Stratego: A Language for Program Transformation Based on Rewriting Strategies
System Description of Stratego 0.5

Eelco Visser

Institute of Information and Computing Sciences, Universiteit Utrecht,
P.O. Box 80089, 3508 TB Utrecht, The Netherlands
`visser@acm.org`, `http://www.cs.uu.nl/~visser`

1 Introduction

Program transformation is used in many areas of software engineering. Examples include compilation, optimization, synthesis, refactoring, migration, normalization and improvement [15]. Rewrite rules are a natural formalism for expressing single program transformations. However, using a standard strategy for normalizing a program with a set of rewrite rules is not adequate for implementing program transformation systems. It may be necessary to apply a rule only in some phase of a transformation, to apply rules in some order, or to apply a rule only to part of a program. These restrictions may be necessary to avoid non-termination or to choose a specific path in a non-confluent rewrite system.

Stratego is a language for the specification of program transformation systems based on the paradigm of rewriting strategies. It supports the separation of strategies from transformation rules, thus allowing careful control over the application of these rules. As a result of this separation, transformation rules are reusable in multiple different transformations and generic strategies capturing patterns of control can be described independently of the transformation rules they apply. Such strategies can even be formulated independently of the object language by means of the generic term traversal capabilities of Stratego.

In this short paper I give a description of version 0.5 of the Stratego system, discussing the features of the language (Section 2), the library (Section 3), the compiler (Section 4) and some of the applications that have been built (Section 5). Stratego is available as free software under the GNU General Public License from `http://www.stratego-language.org`.

2 The Language

In the paradigm of program transformation with rewriting strategies [14] a specification of a program transformation consists of a signature, a set of rules and a strategy for applying the rules. The abstract syntax trees of programs are represented by means of first-order terms. A signature declares the constructors of such terms. *Labeled conditional rewrite rules* of the form `L: l -> r where s`,

```
module lambda-transform
imports lambda-sig lambda-vars iteration simple-traversal
rules
  Beta : App(Abs(x, e1), e2) -> <lsubs>([(x,e2)], e1)
strategies
  simplify = bottomup(try(Beta))
  eager = rec eval(try(App(eval, eval)); try(Beta; eval))
  whnf  = rec eval(try(App(eval, id)); try(Beta; eval))
```

Fig. 1. A Stratego module defining several strategies for transforming lambda expressions using beta reduction. Strategy `simplify` makes a bottom-up traversal over an expression trying beta reduction at each subexpression once, even under lambda abstractions. Strategy `eager` reduces the argument of a function before applying it, but does not reduce under abstractions. Strategy `whnf` reduces an expression to weak head-normal form, i.e., does not normalize under abstractions or in argument positions. Strategies `eager` and `whnf` use the congruence operator `App` to traverse terms of the form `App(e1,e2)`, while strategy `simplify` uses the generic traversal `bottomup`. The strategy `lsubs` is a strategy for substituting expressions for variables. It is implemented in module `lambda-vars` using a generic substitution strategy.

with `l` and `r` patterns, express basic transformations on terms. A rewriting strategy combines rules into a program that determines where and in what order the rules are applied to a term. An example specification is shown in Figure 1.

A *strategy* is an operation that transforms a term into another term or fails. Rules are basic strategies that perform the transformation specified by the rule or fail when either the subject term does not match the left-hand side or the condition fails. Strategies can be combined into more complex strategies by means of a language of strategy operators. These operators can be divided into operators for sequential programming and operators for term traversal. The sequential programming operators *identity* (`id`), *failure* (`fail`), *sequential composition* (`;`), *choice* (`+`), *negation* (`not`), *test*, and *recursive closure* (`rec x(s)`) combine strategies that apply to the root of a term. To achieve transformations throughout a term, a number of *term traversal primitives* are provided. For each constructor `C/n`, the corresponding *congruence* operator `C(s1,...,sn)` expresses the application of strategies to the direct sub-terms of a term constructed with `C`. Furthermore, a number of term traversal operators express *generic traversal* to the direct sub-terms of a term without reference to the constructor of the term. These constructs allow the generic definition of a wide range of traversals over terms. For example, the strategy `all(s)` applies `s` to each direct sub-term of a term. Using this operator one can define `bottomup(s) = rec x(all(x); s)`, which generically defines the notion of a post-order traversal that visits each sub-term applying the parameter strategy `s` to it.

A number of abstraction mechanisms are supported. A *strategy definition* of the form `f(x1,...,xn) = s` defines the new operator `f` with `n` parameters as an abstraction of the strategy `s`. An *overlay* of the form `C(x1,...,xn) = t` captures the pattern `t` in a new pseudo-constructor `C` [9]. Constructors and strategy operators can be overloaded on arity. Strategies implemented in a foreign language (e.g., for accessing the file system) can be called via the `prim` construct.

The distinction between rules and strategies is actually only idiomatic, that is, rules are abbreviations for strategies that are composed from the actual primitives of transformation: matching terms against patterns and building instantiations of patterns. Thus, a rule `L: l -> r where s` is just an abbreviation of the strategy `L = {x1,...,xn: ?l; where(s); !r}`, where the `xi` are the variables used in the rule. The construct `{xs: s}` delimits the scope of the variables `xs` to the strategy `s`. The strategy `?t` matches the subject term against the pattern `t` binding the variables in `t` to the corresponding sub-terms of the subject term. The strategy `!t` builds an instantiation of the term pattern `t` by replacing the variables in `t` by the terms to which they are bound. Decoupling pattern matching and term construction from rules and scopes, and making these constructs into first-class citizens, opens up a wide range of idioms such as contextual rules and recursive patterns [9]. In these idioms a pattern match is passed on to a local traversal strategy to match sub-terms at variable depth in the subject term.

Finally, specifications can be divided into modules that can import other modules. The above constructs of Stratego together with its module system make a powerful language that supports concise specification of program transformations. An operational semantics of System S, the core of the language, can be found in [13,14]. A limitation of the current language is that only a weak type system is implemented. Work is in progress to find a suitable type system that reconciles genericity with type safety.

3 The Library

The Stratego Library [10] is a collection of modules ($\approx$45) with reusable rules ($\approx$130) and strategies ($\approx$300). Included in the library are strategies for sequential control, generic traversal, built-in data type manipulation (numbers and strings), standard data type manipulation (lists, tuples, optionals), generic language processing, and system interfacing (I/O, process control, association tables).

The generic traversal strategies include one-pass traversals (such as topdown, bottomup, oncetd, and spinetd), fixed point traversal (such as reduce, innermost, and outermost), and traversal with environments. The generic language processing algorithms cover free variable extraction, bound variable renaming, substitution, and syntactic unification [11]. These algorithms are parameterized with the pattern of the relevant object language constructs and use the generic traversal capabilities of Stratego to ignore all constructs not relevant for the operation. For example, bound variable renaming is parameterized with the shape of variables and the binding constructs of the language.

4 The Compiler

The Stratego Compiler translates specifications to C code. The run-time system is based on the ATerm library [4], which supports the ATerm Format, a representation for first-order terms with prefix application syntax. The library implements writing and reading ATerms to and from the external format, which is used to exchange terms between tools. This enables component-based development of transformation tools. For example, a Stratego program can transform abstract syntax trees produced by any parser as long as it produces an ATerm representation of the abstract syntax tree for a program.

The compiler has been bootstrapped, that is, all components except the parser are specified in Stratego itself. The compiler performs various optimizations, including extracting the definitions that are used in the main strategy, aggressive inlining to enable further optimizations and merging of matching patterns to avoid backtracking. A limitation of the current compiler is that it does not support separate compilation and that compilation of the generated code by `gcc` is rather slow, resulting in long compilation times (e.g., 3 minutes for a large compiler component). Overcoming this limitation is the focus of current work.

5 Applications

Stratego is intended for use in a wide range of language processing applications including source-to-source transformation, application generation, program optimization, compilation, and documentation generation. It is not intended for interactive program transformation or theorem proving.

Examples of applications that use Stratego are XT, CodeBoost, HSX and a Tiger compiler. XT is a bundle of program transformation tools [6] in which Stratego is included as the main language for implementing program transformations. The bundle comes with a collection of grammars for standard languages and many tools implemented in Stratego for generic syntax tree manipulation, grammar analysis and transformation, and derivation of tools from grammars. CodeBoost is a framework for the transformation of C++ programs [2] that is developed for domain-specific optimization of C++ programs for numerical applications. HSX is a framework for the transformation of core Haskell programs that has been developed for the implementation of the warm fusion algorithm for deforesting functional programs [8]. The Tiger compiler translates Tiger programs [1] to MIPS assembly code [12]. The compiler includes translation to intermediate representation, canonicalization of intermediate representation, instruction selection, and register allocation.

6 Related Work

The creation of Stratego was motivated by the limitations of a fixed (innermost) strategy for rewriting, in particular based on experience with the algebraic specification formalism ASF+SDF [7]. The design of the strategy operators was inspired by the strategy language of ELAN [3], a specification language based on the paradigm of rewriting logic [5]. For a comparison of Stratego with other systems see [13,14]. A survey of program transformation systems in general can be found in [15]. The contributions of Stratego include: generic traversal primitives that allow definition of generic strategies; break-down of rules into primitives match and build giving rise to first-class pattern matching; many programming idioms for strategic rewriting; bootstrapped compilation of strategies; a foreign function interface; component-based programming based on exchange of ATerms.

Acknowledgements. I would like to thank Bas Luttik, Andrew Tolmach, Zino Benaissa, Patricia Johann, Joost Visser, Merijn de Jonge, Otto Skrove Bagge, Dick Kieburtz, Karina Olmos, Hedzer Westra, Eelco Dolstra and Arne de Bruijn for their contributions to the design, implementation and application of Stratego.

References

1. A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.
2. O. S. Bagge, M. Haveraaen, and E. Visser. CodeBoost: A framework for the transformation of C++ programs. Technical report, Universiteit Utrecht, 2000.
3. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. Elan: A logical framework based on computational systems. In J. Meseguer, editor, *ENTCS*, volume 4, 1996. Workshop on Rewriting Logic and Applications 1996.
4. M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software—Practice & Experience*, 30:259–291, 2000.
5. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *ENTCS*, volume 4, 1996. Workshop on Rewriting Logic and its Applications 1996.
6. M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In *ENTCS*, 2001. Language Descriptions, Tools and Applications 2001.
7. A. Van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping*, volume 5 of *AMAST Series in Computing.* World Scientific, Singapore, 1996.
8. P. Johann and E. Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence.* (To appear).
9. E. Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, Trento, Italy, July 1999. Springer-Verlag.
10. E. Visser. *The Stratego Library.* Institute of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, 1999.
11. E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming (WGP2000)*, Ponte de Lima, Portugal, July 6, 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.
12. E. Visser. Tiger in Stratego: An exercise in compilation by transformation. `http://www.stratego-language.org/tiger/`, 2000.
13. E. Visser and Z.-e.-A. Benaissa. A core language for rewriting. In C. Kirchner and H. Kirchner, editors, *ENTCS*, volume 15, September 1998. Rewriting Logic and its Applications 1998.
14. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).
15. E. Visser et al. The online survey of program transformation. `http://www.program-transformation.org/survey.html`.