    - Undefinedness-/unusedness-preserving except for σ.

- $\mathsf{remove}(p)$ (Remove a predicate position)
  - · Preconditions: $p : \mu_1\, \sigma_1 \times \cdots \times \mu_n\, \sigma_n \ \in \Phi(s)$ for some n, μ_i, and σ_i.

[21] The input positions of a should be defined in r, its output positions should be used in r.

· Transformation:
$s' = \{\omega' \rhd g(a_0) \Leftarrow g(a_1) \wedge \cdots \wedge g(a_m) |\ \omega \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_m \in s\}$
where ω' is ω reduced by variables that were used in a_m only, and
$$g(p_0(t_1, \ldots, t_m)) = \begin{cases} p_0(t_1, \ldots, t_{m-1}), & \text{for } p_0 = p, m = n \\ p_0(t_1, \ldots, t_m), & \text{for } p_0 \neq p \end{cases}$$
· Type context: $\Phi(s') = \Phi(s) \setminus \{p : \mu_1\, \sigma_1 \times \cdots \times \mu_n\, \sigma_n\} \cup \{p : \mu_1\, \sigma_1 \times \cdots \times \mu_{n-1}\, \sigma_{n-1}\}$
$\Psi(s') = \Psi(s)$
· Properties:
- Type-preserving except for p.
- Increase and decrease of semantics are both possible. [22]
- Increase and decrease of undefinedness/unusedness are both possible.

- permute$(p, \langle i_1, \ldots, i_n \rangle)$ (Permute arguments of a predicate)
· Preconditions:
- $p : \mu_1\, \sigma_1 \times \cdots \times \mu_n\, \sigma_n\ \in \Phi(s)$ for some n, μ_i, and σ_i.
- $\langle i_1, \ldots, i_n \rangle$ is a permutation of $\langle 1, \ldots, n \rangle$.
· Transformation:
$s' = \{\omega \rhd g(a_0) \Leftarrow g(a_1) \wedge \cdots \wedge g(a_m) |\ \omega \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_m \in s\}$
where $g(p_0(t_1, \ldots, t_m)) = \begin{cases} p_0(t_{i_1}, \ldots, t_{i_n}), & \text{for } p_0 = p, m = n \\ p_0(t_1, \ldots, t_m), & \text{for } p_0 \neq p \end{cases}$
· Type context: $\Phi(s') = \Phi(s) \setminus \{p : \mu_1\, \sigma_1 \times \cdots \times \mu_n\, \sigma_n\} \cup \{p : \mu_{i_1}\, \sigma_{i_1} \times \cdots \times \mu_{i_n}\, \sigma_{i_n}\}$
$\Psi(s') = \Psi(s)$
· Properties:
- Type-preserving except for p.
- Semantics-preserving modulo bijection for permutation.
- Undefinedness-/unusedness-preserving.

4.5 The category *composite*

Composites provide an optional extension of aggregates. A composite is an aggregate that allows for grouping, i.e., components in the aggregate can be grouped to form a single component. In a rule-based program, one can wrap predicate positions as well as functor positions. The associated evolution operators serve for the introduction and the elimination of groups. That is:

- wrap: Wrap a number of components as a single component.
- unwrap: Unwrap a component so that subcomponents are inlined.

[22] A well-behaved scheme for removing predicate positions is that the affected positions do not share variables with other positions. In this case, the removal of the position is semantics-decreasing modulo surjection (i.e., project away position n).

In Sec. 2, we saw several examples of wrapping. Firstly, type aliases were turned into proper datatypes subject to wrapping with a unary functor; see Sec. 2.5. Secondly, argument positions were combined in tuples; see Sec. 2.7.

Predicate positions as composites The operators take a functor for wrapping predicate positions. The number of positions to be wrapped or unwrapped is determined by the arity of the given functor. For simplicity, we define the operators wrap and unwrap such that the wrapped term is located on the first predicate position.

- $\mathsf{wrap}(p, f)$ (Wrap predicate positions as term)
  - · Preconditions:
    - $p : \mu_1\,\sigma_1 \times \cdots \times \mu_n\,\sigma_n \in \Phi(s)$ for some n, μ_i, and σ_i.
    - $f : \sigma'_1 \times \cdots \times \sigma'_{n'} \to \sigma'_0 \in \Psi(s)$ for some n' and σ'_j.
    - $n \geq n'$, $\sigma_1 = \sigma'_1, \ldots, \sigma_{n'} = \sigma'_{n'}$, $\mu_1 = \cdots = \mu_{n'}$.
  - · Transformation: $s' = \{\omega \rhd g(a_0) \Leftarrow g(a_1) \wedge \cdots \wedge g(a_m) |\ \omega \rhd a_0 \Leftarrow a_1 \wedge \cdots \wedge a_m \in s\}$
    where $g(p_0(t_1, \ldots, t_m)) = \begin{cases} p_0(f(t_1, \ldots, t_{n'}), t_{n'+1}, \ldots, t_n), & \text{for } p_0 = p, m = n \\ p_0(t_1, \ldots, t_m), & \text{for } p_0 \neq p \end{cases}$
  - · Type context: $\Phi(s') = \Phi(s) \setminus \{p : \cdots\} \cup \{p : \mu_1\,\sigma'_0 \times \mu_{n'+1}\,\sigma_{n'+1} \times \cdots \times \mu_n\,\sigma_n\}$
    $\Psi(s') = \Psi(s)$
  - · Properties:
    - Type-preserving except for p.
    - Semantics-preserving modulo bijection for adding wrapper.
    - Undefinedness-/unusedness-preserving in a loose sense. [23]

- $\mathsf{unwrap}(p, f)$ (Unwrap predicate position by inlining subterms)
  - · Preconditions:
    - $p : \mu_1\,\sigma_1 \times \cdots \times \mu_n\,\sigma_n \in \Phi(s)$ for some n, μ_i, and σ_i.
    - $f : \sigma'_1 \times \cdots \times \sigma'_{n'} \to \sigma_1 \in \Psi(s)$ for some n' and σ'_j.
    - All assertions for p in s are of the form $s(f(\ldots), \ldots)$.
  - · Transformation:
    $s' = \{g(a_0) \Leftarrow g(a_1) \wedge \cdots \wedge g(a_m) |\ a_0 \Leftarrow a_1 \wedge \cdots \wedge a_m \in s\}$ where
    $g(p_0(t_1, \ldots, t_m)) = \begin{cases} p_0(t'_1, \ldots, t'_{n'}, t_2, \ldots, t_n), & \text{for } p_0 = p, m = n, t_1 = f(t'_1, \ldots, t'_{n'}) \\ p_0(t_1, \ldots, t_m), & \text{for } p_0 \neq p \\ \text{undefined}, & \text{otherwise} \end{cases}$
  - · Type context:
    $\Phi(s') = \Phi(s) \setminus \{p : \cdots\} \cup \{p : \mu_{i_1}\,\sigma'_1 \times \cdots \times \mu_{i_1}\,\sigma'_{n'} \times \mu_2\,\sigma_2 \times \cdots \times \mu_n\,\sigma_n\}$
    $\Psi(s') = \Psi(s)$

[23] Undefined and unused variables are precisely preserved, but undefinedness and unusedness is not preserved in the literate sense because wrapping can change sorts of positions.

· Properties:
  - Type-preserving except for p.
  - Semantics-preserving modulo bijection for removing wrapper.
  - Undefinedness-/unusedness-preserving in a loose sense.

Functor positions as aggregates and composites The preceding development for predicate positions can be adopted very systematically for functor positions, which is omitted for brevity. This concerns the operators of both the aggregate and the composite category.

5 Derived transformation operators

We will now derive a number of operators that support schemes of program adaptation. These higher-level operators can be employed for various evolution scenarios. The operators perform specific analyses to prepare the systematic application of primitive operators. The following operators will be discussed:

- define — add definitions for undefined variables.
- use — add uses for unused variables.
- thread — establish the threading scheme of data flow.
- synthesise — compute data in bottom-up manner.
- refresh — refresh data flow by placing fresh variables.

The derived operators will all be defined in a style such that they can be applied to a single rule, That is, operator applications are of the form $op(\ldots, i)$, where i identifies a rule. It is then straightforward to lift the operators to the level of a full program by simple means of iteration. The derived operators are all sort-driven. That is, operator applications are of the form $op(\sigma, \ldots)$, where the sort σ identifies argument positions or terms that are potentially affected by the adaptation.

The above list is not meant to be complete in any rigorous sense. One can certainly think of further schemes. *REK* [REK04] comprises a few more schemes, including the ones that we encountered in Sec. 2:

- relax — relax data flow by avoiding term matching and building.
- group — group predicate positions to form a position of a product type.
- big2small — turn big-step SOS into small-step SOS.

5.1 Adding definitions and uses

The following operators are meant to define undefined variables, or to use unused variables. This can be either accomplished by instantiating the variables to ground

terms, or by constraining them via additional premises.

Consider the following logic program fragment:

```
program(Decs,Stats,I,O) :-
  declarations(Decs),
  statements(Stats,I,O,Store0,Store1).
```

This clause contributes to an interpreter for an imperative language. The predicate `program/4` takes the abstract syntactical representations of declarations `Decs` and statements `Stats`, and it is meant to compute the program output `O` from the input `I`. The clause is unfinished in so far that the initial store `Store0`, which is needed for the execution of the statements, is not defined. Using an operator for adding definitions, we obtain the following clause:

```
program(Decs,Stats,I,O) :-
  declarations(Decs),
  statements(Stats,I,O,[],Store1).
```

Here we opted for instantiating undefined variables by ground terms, namely `Store0` was instantiated by the empty store `[]`. Alternatively, we could add a premise `nil(Store0)` for the purpose of defining the undefined input store. Then, the adapted clause looks as follows:

```
program(Decs,Stats,I,O) :-
  declarations(Decs),
  nil(Store0),
  statements(Stats,I,O,Store0,Store1).
```

Operators at a glance

define-1(σ, p, i) Add a premise for each predicate position of sort σ that involves undefined variables. The defining premise is built from the unary predicate symbol p and the term on the relevant predicate position.

define-2(σ, t, i) The argument t is a ground term of sort σ that is meant to act as the defining value. The operator affects terms t' that are of sort σ, and that involve undefined variables (of whatever sort). Each such t' is unified with t. Failure of unification implies failure of transformation.

use-1(σ, p, i) Add a premise for each predicate position of sort σ that involves unused variables. The using premise is built from the unary predicate symbol p and the term on the relevant predicate position.

use-2(σ, t, i) The argument t is a ground term of sort σ. The operator affects terms t' that are of sort σ, and that involve unused variables (of whatever sort). Each such t' is unified with t. Failure of unification implies failure of transformation.

The operator define-1(σ, p, i) **— details** The definitions of the operators are quite similar. Hence the description of one operator should suffice:

- Accumulation phase: We iterate over all terms from using argument positions of sort σ in rule i, while we only collect those terms that involve undefined variables. Let $t_1, \ldots, t_n$ be these collected terms.
- Transformation phase: Premises $p(t_1), \ldots, p(t_n)$ are added to the rule i. Here we assume that the type of the predicate p is $p : -\sigma$. The positioning of the added premises potentially matters — depending on the rule-based programming language at hand. A transformation for repositioning can be performed such that the language-specific well-modedness criterion is accommodated.

Preservation properties

- All of the above operators are semantics-decreasing. This is implied by the underlying primitive operators, i.e., the operators for inserting premises or specialising terms, which are semantics-decreasing by themselves.
- The define operators are undefinedness-decreasing; dually for the use operators. Let us now assume that affected positions hold only variables and not functor terms. Let us further assume that these variables do not occur on positions of other sorts. Let us finally assume that the operators are iterated over all rules. Then, the define operators are *strictly* undefinedness-increasing for σ, and undefinedness-preserving for all other sorts; dually for the use operators.

Recovery of semantics preservation It turns out that the operators can naturally be used in a semantics-preserving manner. To this end, let us assume a program to which we add positions of sort σ. This step, as such, is semantics-increasing because the added positions can be instantiated to all possible terms of sort σ. However, let us continue with transformations such that these new positions are defined and used. To this end, the operators define-1 and use-1 are to be employed. Then, the increase of semantics that is caused by adding positions and the decrease of semantics that is caused by adding definitions and uses compensate each other. Here, we assume that the definitions and uses are *unique*, by which mean that they only allow for a single value of sort σ. Hence, at the level of semantic meanings, we can project away uniquely constrained positions of sort σ using a bijective mapping, which implies semantics preservation.

5.2 The threading technique

In Sec. 2, we used the threading technique for environment and state passing. In Fig. 5, we visualise the thread-like data flow for environments and states. For simplicity, the figure is based on the assumption that all predicates participate in threading. The corresponding operator thread(σ, i) implements such data flow as shown

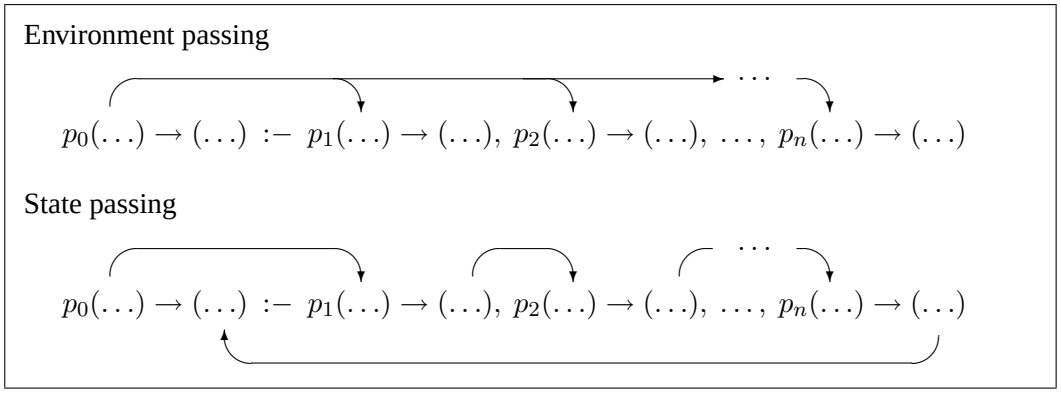

Fig. 5. Threading dependencies in a definite clause ("→" separates inputs and outputs)

in the figure by unifying positions that are connected by arrows. Threading leads to state passing in case there are input and output positions of the sort σ. Threading leads to environment passing in case there are only input positions of sort σ.

The threading order Threading is based on ordering positions in a rule:

- The input positions of the rule's head are the smallest positions.
- The input positions of a premise are smaller than its output positions.
- The order of the premises defines the order of positions from different premises.
- The output positions of the rule's head are the greatest positions.

Based on this order of positions, the operator $\mathsf{thread}(\sigma, i)$ unifies each using position of sort σ with the next smaller defining position of sort σ. Using this general rule of unifying positions, one can see that environment passing is indeed a degenerated form of state passing.

The operator $\mathsf{thread}(\sigma, i)$ **— details**

- Accumulation phase: all terms from argument positions of sort σ are collected. (These are normally variables in case the positions where added just before.) The collection is ordered in the way defined above, and each term is qualified by the information whether it was found on a defining or a using position.
- Iteration phase: We iterate over the collection in increasing order. Given a term on a using position, we unify this term with the next smaller term on a defining position. Failure of unification implies failure of transformation. To make unification less intrusive, i.e., to preserve existing data flow, using positions are only considered in case they involve undefined variables.
- Transformation phase: The various unifications were performed such that a single substitution was accumulated. This substitution is now applied to the rule i using the operator specialise.

Preservation properties

- The operator is semantics-decreasing (which is implied by specialisation).
- The operator is undefinedness- and unusedness-decreasing.
- If we assume that all affected positions take the form of variables, and these variables do not occur on positions of sorts other than σ, then the operator is undefinedness- and unusedness-preserving for all sorts except σ.

Recovery of semantics preservation Using similar arguments as we gave for the define and use operators, we can complete the threading technique into a semantics-preserving transformation. This is expressed by the following scenario:

- We require semantics preservation for a distinguished root predicate p_r.
- For simplicity, we assume that initially there are no predicate positions of sort σ.
- Input positions of sort σ are added to some set X of predicates, while $p_r \notin X$.
- Output positions of sort σ are added to some of the predicates in $X \cup \{p_r\}$.
- The new positions of sort σ are threaded (in all rules).
- Unique definitions are added for undefined positions of sort σ.

These steps can be supported by a dedicated operator that is parameterised by sets of predicates for reading or writing access to the threaded argument, and that uses a simple reachability analysis in the sense of the call graph to compute predicates that have to participate in threading [Läm99b, LR99]. The sequence of steps is both semantics-preserving and undefinedness-preserving. The point is basically that the threaded data structure is not yet used, but only passed on. Whenever a thread starts, then the corresponding using position is uniquely defined. So again threading plus adding unique definitions compensates for the increase of semantics caused by adding the positions for threading.

5.3 Bottom-up computation of data

A programming technique, which is often used in rule-based programs, is to compute data in a bottom-up manner — in the sense of the call graph of predicates. We did not illustrate this technique in Sec. 2, but it is readily implemented and illustrated in *REK* [REK04]. In Fig. 6, we show an attribute grammar, which is adopted from [LR99]. This rule-based program performs bottom-up computations to collect declared and used variables in an imperative program. To this end, there are synthesised attributes VDECS and VUSES. The various semantic rules create and combine sets of variables. In the end, we use the sets of declared and used variables to report on USELESS declarations; see [axiom]. Sets of variables are created with the semantic rules attached to productions [dec], [assign], and [var]; see the singleton sets $\{vdec.\mathsf{ID}\uparrow\}$ and $\{vuse.\mathsf{ID}\uparrow\}$. All the remaining semantic equations follow the scheme of the operator synthesise as described below. That is, the attributes from

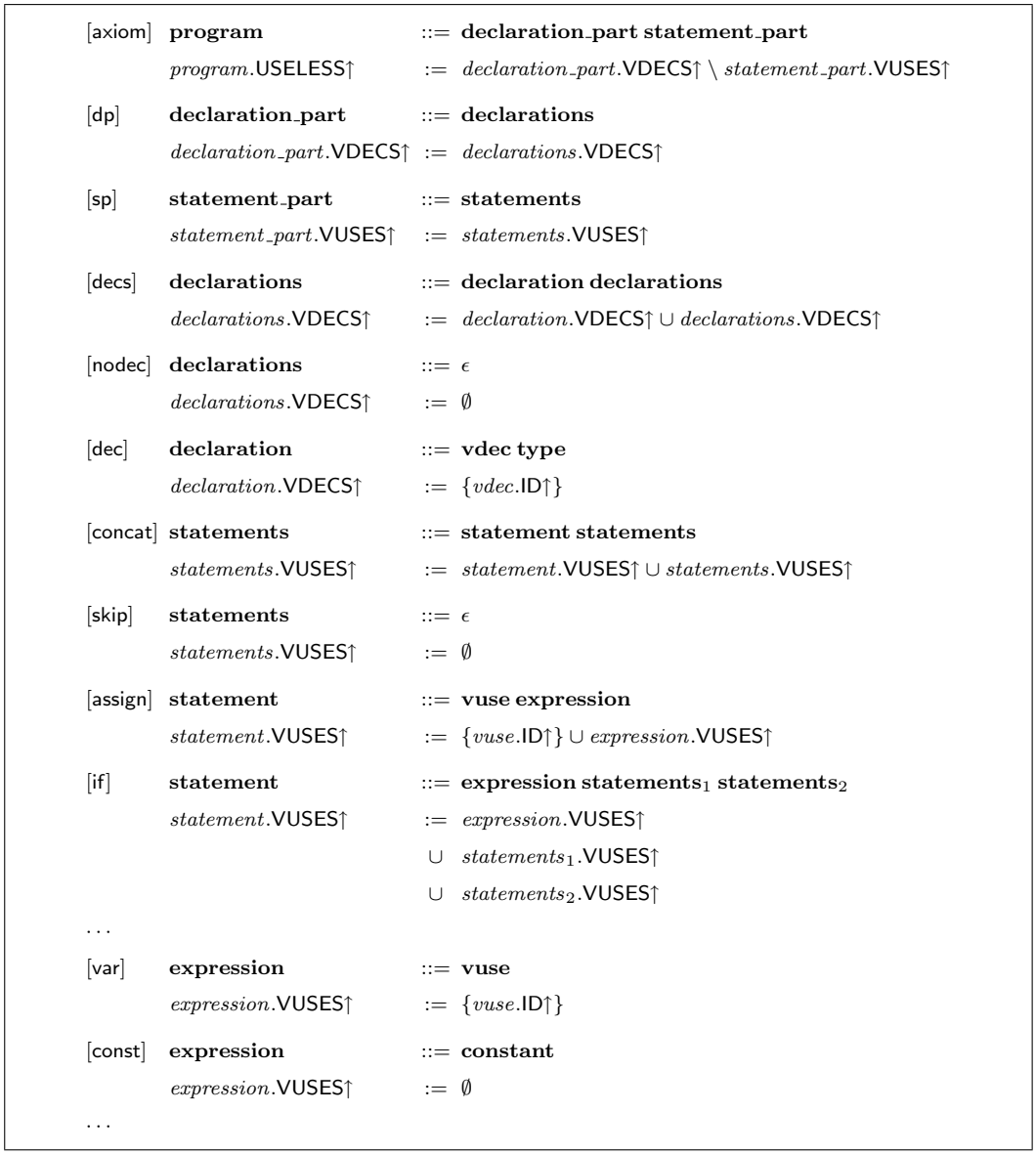

Fig. 6. An attribute grammar for detecting superfluous variable declarations

the right-hand side of the production are combined with a binary operator, namely "$\cup$" in this case, to compute the attribute on the left-hand side.

The operator synthesise(σ, o, i) **— details** The sort σ defines the sort of data to be computed in bottom-up manner. The argument o is the predicate symbol from which we will form premises for the combination of values of sort σ. The adaptation scheme is defined as follows:

- Trivial case: It is checked whether the head of i carries an output position of sort σ, and whether the corresponding term t_0 on this position involves undefined variables. If this check fails, the rule is preserved as is, and we stop here.

- Accumulation phase: All terms from the premises' output positions of sort σ are collected. We assume that the collection preserves the order of the premises. To make the operator synthesise idempotent, we only include terms that involve unused variables. Let $t_1, \ldots, t_n$ be these collected terms.
- Special case $n = 0$: The rule is preserved as is, and we stop here. (We might still resolve the undefined position in the head by using the operator define later.)
- Special case $n = 1$: t_1 and t_0 are unified. Failure of unification implies failure of transformation. The result of unification is applied to i using the operator specialise. We stop here.
- Otherwise, premises are added that compute t_0 from $t_1, \ldots, t_n$:
  - Variant 1: The type of the predicate for combination is $o : +\sigma \times +\sigma \times -\sigma$. We employ fresh variables $v_1, \ldots, v_{n-2}$ of sort σ. The premises $o(t_1, t_2, v_1)$, $o(v_1, t_3, v_2), \ldots, o(v_{n-2}, t_n, t_0)$ are added to the body of r.
  - Variant 2: The type of the predicate for combination is $o : +[\sigma] \times -\sigma$, where $[\sigma]$ denotes a list datatype. [24] A single premise $o([t_1, \ldots, t_n], t_0)$ is added to the body of r. The predicate o is meant to fold over the terms $t_1, \ldots, t_n$.

Preservation properties

- The operator is semantics-decreasing (which is implied by the added premise).
- The operator is undefinedness- and unusedness-decreasing.
- If we assume that all affected positions take the form of variables, and these variables do not occur on positions of sorts other than σ, then the operator is undefinedness- and unusedness-preserving for all sorts except σ.

Recovery of semantics preservation Using similar arguments as we gave for the other schemes of adaptation, we can complete bottom-up computation into a semantics-preserving transformation. We discuss this for variant 1:

- For simplicity, we assume that initially there are no predicate positions of sort σ.
- Output positions of sort σ are added to some predicates.
- The new positions of sort σ are combined using the operator synthesise.
- Unique definitions are added for undefined positions of sort σ.

Here we assume that the value for unique definition is an algebraic unit of the predicate o for combination, be it u. Then, the synthesised value is also necessarily u. Hence, we can project away u using a bijective mapping, which implies semantics preservation. So the idea is again that the initial introduction of the bottom-up scheme is semantics-preserving, while we might deviate from semantics preservation as soon as non-unit values contribute to the synthesis, e.g., by means of

[24] Strictly speaking, our core syntax for rule-based programs does not support polymorphic datatypes and predicates, but this would be a straightforward extension. In fact, *REK* [REK04] supports polymorphism.

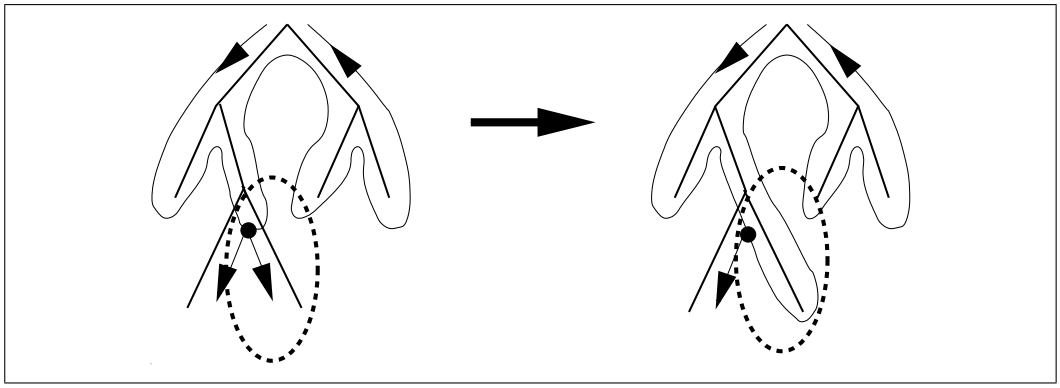

Fig. 7. Extending the scope of threading — visualisation in the call graph

additional rules. The steps listed above can be supported by a dedicated operator. This results in a meta-programming operator that can be viewed as a reconstruction of ELI's powerful *constituents* construct for remote access in attribute grammars [KW94, LR99].

5.4 Refreshment of data flow

The introductory example illustrated adding premises to an SOS rule and adjusting the flow of the store to incorporate the new premises. To this end, we used an operator for refreshment of data flow such that all store positions were temporarily unconnected and hence available for re-threading. Another evolution scenario for refreshment of data flow is illustrated in Fig. 7. We start from a program (cf. left part of the figure) that uses state passing for some parts of the program, while the state is only propagated downwards for other parts that are free of side effects. Now let us assume that the possibility of side effects needs to be accommodated for one of the latter parts. Then, we aim at a program with a revised data flow (cf. right part of the figure) such that downwards passing is generalised to state passing in the relevant part of the program. This scenario can also be accommodated using the operator for refreshment such that positions in the threads are temporarily disconnected as necessary for the inclusion of new output positions into an expanded thread.

The operator $\mathsf{refresh}(\sigma, i)$ **— details** We are going to construct a rule r' that will be a generalisation of the rule r that is referred to with i. To this end, we will also accumulate a substitution θ such that $r = \theta(r')$.

- Initialisation phase: We start with $r' = r$ and the empty substitution for θ.
- Accumulation phase: we iterate over all positions of sort σ in r' and then perform destructive updates on these positions as follows. The term t on the position is replaced by a fresh variable v. We add $v \mapsto t$ to θ.
- Transformation phase: we deploy the generalised rule using the operator generalise, which takes r' and θ as arguments.

Preservation properties

- The operator is semantics-increasing (which is implied by generalisation).
- The operator is undefinedness- and unusedness-increasing for sort σ.
- If we assume that all affected positions take the form of variables, and these variables do not occur on positions of sorts other than σ, then the operator is undefinedness- and unusedness-preserving for all sorts except σ.

Refreshment illustrates the limitations of the proposed preservation properties. The mere observation that the operator is semantics-increasing does not quantify the increased extension of predicates. However, in an intuitive sense, it is quite clear that the increase of semantics is dramatic. Namely, all positions of sort σ will hold fresh and hence unconnected variables. This indicates that our approach to reasoning about evolving programs, as it stands, is rather imprecise.

Recovery of semantics preservation Using similar arguments as we gave for the other schemes of adaptation, we can complete refreshment into a semantics-preserving transformation. Essentially, refreshing is not considered harmful if it can be reversed by threading. That is:

- We refresh predicate positions of sort σ.
- A test is issued to check that threading for sort σ reverses the refreshment.
- Premises are added that ultimately will participate in the data flow.
- Threading is reestablished for σ.

Here we assume that the added premises do not rule out previously valid consequences, neither do the added premises affect the threaded data structure in case the premises participate in state passing. If these side conditions do not hold, then the semantic meanings of the original and the transformed might be related by "$\tilde{<}$", or "$\tilde{>}$", or they might be incomparable.

6 Related work

Evolution of rule-based programs is closely related to the following fields:

- Program calculation from specifications using transformations.
- Stepwise refinement of programs, most notably logic programs.
- Stepwise enhancement of programs, most notably logic programs.
- Extensible specifications or programs, especially Modular SOS.

Besides, our approach can also be viewed as a partial adoption of refactoring [Opd92, Fow99] to the context of rule-based programming. Also, we argued elsewhere [Läm99a] that our transformational approach can be elaborated into an

aspect-oriented programming model [EFB01, KLM+97] for rule-based programs. We will now focus on the main themes listed above.

6.1 Program calculation

This field is also referred to using the term transformational program development. The underlying formula is to develop efficient programs that are derived from high-level, potentially inefficient specifications [BD77, Par90, PP96]. This approach rests on a mathematical style of program construction, where applying and proving algebraic laws is a regular activity [Bd97]. Our approach is more pragmatic: proofs of any form are not to be supplied by the programmer. It is at the heart of program calculation to require semantics preservation. Consequently, software evolution is not an issue, but rather the formal treatment of software implementation or program optimisation.

One stream of transformational programming development is based on ADT-like (say, algebraic) specifications and imperative implementations; see [Par90] for a textbook on the subject. Another stream of this field operates at the level of functional or logic programs; see [PP96] for a seminal survey paper on fold/unfold strategies and others. The notion of program synthesis is very related to transformational program development, while program derivation is subjected to a synthesis-biased view as opposed to a transformation-biased view; see [BDF+03] for an up-to-date survey paper.

Some semantics-preserving evolution operators were adopted from this field — essentially all operators of the category *abstraction*. Our non-strictly semantics-preserving operators, including corresponding evolution relations are original contributions. The two approaches also differ regarding the way how progress is driven. When calculating programs, progress is driven by mathematical insight or by transformation strategies [PP96], which involve some amount of heuristics to derive more efficient programs. In our approach, the *evolution goal* of the programmer drives the process. This goal manifests itself by a certain required type, by the presence or the absence of a certain premise, or by the need for a specific data flow.

6.2 Stepwise refinement

Semantics preservation can be relaxed to include refinement. Then, one is normally interested in the derivation of a program that satisfies a given, maybe incomplete or indeterministic specification. All refinement steps have to meet the specification. The result is then a program that is more specific or determinate than necessarily required by the specification [BW98, dRE98]. This approach has been adopted for logic programming by Komorowski [KT94]. Komorowski's development is based on his earlier work on partial deduction [Kom89], which is the logic-programming term for partial evaluation. The background on partial deduction im-

plies that fold/unfold mutations can be modelled, while additional refinement operators were introduced for the following syntactical operations:

- add: add a clause to a program
- prune: remove a clause from a program
- fatten: add a literal to a clause's body
- thin: remove a literal from a clause's body

These operators are very similar to some of our operators in the *aggregate* category. Also, Komorowski's refinement relation is akin to our relation "$\tilde{\geq}$" used for semantics-decreasing transformations.

Stepwise refinement and our approach differ in the following respects. Firstly, refinement is relative to a specification, while in our approach, there is nothing but the evolving program. Secondly, refinement, as it was adopted for logic programming, is a directed process which basically suggests to start from a 'too rich model' to refine it such that the intended model is obtained. By contrast, our approach is non-monotone in that a given program can be enhanced and cleaned up alternately. Thirdly, we consider a number of original operators. For instance, the categories *parameter*, *reference* and *composite* are missing in Komorowski's work on stepwise refinement. Fourthly, our approach employs types and other program properties to direct transformation operators. Fifthly, our semantic model comes with provisions to deal with incomplete programs; recall the issue of contribution problems. Sixthly, we consider various transformation properties other than refinement. Finally, our approach approach is not restricted to logic programs.

6.3 Stepwise enhancement

Stepwise enhancement [Lak89a, SS94, JS94], as developed by Lakhotia, Sterling et al., is a methodology for developing Prolog programs systematically from two classes of standard components: skeletons and techniques. *Skeletons* are simple Prolog programs with a well-understood control flow. The following skeleton traverses a binary tree with values at leaf nodes (adopted from [NS98]):

```
is_tree(leaf(X)).
is_tree(tree(L, R)) :- is_tree(L), is_tree(R).
```

Another skeleton would be the very similar traversal that only traverses a single branch of the tree. *Techniques* are standard Prolog programming practises, such as building a data structure or 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 add clauses to programs. Unlike skeletons, techniques are not programs but can be viewed as meta-programs.

For instance, the *calculate* technique models computation of a value as follows. 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 example illustrates the application of the *calculate* technique to the above predicate `is_tree/1` (again adopted from [NS98]). The resulting program computes the product of the leave values in the tree. Note that the predicate `is_tree` has been renamed. That is:

```
prod_leaves(leaf(X), X).
prod_leaves(tree(L, R), Z)
 :- prod_leaves(L, X), prod_leaves(R, Y), Z is X * Y.
```

A technique applied to a skeleton is said to yield an *enhancement*. Two enhancements of the same skeleton share computational behaviour, and they can be combined into a single program by *composition*.

Different realisations of stepwise enhancement exist, which we classify as follows:

- The first-order approach: the presentation of examples given above suggest this view: enhancements are properly derived as normal logic programs, as if they were written by the programmer in the first place. Tool support can be involved to actually apply techniques (in the form of meta-programs) to given skeletons.
- The higher-order approach [NS98]: Skeletons and/or techniques can be represented as higher-order predicates that are designed such that their parameters can be filled in to perform specific computations. For example, folding over a tree can be readily described as a higher-order predicate that takes the predicates to be applied to leafs and nodes as arguments.
- The meta-level approach [Lak89b]: enhancements can be simulated by enhancing a vanilla meta-interpreter for Prolog such that it performs additional functionality according to a technique.

Stepwise enhancement and our approach differ in the following respects. Firstly, our schemes of adaptation are not restricted to the addition of computational behaviour; recall operators for removal of premises, parameters, and for the refreshment of data flow. Secondly, stepwise enhancement normally aims at the preservation of computational behaviour of the skeleton. By contrast, our approach to program evolution does not rely on a skeleton in the process, neither does it require preservation of computational behaviour in general. Thirdly, while stepwise enhancement normally starts from techniques, our approach aims at the identification of transformation primitives that can be composed to describe different techniques or schemes of adaptation. Fourthly, in stepwise enhancement, there is no counterpart for our use of type-directed meta-programming. Fifthly, stepwise enhancement has been entirely linked to Prolog, while our approach aims at coverage of different rule-based languages.

6.4 Extensibility vs. evolution

In the introduction, we briefly touched upon the tension between extensibility and evolution. On the one hand, it is easy to claim that evolution is more general because it allows us to revise programs, while extensibility is more or less about *reuse as is*. On the other hand, one could wonder if specifications can maybe become so high-level, abstract, parametric, and modular that they can be reused without revision. Then, revision and evolution are maybe terms that are linked to less sophisticated rule-based notations. We cannot resolve this tension between extensibility and evolution in this paper, but we want to illustrate some issues at least.

Modular SOS Extensibility is a prime attribute of Mosses' Modular SOS [Mos99, Mos02]. Let us consider some typical MSOS rules from [Mos02], namely the small-step rules for an expression assignment:

$$\frac{E_1 - X \to E_1'}{assignment(E_1, E_2) - X \to assignment(E_1', E_2)}$$

$$\frac{E_2 - X \to E_2'}{assignment(L_1, E_2) - X \to assignment(L_1, E_2')}$$

$$\frac{X = \{store = S, store' = S'|U\}, L_1 \in dom(S), S' = \{L_1 = V_2|S\}}{assignment(L_1, V_2) - X \to ()}$$

The first rule progresses with the left-hand side of the assignment. This will ultimately result in a left-value L_1. The second rule progresses with the right-hand side of the assignment. This will ultimately result in a value V_2. The third and final rule realises the assignment of V_2 to L_1 in the store, which is carried in the label X of the transition relation.

A certain, sophisticated use of labels is indeed the key characteristic of MSOS, combined with some other provisions that are analysed carefully in [Mos02]. The MSOS style requires that all semantical components are placed in the labels, and MSOS configurations are then purely value-added syntax trees. MSOS labels are *abstract*, i.e., they can host different kind of information, e.g., bindings, stores, or raised exceptions — without requiring that the MSOS rules are aware of these components. In the end, MSOS labels are records with an open-ended list of components. Full abstraction of labels very much depends on the general provision of *label composition*, which is used in the MSOS rules. That is, labels X_1 and X_2 of subsequent transitions are *composed* by forming a label $X_1; X_2$. For instance, composition of store-like information composition requires that the output store of X_1 is constrained to be identical with the input store of X_2. The abstract and extensible status of labels supports extensibility and modularity of MSOS specifications.

In [Mos99], Mosses identifies label categories that model fundamentally different

ways of processing information: allowing it (i) to be *inspected*, or (ii) to be both *inspected* and *changed*, or (iii) merely to be *provided*. In our transformational reading, these fundamental ways are supported as follows, while referring to schemes of adaptation from Sec. 5:

(i) Insert input positions, and thread them.
(ii) Insert input and output positions, and thread them.
(iii) Insert output positions, and synthesise them.

Consequently, there is correspondence between the abstract and extensible status of MSOS labels vs. the provision of a versatile meta-programming framework, in which we can define schemes of program adaptation including schemes for information processing.

The introductory example suggested that the semantics of a normal method call has to be expanded by premises for executing before and after advice. This raises the question whether the semantics of method calls should have been written maybe in a different style so that intrusion of premises can be avoided, and extensibility continues to be applicable. For instance, we could favour small-step SOS, in the hope that the additional steps for before and after advice can be accommodated by extra MSOS rules, without rewriting the normal semantics of method calls. Then, the remaining problem is that preexisting and added rules do not necessarily get applied in the intended order. For instance, one preexisting rule would have performed method-table look up and prepared the execution of the method body. Further preexisting rules would have executed the body. The new rule for before advice would need to be applied *right before the first step* that starts executing the method body. To this end, one would need to enforce a distinction of advised and unadvised method calls. This can be achieved either by changing the existing rule for method calls, or by assuming an advice-enabled variant of it, or by accomplishing this distinction during the mapping from concrete to abstract syntax.

These problems make us conclude that the generality of a transformational approach is justified. We also note that there are always forms of reuse that *inevitably* require transformations, e.g., fold/unfold mutations or style conversions from big-step to small-step and vice versa. As an aside, a meta-level is also present in MSOS. For instance, the use of unrestricted, extensible records, but also some restrictions, such that side conditions are not affected by label transformation, would require a dedicated type system and a specialised meta-language in the end, which is pointed out in [Mos99, Mos02].

Modular attribute grammars The SOS formalism is not an isolated case of a rule-based notation that can be turned into a highly abstract, parameterised, and modular specification language. For example, attribute grammars have been extended in various directions, with the work by Kastens and Waite [KW94] as both a practical and a sophisticated example. Most notably, this work, which also has been

realised in the attribute grammar system ELI, provides expressiveness for *remote access*. This expressiveness allows the programmer to avoid scattered attributes and computation rules, which makes attribute grammar modules more reusable. We can express remote access in the following manner:

- The *chain* construct corresponds to state passing with threading; see Sec. 5.2.
- The *including* construct corresponds to environment passing; same reference.
- The *constituents* construct corresponds to bottom-up computation; see Sec. 5.3.

This correspondence once more illustrates the enhancement-like merits of our approach. A more detailed account on extensions of the attribute grammar paradigm, and their relation to program transformation is given in [LR99].

7 Concluding remarks

Tracking well-argued changes We presented a pragmatic approach for transforming rule-based programs in the course of evolution. The overall idea is to equip the rule-based programmer with operators to track changes, and with transformation properties to reason about these changes. Throughout the paper, we favoured rule-based notations for SOS specifications and definite clause programs, but we indicated further notations to which the approach is applicable as well. (The reader is forwarded to [Läm99b, LR99] to find proof of this claim for attribute grammars.) Our bias towards SOS and logic programming allowed us to demonstrate evolution scenarios in the well-understood domain of executable semantics descriptions.

Transformation properties Our approach is pragmatic when compared to rigorous transformational program development or mathematical program calculation. That is, our approach is relaxed in that it allows for evolution without insistence on semantics preservation. We provided dedicated evolution relations to describe the direction of evolution, and the disctance between an original and an evolved program. These relations were used directly to define transformation properties for the evolution operators. Preservation properties correspond to one kind of transformation properties. Normally, transformation properties tend to come in triplets: preservation, decrease, increase. For instance, evolutionary transformations might decrease semantics (cf. ‘clean-up’), preserve semantics (cf. ‘refactoring’), or increase semantics (cf. ‘enhancement’). There are other triplets of properties, e.g., decrease, preservation or increase of undefined variables, unused variables, and productive predicates.

Towards a well-argued operator suite This paper makes an effort to identify primitive roles of evolutionary transformations. To this end, the paper integrates and elaborates previous work on operator suites for different languages carried out by

the same author [Läm99b, Läm99a, LR99, LRL00, Läm01]. (Such previous work covers some specialised issues that did not make it into this compilation, e.g., the implementation of meta-programming frameworks [Läm99b, LRL00].) Because the design of the proposed operator suite starts from an abstract categorisation of language constructs and associated operators, the present paper also makes a small contribution to the notion of *language-parametric program restructuring* [LPP04].

A challenge: a rigorous meta-programming framework We opted for a semi-formal style of discussing operator properties. We view the listed properties as correctness criteria for an implementation, but we did not provide a verified reference implementation. Ultimately, one would favour a framework that helps proving properties for primitive operators, while properties of derived operators are maybe even calculated. This is going to be a major challenge for the following reasons: (i) the sheer number of operators and properties, (ii) the unexplored nature of some properties such as well-sortedness preservation, (iii) the limited precision of the preservation properties, which implies imprecise properties for composition as well, and (iv) the self-imposed requirement to deliver a suite for several rule-based languages. Further work is also needed on the set of primitives including means of composition or iteration. The present suite falls short for some complex schematic transformations, e.g., the migration from big-step to small-step SOS as encountered in Sec. 2. That is, some schemes cannot be conveniently expressed in terms of the primitives. Consequently, we often fall back to free-wheeling meta-programming. The full reconciliation of an operator style is a challenging open problem.

Acknowledgement: The author is very grateful for the scrutiny of three anonymous referees who reviewed the paper for the Journal of Logic and Algebraic Programming. The detailed and critical reviews have triggered major efforts. All remaining issues (if any) are in the author's responsibility. I am also very grateful for the help and the patience of Luca Aceto and Wan Fokkink as the editors of this Special Issue on Structural Operational Semantics.

This work was supported, in part, by the German research organisation DFG, in the project *KOKS* ("*Komposition beweisbarkorrekter Sprachbausteine*"), by the German and Slovenian governments in the project SVN 99/028 ("*Adaptive, generic, and aspect-oriented language definitions*"), and by the Dutch research organisation NWO in the project 612.014.006 ("*Generation of program transformation systems*").

I owe gratitude to Jan Kort and Wolfgang Lohmann, with whom I have worked on related problems. This work is dedicated to my former supervisor Günter Riedewald who has introduced me generally to science and specifically to attribute grammars incl. his own brand *Grammars of Syntactical Functions* [Rie79, RMD83].

References

[Abr84] H. Abramson. Definite Clause Translation Grammars. In *Proceedings of the International Symposium on Logic Programming*, pages 233–241, Atlantic City, 1984. IEEE, Computer Society Press.

[ACE96] I. Attali, D. Caromel, and S.O. Ehmety. A natural semantics for Eiffel dynamic binding. *ACM Transactions on Programming Languages and Systems*, 18(6):711–729, November 1996.

[Ada91] S.R. Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, University of Southampton, Faculty of Engineering, Department of Electronics and Computer Science, March 1991.

[AFV01] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2001.

[AL94] A. Aiken and T.K. Lakshman. Directional Type Checking of Logic Programs. In B. Le Charlier, editor, *1st International Symposium on Static Analysis*, volume 864 of *LNCS*, pages 43–60. Springer-Verlag, September 1994.

[And03] J.H. Andrews. The Witness Properties and the Semantics of the Prolog Cut. *Theory and Practice of Logic Programming*, 3(1):1–59, January 2003.

[Apt90] K. Apt. Logic Programing. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–574. Elsevier, 1990. vol. B.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Bd97] R.S. Bird and O. de Moor. *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice Hall, 1997.

[BDF+03] D. Basin, Y. Deville, P. Flener, A. Hamfelt, and J.F. Nilsson. Synthesis of Programs in Computational Logic. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, LNCS. Springer-Verlag, 2003. forthcoming.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.

[BM97] J. Boye and J. Maluszynski. Directional Types and the Annotation Method. *Journal of Logic Programming*, 33(3):179–220, December 1997.

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

[BSG95] K.B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In W.G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming*, volume 952 of *LNCS*, pages 27–51, Åarhus, Denmark, 7–11 August 1995. Springer-Verlag.

[BW98] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, April 1998.

[CF94] R. Cartwright and M. Felleisen. Extensible denotational language specifications. In M. Hagiya and J.C. Mitchell, editors, *Theoretical Aspects of Computer Software: International Symposium*, volume 789 of *LNCS*, pages 244–272. Springer-Verlag, April 1994.

[Des88] T. Despeyroux. TYPOL: A formalism to implement natural semantics. Technical report 94, INRIA, March 1988.

[DM85] P. Deransart and J. Małuszyński. Relating Logic Programs and Attribute Grammars. *Journal of Logic Programming*, 2(2):119–155, 1985.

[DM93] P. Deransart and J. Maluszyński. *A Grammatical View of Logic Programming*. The MIT Press, 1993.

[dRE98] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York, NY, 1998.

[EFB01] T. Elrad, R.E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *CACM*, 44(10):29–32, October 2001. An introduction to the CACM special issue on AOP.

[FKF99] M. Flatt, S. Krishnamurthi, and M. Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 241–270. Springer-Verlag, 1999.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[Jai95] A. 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.

[Jef02] D. Jeffery. *Expressive Type Systems for Logic Programming Languagaes*. PhD thesis, Dept. of Comp. Science and Software Eng., The University of Melbourne, February 2002.

[JS94] A. Jain and L. 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] G. 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.

[KL03] J. Kort and R. Lämmel. A Framework for Datatype Transformation. In B.R. Bryant and J. Saraiva, editors, *Proc. of Language, Descriptions, Tools, and Applications (LDTA 2003)*. Elsevier, April 2003. 20 pages.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M.

Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97—Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer-Verlag.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2:127–145, 1968. Corrections in 5:95-96, 1971.

[Kom89] J. Komorowski. Synthesis of programs in the framework of partial deduction. Reports on Computer Science and Mathematics, Ser. A 81, Department of Computer Science, Åbo Akademi, Finland, 1989.

[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.

[KT94] J. Komorowski and S. Treek. Towards refinement of definite logic programs. In Z.W. Raś and M. Zemankova, editors, *Proc. of the 8th International Symposium on Methodologies for Intelligent Systems*, volume 869 of *LNAI*, pages 315–325, Berlin, October 1994. Springer-Verlag.

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

[Lak89a] A. Lakhotia. *A Workbench for Developing Logic Programs by Stepwise Enhancement*. PhD thesis, Case Western Reserve University, 1989.

[Lak89b] A. Lakhotia. Incorporating 'Programming Techniques' into Prolog Programs. In E.L. Lusk and R.A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming (NACLP '89)*, pages 426–440, Cleveland, Ohio, October 1989. MIT Press.

[Läm99a] R. Lämmel. Declarative aspect-oriented programming. In O. Danvy, editor, *Proceedings PEPM'99, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM'99, San Antonio (Texas), BRICS Notes Series NS-99-1*, pages 131–146, January 1999.

[Läm99b] R. Lämmel. *Functional meta-programs towards reusability in the declarative parad igm*. PhD thesis, Universität Rostock, Fachbereich Informatik, January 1999. Published by Shaker Verlag in 1998, ISBN 3-8265-6042-6.

[Läm01] R. Lämmel. Grammar Adaptation. In J.N. Oliveira and P. Zave, editors, *Proc. Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

[Läm02] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, April 2002. ACM Press.

[LHJ95] S. Liang, P. Hudak, and M.P. Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, January 22–25, 1995*, pages 333–343. ACM Press, 1995.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987. 2nd edition.

[LMM88] J.-L. Lassez, M.J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.

[LPP04] Language-Parametric Program Restructuring — Project web site, February 2004. project funded by the Dutch Research Organisation (NWO) `http://www.cs.vu.nl/lppr/`.

[LR99] R. Lämmel and G. Riedewald. Reconstruction of paradigm shifts. In *Second Workshop on Attribute Grammars and their Applications, WAGA 99*, pages 37–56, March 1999. INRIA, ISBN 2-7261-1138-6.

[LR01] R. Lämmel and G. Riedewald. Prological Language Processing. In M. van den Brand and D. Parigot, editors, *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01), Genova, Italy, April 7, 2001, Satellite event of ETAPS'2001*, volume 44 of *ENTCS*. Elsevier Science, April 2001.

[LRL00] R. Lämmel, G. Riedewald, and W. Lohmann. Roles of Program Extension. In A. Bossi, editor, *Post-workshop proceedings LOPSTR'99*, number 1817 in LNCS. Springer-Verlag, 2000. 20 pages.

[MO84] A. Mycroft and R.A. O'Keefe. A Polymorphic Type System for PROLOG. *Artifical Intelligence*, 23(3):295–307, August 1984.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[Mos92] P.D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[Mos99] P.D. Mosses. Foundations of Modular SOS. Technical report, BRICS, December 1999. Reports Series RS-99-54.

[Mos02] P.D. Mosses. Pragmatics of Modular SOS. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology: 9th International Conference, AMAST'02*, volume 2422 of *LNCS*, pages 21–40. Springer-Verlag, January 2002.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[NS98] L. Naish and L. Sterling. A Higher Order Reconstruction of Stepwise Enhancement. In N.E. Fuchs, editor, *Logic program synthesis and transformation: 7th international workshop, LOPSTR'97, Leuven, Belgium, July 10–12, 1997: proceedings*, volume 1463 of *LNCS*, New York, NY, USA, 1998. Springer-Verlag.

[Opd92] W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Par90] H.A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.

[Pet94] M. 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 Imple-*

mentation and Logic Programming, PLILP'94, volume 844 of *LNCS*, pages 117–131. Springer-Verlag, 1994.

[Plo70] G.D. Plotkin. A Note on Inductive Generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, 1970.

[Plo81] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report FN-19, DAIMI, University of Aarhus, Denmark, September 1981.

[PP96] A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, June 1996.

[PS90a] J. Palsberg and M.I. Schwartzbach. Type subsitution for object-oriented programming. *SIGPLAN Notices*, 25(10), October 1990. Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming, Ottawa, Canada, 21-25 October 1990.

[PS90b] A.J. Power and L.S. Sterling. A notion of Map Between Logic Programs. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7th International Conference on Logic Programming (ICLP'90)*, pages 390–404. The MIT Press, 1990.

[PW80] F.C.N. Pereira and D.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.

[REK04] The Rule Evolution Kit, version 0.77, 11 February 2004. `http://www.cs.vu.nl/rek/`.

[Rey70] J.C. Reynolds. Transformational Systems and the Algebraic Structure of Atomic Formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 135–151. Edinburgh University Press, 1970.

[Rie79] G. Riedewald. *Compilerkonstruktion und Grammatiken syntaktischer Funktionen*. Dissertation B, Rechenzentrum der Universität Rostock, 1979.

[RL89] G. Riedewald and U. Lämmel. Using an attribute grammar as a logic program. In P. Deransart, B. Lorho, and J. Maluszyński, editors, *Programming Languages, Implementation and Logic Programming (PLILP'88), Proc.*, number 348 in LNCS, pages 161–179. Springer-Verlag, May 1989.

[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.

[SK95] K. Slonneger and B.L. Kurtz. *Formal Syntax and Semantics — A Laboratory Based Approach*. Addison-Wesley Publishing Company, 1995.

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

[TS00] F. Tip and P.F. Sweeney. Class hierarchy specialization. *Acta Informatica*, 36(12):927–982, 2000.

[VK76] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.

[vRS91] A. van Gelder, K. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[Wad92] P. Wadler. The essence of functional programming (Invited talk). In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM SIGACT and SIGPLAN, ACM Press, 1992.

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