TLP **8** (2): 201–215, 2008. © 2007 Cambridge University Press
doi:10.1017/S1471068407003134 First published online 23 May 2007 Printed in the United Kingdom

TECHNICAL NOTE
Improving Prolog programs: Refactoring for Prolog

ALEXANDER SEREBRENIK
Laboratory of Quality of Software (LaQuSo), T.U. Eindhoven, HG 5.91,
Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
(*e-mail:* `A.Serebrenik@tue.nl`)

TOM SCHRIJVERS[1]
Department of Computer Science, K.U. Leuven, Celestijnenlaan 200A, B-3001, Heverlee, Belgium
(*e-mail:* `Tom.Schrijvers@cs.kuleuven.be`)

BART DEMOEN
Department of Computer Science, K.U. Leuven, Celestijnenlaan 200A, B-3001, Heverlee, Belgium
(*e-mail:* `Bart.Demoen@cs.kuleuven.be`)

submitted 24 December 2004; revised 28 August 2005, 20 September 2006; accepted 12 February 2007

Abstract

Refactoring is an established technique from the object-oriented (OO) programming community to restructure code: it aims at improving software readability, maintainability, and extensibility. Although refactoring is not tied to the OO-paradigm in particular, its ideas have not been applied to logic programming until now. This paper applies the ideas of refactoring to Prolog programs. A catalogue is presented listing refactorings classified according to scope. Some of the refactorings have been adapted from the OO-paradigm, while others have been specifically designed for Prolog. The discrepancy between intended and operational semantics in Prolog is also addressed by some of the refactorings. In addition, `ViPReSS`, a semi-automatic refactoring browser, is discussed and the experience with applying `ViPReSS` to a large Prolog legacy system is reported. The main conclusion is that refactoring is both a viable technique in Prolog and a rather desirable one.

KEYWORDS: refactoring, software engineering, program transformation, programming environments, tools

1 Introduction

Maintaining and adapting software take up a substantial part of the entire programming effort, in terms of both time and money. Both Erlikh (2000) and Moad (1990) report on the proportion of maintenance costs exceeding 90% of the budget. About 75% of these costs are spent on providing enhancements (in the form of adaptive or perfective maintenance) (Nosek and Palvia 1990; van Vliet 2000).

[1]Research Assistant of the Fund for Scientific Research-Flanders (Belgium) (F.W.O.-Vlaanderen).

Before providing enhancements, it is recommended to improve the design of the software in a preliminary step. This methodology, called *refactoring*, emerged from a number of pioneer results in the OO-community (Opdyke 1992; Fowler *et al.* 1999; Roberts *et al.* 1997) and recently came to prominence for functional (Li *et al.* 2003) and procedural (Garrido and Johnson 2003) languages.

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small source-to-source program transformations, called *refactorings*, that change program structure and organization, 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 can 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 optimization, such as dead-code elimination and redundant argument filtering, can improve program organization and, hence, can be used as refactoring techniques.

In this paper we study refactoring techniques for Prolog. Our goals are threefold. First, we want to show that refactoring is a viable technique for Prolog and many of the existing techniques developed for refactoring, in general, are applicable. Second, Prolog-specific refactorings are possible and the application of some general techniques may be highly specialized toward Prolog. Finally, it should be clear that refactoring is not only viable for Prolog but also very useful for the maintenance of Prolog programs.

To achieve our goals, we present a catalogue of refactoring techniques for Prolog. The listed refactorings are a mix of general and Prolog-specific ones. Most of the refactorings proposed have been implemented in a prototype refactoring browser `ViPReSS`. `ViPReSS` has been successfully applied for refactoring a 50,000 lines-long legacy system.

As completeness of the catalogue is clearly not possible, we aimed to show a wide range of possibilities for future work on combining the formal techniques of program analysis and transformation with software engineering. The formal elaboration of a particular topic may be a substantial study on its own, as shows the work on detecting duplicate code by Vanhoof (2004) that was inspired by a preliminary version of our work.

Outline of the Paper. First, Section 2 provides a brief overview of the refactoring process. Next, the use of several refactoring techniques is illustrated on a small example in Section 3. Then a catalogue of Prolog refactorings is given in Section 4. In Section 5, we introduce `ViPReSS`, and discuss its application in a case study. Finally, in Section 6 we conclude.

2 The refactoring process

The refactoring process consists of applying a number of refactorings, with both localized and global impact, to a software system. The individual significance of a refactoring may be apparent, but often a refactoring seems trivial on its own and only in conjunction with other refactorings or intended changes does the usefulness become clear. That is the reason why it is not feasible to fully automate refactorings. They must be carefully considered in view of the programmer's intentions.

For this reason the process of applying a single refactoring is to be split into a number of distinct activities (Mens and Tourwé 2004). These activities involve decisions to be made by the programmer.

The first decision is *where* the software should be refactored. Making this decision automatically can be a difficult task on its own. Several ways to resolve this may be considered. For instance, one can aim at identifying so-called *bad smells*, i.e., "structures of the code that suggest (sometimes scream for) the possibility of refactoring" (Fowler *et al*. 1999). To this end, program analysis 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. Mode information can be used to detect when this rule is violated.

Next, one should determine *which* refactorings should be applied. Sometimes, the correspondence between bad smells and refactorings is clear. For instance, if the predicate arguments are not ordered according to the "input first output last" rule, one can suggest to the user to reorder the arguments. This refactoring is further discussed in Section 4.3. In more complex situations the relation becomes less obvious: a number of different refactorings are applicable and the user has to choose between them. For example, let module `A` contain a predicate that is mutually recursive with predicate `p` from module `B`, and module `C` contain a predicate that is mutually recursive with predicate `q` from module `B`. This situation can be identified as problematic since no clear hierarchy can be defined between these modules. One possible solution would be to merge the three modules (Section 4.2). Alternatively, one may try to first split `B` into `B1`, containing `p`, and `B2` containing `q` such that there are no circular dependencies between `B1` and `B2` (Section 4.2). If this split is possible, `A` could be merged with `B1`, and `C` with `B2` (Section 4.2). Automatic refactoring tools, so-called *refactoring browsers*, can be expected to make suggestions on where refactoring transformations should be applied. These suggestions can then be either confirmed or rejected by the programmer.

By definition, refactorings should preserve the software's functionality. Hence, the next step consists of *ensuring* that the behavior is indeed preserved. This step, of course, depends on the definition of behavior. In the case of logic programming, behavior comprises computed answers semantics, termination, and side effects such as input–output. It should be observed that particular application domains might require extending the notion of behavior to include such concepts as efficiency or memory use. Moreover, in order for some refactorings to be applicable, certain preconditions should hold like absence of user-defined meta-predicates for dead-code elimination discussed in Section 4.1. Sometimes verification of the preconditions cannot be done automatically, but must be delegated to the user.

Subsequently, *the chosen transformation is applied*. This step might also require user input. Consider, for example, a refactoring that renames a predicate: while automatic tools can hardly be expected to guess the new predicate name, they should be able to detect all program points affected by the change. This refactoring is further studied in Section 4.3.

Finally, the *consistency* between the refactored program code and other related artifacts should be maintained. By artifacts we understand among others software documentation, specifications, and test descriptions. The ability to perform this task automatically strongly depends on the formalisms used to express the corresponding artifacts. For instance, documentation generators such as *lpdoc* (Hermenegildo 2000) make it possible to keep the documentation consistent automatically, whereas ad hoc unstructured comments are much harder to update automatically. Ensuring consistency is considered as future work.

3 Detailed Prolog refactoring example

We illustrate some of the techniques proposed by a detailed refactoring example. Consider the following code fragment from O'Keefe's "The Craft of Prolog" (1994, 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`, which initializes the data structure, `reader_done/1`, which checks whether no more terms can be read, and `reader_next/3`, which gets the next term and advances the reader.

Listing 3.1 - O'Keefe's original 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 refactorings to the above program in order to improve its readability.

First, we use if-then-else introduction (Section 4.4) to get rid of the *red cut* in the `reader_code/3` predicate (modified code is underlined). Recall that a red cut is a cut that alters the meaning (O'Keefe 1994).

Listing 3.2 - Replace cut by if-then-else

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

The result of this automatic transformation reveals two malpractices: the first is producing output before the commit, something O'Keefe himself disapproves of in (1994). This malpractice and the ways to resolve it are further investigated in Section 4.4. The problem is fixed to:

```
Listing 3.3 - 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 a unification in the condition of the if-then-else where an equality test is meant. Consider the case that the `Term` argument is a variable. Then the binding of `Term` to the atom `end_of_file` is certainly unwanted behavior. The transformation in question is discussed in Section 4.4. The following code does not exhibit the problematic behavior:

```
Listing 3.4 - 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 conjunction `read/2, reader_code/3` occurs twice. By applying predicate extraction (Section 4.4) of this common sequence, we get the following:

```
Listing 3.5 - 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 put the input argument first and the output arguments last (Section 4.3 below), a principle also advocated in (O'Keefe 1994):

```
Listing 3.6 - Argument reordering
reader_next(read(Term,Stream,Pos),Term,State) :-
        stream_position(Stream,_,Pos),
        read_next_code(Stream,State).
```

Finally, note that the naming of the two builtins `stream_position/[2,3]` may be confusing to the user. It is easier to distinguish between their functionality based on predicate name than based on arity. We introduce the less confusing names `get_stream_position/2` and `set_stream_position/3`, respectively. In addition, we provide a more consistent naming for `make_reader`, more in line with the other two predicates in the interface. The importance of consistent naming conventions is also stressed in (O'Keefe 1994).

Note that direct renaming of built-ins such as `stream_position` is not possible, but a similar effect can be achieved by extracting the built-in into a new predicate with the desired name. Extracting a predicate and renaming predicates are considered in Sections 4.3 and 4.4, respectively.

To avoid confusion between a built-in predicate `read` and a functor `read`, we rename the latter functor to `reader`.

Listing 3.7 - 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
        ;
                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).
```

This example demonstrates how the code readability can be ameliorated by performing a series of relatively simple transformation steps. We have seen that some of these steps required user's input. Clearly the changes can be performed manually. However, refactoring browsers should be able to guarantee consistency and correctness and furthermore can automatically single out opportunities for refactoring.

Techniques applied above are well-suited for local code improvement; i.e., the objects modified are predicates and clauses. In the next section we also consider techniques for global code restructuring such as duplicate predicates removal (Section 4.1).

4 A catalogue of Prolog refactorings

In this section we present the refactorings that we have found to be useful for Prolog programs. The considered Prolog programs are not limited to pure logic programs, but may contain various built-ins such as those defined in the ISO standard (1995). The only exception are higher-order constructs that are not dealt with automatically, but manually. This is done because higher order constructs such as *call* make it impossible to decide at the compile-time which predicate is going to be called at the corresponding program point during execution. Automating the detection and handling of higher-order predicates is an important part of future work.

The refactorings in this catalogue are grouped by their scope. The scope expresses the user-selected target of a particular refactoring. Hence, refactoring starts by choosing an object in the specified scope. For instance, *split module* (Section 4.2) starts with selecting a module. Then the object is transformed. For us, this means that the module is split. Finally, the changes propagate to the affected code outside the selected scope. The latter might happen when there is a dependency outside the scope. This corresponds to updating import declarations in other modules of the system.

For Prolog programs we distinguish the following four scopes, based on the code units of Prolog: *system* scope (Section 4.1), *module* scope (Section 4.2), *predicate* scope (Section 4.3), and *clause* scope (Section 4.4).

As a starting point for this catalogue, we used Fowler's (2003) for object-oriented languages. We selected those with clear Prolog counterparts and extended the list with Prolog-specific transformations and some well-known program transformations, such as dead code elimination.

In the current technical note we include only a short summary of the refactorings here and refer to the companion technical report (Schrijvers *et al.* 2003). This report contains the full catalogue with detailed description of the refactorings, examples, preconditions, and automatization techniques.

4.1 System scope refactorings

The system scope encompasses the entire code base. The user wants to consider the system as a whole.

4.1.1 Eliminate explicit module qualification

In many Prolog systems, such as Quintus (Intelligent Systems Laboratory 2003a), the module system is nonstrict; i.e. the normal visibility rules can be overridden by a special construct, called *explicit module qualification* and written as `m:q`, where `m` is a module that contains definition of the predicate `q/0`. The refactoring proposed adds import and export declarations to get rid of these special syntax constructions. By forcing the code to conform to a strict module system, a number of quality characteristics are improved. First of all, a strict module system better expresses the idea of information hiding, which is important for software maintainability and readability (Parnas 1972). Moreover, since not all Prolog systems support the above construct, code portability is improved.

4.1.2 Extract common code into predicates

This refactoring looks for common functionality across the system and extracts it into new predicates. The common functionality consists of identical subsequences of goals that are called in different predicate bodies, and extracts them into new predicates. The overall readability of the program improves as the affected predicate bodies get shorter, and the calls to the new predicates can be more meaningful than what they replace. Moreover, the increased sharing simplifies maintenance as now only one copy needs to be modified.

The problem of identifying identical subsequences of goals is related to determining longest repeated subsequences (Crow and Smith 1992; Pitkow and Pirolli 1999).

4.1.3 Hide predicates

This refactoring removes export declarations for predicates that are not imported in any other module. It simplifies the program by reducing the number of entry points into modules and hence the intermodule dependencies.

4.1.4 Remove dead code

Dead code is code that can never be executed and therefore can be safely eliminated without affecting correctness of the execution. Dead code elimination is sometimes performed in compilers for efficiency reasons, but it is also useful for developers: dead code clutters the program. We consider a predicate definition as the unit of dead code.

4.1.5 Remove duplicate predicates

Predicate duplication or cloning is a well-known problem, prominently caused by “copy & paste” and unawareness of available libraries and exported predicates in other modules. The main problem with duplication is its bad maintainability. It is up to the user to decide whether to throw away some of the duplicates and to use one of the remaining definitions instead or to replace all the duplicate predicates by a new version in a new module.

4.1.6 Rename functor

This refactoring renames a term functor across the system. If the functor has several different meanings and only one should be renamed, it is up to the user to identify what occurrence corresponds with what meaning.

4.2 Module scope refactorings

The module scope considers a particular module. Usually a module is implementing a well-defined functionality and is typically contained in one file.

4.2.1 Merge modules

Merging several modules into one can be advantageous in case of strong interdependency of the modules involved. Moreover, merging existing modules and splitting the resulting module can lead to an improved module structure.

4.2.2 Remove dead intramodule code

Similar to *dead code removal* for an entire system (see Section 4.1), this refactoring works at the level of a single module. It is useful for incomplete systems or library modules with an unknown number of uses. Recall that determining the liveness of the code requires knowledge of top-level predicates. In the case of intramodule dead code elimination, the set of top-level predicates is extended with, or replaced by, the exported predicates of the module.

4.2.3 Rename module

This refactoring applies when the name of the module no longer corresponds to the functionality it implements, e.g., due to other refactorings.

4.2.4 Split module

The refactoring is useful to split unrelated parts of a module or make a large module more manageable.

(Moores 1998) has shown that the number of user-defined predicates correlates with the number of errors detected. On the basis of an empirical study, he suggested a threshold of around 35 ± 5 predicates per program. While this is hardly reasonable as a requirement for an entire Prolog system, trespassing the threshold should be used as a guideline when the Split Module refactoring can be applied.

4.3 Predicate scope refactorings

The predicate scope targets a single predicate. 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 automatically.

4.3.1 Add argument

This refactoring should be applied when a callee needs more information from its (direct or indirect) caller, which is very common in Prolog program development. Given a variable in the body of the caller and the name of the callee, the refactoring browser should 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.

4.3.2 Move predicate

This refactoring moves a predicate definition from one module to another. It can improve the overall structure of the program by bringing together interdependent or related predicates, hence improving both cohesion of each one of the modules involved, and coupling of the pair. *Move predicate* appears often after predicate extraction, i.e., *extract common code* or *extract predicate locally*, discussed in Sections 4.1 and 4.4, respectively.

4.3.3 Rename predicate

This refactoring can improve readability and should be applied when the name of a predicate does not reveal its purpose.

4.3.4 Reorder arguments

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 habit has been identified as a good practice and even further refined by O'Keefe (1994) to more elaborate rules. Unfortunately, this practice is difficult to maintain 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.

4.3.5 Specialize predicate

By specializing a predicate, we mean producing a (number of) more specific version(s) of a given predicate provided some knowledge on the intended uses of the predicate. Specialization can simplify code as well as make a meaningful distinction between different uses of a predicate.

4.3.6 Remove redundant arguments

The basic intuition here is that parameters that are no longer used by a predicate should be dropped. It improves readability.

Leuschel and Sørensen (1996) established that the redundancy property is undecidable and suggested two techniques to find safe and effective approximations: top-down goal-oriented RAF (Redundant Argument Filtering) and bottom-up goal-independent FAR (RAF "upside-down"). In the context of refactoring, FAR is the more useful technique, since only FAR deals correctly with exported predicates used in unknown goals.

4.4 Clause scope refactorings

The clause scope affects a single clause in a predicate. Usually, this does not affect any code outside the clause directly.

4.4.1 Extract predicate locally

This refactoring is similar to the system-scope refactoring with the same name. However, it does not aim to automatically discover useful candidates for replacement. The user is responsible for selecting the subgoal that should be extracted, to improve the readability.

4.4.2 Invert if-then-else

The order of "then" and "else" branches can be important for code readability. To enhance readability, it might be worthwhile putting the shorter branch as "then" and the longer one as "else." Alternatively, the negation of the condition may be more readable because, for example, a double negation can be eliminated.

4.4.3 *Replace cut by if-then-else*

This technique aims at improving program readability by replacing cuts (!) by the more declarative if-then-else (`->  ;  `). More detailed discussion on replacing cut by if-then-else is deferred to *Related work and extensions*.

4.4.4 *Replace unification by (in)equality test*

Often full unifications are used instead of equality or other tests. O'Keefe (1994) advocates the importance of steadfast code. Recall that steadfast code produces the right answers for all possible modes and inputs. A more moderate approach is to write code that works for the intended mode only. Unification succeeds in several modes and so does not convey a particular intended mode. Equality (`==`, `=:=`) and inequality (`\==`, `=\=`) checks usually succeed only for one particular mode and fail or raise an error for other modes. Hence their presence makes it easier in the code and at run-time to see the intended mode. Moreover, if only a comparison was intended, then full unification may lead to unwanted behavior in unforeseen cases.

4.4.5 *Produce output after commit*

This refactoring addresses a similar issue as the previous one. Producing output before the commit (cut) does not properly convey the intended mode of a predicate. Moreover, it may lead to unexpected results when used in the wrong mode.

5 The `ViPReSS` refactoring browser

The refactoring techniques presented in Section 4 have been implemented in the prototype refactoring browser `ViPReSS`.[2] It has been implemented on the basis of VIM, a popular clone of the well-known VI editor. The text editing facilities of VIM make it easy to implement techniques like *move predicate* (Section 4.3).

Most of the refactoring tasks have been implemented as SICStus Prolog (Intelligent Systems Laboratory 2003b) programs inspecting source files and/or call graphs. Updates to files have been implemented either directly in the scripting language of VIM or, when many files need updating at once, through `ed` scripts. VIM functions were written to initiate the refactorings and to get user input.

`ViPReSS` has been successfully applied to a large (more than 53 KLOC) legacy system used at the computer science department of the Katholieke Universiteit Leuven to manage the educational activities. The system, called BTW, has been developed and extended since the early eighties by more than 10 programmers, many of whom are no longer employed by the department. The implementation has been done in MasterProLog (IT Masters 2000), which is no longer supported. Therefore, preparing the code for migration to a more modern Prolog dialect and general structure improvement were essential for further evolution of the system.

[2] Vi(m) P(rolog) Re(factoring) (by) S(chrijvers) (and) S(erebrenik).

By using the refactoring techniques, we succeeded in obtaining a better understanding of this real-world system, in improving its structure and maintainability, and in preparing it for intended changes: 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 the unified Bachelor–Master system in Europe.

A preliminary study revealed that many modules were unused. We brought in an expert to help us identify the bulk of these unused modules, including out-of-fashion user interfaces and outdated versions of program files. This reduced the system size to a mere 20,000 lines.

Next, the actual refactoring process was started. As the first phase we applied system-scope refactorings. `ViPReSS` was used to clean up after the bulk dead code removal: 299 predicates in the remaining modules were identified as dead. This reduced 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 files from 294 to 116 with a total of 18,000 lines. Very little time was spent to bring the system into this state. The experts were sufficiently familiar with the system to identify obsolete parts. The system-scope refactorings took only a few minutes each. During this phase most of the work has been done by `ViPReSS`, while the user's involvement was limited to choosing a way to deal with duplicate predicates.

The second phase of refactoring consisted of a thorough code inspection aimed at local improvement. Many malpractices were identified: excessive use of cut (Section 4.4) combined with output construction before commit (Section 4.4) being the most notable one. Additional "bad smells" discovered include bad predicate names such as `q`, unused arguments and unifications instead of identity checks or numerical equalities (Sections 4.3, and 4.4, respectively). Some of these were located by `ViPReSS`, others were recognised by the users, while `ViPReSS` performed 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.

In summary, from 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 better-structured lumber-free code. Now it not only is more readable and understandable but also simplifies implementing the intended changes. From our experience with refactoring this large legacy system and the relative time investments of the global and the local refactorings, we recommend starting out with the global ones and then selectively apply local refactorings as the need occurs.

The current version of `ViPReSS` can be downloaded from `http://www.cs.kuleuven.ac.be/~toms/vipress`.

6 Conclusions

In this paper we have studied refactoring techniques for Prolog. First, we have shown that refactoring is a viable technique for Prolog and that many of the existing techniques developed for refactoring in general are applicable. Our refactoring catalogue contains many such refactorings.

Second, Prolog-specific refactorings are possible and the application of some general techniques may be highly specialized toward Prolog. In this context, the companion technical report (Schrijvers *et al*. 2003) shows how refactoring fits in with existing work on program analysis and transformation in the context of Prolog and how many of these existing techniques may be adapted for the purpose of partially automating the refactoring process. Also, `ViPReSS`, our refactoring browser, integrates several automatable parts of the presented refactorings in the VIM editor.

Finally, it should be clear that refactoring Prolog programs is not just viable but very useful for the maintenance of Prolog programs. Refactoring helps bridge the gap between prototypes and real-world applications. Indeed, extending a prototype to provide additional functionality often leads to cumbersome code. Refactoring allows software developers both to clean up code after changes and to prepare code for future changes. These are important benefits that also apply to logic programming.

As completeness of the catalogue is clearly not possible, we aimed to show a wide range of possibilities for future work on combining the formal techniques of program analysis and transformation with software engineering. Throughout the catalogue, many specific issues for future work have been mentioned. Below we list related work and more general challenges for the future.

6.1 Related and future work

Logic programming has often been used to implement refactorings for other languages; e.g., a meta-logic very similar to Prolog is used to detect, for instance, obsolete parameters in Tourwé and Mens (2003).

Seipel *et al*. (2003) include refactoring among the analysis and visualization techniques that can be easily implemented by means of FnQuery, a Prolog-inspired query language for XML. However, the discussion stays at the level of an example. The M.Sc. thesis of Steinke (2003) was dedicated to refactoring of logic programs. A catalogue of refactorings has been composed and a prototype system has been implemented. However, only predicate-scope refactorings have been considered and only the transformation step has been implemented.

In the logic programming community questions related to refactoring have been intensively studied in the context of program transformation and specialization. There are two important differences with this line of work. First, refactoring improves readability, maintainability, and extensibility rather than performance. Second, for refactoring user input is essential while in the mentioned literature strictly automatic approaches were considered. However, some of the transformations developed for program optimization, e.g., *dead code elimination*, can be considered as refactorings and have an important function in refactoring browsers.

To further increase the level of automation of particular refactorings, additional information such as types and modes can be used.

Future refactoring tools can also benefit from integration with Prolog development environments. Modern Prolog systems are often equipped with features extending the ISO

Standard such as constraint solving over different domains and Constraint Handling Rules, coroutining, interfaces to foreign languages, GUI-development systems and databases. In most of the cases, the refactoring techniques described above can still be applied to improve the code. Certain refactorings may be specially designed for particular extensions. For instance, our experience suggests that simplifying primitive constraints may be useful in the case of CLP.

References

1995. *Information Technology—Programming Languages—Prolog—Part 1: General Core*. ISO/IEC. ISO/IEC 13211-1:1995.

Crow, D. and Smith, B. 1992. DB_HABITS: Comparing minimal knowledge and knowledge-based approaches to pattern recognition in the domain of user–computer interactions. In *Neural Networks and Pattern Recognition in Human–Computer Interaction*, R. Beale and J. Finley, Eds. Ellis Horwood, 39–63.

Erlikh, L. 2000. Leveraging legacy system dollars for e-business. *IT Professional 2,* 3 (May), 17–23.

Fowler, M. 2003. Refactorings in alphabetical order. URL: `http://www.refactoring.com/catalog/`. Accessed April 2, 2007.

Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. 1999. *Refactoring: Improving the design of existing code*. Object Technology Series. Addison-Wesley.

Garrido, A. and Johnson, R. 2003. Refactoring C with conditional compilation. In *18th IEEE International Conference on Automated Software Engineering*, H. Kirchner and C. Ringeissen, Eds. IEEE, 323–326.

Hermenegildo, M. V. 2000. A documentation generator for (C)LP systems. In *Computational Logic - CL 2000, First International Conference, London, UK, July 2000, Proceedings*, J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Moniz Pereira, Y. Sagiv, and P. J. Stuckey, Eds. Lecture Notes in Artificial Intelligence, vol. 1861. Springer Verlag, 1255–1269.

Intelligent Systems Laboratory. 2003a. *Quintus Prolog User's Manual*. P.O. Box 1263, SE-164 29 Kista, Sweden.

Intelligent Systems Laboratory. 2003b. *SICStus Prolog User's Manual*. P.O. Box 1263, SE-164 29 Kista, Sweden.

IT Masters. 2000. *MasterProLog Programming Environment*. URL: `www.itmasters.com`. Accessed September 19, 2006.

Leuschel, M. and Sørensen, M. H. 1996. Redundant argument filtering of logic programs. In *Proceedings of the 6th International Workshop on Logic Program Synthesis and Transformation*, J. Gallagher, Ed. LNCS, vol. 1207. Springer Verlag, 83–103.

Li, H., Reinke, C. and Thompson, S. 2003. Tool support for refactoring functional programs. In *Haskell Workshop 2003*, J. Jeuring, Ed. Association for Computing Machinery.

Mens, T. and Tourwé, T. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering 30,* 2 (February), 126–138.

Moad, J. 1990. Maintaining the competitive edge. *Datamation 36,* 4 (February), 61–66.

Moores, T. T. 1998. Applying complexity measures to rule-based Prolog programs. *The Journal of Systems and Software 44*, 45–52.

Nosek, J. T. and Palvia, P. C. 1990. Software maintenance management: Changes in the last decade. *Journal of Software Maintenance: Research and Practice 2,* 3 (September), 157–174.

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

OPDYKE, W. F. 1992. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois at Urbana–Champaign.

PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15,* 12 (December), 1053–1058.

PITKOW, J. AND PIROLLI, P. 1999. Mining longest repeating subsequences to predict World Wide Web surfing. In *2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, 1–12.

ROBERTS, D., BRANT, J. AND JOHNSON, R. 1997. A refactoring tool for Smalltalk. *Theory and Practice of ObjectSystems (TAPOS) 3*(4), 253–263.

SCHRIJVERS, T., SEREBRENIK, A. AND DEMOEN, B. 2003. *Refactoring Prolog Programs*. Tech. Rep. CW 373, Department of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium.

SEIPEL, D., HOPFNER, M. AND HEUMESSER, B. 2003. Analysing and visualizing Prolog programs based on XML representations. In *Proceedings of the 13th International Workshop on Logic Programming Environments*, F. Mesnard and A. Serebrenik, Eds. 31–45. Published as technical report CW371 of Katholieke Universiteit Leuven.

STEINKE, D. 2003. Refactoring von Logischen Programmen. M.S. thesis, Universität Rostock. URL: `http://e-lib.informatik.uni-rostock. de/fulltext/2003/diploma/SteinkeDirk-2003.ps.gz`. Accessed September 20, 2006.

TOURWÉ, T. AND MENS, T. 2003. Identifying refactoring opportunities using logic meta programming. In *7th European Conference on Software Maintenance and Reengineering, Proceedings*. IEEE Computer Society, 91–100.

VAN VLIET, H. 2000. *Software Engineering: Principles and Practice, 2nd ed*. John Wiley & Sons.

VANHOOF, W. 2004. Searching semantically equivalent code fragments in logic programs. In *Logic-based Program Synthesis and Transformation. 14th International Workshop, LOPSTR 2004, Verona, Italy, August 26–28, 2004, Pre-Proceedings*, S. Etalle, Ed. 1–18.