Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 124 (2005) 113–118
www.elsevier.com/locate/entcs

Strategies in Programming Languages Today

Salvador Lucas

DSIC, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain.
slucas@dsic.upv.es

Abstract

This paper provides a brief account of motivation, themes, and research directions leading to the round table on *Strategies in Programming Languages Today* organized as part of the 4th International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04).

Keywords: Programming Languages, Strategies.

1 Introduction

The use of strategies in computational systems has a long history. Very simple formalisms like the λ-calculus already show that the right choice of the computational (reduction) step is essential to achive a good operational behavior when non-deterministic computations are possible: this is reflected in the well-known differences between the *normal* and *applicative* evaluation orders in λ-calculus. Sophisticated programming languages whose operational principle is based on (variants of) the reduction principle (e.g., CafeOBJ, Clean, Curry, ELAN, Haskell, Lisp, Maude, ML, OBJ*, Toy, ...) have made a choice (and appropriate adaptation) of these evaluation modes leading to the usual distinction between *lazy* and *eager* programming languages. This choice is part of the definition of the language and, once it is fixed in the implementation, it cannot be changed. This is consistent with the declarative spirit of these programming languages, where logic and control are not mixed. The programmer is (only) responsible for the logic of the program; the control is fixed by the designers and implementors.

doi:10.1016/j.entcs.2004.11.023

Although this is basically true, the daily use of these languages shows that perfect solutions are difficult to achieve. This was already clear for the λ-calculus, where the use of the normal and applicative evaluation orders has both pros and cons. The situation is similar in real programming languages, where things often become more complicated and psychological considerations can also be relevant [1] .

Thus, some efforts have been devoted to analyze when the use of a given –fixed– evaluation strategy could be improved and how to do it. For instance, regarding eager programming languages, implementations of Lisp where the list constructor operator (*cons*) did not evaluate its arguments during certain stages of the computation have been studied. Also, since each kind of strategy only behaves properly (i.e., it is normalizing, optimal, etc.) for particular classes of programs, the designers of programming languages have developed some features and language constructs aimed at giving the user more flexible (and explicit) control of the program execution. For instance, algebraic (eager) languages such as Maude, OBJ2, OBJ3, and CafeOBJ, admit the *explicit* specification of a particular class of *strategy annotations*, which (basically) are lists of integers associated to function symbols which specify the ordering in which the arguments are (eventually) evaluated in function calls. Other eager programmming languages such as ELAN incorporate a powerful strategy language as an essential ingredient of its design; the programmer can use this strategy language to specify concrete evaluation strategies for the program.

In lazy functional programming, different kinds of syntactic annotations on the program (such as strictness annotations, or the global and local annotations used in Clean) have been introduced in order to drive local changes in the basic underlying lazy evaluation strategy and obtain more efficient executions. In these languages, constructor symbols are lazy, i.e., their arguments are not evaluated until needed. This permits structures that contain elements which, if evaluated, would lead to an error or fail to terminate. Since there are a number of overheads in the implementation of this feature, lazy functional languages like Haskell allow for explicit syntactic annotations on the arguments of datatype constructors, thus allowing an immediate evaluation.

[1] It is interesting to consider Hudak's remarks about the choice of the argument (evaluation) order for Haskell: *"left-to-right evaluation of arguments"* was prefered *"because it presented the simplest semantics to the user, which was judged to be more important than making the order of arguments irrelevant"* (see [3], Section 2.4.3)).

2 What is the problem?

The previous overview leads to the main goal of the WRS'04 round table: recently, there has been an increasing interest in developing and understanding the explicit use of programmable strategies (or explicit devices to modify a fixed strategy) in a number of programming languages. This is particularly true for Elan, Haskell and Maude, among others. We are, thus, faced with the following general question:

> *What is the concrete role of strategies in the realistic modeling, analysis, and optimization of programming languages?*

More precisely: our abstract models, definitions, and analysis techniques for dealing with reduction strategies are basically developed for full rewriting with Term Rewriting Systems (see [4] for a recent survey) and they do not easily apply to real programs using

(i) strategy languages (or a fixed strategy),
(ii) conditional TRSs,
(iii) type/sort information,
(iv) higher-order functions,
(v) polymorphism,
(vi) data structures,
(vii) built-in symbols,
(viii) ACI symbols,
(ix) shared information (graphs instead of terms),
(x) modules,
(xi) ...

As a motivating example here, consider the use of *innermost* rewriting strategies in programming languages using conditional rules; we are going to see that the definition of innermost rewriting $\to_i$ for these systems can be problematic. The system in the following example borrows a similar one in [1, Conclusions] [2] .

Example 2.1 Consider the following Conditional Term Rewriting System:

$$f(a) \to f(c)$$
$$a \to b \qquad \text{if} \quad f(a) \to f(c)$$

[2] The example in [1, Conclusions] is originally due to Claude Marché.

As soon as we accept that $f(a) \to_i f(c)$, we also have to accept that $a \to_i b$. Then, we get into an apparent contradiction with the usual definition of innermost rewriting (rewrite a redex provided that it does not contain any other redex) because we should then have $f(a) \to_i f(b)$ and, consequently, $f(a) \not\to_i f(c)$. In this case, however, we cannot conclude $a \to_i b$ anymore and we go back again to $f(a) \to_i f(c)$!

The previous example shows that intuitive and very popular strategies (like innermost) become counterintuitive (or unfeasible) in some cases (see also [2] for more examples in this sense). This means that reasoning about strategic programming in real programming languages can be difficult. In this setting, future research should hopefully clarify whether we have the appropriate

(i) notion of strategy
(ii) definition and methods for analyzing termination of programs
(iii) definition and methods for analyzing determinism, unicity of NFs,...
(iv) approach for analyzing complexity and measuring efficiency
(v) strategy languages
(vi) ...

In this sense, the aim of the WRS'04 round table was to provide a first account of experiences, techniques and results coming from the usual practice with well-known programming languages.

3 The landscape of strategies in programming languages

The participants in the round table were leading members of the teams or research communities which develop the aforementioned programming languages: Francisco Durán, as a member of the Maude development group, Claude Kirchner, as the leader of the ELAN team, and Ralf Lämmel from the Haskell community. The notes written by these authors are collected as part of this volume and discuss the state of the art and future developments regarding the role of strategies in the use of the corresponding programming languages.

Kirchner's contribution (*Strategic Rewriting*) provides a short general introduction and overview to the field of strategies in a declarative programming setting: emphasis in the classic *what and how do we program* is made, in connection with the specification and use of strategies in programming. Declarative languages and, in particular, rule-based languages (with their two main components: pattern matching and rule application) are specially well-suited to put strategies to work. The Elan experience is then described as a concrete

(and early) attempt to exploit programmable strategies as a programming tool. Kirchner also enumerates a number of *strategic challenges* including the development of languages and combinators to specify and build strategies; the analysis of properties of strategic rewriting (bringing the standard problems of reachability, termination, confluence, etc., to this field); the evaluation and transformation of strategies; and the use of strategies in real applications and current technology (including theorem proving, programming, XML technology, production and business rules, biocomputing, etc.).

Lämmel's contribution (*Programmable rewriting strategies in Haskell*) focuses on the idea that strategic programming can separate rewriting steps from the overall scheme of *traversal and evaluation*. This is illustrated by means of the *Strafunski* approach which introduces strategies (essentially) as polymorphic functions on datatypes (or 'term types') in Haskell. The merits of this approach are discussed and then a comparison to the generic programming framework is made through the alternative "Scrap Your Boilerplate" approach. Lämmel also finishes with an account of challenges (specially, but not only, oriented to Haskell users) which are 'readily waiting' some research attention: analysis of termination, *stupidity*, shortcutting, composability; new expressiveness opportunitites: type effects for strategies, object syntax à la ASF+SDF, graph management, combination of strategies and attribute grammars, use of programmable strategies in constraint and funcional logic programming, XML processing, etc.

Finally, Duran's contribution (*Maude's Internal Strategies*) pays attention to the use of *reflection* as a suitable tool to introduce strategic programming in reflective programming languages, in particular Maude. Strategies, then, are considered as a kind of metaprogramming capability. Although (as mentioned in the Introduction) Maude already includes a simple strategy language which is able to associate *strategy annotations* to the symbols of the signature, reflection can be used to introduce deeper modifications in the evaluation process. This is done by means of the META-LEVEL module which is briefly described in the paper thus showing how to use it to control the execution of programs. A number of applications of this basic procedure are enumerated: Real Time Maude, where a clear distinction between eager and lazy rules is implemented; Mobile Maude, where an object-fair strategy was implemented and used in that way; new commands for achieving the exhaustive evaluation and normalization of initial expressions; an extension of the aforementioned strategy language to cope with *on-demand* strategy annotations; a much more powerful and complete strategy language including sophisticated traversal and pattern matching facilities; invariant-driven strategies, etc.

References

[1] F. Durán, S. Lucas, J. Meseguer, C. Marché, and X. Urbain. Proving Termination of Membership Equational Programs. In P. Sestoft and N. Heintze, editors, *Proc. of ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation, PEPM'04*, pages 147-158, ACM Press, New York, 2004.

[2] B. Gramlich. On the (Non-)Existence of Least Fixed Points in Conditional Equational Logic and Conditional Rewriting. In I. Guessarian, editor, *Proc. 2nd Int. Workshop on Fixed Points in Computer Science, FICS'00*, pages 38-40, 2000.

[3] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. ACM Computing Surveys, 21(3):359-411, 1989.

[4] V. van Oostrom and R. de Vrijer. Strategies. In TeReSe, *Term Rewriting Systems*, Chapter 9. Cambridge University Press, 2003.