Formal and Practical Aspects of Domain-Specific Languages:

Recent Developments

Marjan Mernik
University of Maribor, Slovenia

MACQUARIE
UNIVERSITY
LIBRARY

Information Science
REFERENCE

Managing Director: Lindsay Johnston
Editorial Director: Joel Gamon
Book Production Manager: Jennifer Romanchak
Publishing Systems Analyst: Adrienne Freeland
Development Editor: Hannah Abelbeck
Assistant Acquisitions Editor: Kayla Wolfe
Typesetter: Henry Ulrich
Cover Design: Nick Newcomer

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Copyright © 2013 by IGI Global. All rights reserved. No part of this publication may be reproduced, stored or distributed in any form or by any means, electronic or mechanical, including photocopying, without written permission from the publisher. Product or company names used in this set are for identification purposes only. Inclusion of the names of the products or companies does not indicate a claim of ownership by IGI Global of the trademark or registered trademark.

Library of Congress Cataloging-in-Publication Data

Formal and practical aspects of domain-specific languages: recent developments / Marjan Mernik, editor.
p. cm.
Includes bibliographical references and index.
Summary: "This book presents current research on all aspects of domain-specific language for scholars and practitioners in the software engineering fields, providing new results and answers to open problems in DSL research"--Provided by publisher.
ISBN 978-1-4666-2092-6 (hardcover) -- ISBN 978-1-4666-2093-3 (ebook) -- ISBN 978-1-4666-2094-0 (print & perpetual access) 1. Domain-specific programming languages. I. Mernik, Marjan, 1964-
QA76.7.F655 2013
005.1'1--dc23
2012015755

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

All work contributed to this book is new, previously-unpublished material. The views expressed in this book are those of the authors, but not necessarily of the publisher.

Editorial Advisory Board

Barrett Bryant, *University of North Texas, USA*
Haiming Chen, *Chinese Academy of Sciences, China*
Jeff Gray, *University of Alabama, USA*
Nigel Horspool, *University of Victoria, Canada*
Paul Klint, *CWI, The Netherlands*
Ivan Luković, *University of Novi Sad, Serbia*
Ulrik Pagh Schultz, *University of Southern Denmark, Denmark*
Anthony Sloane, *Macquarie University, Australia*
Diomidis Spinellis, *Athens University of Economics and Business, Greece*

List of Reviewers

Vasco Amaral, *Universidade Nova de Lisboa, Portugal*
Moussa Amrani, *University of Luxembourg, Luxembourg*
Kyoungho An, *Vanderbilt University, USA*
Paolo Arcaini, *University of Milan, Italy*
Ritu Arora, *Texas Advanced Computing Center, USA*
Ankica Barišić, *Universidade Nova de Lisboa, Portugal*
Bruno Barroca, *Universidade Nova de Lisboa, Portugal*
Jacob Beal, *Raytheon BBN Technologies, USA*
Barrett Bryant, *University of North Texas, USA*
Javier Luis Cánovas Izquierdo, *École des Mines de Nantes, France*
Vanea Chiprianov, *Telecom Bretagne, France*
Tony Clark, *Middlesex University, UK*
Peter J. Clarke, *Florida International University, USA*
William Cockshott, *University of Glasgow, UK*
Daniela da Cruz, *University of Minho, Portugal*
Jesús Sánchez Cuadrado, *Universidad Autónoma de Madrid, Spain*
Milan Čeliković, *University of Novi Sad, Serbia*
Ersin Er, *Hacettepe University, Turkey*
Martin Erwig, *Oregon State University*
Roxana Giandini, *Universidad Nacional de La Plata, Argentina*

Miguel Goulão, *Universidade Nova de Lisboa, Portugal*
Jeff Gray, *University of Alabama, USA*
Sebastian Günther, *Vrije Universiteit Brussels, Belgium*
Pedro Henriques, *University of Minho, Portugal*
Frank Hernandez, *Florida International University, USA*
Jerónimo Irazábal, *Universidad Nacional de La Plata, Argentina*
Vladimir Ivančević, *University of Novi Sad, Serbia*
Geylani Kardas, *Ege University, Turkey*
Ján Kollár, *Technical University of Košice, Slovakia*
Kostas Kolomvatsos, *National and Kapodistrian University of Athens, Greece*
Alexandros Koliousis, *University of Glasgow, UK*
Ivan Luković, *University of Novi Sad, Serbia*
Greg Michaelson, *Heriot-Watt University, UK*
Jesús García Molina, *University of Murcia, Spain*
Nuno Oliveira, *University of Minho, Portugal*
Ulrik Pagh Schultz, *University of Southern Denmark, Denmark*
Claudia Pons, *Universidad Nacional de La Plata, Argentina*
Zoltán Porkoláb, *Eötvös Loránd University, Hungary*
Jaroslav Porubän, *Technical University of Košice, Slovakia*
Samir Ribić, *University of Sarajevo, Bosnia and Herzegovina*
Elvinia Riccobene, *University of Milan, Italy*
Siegfried Rouvrais, *Telecom Bretagne, France*
Patrizia Scandurra, *University of Bergamo, Italy*
Ábel Sinkovics, *Eötvös Loránd University, Hungary*
Anthony Sloane, *Macquarie University, Australia*
Diomidis Spinellis, *Athens University of Economics and Business, Greece*
Robert Tairas, *École des Mines de Nantes, France*
Bedir Tekinerdogan, *Bilkent University, Turkey*
Adam Trewyn, *Vanderbilt University, USA*
Maria João Varanda Pereira, *Instituto Politécnico de Bragança, Portugal*
Didier Verna, *EPITA, France*
Mirko Viroli, *Universita di Bologna, Italy*
Eric Walkingshaw, *Oregon State University, USA*
James Willans, *HSBC, UK*
Yali Wu, *University of Detroit Mercy, USA*

Table of Contents

Preface xvii

Acknowledgment xxiv

Section 1
Internal Domain-Specific Languages

Chapter 1
Extensible Languages: Blurring the Distinction between DSL and GPL 1
Didier Verna, EPITA Research and Development Laboratory, France

Chapter 2
Domain-Specific Language Integration with C++ Template Metaprogramming 32
Ábel Sinkovics, Eötvös Loránd University, Hungary
Zoltán Porkoláb, Eötvös Loránd University, Hungary

Chapter 3
Semantics-Driven DSL Design 56
Martin Erwig, Oregon State University, USA
Eric Walkingshaw, Oregon State University, USA

Chapter 4
An Evaluation of a Pure Embedded Domain-Specific Language for Strategic Term Rewriting 81
Shirren Premaratne, Macquarie University, Australia
Anthony M. Sloane, Macquarie University, Australia
Leonard G. C. Hamey, Macquarie University, Australia

Chapter 5
Comparison Between Internal and External DSLs via RubyTL and Gra2MoL 109
Jesús Sánchez Cuadrado, Universidad Autónoma de Madrid, Spain
Javier Luis Cánovas Izquierdo, École des Mines de Nantes – INRIA – LINA, France
Jesús García Molina, Universidad de Murcia, Spain

Chapter 6
Iterative and Pattern-Based Development of Internal Domain-Specific Languages 132
Sebastian Günther, Vrije Universiteit Brussel, Belgium

Chapter 7
Design Patterns and Design Principles for Internal Domain-Specific Languages............................ 156
Sebastian Günther, Vrije Universiteit Brussel, Belgium

Section 2
Domain-Specific Language Semantics

Chapter 8
Formal Semantics for Metamodel-Based Domain Specific Languages 216
Paolo Arcaini, Università degli Studi di Milano, Italy
Angelo Gargantini, Università di Bergamo, Italy
Elvinia Riccobene, Università degli Studi di Milano, Italy
Patrizia Scandurra, Università di Bergamo, Italy

Chapter 9
Towards Dynamic Semantics for Synthesizing Interpreted DSMLs 242
Peter J. Clarke, Florida International University, USA
Yali Wu, University of Detroit Mercy, USA
Andrew A. Allen, Georgia Southern University, USA
Frank Hernandez, Florida International University, USA
Mark Allison, Florida International University, USA
Robert France, Colorado State University, USA

Chapter 10
A Formal Semantics of Kermeta 270
Moussa Amrani, University of Luxembourg, Luxembourg

Section 3
Domain-Specific Language Tools and Processes

Chapter 11
Software Language Engineering with XMF and XModeler.. 311
Tony Clark, Middlesex University, UK
James Willans, HSBC, UK

Chapter 12
Creating, Debugging, and Testing Mobile Applications with the IPAC Application Creation Environment 341
Kostas Kolomvatsos, National & Kapodistrian University of Athens, Greece
George Valkanas, National & Kapodistrian University of Athens, Greece
Petros Patelis, National & Kapodistrian University of Athens, Greece
Stathes Hadjiefthymiades, National & Kapodistrian University of Athens, Greece

Chapter 13
Abstraction of Computer Language Patterns: The Inference of Textual Notation for a DSL 365
Jaroslav Porubän, Technical University of Košice, Slovakia
Ján Kollár, Technical University of Košice, Slovakia
Miroslav Sabo, Technical University of Košice, Slovakia

Chapter 14
Evaluating the Usability of Domain-Specific Languages 386
Ankica Barišić, Universidade Nova de Lisboa, Portugal
Vasco Amaral, Universidade Nova de Lisboa, Portugal
Miguel Goulão, Universidade Nova de Lisboa, Portugal
Bruno Barroca, Universidade Nova de Lisboa, Portugal

Chapter 15
Integrating DSLs into a Software Engineering Process: Application to Collaborative Construction of Telecom Services 408
Vanea Chiprianov, Telecom Bretagne, France
Yvon Kermarrec, Telecom Bretagne, France
Siegfried Rouvrais, Telecom Bretagne, France

Section 4
Domain-Specific Language Examples

Chapter 16
Organizing the Aggregate: Languages for Spatial Computing... 436
Jacob Beal, Raytheon BBN Technologies, USA
Stefan Dulman, Delft University, The Netherlands
Mirko Viroli, University of Bologna, Italy
Nikolaus Correll, University of Colorado Boulder, USA
Kyle Usbeck, Raytheon BBN Technologies, USA

Chapter 17
DSLs in Action with Model Based Approaches to Information System Development..................... 502
Ivan Luković, University of Novi Sad, Serbia
Vladimir Ivančević, University of Novi Sad, Serbia
Slavica Aleksić, University of Novi Sad, Serbia
Milan Čeliković, University of Novi Sad, Serbia

Chapter 18
A Domain-Specific Language for High-Level Parallelization 533
Ritu Arora, Texas Advanced Computing Center, USA
Purushotham Bangalore, University of Alabama at Birmingham, USA
Marjan Mernik, University of Maribor, Slovenia

Chapter 19
Design and Transformation of a Domain-Specific Language for Reconfigurable Conveyor Systems 553
Kyoungho An, Vanderbilt University, USA
Adam Trewyn, Vanderbilt University, USA
Aniruddha Gokhale, Vanderbilt University, USA
Shivakumar Sastry, The University of Akron, USA

Chapter 20
MoDSEL: Model-Driven Software Evolution Language 572
Ersin Er, Hacettepe University, Turkey
Bedir Tekinerdogan, Bilkent University, Turkey

Compilation of References 595

About the Contributors 638

Index 648

Detailed Table of Contents

Preface xvii

Acknowledgment xxiv

Section 1
Internal Domain-Specific Languages

Chapter 1
Extensible Languages: Blurring the Distinction between DSL and GPL 1
Didier Verna, EPITA Research and Development Laboratory, France

Out of a concern for focus and concision, domain-specific languages (DSLs) are usually very different from general purpose programming languages (GPLs), both at the syntactic and the semantic levels. One approach to DSL implementation is to write a full language infrastructure, including parser, interpreter, or even compiler. Another approach however, is to ground the DSL into an extensible GPL, giving you control over its own syntax and semantics. The DSL may then be designed merely as an extension to the original GPL, and its implementation may boil down to expressing only the differences with it. The task of DSL implementation is hence considerably eased. The purpose of this chapter is to provide a tour of the features that make a GPL extensible, and to demonstrate how, in this context, the distinction between DSL and GPL can blur, sometimes to the point of complete disappearance.

Chapter 2
Domain-Specific Language Integration with C++ Template Metaprogramming 32
Ábel Sinkovics, Eötvös Loránd University, Hungary
Zoltán Porkoláb, Eötvös Loránd University, Hungary

Domain specific language integration has to provide the right balance between the expressive power of the DSL and the implementation and maintenance cost of the applied integration techniques. External solutions may perform poorly as they depend on third party tools which should be implemented, tested and then maintained during the whole lifetime of the project. Ideally a self-contained solution can minimize third-party dependencies. The authors propose the use of C++ template metaprograms to develop a domain specific language integration library based on only the standard C++ language features. The code in the domain specific language is given as part of the C++ source code wrapped into C++ templates. When the C++ source is compiled, the C++ template metaprogram library implementing a full-featured parser infrastructure is executed. As the authors' approach does not require other tool than a standard C++ compiler, it is highly portable. To demonstrate their solution, the chapter implements a type-safe printf as a domain specific language. The library is fully implemented and downloadable as an open source project.

Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., & Felleisen, M. (2011). Languages as libraries. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 132–141).

Walkingshaw, E., & Erwig, E. (2011). A DSEL for studying and explaining causation. In *IFIP Working Conference on Domain-Specific Languages* (pp. 143–167).

KEY TERMS AND DEFINITIONS

Compositionality: The principle that an object can be defined and understood by considering its parts individually, then relating them in a systematic way. A desirable property of a language's design, its semantic domain, and the expressions it contains.

Domain Decomposition: The identification and separation of a semantic domain into its component subdomains and their relationships. Enables the domain to be modeled in a structured and modular way.

Domain Modeling: The representation of a (decomposed) semantic domain in a metalanguage with types and data types, forming a hierarchy of micro DSLs related by language schemas.

Domain-Specific Embedded Language (DSEL): A DSL defined within a metalanguage that uses metalanguage constructs directly as DSL syntax. Also called an internal DSL.

Deep Embedding: A technique for implementing DSELs where abstract syntax is represented by a data type and mapped onto the semantic domain by a valuation function.

Language Schema: A parameterized class of related languages, from which specific languages can be derived by instantiation.

Language Operator: An operation that produces a new language from one or more languages or language schemas (and possibly other arguments). Language operators are the mechanisms by which a language is incrementally extended and built from its component parts.

Semantics-Driven Design: A language design process that begins by identifying, decomposing, and modeling the semantic domain of a language, then systematically extending it with syntax.

Shallow Embedding: A technique for implementing DSELs where syntax is defined by functions that build semantic values directly.

Syntax-Driven Design: The traditional view that the design of a language begins by identifying its (abstract) syntax, and only later describing its semantics.

ENDNOTES

1. We cannot use the constructor name `Line` since it has been used already in the `Pic` data type.
2. Another strategy that is available in Haskell specifically is to overload functions using Haskell's type classes to produce different semantics (Carette et al., 2009).
3. For example, we could reuse `Data.Map` from the Haskell standard libraries.
4. We omit years from dates just for simplicity in this chapter.
5. Note that without an associated year value the stream of dates is actually cyclical.

Chapter 4
An Evaluation of a Pure Embedded Domain-Specific Language for Strategic Term Rewriting

Shirren Premaratne
Macquarie University, Australia

Anthony M. Sloane
Macquarie University, Australia

Leonard G. C. Hamey
Macquarie University, Australia

ABSTRACT

Domain-specific languages are often implemented by embedding them in general-purpose programming languages. The Kiama library used in this chapter for the Scala programming language contains a rewriting component that is an embedded implementation of the Stratego term rewriting language. The authors evaluate the trade-offs inherent in this approach and its practicality via a non-trivial case study. An existing Stratego implementation of a compiler for the Apply image processing language was translated into a Kiama implementation. The chapter examines the linguistic differences between the two versions of the Stratego domain-specific language, and compares the size, speed, and memory usage of the two Apply compiler implementations. The authors' experience shows that the embedded language implementation inflicts constraints that mean a precise duplication of Stratego is impossible, but the main flavor of the language is preserved. The implementation approach allows for writing code of similar size, but imposes a performance penalty. Nevertheless, the performance is still at a practically useful level and scales for large inputs in the same way as the Stratego implementation.

DOI: 10.4018/978-1-4666-2092-6.ch004

Copyright © 2013, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

INTRODUCTION

One popular domain-specific language (DSL) implementation approach is to embed the DSL in a general-purpose *host language* to create an *internal language* (Mernik, 2005; Fowler, 2010; Ghosh, 2011). In this chapter we consider internal languages where the embedding consists only of pure DSL constructs that are written directly using the host language and an unchanged host language compiler performs the only compilation or translation step. In essence, the DSL is a host language library but the syntactic flexibility of the host language is exploited to implement the DSL syntax.

The embedding approach is attractive for a number of reasons, but has associated drawbacks. Reusing the host language compiler significantly simplifies implementation of the DSL compared to implementing a standalone version of the language – all host language tools can be reused. The main drawback is that the DSL syntax and semantics may not be directly realizable in the host language, resulting in compromises. Host language tools may not present a domain-specific view of programs, requiring the programmer to be aware of the way in which the DSL is implemented. Nevertheless, the pure embedding approach can be useful, particularly where the target users are developers who are familiar with the host language and where there is resistance to the adoption of new tools or build processes.

How can we evaluate the embedding approach beyond these generic high-level considerations? Our view is that real insight can only be gained through case studies of DSLs of varying styles and domains. In particular, if an external DSL has already been designed and a standalone implementation exists, then a side-by-side comparison with a new embedded version can be especially revealing. Evaluating how close the internal implementation gets to the existing external implementation provides a measure of the success of this approach. It is not our intention to undertake a full comparison of the implementation techniques, but to treat the existing external implementation as an ideal and to evaluate the ability of the embedded approach to reproduce that ideal.

Recently we have completed a project that studied an embedded implementation of a strategic term rewriting DSL and compared it to an existing implementation (Premaratne, 2011). (The code from the project is available at https://wiki.mq.edu.au/display/plrg/stragma.) The Kiama library (Sloane, 2011) contains term rewriting features inspired by an existing external language called Stratego (Visser, 2004). Kiama is embedded in the Scala general-purpose language (Odersky, 2008), which was chosen for its excellent support of pattern matching, flexible syntax and powerful static type system. When we developed the rewriting component of Kiama we adhered to the Stratego design as much as was possible and sensible, given the constraints and opportunities provided by Scala.

In this project, Stratego played the role of a well-established external DSL that is implemented by a compiler and Kiama played the role of an internal implementation of that DSL. We evaluated the two implementations using a non-trivial Stratego application that one of us had developed previously: a compiler for the Apply image processing language (Hamey, Webb & Wu, 1987; Hamey, 2007; Hamey & Goldrei, 2008). The Apply compiler translates Apply programs into C, conducting standard semantic analysis and non-trivial optimization along the way. We translated the existing Stratego implementation of the Apply compiler into Scala using the Kiama library. The two implementations perform the same analyses and translations and both run on the Java Virtual Machine. Since the focus of the comparison was the rewriting DSL, the two implementations of Apply share a common implementation of other passes such as parsing and pretty printing.

This chapter presents the results of the project with a focus on the trade-offs that the embedding approach imposes on the rewriting language and its implementation. We take the perspective that the Stratego language and compiler represent an ideal in the sense that the designer had full freedom to vary them as desired. A comparison with the Kiama version reveals the compromises from that ideal that a developer has to make when using an embedding approach. Therefore the comparison reveals important insights into embedding in general and, more specifically, explores Scala's suitability as a host language. As far as we are aware, this project is the first side-by-side comparison of two implementations of the same non-trivial DSL.

We do not consider other comparative dimensions, such as usability or social and training-based reasons for choosing one approach over the other, since we do not have data from users other than ourselves. Comparing the speed of development of the embedded DSL with Stratego was not possible since we do not have data from the Stratego development. Nor do we consider other alternatives for implementation of the DSL, such as embedding it in host languages other than Scala. We focus on the implementations that were actually used in the case study. All of these other aspects are interesting potential topics for future work.

Overall, we find that the embedding approach produces a language that is very similar to the Stratego language, but that compromises are required to fit Stratego's features into the Scala language in a natural way. Most notably, differences in the treatment of name binding and pattern matching mean that the rewrites in Kiama are expressed somewhat differently to the Stratego version. The performance of the two implementations is similar; the Kiama version incurs a penalty, but not one that limits practical use, particularly since the performance scales in the same way as the Stratego version as inputs grow large.

The structure of the rest of the chapter is as follows. We first discuss previous work that has evaluated pure embedding as an approach for building DSLs. Then we give an overview of strategic term rewriting, Stratego, and Kiama's rewriting library. Next we present the methodology we used in our comparison to ensure that the results were meaningful. Following that is an overview of the Apply language, a description of the Stratego Apply compiler implementation, and discussion of how the same compilation task was achieved using Kiama. The last two sections contain evaluation: first a discussion of the language-level differences between Stratego and Kiama's rewriting library due to the embedding approach and the characteristics of Scala, and then an analysis of the time and space performance of the two Apply compilers.

We conclude with a consideration of Future Research Directions, most notably to encourage other researchers to conduct similar experiments so that the research community can move towards a more comprehensive understanding of language embedding across a variety of domains and host languages. A secondary benefit of our study is that we have identified a number of areas where Kiama's version of strategy term rewriting is deficient when compared to Stratego, most notably support for concrete syntax, dynamic rules and congruence operations. The Future Research Directions section also contains a discussion of these aspects.

BACKGROUND

Much has been written about domain-specific languages and their implementation. Recent excellent examples are the books by Fowler and Parsons (2010) and Ghosh (2011) that together provide developers with a comprehensive introduction to the motivations for DSLs and survey modern implementation techniques, including comparisons with the pure embedded DSL approach that is evaluated in this chapter. However, since the purpose of these books is to introduce the DSL approach and they have a great deal of ground to cover, their examples and evaluations of any one technique are necessarily brief.

Many researchers have written about and evaluated particular embedded DSLs (van Deursen; 2000). In the vast majority of these cases the DSLs are designed from scratch to solve a particular problem. As such, the DSLs are evaluated for their ability to solve the problem for which they were designed, rather than for the suitability of the embedding approach itself. Usually the evaluation of the latter occurs in the form of high-level observations about how easy it was to develop the DSL but with little detailed comparison to alternative approaches.

Thus, it is rare to find a detailed, side-by-side comparison between pure embedding and other different DSL implementation techniques. The closest work of which we are aware is Kosar *et al*'s comparison between external DSLs and application programming interfaces (APIs) in a general-purpose language

(Kosar *et al*, 2010; Kosar *et al*, 2011). Their papers report relatively small empirical studies and show that DSLs do seem to confer a benefit in terms of program comprehension and understanding. We can get some insight into pure embedding from these studies if we regard the APIs as being embedded DSLs, but is not a clear insight. A closer comparison is performed in Kosar *et al* (2008) where a variety of DSL implementation approaches are compared along dimensions such as similarity to domain notation. As is common for this kind of study, the case considered is fairly simple, but it does point the way toward a more comprehensive evaluation of DSL implementations.

This chapter complements the earlier work by studying a bigger, pure embedded DSL and a non-trivial application that uses it. Unlike the empirical evaluations of Kosar *et al*, our example is a complete, complex program transformation language. This complexity constrains our study to be narrower because

it is not feasible to build many implementations of such a complex DSL. Also, rather than design a case-study DSL with the embedding approach assumed, as is done in much of the literature, we have built an embedded DSL to approximate an existing external DSL. Therefore, we are evaluating how well the embedding approach can solve a pre-defined problem. We are interested in the technical ways in which the embedded language and its implementation provide the same end-user capabilities as the existing external language and its implementation.

The rest of this section provides some background on the domain of program transformation that is addressed by the DSL.

Strategic Term Rewriting and Stratego

Strategic term rewriting is a formalism for specifying transformations that operate on tree-structured data (the terms) (Visser, 2005). In one common application, the terms are parse trees of programs, and the purpose of term rewriting is to transform those programs into other programs. Often the transformed programs are written in a language that is different from that used by the original programs. In this application, strategic term rewriting implements the core of a compiler for some other DSL. This chapter concerns a compiler for an image processing DSL.

Stratego is an external DSL for strategic term rewriting (Visser, 2004) and the Kiama library contains a version of Stratego as an internal Scala DSL. This section gives a brief overview of term rewriting in Stratego and describes how Kiama corresponds to Stratego.

Suppose that we want to simplify constant arithmetic expressions such as `(4 * 0) + 2`. We can write a term to represent this expression as follows:

```
Add(Mul(Int(4), Int(0)), Int(2))
```


A *strategy* in Stratego is a rewriter that can be applied to the subject term, either succeeding and producing a new subject term, or failing. The simplest kind of strategy is a rule that matches a pattern against the subject term. If the pattern matches the rule constructs a new term and succeeds otherwise the rule fails.

Suppose that we want to simplify our expressions by replacing multiplication of a term by zero with zero, and by replacing addition of zero to a term with that term. We can write these simplification rules `MulZero` and `AddZero` in Stratego as follows.

```
MulZero: Mul(x, Int(0)) -> Int(0)
AddZero: Add(Int(0), y) -> y
```

In this syntax, the identifier gives a name to the rule, the part before the arrow is the pattern to be matched to a subject term, and the part after the arrow is the replacement term.

The two rules `MulZero` and `AddZero` can be combined into a single simplification strategy `Simplify` as follows:

```
Simplify = try(MulZero + AddZero)
```

This strategy uses Stratego's `try` strategy and '+' operator to combine the individual rules. When this strategy is applied to the subject term, it will attempt to simplify the subject term using either the `MulZero` or `AddZero` rules. Stratego operators use the success or failure of strategies to control the rewriting process. The '+' operator combines two strategies so that the combination non-deterministically chooses one of the strategies to successfully apply to the subject term; if neither strategy succeeds, then the '+' operator itself fails.

The `try` strategy is a higher-order strategy that "swallows" failure. An expression `try(`*s*`)` (where *s* is itself a strategy or rule) *succeeds* with the result of *s* if *s* succeeds when it is applied to the subject term. If *s* fails when applied to the subject term then `try(`*s*`)` succeeds leaving the subject term unchanged. In fact, `try` is defined in terms of more primitive operations as follows:

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

The operator '<+' represents deterministic choice. An expression s_1 <+ s_2 first tries strategy s_1 on the subject term; if it succeeds then its result is the new subject term. If s_1 fails, then s_2 is applied to the original subject term and its result (success or failure) is the result of the whole expression. Using '<+', `try` first tries *s*; if *s* fails, `try` uses the identity strategy `id` to succeed with the subject term unchanged.

`Simplify` by itself is no good to us for full simplification of a term, since it only applies its constituent rules at the root of the term. Stratego also provides higher-order generic traversal strategies that are the building blocks of programs that traverse into terms. For example, we may wish to apply our `Simplify` strategy in a bottom-up fashion to the whole term, so that simplifications are applied within sub-terms first and the results can then be exploited at higher levels. We can define a strategy to perform this whole term simplification as follows:

```
SimplifyAll = bottomup(Simplify)
```

Here, `bottomup` is a higher-order strategy that expresses the bottom-up traversal pattern without being specific about what is performed during the traversal. `bottomup` is defined as follows:

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

This implementation says to first recursively process all the sub-terms of the subject term, and then finally to apply the strategy argument *s* to the subject term itself. In detail, the semicolon operator is sequential composition. In 's_1; s_2', if s_1 succeeds, then its result is passed to s_2, otherwise

the whole expression fails. The strategy `all(s)` is a generic traversal that applies *s* to all of the immediate sub-terms of the root of the subject term. If *s* succeeds on all of the sub-terms, the resulting new sub-terms are combined using the constructor that appears at the root of the original subject term and `all(s)` succeeds. If *s* fails on any of the children, then `all(s)` fails. This is where the `try` strategy in Simplify is important: the `AddZero` and `MulZero` rules do not match at all nodes in the tree, but `try` ensures that the `Simplify` strategy succeeds at the non-matching nodes, preserving them unchanged. This allows `SimplifyAll` to successfully process the entire tree.

This example touches on only a few of the library strategies that Stratego offers and only shows some of the power of this approach. The primitives such as deterministic and non-deterministic choice, sequencing, and generic traversals can be combined together to yield a vast array of complex rewriting processes. The Stratego language is a concise notation for these processes.

Kiama

The main goal of Kiama's rewriting library is to make the power of strategic term rewriting accessible to mainstream programmers. Using Stratego requires installation of a new toolset, conversion of data into the format that Stratego needs, and so on. In comparison, Kiama is a Scala library so any Scala program can use its facilities in a lightweight fashion with no new tools or data conversion being required.

When we developed Kiama's rewriting library, we tried to be faithful to the Stratego language as much as was possible and as much as was sensible. Kiama is constrained by Scala, so some things that Stratego offers are not possible due to clashes with Scala syntax and semantics. Others are not sensible because they conflict with the way that Scala programmers normally construct programs. We compare the languages in detail later in the chapter. For now, we illustrate Kiama by showing how the arithmetic simplification example above would be written in this embedded DSL.

The basic rules `MulZero` and `AddZero` can be written in Scala using Kiama's `rule` method. For example, `MulZero` would be

```
val MulZero =
    rule {
        case Mul(x, Int(0)) => Int(0)
    }
```

The only part of this rule definition that is specific to Kiama is the `rule` method; all other syntax and semantics is standard Scala. The argument to `rule` is a function literal, in this case standing for a partial function that performs the pattern match and, if the match is successful, returns the simplification. Kiama benefits greatly from the high level of support for pattern matching in Scala, but there are some consequences of using that support, which we discuss later.

In Kiama, strategies are implemented as functions that take the subject term and return a Scala *option* value to indicate success or failure. Option values come in two varieties: `None` that represents an optional value that is not present, and `Some(v)` that represents a value `v` that is present. A Kiama strategy that fails returns `None`, and one that succeeds returns `Some(v)` where `v` is the new value of the subject term.

`rule` lifts its partial function argument to this option encoding. If the partial function is defined on the current subject term, then it is applied and the function result is wrapped in a `Some` to indicate success and the new subject term. If the partial function is not defined on the subject term, then `None` is returned to indicate that the rule has failed.

Returning to the example, `Simplify` can be defined using Kiama in a similar way to the Stratego definition, except that `try` is a Scala keyword, so Kiama uses `attempt` instead as demonstrated in Box 1.

An Evaluation of a Pure Embedded Domain-Specific Language for Strategic Term Rewriting

`attempt` is defined in a similar fashion to `try` in Stratego, except that more type information must be provided as presented in Box 2.

The argument `s` is passed by name (indicated by the prefix double arrow) to avoid premature evaluation of recursive strategies. `id` is the identity strategy as in Stratego. Thus, `attempt` returns a strategy that first tries `s` and, if that fails, leaves the subject term unchanged.

In these Kiama definitions we are using '+' and '<+' as if they were pre-defined operators, but they are actually methods of Kiama's Strategy class. Scala allows the period of a method call `o.m(a)` to be omitted when there is a single argument *a*, so it can be written *o m a*. Scala understands `s <+ id` to be `s.<+(id)`, invoking the '<+' method of the Strategy class.

The implementation of '<+' is straight-forward. It returns a strategy that first applies the left operand of '<+' to the subject term. If that application returns `Some(v)` for some value `v`, then the left operand has succeeded and that option becomes the result of the whole operation. If the application of the left operand returns `None`, then it has failed, so the strategy produced by '<+' then applies the right operand of '<+' to the subject term, returning whatever it produces. Similar approaches are used to define other primitive operations.

In the example, `SimplifyAll` and `bottomup` can be defined easily using Kiama as presented in Box 3.

In the `bottomup` definition, the Kiama method '<*' is used for sequential composition whereas ';' is used in Stratego. The syntactic change is necessary because Scala uses the semicolon character as a statement terminator and does not allow us to reuse it as a method name.

This simple example shows that the main aspects of the Stratego language and Stratego programs can be expressed in a natural way in Kiama and Scala. Some compromises are already evident: `attempt` replacing `try` and '<*' replacing ';', and additional type information must be provided for Kiama. We will compare the languages in more detail later in the chapter.

A major advantage of the embedding approach is the simplicity of the implementation of the rewriting language. The strategy libraries in Stratego and Kiama are of similar size since they use a similar syntax, but the core implementation of Kiama is smaller, only around 650 lines of mostly straightforward Scala code, excluding comments and blank lines. In contrast, Stratego has been bootstrapped so that its implementation consists of over 2000 lines of Stratego that produce Java output. (Note that the implementation language for the rewriting language (Scala or Java) is unrelated

Box 1.

```
val Simplify = attempt(MulZero + AddZero)
```

Box 2.

```
def attempt(s: => Strategy): Strategy = s <+ id
```

Box 3.

```
val SimplifyAll = bottomup(Simplify)
def bottomup(s: => Strategy): Strategy =
    all(bottomup(s)) <* s
```

to the term language actually manipulated by a rewriting program.)

The differing implementation sizes are likely to have an impact on reliability, since we can expect that debugging and maintaining a small library is easier than doing the same for a non-trivial compiler. The Kiama implementation delegates a larger proportion of its functionality to the Scala compiler than the Stratego one does to the Java compiler. A comparison of reliability over a long period of time would be interesting but we do not have this data.

METHODOLOGY

The general question that we are investigating in this case study is whether embedding is a reasonable alternative to building an external domain-specific language implementation. Our methodology separates this question into two parts: one comparing the languages and one comparing their implementations. This section provides an overview of our approach to these sub-questions; we consider their answers in detail later in the chapter.

Our approach is to base our comparisons on a case study of a non-trivial application written in the domain-specific language: a compiler for the Apply image processing language designed and implemented in Stratego by one of us (Hamey, 2007; Hamey & Goldrei, 2008), which we will refer to as *Stratego-Apply*. Another of us translated Stratego-Apply into Kiama as far as was possible using a direct translation to produce *Kiama-Apply* (Premaratne, 2011).

Since we are only interested in comparing the rewriting language implementations, our comparison considers only the core phases of the Apply compilers that are implemented by Stratego and Kiama. Phases such as parsing the input text and pretty printing the output text are not implemented by the term rewriting DSL. These phases are shared between the two implementations and are not included in any measurements.

In both implementations, a parser that is generated from a specification developed for the original external implementation analyzes the input text. This parser produces an ATerm, a data representation specifically designed for term rewriting (van den Brand, 2000; van den Brand, 2007). The Kiama implementation further translates that ATerm into a native Scala data structure. In both versions, the core rewriting phases transform the term that represents the input text into a term that represents the target C program. Kiama-Apply then translates the Scala term back to ATerm format. Finally, both implementations used a shared pretty-printer to convert the final ATerm into the target code. We required that Kiama-Apply produce exactly the same target ATerms as Stratego-Apply for all available input texts, so that we can be confident that the same output text is produced by the two implementations.

A comparison of external and internal languages must consider both syntax and semantics, so we further sub-divide along those lines. The syntactic constructs that are possible in an external implementation are only limited by the imagination of the language designer and the parsing method being used. An internal implementation must reuse the syntax of the host language and take advantage of any available syntax extension facilities. Therefore an internal implementation of an existing external language is unlikely to be able to duplicate the external syntax exactly. The question becomes: how close can we get? We consider this question by examining the main language constructs as they are used in the Apply case study.

Similar to syntax, the semantics of an internal language is bound by the semantics of the host language. The semantics of an external language is unrestricted in the sense that the implementation is able to perform any interpretation or translation that is needed. The degree to which the internal language can faithfully reproduce the semantics

of the existing external language depends on the similarity between the semantic models of the external language and host language. While in theory it is possible to reproduce the semantics of any external language in any Turing-complete host language, an awkward embedding due to widely differing semantic models is likely to be unsatisfactory. Again, we use the Apply case study code to examine these differences.

Assuming that a satisfactory syntactic and semantic correspondence can be achieved, it remains to compare the specific implementations of the Apply compiler. We consider the size of programs written in the rewriting DSL as a measure of implementation difficulty and the time-space performance of the implemented Apply compiler as a measure of usability. While a side-by-side comparison can only provide approximate estimates of these measures, it is sufficient to demonstrate the practicality or otherwise of the embedding approach. We expect that a custom external language implementation should outperform an internal one due to the opportunity for domain-specific optimization, particularly optimization of data representation. Somewhat balancing this advantage is the likelihood that the host language implementation will have been the focus of much more effort and general-purpose optimization than the implementation of the external language.

APPLY AND ITS COMPILERS

Apply is a language for expressing image processing operations. It has two main goals: efficient implementation and portability. Apply programs are portable across uniprocessor and parallel architectures and across different image data representations, while maintaining an efficiency that equals or exceeds good quality hand-written code (Hamey, 2007). This section provides a brief overview of Apply and its implementation sufficient to demonstrate the non-trivial nature of the domain and the processing performed by the Apply compiler. We skim over many details since they are not necessary to understand the rest of the chapter. A reader who wishes to learn more about image processing with Apply is directed to Hamey (2007) in the first instance.

Apply uses a processing model in which the programmer writes a computation for a single pixel location and the compiler generates code that performs the computation on all of the pixels in an image. This programming model allows the compiler to generate efficient code for a wide variety of architectures and to exploit parallelism since the pixel computations are independent.

As an example, consider the typical small Apply program in Figure 1 which implements part of a standard Sobel edge detector computation to identify boundaries between regions in an image. Syntactically, Apply is based on a 1980s version of the Ada programming language (Ada, 1982). This Sobel procedure considers a *window* into the image centered on a given pixel and extending one pixel outward in each direction (`from` argument in line 1). The output is a single byte that becomes the new intensity of the pixel under consideration (`to` argument in line 2). The algorithm combines the intensities of six of the neighbors of the pixel under consideration to get a new intensity for that pixel (lines 6 and 7). To keep the value in range it then takes the absolute value (line 8) and range limits it to fit into a byte (line 9). Finally, it returns the result (line 10). Of course, the pixels at the edge of the image do not have all of the six neighbors that are referenced by the procedure. The `border` keyword on the input image window (line 1) specifies what happens when the computation attempts to access a pixel that falls outside the input image bounds. In this example, the constant value `0` is used for any such pixel locations.

Apply programs are portable to different target environments, including different image data structures, alternative target languages and different parallel programming methodologies. To

Figure 1. A simple Apply program: Partial Sobel edge detection

```
 1 procedure sobel(from: in window (-1..1, -1..1) of byte border 0,
 2                 to: out window of byte)
 3 is
 4   x : integer;
 5 begin
 6   x := from(-1,-1) + 2 * from(-1,0) + from(-1,1)
 7       - from(1,-1) - 2 * from(1,0) - from(1,1);
 8   if x < 0 then x := -x; end if;
 9   if x > 255 then x := 255; end if;
10   to := x;
11 end sobel;
```

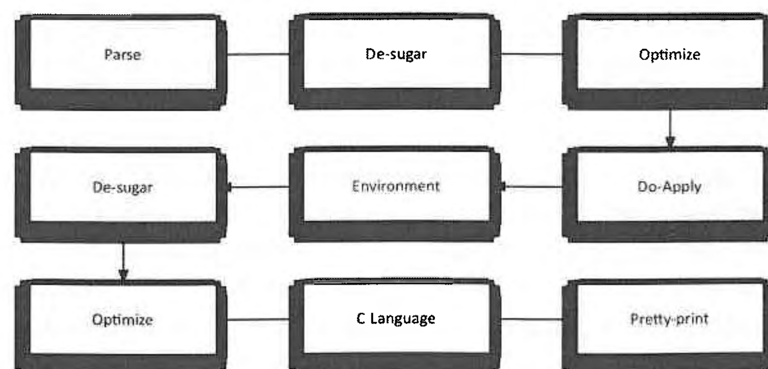
provide this level of portability, the Stratego-Apply compiler is divided into phases (Figure 2). A specific intermediate phase (Environment) is used to adapt the Apply program to the target environment; replacing this phase with a different module will target the Apply compiler to a different environment. Similarly, the final compiler phases (C Language and Pretty-print) can be replaced to adapt the compiler to different target languages.

The Desugar phase is a small component of the compiler that performs some initial transformations to simplify the structure of the Abstract Syntax Tree (AST) for subsequent processing. For example, Desugar converts a list of declared variables that are associated with a single type declaration into a list of variables each of which has its own type declaration.

The Optimize phase is a significant component of the compiler that is responsible for a large portion of the transformation of the program. This phase performs both constant propagation optimizations, where expressions whose values are statically known to be constant are replaced by those constants, and constraint propagation optimizations, where statically known constraints on the values of variables are used to simplify code that uses those variables. The latter optimizations are particularly important for improving the generated code, particularly when constraints on loop indices are used to eliminate unnecessary bounds checks on image pixel accesses. These optimizations have a very significant effect on the execution speed of the generated code.

The Do-Apply phase prepares the Apply program for the Environment phase by inserting a placeholder for the image processing looping

Figure 2. The phases of the Apply compilers



structure and associating more type information with variables. These transformations make it easier to write the Environment phase for different target environments.

The Environment phase uses the type information inserted by Do-Apply to generate the pixel accesses that are appropriate to the target environment, and to implement the image border handling (i.e., the special cases for pixels that occur at the border of an image and therefore do not have a full set of neighbors). Environment also inserts the loops that actually traverse the image. In the Apply compiler used in this paper, the Environment phase supports a uniprocessor implementation with images stored as matrices of pixels. Even in this simple target environment, Apply modules significantly outperform hand-written code because of the sophisticated looping structures and optimizations employed by the Apply compiler (Hamey, 2007).

Both of the Desugar and Optimize phases are performed first on the initial Apply program and then repeated after the Environment phase. This allows the Environment phase to be written with the full power and optimization capabilities of the Apply language. After the second optimization phase has run, the final two phases (C Language and Pretty Printing) translate the intermediate representation into an abstract syntax tree representing the generated C program, and print that tree to obtain the C program text, respectively.

To demonstrate an intermediate stage of program transformation and illustrate the complexity of the transformations, Figure 3 presents a pretty print of the AST for the Sobel program (Figure 1) after the Environment phase and before the second Optimize phase. (Some details have been omitted and the code has been reformatted slightly to keep Figure 3 as small as possible. See Hamey (2007) for a full description.) We use an extended concrete syntax to represent structures that are not part of the Apply language, allowing the compiler writer to express transformations using Stratego's concrete syntax capability. The extended syntax is identified by special tokens that commence with the '`@`' symbol. In Figure 3, the notation '`@:=`' represents a C-style assignment and '`@(...)`' represents array subscripting. In Apply, as in Ada, the same syntax is used for function calls and array subscripts, so the extended syntax is required to differentiate array subscripts from function calls.

The most obvious change in Figure 3 compared to Figure 1 is the expansion of the code. In particular, the Environment phase has embedded the Apply function body (lines 6-10 of Figure 1) into two loop blocks – the first loop block (lines 8-23 of Figure 3) processes pixels that are close to the edge of the image, explicitly modifying the `column` index variable to skip over the center of the image (lines 17-21). The second loop block (lines 24-32) processes the middle region of the image where the pixels that are being accessed fall entirely within the image. Border handling is required in the first loop block and implemented using inline if constructs (`@if` extended syntax) to compare the array subscripts against the image bounds and return the border value `0` for out-of-bound accesses (lines 12 and 13). The second loop block does not require border handling, but the Environment phase uses the same implementation code and relies on the subsequent Optimize phase to eliminate the unnecessary bounds checks using constraints on the loop index variables. This optimized two-loop structure greatly improves execution speed compared to a structure where a single loop block processes the entire image and the image bounds are checked for every pixel access.

Figure 3 also demonstrates the Apply assert statement which provides the constraints for optimization (lines 10, 11, 14, 15, 28 and 29). The `@cfor` loop (line 9) is extended syntax for a C-style `for` loop; the `assert` statements within it inform the compiler of the range constraints for the `row` and `column` loop index variables. The Optimize phase automatically generates similar constraints from Apply-language conditional and looping structures, and uses the constraints to optimize

Figure 3. Sobel Apply program after Environment stage

```
 1 procedure sobel (from : @in array () of byte border 0,
 2                  to : @out array () of byte,
 3                  height, width : in integer)
 4 is
 5   x : integer;
 6   row, column, app_index, app_size : integer;
 7 begin
 8   for row in 0..height - 1 loop
 9     @cfor column @:= 0; column <= width - 1; loop
10       assert column >= 0 and column <= width - 1;
11       assert row >= 0 and row <= height - 1;
12       x := @if (row >= - -1 and row < height - -1 and column >= - -1 and
13                column < width - -1, from @((row + -1) * width + column + -1), 0) +
...;
14       if x < 0 then assert x < 0; x := - x; end if;
15       if x > 255 then assert x > 255; x := 255; end if;
16       to @(row * width + column) := x;
17       if column = - -1 - 1 and row >= - -1 and row < height - 1 then
18         column := column + width - 1 + -1 + 1;
19       else
20         column := column + 1;
21       end if;
22     end loop;
23   end loop;
24   for row in - -1..height - 1 - 1 loop
25     for column in - -1..width - 1 - 1 loop
26       x := @if (row >= - -1 and row < height - -1 and column >= - -1 and
27                column < width - -1, from @((row + -1) * width + column + -1), 0) +
...;
28       if x < 0 then assert x < 0; x := - x; end if;
29       if x > 255 then assert x > 255; x := 255; end if;
30       to @(row * width + column) := x;
31     end loop;
32   end loop;
33 end sobel;
```

the code. In this example, the constraints generated from the second looping block are used later to eliminate the inline if constructs, resulting in efficient C code where the second looping block performs no bounds checks on pixel accesses.

Kiama-Apply

Each Stratego-Apply module was directly translated to a Scala class to construct Kiama-Apply.

The aim was to measure the effectiveness of the embedding method, not necessarily measure the best possible Scala implementation. Therefore, the strategies and rules were translated as directly as possible into equivalent Scala code that uses Kiama.

We will see in the Language Comparison section that some forms of binding through pattern matching in Stratego are hard to duplicate exactly in Scala since pattern matching cannot be separated from the actions carried out after a pattern matches. Therefore, in some cases, we used more idiomatic Scala rather than use a convoluted approach to mimic the Stratego code. For example, Stratego conditional choice expressions were often replaced with a sequence of pattern matching cases, and pattern guards were sometimes used to express constraints that would normally be handled in Stratego by side-conditions expressed using the `where` strategy.

Some Stratego library strategies that were used in Stratego-Apply but are not present in the Kiama library were written and included in the Kiama-Apply support module. Also, the Environment stage of Stratego-Apply makes heavy use of concrete syntax to express patterns and terms, instead of writing them in prefix notation (Bravenboer, 2008). In some places, we developed new infix operators to make it easier to compose code fragments, in lieu of proper concrete syntax support.

We did not spend much time optimizing the Scala code to get the best performance. However, we did do some profiling to remove obvious hot spots. This approach is justified since we can assume that Stratego has had some basic performance optimization over its years of existence, but we have no reason to believe that its performance is as good as it can get. Of course, any performance measurements from this kind of comparison should be taken as indicative of the particular case being measured, and not necessarily representative of general performance. For this reason in the Implementation Comparison section we are mainly interested in trends that show performance changes as problem size increases, rather than in absolute numbers.

LANGUAGE COMPARISON

We now turn to a comparison of the two rewriting languages based on our experiences developing the two Apply compilers. As discussed earlier, we regard Stratego as an ideal and Kiama's rewriting library as an approximation to that ideal. The designer of Stratego had complete freedom when designing the syntax and semantics of the language. In Kiama we were constrained both by having Stratego as a target and by using Scala as the host language. What is the effect of this reduced flexibility? This section discusses the main differences between Stratego and Kiama's version of the DSL, identifying aspects of syntax, name binding, typing, and domain focus as key areas. We also identify specific features of Scala that enhance Kiama's ability to approximate Stratego, particularly its syntactic flexibility.

Expressions

Stratego has a rich expression language for describing strategies. As we saw in the arithmetic simplification example earlier, Kiama is able to closely approximate both the syntax and semantics of the strategy language. Scala is extremely helpful: the ability to use arbitrary characters in method names and to omit the period and parentheses in method calls means that Stratego's syntax can be duplicated almost exactly. Many general-purpose languages are less helpful, permitting only pre-defined operators to be overloaded or even restricting the library interface to conventional function-call syntax.

Even with this substantial help, one Stratego operator did require more thought. The ternary guarded deterministic choice operator, written $s_1 < s_2 + s_3$, is actually more primitive than the '<+' operator which we saw earlier. $s_1 < s_2 + s_3$ first applies s_1, and if s_1 succeeds, then applies s_2 to the result of s_1; if s_1 fails, s_3 is applied to the original subject term. s_1 <+ s_2 is therefore just syntactic sugar for s_1 < `id` + s_2.

Scala provides no direct assistance for ternary or higher-arity operators. Kiama's solution, conceived by our colleague Lennart Kats, is to define the '+' operator to play two roles. When the expression $s_2 + s_3$ is used as a strategy it has the expected non-deterministic choice semantics; when it is used as the second argument of the '<' binary operator (i.e., as s_1 < (s_2 + s_3)) it simply acts as a tuple to hold the s_2 and s_3 strategies for use by the guarded choice. Scala defines '+' to have a higher precedence than '<', so the parentheses can be omitted.

The other main commonly used syntactic construct in Stratego expressions is the functional notation used in strategy definitions such as `bottomup`. This notation has a direct analog in Scala's method call syntax, as shown in earlier examples.

Kiama extends Stratego to some extent. For example, in Kiama, any term can be used as a strategy, with the meaning that the current subject term is discarded and replaced by the given term. This feature is the counterpart to Stratego's explicit exclamation mark build operator. Such a strategy always succeeds. Kiama implements this kind of conversion using Scala's user-defined implicit operations. If the Scala compiler determines that a value of type *U* is required in some context, but the value provided is of type *T*, then it will look for an operation of type *T* => *U* that is marked with the `implicit` keyword. If such an operation is found, the compiler inserts a call to it to make the expression type correct. This facility means that the Scala type system can be extended with domain-specific conversions. Kiama terms can be used as strategies because there is an implicit operation that converts a term into a rule that matches anything and returns the term.

Binding Constructs

An important way in which external DSLs and internal DSLs often differ is in the way that they handle binding. Names are routinely bound to expressions and used elsewhere in a DSL program. External DSLs frequently have binding and scoping constructs that are non-standard, so it is not always possible to imitate them exactly in an internal language. The internal language must live with the binding constructs of the host language and the rules that govern their use.

Stratego has two main binding constructs: one for rules and one for strategies, as illustrated earlier in the Background section. Kiama uses Scala's value and method definition constructs to handle these two cases. Rules are typically defined as values; the availability of lazy values and the fact that Kiama's methods take their arguments by name enable recursive rules to be defined. Higher-order strategies are defined as methods that instantiate a strategy with arguments, with recursion coming for free.

A Stratego rule is syntactic sugar for a strategy that matches a pattern and, if the match succeeds, builds a new term, possibly using values that were bound during pattern matching. For example, a basic rule of the form *name* : p_1 -> p_2, is equivalent to the strategy definition *name* = ? p_1; ! p_2, where the question mark indicates a match of pattern p_1 and the exclamation mark indicates a build using pattern p_2. Implicit in the rule definition is that any free variables of p_1 are bound in a new local scope associated with the rule. This translation is generalized when the rule has arguments or when side conditions are placed on the matching process.

Kiama, on the other hand, reuses Scala's pattern matching facilities. Therefore a pattern match cannot be regarded as a primitive operation in the same way as in Stratego. A basic rule in Kiama is implemented by a value that is bound to the result of the rule method:

```
val name = rule { case p1 => p2 }
```

Therefore the match and the build are necessarily combined in Kiama. In practice, the same rules can be expressed, but the Kiama version is usually more verbose since there is no shorthand for matching.

Stratego's rule binding construct uses *implicit composition*: when presented with more than one rule definition with the same name, the rule bodies are combined using a deterministic choice operator to form a single rule definition that tries each body in turn until one succeeds or the rules are exhausted. More generally, implicit composition is when a name is used more than once to denote pieces of the definition of some entity, not different entities. It is frequently used in external DSLs to save the programmer from having to explicitly write out the composition operation. However, implicit composition is hard to realize in an internal language since general-purpose languages typically do not provide it as

a primitive and there is no opportunity to collect the individual definitions.

A common problem with implicit composition is that the order of composition must be precisely defined, or it must be left undefined and cannot be relied upon by the programmer. Stratego chooses the latter approach which means that problems can occur if the patterns of the individual rules overlap. Scala does not have implicit composition for value or method definitions, so Kiama users must explicitly combine their rules, either in a single pattern-matching construct or by using an explicit choice operator.

Apart from pattern matching, Stratego's other binding facilities can be achieved in Kiama using normal Scala bindings. Stratego has an explicit scope operator, written using braces. New scopes can be introduced so that names bound outside are hidden. For example, the construct {*x*: *s*} introduces a scope that hides any existing binding of *x* during the application of the strategy *s*. Kiama uses normal Scala scoping mechanisms, including local blocks, so there is no need for another scoping construct. Local bindings in rule or strategy definitions take the place of Stratego's let construct. Stratego's strategy definitions syntactically differentiate between arguments that are terms and those that are strategies. No such distinction is needed in Kiama since the argument's type controls how it can be used. Finally, Stratego has a module system that is subsumed by Scala's extensive features for modularity and composition.

Semantics

As mentioned earlier, Stratego has a simple success or failure semantics for strategy application. The composition operations use the success or failure of component strategies to control subsequent strategy invocations. Kiama encodes success or failure as a simple option value, so it is easy to implement the composition operations.

Kiama distinguishes between rules and strategies more than Stratego, because of its use of Scala function literals to define the pattern matching inherent in the bodies of rules. Therefore, Kiama has additional constructs to enable rules and strategies to be combined in ways that are more natural in Stratego's unified approach. For example, Kiama provides a `rulefs` method that is analogous to `rule` but instead of the argument function returning a term, it returns a strategy.

The general form of Stratego's rule construct incorporates side-conditions expressed in a `where` clause. A rule p_1 -> p_2 `where` *s* is equivalent to ? p_1; `where`(*s*); !p_2, with `where`(*s*) being equivalent to {*x*: ?*x*; *s*; !*x*}, where *x* is free in *s*. By this definition, `where`(*s*) applies strategy *s* to the subject term, discarding any changes made by *s* to the subject term but retaining the success or failure and also retaining any variable bindings effected in *s*. Thus, *s* may be used to bind variables that can be used in p_2 (but not in p_1). The corresponding construct in Kiama is more limited since its bindings must obey the lexical nesting rules of Scala. Hence, in the Kiama version of a rule with a `where` clause the pattern matching in *s* must be moved so that it is nested within the right-hand side of the case that is executed after a successful match of p_1. In that position the bindings from *s* can be used by subsequent term construction in p_2.

Types

Stratego and Kiama differ significantly in the way that they approach typing issues. The types of the structures operated on by Stratego programs are expressed by signatures that specify the available constructors, their arities, and the types of their children. Stratego statically verifies that the arities are respected within rules and strategies, but does not require that the types of sub-terms match the types specified in the constructor definitions. The Stratego-Apply compiler takes advantage of this

flexibility in a number of ways. For example, the signature of the term that represents the input text contains some anonymous constructors, which means that an implicit conversion of type may be performed when terms are built. Stratego-Apply also constructs terms that mix source and target constructs (e.g., Apply statements and C statements). In the Stratego/XT system, a separate program can be used to verify that a particular term matches a given signature.

In contrast, Kiama terms must be correctly typed at all times. Therefore the term signature is extended in Kiama-Apply to name anonymous constructors and to insert applications of them at appropriate places when the initial Scala value is constructed. Transformations that mix syntaxes require a common super type for the terms from the two languages that are to be mixed. As a result of these effects of stronger typing, the strategies are a bit longer in Kiama-Apply than in Stratego-Apply. However, in the Kiama version the Scala compiler is able to check that the terms being constructed are correctly typed, so Kiama is more statically type safe than Stratego.

Beyond their types, the representation of the terms is also a major difference. Stratego operates on terms implemented by the ATerm library. ATerms are designed specifically for term rewriting and are optimized to reduce duplication. Kiama terms are instances of Scala case classes (Odersky 2008, Chapter 15); in other words, they are standard Scala objects with some extra compiler-provided support for construction and pattern matching. Therefore, Kiama gains none of the benefits of using a representation designed for the rewriting domain, but Kiama terms can be used by other Scala code in a natural way.

Typing issues also arise for strategies. Stratego strategies are indistinguishable by type and Kiama follows this lead. This design limits the static checking that can be performed. For example, it is not possible to statically determine that $s_1 ; s_2$ will always fail by proving that the terms produced by s_1 are not acceptable to s_2. In both languages, such errors can only be detected through testing.

Breaking Out of the Domain

A major motivation for DSLs is to provide a restricted context in which a specific kind of problem can be solved. By keeping the number of constructs small, limiting the ways in which they can be combined, and defining a simple semantics, a DSL can enforce a discipline over development that helps to control application complexity and enables domain-specific optimizations. However, a DSL can also be seen as a straightjacket that forces all problems to be considered from its limited viewpoint (Mernik, 2005). Some problems may be impossible or inconvenient to solve using just the DSL.

Considering the Stratego DSL, we can ask whether it is natural to view all transformation problems or parts thereof as instances of the application of strategies to terms. For example, consider a problem that requires numeric data to be manipulated. The Stratego library provides strategies such as `add` which matches a tuple of values and adds them together if they are numbers. In effect, addition is lifted to the strategy domain. It is arguably more natural to just access two integers in a term, add them together and use the result to construct a new term. In a statically typed, internal DSL with access to host language arithmetic operations, this addition is easy and safe to use.

Another example of the power that can be achieved by an internal version of a DSL breaking out of the problem domain concerns complex data structures. In Stratego we can represent arbitrary data structures as terms, but it is common to want to reuse more standard ones such as lists, sets, or hash tables. Stratego provides syntactic sugar for list pattern matching and construction, but these operations still live in the rewriting world. Operations such as mapping over, filtering, or sorting a list are written as strategies. Sets are encoded as lists. Keeping within the strategic rewriting domain is attractive from a purity perspective, but is not a particularly natural way to think about these common data structures and operations on them. In fact, Stratego doesn't stay pure for all data structures; hash tables are provided as primitive values and their operations are defined outside the rewriting domain.

In contrast, Kiama's version of the DSL focuses entirely on the core rewriting problem domain and does not attempt to import data structures into that domain. Accordingly, the DSL is more lightweight and easier for a Scala programmer to pick up than if a new encoding of data structures had to be learned. For example, Scala collections can be intermixed freely with problem-specific terms. All rewriting operations are agnostic to the type of structure that is being processed. Operations such as sorting can be performed directly on the data structures rather than having to be expressed as rewrite rules. Data structures can also be used in rewriting operations without requiring them to be part of the subject term. For example, in the Apply context it could be useful to compute lookup tables that are accessed by rewrites during analysis and optimization.

Summary

Overall, Kiama's rewriting DSL syntax and semantics are quite close to Stratego's. Some Scala features, most notably pattern matching, user-defined infix operators and implicit conversion allow Kiama to closely approximate the syntax of Stratego rules and strategy definitions. The semantics of strategy evaluation was easily realized by the Kiama version and integrated nicely with Scala partial functions. The main semantic alteration was the inability to separate the build and match operations as in Stratego, since pattern matching in Scala is integrated with the specification of the action to take when a pattern matches. This change required some rephrasing of rewrites in the Kiama-Apply compiler. Scala's stronger typing also forced some changes, most notably by requiring terms to be properly typed. As expected, many features of Stratego, including definition syntax and modularity support, required little special treatment in Kiama since they come for free with Scala. The main exception was the inability of Kiama definitions to implicitly compose since Scala composition is explicit. The Kiama version was also able to avoid explicit support for auxiliary data structures such as lists and maps since these are available in Scala.

One aspect of language embedding that was not explicitly used in the Kiama-Apply compiler, but is worth mentioning, is the opportunity for language extension. It is easy for a programmer to add new primitives to the rewriting language since they are just normal Scala definitions. Extending Stratego would require adding new syntax definitions, semantic checks and translations to its compiler, a much bigger undertaking. On the other hand, perhaps the ability to extend the language is not desirable, since arbitrary extensions can obscure the meaning of a program.

IMPLEMENTATION COMPARISON

Having discussed linguistic differences, we now consider the two Apply compiler implementations. We discuss the size of the two implementations, as well as compare their run-time performance in terms of speed and memory usage.

It is important to realize that our aim is not to conduct a detailed low-level performance comparison of the two implementations. Such a comparison is largely meaningless for analyzing the practicality of embedding, since so much of the performance is influenced by the implementations of the Java and Scala compilers and the optimizations performed by the Java run-time. Also, while neither of the implementations exhibits major bottlenecks, both could be further optimized, and that would change any detailed measurements.

Instead, our study aims to determine whether the embedding approach is practical by comparing it with the external implementation on as level a playing field as possible. We are concerned with overall trends rather than low-level details. In particular, we are interested in scalability. Internal DSLs often perform well enough for small programs but cope less well with large ones, since they don't have the benefit of domain-specific optimization.

Experimental Setup

We compare the implementations running on the Java Virtual Machine (JVM) in byte-code interpretation mode. Table 1 summarizes the software and hardware versions used in the experiments.

Stratego has a native compiler (via C), but Scala's main implementation is for the JVM, so we use the Stratego to Java compiler, and the Java version of the ATerm library. The Java version of Stratego is younger than the native code implementation but it is used extensively in the Spoofax Language Workbench (Kats, 2010) so its performance is sufficiently optimized to provide a good baseline.

Performance metrics were captured using YourKit, a professional profiling tool (YourKit, 2011). This profiler supports accurate instrumentation through the use of the JVMTI API. Of the metrics captured, our interest is in the CPU processing time and memory requirements to complete a syntax tree rewrite. We report CPU time instead of wall clock time because CPU time measurements are usually more accurate.

Table 1. Software and hardware versions used in the experiments

Stratego/XT	0.17 (Java backend)
Kiama	1.2 with Scala 2.9.1
Java	1.6 (build 26, 64-bit)
YourKit	10.0.0
Experimental machine	Quad-Core Intel Xeon 2.66, 10GB memory, Mac OS X 10.7.1

Memory was measured by the program's consumption of the JVM heap; specifically, we calculated the retained size of the applications object graph with the class loader as the root object. The retained size is a measure of both strong references and weak references; unreachable objects and objects on the finalization queue are not included in the byte count.

Any good JVM implementation comes with various optimizations built in. Byte-code interpretation mode was invoked using the `-Xint` flag so that the measurements are not influenced by optimization opportunities or overhead of just-in-time compilation.

Any measurement will include some perturbation due to fluctuations in the measurement environment. To minimize the impact of such variations each test was run a few hundred times from which a statistical average was obtained. For brevity only the averages are presented.

Code Size

Table 2 shows that the programs of the Apply language compiler are not trivial by listing the non-blank, non-commented line counts for the individual compiler phases and other categories of code. The Support category includes code for tasks not directly related to rewriting, such as command-line processing. Most Kiama-Apply phases are larger than their Stratego-Apply counterparts. There are two to three lines of Scala code for each Stratego one line rule due mostly to the coding style, lack of concrete syntax, and extra code required to maintain strong typing guarantees. A small extra amount of code is present in the Kiama-Apply modules to provide interfaces for our test harness. The code size increase for Optimize is greater than for the other phases because this category aggregates a number of smaller optimization modules, each of which contributes code size increases as described.

Table 2. Non-blank, non-commented lines of code (LOC) in the Apply compiler phases and support categories

	Stratego-Apply (LOC)	Kiama-Apply (LOC)
De-sugar	30	53
Do-Apply	54	119
Optimize	588	985
Environment	416	467
C language	539	503
Support	222	311
Signatures	126	1737

The Support category includes code for tasks not directly related to rewriting, such as command-line processing. The Signatures category includes code that defines the term structures. Kiama-Apply is much larger in this category since we also include the code that translates ATerms into Scala values and vice-versa. A normal use of Kiama's rewriting library would not use ATerms at all, so this code would not be required.

A Typical Compilation

As discussed in the section Apply and Its Compilers, the Apply compilers are composed of a multi-phase pipeline (Figure 2). To begin our analysis, we present a scenario demonstrating how long a typical compilation takes in both Stratego-Apply and Kiama-Apply. In this example we consider the Apply implementation of the example Sobel edge detection algorithm.

Table 3 presents the timings for each transformation phase during the compilation of the Sobel module with both the Stratego-Apply and Kiama-Apply implementations. Our timings do not include measurements for parsing and the pretty printing phases because these phases are not primarily concerned with syntax tree rewrites.

Table 3. Compilation times and memory use when processing the Sobel edge detection algorithm

	Stratego-Apply		Kiama-Apply	
	Time (ms)	Memory (MB)	Time (ms)	Memory (MB)
De-sugar (1st time)	335	1.88	499	2.74
Optimize (1st time)	955	2.39	1175	2.96
Do-Apply	419	1.96	642	2.80
Environment	491	2.07	1054	2.83
De-sugar (2nd time)	380	1.93	572	2.77
Optimize (2nd time)	1401	2.46	3162	3.05
Total	≈ 4 seconds		≈ 7 seconds	

Each transformation phase is an independent module in Stratego-Apply and an independent class in Kiama-Apply. A phase reads in an ATerm, transforms it, and writes the transformed term out to disk to be used as input for the next phase. (The same ATerm library is used by the two implementations.) This approach is useful for compiler development and debugging. However in a production compiler, syntax trees would be passed internally from one phase to the next, with the input/output component only appearing at the start and end of the pipeline. For this reason, our measurements *exclude* the input and output components of the processing.

From this data we see that an end-to-end transformation in Stratego-Apply took approximately four seconds, whereas Kiama-Apply took approximately seven seconds. Thus the Kiama-Apply implementation is slower by a factor of approximately 40%. In addition the Kiama-Apply implementation consumes about 26% more memory. These performance measurements are typical of what we observed in other similar experiments

and show that Kiama-Apply, while slower than Stratego-Apply, is still suitable for regular use. In the rest of this section we investigate how the performance scales as the input Apply program size varies and how it compares when rewriting is isolated from traversal.

Experimental Input

Due to a limited supply of large Apply programs, for the set of experiments investigating performance in more detail, we used a code generator to produce the input text of programs. The generator produces programs that perform image manipulations similar to the steps of the Sobel example shown earlier. Command-line options can be used to control the number of operations and the processing window size. Increasing the number of operations increases the width of the syntax tree, while increasing the window size increases the complexity of the processing statements and hence the depth of the tree. Thus, we can vary the input size to examine the effect of scale on the performance of the two implementations. The remaining experiments use test data generated by incrementing the window size in steps of two and operations in steps of five. In the experiments we report the syntax tree sizes in terms of number of nodes, which explains the non-obvious X-axis labels.

Experiment 1: Memory Use as Tree Size Varies

Recall that the "C Language" phase of the Apply compilers is responsible for translating the compiler's intermediate representation into an abstract syntax tree that represents the generated C program. For our first experiment we ran the C Language phases of the Stratego-Apply and Kiama-Apply compilers on a variety of syntax trees to measure their memory usage (Figure 4). The C Language phase was chosen because it exercises the rewriting machinery in a significant way, visiting all parts of the AST. The performance reported for this phase is also typical of the other phases. The syntax trees in this experiment are of increasing width, achieved by increasing the window size in the Apply code generator. As is to be expected, we observed that both compilers' memory consumption increased as the size of the syntax tree increased. The rate of memory consumption growth for Kiama-Apply was less than for Stratego-Apply. In addition, Stratego-Apply consumed less memory than Kiama-Apply for small to medium sized trees but more for large trees. The crossover point appears at around 50,000 tree nodes.

Our profiling results reveal that in the first data set from Figure 4 the ATerms occupied 9% of Stratego-Apply's memory usage compared to 45% for Kiama-Apply. However, for the last data set where the trees were at their largest the ATerms occupied close to 60% of the heap memory for both Stratego-Apply and Kiama-Apply. It is interesting to note that before the ATerms were read, the C Language phases in Stratego-Apply and Kiama-Apply occupied around the same amount of memory (approximately 1.7MB).

From this experiment we conclude that Kiama-Apply uses a reasonable amount of memory, particularly for large trees. We have not precisely identified the reason for the increasing memory overhead for Stratego-Apply. Preliminary analysis indicates that it may be due to the caching approach used by the ATerm library to avoid duplication.

Experiment 2: Execution Time when Varying the Width of the Tree

This experiment compares the CPU processing time for the C Language phase on syntax trees with fixed depth and varying widths (Figure 5). Increasing the number of operations for our code generator varied the widths of the trees.

Figure 5 shows that, for a fixed tree depth, the execution time is a linear function of the tree width for both Stratego-Apply and Kiama-Apply. We also observe that for trees with relatively small widths the performance of Stratego-Apply and Kiama-Apply was similar. However, as the width of the tree increased, Kiama-Apply's execution time increased more rapidly than Stratego-Apply's. In ATerms, these wider trees are represented by longer arrays at each node of the tree. A longer array represents more terms that undergo pattern matching and possible rewriting. From this we observe that pattern matching and term rewriting takes longer in the internal language. Overall, Kiama-apply is slower than Stratego-apply but scales in the same way.

Figure 4. C Language phase memory usage as tree size varies: X-axis: tree size in number of nodes; Y-axis: memory retained size in bytes

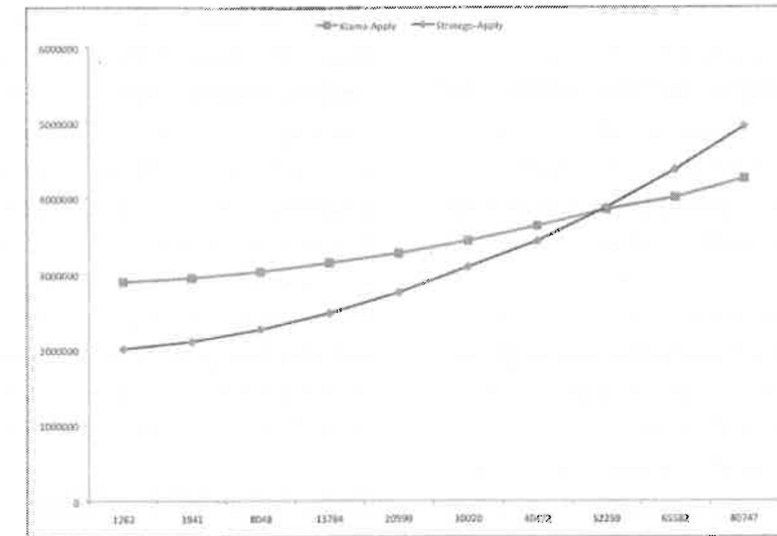
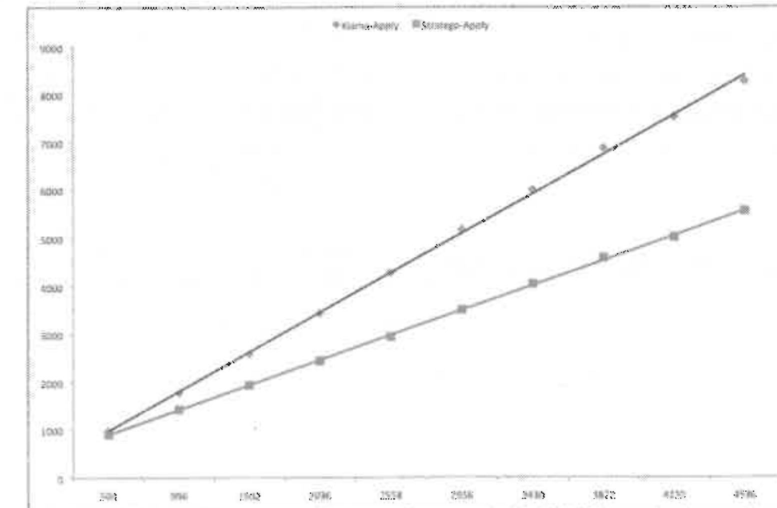


Figure 5. C Language phase CPU time as tree width varies: X-axis: tree width in nodes; Y-axis: CPU time in milliseconds



Experiment 3: Execution Time when Varying the Height of the Tree

This experiment compares the execution time of the two Desugar phases on trees of varying height and consistent widths. Desugar was chosen for this experiment because it performs a significant number of rewrites. Increasing the processing window size for our code generator varied the heights of the trees.

From Figure 6 we observe that the execution time is an exponential function of the tree height for both Stratego-Apply and Kiama-Apply. However, the base of the exponential function is greater for Kiama-Apply. The primary traversal strategy used in this part of the experiment was the `topdown` traversal strategy, which is expressed in Kiama as

```
s <* all(topdown(s))
```

The `all` strategy iterates each sub-term of a subject term applying a strategy as it visits each subject term. This strategy is recursive and so navigates the multidimensional structures by repeating the `all` for each nested structure. This implementation explains the exponential results we see in Figure 6.

Kiama-Apply spends approximately 50% of total CPU time in traversal when rewriting. In contrast, Stratego-Apply spends much of its time building its cache to produce the optimized construct necessary for the tree rewrite. This highlights a fundamental difference in architecture between Kiama and Stratego in the way terms are stored and accessed. Where Stratego spends more time upfront to construct an internal term representation that then gives it a quick access mechanism for traversal, Kiama spends less time constructing such structures at the cost of a slower traversal.

Experiment 4: Isolating Rewrite Times

The previous experiment pointed to a difference in rewriting times in the two implementations. To get more insight into this difference, we decompose the total execution time of a transformation into the time spent traversing the tree (t) and the time spent rewriting parts of the tree (r). This decomposition is only valid for one-pass traversals but not fixed-point traversals (Visser, 2004). To measure the transformation time (t) the rewrites in Desugar were temporarily disabled during this experiment. The results presented reveal that rewrite times in Stratego-Apply are quicker than that of Kiama-Apply by a factor of around five.

Figure 6. Desugar phase CPU time as tree height varies: X-axis: maximum tree depth in steps from root; Y-axis: CPU time in milliseconds

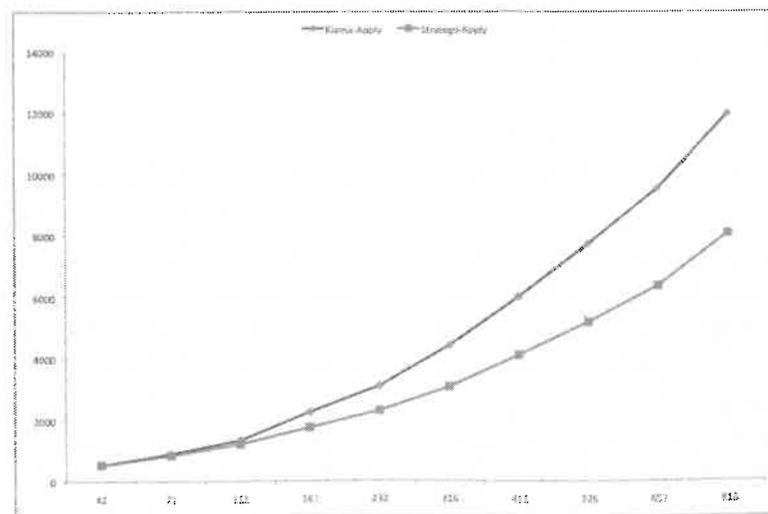
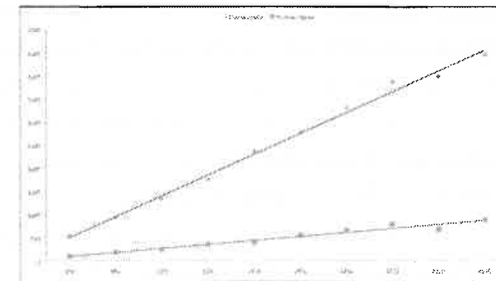


Figure 7. CPU time for rewrites as tree width varies: X-axis: tree width in nodes; Y-axis: CPU time in milliseconds



From Figure 7 we observe that as the width of the tree increases the rewrite times in both Stratego-Apply and Kiama-Apply increase linearly. We expected to see the implementation specifically designed for rewriting to out-perform one based on a general-purpose language without any special support for rewriting. Figure 7 does show this difference, since Kiama-Apply is quite a bit slower than Stratego-Apply. However, the Kiama-Apply performance is not unrealistic and we see a similar linear scaling pattern, so Kiama remains usable even at large tree sizes.

SUMMARY

As we expected, when a complete rewriting phase is measured Kiama-Apply is slower than Stratego-Apply and uses more memory. However, the performance is sufficiently good for most applications. Our more detailed experiments on synthetic input showed that the scaling behavior of the two implementations is similar. Therefore, we have confidence that Kiama's rewriting library is feasible even for large inputs. Performance of the library is likely to get better as the Scala compiler performs more optimization and the library is improved.

FUTURE RESEARCH DIRECTIONS

Future research directions flowing from the work presented in this chapter can be divided into two main topics: general research into DSL implementation via case studies similar to the one present here, and continued investigation of the term rewriting domain.

We strongly encourage other researchers to conduct side-by-side evaluations of the kind we have presented here. These evaluations can be time-consuming and may seem pointless to some degree, since an existing DSL is being duplicated. However, the true benefits and pitfalls of an implementation approach cannot really be appreciated until a proper evaluation is performed, so it is important that language researchers conduct them. Although we have focused on an embedding approach in this chapter, side-by-side comparison can be of benefit for any implementation method.

This chapter has presented an analysis of the two language implementations along various dimensions that made sense for this domain and implementation approach. It would be interesting to generalize these dimensions to develop a standard approach to language comparison that will help researchers structure their experiments. Further down the line, it is desirable to try to develop general criteria for assessing the suitability of different DSL implementation approaches, for which a starting point is presented in Kosar *et al* (2008). This task is non-trivial since the characteristics of DSLs and implementation approaches differ greatly. For example, even within the general embedding approach there are many different ways to implement a DSL, ranging from the pure approach used in this chapter to a more translation-based approach where an intermediate form is created. Capturing the advantages and disadvantages of all of these approaches in

a succinct, comprehensible, general way is an important topic for future research.

In the rest of this section we outline some areas where the Kiama rewriting library could be extended, in some cases to remedy deficiencies compared to Stratego, and in others to go beyond. First and most simply, researchers have developed type systems for strategic rewriting (Lämmel, 2003). We expect that those systems could be instantiated in Scala's type system to provide more static assurances about the correctness of Kiama rewriting programs.

Concrete Syntax

Stratego provides an extremely useful facility for expressing patterns and term construction using the concrete syntax of the language(s) that are being processed (Bravenboer, 2008). Concrete syntax support means that verbose prefix constructor terms can be avoided, making the rules much easier to write and understand. Concrete syntax support is achieved in Stratego by a meta level of processing that combines the Stratego grammar with the grammar for the concrete syntax fragments. The resulting parser is able to process the input text to replace the concrete syntax with equivalent prefix term syntax.

Kiama does not have any support for concrete syntax at the moment. We plan to build a limited form of it as a plug-in for the Scala build tool. We will recognize concrete syntax fragments and pass them to a user-specified processor for parsing and pretty printing. The resulting code will be inserted into the Scala code for regular processing. Because the plug-in will not understand Scala syntax, the interaction between the code fragments and Scala code will be limited. Nevertheless, it should be sufficient for most purposes and keeping it separate from the rewriting library will mean that this facility can be used for other syntax tree matching and manipulation such as within tree decoration written using Kiama's attribute grammar library (Sloane, 2011).

Dynamic Rules

Stratego has extensive support for dynamic rewrite rules (Bravenboer, 2005). The basic idea is that rules can be constructed, manipulated and removed during rewriting. This facility is typically used to record context-specific information that will be used in some part of a term traversal and then "forgotten" as the traversal proceeds.

We have developed custom support for dynamic rules for Kiama-Apply, since we aimed to duplicate Stratego-Apply as closely as possible. We plan to revisit this code to see whether it can be usefully generalized and included in Kiama. At that time we will carefully consider whether applications of dynamic rules could more consistently be achieved using other mechanisms, such as by using Kiama's attribute grammar library to decorate terms with context-specific information. Nevertheless, we expect that having some form of dynamic rules will be useful.

Congruences

A Stratego congruence is a strategy that is named after a constructor. It provides a convenient syntactic sugar for matching on the constructor to extract the sub-terms, processing each sub-term with a potentially different strategy, and assembling the results. For example, if `Add` is a constructor with two sub-terms, then the congruence `Add` (s_1, s_2) first matches the subject term against the pattern `Add` (t_1, t_2), then applies s_1 to t_1 and s_2 to t_2. If both applications succeed then the results are combined as the sub-terms of a new `Add` term.

Kiama has some support for congruences but can't quite emulate the concise Stratego notation without some boilerplate. Since we are using a pure embedding, the patterns we need cannot be synthesized from the constructor definitions. Therefore we need boilerplate for each constructor that matches the term and extracts the sub-terms. The sub-terms are then passed to a generic Kiama routine that applies the individual strategies and constructs the new term, if appropriate. We do not currently use this congruence support in Kiama-Apply. Some strategies would be simplified if we did. We also don't have congruence support for data structures such as lists. It should be relatively easy to add using the high-level interfaces provided by Scala collections.

CONCLUSION

We have evaluated the effectiveness of an internal implementation of a domain-specific language by comparing it to an existing external implementation that performs the same task. A non-trivial compilation case study provided the context for a realistic comparison. We saw that the two languages were very similar, largely due to the power of the Scala host language, particularly its ability to define domain-specific operators. Differences in name binding resulted from our desire to reuse Scala's powerful pattern matching facilities. Scala's stronger type system required more discipline in the construction of terms.

A comparison of the two implementations of the case study compiler showed that their sizes were similar, but the internal version was slower and used more memory. This difference is to be expected, since the internal version is running on a generic run-time whereas the external one uses a run-time designed specifically for term rewriting. Nevertheless, the internal language is still practically useful since its performance is close enough to the external language and scales in the same way on large inputs. The performance deficiency is somewhat offset by a simpler implementation.

We hope that this study inspires other researchers to conduct similar evaluations. The methodology used here should translate into other settings without too much trouble. Issues such as expression syntax and binding constructs will be common to other DSLs, but will induce different solutions in different host languages. More case studies will allow the relative advantages of different host languages to be compared.

ACKNOWLEDGMENT

Much of the design and implementation work for Kiama's rewriting library was performed while Sloane was visiting the Delft University of Technology, supported by NWO project 040.11.001, Combining Attribute Grammars and Term Rewriting for Programming Abstractions. We thank Eelco Visser and Lennart Kats for providing their Stratego expertise. YourKit, LLC generously provided an open source license to their Java profiler.

REFERENCES

Ada. (1982). *Reference manual for the Ada programming language.* MIL-STD 1815 edition, U.S. Department of Defense.

Bravenboer, M. (2008). *Exercises in free syntax. Syntax definition, parsing, and assimilation of language conglomerates.* PhD. Thesis, Utrecht University.

Bravenboer, M., van Dam, A., Olmos, K., & Visser, E. (2005). Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae, 69*, 1–56.

Fowler, M., & Parsons, R. (2010). *Domain specific languages.* Addison-Wesley Professional.

Ghosh, D. (2011). *DSLs in action.* Manning.

Hamey, L. G. C. (2007). Efficient image processing with the Apply language. *Proceedings of International Conference on Digital Image Computing: Techniques and Applications* (pp. 533-540). IEEE Computer Society Press.

Hamey, L. G. C., & Goldrei, S. N. (2008). Implementing a domain-specific language using Stratego/XT: An experience paper. *Electronic Notes in Theoretical Computer Science, 203*(2), 37–51. doi:10.1016/j.entcs.2008.03.043

Hamey, L. G. C., Webb, J. A., & Wu, I.-C. (1987). Low-level vision on Warp and the Apply programming model. In Kowalik, J. (Ed.), *Parallel computation and computers for artificial intelligence* (pp. 185–199). Kluwer Academic Publishers. doi:10.1007/978-1-4613-1989-4_10

Hamey, L. G. C., Webb, J. A., & Wu, I.-C. (1989). An architecture independent programming language for low-level vision. *Computer Vision Graphics and Image Processing, 48*, 246–264. doi:10.1016/S0734-189X(89)80040-4

Kats, L. C. L., & Visser, E. (2010). The Spoofax language workbench. *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (pp. 444-463). ACM Press.

Kosar, T., Martínez López, P. E., Barrientos, P. A., & Mernik, M. (2008). A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology, 50*(5), 390–405. doi:10.1016/j.infsof.2007.04.002

Kosar, T., Mernik, M., & Carver, J. C. (2012). Program comprehension of domain-specific and general-purpose languages: Comparison using a family of experiments. *Empirical Software Engineering, 17*(3), 276–304. doi:10.1007/s10664-011-9172-x

Kosar, T., Oliveira, N., Mernik, M., Pereira Varanda, J. M., Črepinšek, M., Da Cruz, D., & Henriques, R. P. (2010). Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems, 7*(2), 247–264. doi:10.2298/CSIS1002247K

Lämmel, R. (2003). Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming, 54*, 1–64. doi:10.1016/S1567-8326(02)00028-0

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *Computing Surveys, 37*(4), 316–344. doi:10.1145/1118890.1118892

Odersky, M., Spoon, L., & Venners, B. (2008). *Programming in Scala*. Artima Press.

Premaratne, S. (2011). *Strategic programming approaches to tree processing*. Master's thesis, Macquarie University.

Sloane, A. M. (2011). Lecture Notes in Computer Science: *Vol. 6491. Lightweight language processing in Kiama. Generative and Transformational Techniques in Software Engineering III* (pp. 408–425). Springer-Verlag. doi:10.1007/978-3-642-18023-1_12

Sloane, A. M., Kats, L., & Visser, E. (in press). A pure embedding of attribute grammars. *Science of Computer Programming*. doi:doi:10.1016/j.scico.2011.11.005

Sloane, A. M., Kats, L. C. L., & Visser, E. (2010). A pure object-oriented embedding of attribute grammars. *Electronic Notes in Theoretical Computer Science, 253*(7), 205–219. doi:10.1016/j.entcs.2010.08.043

van den Brand, M. G. J., de Jong, H. A., Klint, P., & Oliver, P. A. (2000). Efficient annotated terms. *Software, Practice & Experience, 30*(3), 259–291. doi:10.1002/(SICI)1097-024X(200003)30:3<259::AID-SPE298>3.0.CO;2-Y

van den Brand, M. G. J., & Klint, P. (2007). ATerms for manipulation and exchange of structured data: It's all about sharing. *Journal of Information and Software Technology, 49*(1), 55–64. doi:10.1016/j.infsof.2006.08.009

van Deursen, A., Klint, P., & Visser, J. (2000). Domain-specific languages: An annotated bibliography. *SIGPLAN Notices, 35*(6), 26–36. doi:10.1145/352029.352035

Visser, E. (2004). Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. *Proceedings of the Domain-Specific Program Generation Workshop, Lecture Notes in Computer Science Vol. 3016,* (pp. 216-238). Springer-Verlag.

Visser, E. (2005). A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation, 40*, 831–873. doi:10.1016/j.jsc.2004.12.011

Visser, E. (2006). Stratego/XT 0.16: components for transformation systems. *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation* (pp. 95-99). ACM Press.

YourKit. (2011). Retrieved from http://www.yourkit.com/

ADDITIONAL READING

Baader, F., & Nipkow, T. (1998). *Term rewriting and all that*. Cambridge University Press.

Castleman, K. R. (1996). *Digital image processing*. Prentice-Hall.

Dubochet, G. (2006). On embedding domain-specific languages with user-friendly syntax. *Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development* (pp. 19-22).

Gonzales, R. C., & Woods, R. E. (2006). *Digital image processing*. Prentice-Hall.

Hofer, C., Ostermann, K., Rendel, T., & Moors, A. (2008). Polymorphic embedding of DSLs. *Proceedings of the 7th International Conference on Generative Programming and Component Engineering* (pp. 137-148). ACM Press.

Hudak, P. (1998). Modular domain specific languages and tools. *Proceedings of the 5th International Conference on Software Reuse* (pp. 134-142). IEEE Computer Society Press.

Jain, A. (1989). *Fundamentals of digital image processing*. Prentice-Hall.

Kats, L., Sloane, A. M., & Visser, E. (2009). Decorated attribute grammars: Attribute evaluation meets strategic programming. *Proceedings of the International Conference on Compiler Construction, Lecture Notes in Computer Science Vol. 5501,* (pp. 142-157). Springer-Verlag.

Kats, L., & Visser, E. (2010). The Spoofax language workbench: Rules for declarative specification of languages and IDEs. *Proceedings of the 2010 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications,* (pp. 444-463).

Klint, P., van der Storm, T., & Vinju, J. (2011). Lecture Notes in Computer Science: *Vol. 6491. EASY meta-programming in Rascal. Generative and Transformational Techniques in Software Engineering III* (pp. 222–289). Springer-Verlag. doi:10.1007/978-3-642-18023-1_6

Martí-Oliet, N., & Meseguer, J. (2002). Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science, 285*(2), 121–154. doi:10.1016/S0304-3975(01)00357-7

Moors, A., Rompf, T., Haller, P., & Odersky, M. (2012). Scala-virtualized. *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (pp. 117-120). ACM Press.

Sonka, M., Hlavac, V., & Boyle, R. (1998). *Image processing, analysis, and machine vision*. Brooks/Cole.

Ureche, V., Rompf, T., Sujeeth, A., Chafi, H., & Odersky, M. (2012). StagedSAC: A case study in performance-oriented DSL development. *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (pp. 73-82). ACM Press.

Visser, E. (2008). WebDSL: A case study in domain specific language engineering. In *Generative and Transformational Techniques in Software Engineering II* (*Vol. 5235*, pp. 291–376). Lecture Notes in Computer Science Springer-Verlag. doi:10.1007/978-3-540-88643-3_7

KEY TERMS AND DEFINITIONS

Apply: An image processing domain-specific language based on pixel-centered operations that are applied consistently across an image, automatically taking into account side conditions such as image edges. Implemented in Stratego as an optimizing compiler that produces C code.

Generic Traversal Strategy: A strategy that describes how to traverse to sub-terms of the term being rewritten, independently of the particular structure being traversed.

Internal DSL: A domain-specific language that is embedded as a library in a host general-purpose programming language.

Kiama: A language-processing library built by embedding various formalisms into the Scala programming language. Includes a strategic term rewriting component whose design is based on the Stratego language.

Language Embedding: An approach to language implementation where the syntax and semantics of another host general-purpose language are reused.

Name Binding: The association of a name with a program entity, such as in a definition of a value or method, or as a result of a successful pattern matching operation.

Pattern Matching: An operation that compares a pattern against a piece of data. A match either succeeds, possibly binding some names to parts of the data, or fails.

Pure Internal DSL: An internal DSL where the embedding is performed without any extra translation step. In other words, DSL programs are compiled and executed purely as host language programs.

Rewrite Rule: A rewriting specification that matches a pattern against the term being rewritten and either succeeds, constructing a new term, or fails.

Scala: A modern object-oriented and functional programming language whose main implementation targets the Java Virtual Machine.

Strategic Term Rewriting: A style of term rewriting where the application of rewrite rules is controlled by high-level strategies rather than by a fixed scheme.

Stratego/XT: A widely used strategic term rewriting language and associated tools. Stratego has both a compiler that produces native code via C, and a compiler that produces Java code.

Strategy: A rewriting specification that generalizes rewriting rules by allowing a combination of pattern matching, term construction, rules and choice. Rewriting is guided by the success or failure of the component operations.

Term Rewriting: A transformation process where tree-structured data (or terms) are rewritten by rules that pattern match on term structure.

Chapter 5
Comparison Between Internal and External DSLs via RubyTL and Gra2MoL

Jesús Sánchez Cuadrado
Universidad Autónoma de Madrid, Spain

Javier Luis Cánovas Izquierdo
École des Mines de Nantes – INRIA – LINA, France

Jesús García Molina
Universidad de Murcia, Spain

ABSTRACT

Domain Specific Languages (DSL) are becoming increasingly more important with the emergence of Model-Driven paradigms. Most literature on DSLs is focused on describing particular languages, and there is still a lack of works that compare different approaches or carry out empirical studies regarding the construction or usage of DSLs. Several design choices must be made when building a DSL, but one important question is whether the DSL will be external or internal, since this affects the other aspects of the language. This chapter aims to provide developers confronting the internal-external dichotomy with guidance, through a comparison of the RubyTL and Gra2MoL model transformations languages, which have been built as an internal DSL and an external DSL, respectively. Both languages will first be introduced, and certain implementation issues will be discussed. The two languages will then be compared, and the advantages and disadvantages of each approach will be shown. Finally, some of the lessons learned will be presented.

INTRODUCTION

Software applications are normally written for a particular activity area or problem domain. When building software, developers have to confront the semantic gap between the problem domain and the conceptual framework provided by the software language used to implement the solution. They must express a solution based on domain concepts using the constructs of a general purpose programming language (GPL), such as Java or C#, which typically leads to repetitive and error

DOI: 10.4018/978-1-4666-2092-6.ch005

Copyright © 2013, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.