Pattern matching is a fundamental feature in many applications such as functional programming, logic programming, theorem proving, term rewriting and rule-based expert systems. Usually, patterns size is not constrained and ambiguous patterns are allowed. This generality leads to a clear and concise programming style. However, it yields challenging problems in compiling of such programming languages. In this chapter, patterns are pre-processed into a deterministic finite automaton.

With ambiguous or overlapping patterns a subject term may be an instance of more than one pattern. In this case, pattern matching order in lazy evaluation affects the size of the matching automaton and the matching time. Furthermore, it may even impact on the termination properties of term evaluations.

In this chapter, we engineer good traversal orders that allow one to design an efficient adaptive pattern-matchers that visit necessary positions only. We do so using genetic programming to evolve the most adequate traversal order given the set of allowed patterns. Hence, we improve time and space requirements of pattern-matching as well as termination properties of term evaluation.
4.1 Introduction

Pattern matching is an important operation in several applications such as functional, equational and logic programming [5, 18], theorem proving [4] and rule-based expert systems [3]. With ambiguous patterns, an input term may be an instance of more than one pattern. Usually, patterns are partially ordered using priorities.

Pattern-matching automata have been studied for over a decade. Pattern-matching can be achieved as in lexical analysis by using a finite automaton [2, 7, 8, 13, 20]. Gräf [7] and Christian [2] construct deterministic matching automata for unambiguous patterns based on the left-to-right traversal order. In functional programming, Augustsson [1] and Wadler [22] describe matching techniques that are also based on left-to-right traversal of terms but allow prioritised overlapping patterns. Although these methods are economical in terms of space usage, they may re-examine symbols in the input term. In the worst case, they can degenerate to the naive method of checking the subject term against each pattern individually. In contrast, Christian’s [2] and Gräf’s [7] methods avoid symbol re-examination at the cost of increased space requirements. In order to avoid backtracking over symbols already examined, like Gräf’s our method introduces new patterns. These correspond to overlaps in the scanned prefixes of original patterns. When patterns overlap, some of the added patterns may be irrelevant to the matching process. In previous work [15], we proposed a method that improves Gräf’s in the sense that it introduces only a subset of the patterns that his method adds. This improves both space and time requirements as we will show later. Sekar [20] uses the notion of irrelevant patterns to compute traversal orders of pattern-matching. His algorithm eliminates the pattern $\pi$ whenever its match implies a match for a pattern of higher priority than $\pi$. In contrast with Sekar’s method, we do not introduce irrelevant patterns at once [15].

The pattern-matching order in lazy evaluation may affect the size of the matching automaton, the matching time and in the worst case, the termination properties of term evaluations [22]. The adaptive strategy is the top-down left-to-right lazy strategy used in most lazy functional languages [20]. It selects the leftmost-outermost redex but may force the reduction of a subterm if the root symbol of that subterm fails to match a function symbol in the patterns. So, the order of such reductions coincides with that of pattern-matching. Using left-to-right pattern matching, a subject term evaluation may fail to terminate only because of forcing reductions of subterms when it is unnecessary before declaring a match. For the left-to-right traversal order, such unnecessary reductions are required to ensure that no backtracking is needed when pattern-matching fails.

In this chapter, we study strategies to select a traversal order that should normally improve space requirements and/or matching times. Indexes for a pattern are positions whose inspection is necessary to declare a match. Inspecting them first for patterns which are not essentially strongly sequential