Adapting code metrics to Prolog

Sławomir Rudnicki

Abstract

Little effort has been put into developing methods for measuring quantitative values of complexity and maintainability of logic-based programs. This work presents the approach towards adapting known software metrics to Prolog code, as well as *Metrics for Prolog*, a simple Prolog metrics tool.

1 Introduction

In the software industry, there is a need to control the complexity and maintainability of the produced code. The most well-known approaches to measuring this maintainability, which provide quantitative results, were introduced in the late 1970s by Thomas McCabe Sr.[4], who based his metric on computational features of programs, and Maurice Halstead[2], who in turn proposed metrics that relied on psychological aspects of understanding software code.

There has not been much effort put into measuring the complexity of code when it comes to declarative and logic-based programming. I have developed a simple tool, called *Metrics for Prolog* or *MFPL*, that provides some simple metrics for code written in Prolog, a logic-based programming language.

This document presents a discussion of whether and how some software metrics may be adapted to modern Prolog code.

2 The Prolog language

Prolog is a declarative programming language, whose syntax is based on first-order logic. Since its creation around the year 1972, it has found multiple applications in the fields of artificial intelligence and natural language processing.

Prolog programs consist of definitions of *relations* on *terms*. Intuitively, a run of the program seeks to determine whether a relation holds for the given arguments, or to find a set of arguments such that a given relation holds. In this document, I will briefly describe the basics of logic programming to the extent that is needed to understand the problems of adapting code metrics to it. A thorough introduction to the core syntax and semantics of Prolog itself can be found in the classic book by Clocksin and Mellish[1].

3 Local complexity measure

An early approach towards measuring code metrics of Prolog has been presented in a 1985 work by Markusz and Kaposi[3]. They proposed a metric designed specifically for Prolog, which is simple to compute and gives a view on the complexity of given clauses (or *partitions*). The metric, which will here be called the *local complexity measure*, or LC, allows for measuring both the complexity level and the size of the whole file or program.

In *MFPL*, I have implemented a modified version of what Markusz and Kaposi describe as their first attempt to developing their complexity measure. It is defined as:

$$C = P_1 + P_2 + P_3 + P_4,$$

where:

- P_1 is the number of new data entities in the left hand side of the partition. I have interpreted this as the number of new non-trivial variables introduced in the positive atom.

- P_2 is the number of subproblems the clause divides the problem into. This is interpreted as the number of atoms in the right hand side of the clause.

- P_3 measures the complexity of relations between the subproblems. Markusz and Kaposi suggested that 2 should be added here for every recursive call of the defined predicate. Because the set of Prolog operators that change the relationships between the subproblems has changed since their work, I have decided to modify this approach by also adding 1 for each disjunction (introduced by the operator `;`) and implication (introduced by operators `->` and `*->`).

- P_4 is the number of variables introduced in the right hand side of the clause.

However in the original work the measure was only applied to clauses, in *MFPL* it is also used to measure facts and commands.

4 Halstead's metrics

Halstead[2] proposed metrics for analysing the effort one needs to make to code a program and then maintain it. The metrics are largely based on psychological research of how one understands a program by making "mental comparisons" of tokens.

Halstead's metrics are defined using two groups of source code tokens: *operators* and *operands*. Let us denote:

- N_1 – the number of all operators,

- N_2 – the number of all operands,

- n_1 – the number of distinct operators,

- n_2 – the number of distinct operands.

The following metrics are then derived from those counts:

Program length

$$N = N_1 + N_2,$$

denotes the total length of the program, measured as the number of all tokens.

Vocabulary

$$n = n_1 + n_2,$$

denotes the size of the program's vocabulary, i.e. the number of distinct tokens that one has to consider to fully understand the program.

Volume

$$V = N \cdot \log_2 n,$$

which measures the size of the program when coded in binary form. The metric is based on the fact that every token of the analysed program can be coded with $log_2 n$ bits.

Difficulty

$$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2},$$

measures how difficult the program is to create and maintain. According to Halstead, code is more sophisticated and error-prone if distinct operands occur many times in it..

Level

$$L = \frac{1}{D}.$$

The inverse of difficulty, this metric may indicate whether a given program is written on a high- or low-level. For example, assembler code will tend to have a lower level value than Java code.

Effort

$$E = V \cdot D.$$

The effort needed to write or understand a program is proportional to its size and difficulty.

Halstead moves on to propose experimentally derived, higher-level metrics of code, including:

Time

$$T = E/18,$$

which is the time needed to "mentally" implement the program, in seconds. The constant 18 was set by experiment for procedural languages such as Java; for Prolog, it has to be changed to a lower value since the language syntax is simpler. In *MFPL*, the constant has been set to 12.

Bugs

$$B = E^{\frac{2}{3}}/3000,$$

which is the number of bugs that are delivered, on average, inside the program.

4.1 Operators and operands

The main problem with Halstead's metrics, however, is the definition of what is an operator and what is an operand. For popular languages, those notions have been thoroughly discussed. Nandy[5] sums up what seems to be the most common understanding of what operators and operands are in C, C++ and Java.

In declarative programming, it is not obvious how to discern operators from operands. All language constructs of Prolog are in fact terms of the form:

$$T(x_1, x_2, \dots, x_k).$$

Here, the number k is the *arity* of the predicate symbol T, and $x_1, \dots, x_k$ are its *arguments*. We usually write T/k to denote a term and its arity.

When we thus write:

$$A\ is\ B + 1,$$

Prolog understands it as nothing but:

$$is(A, +(B, 1)).$$

When one takes into account the fact that function names in Java function calls are commonly considered to be operands, it is therefore tempting to view all subterms of a term as operands.

To add to this, Prolog is often used to write meta-programs – programs that operate on Prolog source code. Some programs may even manipulate parts of themselves during normal operation, further blurring the notions of operators and operands. Of course, *MFPL* itself is an example of a meta-program.

In *MFPL*, I have chosen a simple way of dividing tokens into operands and operators. In a few words, it can be described as *If it looks like an operator, it is an operator*. Operators are therefore all symbols that may be used without the parenthetical notation $T(x, y)$ and may be instead written as:

- $T\,x\,y$ (prefix notation),
- $x\,T\,y$ (infix notation), or
- $x\,y\,T$ (postfix or reverse Polish notation).

All other relational symbols will be viewed as operands, even if they are never an argument for another symbol.

The above decision is one that seems to be most in line with the psychological basis of Halstead's metrics. When reading a Prolog program, or any other code, we naturally consider tokens that take advantage of the infix notation as operators, because this is what we are used to from studying mathematics. Hovewer informal, this observation would certainly be more important from Halstead's point of view than the actual internal representation of terms.

We will therefore consider all built-in Prolog operators to be operators in terms of Halstead's metrics. The built-in operators are, for example:

- the rule-building predicate :-$/2$,
- the conjunction and disjunction operators $,/2$ and $;/2$,
- arithmetic operators, including $is/2$, $+/2$, $*/2$,
- arithmetic relations, such as $=</2$,
- bitwise and logical operators, including the logical alternative operator $/2$ and the bitwise shift operator $<</2$.

ISO-Prolog contains a mechanism that allows the user to define their own operators using the built-in predicate $op/3$. By writing:

```
op(400, fx, '***'),
```

we can allow the symbol `***/1` to be used in prefix notation. *MFPL*'s Halstead metrics will treat all such tokens as operators.

There are also a few purely syntactical tokens that will also be treated as operators:

- the pair of parentheses () surrounding a symbol's arguments will be viewed as two distinct operators.
- every comma between arguments of a term will be viewed as an operator.
- the dot that stands after every Prolog term will be viewed as a single operator, by analogy to C's semi-colon,
- the list notation `[H | T]` will be treated as one operator.

4.2 Variable distinctiveness

Halstead's metrics heavily rely on which operands and operators we consider to be *the same*. Whereas this is no problem with operators, atoms and integers, it becomes problematic when it comes to variables. In *MFPL*, I decided to make the following assumptions:

- Variables occurring in different clauses of the same predicate are considered distinct, even if they have the same name. This is because the variables do not share a common scope (as we would call it if Prolog were a procedural language).
- Singleton variables (i.e. variables that occur only once in a clause) are all considered distinct, even though they are often denoted by the same name "_". This assumption is again derived from the psychological basis of Halstead's metrics: even though the programmer does not have to "mentally compare" the singleton variable to other variables, he has to take note of its position as an argument of a term and only then can they consider it irrelevant.

5 Other metrics

Logic-based programming strongly differs from imperative programming in that the programmer states a problem and lets the built-in inference mechanism find the solution instead of providing means to solve the problem. Thus, not every software metric can be sensibly adapted to Prolog. For example, the cyclomatic complexity (CC) metric introduced by Thomas J. McCabe[4], however general, can only apply to procedural languages. Without going into the details, CC is equal to the minimum number of execution paths that span the whole execution tree, in which branching is introduced by checking a logical condition.

Due to the declarative nature of Prolog, not every language construct can be interpretted as a part of the execution tree in a straightforward way. As a simple example, let's take the following one-clause predicate:

```
succ(A, B) :- B is A+1.
```

When we start to think in procedural terms, as suggested by McCabe's metric, calling this predicate may have two different outcomes. On one hand, it may act as a simple assignment statement, equivalent to `B := A+1`. On the other hand, it may be a logical check, equivalent to `B == A+1` and would as such introduce a branch in the execution tree.

Such problems multiply when more complex language constructs are considered. Therefore, I did not attempt to include McCabe's metric, or other "procedural" metrics, in *MFPL*.

References

[1] W.F. Clocksin, C.S. Mellish, *Programming in Prolog*, Springer-Verlag Telos, 1994.

[2] M.H. Halstead, *Elements of Software Science*, Elsevier, North-Holland, New York, 1977.

[3] Z. Markusz and A.A. Kaposi *Complexity Control in Logic-based Programming*, The Computer Journal, 28: pp. 487-495, 1985.

[4] T.J. McCabe, *A Complexity Measure*, IEEE Transactions on Software Engineering: pp. 308–320, 1976.

[5] I. Nandy, *Halstead's Operators and Operands* Scribd.com, 2007.