Refactoring Prolog programs

Tom Schrijvers
Alexander Serebrenik
Bart Demoen

Refactoring Prolog programs

Tom Schrijvers
Alexander Serebrenik
Bart Demoen

Department of Computer Science, K.U.Leuven

Abstract

Refactoring is a technique to restructure code in a disciplined way originating from the OO-community. It aims to improve software readability, maintainability and extensibility. Unlike the existing results on program transformation refactoring can require user input to take certain decisions. In this paper we apply the ideas of refactoring to Prolog programs. We start by presenting a catalogue of refactorings. Then we discuss `ViPReSS` , our refactoring browser, and our experience with applying `ViPReSS` to a big Prolog legacy system.

Refactoring Prolog programs

Tom Schrijvers*, Alexander Serebrenik, Bart Demoen

Department of Computer Science, K.U. Leuven
Celestijnenlaan 200A, B-3001, Heverlee, Belgium
E-mail: {Tom.Schrijvers, Alexander.Serebrenik, Bart.Demoen}@cs.kuleuven.ac.be

Abstract. *Refactoring* is a technique to restructure code in a disciplined way originating from the OO-community. It aims at improving software readability, maintainability and extensibility. In this paper we apply the ideas of refactoring to Prolog programs. We start by presenting a catalogue of refactorings. Then we discuss `ViPReSS`, our refactoring browser, and our experience with applying `ViPReSS` to a large Prolog legacy system.
Keywords: refactoring, Prolog, logic programming, software engineering

1 Introduction

Program changes take up a substantial part of the entire programming effort. Often changes are required to incorporate additional functionality or to improve the efficiency. In both cases, a preliminary step of improving the design without altering the external behaviour can be recommended. This methodology, called *refactoring*, emerged from a number of pioneer results in the OO-community [8, 19, 22, 23] and recently came to prominence for functional languages [14, 31]. More formally, refactoring is a source-to-source program transformation that changes program structure and organisation, but not program functionality. The major aim of refactoring is to improve readability, maintainability and extensibility of the existing software. While performance improvement is not considered as a crucial issue for refactoring, it should be noted that well-structured software is more amenable to performance tuning. We also observe that certain techniques that were developed in the context of program optimisation can improve program organisation and, therefore, can be considered as refactoring techniques.

To achieve the goals above three questions have to be answered, namely, *what* transformation should be used, and *where* and *how* transformations have to be performed. Neither of the steps, unlike the results on automated program transformation, aims at transforming the program entirely automatically. The decision on whether the transformation should be applied is left to the program

---

developer. However, providing automated support for refactoring is a useful and important challenge.

Deciding automatically *where* to apply a transformation can be a difficult task on its own. A number of ways to resolve this may be considered. First, programming analysis approaches can be used. For example, it is common practice while ordering predicate arguments to start with the input arguments and end with the output arguments. Using mode information can be used to detect when this rule is violated and to suggest the user to reorder the arguments. Second, one can try and predict further refactorings based on the program transformations already applied. For instance, if constraints have been simplified by using De Morgan's rules, the system can propose to apply the same rules to rewrite additional constraints. Eventually useful sequences of refactoring steps can be learned analogously to the automated macro construction [12]. By following these approaches the automatic refactoring tools, so called *refactoring browsers*, can be expected to make suggestions on when and how refactoring transformations should be applied. These suggestions should then be either confirmed or rejected by the program developer.

Answering *how* the program should be transformed might also require the user's input. We illustrate this point by observing that one possible refactoring renames a predicate: while automatic tools can hardly be intelligent enough to guess the new predicate name, they should be able to detect all program points affected by the change. Moreover, assuming certain properties, such as absence of the user-defined meta-predicates, on the original program can lead to a more readable and compact transformed program. In this case the user is requested to confirm that the properties in question indeed hold.

Logic programming languages and refactoring have already been put together at different levels. Tarau [30] discusses re-implementation of the Prolog language itself on the conceptual level and he calls this refactoring. However, this approach differs significantly from the traditional notion of refactoring as it has been introduced by Fowler [8]. We follow the latter definition. Recent relevant work is [32] in the context of object oriented languages: a meta logic very similar to Prolog is used to detect for instance obsolete parameters. Refactoring of such code consists in removing obsolete parameters and also the problem of cascading refactoring is discussed. It is important that their method for detecting refactorings and for the refactorings themselves is based on logic. None of these papers, however, considered applying refactoring techniques to logic programs. In our previous paper [27] we have emphasised the importance of refactoring for logic programming and discussed the applicability of the refactoring techniques developed for object-oriented languages [7] to Prolog and CLP-programs. Seipel *et al.* [26] included refactoring among the analysis and

visualisation techniques that can be easily implemented by means of FNQUERY, a Prolog-inspired query language for XML. However, the discussion stayed at the level of an example and no detailed study has been conducted.

The contributions of the current paper are twofold. We start by presenting a refactoring catalogue for Prolog in Section 2. While some of the transformations could be borrowed from the OO-world we found it useful to design a number of techniques exclusively for logic programming. Next, in Section 3 we introduce ViPReSS , our refactoring browser, implementing refactorings of the catalogue. ViPReSS has been successfully applied for refactoring a 50,000 lines-long legacy system developed and used at K.U.Leuven to manage educational activities of the Computer Science department. Finally, in Section 4 we conclude. A detailed refactoring example is given in the Appendix.

2 Catalogue of Prolog refactorings

In this section we present a number of refactorings that we have found to be useful when Prolog programs are considered. We stress that the programs are not limited to pure logic programs, but may contain various built-ins such as those defined in the ISO standard [2]. The only exception are higher-order constructs that are not dealt with automatically, but manually. Automating the detection and handling of higher-order predicates is an important part of future work.

The refactorings in this catalogue are grouped by scope. The scope expresses the user-selected target of a particular refactoring. While the particular refactoring may affect code outside of the selected scope, it is only because the refactoring operation detects a dependency outside the scope.

For Prolog programs we distinguish the following four scopes, based on the code units of Prolog. It should be noted that a similar distinction can be made for other languages based on their code units.

– The *system* scope encompasses the entire code base. Hence the user does not want to transform a particular subpart, but to affect the entire system as a whole. An example is dead code detection and removal. The deadness property can in general only be inferred with the entire system in scope.
  This scope is evidently appropriate for all programming paradigms. It might even apply to multi-language systems.
– The *module* scope considers a particular module. Usually a module is implementing a well-defined functionality and is typically arranged as one file. For example, a module can be renamed, split into separate modules or several modules can be merged together.
  This scope corresponds to refactorings operating on classes in Java or modules in Haskell.

– The *predicate* scope targets a single predicate. For example, the moving of a predicate to a different module, affects the predicate directly. The code that depends on the predicate may need updating as well. But this is considered an implication of the refactoring of which either the user is alerted or the necessary transformations are performed implicitly.

  The counterparts of predicates in Prolog are methods in Java and functions in Haskell.

– The *clause* scope affects a single clause in a predicate. Usually, this does not affect any code outside of the clause directly. An example is when a part of the clause body is replaced by a call to a new predicate that consists of the original goal. Code outside of the clause scope is affected in as much as that the new predicate is introduced and the necessary import declarations are added or removed.

  In object-oriented and functional programming languages this scope affects methods and function bodies respectively.

2.1 System scope refactorings

In this subsection we present a number of program transformations that can modify the system as a whole. Incomplete information about the system can compromise correctness of these transformations. For instance, code can be erroneously identified as unreachable and eliminated. It also should be observed that applying these refactorings requires systematically searching in thousands lines of code. Hence, providing automatic support for these techniques is not only desired but essential. Given a refactoring browser supporting these refactorings one can always recommend the programmer to start with them as they can improve her general understanding of the system structure and do not require familiarity with implementation details.

Dead code detection and elimination
This refactoring aims at removing the so called "dead code", i.e., code that cannot be reached during any possible program execution. Observe that while Prolog compilers can be capable of eliminating dead code for efficiency reasons this optimisation is usually not visible for the user. However, dead code is not only useless in general but it also clutters the program. Hence, explicitly removing the dead code improves the structure of the program and leads to better readable and maintainable code.

We consider a predicate definition in its entirety as a code unit that can be dead, as opposed to a clause or a subgoal in the body of a clause. Usually the predicate in its entirety implements a complete relation, as opposed to a true subset of its clauses. Hence, when clauses are removed from the predicate

4

definition, its meaning is altered. While this may preserve the semantics of the program as it is, it can certainly lead to errors and confusion in future uses of the predicate. However, if the entire predicate definition is eliminated because it is not used, no erroneous reuse is possible.

This refactoring consists of two steps, detection and elimination, that should be clearly separated. At the first step, the refactoring browser marks the unreachable code. Next the user can decide whether the marked code should be removed, commented out or left as is. Our experience suggests that big systems that evolve over many years tend to have a significant number of unreachable predicates implementing obsolete functionality, like for example in the case study presented in Section 3.

An additional question that should be considered in the context of dead code elimination is detecting and eliminating "dead" imports. In other words, a module should import only those predicates from only those modules that are genuinely needed. Again, eliminating "dead" imports and modules simplifies relations between the modules in the system and therefore, improves its structure as a whole.

Identifying and removing predicate duplication

Predicate duplication or cloning is a well-known problem usually resulting from a bad programming practice known as "copy and paste". One of the biggest problems with the duplicated code is its bad maintainability. Indeed, changes to the duplicated code should be applied everywhere it has been cloned. Hence, identifying and removing such predicates improves the maintainability of the system. Moreover, duplicate code can hint at a design flaw: functionality that should have been shared between different modules was not taken into account. Thus, the refactoring in question can improve the overall system structure. Similarly to dead code detection and elimination, identifying and removing predicate duplication consists of two steps. First, mark the duplicate predicates. Second, based on user input, either remove all but one of the clones, replace all clones by a single new copy in a separate module or do not perform any changes at all.

Predicate extraction

Despite its name this refactoring is also a system scope technique. Predicate extraction aims to discover *common* functionality and to extract it. Unlike the previous refactoring technique predicate definitions are not duplicated in their entirety but identical subsequences of goals are called in different places.

We suggest hence to introduce a new predicate corresponding to these subsequences. By doing this we aim to improve readability of the program (bodies of the clauses will become shorter; the new predicate in general will capture the

intended meaning of the sequence of calls better than the sequence itself), its structure and maintainability.

More formally, predicate extraction starts by scanning all clauses of the program and identifying commonly repeated predicate sequences in their bodies. Next, the user inspects the sequences and chooses those that she wants to use as predicates definitions. Finally, the corresponding parts of the clauses are automatically replaced by the calls to the new predicate.

Predicate extraction seems to be very promising in the context of domain specific languages. Indeed, given a set of combinators it can be used as a tool to find a good set of operators which can be used to generate all other combinators. Finding such a set is usually considered a challenging task in the domain specific language development [21].

Remove argument from predicate
The basic intuition behind this refactoring is that the parameters that are no longer used by a predicate should be dropped. This problem has been studied, among others, by Leuschel and Sørensen [13] in the context of logic programs specialisation, and by Alpuente *et al.* [4] for term rewrite systems.

Leuschel and Sørensen established that, in general, removing *all* redundant arguments from a given program is undecidable. They also suggested two techniques to find safe and effective approximations: top-down goal-oriented RAF and bottom-up goal-independent FAR. We believe that in the context of refactoring FAR is the more useful technique. First of all, FAR is the only possibility if exported predicates are considered. Moreover, the refactoring-based software development regards the development process as a sequence of small "change - refactor - test" steps. These changes, hence, most probably will be local.

The argument-removing technique should consist of two steps. First, unused argument positions are marked. Second, depending on user input, marked argument positions are dropped. Similarly to removing unused predicates (dead code elimination) by removing unused argument positions from predicates we improve readability of the existing code. The core of the process is therefore the marking step.

FAR marks an argument position in the head of the clause as unused if it is occupied by a variable that appears exactly once in the argument position that has not been marked as unused. The marking process proceeds bottom-up per strongly connected component (SCC) of the predicate dependency graph. The SCCs are considered according to the topological ordering on the predicate dependency graph.

Example 1. To illustrate this recursive definition consider the following example:

```
┌─────────────────────── Remove Argument Example ───────────────────────┐
│                                                                        │
│   p(X) :- q(X̱), r(1).                                                  │
│   q(Y̱).                                                                │
│                                                                        │
└────────────────────────────────────────────────────────────────────────┘
```

Since predicate p depends on q and r, we start by these predicates and then consider p. The first argument position of q is occupied by the variable Y. This argument position is unused since Y appears only once in the second clause. We will show next that the first argument position of p is also unused. First, it is occupied by a variable (X). Second, it appears in the head of the clause and in the call to q as its first argument position. However, the first argument position of q is marked as unused. Thus, X appears exactly once in the argument position that has *not* been marked as unused, i.e., in the head of the clause for p. Hence, the first argument position of p is also unused.   □

Rename functor

The purpose of this refactoring is to improve the readability of the code by providing a functor with a more meaningful name. This technique is a system-scope refactoring since the same functor can be repeatedly used in different parts of the system. We suggest to apply this technique when the same name is used to denote a functor and a predicate or when the same functor is used in different contexts. To illustrate the latter point, recall that in the famous map colouring example of Sterling and Shapiro [29] functor *cons* is used both to denote the list of countries on the map and the list of neighbours of each one of the countries.

Hide predicates

This refactoring technique "hides" predicates exported by a module but unused elsewhere in the system, i.e., it removes the export declarations for these predicates.

2.2 Module scope refactorings

Refactoring techniques presented in this subsection consider modules—well-defined collections of predicates that usually implement a part of the system functionality and typically are located in one file. Modules can be regarded as basic building blocks for developing large-scale applications. Unlike the system scope refactorings described above module scope refactorings can be applied when only a part of a system is available for refactoring. This typically will be the case when libraries or multi-language software are considered.

Rename module
This refactoring technique should be applied when the name of the module no longer corresponds to the functionality it implements. Renaming a module involves updating the import statements in modules that use it.

Split module
Splitting module can be seen as a pendant of the "extract class" refactoring suggested for object-oriented languages by Fowler [7]. The refactoring proceeds in two steps. First, a module splitting is suggested. Then, after user confirmation the program is transformed and relevant import statements are updated. This refactoring is most profitable for large modules performing a number of non-related tasks.

Merge modules
Merging a number of modules in one can be advantageous in case of strong interdependency of the modules involved. This refactoring follows a well-known programming guideline that suggests to avoid circular dependencies between modules. Refactoring browsers are expected to discover interrelated modules and, by taking software metrics as above into account, to suggest the modules to be merged. Upon user confirmation the actual transformation can be performed.

Dead code detection and elimination
Problem of the dead code detection and elimination can also be considered on the level of module. As above, the problem is in general undecidable but (in absence of higher-order predicates) it can be safely approximated. Predicate dependency graph can be, for instance, used to this end. The latter approach is taken by some of the state-of-the-art Prolog compilers such as XSB [24]: when a potentially unreachable predicate is discovered, a warning is issued.

2.3 Predicate scope refactorings

Many important refactorings are applied to a given predicate. The predicate is the basic construct in logic programming and it can be seen as a counterpart of methods in object-oriented languages. Hence, the refactorings we discuss in this subsection are somehow similar to method scope refactorings of the on-line refactoring catalogue [7].

Rename predicate
This is the exact counterpart of the "rename method" refactoring. It aims at improving the readability of the code and should be applied when the name of the predicate does not reveal its purpose. Renaming a predicate requires updating

8

the calls to it as well as the interface between the module defining the predicate and modules using it. While this refactoring might sound trivial it is still useful: to our uttermost surprise we have discovered a predicate named q in our case study system.

Move predicate

This refactoring is quite similarly to "rename predicate". It corresponds exactly to the "move method" refactoring of Fowler [7]. Moving predicate from one module to another one can improve the overall structure of the program by bringing together interdependent or related predicates. This refactoring has also been used as one of the internal steps during code duplication elimination.

Add argument to predicate

This refactoring should be applied when a callee needs more information from its (direct or indirect) caller. Our experience suggests that the situation is very common while developing Prolog programs. It can be illustrated by the following example:

Example 2. Consider the following program.

```
 ――――――――――― Original Code ―――――――――――

  compiler(Program,CompiledCode) :-
          translate(Program,Translated),
          optimise(Translated,CompiledCode).
  optimise([assignment(Var,Expr)|Statements],CompiledCode) :-
          optimise_assignment(Expr,OptimisedExpr), ...
  optimise([while(Test,Body)|Statements],CompiledCode) :-
          optimise_test(Test,OptimisedTest), ...
  optimise([if(Test,Then,Else)|Statements],CompiledCode) :-
          optimise_test(Test,OptimisedTest), ...
  optimise_test(Test,OptimisedTest) :- ...
```

Assume that a new code-based test analysis (analyse) has been implemented. Since this analysis requires the original program code as an input, the only place to plug the call to analyse is in the body of compiler:

```
 ――――――――――― Extended Code ―――――――――――

  compiler(Program,CompiledCode) :-
          analyse(Program,AnalysisResults),
          translate(Program,Translated),
          optimise(Translated,CompiledCode).
```

9

In order to profit from the results of `analyse` the variable `AnalysisResults` should be passed all the way down to `optimise_test`. In other words, an extra argument should be added to `optimise` and `optimise_test` and its value should be initialised to `AnalysisResults`. $\square$

Hence, given a variable in the body of the caller and the name of the callee, the refactoring browser should be powerful enough to propagate this variable along all possible computation paths from the caller to the callee. This refactoring is an important preliminary step preceding additional functionality integration or efficiency improvement.

Argument reordering
Our experience suggests that while writing predicate definitions Prolog programmers tend to begin with the input arguments and to end with the output arguments. This methodology has been identified as a good practice and even further refined by O'Keefe [18] to more elaborate rules. Unfortunately, this practice can be violated when additional arguments are added later. We observed that failure to confirm to this "input first output last" expectation pattern is experienced as very confusing. Hence, in order to improve readability of the code argument reordering can be suggested: given the predicate name and the intended order of the arguments, the refactoring browser should produce the code such that the arguments of the predicate have been appropriately reordered.

It also should be noted that most Prolog systems use indexing on the first argument. Note that argument reordering can improve the efficiency of the program execution in this way.

2.4   Clause scope refactorings

Refactoring techniques considered in this subsection affect directly a clause (or a number of clauses). Unlike the refactorings discussed in the previous subsection this clause (or clauses) usually constitute a subset of a predicate definition.

If-Then-Else introduction
This technique aims at improving program readability by replacing some of the cuts (!) appearing in it by if-then-else (`->` `;`) control structures. Cut was introduced in order to prune Prolog's search space during program execution. Despite the controversy on the use of cut inside the logic programming community, and recurring attempts to banish it from Prolog [5, 10, 17], it is commonly used in practical applications both for efficiency and for correctness reasons. We suggest a transformation that replaces some uses of cut by the more declarative and potentially more efficient if-then-else.

10

Example 3. Our transformation allows us to replace the left-hand side program with the right-hand side program:

```
                       ── If-Then-Else Introduction ──
┌─────────────────────────────────────────────────────────────────┐
│                                                                   │
│  fac(0,1) :- !.             fac(N,F) :-                           │
│  fac(N,F) :-                    ( N = 0, F = 1 ->                 │
│        N1 is N - 1,                 true                          │
│        fac(N1,F1),              ;                                 │
│        F is N * F1.                 N1 is N - 1,                  │
│                                     fac(N1,F1),                   │
│                                     F is N * F1                   │
│                                 ).                                │
│                                                                   │
└─────────────────────────────────────────────────────────────────┘
```

□

The right-hand side program shows that the refactoring preserves operational semantics. Moreover, assuming that `N` is the input and `F` the output of `fac/2`, the refactoring reveals hidden malpractices. Firstly, the generation of output before a commit (cut), disapproved for example by O'Keefe in [18], is exposed. To illustrate this issue consider what happens with a call `fac(0,2)`: instead of failing it does not terminate. Secondly, a full unification is used in `N = 0` instead of the numerical comparison `N =:= 0` or maybe even better the numerical inequality `N =< 0` which turns the predicate into a total function.

If-Then-Else inversion
The idea behind this transformation is that while logically the order of the "then" and the "else" branches does not matter, it can be important for code readability.

Indeed, an important readability criterion is to have an intuitive and simple condition. The semantics of the if-then-else construct in Prolog have been for years a source of controversy [1] until it was finally decided in the ISO standard [2]. The main issue is that its semantics differ greatly from those of other programming languages. Restricting oneself to only conditions that do not bind variables but only perform tests[1], makes it easier to understand the meaning of the if-then-else.

To enhance readability it might be worth putting the shorter branch as "then" and the longer one as "else". Alternatively, the negation of the condition may be more readable, for example a double negation can be eliminated. In addition this transformation might disclose other transformations that simplify or clean up the code.

---

[1] This is similar to the guideline in imperative languages not to use assignments or (boolean) functions with side effects in conditions.

Hence, we suggest a technique replacing (P -> Q ; R) with (\+ P -> R ; P, Q). Of course, for a built-in P the appropriate negated built-in is chosen instead of \+ P.

The call to P in the "else" branch is there to keep any bindings generated in P. If it can be inferred that P cannot generate any bindings (e.g. because P is a built-in known not to generate any bindings), then P can be omitted from the "else" branch.

Example 4. Consider the following example:

```
───────────── If-Then-Else Inversion ─────────────

  p(L, M) :-                         p(L, M) :-
      ( L = [] ->                        ( L \= [] ->
            M = empty                          M = non-empty
      ;                                  ;
            M = non-empty                      L = [],
      ).                                       M = empty
                                         ).

```

Note that the program on the right-hand side does not create bindings in the condition. Omitting L = [] on the right-hand side leads to a wrong answer when the first argument of p is free. However it may not be the intention of the programmer to do this full unification. Then the ugliness of the new code encourages him to consider this and replace L \= [] with L \== [] and simply omit L = [] if this corresponds to his intended (operational) semantics.

□

Local predicate extraction

Similarly to the system-scope refactoring with the same name this technique replaces body subgoals with a call to a new predicate defined by these subgoals. Unlike system-scope predicate extraction here we do not aim to automatically discover useful candidates for replacement or to replace similar sequences in the entire system. Given a part of a clause body marked by the user and a name for a new predicate, this part is replaced by the appropriate call to the new predicate and the new predicate is defined by using the marked part. By restructuring a clause this refactoring technique can improve its readability. Suitable candidates for this transformation are clauses with overly large bodies or clauses performing two distinct subtasks. By cutting the bodies of clauses down to size and isolating subtasks, it becomes easier for programmers to "grok" their meaning.

Example 5. The following example illustrates how the program on the left can be improved with local predicate extraction to the program on the right:

```
┌──────────────── If-Then-Else Inversion ────────────────┐

  weather_in_us(City) :-        weather_in_us(City) :-
      temperature(City,TF),         temperature(City,TF),
      Temp is TF - 32,              f2c(TF, TC),
      TC is Temp * 5/9,             report(City, TC).
      write('Temperature in '),  f2c(TF,TC) :-
      write(City), write(' is '),    Temp is TF - 32,
      write(TC), write(' degrees     TC is Temp * 5/9.
      centigrade.'), nl.         report(City, TC) :-
                                     write('Temperature in '),
                                     write(City), write(' is '),
                                     write(TC), write(' degrees
                                      centigrade.'), nl.

└────────────────────────────────────────────────────────┘
```

□

Unfolding

In the context of logic programming unfolding is one of the best-known program transformations. Given a clause $H \leftarrow B_1, \ldots, B_n$ it replaces a body subgoal B_i with the appropriately instantiated body of a clause $H' \leftarrow B'_1, \ldots, B'_{n'}$ such that H' and B_i are unifiable. Formally, if $mgu(H', B_i) = \theta$ the result of the unfolding is $(H \leftarrow B_1, \ldots, B_{i-1}, B'_1, \ldots, B'_{n'}, B_{i+1}, \ldots, B_n)\theta$. Object-oriented counterparts of unfolding would be so called "middle man elimination" and "method inlining". While in general unfolding does not necessarily result in a more readable predicate definition it can be useful in specific cases such as the following example.

Example 6.

```
┌──────────────────────── Unfolding ────────────────────────┐

  :- module(test, [p/1]).  :- module(test, [p/1]).   :- module(test, [p/1]).

  p(s(X)) :- q(X).         p(s(s(X))) :- r(X).        p(s(s(X))) :- r(X).
  q(s(X)) :- r(X).         q(s(X))    :- r(X).        r(X)        :- ...
  r(X)    :- ...           r(X)       :- ...

└───────────────────────────────────────────────────────────┘
```

In the `test` module on the left-hand side the call to `q` can be unfolded, resulting in the program in the middle. At the next step a dead code elimination refactoring can be applied to remove the (now unreachable) code for `q`.   □

Logical loops

13

Logical loops have been introduced by Schimpf [25]. It has been claimed that structures of the form `foreach(Element,List) do Body` are better readable than the traditional Prolog definition `loop([Element|Remainder]) :- Body, loop(Remainder).`. Clearly, logical loops are closer both to traditional mathematical notation $\forall x \in S$ and the `for`-loops in imperative languages. However, trained Prolog programmers might experience logical loops as *less intuitive* since recursion is hidden in the call to `foreach`. Still, we believe that replacing (direct) recursion with a logical loop should be considered as an important refactoring technique [26]. Refactoring browsers can be expected to detect potential candidates for refactoring and to perform the transformation upon user confirmation.

3 The `ViPReSS` refactoring browser

The refactoring techniques presented above have been implemented in the `ViPReSS` refactoring browser. In order to facilitate acceptance of the tool `ViPReSS` by the developers community has been implemented not as a stand-alone application but on the basis of VIM (`http://www.vim.org/`), a popular clone of the well-known VI editor that for the third year in a row has been voted as a favourite editor by Linux Journal readers [3]. Moreover, techniques like *predicate duplication* provided by refactoring browsers such as HaRe [14] are already included in the underlying editor facilities.

Most of the refactoring tasks have been implemented as SICStus Prolog programs inspecting source files and/or call graphs. Updates to files have been implemented either directly in the scripting language of VIM or, in the case many files had to be updated at once, through `ed` scripts. VIM functions to initiate the refactorings and to get user input have been written.

`ViPReSS` has been successfully applied to a large (more than 53,000 lines) legacy system used at the Computer Science department of Katholieke Universiteit Leuven to manage the teaching activities. The system, called BTW (Flemish for value-added tax), has been under development since the early eighties. System implementation involved more than ten different programmers, many of whom are no longer employed by the department. The implementation has been done in MasterProLog [11], software product that to the best of our knowledge will be no longer supported starting from spring 2004. We believe that this mix of complications is typical for real-world applications. By using the refactoring techniques we aimed to obtain a better understanding of the BTW -system, to improve its structure and maintainability, and to prepare it for further intended changes such as porting it to a state-of-the-art Prolog system and adapting it

to new educational tasks the department is facing as a part of introducing the unified Bachelor-Master system in Europe.

We started by removing some parts of the system that have been identified by the expert as obsolete. These parts included out-of-fashion user interfaces and outdated versions of program files. The bulk of dead code was eliminated in this way, reducing the system size to a mere 20,000 lines. It should be noted that the system scope refactorings could have been applied after simply removing the outdated toplevel predicates.

Next, we applied most of the system-scope refactorings described above. Even after removal of dead code by the experts `ViPReSS` identified and eliminated 299 dead predicates. This reduced the the size by another 1,500 lines. Moreover `ViPReSS` discovered 79 pairwise identical predicates. In most of the cases, identical predicates were moved to new modules used by the original ones. The previous steps allowed us to improve the overall structure of the program by reducing the number of system files from 294 to 116. The size of the system has simultaneously diminished to 18,000 lines. Very little time was spent to bring the system into this state. The experts were sufficiently familiar with the system to immediately identify obsolete parts. The system-scope refactorings took only a few minutes each.

The second step of refactoring consisted of a thorough code inspection aimed at local improvement. Many malpractices have been identified: excessive use of cut combined with producing the output before commit being the most notorious one. Additional "bad smells" we discovered include bad predicate names such as q, unused arguments and unifications used instead of identity checks or numerical equalities. Some of these were located by `ViPReSS` , other ones were recognised by the users, while `ViPReSS` was used to perform the corresponding transformations. This step is more demanding of the user. She has to consider all potential candidates for refactoring separately and decide on what transformations apply. Hence, the lion's share of the refactoring time is spent on these local changes.

To summarise our experience with the case study we learned that automatic support for refactoring techniques is essential and that `ViPReSS` is well-suited for this task. As the result of applying refactoring to BTW we obtained a better-structured lumber-free code. This code is not only more readable and understandable but it also simplifies implementing the future intended changes. Considering the relative time investments of the global and the local refactorings, we recommend to start out with the global ones and then selectively apply local refactorings as the need occurs.

4 Conclusions and Future Work

In this paper we have shown that the ideas of refactoring, originating in the OO-community, are applicable and important for logic programming. Refactoring promises to bridge the existing gap between prototypes and real-world applications. Indeed, extending the prototype to provide additional functionality often leads to cumbersome code. Refactoring allows software developers both to clean up the code after changes have been introduced and to prepare the code for changes intended in the future.

We have presented a catalogue of refactorings at different scopes of a program, containing both previously known refactorings for object-oriented languages now adapted for Prolog and entirely new Prolog-specific refactorings. Although the presented refactorings do require human input to make certain choices, as it is in the general spirit of refactoring, a large part of the work can be automated. Indeed, we have implemented the automatable parts of the presented refactorings and implemented them in the existing editor VIM, naming the resulting system `ViPReSS` .

In the logic programming community questions related to refactoring have been intensively studied in context of program transformation and specialisation [6, 13, 20]. There are two important differences with this line of work. First of all, refactoring does not aim at optimising program performance in terms of CPU time or memory use but at improving program readability, maintainability and extensibility. Second, user input is essential in the refactoring process while traditionally only automatic approaches were considered. Moreover, usually program transformations are designed to be implemented in the compiler and hence, they are "invisible" to the program developer. However, it should be noted that some of the transformations developed for program optimisation can be considered as refactorings and should be implemented in refactoring browsers. This is the case, for instance, for *dead code elimination* and *removing redundant arguments*.

To further increase the level of automation of particular refactorings additional information is necessary such as types and modes. This allows for example to reorder the arguments in a predicate putting the output variables last as recommended by [18], or to improve the output of if-then-else inversion. To obtain this information the refactoring system could be extended with type and mode analyses [15, 16]. On the other hand, it seems worthwhile to consider the proposed refactorings in the context of languages with type and mode declarations as Mercury [28] and HAL [9], especially as these languages claim to be of greater industrial relevance than traditional Prolog.

Moreover, dealing with higher order is essential for refactoring in a real world context. The above mentioned languages with explicit declarations for such constructs would again facilitate the implementation of an industrial strength refactoring environment.

References

1. The `->` operator. *Association for Logic Programming Newsletter*, 4(2):10–12, 1991.
2. *Information technology—Programming languages—Prolog—Part 1: General core*. ISO/IEC, 1995. ISO/IEC 13211-1:1995.
3. Linux journal announces winners of ninth annual readers' choice awards. *Linux Journal*, 115, nov 2003.
4. M. Alpuente, S. Escobar, and S. Lucas. Removing redundant arguments of functions. In H. Kirchner and C. Ringeissen, editors, *9th International Conference on Algebraic Methodology And Software Technology (AMAST 2002)*, volume 2422 of *LNCS*, pages 117–131. Springer Verlag, 2002.
5. S. K. Debray and D. S. Warren. Towards banishing the cut from Prolog. *IEEE Transactions on Software Engineering*, 16(3):335–349, 1990.
6. S. Etalle, M. Gabbrielli, and M. C. Meo. Transformations of CCP programs. *ACM Transactions on Programming Languages and Systems*, 23(3):304–395, May 2001.
7. M. Fowler. Refactorings in alphabetical order. Available at `http://www.refactoring.com/catalog/index.html`, 2003.
8. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999.
9. M. García de la Banda, B. Demoen, K. Marriott, and P. Stuckey. To the gates of HAL: a HAL tutorial. In *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, number 2441 in LNCS, pages 47–66. Springer-Verlag, 2002.
10. M. Hoshida and M. Tokoro. ALEX: The logic programming language with explicit control and without cut-operators. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Logic Programming '88, Proceedings of the 7th Conference, Tokyo, Japan, April 11-14, 1988*, volume 383 of *LNCS*, pages 82–95. Springer Verlag, 1989.
11. IT Masters. MasterProLog Programming Environment. `http://www.itmasters.com/`, 2000.
12. N. Jacobs and H. Blockeel. The learning shell : Automated macro construction. In M. Bauer and P. Gmytrasiewicz, editors, *User Modeling 2001*, volume 2109 of *Lecture Notes in Artificial Intelligence*, pages 34–43. Springer Verlag, 2001.
13. M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Proceedings of the 6th International Workshop on Logic Program Synthesis and Transformation*, volume 1207 of *Lecture Notes in Computer Science*, pages 83–103. Springer Verlag, 1996.
14. H. Li, C. Reinke, and S. Thompson. Tool support for refactoring functional programs. In J. Jeuring, editor, *ACM SIGPLAN 2003 Haskell Workshop*. Association for Computing Machinery, 2003.
15. L. Lu. A mode analysis of logic programs by abstract interpretation. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Perspectives of System Informatics, Second International Andrei Ershov Memorial Conference, Akademgorodok, Novosibirsk, Russia, June 25-28, 1996, Proceedings*, volume 1181 of *Lecture Notes in Computer Science*, pages 362–373. Springer Verlag, 1996.

16. L. Lu. A precise type analysis of logic programs. In *Proceedings of the 2nd international ACM SIGPLAN conference on on Principles and practice of declarative programming, September 20-23, 2000, Montreal, Canada*, pages 214–225. ACM Press, 2000.

17. C. Moss. Cut and paste — defining the impure primitives of prolog. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 686–694, London, England, jul 1986. published as Lecture Notes in Computer Science 225 by Springer-Verlag.

18. R. A. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, MA, USA, 1994.

19. W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

20. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19/20:261–320, May/July 1994.

21. S. Peyton Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering. *ACM SIGPLAN Notices*, 35(9):280–292, 2000.

22. D. Roberts. *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

23. D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of ObjectSystems (TAPOS)*, 3(4):253–263, 1997.

24. K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, S. Dawson, and M. Kifer. The XSB Programmer's Manual: version 2.5, vols. 1 and 2, 2001.

25. J. Schimpf. Logical loops. In *Proceedings of the 18th International Conference on Logic Programming, ICLP 2002, Copenhagen, Denmark*, pages 224–238, 2002.

26. D. Seipel, M. Hopfner, and B. Heumesser. Analysing and visualizing Prolog programs based on XML representations. In F. Mesnard and A. Serebrenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 31–45, 2003. Published as a technical report CW 371 of Katholieke Universiteit Leuven.

27. A. Serebrenik and B. Demoen. Refactoring logic programs. Ninetheen International Conference on Logic Programming, ICLP 2003, Mumbay, India, December 9-13, 2003, TATA Institute of Fundamental Research, 2003.

28. Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Australian Computer Science Conference*, pages 499–512, February 1995.

29. L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, MA, USA, 1994.

30. P. Tarau. Fluents: A refactoring of Prolog for uniform reflection an interoperation with external objects. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Moniz Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *LNAI*, pages 1225–1239. Springer Verlag, 2000.

31. S. Thompson and C. Reinke. Refactoring Functional Programs. Technical Report 16-01, Computing Laboratory, University of Kent at Canterbury, October 2001.

32. T. Tourwé and T. Mens. Identifying refactoring opportunities using logic meta programming. In *7th European Conference on Software Maintenance and Reengineering, Proceedings*, pages 91–100. IEEE Computer Society, 2003. Available at `http://prog.vub.ac.be/~ttourwe/publications.html`.

Appendix: Detailed Example

Consider the following example borrowed from O'Keefe's "The Craft of Prolog" [18], p. 195. It describes three operations on a *reader* data structure used

to sequentially read terms from a file. The three operations are make_reader/3 to initialise the data structure, reader_done/1 to check whether no more terms can be read and reader_next/3 to get the next term and advance the reader.

```
                    ─── O'Keefe's orignal version ───

  make_reader(File,Stream,State) :-
          open(File,read,Stream),
          read(Stream,Term),
          reader_code(Term,Stream,State).

  reader_code(end_of_file,_,end_of_file) :- ! .
  reader_code(Term,Stream,read(Term,Stream,Position)) :-
          stream_position(Stream,Position).

  reader_done(end_of_file).

  reader_next(Term,read(Term,Stream,Pos),State)) :-
          stream_position(Stream,_,Pos),
          read(Stream,Next),
          reader_code(Next,Stream,State).
```

We will now apply several of the previously presented refactorings to the above program to improve its readability.

First of all, we use if-then-else introduction to get rid of the ugly red cut in the reader_code/3 predicate:

```
                    ─── If-then-else introduction ───

  reader_code(Term,Stream,State) :-
          ( Term = end_of_file,
            State = end_of_file ->
                  true
          ;
                  State = read(Term,Stream,Position),
                  stream_position(Stream,Position)
          ).
```

This automatic transformation reveals two malpractices, the first of which is producing output before the commit, something O'Keefe himself disapproves of (p. 97). This is fixed manually to:

```
┌──────────────── Output after commit ────────────────┐
│                                                      │
│  reader_code(Term,Stream,State) :-                   │
│        ( Term = end_of_file ->                       │
│                State = end_of_file                   │
│        ;                                             │
│                State = read(Term,Stream,Position),   │
│                stream_position(Stream,Position)      │
│        ).                                            │
│                                                      │
└──────────────────────────────────────────────────────┘
```

The second malpractice is doing a unification in the condition of the if-then-else where actually an equality test is meant. Consider that the Term argument is a variable. Then the binding is certainly unwanted behaviour. Manual change generates the following code:

```
┌──────────────────── Equality test ────────────────────┐
│                                                        │
│  reader_code(Term,Stream,State) :-                     │
│        ( Term == end_of_file ->                        │
│                State = end_of_file                     │
│        ;                                               │
│                State = read(Term,Stream,Position),     │
│                stream_position(Stream,Position)        │
│        ).                                              │
│                                                        │
└────────────────────────────────────────────────────────┘
```

Next, we notice that the sequence read/2, reader_code/3 occurs twice, either by simple observation or by computing common sequences. By applying predicate extraction of this common sequence, we get:

```
┌──────────────── Predicate extraction ────────────────┐
│                                                       │
│  make_reader(File,Stream,State) :-                    │
│        open(File,read,Stream),                        │
│        read_next_state(Stream,State).                 │
│                                                       │
│  reader_next(Term,read(Term,Stream,Pos),State)) :-    │
│        stream_position(Stream,_,Pos),                 │
│        read_next_state(Stream,State).                 │
│                                                       │
│  read_next_state(Stream,State) :-                     │
```

```
        read(Stream,Term),
        reader_code(Term,Stream,State).
```

Next we apply O'Keefe's own principle of putting the input argument first and output arguments last (p. 14–15):

```
──── Argument reodering ────

reader_next(read(Term,Stream,Pos),Term,State) :-
        stream_position(Stream,_,Pos),
        read_next_code(Stream,State).
```

To finish up, we introduce less confusing and overlapping names for the read/3 functor, the stream_position/[2,3] built-ins and a more consistent naming for make_reader, more in line with the other two predicates in the interface. O'Keefe stresses the importance of consistent naming conventions (p. 213).

Note that direct renaming of built-ins such as stream_position is not possible, but with some cleverness a similar effect can be achieved: simply extract the built-in into a new predicate with the desired name.

```
──── Renaming ────

reader_init(File,Stream,State) :-
        open(File,read,Stream),
        reader_next_state(Stream,State).

reader_next(reader(Term,Stream,Pos),Term,State)) :-
        set_stream_position(Stream,Pos),
        reader_next_state(Stream,State).

reader_done(end_of_file).

reader_next_state(Stream,State) :-
        read(Stream,Term),
        build_reader_state(Term,Stream,State).

build_reader_state(Term,Stream,State) :-
        ( Term == end_of_file ->
                State = end_of_file
        ;
```

21

```
                State = reader(Term,Stream,Position),
                get_stream_position(Stream,Position)
        ).

set_stream_position(Stream,Position) :-
        stream_position(Stream,_,Position).
get_stream_position(Stream,Position) :-
        stream_position(Stream,Position).
```

The above changes could be performed manually, but refactoring browsers such as ViPReSS guarantee you consistency, correctness and furthermore can automatically single out candidates for refactoring.