Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems

in StrategoXT 0.9

Eelco Visser

Technical Report UU-CS-2004-011
Institute of Information and Computing Sciences
Utrecht University

February 2004

To appear as
E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9. In Lengauer et al., editors, Domain-Specific Program Generation, Lecture Notes in Computer Science. Spinger-Verlag, 2004.

See also
http://www.stratego-language.org/Stratego/ProgramTransformationWithStrategoXT

Copyright © 2004 Eelco Visser

Address:
Institute of Information and Computing Sciences
Utrecht University
P.O.Box 80089
3508 TB Utrecht

Eelco Visser <visser@acm.org>
http://www.cs.uu.nl/people/visser

Program Transformation with Stratego/XT

Rules, Strategies, Tools, and Systems in Stratego/XT 0.9

Eelco Visser

Institute of Information and Computing Sciences, Utrecht University
P.O. Box 80089 3508 TB, Utrecht, The Netherlands
visser@acm.org, http://www.stratego-language.org

Abstract. Stratego/XT is a framework for the development of transformation systems aiming to support a wide range of program transformations. The framework consists of the transformation language Stratego and the XT collection of transformation tools. Stratego is based on the paradigm of rewriting under the control of programmable rewriting strategies. The XT tools provide facilities for the infrastructure of transformation systems including parsing and pretty-printing. The framework addresses the entire range of the development process; from the specification of transformations to their composition into transformation systems. This chapter gives an overview of the main ingredients involved in the composition of transformation systems with Stratego/XT, where we distinguish the abstraction levels of rules, strategies, tools, and systems.

1 Introduction

Program transformation, the automatic manipulation of source programs, emerged in the context of compilation for the implementation of components such as optimizers [28]. While compilers are rather specialized tools developed by few, transformation systems are becoming widespread. In the paradigm of generative programming [13], the generation of programs from specifications forms a key part of the software engineering process. In refactoring [21], transformations are used to restructure a program in order to improve its design. Other applications of program transformation include migration and reverse engineering. The common goal of these transformations is to increase programmer productivity by automating programming tasks.

With the advent of XML, transformation techniques are spreading beyond the area of programming language processing, making transformation a necessary operation in any scenario where structured data play a role. Techniques from program transformation are applicable in document processing. In turn, applications such as Active Server Pages (ASP) for the generation of web-pages in dynamic HTML has inspired the creation of program generators such as Jostraca [31], where code templates specified in the concrete syntax of the object language are instantiated with application data.

Stratego/XT is a framework for the development of transformation systems aiming to support a wide range of program transformations. The framework consists of the transformation language Stratego and the XT collection of transformation tools. Stratego is based on the paradigm of rewriting under the control of programmable rewriting strategies. The XT tools provide facilities for the infrastructure of transformation

systems including parsing and pretty-printing. The framework addresses all aspects of the construction of transformation systems; from the specification of transformations to their composition into transformation systems. This chapter gives an overview of the main ingredients involved in the composition of transformation systems with Stratego/XT, where we distinguish the abstraction levels of rules, strategies, tools, and systems.

A transformation rule encodes a basic transformation step as a rewrite on an abstract syntax tree (Section 3). Abstract syntax trees are represented by first-order prefix terms (Section 2). To decrease the gap between the meta-program and the object program that it transforms, syntax tree fragments can be described using the concrete syntax of the object language (Section 4).

A transformation strategy combines a set of rules into a complete transformation by ordering their application using control and traversal combinators (Section 5). An essential element is the capability of defining traversals generically in order to avoid the overhead of spelling out traversals for specific data types. The expressive set of strategy combinators allows programmers to encode a wide range of transformation idioms (Section 6). Rewrite rules are not the actual primitive actions of program transformations. Rather these can be broken down into the more basic actions of matching, building, and variable scope (Section 7). Standard rewrite rules are context-free, which makes it difficult to propagate context information in a transformation. Scoped dynamic rewrite rules allow the run-time generation of rewrite rules encapsulating context information (Section 8).

A transformation tool wraps a composition of rules and strategies into a stand-alone, deployable component, which can be called from the command-line or from other tools to transform terms into terms (Section 10). The use of the ATerm format makes exchange of abstract syntax trees between tools transparent.

A transformation system is a composition of such tools performing a complete source-to-source transformation. Such a system typically consists of a parser and a pretty-printer combined with a number of transformation tools. Figure 1 illustrates such a composition. The XTC transformation tool composition framework supports the transparent construction of such systems (Section 10).

Stratego/XT is designed such that artifacts at each of these levels of abstraction can be named and reused in several applications, making the construction of transformation systems an accumulative process. The chapter concludes with a brief overview of typical applications created using the framework (Section 12). Throughout the chapter relevant Stratego/XT publications are cited, thus providing a bibliography of the project.

2 Program Representation

Program transformation systems require a representation for programs that allows easy and correct manipulation. Programmers write programs as texts using text editors. Some programming environments provide more graphical (visual) interfaces for programmers to specify certain domain-specific ingredients (e.g., user interface components). But ultimately, such environments have a textual interface for specifying the details. Even if

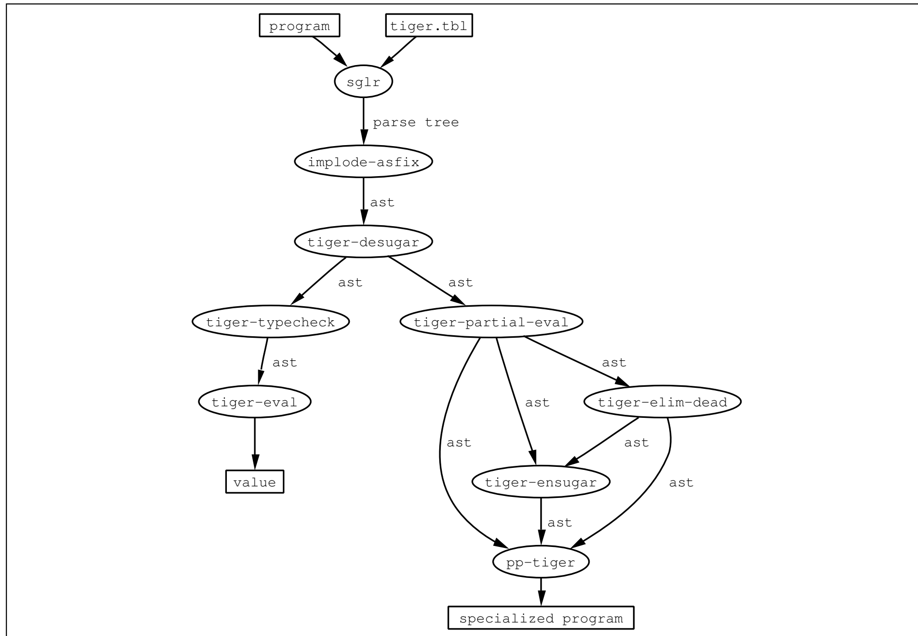

Fig. 1. Composition of a transformation system from tools.

programs are written in a 'structured format' such as XML, the representation used by programmers generally is text. So a program transformation system needs to manipulate programs in text format.

However, for all but the most trivial transformations, a structured rather than a textual representation is needed. Bridging the gap between textual and structured representation requires parsers and unparsers. XT provides formal syntax definition with the syntax definition formalism SDF, parsing with the scannerless generalized-LR parser SGLR, representation of trees as ATerms, mapping of parse trees to abstract syntax trees, and pretty-printing using the target-independent Box language.

2.1 Architecture of Transformation Systems

Program transformation systems built with Stratego/XT are organized as data-flow systems as illustrated by the data-flow diagram in Figure 1 which depicts the architecture of an interpreter and a partial evaluator for Appel's Tiger language. A program text is first parsed by `sglr`, a generic scannerless generalized-LR parser taking a parse table and a program as input, producing a parse tree. The parse tree is turned into an abstract syntax tree by `implode-asfix`. The abstract syntax tree is then transformed by one or more transformation tools. In the example, `tiger-desugar` removes and `tiger-ensugar` reconstructs syntactic sugar in a program, `tiger-typecheck` verifies the type correctness of a program and annotates its variables with type information, `tiger-eval` is an interpreter, and `tiger-partial-eval` is a partial evaluator. If the application of

these transformations results in a program, as is the case with partial evaluation, it is pretty-printed to a program text again by `pp-tiger` in the example.

2.2 Concrete Syntax Definition

Parsers, pretty-printers and signatures can be derived automatically from a syntax definition, a formal description of the syntax of a programming language. Stratego/XT uses the syntax definition formalism SDF [22,34] and associated generators. An SDF definition is a declarative, integrated, and modular description of *all* aspects of the syntax of a language, including its lexical syntax. The following fragment of the syntax definition for Tiger illustrates some aspects of SDF.

```
module Tiger-Statements
imports Tiger-Lexical
exports
  lexical syntax
    [a-zA-Z][a-zA-Z0-9]*            -> Var
  context-free syntax
    Var ":=" Exp                    -> Exp {cons("Assign")}
    "if" Exp "then" Exp "else" Exp -> Exp {cons("If")}
    "while" Exp "do" Exp            -> Exp {cons("While")}
    Var                             -> Exp {cons("Var")}
    Exp "+"  Exp                    -> Exp {left,cons("Plus")}
    Exp "-"  Exp                    -> Exp {left,cons("Minus")}
```

The lexical and context-free syntax of a language are described using context-free productions of the form s_1 ... s_n -> s_0 declaring that the concatenation of phrases of sort s_1 to s_n forms a phrase of sort s_0. Since SDF is modular it is easy to make extensions of a language.

2.3 Terms and Signatures

Parse trees contain all the details of a program including literals, whitespace, and comments. This is usually not necessary for performing transformations. A parse tree is reduced to an *abstract syntax tree* by eliminating irrelevant information such as literal symbols and layout. Furthermore, instead of using sort names as node labels, *constructors* encode the production from which a node is derived. For this purpose, the productions in a syntax definition contain *constructor annotations*. For example, the abstract syntax tree corresponding to the expression `f(a + 10) - 3` is shown in Fig. 2. Abstract syntax trees can be represented by means of *terms*. Terms are applications C(t_1,...,t_n), of a constructor C to terms t_i, lists [t_1,...,t_n], strings "...", or integers n. Thus, the abstract syntax tree in the example, corresponds to the term `Minus(Call(Var("f"), [Plus(Var("a"), Int("10"))]), Int("3"))`.

The abstract syntax of a programming language or data format can be described by means of an *algebraic signature*. A signature declares for each constructor its arity m, the sorts of its arguments S_1*...*S_m, and the sort of the resulting term S_0 by means of a constructor declaration c : S_1*...*$S_m \rightarrow S_0$. A term can be validated against a

```
module Tiger-Statements
signature
  constructors
    Assign : Var * Exp -> Exp
    If     : Exp * Exp * Exp -> Exp
    While  : Exp * Exp -> Exp
    Var    : String -> Exp
    Call   : String * List(Exp) -> Exp
    Plus   : Exp * Exp -> Exp
    Minus  : Exp * Exp -> Exp
```

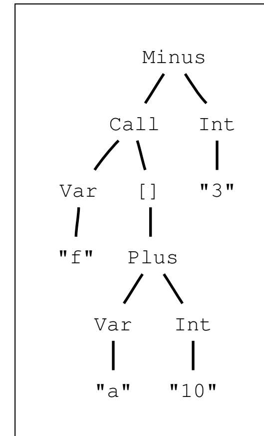

Fig. 2. Signature and abstract syntax tree

signature by a *format checker* [35].

Signatures can be derived automatically from syntax definitions. For each production $A_1...A_n \rightarrow A_0\{\texttt{cons}(c)\}$ in a syntax definition, the corresponding constructor declaration is $c : S_1 * ... * S_m \rightarrow S_0$, where the S_i are the sorts corresponding to the symbols A_j after leaving out literals and layout sorts. Thus, the signature in Figure 2 describes the abstract syntax trees derived from parse trees over the syntax definition above.

2.4 Pretty-Printing

After transformation, an abstract syntax tree should be turned into text again to be useful as a program. Mapping a tree into text is the inverse of parsing, and is thus called *unparsing*. When an unparser makes an attempt at producing human readable, instead of just compiler parsable, program text, an unparser is called a *pretty-printer*. Stratego/XT uses the pretty-printing model as provided by the Generic Pretty-Printing package GPP [14]. In this model a tree is unparsed to a Box expression, which contains text with markup for pretty-printing. A Box expression can be interpreted by different back-ends to produce formatted output for different displaying devices such as plain text, HTML, and LaTeX.

3 Transformation Rules

After parsing produces the abstract syntax tree of a program, the actual transformation can be applied to it. The Stratego language is used to define transformations on terms. In Stratego, rewrite rules express basic transformations on terms.

3.1 Rewrite Rules

A rewrite rule has the form `L : l -> r`, where `L` is the label of the rule, and the term patterns `l` and `r` are its left-hand side and right-hand side, respectively. A term pattern

is either a variable, a nullary constructor `C`, or the application `C(p1,...,pn)` of an `n`-ary constructor `C` to term patterns `pi`. In other words, a term pattern is a term with variables. A conditional rewrite rule is a rule of the form `L : l -> r where s`, with `s` a computation that should succeed in order for the rule to apply. An example rule is the following constant folding rule

```
EvalPlus : Plus(Int(i), Int(j)) -> Int(k) where <add>(i, j) => k
```

which reduces the addition of two constants to a constant by calling the library function `add` for adding integers. Another example, is the rule

```
LetSplit : Let([d1, d2 | d*], e*) -> Let([d1], Let([d2 | d*], e*))
```

which splits a list of let bindings into separate bindings.

3.2 Term Rewriting

A rule `L: l -> r` applies to a term `t` when the pattern `l` matches `t`, i.e., when the variables of `l` can be replaced by terms in such a way that the result is precisely `t`. Applying `L` to `t` has the effect of transforming `t` to the term obtained by replacing the variables in `r` with the subterms of `t` to which they were bound during matching. For example, applying rule `EvalPlus` to the term `Plus(Int(1), Int(2))` reduces it to `Int(3)`

Term rewriting is the exhaustive application of a set of rewrite rules to a term until no rule applies anywhere in the term. This process is also called normalization. For example, `Minus(Plus(Int(4), Plus(Int(1), Int(2))), Var("a"))` is reduced to `Minus(Int(7), Var("a"))` under repeated application of rule `EvalPlus`.

While exhaustive rewriting is the standard way that most rewriting engines apply rewrite rules, in Stratego one has to define explicitly which rules to apply in a normalization and according to which strategy. For example, a simplifier which applies a certain set of rules using the standard innermost strategy is defined as:

```
simplify = innermost(EvalPlus + LetSplit + ...)
```

The mechanism behind this definition will be explained in Section 5.

4 Concrete Object Syntax

In the previous section we saw that rewrite rules can be used to define transformations on abstract syntax trees representing the programs to be transformed, rather than on their text-based representations. But the direct manipulation of abstract syntax trees can be unwieldy for larger program fragments. Therefore, Stratego supports the specification of transformation rules using the *concrete syntax* of the object language [40]. In all places where normally a term can be written, a code fragment in the concrete syntax of the object language can be written. For example, using concrete syntax, the constant folding rule for addition can be expressed as:

```
EvalPlus : |[ i + j ]| -> |[ k ]| where <add>(i, j) => k
```

instead of the equivalent transformation on abstract syntax trees on the previous page. The use of concrete syntax is indicated by quotation delimiters, e.g.,. the `|[` and `]|` delimiters above. Note that not only the right-hand side of the rule, but also its matching left-hand side can be written using concrete syntax.

In particular for larger program fragments the use of concrete syntax makes a big difference. For example, consider the instrumentation rule

```
TraceFunction :
  |[ function f(x*) : tid = e ]| ->
  |[ function f(x*) : tid =
       (enterfun(f); let var x : tid in x := e; exitfun(f); x end) ]|
   where new => x
```

which adds calls to `enterfun` at the entry and `exitfun` at the exit of functions. Writing this rule using abstract syntax requires a thorough knowledge of the abstract syntax and is likely to make the rule unreadable. Using concrete syntax the right-hand side can be written as a normal program fragment with holes. Thus, specification of transformation rules in the *concrete syntax* of the object language closes the conceptual distance between the programs that we write and their representation in a transformation system.

The implementation of concrete syntax for Stratego is completely generic; all aspects of the embedding of an object syntax in Stratego are user-definable including the quotation and anti-quotation delimiters and the object language itself, of course. Indeed in [40] a general schema is given for extending arbitrary languages with concrete syntax, and in [20] the application of this schema to Prolog is discussed.

5 Transformation Strategies

In the normal interpretation of term rewriting, terms are normalized by exhaustively applying rewrite rules to it and its subterms until no further applications are possible. But because normalizing a term with respect to *all* rules in a specification is not always desirable, and because rewrite systems need not be confluent or terminating, more careful control is often necessary. A common solution is to introduce additional constructors and use them to encode control by means of additional rules which specify where and in what order the original rules are to be applied. The underlying problem is that the rewriting strategy used by rewriting systems is fixed and implicit. In order to provide full control over the application of rewrite rules, Stratego makes the rewriting strategy *explicit* and *programmable* [27,42,41]. Therefore, the specification of a simplifier using innermost normalization in Section 3 required explicit indication of the rules *and* the strategy to be used in this transformation.

5.1 Strategy Combinators

There are many strategies that could potentially be used in program transformations, including exhaustive innermost or outermost normalization, and single pass bottom-up or topdown transformation. Instead of providing built-in implementations for each of these strategies, Stratego provides basic combinators for the composition of strategies.

Such strategies can be defined in a highly generic manner. Strategies can be parameterized with the set of rules, or in general, the transformation, to be applied by the strategy. Thus, the specification of rules can remain separate from the specification of the strategy, and rules can be reused in many different transformations.

Formally, a *strategy* is an algorithm that transforms a term into another term or fails at doing so. Strategies are composed using the following strategy combinators: sequential composition (`s1 ; s2`), determistic choice (`s1 <+ s2`; first try `s1`, only if that fails `s2`), non-deterministic choice (`s1 + s2`; same as `<+`, but the order of trying is not defined[1]), guarded choice (`s1 < s2 + s3`; if `s1` succeeds then commit to `s2` else `s3`), testing (`where(s)`; ignores the transformation achieved), negation (`not(s)`; succeeds if `s` fails), and recursion (`rec x(s)`).

Strategies composed using these combinators can be named using strategy definitions. A strategy definition of the form f(x_1,...,x_n) = s introduces a user-defined operator `f` with n strategy arguments, which can be called by providing it n argument strategies as f(s_1,...,s_n). For example, the definition

```
try(s) = s <+ id
```

defines the combinator `try`, which applies `s` to the current subject term. If that fails it applies `id`, the identity strategy, to the term, which always succeeds with the original term as result. Similarly the `repeat` strategy

```
repeat(s) = try(s; repeat(s))
```

repeats transformation `s` until it fails. Note that strategy definitions do not explicitly mention the term to which they are applied; strategies combine term transformations, i.e., functions from terms to terms, into term transformations.

5.2 Term Traversal

The strategy combinators just described combine strategies which apply transformation rules at the roots of their subject terms. In order to apply a rule to a subterm of a subject term, the term must be traversed. Stratego defines several primitive operators which expose the direct subterms of a constructor application. These can be combined with the operators described above to define a wide variety of complete term traversals.

Congruence operators provide one mechanism for term traversal in Stratego. For each constructor `C` there is a corresponding congruence operator, also denoted `C`. If `C` is an `n`-ary constructor, then the corresponding congruence operator defines the strategy C(s_1,...,s_n). Such a strategy applies only to terms of the form C(t_1,...,t_n). It results in the term C(t'_1,...,t'_n), provided the application of each strategy s_i to each term t_i succeeds with result t'_i. If the application of s_i to t_i fails for any i, then the application of C(s_1,...,s_n) to C(t_1,...,t_n) also fails. Congruences allow the specification of data-type specific traversals such as

[1] Using the `+` operator amounts to declaring that the order in which the argument strategies are tried does not matter and that the compiler is allowed to pick any order. This is typically the case when two rules are mutually exclusive.

```
map(s) = [] + [s | map(s)]
```

which applies a transformation s to the elements of a list. Another example of the use of congruences is the following control-flow strategy [29]

```
control-flow(s) = Assign(id, s) + If(s, id, id) + While(s, id)
```

which applies the argument strategy s, typically a (partial) evaluation strategy, only to selected arguments in order to defer evaluation of the others.

While congruence operators support the definition of traversals that are *specific* to a data type, Stratego also provides combinators for composing *generic traversals*. The operator all(s) applies s to each of the direct subterms t_i of a constructor application C(t_1,...,t_n). It succeeds if and only if the application of s to each direct subterm succeeds. In this case the resulting term is the constructor application C(t'_1,...,t'_n), where each term t'_i is obtained by applying s to t_i. Note that all(s) is the identity on constants, i.e., on constructor applications without children. An example of the use of all is the definition of the strategy bottomup(s):

```
bottomup(s) = all(bottomup(s)); s
```

The strategy expression (all(bottomup(s)); s) specifies that s is first applied recursively to all direct subterms — and thus to all subterms — of the subject term. If that succeeds, then s is applied to the resulting term. This definition of bottomup thus captures the generic notion of a bottom-up traversal over a term. Variations on this one-pass traversal are defined by the following strategies:

```
topdown(s) = s; all(topdown(s))
alltd(s)   = s <+ all(alltd(s))
oncetd(s)  = s <+ one(oncetd(s))
```

Topdown(s) applies s throughout a term starting at the top. Alltd(s) applies s along a frontier of a term. It tries to apply s at the root, if that succeeds the transformation is complete. Otherwise the transformation is applied recursively to all direct subterms. Oncetd(s) is similar, but uses the one combinator to apply a transformation to exactly one direct subterm. One-pass traversals such as shown above can be used in the definition of *fixpoint* traversals such as innermost

```
innermost(s) = bottomup(try(s; innermost(s)))
```

which exhaustively applies a transformation s starting with innermost terms.

6 Transformation Idioms

The explicit control over the rewriting strategy using strategy combinators admits a wide variety of transformation idioms. In this section we discuss several such idioms to illustrate the expressiveness of strategies.

6.1 Cascading Transformations

The basic idiom of program transformation achieved with term rewriting is that of *cascading transformations*. Instead of applying a single complex transformation algorithm to a program, a number of small, independent transformations are applied in combination throughout a program or program unit to achieve the desired effect. Although each individual transformation step achieves little, the cumulative effect can be significant, since each transformation feeds on the results of the ones that came before it.

One common cascading of transformations is accomplished by exhaustively applying rewrite rules to a subject term. In Stratego the definition of a cascading normalization strategy with respect to rules `R1`, ... ,`Rn` can be formalized using an `innermost` strategy:

```
simplify =
  innermost(R1 <+ ... <+ Rn)
```

However, other strategies are possible. For example, the GHC simplifier [30] applies rules in a single traversal over a program tree in which rules are applied both on the way down and on the way up. This is expressed in Stratego by the strategy

```
simplify  =
  downup(repeat(R1 <+ ... <+ Rn))
downup(s) =
  s; all(downup(s)); s
```

6.2 Staged Transformations

In staged computation, transformations are not applied to a subject term all at once, but rather in stages. In each stage, only rules from some particular subset of the entire set of available rules are applied. In the TAMPR program transformation system [5,6] this idiom is called *sequence of normal forms*, since a program tree is transformed in a sequence of steps, each of which performs a normalization with respect to a specified set of rules. In Stratego this idiom can be expressed directly as

```
simplify =
  innermost(A1 <+ ... <+ Ak)
  ; innermost(B1 <+ ... <+ Bl)
  ; ...
  ; innermost(C1 <+ ... <+ Cm)
```

Staged transformations can be applied fruitfully in combination with cascading transformations: a transformation is expresssed as a sequence of stages, where each stage is a cascading transformation. On the other hand, the steps in a staged transformation can use quite different idioms from one another, and can even involve complex monolithic computations. The advantage of separating rules from strategies is particularly compelling in this case of staged transformations. Since rules are defined independently of the particular stages in which they are used, it is easy to reuse them in many different stages.

6.3 'Local' Transformations

In conventional program optimization, transformations are applied throughout a program. In optimizing imperative programs, for example, complex transformations are applied to entire programs [28]. In GHC-style compilation-by-transformation, small transformation steps are applied throughout programs. Local transformation is a style of transformation that is a mixture of these ideas. Instead of applying a complex transformation algorithm to a program we use staged, cascading transformations to accumulate small transformation steps for large effect. However, instead of applying transformations throughout the subject program, we often wish to apply them locally, i.e., only to selected parts of the subject program. This allows us to use transformations rules that would not be beneficial if applied everywhere. A typical strategy achieving such a transformation follows the pattern

```
transformation =
  alltd(
    trigger-transformation
    ; innermost(A1 <+ ... <+ An)
  )
```

The strategy `alltd(s)` descends into a term until a subterm is encountered for which the transformation `s` succeeds. In this case the strategy `trigger-transformation` recognizes a program fragment that should be transformed. Thus, cascading transformations are applied locally to terms for which the transformation is triggered. Of course more sophisticated strategies can be used for finding application locations, as well as for applying the rules locally. Nevertheless, the key observation underlying this idiom remains: Because the transformations to be applied are local, special knowledge about the subject program at the point of application can be used. This allows the application of rules that would not be otherwise applicable.

7 First-Class Pattern Matching

So far it was implied that the basic actions applied by strategies are rewrite rules. However, the distinction between rules and strategies is methodological rather than semantic. Rewrite rules are just syntactic sugar for strategies composed from more basic transformation actions, i.e., matching and building terms, and delimiting the scope of pattern variables [42,41]. Making these actions first-class citizens makes many interesting idioms involving matching directly expressible.

To understand the idea, consider what happens when the following rewrite rule is applied:

```
EvalPlus : Plus(Int(i), Int(j)) -> Int(k) where <add> (i, j) => k
```

First it matches the subject term against the pattern `Plus(Int(i), Int(j))` in the left-hand side. This means that a substitution for the variables `i`, and `j` is sought, that makes the pattern equal to the subject term. If the match fails, the rule fails. If the match succeeds, the condition strategy is evaluated and the result bound to the variable `k`. This

binding is then used to instantiate the right-hand side pattern `Int(k)`. The instantiated term then replaces the original subject term. Furthermore, the rule limits the scope of the variables occurring in the rule. That is, the variables `i`, `j`, and `k` are local to this rule. After the rule is applied the bindings to these variables are invisible again.

Using the primitive actions match (`?pat`), build (`!pat`) and scope ($\{x_1, \ldots, x_n : \texttt{s}\}$), this sequence of events can be expressed as

```
EvalPlus =
  {i,j,k: ?Plus(Int(i), Int(j)); where(!(i,j); add; ?k); !Int(k)}
```

The action `?pat` *matches* the current subject term against the pattern `pat`, binding all its variables. The action `!pat` *builds* the instantiation of the pattern `pat`, using the current bindings of variables in the pattern. The scope $\{x_1, \ldots, x_n : s\}$ delimits the scope of the term variables x_i to the strategy s. In fact, the Stratego compiler desugars rule definitions in this way. In general, a labeled conditional rewrite rule

```
R : p1 -> p2 where s
```

is equivalent to a strategy definition

```
R = {x1,...,xn : ?p1; where(s); !p2}
```

with `x1,...,xn` the free variables of the patterns `p1` and `p2`. Similarly, the strategy application `<s> pat1 => pat2` is desugared to the sequence `!pat1; s; ?pat2`. Many other constructs such as anonymous (unlabeled) rules `\ p1 -> p2 where s \`, application of strategies in build `Int(<add>(i,j))`, contextual rules [35], and many others can be expressed using these basic actions.

7.1 Generic Term Deconstruction

Another generalization of pattern matching is *generic term deconstruction* [36]. Normally patterns are composed of *fixed* constructor applications `C`$(p_1, \ldots, p_n)$, where the constructor name and its arity are fixed. This precludes generic transformations where the specific name of the constructor is irrelevant. Generic traversals provide a way to transform subterms without spelling out the traversal for each constructor. However, with generic traversal the structure of the term remains intact. For analysis problems, an abstract syntax tree should be turned into a value with a different structure. The term deconstruction `pat1#(pat2)` allows accessing the constructor and subterms of a term generically.

As an example, consider the strategy `exp-vars`, which collects from an expression all its variable occurrences:

```
exp-vars =
  \ Var(x) -> [x] \
  <+ \ _#(xs) -> <foldr(![], union, exp-vars)> xs \

foldr(z, c, f) =
  []; z
  <+ \ [h | t] -> <c>(<f>h, <foldr(z, c, f)>t) \
```

If the term is a variable, a singleton list containing the variable name `x` is produced. Otherwise the list of subterms `xs` is obtained using generic term deconstruction (the underscore in the pattern is a wildcard matching with any term); the variables for each subterm are collected recursively; and the union of the resulting lists is produced. Since this is a frequently occuring pattern, the `collect-om` strategy generically defines the notion of collecting outermost occurrences of subterms:

```
exp-vars =
  collect-om(?Var(_))

collect-om(s) =
  s; \ x -> [x] \
  <+ crush(![], union, collect-om(s))

crush(nul, sum, s) :
  _#(xs) -> <foldr(nul, sum, s)> xs
```

Note how `exp-vars` is redefined by passing a pattern match to `collect-om`.

8 Scoped Dynamic Rewrite Rules

Programmable rewriting strategies provide control over the application of rewrite rules. But a limitation of pure rewriting is that rewrite rules are context-free. That is, a rewrite rule can only use information obtained by pattern matching on the subject term or, in the case of conditional rewriting, from the subterms of the subject term. Yet, for many transformations, information from the context of a program fragment is needed. The extension of strategies with *scoped dynamic rewrite rules* [37] makes it possible to access this information.

Unlike standard rewrite rules in Stratego, dynamic rules are generated at run-time, and can access information available from their generation contexts. For example, in the following strategy, the transformation rule `InlineFun` defines the replacement of a function call `f(a*)` by the appropriate instantiation of the body `e1` of its definition:

```
DeclareFun =
  ?fdec@|[ function f(x1*) ta = e1 ]|;
  rules(
    InlineFun :
      |[ f(a*) ]| -> |[ let d* in e2 end ]|
      where <rename>fdec => |[ function f(x2*) ta = e2 ]|
          ; <zip(BindVar)>(x2*, a*) => d*
  )
BindVar :
  (FArg |[ x ta ]|, e) -> |[ var x ta := e ]|
```

The rule `InlineFun` is generated by `DeclareFun` in the context of the *definition* of the function `f`, but applied at the *call sites* `f(a*)`. This is achieved by declaring `InlineFun` in the scope of the match to the function definition `fdec` (second line); the variables bound in that match, i.e., `fdec` and `f`, are inherited by the `InlineFun` rule declared

within the `rules(...)` construct. Thus, the use of `f` in the left-hand side of the rule and `fdec` in the condition refer to inherited bindings to these variables.

Dynamic rules are first-class entities and can be applied as part of a global term traversal. It is possible to restrict the application of dynamic rules to certain parts of subject terms using rule scopes, which limit the live range of rules. For example, `DeclareFun` and `InlineFun` as defined above, could be used in the following simple inlining strategy:

```
inline = {| InlineFun
          : try(DeclareFun)
          ; repeat(InlineFun + Simplify)
          ; all(inline)
          ; repeat(Simplify)
          |}
```

This transformation performs a single traversal over an abstract syntax tree. First inlining functions are generated for all functions encountered by `DeclareFun`, function calls are inlined using `InlineFun`, and expressions are simplified using some set of `Simplify` rules. Then the tree is traversed using `all` with a recursive call of the inliner. Finally, on the way up, the simplification rules are applied again. The dynamic rule scope `{|L : s|}` restricts the scope of a generated rule `L` to the strategy `s`. Of course an actual inliner will be more sophisticated than the strategy shown above; most importantly an inlining criterium should be added to `DeclareFun` and/or `InlineFun` to determine whether a function should be inlined at all. However, the main idea will be the same.

After generic traversal, dynamic rules constituted a key innovation of Stratego that allow many more transformation problems to be addressed with the idiom of strategic rewriting. Other applications of dynamic rules include bound variable renaming [37], dead-code elimination [37], constant-propagation [29] and other data-flow optimizations, instruction selection [9], type checking, partial evaluation, and interpretation [19].

9 Term Annotations

Stratego uses terms to represent the abstract syntax of programs or documents. A term consists of a constructor and a list of argument terms. Sometimes it is useful to record additional information about a term without adapting its structure, i.e., creating a constructor with additional arguments. For this purpose terms can be annotated. Thus, the results of a program analysis can be stored directly in the nodes of the tree.

In Stratego a term always has a list of annotations. This is the empty list if a term does not have any annotations. A term with annotations has the form $\texttt{t}\{a_1, \ldots, a_m\}$, where t is a term as defined in Section 2, the a_i are terms used as annotations, and $m \geq 0$. A term $t\{\}$ with an empty list of annotations is equivalent to t. Since annotations are terms, any transformations defined by rules and strategies can be applied to them.

The annotations of a term can be retrieved in a pattern match and attached in a build. For example the build `!Plus(1, 2){Int}` will create a term `Plus(1, 2)` with

the term `Int` as the only annotation. Naturally, the annotation syntax can also be used in a match: `?Plus(1, 2){Int}`. Note however that this match only accepts `Plus(1, 2)` terms with just one annotation, which should be the empty constructor application `Int`. This match will thus not allow other annotations. Because a rewrite rule is just sugar for a strategy definition, the usage of annotations in rules is just as expected. For example, the rule

```
TypeCheck : Plus(e1{Int}, e2{Int}) -> Plus(e1, e2){Int}
```

checks that the two subterms of the `Plus` have annotation `Int` and then attaches the annotation `Int` to the whole term. Such a rule is typically part of a typechecker which checks type correctness of the expressions in a program *and* annotates them with their types. Similarly many other program analyses can be expressed as program transformation problems. Actual examples in which annotations were used include escaping variables analysis in a compiler for an imperative language, strictness analysis for lazy functional programs, and bound-unbound variables analysis for Stratego itself.

Annotations are useful to store information in trees without changing their signature. Since this information is part of the tree structure it is easily made persistent for exchange with other transformation tools (Section 10). However, annotations also bring their own problems. First of all, transformations are expected to preserve annotations produced by different transformations. This requires that traversals preserve annotations, which is the case for Stratego's traversal operators. However, when transforming a term it is difficult to preserve the annotations on the original term since this should be done according to the semantics of the annotations. Secondly, it is no longer straightforward to determine the equality relation between terms. Equality can be computed with or without (certain) annotations. These issues are inherent in any annotation framework and preclude smooth integration of annotations with the other features discussed; further research is needed in this area.

10 Transformation Tools and Systems

A transformation defined using rewrite rules and strategies needs to be applied to actual programs. That is, it needs to read an input program, transform it, and write an output program. In addition, it needs to take care of command-line options such as the level of optimization. The Stratego Standard Library provides facilities for turning a transformation on terms into a transformation on files containing programs or intermediate representations of programs.

ATerm Exchange Format The terms Stratego uses internally correspond exactly with terms in the ATerm exchange format [7]. The Stratego run-time system is based on the ATerm Library which provides support for internal term representation as well as their persistent representation in files, making it trivial to provide input and output for terms in Stratego, and to exchange terms between transformation tools. Thus, transformation systems can be divided into small, reusable tools

Foreign Function Interface Stratego has a foreign function interface which makes it possible to call C functions from Stratego functions. The operator prim($f, t_1, ..., t_n$) calls the C function f with term arguments t_i. Via this mechanism functionality such as arithmetic, string manipulation, hash tables, I/O, and process control are incorporated in the library without having to include them as built-ins in the language. For example, the definition

```
read-from-stream =
  ?Stream(stream)
  ; prim("SSL_read_term_from_stream", stream)
```

introduces an alias for a primitive reading a term from an input stream. In fact several language features started their live as a collection of primitives before being elevated to the level of language construct; examples are dynamic rules and annotations.

Wrapping Transformations in Tools To make a transformation into a tool, the Stratego Standard Library provides abstractions that take care of all I/O issues. The following example illustrates how a `simplify` strategy is turned into a tool:

```
module simplify
imports lib Tiger-Simplify
strategies
  main = io-wrap(simplify-options, simplify)
  simplify-options =
    ArgOption("-O", where(<set-config> ("-O", <id>)),
              !"-O n      Set optimization level (1 default)")
```

The `main` strategy represents the tool. It is defined using the `io-wrap` strategy, which takes as arguments the non-default command-line options and the strategy to apply. The wrapper strategy parses the command-line options, providing a standardized tool interface with options such as `-i` for the input and `-o` for the output file. Furthermore, it reads the input term, applies the transformation to it, and writes the resulting term to output. Thus, all I/O complexities are hidden from the programmer.

Tool Collections Stratego's usage of the ATerm exchange format and its support for interface implementation makes it very easy to make small reusable tools. In the spirit of the Unix pipes and filters model, these tools can be mixed and matched in many different transformation systems. However, instead of transforming text files, these tools transform structured data. This approach has enabled and encouraged the construction of a large library of reusable tools. The core library is the XT bundle of transformation tools [17], which provides some 100 more or less generic tools useful in the construction and generation of transformation systems. This includes the implementation of pretty-printing formatters of the generic pretty-printing package GPP [14], coupling of Stratego transformation components with SDF parsers, tools for parsing and pretty-printing, and generators for deriving components of transformation systems from a syntax definition. A collection of application-specific transformation components based on the XT library is emerging (see Section 12).

```
io-tiger-pe =
  xtc-io-wrap(tiger-pe-options,
    parse-tiger
  ; tiger-desugar
  ; tiger-partial-eval
  ; if-switch(!"elim-dead", tiger-elim-dead)
  ; if-switch(!"ensugar",   tiger-ensugar)
  ; if-switch(!"pp",        pp-tiger)
  )
tiger-partial-eval =
  xtc-transform(!"Tiger-Partial-Eval", pass-verbose)
...
```

Fig. 3. Example transformation tool composition.

Transformation Tool Composition A transformation system implements a complete source-to-source transformation, while tools just implement an aspect. Construction of complete transformation systems requires the composition of transformation tools. For a long time composition of transformation tools in XT was done using conventional means such as makefiles and shell scripts. However, these turn out to cause problems with determining the installation location of a tool requiring extensive configuration, transforming terms at the level of the composition, and poor abstraction and control facilities.

The XTC model for transformation tool composition was designed in order to alleviate these problems. Central in the XTC model is a repository which registers the locations of specific versions of tools. This allows a much more fine-grained search than is possible with directory-based searches. A library of abstractions implemented in Stratego supports transparently calling tools. Using the library a tool can be applied just like a basic transformation step. All the control facilities of Stratego can be used in their composition. Figure 3 illustrates the use of XTC in the composition of a partial evaluator from transformation components, corresponding to the right branch of the data-flow diagram in Figure 1.

11 Stratego/XT in Practice

The Stratego language is implemented by means of a compiler that translates Stratego programs to C programs. Generated programs depend on the ATerm library and a small Stratego-specific, run-time system. The Stratego Standard Library provides a large number of generic and data-type specific reusable rules and strategies. The compiler and the library, as well as number of other packages from the XT collection are bundled in the Stratego/XT distribution, which is available from www.stratego-language.org [43] under the LGPL license. The website also provides user documentation, pointers to publications and applications, and mailinglists for users and developers.

12 Applications

The original application area of Stratego is the specification of optimizers, in particular for functional compilers [42]. Since then, Stratego has been applied in many areas of language processing:

- Compilers: typechecking, translation, desugaring, instruction selection
- Optimization: data-flow optimizations, vectorization, ghc-style simplification, deforestation, domain-specific optimization, partial evaluation, specialization of dynamic typing
- Program generators: pretty-printer and signature generation from syntax definitions, application generation from DSLs, language extension preprocessors
- Program migration: grammar conversion
- Program understanding: documentation generation
- Document generation and transformation: XML processing, web services

The rest of this section gives an overview of applications categorized by the type of the source language.

Functional Languages Simplification in the style of the Glasgow Haskell Compiler [30] was the first target application for Stratego [42], and has been further explored for the language Mondrian and recently in an optimizer for the Helium compiler. Other optimizations for functional programs include an implemenation of the warm fusion algorithm for deforestation of lazy functional programs [23], and a partial evaluator for a subset of Scheme (similix) [32].

Imperative Languages Tiger is the example language of Andrew Appel's text book on compiler construction. It has proven a fruitful basis for experimentation with all kinds of transformations and for use in teaching [43]. Results include techniques for building interpreters [19], implementing instruction selection (maximal munch and burg-style dynamic programming) [9], and specifying optimizations such as function inlining [37] and constant propagation [29].

These techniques are being applied to real imperative languages. CodeBoost [2] is a transformation framework for the domain-specific optimization of C++ programs developed for the optimization of programs written in the Sophus style. Several application generators have been developed for the generation of Java and C++ programs.

Transformation Tools The Stratego compiler is bootstrapped, i.e., implemented in Stratego, and includes desugaring, implementation of concrete syntax, semantic checks, pattern match compilation, translation to C, and various optimizations [24].

Stratego is used as the implementation language for numerous meta-tools in the XT bundle of program transformation tools [17]. This includes the implementation of pretty-printing formatters of the generic pretty-printing package GPP [14] and the coupling of Stratego transformation components with SDF parsers.

Other Languages In a documentation generator for SDL [16], Stratego was used to extract transition tables from SDL specifications.

XML and Meta-Data for Software Deployment The structured representation of data, their easy manipulation and external representation, makes Stratego an atractive language for processing XML documents and other structured data formats. For example, the Autobundle system [15] computes compositions (bundles) of source packages by analyzing the dependencies in package descriptions represented as terms and generates an online package base from such descriptions. Application in other areas of software deployment is underway. The generation of XHTML and other XML documents is also well supported with concrete syntax for XML in Stratego and used for example in xDoc, a documentation generator for Stratego and other languages.

13 Related Work

Term rewriting [33] is a long established branch of theoretical computer science. Several systems for program transformation are based on term rewriting. The motivation for and the design of Stratego were directly influenced by the ASF+SDF and ELAN languages. The algebraic specification formalism ASF+SDF [18] is based on pure rewriting with concrete syntax without strategies. Recently traversal functions were added to ASF+SDF to reduce the overhead of traversal control [8]. The ELAN system [4] first introduced the ideas of user-definable rewriting strategies in term rewriting systems. However, generic traversal is not provided in ELAN. The first ideas about programmable rewriting strategies with generic term traversal were developed with ASF+SDF [27]. These ideas were further developed in the design of Stratego [42,41]. Also the generalization of concrete syntax [40], first-class pattern matching [35], generic term deconstruction [36], scoped dynamic rewrite rules [37], annotations, and the XTC component composition model are contributions of Stratego/XT. An earlier paper [38] gives a short overview of version 0.5 of the Stratego language and system, before the addition of concrete syntax, dynamic rules, and XTC.

Other systems based on term rewriting include TAMPR [5,6] and Maude [11,10]. There are also a large number of transformation systems not based (directly) on term rewriting, including TXL [12] and JTS [3]. A more thorough discussion of the commonalities between Stratego and other transformation systems is beyond the scope of this paper. The papers about individual language concepts cited throughout this paper discuss related mechanisms in other languages. In addition, several papers survey aspects of strategies and related mechanisms in programming languages. A survey of strategies in program transformation systems is presented in [39], introducing the motivation for programmable strategies and discussing a number of systems with (some) support for definition of strategies. The essential ingredients of the paradigm of 'strategic programming' and their incarnations in other paradigms, such as object-oriented and functional programming, are discussed in [25]. A comparison of strategic programming with adaptive programming is presented in [26]. Finally, the program transformation wiki [1] lists a large number of transformation systems.

14 Conclusion

This paper has presented a broad overview of the concepts and applications of the Stratego/XT framework, a language and toolset supporting the high-level implementation of program transformation systems. The framework is applicable to many kinds of transformations, including compilation, generation, analysis, and migration. The framework supports all aspects of program transformation, from the specification of transformation rules, their composition using strategies, to the encapsulation of strategies in tools, and composition of tools into systems.

An important design guideline in Stratego/XT is separation of concerns to achieve reuse at all levels of abstractions. Thus, the separation of rules and strategies allows the specification of rules separately from the strategy that applies them and a generic strategy can be instantiated with different rules. Similarly a certain strategy can be used in different tools, and a tool can be used in different transformation systems. This principle supports reuse of transformations at different levels of granularity.

Another design guideline is that separation of concerns should not draw artificial boundaries. Thus, there is no strict separation between abstraction levels. Rather the distinctions between these levels is methodological and idiomatic rather than semantic. For instance, a rule is really an idiom for a certain type of strategy. Thus, rules and strategies can be interchanged. Similarly, XTC applies strategic control to tools and allows calling an external tool as though it were a rule. In general, one can mix rules, strategies, tools, and systems as is appropriate for the system under consideration, thus making transformations compositional in practice. Of course one has to consider trade-offs when doing this, e.g., the overhead of calling an external process versus the reuse obtained, but there is no technical objection.

Finally, Stratego/XT is designed and developed as an open language and system. The intial language based on rewriting of abstract syntax trees under the control of strategies has been extended with first-class pattern matching, dynamic rules, concrete syntax, and a tool composition model, in order to address new classes of problems. The library has accumulated many generic transformation solutions. Also the compiler is component-based, and more and more aspects are under the control of the programmer.

Certain aspects of the language could have been developed as a library in a general purpose language. Such an approach, although interesting in its own right, meets with the syntactic and semantic limitations of the host language. Building a domain-specific language for the domain of program transformation has been fruitful. First of all, the constructs that matter can be provided without (syntactic) overhead to the programmer. The separation of concerns (e.g., rules as separately definable entities) that is provided in Stratego is hard to achieve in general purpose languages. Furthermore, the use of the ATerm library with its maximal sharing (hash consing) term model and easy persistence provides a distinct run-time system not available in other languages. Rather than struggling with a host language, the design of Stratego has been guided by the needs of the *transformation* domain, striving to express transformations in a natural way.

Symbolic manipulation and generation of programs is increasingly important in software engineering, and Stratego/XT is an expressive framework for its implementation. The ideas developed in the project can also be useful in other settings. For example, the approach to generic traversal has been transposed to functional, object-oriented,

and logic programming [25]. This paper describes Stratego/XT at release 0.9, which is not the final one. There is a host of ideas for improving and extending the language, compiler, library, and support packages, and for new applications. For an overview, see www.stratego-language.org.

Acknowledgments Stratego and XT have been developed with contributions by many people. The initial set of strategy combinators was designed with Bas Luttik. The first prototype language design and compiler was developed with Zino Benaissa and Andrew Tolmach. The run-time system of Stratego is based on the ATerm Library developed at the University of Amsterdam by Pieter Olivier and Hayco de Jong. SDF is maintained and further developed at CWI by Mark van den Brand and Jurgen Vinju. The XT bundle was set up and developed together with Merijn de Jonge and Joost Visser. Martin Bravenboer has played an important role in modernizing XT, collaborated in the development of XTC, and contributed several packages in the area of XML and Java processing. Eelco Dolstra has been very resourceful when it came to compilation and porting issues. The approach to data-flow optimization was developed with Karina Olmos. Rob Vermaas developed the documentation software for Stratego. Many others developed applications or otherwise provided valuable feedback including Otto Skrove Bagge, Arne de Bruijn, Karl Trygve Kalleberg, Dick Kieburtz, Patricia Johann, Lennart Swart, Hedzer Westra, and Jonne van Wijngaarden. Finally, the anonymous referees provided useful feedback on an earlier version of this paper.

References

1. `http://www.program-transformation.org`.
2. O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 65–74, Amsterdam, September 2003. IEEE Computer Society Press.
3. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 2–5 1998. IEEE.
4. P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. *ELAN: User Manual*. Loria, Nancy, France, v3.4 edition, January 27 2000.
5. J. M. Boyle. Abstract programming and program transformation—An approach to reusing programs. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume 1, pages 361–413. ACM Press, 1989.
6. J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transforming system: Simplifying the development of numerical software. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 353–372. Birkhäuser, 1997.
7. 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.
8. M. G. J. van den Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 12(2):152–190, April 2003.
9. M. Bravenboer and E. Visser. Rewriting strategies for instruction selection. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 237–251, Copenhagen, Denmark, July 2002. Springer-Verlag.

10. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
11. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. 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*, pages 65–89, Asilomar, Pacific Grove, CA, September 1996. Elsevier.
12. J. R. Cordy, I. H. Carmichael, and R. Halliday. *The TXL Programming Language, Version 8*, April 1995.
13. K. Czarnecki and U. W. Eisenecker. *Generative Programming*. Addison Wesley, 2000.
14. M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.
15. M. de Jonge. Source tree composition. In C. Gacek, editor, *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *LNCS*, pages 17–32. Springer-Verlag, April 2002.
16. M. de Jonge and R. Monajemi. Cost-effective maintenance tools for proprietary languages. In *Proceedings: International Conference on Software Maintenance (ICSM 2001)*, pages 240–249. IEEE Computer Society Press, November 2001.
17. M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand and D. Perigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, April 2001.
18. 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.
19. E. Dolstra and E. Visser. Building interpreters with rewriting strategies. In M. G. J. van den Brand and R. Lämmel, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, volume 65/3 of *Electronic Notes in Theoretical Computer Science*, Grenoble, France, April 2002. Elsevier Science Publishers.
20. B. Fischer and E. Visser. Retrofitting the AutoBayes program synthesis system with concrete object syntax. In Lengauer et al., editors, *Domain-Specific Program Generation*, Lecture Notes in Computer Science. Spinger-Verlag, June 2004. (to appear).
21. M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
22. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
23. 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*, 29(1–4):1–34, 2000.
24. P. Johann and E. Visser. Fusing logic and control with local transformations: An example optimization. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.
25. R. Lämmel, E. Visser, and J. Visser. The essence of strategic programming, October 2002. (Draft).
26. R. Lämmel, E. Visser, and J. Visser. Strategic Programming Meets Adaptive Programming. In *Proceedings of Aspect-Oriented Software Development (AOSD'03)*, pages 168–177, Boston, USA, March 2003. ACM Press.
27. 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.
28. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
29. K. Olmos and E. Visser. Strategies for source-to-source constant propagation. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies (WRS'02)*, volume 70 of *Electronic Notes in Theoretical Computer Science*, page 20, Copenhagen, Denmark, July 2002. Elsevier Science Publishers.
30. S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.
31. R. J. Rodger. Jostraca: a template engine for generative programming. *European Conference for Object-Oriented Programming*, 2002.
32. L. Swart. Partial evaluation using rewrite rules. A specification of a partial evaluator for Similix in Stratego. Master's thesis, Utrecht University, Utrecht, The Netherlands, August 2002.
33. Terese. *Term Rewriting Systems*. Cambridge University Press, March 2003.
34. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
35. 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.
36. E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming (WGP'00)*, Ponte de Lima, Portugal, July 2000. Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht.
37. E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 2001.
38. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.
39. E. Visser. A survey of rewriting strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.
40. E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.
41. 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)*, volume 15 of *Electronic Notes in Theoretical Computer Science*, Pont-à-Mousson, France, September 1998. Elsevier Science Publishers.
42. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.
43. `http://www.stratego-language.org`.