Stratego/XT 0.16: Components for Transformation Systems

Martin Bravenboer
Department of Information and Computing Sciences, Utrecht University
The Netherlands
martin@cs.uu.nl

Karl Trygve Kalleberg
Department of Informatics
University of Bergen, Norway
karltk@ii.uib.no

Rob Vermaas
Machina, Utrecht
The Netherlands
rob@levellers.nl

Eelco Visser
Department of Information and Computing Sciences, Utrecht University
The Netherlands
visser@acm.org

Abstract

Stratego/XT is a language and toolset for program transformation. The Stratego language provides rewrite rules for expressing basic transformations, programmable rewriting strategies for controlling the application of rules, concrete syntax for expressing the patterns of rules in the syntax of the object language, and dynamic rewrite rules for expressing context-sensitive transformations, thus supporting the development of transformation components at a high level of abstraction. The XT toolset offers a collection of flexible, reusable transformation components, as well as declarative languages for deriving new components. Complete program transformation systems are composed from these components. In this paper we give an overview of Stratego/XT 0.16.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors

General Terms Languages, Design

Keywords Transformation Systems, Program Transformation, Transformation Components, Term Rewriting, Rewriting Strategies, Compilers, Program Optimization, Program Analysis, Parsers, Pretty-printers

1. Introduction

Stratego/XT is a development environment for creating stand-alone transformation systems. It combines *Stratego*, a language for implementing transformations based on the paradigm of programmable rewriting strategies, with *XT*, a collection of reusable components and tools for the development of transformation systems. In general, Stratego/XT is intended for the *analysis*, *manipulation* and *generation* of programs, though its features make it useful for transforming any structured documents. In practice, Stratego/XT has been used to build many types of transformation systems including compilers, interpreters, static analyzers, domain-specific optimizers, code generators, source code refactorers, documentation generators, and document transformers. These systems involved numerous types of transformations, including desugaring of syntactic abstractions; assimilation of language embeddings [7]; bound variable renaming; optimizations, such as function inlining; data-flow transformations such as constant propagation, copy propagation, common-subexpression elimination, and partial evaluation [4, 14]; instruction selection [6]; and several analyses including type checking [5] and escaping variables analysis.

In this paper, we give an overview of Stratego/XT. In Section 2 we describe the transformation infrastructure provided by the XT toolset. In Section 3 we sketch the technical foundations of the Stratego language. In Section 4 we outline the experience with using Stratego/XT in concrete projects. In Section 5 we describe the availability of software, documentation, and support. Along the way we refer to previous publications for further information about implemenation aspects and applications of Stratego/XT. These publications also provide discussions of the relation to other systems.

2. The XT Transformation Tools

XT [10] is a collection of generic, reusable tools serving two purposes. First, it provides several domain-specific languages designed for the development of language-specific transformation components. Second, XT contains ready-made, generic components. The existing and generated components all fit together in a flexible and scalable component model. XT relies on the SDF parsing technology, providing a parser generator and a scannerless, generalized-LR parser (SGLR) [15]. XT extends the SDF distribution with tools for unit testing grammars; a well-formedness checker for ASTs; a pretty-print generator based on the Box layout language [8]; and XML conversion tools.

The syntax definition language SDF is central to XT. It is used to specify the syntax of programming languages in a declarative way. Different code generators take the syntax definition as input, deriving several artifacts: a parser which directly constructs an AST from a source code file, a format checker for such ASTs (used to determine the correctness of subsequent transformations on the AST), a Stratego signature (data declaration) for the AST, and a pretty-printer for turning ASTs back into text.

PEPM '06 January 9–10, 2006, Charleston, South Carolina, USA.

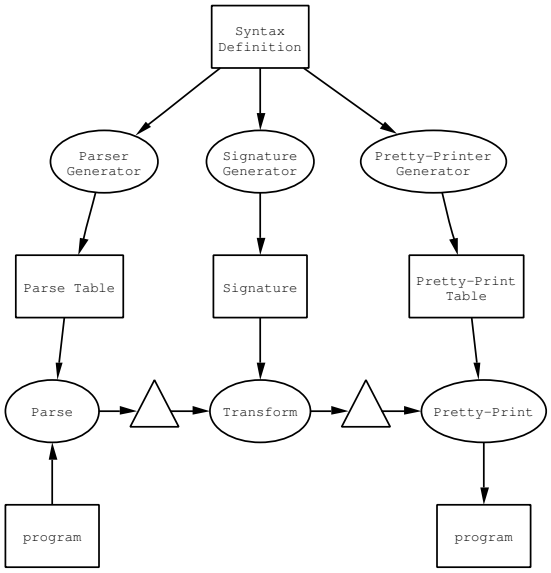

Figure 1. Transformation infrastructure

Figure 1 depicts the architecture of a transformation system and shows how XT facilitates the development of its components. The `Parser` and `Pretty-Printer` in the pipeline are entirely derived from the SDF definition. The SDF grammar enjoys its central position in this picture for a good reason. Unlike most other grammar formalisms, it is highly *modular* and *declarative*. This allows the formalism to easily scale to large language declarations, but more interestingly, it also allows for easy language composition and embedding, see Section 4 for details. The `Transform` component (which may be a series of components), is written in the Stratego language, described next.

Components are connected using the Annotated Term Format (ATerms) [2], an exchange format that supports easy persistence of terms used to represent programs.

3. The Stratego Language

In transformation systems built with Stratego/XT, transformation components are implemented using the Stratego language [20, 18, 4]. Stratego provides rewrite rules for expressing basic transformations, programmable rewriting strategies for controlling the application of rules, concrete syntax for expressing the patterns of rules in the syntax of the object language, and dynamic rewrite rules for expressing context-sensitive transformations.

Terms Stratego programs transform *terms* that are essentially of the form $c(t_1, ..., t_n)$, i.e. the application of a constructor c to zero or more terms t_i. Terms are equivalent to trees and are used to represent parse or abstract syntax trees of programs, or any other structured documents. The terms used internally are equivalent to the ATerms used for exchange between tools. Thus, Stratego programmers directly manipulate ATerms.

Stratego is agnostic to the producers and consumers of terms, i.e. Stratego is not concerned with turning programs into terms (parsing), or vice versa (unparsing). This means that a Stratego program can be connected with any parsers and unparsers, not just with parsers/unparsers produced from SDF definitions. This allows existing front-ends to be used in combination with Stratego, as long as the front-end is adapted to produce ATerms.

Rewrite Rules *Rewrite rules* are the basic units of transformation. A rewrite rule has the form $R : p_1 \rightarrow p_2$, where R is the name of the rule, p_1 the *left-hand side pattern* of the rule and p_2 the *right-hand side pattern*. A pattern is a term with variables. Applying a rule R to a term t, entails matching p_1 against t, binding the variables in the pattern. If they match, the rule replaces t with the instantiation of the right-hand side p_2, replacing its variables with the terms found during matching. When matching fails, the application of a rule fails as well.

Rewriting Strategies Traditional *term rewriting* is the exhaustive application of a set of rewrite rules to a term until no more rules apply. This procedure is usually not adequate for program transformation, however. One rule may be the inverse of another, leading to non-termination, or different rule application orders may give different results (non-confluence). Stratego sidesteps these issues by allowing the programmer to declare the order of application using *programmable rewriting strategies* [20]. Strategies are composed from rules and other strategies using combinators. For example, the left-choice combinator s_1 `<+` s_2 first tries to apply strategy s_1 and if that fails, applies strategy s_2. Primitive strategies include `id` which always succeeds and `fail`, which always fails. Stratego provides combinators for composing *generic traversals*, which are used to traverse a term, controlling where in the term a transformation should be applied. An example of such a generic traversal is `topdown(`s`)`, which traverses an entire term and applies strategy s in pre-order. If s fails at any point, so does `topdown`. `topdown(s <+ id)` will traverse the tree, apply `s` wherever possible and ignore any failures. Stratego is not the only language which uses the concept of strategies. See [19] for a survey of strategies in program transformation.

Dynamic Rules Rewrite rules are *context-free*, i.e. only have access to the term to which they are applied. To express *context-sensitive* transformations, Stratego has introduced *dynamic rewrite rules* [4, 14], which allow the definition of rewrite rules at run-time. Such rules can inherit information from the context in which they are defined and propagate this to the location where they are applied.

Concrete Syntax Finally, Stratego supports the use of *concrete syntax* [17] in the patterns of rewrite rules. That is, rather than expressing abstract syntax tree patterns using nested constructor applications, one can use the concrete syntax of the object language. For example, a pattern, matching an assignment of an expression to a variable, may be expressed as `Assign(Var(`x`), `e`)` using terms, and may be written as `⟦ `x` := `e` ⟧` using concrete syntax.

Example The features of Stratego are illustrated in **Figure 2**, which defines a flow-sensitive, intraprocedural constant propagation transformation for an imperative language with assignments and structured control flow, as illustrated by the following transformation:

```
(a := 1;
 if foo()
 then (b := a + 1)
 else (b := 2; a := 3);
 b := a + b)
```

⇒

```
(a := 1;
 if foo()
 then (b := 2)
 else (b := 2; a := 3);
 b := a + 2)
```

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

EvalIf :
  |[ if 0 then e1 else e2 ]| -> |[ e2 ]|

propconst =
  PropConst
  <+ propconst-assign
  <+ propconst-if
  <+ propconst-while
  <+ all(propconst)
    ; try(EvalBinOp <+ EvalIf)

propconst-assign =
  |[ x := <propconst => e> ]|
  ; if <is-value> e
    then rules( PropConst :  |[ x ]| -> |[ e ]| )
    else rules( PropConst :- |[ x ]| ) end

propconst-if =
  |[ if <propconst> then <id> else <id> ]|
  ; (EvalIf; propconst
     <+ (|[ if <id> then <propconst> else <id> ]|
         /PropConst\
              |[ if <id> then <id> else <propconst> ]|))

propconst-while =
  |[ while <id> do <id> ]|
  ; (|[ while <propconst> do <id> ]|; EvalWhile
     <+ /PropConst\*
        |[ while <propconst> do <propconst> ]|)
```

Figure 2. Flow-sensitive constant propagation

The rewrite rules `EvalBinOp` and `EvalIf` express constant folding. Typically there would be many more constant folding rules for the constructs of a language.

The `propconst` strategy traverses a function body in a bottom-up manner using the generic traversal combinator `all`, applying constant folding rules after transforming subterms. That is, the expression `all(`s`)` denotes a *one-level traversal* that applies the strategy s to each direct subterm of the subject term. Thus, `all(propconst)` recursively applies the `propconst` strategy to the subterms. The other elements of the choice handle special cases, where a uniform traversal is not appropriate.

The `PropConst` rule is defined dynamically during the transformation and replaces variable occurrences with their constant value.

The `propconst-assign` strategy matches assignment statements, transforming the right-hand side with a recursive invocation of `propconst`, but not the left-hand side; replacing the left-hand side variable with a constant would not be correct. Next the `propconst-assign` strategy inspects the result `e` of transforming the right-hand side. If it is a constant value, the `PropConst` rule is defined to rewrite occurrences of the variable `x` to the constant right-hand side of the assignment. In case the expression is not a constant, the rule is undefined, to prevent propagation of a value previously associated with the variable.

The `propconst-if` and `propconst-while` strategies define flow-sensitive propagation through control flow constructs [4, 14]. The s_1`/PropConst\`s_2 operator applies the strategies s_1 and s_2 strategies sequentially, for each using the same set of dynamic rules for `PropConst`. Afterwards it takes the *intersection* of the rule-sets resulting from each invocation. Thus, only those rules consistent with both branches are kept. Similarly, `/PropConst\*`s performs a fixed point iteration until the `PropConst` rule-set is stable.

4. Experience

Stratego/XT is being applied in a number of research and industrial projects. The experience from these projects has been influential on the design and implementation of Stratego/XT.

Stratego/XT Stratego and the XT tools are bootstrapped, that is, they are used in their own implementation. The 0.16 release of Stratego/XT counts well over 50K lines of Stratego code. Additionally, many extensions and utilities have been built using Stratego/XT, including the Stratego Shell, an interactive interpreter for Stratego; xDoc, a Javadoc-like source code documentation system for Stratego code; and Aspect Stratego [12], a language extension to Stratego which adds pointcuts and advice.

Java Besides generic tools, it is important to have ready-made components for specific programming languages. For Java, we have developed a modular Java 1.5 syntax definition, a high-quality pretty-printer, a disambiguation phase, an extensible type checker (including generics), an extensive reflection library for use in Stratego, and tools for reading and writing Java bytecode to terms. These tools are used in the implementation of *language embeddings*, a recently added application area of Stratego/XT [7, 3]. Extensions of Java with domain-specific languages for user-interfaces and regular expressions are available. JavaJava [5] is an advanced code generation tool, based on the extensible type checker and the GLR parsing technology used in Stratego/XT.

Also, we have developed an AspectJ grammar, wich is a modular extension of the Java syntax definition. Thus, the AspectJ grammar only defines the syntax extensions to Java provided by AspectJ, and is programmed against the public interface of the Java grammar. This sort of language composition is possible because of the scannerless, GLR nature of the SGLR parser.

C/C++ The Transformers project at LRDE, EPITA, France has produced a disambiguating front-end for C99, and is doing the same for C++ 2003. Disambiguation of both languages requires semantic analysis, which is implemented in an attribute grammar extension to SDF, which in turn is implemented in Stratego. CodeBoost [1] is a source-to-source optimizer for C++ code, developed at the University of Bergen, Norway, supporting domain-specific optimization of numerical software. Codeboost includes a semantic analyser for substantial parts of C++, written entirely in Stratego. Proteus [22] is a C/C++ transformation framework, based on Stratego, constructed at Lucent, USA. It compiles a high-level transformation language (YATL) to Stratego. Compared to CodeBoost, it relies on a different C++ parser, retains layout and deals gracefully with pre-processor directives.

Miscellaneous Stratego/XT has been used to build several other (experimental) compilers and front-ends. OctaveC is a compiler for Octave, a clone of Matlab. It includes loop vectorization, and partial evaluation [13]. Tiger in Stratego is a demonstration compiler that includes all aspects of compilation, from type checking, via optimizations, to instruction selection. Prolog Tools provides a language front-end and DSL embedding for Prolog [9]. Spoofax [11] is an Eclipse plugin that provides content-aware, syntax highlighting editors for SDF and Stratego. Our BibTEX transformation tools provide extractions and refactorings on BibTEX bibliography files.

5. Availability

Software Stratego/XT is an open source project, distributed under the GNU LGPL license. This license allows closed source transformation systems based on Stratego/XT. The Subversion source code repository is publicly readable. From this repository integration releases are generated for a number of different platforms after every commit by a continuous build system. This buildfarm helps

with testing portability and backwards compatibility. Additionally, many satellite projects, such as the Java front-end, are continuously tested against the integration releases. For each major release, source drops and binaries for most popular platforms and operating systems are produced.

Documentation The primary source of documentation is the manual. It offers an extensive introduction of the XT architecture, and also a complete Stratego tutorial. The tutorial includes several program transformation examples shown on a small, imperative language. The reference material includes complete manual pages for all the XT command-line tools, and online API documentation of the library, which is also available for download.

Support The Stratego/XT website [21] contains pointers to mailing lists for users and developers, a wiki, release pages, documentation and issue tracking. Additionally, the developers are available for questions or chat on IRC.

6. Discussion

Previous Work Compared to earlier publications about Stratego/XT [10, 16, 18] we have gained new experience with the development of transformation systems for Java, C, Octave, and BibTEX. Motivated by these projects new language constructs such as dynamic rules and concrete object syntax have matured. Stratego/XT now also provides new tools for testing, validating, and debugging, to help in developing reliable transformation systems.

Usability Being a research project, Stratego/XT has enjoyed rapid evolution but lagging and incomplete documentatiton. In the last few years, the system has reached a new level of maturity and stability, allowing us to document core parts of the system. Creating complete documentation is a huge effort, given the size of system, but we are working on improving the situation for our developers, with manual pages for all tools, improved API documentation, and an extensive manual with examples. Still, users need to invest time learning the foundations, the Stratego language and the XT tools. We are continually working on making the initial learning curve small.

Reuse Stratego/XT promotes reuse at all levels of granularity [18]. First, the focus on *transformation components* strongly promotes reuse of large-grained components. In many cases, users of Stratego/XT do not start with the development of a parser, but can immediately get started with the actual transformation. Stratego/XT has a varied selection of actively developed front-ends (Section 4). Second, the focus on domain-specific languages for different phases of a transformation system is a substantial time saver. In this way, implementations are more abstract, easier to maintain, and easier to read. Third, the extensive Stratego library, with its generic traversals, generic transformations for scoping, control- and data-flow and many other convenience functions for program transformation allows developers to write their transformations concisely.

7. Conclusion

Stratego/XT has considerably matured in the last few years of intensive development and research. We have been successful in exploring the implementation of individual transformations, and the range of transformations that we know how to encode effectively and elegantly grows. Along the way we keep discovering better idioms and abstractions for implementing transformations.

The goal of Stratego/XT is to support a wide-range of transformations and to provide a new level of abstraction for the implementation of transformation systems by third parties. Experience shows that external users can succesfully build non-trivial transformation systems on top of Stratego/XT.

8. Acknowledgements

Many people have contributed to the development of Stratego/XT over the years. Hayco de Jong, Jurgen Vinju, and Mark van den Brand at CWI do a great job as maintainers of the ATerm library and SDF/SGLR, projects which are all fundamental parts of Stratego/XT. Bas Luttik co-invented generic traversal strategies with Eelco Visser; Zino Benaissa and Andrew Tolmach were involved in the design of the very first version of Stratego; Merijn de Jonge and Joost Visser co-developed the first version of the XT toolset; Karina Olmos and Arthur van Dam were involved in the design and implementation of dynamic rules; The redesign of dynamic rules reported in those papers was triggered by Ganesh Sittampalam at the Stratego User Days in 2004. Patricia Johann was involved in the design and correctness proof of innermost fusion; Anya Bagge developed CodeBoost, one of the first big applications of Stratego/XT. Eelco Dolstra has been an indispensable resource for tracing bugs in the back-end, and has also provided the Nix build-farm that plays a crucial role in our development and deployment process. Jan Heering and Magne Haveraaen provided moral support and employed Stratego in the Saga/CodeBoost project at the Univerity of Bergen. Martin Bravenboer has acted as the release manager for Stratego/XT in the later years, and made many other contributions, including the format checking tools, the Stratego Shell, and the Dryad Java compiler. Rob Vermaas developed the xDoc documentation system and has been an active developer of Stratego/XT and OctaveC for several years. Karl Trygve Kalleberg helped develop CodeBoost, wrote the AspectStratego language and the Spoofax editor. He actively helps maintain Stratego/XT.

Stratego/XT is the brainchild of Eelco Visser, who has been intimately involved in all extensions and additions to the language and toolset since its inception.

Finally, the feedback from our users has been a constant source for improvements. The Transformers group at Epita led by Akim Demaille are early adopters of our improvements and extensions, have contributed many bugreports and improvements, and invest a lot of effort in the development of C/C++ transformation system based on Stratego/XT. Daniel Waddington's Proteus group at Lucent Bell Labs has shown us that people can build complex transformation systems with Stratego/XT without our involvement, even before we had proper documentation. Several generations of students in the courses on Software Generation, High-Performance Compilers, and Program Transformation at Universiteit Utrecht have provided valuable feedback and sometimes innovations. Their suffering through imperfect implementation and lack of documentation have not been in vain. About 15 of those students ended up doing a master's thesis project related to Stratego/XT, contributing to the system and research.

We would also like to thank the anonymous reviewers for comments on early versions of this article.

References

[1] 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, *3rd IEEE Itl Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 65–74, Amsterdam, The Netherlands, Sep 2003. IEEE Comp. Soc. Press.

[2] M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[3] M. Bravenboer, R. de Groot, and E. Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, Braga, Portugal, July 2005.

[4] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69:1–56, 2005.

[5] M. Bravenboer, R. Vermaas, J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In R. Glück and M. Lowry, editors, *Proc. of Fourth Itl Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *LNCS*, pages 157–172, Tallin, Estonia, Sep 2005. Springer.

[6] M. Bravenboer and E. Visser. Rewriting strategies for instruction selection. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *LNCS*, pages 237–251, Copenhagen, Denmark, July 2002. Springer.

[7] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proc. the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[8] 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.

[9] B. Fischer and E. Visser. Retrofitting the AutoBayes program synthesis system with concrete object syntax. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 239–253. Spinger-Verlag, 2004.

[10] 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 *ENTCS*. Elsevier, April 2001.

[11] K. T. Kalleberg. www.spoofax.org.

[12] K. T. Kalleberg and E. Visser. Combining aspect-oriented and strategic programming. In N. M.-O. Horatiu Cirstea, editor, *Proceedings of the 6th International Workshop of Rule-Based Programming (RULE)*, ENTCS, Nara, Japan, April 2005. Elsevier.

[13] K. Olmos and E. Visser. Turning dynamic typing into static typing by program specialization. In D. Binkley and P. Tonella, editors, *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 141–150, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[14] K. Olmos and E. Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In R. Bodik, editor, *14th International Conference on Compiler Construction (CC'05)*, volume 3443 of *LNCS*, pages 204–220. Springer-Verlag, April 2005.

[15] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[16] 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 *LNCS*, pages 357–361. Springer, May 2001.

[17] 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 *LNCS*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[18] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 216–238. Spinger-Verlag, June 2004.

[19] E. Visser. A survey of strategies in rule-based program transformation systems. *J. Sym. Comp.*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.

[20] 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.

[21] E. Visser et al. www.stratego-language.org.

[22] D. G. Waddington and B. Yao. High fidelity C++ code transformation. In *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications*, ENTCS. Elsevier, April 2005.