Reusability and portability of logic programming

GIANCARLO SUCCI$^1$, CARL UHRlK$^2$, and MARCO RONCHETTI$^1$

$^1$Laboratorio di Ingegneria Informatica Dipartimento di Informatica e Studi Aziendali, Università di Trento via F. Zeni 8, I-38068 Rovereto, Italia
$^2$Faculty of the Dept. of Technology Programs University of Phoenix, 7800 E. Dorado Pl, Englewood, CO

Received 14th November 1994

This paper investigates the relationships among logic languages, software reusability and software portability, with the perspective that if these three, almost orthogonal topics become synergistic, they will have an extremely positive impact on the software development process. More specifically, the discussion aims to demonstrate both that logic programs are more reusable and portable, even in extreme situations, and that, however, some problems still limit the effective application of such reusable and portable logic paradigm. After a precise definition of the terms, this paper analyses the potential for reusability and of portability of programs written in logic languages, providing examples supporting the claims and giving evidences also of the opportunities available in the field of parallel programming. The limits of this approach are then outlined and discussed. Eventually the potentials for overcoming such limits are evidenced.

Keywords: software reuse, logic programming, abstract machines

1. Introduction

Logic programming is a programming paradigm, software reusability is a methodology of software production, and software portability is a property of software. The first two are not means but instruments, while the last is a desired property, but not the ultimate goal. The ultimate goal is to produce software efficiently and effectively, and this objective must always to be taken into consideration as the implicit reference point of the discussion. Theoretical studies as well as empirical evidence demonstrate that logic programming can impact software production positively, that a proper reuse policy can reduce the costs and the risks in the process of software development and that software portability is one of the key factors that can make a software product a success especially dealing with heterogeneous and open environments.

A long debate about the pros and cons of logic programming has been taking place for several years. This paper is a contribution to this discussion written by people quite familiar with both logic programming and software reusability.

This paper deals with the problem of how logic languages can affect positively software reusability and portability in the sense that using a logic language both facilitates implementing

0963-9306 © 1996 Chapman & Hall
a suitable reuse policy and makes the portability of the resulting code much higher, so that software production can be increased.

2. Defining the terms

We first define the exact terms of the discourse, i.e. what is meant here by logic programming, software reusability and software portability.

2.1 Logic programming

A few years ago R. Kowalski [1] coined the equation:

\[ \text{Algorithm} = \text{Logic} + \text{Control} \]

meaning that each program has always two components, the Logic, namely the rationale standing behind the program and the Control, i.e. the sequence of operations to be performed to accomplish the goal of the program itself.

Imperative languages, such as Pascal, C, C++, ..., keep the control explicit, leaving implicit the logic or, in the best case, confining it to comments, sometimes formally expressed as invariant assertions and more often intuitively explained in natural languages. Declarative languages, such as Prolog [2], LISP\(^1\) [3], Miranda [4], SEL [5], and others, describe the logic explicitly, leaving implicit the control.

Declarative languages are divided into three major classes:

- **logic languages** in a strict sense which transpose mathematical logic into a programming language; the most widely known example of this class is Prolog, based on Horn clauses [6];
- **functional languages**, which are based on the concept of functional abstraction; the archetype of all functional languages is the \(\lambda\)-calculus [7], while their most famous example is LISP [3];
- **equational languages**, which are founded on the concept of term rewriting [8].

A great advantage of logic languages is strictly connected to their essence: no control flow ordering is defined on the computation; this implies that:

(a) the instructions (we should say 'the logic statements') never have side effects, therefore it is easier for programmers to know what they are doing;
(b) the assignment instruction is incompatible with this paradigm of programming because if the same variable were assigned twice then its value would depend on the last assignment, i.e. on the control flow; instead of assignments, there are name definitions used as short-cuts for long equations; this property is called referential transparency, since any reference to any name does not depend on where it is placed [9].

\(^1\) We are dealing with the pure subset of Prolog and LISP, therefore we are not taking into account meta logical and extra logical features.
An example can help understanding these points. Consider the function that computes the solution for a quadratic equation (for clarity we assume that the coefficient of the second order term is different from zero, i.e. that the equation is not reducible to linear). The Miranda [4] definition of such a function is the following:

\[
\text{sol}(a,b,c) = (x1,x2), \ delta >= 0 \\
\text{where} \\
x1 = (-b -d)/(2*a) \\
x2 = (-b +d)/(2*a) \\
d = \sqrt{\text{delta}} \\
delta = b*b - 4*a*c
\]

\[
\text{sol}(a,b,c) = ((r1,c1),(r2,c2)), \ delta < 0 \\
\text{where} \\
r1 = -b/(2*a) \\
c1 = -d/(2*a) \\
r2 = -b/(2*a) \\
c2 = +d/(2*a) \\
d = \sqrt{-\text{delta}} \\
delta = b*b - 4*a*c
\]

This definition is self-explanatory; there are two statements, one for the case of a real solution and the other for the case of a complex solution. Each solution is represented as a couple, and the complex numbers are also couples of a real part and an imaginary part. The scope of each name definition is defined by the so-called offset rule: each definition has significance defined in the context of statements preceding a previous where statement (the same as in math definitions) and where statements can be nested; all the definitions upon which the where construct depends must be indented accordingly, i.e. to the right of it.

Now compare the above declarative definition with the following Pascal-like imperative one.

```pascal
type RealSolutionType = record
  x1, x2 : real;
end;

function sol(real a, b, c) : RealSolutionType;
var
  d, delta : real;
begin
  sol.exists := False;
  delta := b*b - 4*a*c;
  if delta < 0 then return;
  d := sqrt(delta);
  sol.x1 := (-b -d)/(2*a);
  sol.x2 := (-b +d)/(2*a);
  sol.exists := True
end sol;
```

Here the case of a complex solution is omitted. It would have amounted to defining a complex number type and a variant record for the two possible kinds of solutions. Despite this omission, the above imperative solution is less clear than the previous logic programming example (at least for those people who are not too well indoctrinated into the imperative paradigm), and the
following observations present some of the problems:

- the type of the solution must be clearly stated and defined somewhere, not always close to the function definition, as here;
- the ordering of the instructions is strictly relevant: if the ordering was changed, the algorithm would not work;
- here we are lucky that there are no global variables so no risks of side effects occurs, but this is hardly ever the case.

Other advantages of logic languages over imperative ones are:

- Implicit parallelism. Since no explicit control is imposed on the program, parallelism can easily be automatically exploited by the compiler (this form of parallelism is termed transparent parallelism).
- Abstract analysis. Explicitly specifying only the logic of the program has the consequence that various forms of abstract manipulation are intrinsically simpler, e.g. mode analysis, grainsize analysis, strictness analysis).
- Program verification. The absence of explicit control and assignment statements make declarative programs closer to their specification, thus simplifying the task of program verification.
- Fast prototyping. As the user has only to specify the logic underlying a program, declarative programs tend to be much shorter than their imperative counterparts, and hence declarative languages are good tools for rapid prototyping.

Their major drawback, however, is their inefficiency. Nevertheless current research demonstrates that it is possible to have declarative languages achieving almost the same efficiency as imperative ones, as is well proven, for instance, by Van Roy, who started his experiments back in the mid 1980s [10].

### 2.2 Software reusability

Software reusability has been emerging as one of the most important research areas of software engineering. The message from the marketplace is clear: software development costs must be reduced while software quality and sophistication are increased. Increasingly, the market demands software-intensive systems which are more robust and simpler to develop. The market is also evolving and time-to-market often differentiates success from failure. Thus far, it has only been possible to meet market demands for complex systems by expensive handcrafting. Software reusability may be the philosophical stone for the software crisis.

Several competing definitions are given for software reusability. Caustically Tracz says that ‘Software Reuse is the reuse of software designed to be reused’ [11]; more globally Basili suggests that ‘Software Reuse is the reuse of everything associated with a software project including knowledge’ [12]. The difference between the two approaches is quite clear and evidences two approaches to reusability:

- the first one produces code for later reuse,
- the second one deals with the problem of reusing legacy code.
The two problems are quite different, even though the first may seem a subset of the second; sometimes the first problem is called 'design for reuse' and the second 'design with reuse'; in any case, a common factor between the two is the need to understand the code. Another common issue is that in order to understand and reuse a segment of code it is necessary to determine its preconditions and its postconditions. The most desirable situation is when such conditions can be represented in terms of two sets of variable values, input and output, without any other rippling effect [13], i.e. when we can give a pure function model of the code [14].

Summing up, the advantages of software reusability can be classified in two categories:

(a) the development time, which is reduced because instead of always rebuilding everything from the beginning, previously existing elements are recycled;

(b) the reliability of software: an existing element has a more predictable behavior, since it has not only been tested by developers but also used by real users who may have experienced and reported its bugs, its anomalies and other information usually not documented.

There are also limitations to software reusability. The first major drawback is the cost associated with the process of organizing and putting into effect a corporate reuse policy: despite the effectiveness of adopting a reuse policy as an investment in the long run, in the short term it can be too expensive and not justified by real benefits [15]. Unfortunately, some (or most) software houses are run on a short term basis. Another disadvantage can be that a piece of software can be made much more contorted in order to make it more general, believing that in shape it would be more reusable; however, an appropriate approach to design for reuse and domain analysis may help in limiting such a risk.

2.3 Software portability

The concept of portability is rather intuitive and corresponds to how simply we can move a chunk of code from one machine to a different one.

More formally, we can define the portability \( \Pi(P, A, A_{os}, B, B_{os}) \) of a program \( P \) developed for a machine \( A \) with an operating system \( A_{os} \) to a machine \( B \) with operating system \( B_{os} \) as the easiness of making \( P \) running on \( B \) using \( B_{os} \). Using this definition, we can define the general concept of portability \( \Pi(P) \) of a program \( P \) as the minimum of its portabilities to any target architecture and operating system, i.e.

\[
\Pi(P) = \min \{ \Pi(P, A, A_{os}, B, B_{os}) \}
\]

This amounts to saying that given \( P \) that was originally developed for a PC 386 with DOS, if it can be easily ported on a PC 486 still with DOS but the task is to port it on a Pentium with Windows, then its value of portability is very low; this corresponds to the intuitive notion of portability which refers generally to the porting to any machine, not just to one.

Although a clear and measurable definition of easiness is still to come, people talking of the portability of a program usually restrict their attention to architectures of the same kind as the one in which the program has been initially developed. If the program \( P \) originated on a sequential
machine such as the Sun 3/50, one may be interested in porting it to another sequential machine, such as the Sun 10 or an Intel 80486 with Windows or a PowerPC with system 7, but it is very rare when discussing the portability of P to refer to the easiness of porting P to a Cray 2 and making full use of the parallelism.

The importance of the portability of a program is dramatic: whenever a new kind of machine or operating system comes out or a new machine is bought it is essential that as many programs as possible compile and run on it without too much trouble. Portability saves work, time and therefore money! Furthermore, as for reusability, portability increases the reliability of a program since the more a program is used on a wide range of architecture, the more likely it is that bugs are detected and fixed. However, programmers often pay insufficient attention to portability. If anyone experienced the problem of porting software from a Sun 3/X with SunOS 4.0.3 to a Sparc Sun 4 still with SunOS 4.0.3, she/he will know how often people prefer to cast pointers directly in the source code rather than using the macros provided by the standard header files such as varargs.h! It is therefore extremely important that it is possible to define a standard mechanism to ensure a high level of program portability.

3. Reusability of logic programs

3.1 Understanding logic programs

As stated earlier, the first condition for a code to be reused is that it can be understood. Understanding a program can be viewed according to three different levels [16]:

(a) the level of representing the computer program in an abstract way as a set of entities connected through meaningful relations;
(b) the level of modelling in a formal way the interesting portion of the world;
(c) the level of mapping between entities in (a) and entities in (b).

Logic programming helps the most at level (a): a program is represented as a set of terms related to one-another through facts and predicates; no details about implementation are present\(^2\). Level (b) is up to the programmer: it is her/his duty to represent the world in a suitable way. Level (c) should also be easier in a logic environment, since the logic program should be nothing more than an executable specification of a solution model. However the should in the previous sentence is mandatory, since the representation of the code strictly depends upon the reasoning of people.

3.2 Side effects, ripple effects and logic programs

A major problem connected with understanding code with the goal of reusing it is the presence of side effect instructions. Side effect instructions are dangerous since they do not only what they are supposed to, but also something else, causing some other part of the program to behave unexpectedly when they are changed. A similar problem is the ripple effect, that is the fact that changing a

\(^2\) Remember that we are speaking of pure logic programs, not of Prolog or LISP that are not pure logic languages. Candidates for such discussion can be Miranda, Scheme, SCL and any other pure formalism.
piece of code in one part of a program affects also the behaviour in other parts of the program [17]. These problems are partially solved using a logic paradigm of computing: logic languages do not have side effects by definition since they do not specify the control of a program but its logic. Furthermore the fact that logic languages exhibit referential transparency limits strongly the ripple effect since the order of the instructions is not significant for the result of the computation [18]; it is nice to be able to use portions of code without being bothered too much by the ordering of any details of the blocks that are composed. Referential transparency guarantees the programmer freedom from any concerns of side effects or ordering effects. Recall the example of section 2.1, the ordering of the lines was irrelevant for the Miranda definition:

\[
x1 = (-b - d)/(2*a)
\]
\[
x2 = (-b + d)/(2*a)
\]
\[
d = \sqrt{\text{delta}}
\]
\[
delta = b*b - 4*a*c
\]

Using an old fashioned, imperative-oriented vocabulary, the ‘variables’ \( d \) and \( \text{delta} \) are ‘used before set’. This fact does not raise any problem since here there is no assignment, just names definitions. Wherever such names are defined, their values are the same. It is up to the compiler to find the place where such definitions are used. Compare now the above declarative segment with the imperative counterpart; consider the following four lines taken from the code of section 2.1:

\[
delta := b*b - 4*a*c;
\]
\[
d := \sqrt{\text{delta}};
\]
\[
\text{sol}.x1 := (-b - d)/(2*a);
\]
\[
\text{sol}.x2 := (-b + d)/(2*a);
\]

Here the ordering of the first three instructions must be strictly preserved, since otherwise the algorithm is wrong; for instance if we had:

\[
\text{sol}.x1 := (-b - d)/(2*a);
\]
\[
\text{sol}.x2 := (-b + d)/(2*a);
\]
\[
delta := b*b - 4*a*c;
\]
\[
d := \sqrt{\text{delta}};
\]

then if the variable \( d \) had been previously initialized, the compiler would not generate any error message (assuming a smart compiler which recognizes unassigned variables appearing in the RHS of an assignment), and the result would be unpredictable.

3.3 Patterns and logic programs

A relevant role in the field of code understanding and reuse is played by patterns and frameworks [19,20]. Using logic languages it is possible to define functions that perform some general kind of operation using as a parameter some specific function, which is then specified applying those functions\(^3\) to ‘basic’ functions to obtain the desired result. For instance it is possible to define the pattern of applying a given function to all the elements of a list; the function \( \text{map} \) does this: it takes as argument a parameter function and a list and applies the parameter function to all the

\(^3\) Such functions are called higher order functions, to stress the fact that they are used to produce other functions.
elements of the list; this can be specified using Miranda [44] in the following way:

\[
\begin{align*}
\text{map} f \ [\ ] &= [] \\
\text{map} f \ [h \mid t] &= [f \ h \mid \text{map} \ f \ t]
\end{align*}
\]

In this way if one wants to define the function \textit{IncrList} that increments all the elements of a list, the function \textit{map} can be used with a function \textit{IncrEl} that given an integer returns its successor:

\[
\begin{align*}
\text{IncrEl} \ x &= x + 1 \\
\text{IncrList} &= \text{map} \ \text{IncrEl}
\end{align*}
\]

Intuitively, the first line defines the function that returns the successor of an element and the second line declares that \textit{IncrList} is a 'synonym' for the pair \textit{map} \textit{IncrEl}; therefore \textit{IncrList} \ [1, 2, 3] produces the same results as \textit{map} \textit{IncrEl} \ [1, 2, 3], namely \ [2, 3, 4].

Furthermore, software reusability is strongly enforced when we can treat a software component as a black box [21]. This aspect is entirely satisfied using logic languages since by definition logic languages specify the logic of a program and not its control: no side effect occurs using a pure logic framework.

4. Portability of logic programs

Logic programs are \textit{per se} portable. Since the programmer specifies only its logic and not the control, there is no conceptual limit to the portability of a logic program. The problems arise when the efficiency of such programs is considered; however research in this area [10, 22 - 24] demonstrates that it is possible to achieve almost the same efficiency for logic programs as that of imperative ones.

Logic programs provide complete portability not only in the framework of sequential architectures, but also for parallel architectures, both for MIMD and SIMD, as proved in [25 - 27]. Again, the constraints are on the efficiency of such languages, but this is also a fertile area of research.

4.1 Developing compilers for portable logic languages

A potential limitation in the portability of a logic program could be the difficulty of writing a compiler for it. In other words, while a program written with a logic language can be ported to many different platforms, provided that the suitable compiler exists, writing such a compiler is not easy and usually results in systems that are not portable, since each compiler tries to take advantage of the architecture of its target machine. Furthermore, writing a compiler for a logic language is a much more difficult task than for a \textit{conventional} imperative language: the assembler of any machine is by definition an imperative language\(^4\). Usually the compiler of a logic language is divided in two modules, using as intermediate medium an abstract machine [28]:

(a) a translator from the source code to an imperative assembler for the abstract machine;
(b) the implementation of the abstract machine on the real architecture.

\(^4\) The special purpose machines, such as the LISP machine are not considered here, since their results and applicability were rather limited.
The architecture of the abstract machine is usually quite generic [29], as any Von Neuman machine, and it maintains lots of high-level constructs such as explicit loops and function calls. Therefore in module (a) the focus is only on the translation from the logic paradigm to the imperative one and on all the possible optimizations that can be accomplished, such as tail recursion, last call optimization, environment trimming and so on.

Module (b) takes advantage of the genericity of the abstract machine architecture since it is not constrained to any particular approach and can be adapted according to the needs of many different environments [30]; for the SAM [26], the Subset Abstract Machine of SEL [31], there are different implementations for the transputer and the connection machine that accept as input exactly the same abstract assembler, as is depicted in Fig. 1. This approach is taken in almost all the approaches to the compilation of logic languages, for instance in the G-Machine [31,32], in Tim [33], in GRIP [34], in the WAM [24,35], and so on.

With this twofold schema, module (a) is architecture independent and can be the same for almost any implementation of a given language, while module (b) is language independent and architecture dependent. This module is where most of the effort has to be placed to have an efficient execution of the language in the new environment. Yet however hard it might be to implement, it is implemented only once and then any program can benefit from it.

### 4.2 Portability and multimodal parallelism

Logic languages exhibit implicit parallelism, as mentioned in section 2.1. Implicit parallelism has a corollary which is strongly enforced in logic languages: **multimodal parallelism**.

There are two the forms of parallelism that are usually exploited by parallel machines [26]:

- process parallelism [36],
- data parallelism [37].

The most common parallel machines exploit only one of the two forms of parallelism: for instance SIMD machines take advantage of data parallelism⁵ while MIMD machines exploit process parallelism. This fact is reflected in imperative languages, where each kind of parallelism needs to be expressed by a well-defined language construct, and, what is much worse, languages are

---

⁵ The fact that not all data parallelism can be expressed in terms of SIMD architectures is discussed in [28]; the topic is omitted, since it is not strictly related to the content of this paper.
hardly ever suited to more than one form. Therefore, for instance, to program a SIMD machine such as the Connection Machine one uses C* [38,39] while for the transputer entirely different primitives are supplied, for instance those of CSTools [40–42]. On the other hand, in logic languages the parallelism is not fixed by an unmodifiable control flow, but it is deduced automatically by the compiler, therefore both process and data parallelism can be present in the same algorithm. This is what is called multimodal parallelism. Consider the following assertion:

\[
\begin{align*}
\text{DoubleList} &= \text{map (multiply 2)} \\
\text{IncrList} &= \text{map (plus 1)} \\
\text{IncrAndDouble 1} &= (\text{IncrList 1}, \text{DoubleList 1})
\end{align*}
\]

Both DoubleList and IncrList have data parallelism: the multiplication by 2 and the increment of 1 can be applied in one step to all the elements of the list; furthermore, in IncrAndDouble the two can run simultaneously taking advantage of process parallelism, since these two operations neither depend on nor interfere with one another. This means that the same code can be ported on different platforms with different forms of parallelism.

This structure fits also the framework of a heterogeneous platform where different parallel machines are attached to the very same host, in an environment as the one suggested in [43]: the SAM assembler provides instructions that fit different kinds of parallel architectures so that the front-end host can decide where each chunk of code will run; a complete discussion of this topic together with a sample environment with a Sun 4 host connected to a transputer system and a Connection Machine 2 is discussed in [28]. Figure 2 presents this scenario. Furthermore this environment can take full advantages of the new machines where data parallelism and process parallelism are combined, like the CM5 [44] and the J-Machine [43]. All these considerations lead to the conclusion that the core of the compiler construction is the implementation of the abstract machine on the different target architectures. But this problem, however hard, is not much harder than the well standardized problem of writing a compiler of an imperative language for a (possibly parallel) machine.
5. What is lacking in this approach

Despite the desire of the writers, logic languages are not the philosophers' stone for easing software reusability and increasing software portability. They are an excellent programming paradigm, however they present some pitfalls. This section is devoted to exploring such pitfalls and to identifying possible solutions.

The first limit of logic languages is the fact that they are not widely used and it is not obvious that a programmer used, say, to C can convert him/herself to any of these: it is much easier for her/him to learn and adapt him/herself to C++, at least in the short term, and lots of software industries seem not interested in long-term investment.

Logic languages have limitation in handling I/Os and GUIs. It is usually not immediate to define a paradigm for representing such transaction-oriented operations. Sometimes they are represented with side effect, non-pure assertions, such as in Prolog as explained in [2] or in LISP (see [8]), however this amounts to admitting that such a paradigm is not per se enough for any common programmer needs and this is not the scope of this research. Haskell defines two approaches for handling I/O and both of them are quite involved. Stream-oriented logic languages, such as one of the CFP family [45], handle I/O instructions more easily; however, involved I/O operations are still hard to manage, also using them [46].

Logic languages do not provide any serious support to code modularizing. Haskell defines a framework for building ADT with some inheritance, but this approach is too complicated to be used by a normally educated programmer and does not solve the real problem of having serious and easy structured information hiding and data abstraction mechanisms. λ-prolog uses the higher order logic concept of theories to represents modules, however, a sound, complete and efficient λ-prolog machine has yet to come [47]. The new object-oriented logic languages may be a solution to such a problem, but this is still a matter of research.

Logic languages are neither widespread nor standardized, and the difference among them can be huge: this is perhaps the major drawback of such systems. Therefore it is impossible to teach people an almost universal programming paradigm and to let programmers replicate its pattern to any real language, as happens in the imperative side with Pascal: most students have been learning as a first language Pascal, and then the conversion to C, C++, Ada and many other languages is just a matter of understanding how to express the same instructions using a different (but not too different) formalism; an averagely educated programmer cannot find any evident analogies between a piece of LISP code and one of Prolog code. We understand that as long as people do not start using a logic paradigm, logic languages will not be standardized; however, the only solution to approach a recursive problem such as this is to start making normal programmers aware of the benefits of logic languages, and logic language designers aware of the problems of normal people.

6. Conclusion and further research

This paper demonstrates that there are strong connections between software reusability, software portability and logic languages. The keystones of these connections are the following
properties of logic languages:

- potential for easier understandability than conventional imperative languages, provided that an intuitive logic paradigm has been used;
- absence of side effects and ripple effects;
- potential for defining patterns;
- easy development of portable compilers;
- multimodal parallelism.

However, some are present in this approach, even if there seems to be means for overcoming them. The major drawback of this approach is the lack of industrial evidence of its effectiveness, and it is hard to persuade people from industry to try it in the real world. Other topics to be investigated further are I/O handling, data structuring, object orientation and standards. There is active research in these fields and the trends seem to indicate effective results.

A final remark can be made on the relevance of teaching declarative languages in order to educate people into software reusability: experience has been gathered on a senior class on software engineering: a first analysis of the results seems very promising, when such experience is more mature it will be the core of a further paper.

Acknowledgements

The starting point of this work has been the research conducted by the first author under the supervision of Professor Bharat Jayaraman at SUNY/Buffalo: special thanks go to him.

The authors thank Professor Gianni Jacucci, the initiator of the research on software reuse at the Università di Trento, Professor Vincenzo D’Andrea and Professor Maurizio Marchese for their fruitful comments. Also thanks go to the anonymous referees for their comments.

References

24. Y. N. Patt and A. Chen. A Performance Comparison Between the VLSI-PLM and the MC69020 as Prolog Processor, Computer Science Division, Univ. of California, Berkely, 1989.