Evolution scenarios for rule-based implementations of language-based functionality

Ralf Lämmel

Vrije Universiteit Amsterdam
Centrum voor Wiskunde en Informatica

September 20, 2004

Abstract

We work through a sequence of evolution scenarios for language-based functionality implemented as rule-based programs. We identify and illustrate different dimensions along which such functionality can evolve, including the following: (i) coding style; (ii) coding details; (iii) data model; (iv) crosscutting concerns; and (v) patches.

We focus at language interpreters as examples of language-based functionality, but similar scenarios exist for type checkers, static analyses, program transformations, and other sorts of language-based functionality. We opt for Prolog as the rule-based programming language used for the implementation of language-based functionality. We employ the Prolog-based Rule Evolution Kit (REK) for the operationalisation of the evolution scenarios by means of evolutionary transformations.

We compile a list of exercises that are meant to help with the digestion of the scenarios and with the further exploration of the overall subject.

Keywords: evolution, evolutionary transformations, rule-based programming, language interpreters, language-based functionality, SOS, meta-programming, Prolog, REK

Note: This article complements an invited lecture at the Workshop on Structural Operational Semantics, August 30, 2004, London. Some fragments of this article were extracted from [12] — “Evolution of rule-based programs”.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
1 Preamble

We are going to look at the evolution of language-based functionality. By language-based functionality we mean implementations of interpreters, type checkers, program analyses, and program transformations. In this paper, we will focus at evolution scenarios for language interpreters. With regard to evolution, we are particularly interested in a transformational view on program adaptation and program extension. That is, we are going to operationalise evolution scenarios in terms of evolutionary transformations.

We will encode language-based functionality in Prolog. The rule-based programming language Prolog has a strong record in this respect. For instance, one can encode SOS specifications more or less directly in Prolog, so that one gets a language interpreter for free \[3]. One can also encode attribute grammars in Prolog (while some limitations apply) \[2]. We will encode the evolutionary transformations in Prolog, too. Again, Prolog has a strong record with regard to meta-programming. In particular, we employ the Rule Evolution Kit (REK \[26]), which is a Prolog library that provides operators for evolutionary transformations to be performed on Prolog programs.

2 An initial interpreter

The evolution scenarios from the subsequent sections will all deal with adaptations or extensions of an interpreter for a simple expression-oriented language, which we call PURE. We will spell out this interpreter for clarity.

The following Prolog session records an attempt to compute the expression \(\text{let } x = 2 \text{ in let } y = 3 \text{ in } x \times y\):

\[
\text{?- evaluate( let(x,const(2),}
\text{\text{let(y,const(3),}
\text{\text{binary(var(x),*,var(y))})}
\text{\text{,[]}
\text{\text{, Val ).}
\text{Val = 6}
\text{Yes}
\text{?- }\%	ext{ halt}
\]

The predicate \text{evaluate/3} interprets PURE expressions. The first argument is the term representation of the aforementioned expression. The second argument is [], which denotes the empty variable environment. The third argument, \text{Val}, is bound to 6 by expression evaluation.
The type of the predicate `evaluate/3` is the following, using REK’s notation:\footnote{This notation is adopted from \cite{14}. Each predicate position is typed by a mode and a sort. The mode “+” marks inputs (or arguments), while the mode “−” marks outputs (or results). Sorts are predefined types (e.g., `number`), aliases, or algebraic data types.}

\begin{verbatim}
:- profile evaluate(+exp,+varenv,-val).
\end{verbatim}

REK type-checks predicates against such profiles.

REK also uses types in evolutionary transformations.

The expression syntax is defined by the following algebraic data type:

\begin{verbatim}
:- data exp = const(number) % Number constants
    | binary(exp,op,exp) % Binary operations
    | if(exp,op,exp) % If-then-else
    | var(varid) % Read a variable
    | let(varid,exp,exp) % Bind a variable in a scope
:- data op = (+) | .... % Arithmetic operations
:- alias varid = atom. % Variable identifiers
:- alias val = number.
\end{verbatim}

Values (i.e., results of expression evaluation) are numbers:

\begin{verbatim}
:- alias val = number.
\end{verbatim}

Environments are lists of identifier-value pairs:

\begin{verbatim}
:- alias varenv = [(varid,val)].
\end{verbatim}

We will now give the interpreter rules, which are entirely straightforward.

Constants are trivially interpreted by this Prolog fact:

\begin{verbatim}
evaluate(const(Number),_VarEnv,Number).
\end{verbatim}

Binary expressions give rise to the following Prolog rule, which resembles a big-step SOS rule as in Kahn’s Natural semantics modulo notational transcription:

\begin{verbatim}
evaluate(binary(Exp0,Op,Exp1),VarEnv,Val2)
    :- evaluate(Exp0,VarEnv,Val0), % Compute the left operand
        evaluate(Exp1,VarEnv,Val1), % Compute the right operand
        applyBop(Op,Val0,Val1,Val2). % Apply the binary operation
\end{verbatim}

(We omit the trivial helper predicate `applyBop/4`.)

If-then-else expressions are interpreted by two rules depending on the value of the condition; the number zero denotes `False`; all others `True`:

\begin{verbatim}
evaluate(if(Exp0,Exp1,_Exp2),VarEnv,Val1)
    :- evaluate(Exp0,VarEnv,Val0), % Compute the condition
        isNotZero(Val0), % Succeed for non-zero condition
        evaluate(Exp1,VarEnv,Val1). % Compute then expression

evaluate(if(Exp0,_Exp1,Exp2),VarEnv,Val1)
    :- evaluate(Exp0,VarEnv,Val0), % Compute the condition
        isZero(Val0), % Succeed for zero condition
        evaluate(Exp2,VarEnv,Val1). % Compute else expression
\end{verbatim}
Variables are looked up from the environment:

\[
evaluate(var(\text{Varid}), \text{VarEnv}, \text{Val})
\]  
\[
:- \ \text{lookupVarEnv}(\text{VarEnv}, \text{Varid}, \text{Val}).
\]

(We omit the predicate \text{lookupVarEnv} for environment look-up.)

A \text{let} binding is propagated via the environment:

\[
evaluate(let(\text{Varid}, \text{Exp0}, \text{Exp1}), \text{VarEnv0}, \text{Val1})
\]
\[
:- \ \text{evaluate}(\text{Exp0}, \text{VarEnv0}, \text{Val0}),
\]
\[
\text{modifyVarEnv}(\text{VarEnv0}, \text{Varid}, \text{Val0}, \text{ValEnv1}),
\]
\[
\text{evaluate}(\text{Exp1}, \text{VarEnv1}, \text{Val1}).
\]

(We omit the predicate \text{modifyVarEnv} for environment modification.)

The following sections concern the evolution of the PURE interpreter.

3 Evolution in the sense of style conversion

The original interpreter was given in big-step style. Some language extensions are more easily accommodated when small-step style is chosen, e.g., exception handling or concurrency are of that kind. We will therefore convert the PURE interpreter from big-step to small-step style.

Let us consider again the interpreter rule for binary expressions:

\[
evaluate(binary(\text{Exp0}, \text{Op}, \text{Exp1}), \text{VarEnv}, \text{Val2})
\]
\[
:- \ \text{evaluate}(\text{Exp0}, \text{VarEnv}, \text{Val0}),
\]
\[
\text{applyBop}(\text{Op}, \text{Val0}, \text{Val1}, \text{Val2}).
\]

This rule is clearly in big-step style because all subexpressions are reduced by recursive applications of \text{evaluate/3} in the body of the rule. Not even the \text{type} of the predicate is ready for small steps. We contrast the current and the required profile for \text{evaluate/3}:

\[
:- \ \text{profile} \ \text{evaluate}(\text{exp}, \text{varenv}, \text{-val}). \ % \text{inherently big-step style}
\]
\[
:- \ \text{profile} \ \text{evaluate}(\text{exp}, \text{varenv}, \text{-exp}). \ % \text{amenable to small steps}
\]

(We tend to underline adapted program fragments as shown above.)

That is, for small-step style, the predicate would describe a relation on two expressions (including reduced values) rather than a relation between an expression and a value. We adapt the type of \text{evaluate/3} by embedding values into expressions using a new functor \text{val2exp/1}.

We add the functor to the expression syntax as follows:

\[
:- \ \text{data} \ \text{exp} = \text{val2exp}(\text{val}).
\]

For the sake of typeability preservation, the proposed adaption of the type
of `evaluate/3` necessitates the adaptation of all rules that use or define `evaluate/3`. That is, we systematically wrap its result positions by `val2exp/1`. For instance, the rule for binary expressions is adapted as follows:

```prolog
evaluate(binary(Exp0,Op,Exp1),VarEnv,val2exp(Val2))
   :- evaluate(Exp0,VarEnv,val2exp(Val0)),
      evaluate(Exp1,VarEnv,val2exp(Val1)),
      applyBop(Op,Val0,Val1,Val2).
```

In fact, REK provides an operator `othertype` that automates just that. The following application of `othertype` converts all PURE rules to the small-step-enabled type:

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

This goal clause illustrates the REK-style of recording evolutionary transformations. One can also issue such transformations directly from within a Prolog session. One can also envisage interactive tool support for evolutionary transformations.

We note that the types of the operands `evaluate` and `val2exp` control the application of the `othertype` operator. That is, the domain `val` and the co-domain `exp` of the functor `val2exp` specify the type of the relevant predicate position before and after the transformation.

The hard part of the big-step to small-step conversion is to take apart the big-step rules. Looking at the rule for binary expressions, it is intuitively clear that we want to end up with three small-step rules: one for each of the two operands, and another for the actual application of the binary operation.

Here is the first small-step rule that makes progress with the left operand:

```prolog
evaluate(binary(Exp0,0p,Exp1),VarEnv,binary(Exp2,0p,Exp1))
   :- evaluate(Exp0,VarEnv,Exp2).
```

There is similar rule that makes progress with the right operand instead:

```prolog
evaluate(binary(Exp0,0p,Exp1),VarEnv,binary(Exp0,0p,Exp2))
   :- Exp0 = val2exp(_Val),
      evaluate(Exp1,VarEnv,Exp2).
```

This rule is such that the left operand must be in reduced form before the right operand is considered. We assume that the order of the premises in the big-step rule suggests such a deterministic evaluation order.

Once both operands are in reduced form, we can compute the final value:

```prolog
evaluate(binary(val2exp(Val0),0p,val2exp(Val1)),_VarEnv,val2exp(Val2))
   :- applyBop(0p,Val0,Val1,Val2).
```

Again, REK provides an operator `big2small` that automates just that. The following application converts all `big` PURE rules to small-step style:

```prolog
:- big2small(evaluate,exp).
```
The big2small operator performs systematic analyses to recover structural induction and relevant data flow from the big-step rules.\(^2\)

To complete the scenario we complement the obtained small-step relation by an extra predicate that takes the transitive closure. As a helper step, we rename the new predicate evaluate/3 to evaluateSmall/3 so that we better point out its status. REK provides a suitable rename operator for this purpose:

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

Here is the Prolog predicate that takes the transitive closure:

```prolog
:- profile evaluateBig(+exp,+varenv,-val).
evaluateBig(Exp0,VarEnv,Val) :-
  evaluateSmall(Exp0,VarEnv,Exp1), % Do one step!
  ( evaluateBig(Exp1,VarEnv,Val) % Do more steps if possible!
    ; Exp1 = val2exp(Val) ). % All steps done; value found.
```

We leave it as an exercise to the reader to add language constructs that take advantage of the established small-step style.

We also note that there are actually many other style conversions that could help during program evolution: conversion from small-step to big-step style, mixed style variations for mutually recursive predicates, CPS conversion, and others. Moreover, specific rule-based programming languages or domain-specific language-based functionality give rise to accordingly specific style conversions; we mention monad introduction (or monadification) as an interpreter-like example adopted from functional programming [10,5]. Another example is removal of left-recursion in a grammar, or even in an attribute grammar, as relevant in frontend implementation [19].

We will turn back to big-step style for the rest of the paper.

### 4 Evolution with regard to the data model

Suppose we want to add object-oriented constructs to the PURE language. It is straightforward to extend the sort exp for expression forms:

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

\(^2\) The current implementation of the operator in REK is ad-hoc. Defining a general, evidently sound, and transparent big-step to small-step conversion is a research challenge.
However, it turns out that the interpretation of these constructs cannot be accomplished by just adding rules to the original PURE interpreter. This interpreter is not fit for such a conservative extension.

One problem is that, so far, the type of evaluation results is hard-coded to coincide with \texttt{number}. Now we need to discriminate numbers and object references. In fact, the crux of the problem is that \texttt{val} is a type alias so far:

\begin{verbatim}
:- alias val = number. % The offending closed-world assumption
\end{verbatim}

As an intermediate step, we need to turn \texttt{val} into a data type:

\begin{verbatim}
:- data val = num(number). % Object references to be added later
\end{verbatim}

Alas, just redefining \texttt{val}, as shown, is insufficient. The original rules need to be adapted to reflect the type changes. For instance, the normal evaluation of constants will not type-check without further provisions:

\begin{verbatim}
evaluate(const(Number),_VarEnv,Number). % Type error!!!
\end{verbatim}

Whenever preexisting PURE rules commit to numbers, they need to be adapted. The Prolog fact for the evaluation of constants needs to be adapted such that returned number is wrapped by the functor \texttt{num/1}:

\begin{verbatim}
evaluate(const(Number),_VarEnv,num(Number)).
\end{verbatim}

REK offers operators \texttt{newtype} and \texttt{relax}, which can be used jointly to automate this evolutionary transformation:

\begin{verbatim}
:- newtype(val,num). % Turn alias into data type
:- relax(val). % Eliminate vacuous matching and building
\end{verbatim}

This sequence delivers the required data type \texttt{val}, and all positions of type \texttt{val} are wrapped in \texttt{num}, whenever necessary. The application of the \texttt{newtype} operator really wraps all positions of type \texttt{val} in \texttt{num}, while the application of the \texttt{relax} operator eliminates vacuous occurrences of \texttt{num}. To understand this detail, let us consider the rule for \texttt{let} expressions, as it looks like, before the \texttt{relax} operator is applied:

\begin{verbatim}
evaluate(let(Varid,Exp0,Exp1),VarEnv0,num(Number1))
    :- evaluate(Exp0,VarEnv0,num(Number0)),
        modifyVarEnv(VarEnv0,Varid,num(Number0),VarEnv1),
        evaluate(Exp1,VarEnv1,num(Number1)).
\end{verbatim}

All occurrences of \texttt{num} are vacuous in the sense that matched numbers are never used as is, but they are always re-wrapped in \texttt{num}. Hence, the rule can be simplified as follows, which is indeed automated by the \texttt{relax} operator:

\begin{verbatim}
evaluate(let(Varid,Exp0,Exp1),VarEnv0,Val1)
    :- evaluate(Exp0,VarEnv0,Val0),
        modifyVarEnv(VarEnv0,Varid,Val0,VarEnv1),
        evaluate(Exp1,VarEnv1,Val1).
\end{verbatim}

This simplification reflects that a \texttt{let} expression would be useful for both numbers and object references. Technically, the \texttt{relax} operator supports a certain
form of anti-unification as pioneered by Plotkin and Reynolds \cite{24,27,18}.

It remains to add object references:

\begin{verbatim}
:- data val = ref(ref). % Add functor for object references
:- alias ref = integer. % Object references as integers
\end{verbatim}

For completeness’ sake, we also add all semantic domains that will be eventually needed by the interpreter rules for the object-oriented language extension:

\begin{verbatim}
:- alias this = maybe(ref). % The current object
:- alias vmt = [(class,meth,[varid],exp)]. % Method tables
:- alias store = [(ref,class,obj)]. % Object stores
:- alias obj = [(field,val)]. % Field-value pairs
\end{verbatim}

So far, we have evolved the original interpreter for the PURE language so that at least its data model is fit for the object-oriented language extension. More effort is necessary to complete the actual language extension, as we will discuss now.

\section{Evolution with regard to crosscutting concerns}

The object-oriented language extension also requires an enhanced predicate for expression evaluation. So far, we only pass around an environment for variables in the sense of method arguments and temporary results (cf. \textit{let}). We need to add parameters for the virtual method table, for the current object, and for the object store. The original rules have to be adapted such that they participate in a data flow for these new semantic components.

Such dispersed data flow or computation that affects many or all existing rules is best viewed as the implementation of a crosscutting concern in the sense of aspect-oriented programming. (This link between aspect-oriented programming, rule-based programming and program transformation is explored in some detail in \cite{9}.)

For comparison, here is the original type of the \texttt{evaluate} predicate:

\begin{verbatim}
:- profile evaluate(+exp,+varenv,-val).
\end{verbatim}

The required predicate must be of the following type:

\begin{verbatim}
:- profile evaluate(+exp,+varenv,+vmt,+this,+store,-val,-store).
\end{verbatim}

(By default, we associate arguments to the left, and results to the right. This convention is also used by REK, when positions are added to predicates.)

For comparison, here is the original rule for binary expressions:

\begin{verbatim}
evaluate(binary(Exp0,Op,Exp1),VarEnv,Val2)
    :- evaluate(Exp0,VarEnv,Val0),
       evaluate(Exp1,VarEnv,Val1),
       applyBop(Op,Val0,Val1,Val2).
\end{verbatim}
The enhanced rule looks as follows:

\[
\text{evaluate(binary(Exp0,Op,Exp1),VarEnv,Vmt,This,Store0,Val2,Store2)} \\
:\text{:- evaluate(Exp0,VarEnv,Vmt,This,Store0,Val0,Store1),} \\
\text{evaluate(Exp1,VarEnv,Vmt,This,Store1,Val1,Store2),} \\
\text{applyBop(Op,Val0,Val1,Val2).}
\]

We note that the virtual method table is read-only: \text{Vmt} is propagated from the head of the rule to all recursive occurrences of the \text{evaluate} predicate; likewise for the current object \text{This}. By contrast, the object store is subject to a state passing regime: \text{Store0} enters the rule via the head; it is threaded through the rule using \text{Store1} and \text{Store2}; the latter is returned via the head.

REK again offers operators that automate these enhancements. A first wave of transformations add all the required predicate positions:

\[
\text{:- add(+vmt,evaluate).} \quad \% \text{To pass on the virtual method table} \\
\text{:- add(+this,evaluate).} \quad \% \text{To pass on the current object} \\
\text{:- add(+store,evaluate).} \quad \% \text{To receive the object store} \\
\text{:- add(-store,evaluate).} \quad \% \text{To return the object store}
\]

Foremost, these transformations lead to the expected type. That is, REK reports the following type for the \text{evaluate} predicate indeed:

\[
\text{:- profile evaluate(+exp,+varenv,+vmt,+this,+store,-val,-store).}
\]

However, the rules are in a transient state: fresh variables were added as parameters for all occurrences of \text{evaluate}, but no specific data flow has been encoded yet. So we are faced with the following intermediate result:

\[
\text{evaluate(binary(Exp0,Op,Exp1),VarEnv,\_Vmt0,\_This0,\_Store0,Val2,\_Store5)} \\
:\text{:- evaluate(Exp0,VarEnv,\_Vmt1,\_This1,\_Store1,Val0,\_Store2),} \\
\text{evaluate(Exp1,VarEnv,\_Vmt2,\_This2,\_Store3,Val1,\_Store4),} \\
\text{applyBop(Op,Val0,Val1,Val2).}
\]

REK’s \text{thread} operator can be used to establish the correct data flow. Using the following second wave of transformations we enhance all interpreter rules, as it was illustrated earlier for binary expressions:

\[
\text{:- thread(vmt).} \quad \% \text{Arrange for propagation of virtual method table} \\
\text{:- thread(this).} \quad \% \text{Arrange for propagation of current object} \\
\text{:- thread(store).} \quad \% \text{Arrange for state passing of object stores}
\]

Conceptually, the \text{thread} operator supports a versatile combination of the regimes for environment and state passing. Technically, the \text{thread} operator establishes data flow by unification of variables in a rule. To this end, all parameters of the given sort (cf. \text{vmt}, \text{this}, and \text{store}) are ordered in a certain left-to-right order such that each undefined, using occurrence can be unified with the nearest, defining occurrence; we refer to [12] for details.

The overall benefit of using transformations for the implementation of crosscutting concerns is that no tangled code needs to be written. Also, the original (simple) rules can be reused, and the crosscutting concerns are described sep-
arately — as applications of transformation operators. This is clearly a form of separation of concerns.

We note that crosscutting concerns other than threading can be subject to the evolution of rule-based programs. The so-called techniques for stepwise enhancement of Prolog programs \[8,28,7\] and related transformational frameworks for rule-based programming \[9,13,15\] facilitate other crosscutting deltas. For instance, there are techniques for a kind of monoidal computation in a rule set, where operations of a monoid are used to synthesise information (such as output) that is contributed by subcomputations. Yet another kind of a crosscutting concern is to reflection-enable a rule-based program, i.e., a program is enhanced such that it can introspect its own execution.

Finally, there are also evolutionary transformations for slicing out crosscutting concerns, when their present implementation is considered inappropriate. For instance, there are (approximative) inverses of the operators \texttt{add} and \texttt{thread}. The inverse of \texttt{add} allows one to remove positions and the associated data flow. The approximative inverse of \texttt{thread} allows one to refresh predicate positions such that they do not carry the same variable, and thereby a different data flow can be established subsequently.

6 Evolution in the sense of conservative extension

We are now in the position to add the rules for the object-oriented constructs. The previous advances of the data model and data flow have made it possible to perform a truly conservative extension in the end: the rules to be added do not affect the reduction of programs that only refer to PURE constructs [1]. This sort of evolution is very simple because it basically means to ‘put together’ two rule sets as opposed to an invasive transformation of rules.

For brevity, we only provide details for one object-oriented construct. There are no surprises. The following rule evaluates method calls:

\[
evaluate\left(\text{call}(\text{Exp}_0,\text{Meth},\text{Exps}),\text{VarEnv}_0,\text{Vmt},\text{This}_0,\text{Store}_0,\text{Val},\text{Store}_3)\right) :
\text{evaluate}(\text{Exp}_0,\text{VarEnv}_0,\text{Vmt},\text{This}_0,\text{Store}_0,\text{ref}(\text{Ref}),\text{Store}_1),
\text{evaluatelist}(\text{Exps},\text{VarEnv}_0,\text{Vmt},\text{This}_0,\text{Store}_1,\text{Vals},\text{Store}_2),
\text{lookupVmt}(\text{Vmt},\text{Store}_2,\text{Ref},\text{Meth},\text{Args},\text{Exp}_1),
\text{zip}((\text{Args},\text{Vals},\text{VarEnv}_1),
\text{evaluate}(\text{Exp}_1,\text{VarEnv}_1,\text{Vmt},\text{just}(\text{Ref}),\text{Store}_2,\text{Val},\text{Store}_3).\]

For clarity, we explain the premises one by one:

- \texttt{evaluate(...)}: the callee object is determined.
- \texttt{evaluatelist(...)}: the actual parameters are computed.
- \texttt{lookupVmt(...)}: the method is looked up.
- \texttt{zip(...)}: formal parameters are mapped to actual parameters.
- \texttt{evaluate(...)}: the method body is evaluated.
By adding this rule and further rules for get/set field access and other constructs, we have completed the object-oriented language extension. In our view, evolution of rule-based programs is a continuous process, where activities of the following kind alternate:

- Restructuring to prepare for extension or revision.
- Extension to add new concerns by modular composition or weaving.
- Revision to remove or to change inappropriate parts.

7 Evolution in the sense of point-wise restructuring

Style conversions as of Sec. 3 are highly systematic restructuring transformations that require little user intervention. Clearly, rule-based programs can also be subjected to more specific restructuring transformations, where the programmer points out locations of interest.

We will improve one particular detail of the interpreter that we obtained so far. That is, we are going to reduce the number of arguments of the predicate for expression evaluation. For comparison, the current profile is this:

\[
\text{- profile evaluate(+exp,+varenv,+vmt,+this,+store,-val,-store)}.
\]

It seems that having three positions +varenv,+vmt,+this is somewhat outrageous since these positions are all concerned with environment-like information. All this information is passed on to subcomputations. So we aim at compound environments that comprise the following components:

\[
\text{- alias env = (varenv,vmt,this)}.
\]

The profile of the predicate for evaluation is simplified as follows:

\[
\text{- profile evaluate(+exp,+env,+store,-val,-store)}.
\]

For comparison, these are two verbose rules that call for simplification:

\[
\begin{align*}
\text{evaluate} & (\text{const}(\text{Number})), \text{VarEnv}, \text{Vmt}, \text{This}, \text{Store}, \text{num}(\text{Number}), \text{Store}). \\
\text{evaluate} & (\text{binary}(\text{Exp0}, \text{Op}, \text{Exp1}), \text{VarEnv}, \text{Vmt}, \text{This}, \text{Store0}, \text{Val2}, \text{Store2}) \\
& \quad :\!- \text{evaluate}(\text{Exp0}, \text{VarEnv}, \text{Vmt}, \text{This}, \text{Store0}, \text{Val1}, \text{Store1}), \\
& \quad \quad \text{evaluate}(\text{Exp1}, \text{VarEnv}, \text{Vmt}, \text{This}, \text{Store1}, \text{Val1}, \text{Store2}), \\
& \quad \quad \text{applyBop}(\text{Op}, \text{Val0, Val1, Val2}).
\end{align*}
\]

Rules such as those above are not really concerned with any environment-like information. So these rules become more concise by the use of a single environment-like parameter:

\[
\begin{align*}
\text{evaluate} & (\text{const}(\text{Number})), \text{Env}, \text{Store}, \text{num}(\text{Number}), \text{Store}). \\
\text{evaluate} & (\text{binary}(\text{Exp0}, \text{Op}, \text{Exp1}), \text{Env}, \text{Store0}, \text{Val2}, \text{Store2}) \\
& \quad :\!- \text{evaluate}(\text{Exp0}, \text{Env}, \text{Store0}, \text{Val0}, \text{Store1}), \\
& \quad \quad \text{evaluate}(\text{Exp1}, \text{Env}, \text{Store1}, \text{Val1}, \text{Store2}), \\
& \quad \quad \text{applyBop}(\text{Op}, \text{Val0, Val1, Val2}).
\end{align*}
\]

For rules that are concerned with environment components, tuple patterns are
sufficient for access to the components. Indeed, tuple patterns replace the use of multiple positions in the original rules. For instance, the rule for method calls is restructured as follows:

\[
\text{evaluate}(\text{call}(\text{Exp}_0, \text{Meth}, \text{Exps}), (\text{VarEnv}_0, \text{Vmt}, \text{This}), \text{Store}_0, \text{Val}, \text{Store}_3) :-
\text{evaluate}(\text{Exp}_0, (\text{VarEnv}_0, \text{Vmt}, \text{This}), \text{Store}_0, \text{ref}(\text{Ref}), \text{Store}_1),
\text{evaluatelist}(\text{Exps}, (\text{VarEnv}_0, \text{Vmt}, \text{This}), \text{Store}_1, \text{Vals}, \text{Store}_2),
\text{lookupVmt}(\text{Vmt}, \text{Store}_2, \text{Ref}, \text{Meth}, \text{Varids}, \text{Exp}_1),
\text{zip}(\text{Varids}, \text{Vals}, \text{VarEnv}_1),
\text{evaluate}(\text{Exp}_1, (\text{VarEnv}_1, \text{Vmt}, \text{just}(\text{Ref})), \text{Store}_2, \text{Val}, \text{Store}_3).
\]

The grouping effort is simply automated by REK’s \texttt{group} operator:

\[
:- \text{group}(\text{env}).
\]

That is, the tuple type \texttt{env} is viewed as an enumeration of positions that should be grouped. The \texttt{group} operator searches all predicate profiles for positions that add up to the tuple type. Each such group of positions is replaced by a single position of the tuple type.

The illustrated grouping transformation operates at the level of predicate positions. One can also consider forms of restructuring that operate at other levels, e.g., the level of functor positions or the level of rule bodies. Folklore examples of transformations at the level of rule bodies are folding and unfolding, where unfolding means to symbolically perform predicate application, and folding is the inverse [23].

8 Evolution in the sense of patching

We are going to perform one more interpreter extension. That is, we enable logging of method calls. Thereby, we obtain a simple debugging facility for the interpreted object-oriented language. The following logging file makes it all clear; we show the computation of 3!:

\[
\begin{align*}
\text{Call: } & 1.\text{fac}([ (x, \text{num}(3))] ) \\
\text{Call: } & 1.\text{fac}([ (x, \text{num}(2))] ) \\
\text{Call: } & 1.\text{fac}([ (x, \text{num}(1))] ) \\
\text{Call: } & 1.\text{fac}([ (x, \text{num}(0))] ) \\
\text{Return: } & \text{num}(1) \\
\text{Return: } & \text{num}(1) \\
\text{Return: } & \text{num}(2) \\
\text{Return: } & \text{num}(6)
\end{align*}
\]

For each method call, we show the object reference of the callee, the name of the method being called, and the argument environment. Here we assume that the Factorial method is provided by an object with reference 1. Indentation expresses the caller-callee relationships. Upon completion of a method call, the return value is shown.
It is relatively straightforward to adapt the OO interpreter for this purpose. The interpreter rule for method calls is affected as follows:

\[
\text{evaluate}\left(\text{call(Exp0,Meth,Exps),(VarEnv0,Vmt,This)},Store0,Val,Store3\right) \\
\text{:- evaluate(Exp0,(VarEnv0,Vmt,This),Store0,ref(Ref),Store1),} \\
\text{evaluatelist(Exps,(VarEnv0,Vmt,This),Store1,Vals,Store2),} \\
\text{lookupVmt(Vmt,Store2,Ref,Meth,Varids,Exp1),} \\
\text{zip(Varids,Vals,VarEnv1),} \\
\text{printCall(Ref,Meth,VarEnv1),} \\
\text{evaluate(Exp1,(VarEnv1,Vmt,just(Ref)),Store2,Val,Store3),} \\
\text{printReturn(Val).}
\]

So we basically have added two literals to the body of the rule. The predicate \text{printCall/3} is supposed to print the logging lines “Call:...” and to increment indentation, while the predicate \text{printReturn/1} is supposed to print the logging lines “Return:...” and to decrement indentation. (We omit the trivial definitions of \text{printCall/3} and \text{printReturn/1}.)

REK offers the \text{inject} operator, which allows one to enhance the body of a given rule. We can use this operator to describe the logging functionality as a delta on the rule for method calls:

\[
\text{:- inject}\left(\text{ evaluate(call(_,Meth,_,),_,_,_,_,_)} \\
\text{:- evaluate(_,_,_,ref(Ref),_),} \\
\text{_,} \\
\text{_,} \\
\text{zip(_,_,VarEnv),} \\
\text{\{ printCall(Ref,Meth,VarEnv) \},} \\
\text{evaluate(_,_,_,Val,_)}, \\
\text{\{ printReturn(Val) \} }\right).
\]

That is, the \text{inject} operator takes the ‘sketch’ of a rule that is annotated with additional body literals. The new literals are surrounded by \{\ldots\}. The affected rule is determined by matching the sketch against the existing rule base. ‘Don’t care’ variables, i.e., ‘\_’, can be used in the sketch to abstract from irrelevant literals and predicate positions. The matched and the additional literals can share variables, such as \text{Ref}, \text{Meth}, and others above.

An operator for patching is superior when compared to manual editing because the original rule and the additional functionality remain separated in the source code. This is another form of separation of concerns.

\section{9 Exercises}

We will now compile a list of exercises that are meant to help with the digestion of the evolution scenarios from the preceding sections, and with the further exploration of the overall subject. A good handle on some of the exer-
cises might require further reading, in which case we provide some hints and pointers. Some of the exercises are more like challenges.

Let us first generalise the discussion by giving up on our restriction to Prolog. Other forms of rule-based programming make equally sense, e.g., first-order functional programming, conditional term rewriting, or attribute grammar programming. We also want to look at language-based functionality other than interpreters.

**Exercise 1 (Non-Prolog, non-interpreter examples)**

**a)** Pick an attribute grammar system, and study the following evolution scenario for a frontend that includes static semantics and construction of abstract syntax trees:

a.1) Specify an attribute grammar for a language with assignments.
a.2) Extend the attribute grammar from a.1) to cover records and arrays.
a.3) Describe the move from a.1) to a.2) as an evolutionary transformation. To this end, identify somewhat general operators for evolutionary transformations that seem to be useful in this context.

(A survey on attribute grammar systems with a discussion of techniques for modularity, extensibility, and adaptability can be found in [13].)

**b)** Pick a rewriting framework that provides some sort of traversal strategies [16] (e.g., Stratego, Strafunski, ASF+SDF), and study the following evolution scenario for a Java metrics tool:

b.1) Compute the nesting depth of conditional statements. (The set of conditional Java statements includes if-then-else, while, try and several others. Some related efforts can be found in [17].)
b.2) Adapt the solution such that the nesting depth is computed for each form of conditional statement separately. Identify at least two different designs for the adaptation.
b.3) Describe the move from b.1) to b.2) as an evolutionary transformation.

**c)** Pick a graph transformation engine, such as Progres [25], and study the following evolution scenario for model transformations of (simplified) UML class diagrams in the meta-modelling and MDA context [21]:

c.1) Specify dead-class elimination for a given set of classes that is qualified by a set of public classes to be retained definitely.
c.2) Adapt the solution such that two phases are distinguished. Firstly, unreachable classes are marked for elimination. Secondly, all marked classes are eliminated.
c.3) Describe the move from c.1) to c.2) as an evolutionary transformation.

We will now look at semantics preservation for evolutionary transformations. There are two sorts of semantics-preserving transformations: restructuring
transformations vs. extensions. For instance, the \texttt{group} operator restructures in a semantics-preserving manner: it groups arguments of predicates, while it does not alter or extend the behaviour of the rule-based program at hand. The following exercise concerns the status of other REK operators to be semantics-preserving or not. For the sake of a concrete semantics, we assume that these exercises refer to definite-clause programs (i.e., pure Prolog):

\textbf{Exercise 2 (Semantics-preserving or not?)}

\begin{enumerate}
\item \textit{a)} Explain why the big-step-to-small-steps conversion as of Sec. 3 is semantics-preserving.
\item \textit{b)} \textit{a)} cont’d: Is it a restructuring transformation or an extension?
\item \textit{c)} Identify sufficient conditions for the injected premises in Sec. 8 so that this evolutionary transformation becomes semantics-preserving.
\item \textit{d)} \textit{c)} cont’d: Work out an interpreter example, where injection of some premises could be intentionally semantics-changing.
\item \textit{e)} Is the addition of extra parameters as in Sec. 5 more than restructuring? Is the addition of parameters still semantics-preserving? Explain your answer.
\item \textit{f)} In a strict sense, the \texttt{thread} operator is not semantics-preserving. Provide a simple Prolog snippet to illustrate that the unification caused by threading is not semantics-preserving.
\item \textit{g)} Argue that the sequential composition of adding positions with the \texttt{add} operator and subsequently imposing a data-flow regime on them with the \texttt{thread} operator is semantics-preserving. Provide a simple Prolog snippet to illustrate your explanation.
\end{enumerate}

(The reference \cite{12} is supposed to be helpful.)

While we emphasised semantics preservation above, we should note that evolutionary transformations cannot be restricted to semantics preservation; we also refer to \cite{4,12} to this end. Semantics preservation is an illusion in software maintenance.

What we called restructuring transformations above, is very similar to the notion of refactoring in the sense of Opdyke and Fowler \cite{22,6}: behaviour-preserving transformations that adapt the structure of a program in some useful way — be it to improve the program design or to prepare for program adaptations or extensions. Refactoring is normally linked to OO programming, while little is known about refactoring of rule-based programs — except perhaps for fold/unfold transformations \cite{23}. The following exercise concerns the transposition of refactorings from mainstream OO programming to rule-based programming.
Exercise 3 (Refactoring across language)

a) Discuss the REK operator group (cf. Sec. 7) from the perspective of Fowler’s catalogue for OO refactoring [6]. That is, what known OO refactorings appear to be related to the group operator? What’s the relation?

b) Identify three OO refactorings that can be transposed to the rule-based situation. Illustrate your findings with related pairs of refactoring samples for OO and rule-based programming.

(The reference [11] provides background on “refactoring across language”.)

There is little doubt about the suitability of rule-based programming when it comes to the implementation of language-based functionality. Pattern matching and rule-based decomposition leads to concise, readable, and declarative solutions. The evolution scenarios, that we discussed, are all about first-order rule-based programming. So the question arises whether evolution of language-based functionality is less of a problem in case we employ expressiveness other than first-order rule-based programs, e.g., monadic-style programming, higher-order style, advanced specifications constructs for attribute grammars or SOS? Would such extra expressiveness eliminate the need for (invasive) evolutionary transformations? What are the trade-offs with regard to complexity of the formalism, required amount of anticipated evolution, availability of the apparatus for formal reasoning, and others. In the following, we deal with monadic style and Modular SOS by means of an exercise. In [13], we have discussed related issues for extensions of the basic attribute grammar paradigm.

Exercise 4 (Beyond first-order rule-based programming)

a) Reconstruct the series of evolution scenarios from the previous sections in Haskell, possibly employing higher-order functions and monadic style [29].

Background: A selling point of the monadic style is that we could use it for any kind of environment and statement passing rather than ‘weaving data flow’, as we did for sorts vmt, this and store in Sec. 5. So if we wrote the initial interpreter in monadic style, then the effort in Sec. 5 can be saved. The migration from non-monadic to monadic style, when necessary, is a invasive transformation by itself. Haskell does not suggest straightforward non-transformational solutions for the other evolution scenarios.

b) We also use monadic-style interpreters for the following scenario. Specify an interpreter with different functions for expression evaluation and statement execution, while expression evaluation is free of side effects for the language at hand. There are the following options for using monadic style in this situation:

b.1) Both interpreter functions live in the same monad.
b.2) Monad transformation is used to leave out state where not needed.

Discuss the loss of precision associated with b.1. Find more complex examples where the idea of ‘one monad everywhere’ would really stop to be appealing. Discuss the evolution problems associated with b.2, i.e., for what sort of evolution scenario would this encoding need to be adapted?

c) Reconstruct the series of evolution scenarios from the previous sections in Modular SOS (MSOS; [20]), possibly employing small-step style, label categories for semantic objects, and supplementary idioms. In detail:

c.1) Compare the use of transition labels with the use of monadic style in Ex. 4.b. In particular, illustrate the reappearance of issue b.1. Also, is there any MSOS analogue for issue b.2?

c.2) Attempt to circumvent the invasive injection from Sec. 8 by using small-step style, by setting up the term structure of subjects accordingly, by perhaps even considering an extra elaboration phase before expression evaluation, where the concrete syntax is mapped to an annotated abstract syntax.

10 Wrap-up

The subfield of software evolution that deals with language-based functionality is a relatively young field. This field however receives input from other disciplines such as program and data refinement, program synthesis, transformational program development (from specifications), data re-engineering, and grammarware engineering. Evolution of language-based functionality is an emerging field whose relevance is emphasised by language engineering for model-driven software development; see the recent Dagstuhl seminar on this topic: http://www.dagstuhl.de/04101/. That is, the increasing importance of modelling and meta-modelling in software development calls for better understanding of the evolution of languages (or meta-models) and language-based functionality (or model-driven transformations).

We have studied evolution of language-based functionality in the context of rule-based programming. We have deployed evolutionary transformations as a general method for restructuring, extending, shrinking and revising language-based functionality (and other rule-based programs).

Future work on the subject has to provide practically useful tool support for the evolution of rule-based programs, a comprehensive analysis of basic and composed evolution operators, and a meaningful, formal model of evolution, with coverage of transformations that are not strictly semantics-preserving.
References


