A Bootstrapped Compiler for Strategies (Extended Abstract)

Eelco Visser

Department of Computer Science, Universiteit Utrecht
P.O. Box 80089, 3508 TB Utrecht, The Netherlands
`http://www.cs.uu.nl/~visser/`, `visser@acm.org`

Abstract. Stratego is a language for the specification of program transformation based on rewriting strategies. The Stratego compiler is based on program transformation; it transforms a high-level Stratego specification via several intermediate representations to C. Several optimizations are performed on the intermediate representations. The compiler is bootstrapped, i.e., it is specified in Stratego itself. In this paper we give an overview of the Stratego compiler: architecture; issues in the compilation of strategies; some high-lights of the specification; and experience with using strategies for writing program transformations.

1 Introduction

One of the shortcomings of implementing algebraic specifications by means of term rewriting is the necessity to encode the strategy to apply 'equations' by means of functions. This obscures the equational nature of specifications and hampers their modularity because rules become part of one particular strategy. This observation was done in several projects that used ASF+SDF [3] to define language processors, in particular program transformations. For example: normalization of box expressions for pretty-printing [2]; normalization of syntax definitions [8]; transformation of C++ programs [4].

Stratego is a language for the specification of program transformations that overcomes this shortcoming by providing user-definable rewriting strategies. A rewriting strategy is an expression in a language of strategy operators that combine rules (usually via their labels) in a program that traverses a term and applies the rules.

The preliminary ideas for the strategy operators in Stratego, inspired by the specification formalism ELAN [1], include operators for sequential non-deterministic programming, data-type specific and generic term traversal. In [6] an implementation of these operators in ASF+SDF is described. That interpretive implementation style can be used to write strategies for ASF+SDF equations and can be applied in other settings as well; the traditional way of defining a strategy with functions is replaced by a style in which an evaluation function interprets a strategy expression. This allows the concise specification of various strategies and the use of one rule with many different strategies.

These preliminary ideas are further developed in [10, 11]. Rewrite rules are no longer primitives, but are broken down into operations for matching, building and variable scope. System S, the resulting set of strategy operators, provides the primitive operations for definining both rewrite rules and strategies for applying them. A Stratego specification provides syntactic abstractions on top of System S. For example, a rewrite rule is an abstraction for a sequence of operations that first matches the subject term against a pattern, then satisfies a condition and finally builds the instantiation of a term pattern. The identification of this intermediate level allows the definition of very expressive abstractions such as contextual rules and overlays [9].

The Stratego compiler first translates a high-level Stratego specification to a System S expresssion, which is then translated to a list of abstract machine instructions that are implemented in C. The first compiler was written in SML. Based on the first experience with that compiler an improved compiler was specified in Stratego itself and bootstrapped. Bootstrapping proves to be a good approach for developing the compiler and the language because it provides a realistic case study and a good test case for the compiler. The compiler is being used in several program transformation case studies, such as a specificiation of an optimizer for a functional language [11], a deforestation algorithm for a functional language [5] and a transformation tool to speed up C++ programs for high-performance computing.

In this abstract we give an overview of System S and Stratego, present the architecture of the compiler and show some examples of the use of strategies in the compiler. In the full paper we will further elaborate the application of strategies in the compiler and give a first evaluation of the use of strategies in program transformation.

2 The Language

This section introduces System S, a calculus for the definition of tree transformations, and Stratego, a specification language providing syntactic abstractions for System S expressions. For an operational semantics see [10, 11].

2.1 System S

System S is a hierarchy of operators for expressing term transformations. The first level provides control constructs for sequential non-deterministic programming, the second level introduces combinators for term traversal and the third level defines operators for binding variables and for matching and building terms.

First-order terms are expressions over the grammar

```
t := x | C(t1,...,tn) | [t1,...,tn] | (t1,...,tn)
```

where `x` ranges over variables and `C` over constructors. The arity and types of constructors are declared in signatures. The notation `[t1,...,tn]` abbreviates the list `Cons(t1,...,Cons(tn,Nil))`. Transformations in System S are applied to ground terms, i.e., terms withouth variables.

```
module traversals
imports lists
strategies
  try(s)    = s <+ id              map(s)    = rec x(Nil + Cons(s, x))
  repeat(s) = rec x(try(s; x))     list(s)   = rec x(Nil + Cons(s, x))
  topdown   = rec x(s; all(x))     alltd(s)  = rec x(s <+ all(x))
  bottomup  = rec x(all(x); s)     oncetd(s) = rec x(s <+ one(x))
  downup(s) = rec x(s; all(x); s)  sometd(s) = rec x(s <+ some(x))
  onebu(s)  = rec x(one(x) <+ s)   somebu(s) = rec x(some(x) <+ s)
  downup2(s1, s2) = rec x(s1; all(x); s2)
```

Fig. 1. Specification of several generic term traversal strategies.

Level 1: Sequential Non-deterministic Programming Strategies are programs that attempt to transform ground terms into ground terms, at which they may succeed or fail. In case of success the result of such an attempt is a transformed term. In case of failure the result is an indication of the failure. Strategies can be combined into new strategies by means of the following operators: The *identity* strategy `id` leaves the subject term unchanged and always succeeds. The *failure* strategy `fail` always fails. The *sequential composition* `s1; s2` first attempts to apply `s1` to the subject term and, if that succeeds, applies `s2` to the result. The *non-deterministic choice* `s1 + s2` attempts to apply either `s1` or `s2`. It succeeds if either succeeds and it fails if both fail; the order in which `s1` and `s2` are tried is unspecified. The *deterministic choice* `s1 <+ s2` attempts to apply either `s1` or `s2`, in that order. The *recursive closure* `rec x(s)` attempts to apply `s`, where at each occurence of the variable `x` in `s`, the strategy `rec x(s)` is applied. The *test* strategy `test(s)` tries to apply `s`. It succeeds if `s` succeeds, and reverts the subject term to the original term. It fails if `s` fails. The *negation* `not(s)` succeeds (with the identity transformation) if `s` fails and fails if `s` succeeds. Two examples of strategies defined with these operators are `try` and `repeat` in Figure 1.

Level 2: Term Traversal The Level 1 constructs apply transformations to the root of a term. In order to apply transformations throughout a term it is necessary to traverse it. For this purpose, System S provides the following operators: For each `n`-ary constructor `C` the *congruence* operator `C(s1,...,sn)` is defined. It applies to terms of the form `C(t1,...,tn)` and applies `si` to `ti` for `1 <= i <= n`. An example of the use of congruences is the operator `map(s)` in Figure 1 that applies `s` to each element of a list.

Congruences can be used to define traversals over specific data structures. Specification of generic traversals (e.g., pre- or post-order over arbitrary structures) requires more generic operators. The operator `all(s)` applies `s` to all children of a constructor application `C(t1,...,tn)`. In particular, `all(s)` is the identity on constants (constructor applications without children). The strategy `one(s)` applies `s` to one child of a constructor application `C(t1,...,tn)`; it is precisely the failure strategy on constants. The strategy `some(s)` applies `s` to some of the children of a constructor application `C(t1,...,tn)`, i.e., to at least one and as many as possible. Like `one(s)`, `some(s)` fails on constants.

Figure 1 defines various traversals based on these operators. For instance, `oncetd(s)` tries to find *one* application of `s` somewhere in the term starting at the root working its way down; `s <+ one(x)` first attempts to apply `s`, if that fails an application of `s` is (recursively) attempted at one of the children of the subject term. If no application is found the traversal fails. Compare this to the traversal `alltd(s)`, which finds *all* outermost applications of `s` and never fails.

Level 3: Match, Build and Variable Binding The operators introduced thus far are useful for repeatedly applying transformation rules throughout a term. Actual transformation rules are constructed by means of pattern matching and building of pattern instantiations.

A match `?t` succeeds if the pattern term `t` matches the subject term. As a side-effect, any variables in `t` are bound to the corresponding subterms of the subject term. If a variable was already bound before the match, then the binding only succeeds if the terms are the same. This enables non-linear pattern matching, so that a match such as `?F(x, x)` succeeds only if the two arguments of `F` in the subject term are equal. This non-linear behaviour can also arise accross other operations. For example, the two consecutive matches `?F(x, y); ?F(y, x)` succeed exactly when the two arguments of `F` are equal. Once a variable is bound it cannot be unbound.

A build `!t` replaces the subject term with the instantiation of the pattern `t` using the current bindings of terms to variables in `t`. A scope `{x1,...,xn: s}` makes the variables `xi` local to the strategy `s`. This means that bindings to these variables outside the scope are undone when entering the scope and are restored after leaving it. The operation `where(s)` applies the strategy `s` to the subject term. If successful, it restores the original subject term, keeping only the newly obtained bindings to variables.

2.2 Stratego

The specification language Stratego provides syntactic abstractions for System S expressions. A specification consists of a collection of modules that define signatures, transformation rules and strategy definitions.

A signature declares the sorts and operations (constructors) that make up the structure of the language(s) being transformed. Example signatures are shown in the modules in Figure 2. A strategy definition `f(x1,...,xn) = s` introduces a new strategy operator `f` parameterized with strategies `x1` through `xn` and with body `s`. Such definitions cannot be recursive, i.e., they cannot refer (directly or indirectly) to the operator being defined. All recursion must be expressed explicitly by means of the recursion operator `rec`. Labeled transformation rules are abbreviations of a particular form of strategy definitions. A conditional rule `L : l -> r where s` with label `L`, left-hand side `l`, right-hand side `r`, and condition `s` denotes a strategy definition `L = {x1,...,xn: ?l; where(s); !r}`. Here, the body of the rule first matches the left-hand side, and then attempts

```
module terms
imports list-cons
signature
  sorts Term
  operations
    Wld :                        Term (* _ *)
    Var : String              -> Term (* x *)
    Int : Int                 -> Term (* 0, 1, 2 ... *)
    Str : String              -> Term (* "", "a", ... *)
    Op  : String * List(Term) -> Term (* f(t1,...,tn) *)

module strategy
imports terms
signature
  sorts SVar Strat SDef
  operations
    Id      :                                Strat (* id *)
    Fail    :                                Strat (* fail *)

    Test    : Strat                       -> Strat (* test s *)
    Not     : Strat                       -> Strat (* not s *)

    Seq     : Strat * Strat               -> Strat (* s1 ;  s2 *)
    Choice  : Strat * Strat               -> Strat (* s1 +  s2 *)
    LChoice : Strat * Strat               -> Strat (* s1 <+ s2 *)

    SVar    : String                      -> SVar
    Rec     : String * Strat              -> Strat (* rec x (s) *)
    Let     : SDef * Strat                -> Strat (* let sdef in s2*)
    SDef    : String * List(String) * Strat -> SDef  (* f(xs) = s *)
    Call    : SVar * List(Strat)          -> Strat (* f(ss) *)

    Path    : Int * Strat                 -> Strat (* i(s) *)
    Cong    : String * List(Strat)        -> Strat (* f(s1,...,sn) *)
    One     : Strat                       -> Strat (* one(s) *)
    Some    : Strat                       -> Strat (* some(s) *)
    All     : Strat                       -> Strat (* all(s) *)

    Match    : Term                       -> Strat (* ?t *)
    Build    : Term                       -> Strat (* !t *)

    Scope   : List(String) * Strat        -> Strat (* {xs: s} *)
    Where   : Strat                       -> Strat (* where s *)

    Prim    : String -> Strat
```

Fig. 2. Abstract syntax of terms and System S expressions.

to satisfy the condition `s`. If that succeeds, then it builds the right-hand side `r`. The rule is enclosed in a scope that makes all term variables `xi` occurring in `l`, `s` and `r` local to the rule. If more than one definition is provided with the same name, e.g., `f(xs) = s1` and `f(xs) = s2`, this is equivalent to a single definition with the sum of the original bodies as body, i.e., `f(xs) = s1 + s2`.

The following definitions provide a useful shorthand. The notation `<s> t` denotes `!t; s`, i.e., the strategy that builds the term `t` and then applies `s` to it. The notation `s => t` denotes `s; ?t`, i.e., the strategy that applies `s` to the current subject term and then matches the result against `t`. The combined notation `<s> t => t'` thus denotes `(!t; s); ?t'`. The `<s> t` notation can also be used in a build expression. For example, the strategy expression `!F(<s> t, t')` corresponds to `{x: <s> t => x; !F(x,t')}`, where `x` is a new variable.

2.3 Library

The language comes with a growing library of strategy operators with functionality for

- Simple traversals (such as in Figure 1)
- Fixed-point traversals
- List operations
- Tuple operations
- Manipulation of expressions with (bound) variables, such as variable renaming, substitution, collection of the set of free variables etc. These operations are language independent and can be specialized to a language by instantiation of generic operations. Note that this concerns *object* variables in the language being manipulated, which are different from the meta-variable used in rules.

3 The Compiler

A Stratego specification defines a transformation on terms. The Stratego compiler translates a specification to an executable program that reads in a term, transforms it according to the specification and outputs the resulting transformed term. In this section we discuss the architecture of the compiler and the run-time system used in the generated programs. In the next section we give some examples of the specification of the compiler in Stratego itself.

3.1 Architecture

The overall architecture of the compiler is shown in Figure 3. The compiler consists of four main components: front-end, optimizer, matching-tree, and back-end.

The front-end takes a Stratego specification (in abstract syntax form) and translates it to a list of System S expressions (SSE). The front-end itself is composed of five stages: joining sections of the same kind (normalization); translation

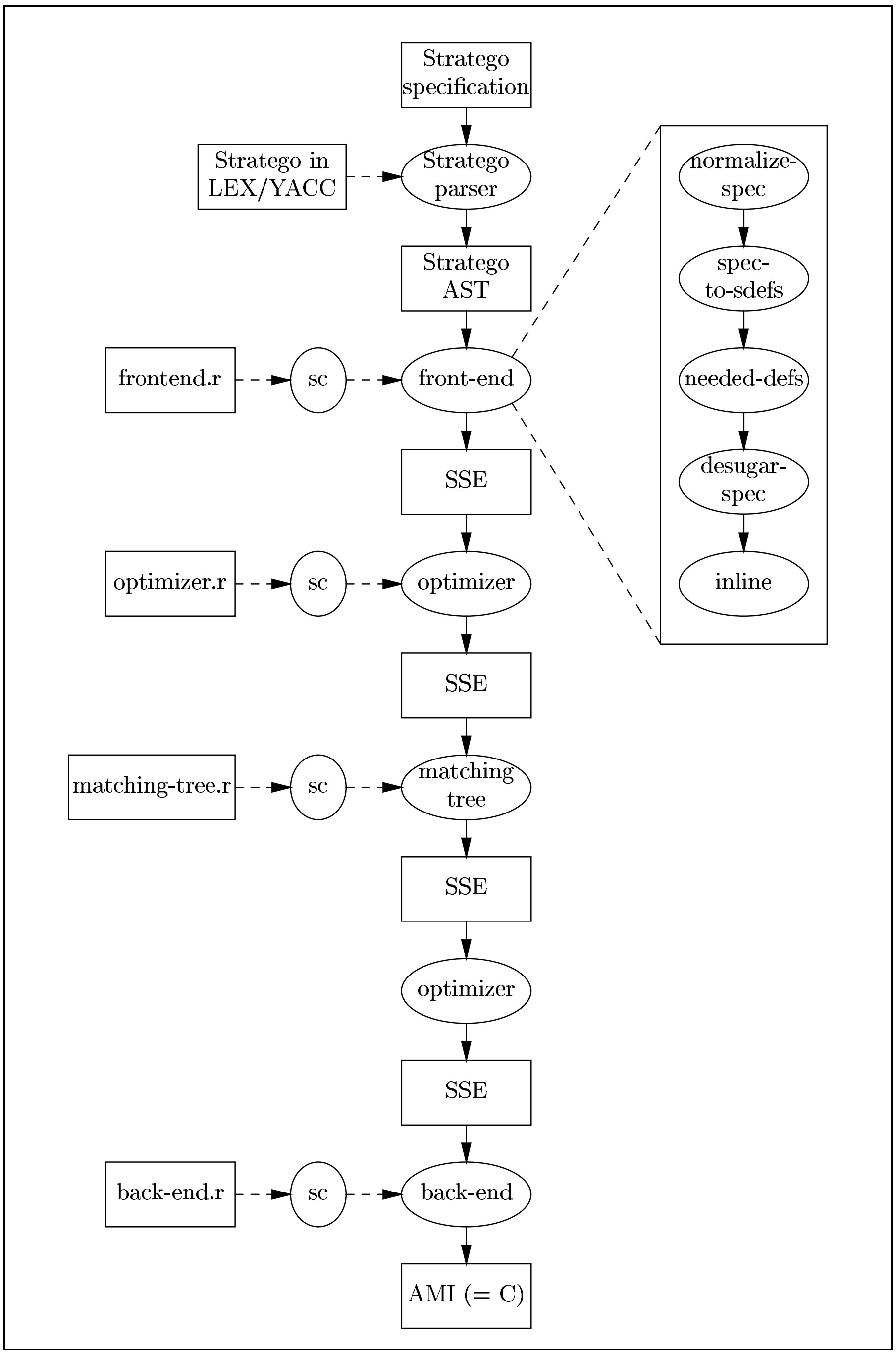

Fig. 3. Architecture of the Stratego compiler (sc)

of rules and signatures to strategy definitions; extraction of the definitions that are actually needed for implementation of the operator `main`; elimination of the syntactic abstractions (sugar) of the Stratego level; and (selective) inlining definitions. The result of this stage is a list of strategy definitions for parameterless operators.

The optimizer simplifies a System S expression by applying algebraic laws. The matching-tree automaton transformer targets expressions that are choices of strategies starting with a match. Common prefixes are extracted to prevent inspecting a term more than once. The back-end translates an expression to abstract machine instructions.

3.2 Run-time System

The abstract machine instructions produced by the compiler are implemented as macros or procedures in C. These procedures make use of a run-time system that supports stack and term management. The return- and choice-stacks are needed for control. The term-stack is used to deconstruct terms in matching and term traversal. For the representation of terms the ATerms package [7] is used. This package provides an implementation of terms based on hash-consing and supports garbage collection.

The run-time system is implemented in C. Therefore, the compiler and generated transformation programs only depend on `gcc`. The Stratego compiler (sc) is being developed on a Linux platform and is also used on Sun machines. Although this has not been done, there should be no problem in porting the compiler to Windows NT platforms with GNU software.

4 Examples

In this section we give some examples to illustrate the specification of the compiler in Stratego.

4.1 Example 1: Pipelines

The `main` strategy of the front-end defines a pipeline of operations. In addition to the operations discussed in the previous section, use-def analysis is used to determine if variables are used in builds without being bound in match operations.

Note that in general such a pipeline may fail to apply to a term. In this case, if an error is detected either in the `use-def` analysis (due to undeclared or unitialized variables) or in the `needed-defs` transformation (missing definition), the pipeline fail. In these cases an error message is derived from the failure.

```
module frontend
imports normalize-spec spec-to-sdefs needed-defs
        desugar inlining use-def
strategies

  main = normalize-spec;
         where(spec-use-def);
         spec-to-sdefs;
         needed-defs;
         desugar-spec;
         inline
```

4.2 Example 2: Desugaring

The following specification is a fragment from module `desugar.r` that defines the elimination of syntactic abstractions (sugar). Rules `Bapp2` transforms a build expression such as `!F(<s> G(y))` to `{x: <s> G(y) => x; !F(x)}` in order to extract the strategy application inside the build.

Rule `Bapp2` uses a contextual pattern `t[App(s, t')]` to reach an application `App(s, t')` at an arbitrary depth inside the term matching `t`. This application is replaced with the newly generated variable `Var(x)` in the right-hand side by means of the context `t[Var(x)]`. The right-hand side replaces the build by a scope construct that declares a new local variable `x`. Inside the scope first `s` is applied to `t'` and the result matched against the new variable `x`. This variable is then used inside the term `t` in the build.

The strategy `desugar` desugars an expression by applying a set of rules, including `Bapp2`, repeatedly in a topdown traversal. The strategy `desugar-spec` applies this strategy to each definition body in a list of definitions.

```
module desugar
 ...
rules

  Bapp2 : Build(t[App(s, t')]) ->
          Scope([x], Seq(BAM(s, t', Var(x)), Build(t[Var(x)])))
          where new => x

strategies

  desugar = topdown(try(desugarRule);
                    repeat(HL + (Bapp0 <+ Bapp1 <+ Bapp2)))

  desugar-spec = map(SDef(id, id, desugar))
```

4.3 Example 3: Compilation

The back-end of the compiler translates expressions to lists of abstract machine instructions. For each language construct a rule defines the pattern of instructions it corresponds to. For example, the following rules define the translation of left choice `<+` and the generic traversal operator `all`.

```
module compiler
 ...
rules
  C : Instr(LChoice(s1, s2), env, rcs) ->
      Block([Cpush(fc),
             Instr(s1, env, rcs), Cpop, Goto(sc),
             Label(fc), Instr(s2, env, rcs),
             Label(sc)])
      where new => sc ; new => fc

  C : Instr(All(s), env, rcs) ->
      Block([AllInit,
             Label(c1),
             AllNextSon(c2),
             Instr(s, env, rcs),
             Goto(c1),
             Label(c2),
             AllBuild])
      where new => c1 ; new => c2
```

These rules are combined in the compilation strategy `compile`. The default `C` rules are applied after the specialized `Cs` rules. After translation the nested code is flattened to a single list. Some simple optimizations are performed by `peephole` and finally, the code is wrapped in some support code by `Assemble`.

```
module backend
  ...
strategies

  compile = map(MkInstr;
                topdown(repeat(Cs <+ C)));
            flatten-blocks;
            peephole;
            Assemble
```

References

1. P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar, Pacific Grove, CA, September 1996. Elsevier.
2. M. van den Brand and E. Visser. From Box to TEX: An algebraic approach to the generation of documentation tools. Technical Report P9420, Programming Research Group, University of Amsterdam, July 1994.
3. A. Van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific, Singapore, September 1996.
4. T. Dinesh, M. Haveraaen, and J. Heering. An algebraic programming style for numerical software and its optimization. CWI Report SEN-R9844, CWI, Amsterdam, The Netherlands, December 1998.
5. P. Johann and E. Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. Technical report, Department of Computer Science, Universiteit Utrecht, 1999.
6. B. Luttik and E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.
7. P. A. Olivier and H. A. de Jong. Efficient annotated terms. Technical report, Programming Research Group, University of Amsterdam, August 1998.
8. E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997.
9. E. Visser. Strategic pattern matching. In *Rewriting Techniques and Applications (RTA'99)*, Lecture Notes in Computer Science, Trento, Italy, July 1999. Springer-Verlag.
10. E. Visser and Z.-e.-A. Benaissa. A core language for rewriting. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications (WRLA'98)*, Electronic Notes in Theoretical Computer Science, Pont-à-Mousson, France, September 1–4 1998. Elsevier.
11. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming (ICFP'98)*, pages 13–26, Baltimore, Maryland, September 1998. ACM.