Is Strategic Programming a Viable Paradigm?

P. Klint

Centrum voor Wiskunde en Informatica (CWI)
Kruislaan 413, 1098 SJ Amsterdam
The Netherlands

Abstract This discussion paper for the Workshop on Reduction Strategies in Programming and Rewriting (WRS'01) speculates on the viability of strategic programming in mainstream programming and in two selected application areas: theorem proving and programming-in-the-large.

1 Introduction

Since the early days of computing evaluation strategies have played a role in many programming languages. In this discussion paper I will explore the viability of "strategic programming", a paradigm in which a strategy language describes in what order certain atomic steps have to be performed. Typically, the strategy language contains primitives like sequential composition, non-deterministic choice and repetition. The atomic steps can come from different paradigms ranging from a single rewrite step or a function, to imperative statements or method calls in an object-oriented framework. The following issues will be discussed:

- What has been the success of evaluation strategies in programming languages aiming at programming-in-the-small, i.e., the construction of programs at the level of individual components?
- The same question, for other applications such as programming-in-the-large (i.e., the construction of complete systems out of individual components) and theorem proving.
- What are the implications for strategic programming?

2 Programming-in-the-small

2.1 Innermost evaluation

Prehistoric languages like assembler, Fortran and Cobol all use innermost ("eager") evaluation as their execution model. The first general-purpose languages to propose slight deviations from innermost evaluation are Algol 60 and Lisp.

In Algol 60 *call-by-name* parameters of procedures provided a deviation from the standard innermost evaluation order. In the case of a call-by-value parameter of a procedure P, the value of the actual parameter is computed once and used throughout the execution of the body of P. Note that the actual parameter may be a complicated expression. In the case of call-by-name, the value of the formal parameter is determined by evaluating the value of the actual parameter expression *at each occurrence* in P. This is very much like outermost evaluation. However, since variables occurring in the actual parameter expression may be changed between two occurrences, successive values need not be the same. A typical application of call-by-name in Algol 60 is the successive computation of an expression for different parameter values. This technique, known as Jensen's device, has been used with success in numeric applications.

In Lisp similar mechanisms exist. On the one hand *quoted expressions* (program fragments that should not be evaluated) and on the other hand an *explicit evaluation function*. By combining these two mechanisms very complicated evaluation orders can be implemented.

Although call-by-name is a simple mechanism, its ramifications when incorporated in an imperative language became only clear long after Algol 60 and Lisp were designed. Inclusion in a functional language is more natural.

2.2 Outermost evaluation

In more modern times, at the other end of the spectrum, functional languages like Haskell or Clean use outermost ("lazy") evaluation as execution model. The claimed advantages of outermost evaluation are the possibility to represent infinite data structures (taking the n-the element of an infinite list will only inspect the first n list-elements) and potential efficiency gains since unnecessary computations will not be performed. A well-known disadvantage is that debugging of programs in lazy languages is much harder since computations are performed in an order that is unfamiliar to the programmer.

2.3 Backtracking

Not in the main stream, but popular in its days, is a language like SNOBOL4 with a drastically different execution model. Its application area is string processing and its main execution model is pattern-directed execution. The execution of pattern matching is based on backtracking. SNOBOL4 patterns provide a wealth of features:

- Atomic actions like matching a string or evaluating an arbitrary predicate.
- Operators for sequential composition, alternative, and repetition.
- Two flavors of assignment of matched sub-patterns to variables: *immediate assignment* (during the match) and *conditional assignment* (at the end of a completely successful match).
- Immediate and conditional execution of arbitrary functions.
- Dynamic creation of patterns during pattern matching.

One can write amazingly short programs in SNOBOL4 to solve complex problems.

Another language based on backtracking is, of course, Prolog. In pure Prolog, backtracking is used to answer queries given sets of facts and rules. For efficiency reasons, all Prolog dialects contain a *cut* operator that can be used to influence the backtracking process. When it is known that parts of the search space will not yield an answer, the cut operator can be used to skip that part of the search space. Backtracking-based languages are very expressive but the flow-of-control in a program may be very hard to follow.

It is interesting to observe that these backtracking-based execution models have been superseded by more limited, declarative, mechanisms. In the case of string matching, declarative syntax definitions and implementations based on parser generation have taken over. In the case of logical programming, more controlled paradigms are being tried.

2.4 Implications for Strategic Programming

In practice, only evaluation models are in use that are based on innermost evaluation. Language features that deviate from innermost evaluation are consistently considered to be difficult. Probably the most successful exception is the if-then-else statement that is always evaluated in an outermost manner. What can we learn from this? I have at least the following conjectures:

- Innermost evaluation is the most natural evaluation order for programming-in-the-small.
- Successful paradigms stay as close as possible to innermost evaluation.

I can only speculate why these conjectures would be true:

- Programming language education emphasizes innermost evaluation.
- In a language based on a single, fixed, strategy one can mostly forget about the strategy and concentrate on the atomic operations. This makes reading and writing of programs simpler.
- Human cognition favors innermost evaluation.
- The cognitive overhead of complex evaluation strategies is too large when the atomic operations are too small.
- Innermost evaluation can be implemented very efficiently.

Using strategies for programming-in-the-small is a balancing act between the conceptual complexity of the strategies used and the simplicity of the atomic actions.

3 Other applications

The imbalance between strategies and atomic actions can only be compensated for in applications with one of the following two characteristics:

- The strategy is an essential part of the application itself.
- The complexity of atomic steps is relatively high.

I will now discuss two application areas that fit this description.

3.1 Theorem proving

In the case of theorem proving axioms and proof rules are given and the problem is to prove or refute a potential theorem. The order in which proof rules are being applied is an essential part of the problem solution and it is very natural to express this order in the form of a strategy (also known as "tactic"). This is relevant for automatic theorem proving (the strategies guide the prover) as well as for interactive theorem proving (the strategy records decisions taken as a result of user interaction). It is also true that an individual proof step may constitute a complex computation.

3.2 Programming-in-the-large

In the case of programming-in-the-large the problem is how to combine program components to form a complete application. This problem can be approached from a static as well as from a dynamic point of view. In the former case, static import relations, parameter bindings and the like have to be considered. In the latter case, the dynamic interaction of components is to be considered. This is the realm of so-called *coordination languages* where some form of concurrent calculus (π-calculus, ACP, and the like) is used to describe the protocol between the cooperating components. This application area has precisely the characteristics mentioned before:

- The cooperation protocol ("strategy") between components is an essential part of the application.
- The granularity is high, i.e., components ("actions") perform semantically significant tasks.

In other words, understanding the cooperation protocol in relation to the components does not create a significant cognitive overhead as was the case for programming-in-the-small.

It is striking to see that many of the primitives in strategy languages (sequential composition, choice, repetition) are identical to the primitives in coordination languages. Most prominently missing from strategy languages are more process-oriented features like communication and dynamic process creation. A merger of strategy languages and coordination languages looks like an intriguing field of research.

4 Conclusions

In the above speculations the following issues have surfaced:

- Mainstream programming languages are (and probably will be) based on innermost evaluation.
- Strategic programming is not likely to become a mainstream paradigm but can find its niches.

- Strategic programming can most profitably be applied in application areas where strategy is an essential aspect of the application and/or the complexity of atomic steps is relatively high.
- Strategic programming can profit from work on coordination languages.