Abstractions for Language-Independent Program Transformations

Karl Trygve Kalleberg

Thesis for the degree of Philosophiae Doctor (PhD) at the University of Bergen

2007-05-11

ISBN 978-82-308-0441-4
Bergen, Norway 2007
Copyright Karl Trygve Kalleberg
Produced by: Allkopi Bergen

Abstractions for Language-Independent Program Transformations

Karl Trygve Kalleberg
Department of Informatics

Thesis for the degree of Philosophiae Doctor (PhD) at the University of Bergen

2007-05-11

Contents

Acknowledgements xi

I Introduction 1

1 Introduction 3
- 1.1 Software Evolution 4
- 1.2 Program Transformation 4
  - 1.2.1 Strategic Programming 5
- 1.3 Program Models 6
- 1.4 Language Abstractions for Program Transformations 7
  - 1.4.1 Extensible Languages 8
- 1.5 Method 9
- 1.6 Contributions 9
- 1.7 Outline 10
- 1.8 Summary 12

II Software Transformation Systems 13

2 Programmable Software Transformation Systems 15
- 2.1 Software Transformation Systems 15
  - 2.1.1 Anatomy of a Transformation System 16
  - 2.1.2 Features of Software Transformation Systems 17
- 2.2 Feature Models 18
- 2.3 Program Representation 19
  - 2.3.1 Runtime Representation 20
  - 2.3.2 Storage Representation 30
- 2.4 Transformation Language 32
  - 2.4.1 Organisation 32
  - 2.4.2 Transformation Atoms 36
  - 2.4.3 Typing 47
- 2.5 Discussion 49
- 2.6 Summary 50

3 Strategic Term Rewriting 51
- 3.1 Term Rewriting 51
  - 3.1.1 Algebraic Signatures and Language Signatures 51
  - 3.1.2 Patterns and Terms 53

3.1.3 Rewrite Rules 55
3.1.4 Rewriting Strategies 55
3.2 System S – Strategic Term Rewriting 56
3.2.1 Primitive Operators and Strategy Combinators 58
3.2.2 Primitive Traversal Strategies 59
3.2.3 Building and Matching Terms 61
3.2.4 Variable Scoping 61
3.2.5 Rewrite Rules 62
3.2.6 Additional Constructs 62
3.3 Stratego 63
3.3.1 Signatures, Patterns and Terms 63
3.3.2 Congruences 65
3.3.3 Scoped, Dynamic Rules 65
3.3.4 Overlays 67
3.3.5 Modules 67
3.3.6 Stratego/XT 67
3.4 Summary 68

III Abstractions for Language Independence 69

4 Program Object Model Adapters 71
4.1 Introduction 71
4.2 The Program Object Model Adapter 73
4.2.1 Architecture Overview 73
4.2.2 Design Overview 74
4.3 Implementation 81
4.3.1 Term Interface 81
4.3.2 Design Considerations 86
4.4 Related Work 88
4.5 Discussion 89
4.6 Summary 90

5 Modularising Cross-Cutting Transformation Concerns 91
5.1 Introduction 91
5.2 Constant Propagation 92
5.3 AspectStratego 95
5.3.1 Joinpoints 96
5.3.2 Pointcuts 96
5.3.3 Advice 98
5.3.4 Cloning 99

5.3.5 Weaving . . . 99
5.3.6 Modularisation . . . 100
5.4 Case Studies . . . 100
5.4.1 Logging . . . 101
5.4.2 Type Checking . . . 102
5.4.3 Extending Algorithms . . . 103
5.5 Implementation of the Weaver . . . 109
5.5.1 A Weaving Example . . . 111
5.5.2 Aspects as Meta Programs . . . 112
5.6 Discussion . . . 112
5.7 Summary . . . 114

IV Supportive Abstractions for Transformations 115

6 An Extensible Transformation Language 117
6.1 An Extensible Compiler . . . 117
6.1.1 Declaring Syntax and Assimilator . . . 117
6.1.2 Language Extensions . . . 119
6.1.3 Compiler Pipeline . . . 119
6.1.4 StrategoCore . . . 121
6.2 An Extensible Runtime . . . 122
6.2.1 Design . . . 122
6.2.2 Implementation . . . 123
6.2.3 Performance . . . 123
6.3 Light-Weight and Portable Transformation Components . . . 124
6.3.1 Transformlets . . . 124
6.3.2 Implementation . . . 125
6.4 Summary . . . 125

7 Strategic Graph Rewriting 129
7.1 Abstract . . . 129
7.2 Introduction . . . 130
7.3 Extending Term Rewriting Strategies to Term Graphs . . . 131
7.3.1 Term Rewriting Strategies . . . 131
7.3.2 References . . . 133
7.3.3 Rewrite Rules and References . . . 134
7.3.4 Term Graph Traversal . . . 135
7.4 From Terms to Term Graphs . . . 137
7.4.1 Use-Def Chains . . . 137
7.4.2 Call Graphs . . . 139

7.4.3 Flow Graphs . . . 140
7.5 Graph Algorithms and Applications . . . 141
7.5.1 Finding Mutually Recursive Functions . . . 142
7.5.2 Lazy Graph Loading . . . 142
7.6 Implementation . . . 143
7.7 Related Work . . . 144
7.8 Discussion and Further Work . . . 145
7.9 Conclusion . . . 145

V Case Studies 147

8 Language Extensions as Transformation Libraries 149
8.1 Abstract . . . 149
8.2 Introduction . . . 150
8.3 The Alert DSAL . . . 151
8.3.1 The TIL Language . . . 152
8.3.2 Alert Declarations and Handlers . . . 152
8.4 Implementation of TIL+Alert . . . 155
8.4.1 DSAL = library + notation . . . 155
8.4.2 Type Checking . . . 156
8.4.3 Alert Weaving . . . 157
8.4.4 Coordination . . . 160
8.5 Discussion . . . 160
8.5.1 Program Transformation . . . 160
8.5.2 Program Transformation Languages for Aspect Implementation . . . 162
8.5.3 Related Work . . . 164
8.6 Conclusion . . . 165
8.7 TIL Grammar . . . 166

9 Interactive Transformation and Editing Environments 167
9.1 Introduction . . . 167
9.2 Core Functionality . . . 169
9.2.1 Architecture . . . 169
9.2.2 Build Weave . . . 170
9.2.3 Project Rebuilding . . . 170
9.3 Editor . . . 171
9.3.1 Content Completion . . . 171
9.3.2 Syntax Highlighting . . . 171
9.3.3 Parenthesis Highlighter . . . 172

9.3.4 Outline 172
9.3.5 Source Code Navigation 173
9.3.6 Build Console 173
9.4 Scripting 174
9.4.1 Script View 174
9.4.2 Script Console 174
9.4.3 Analysing and Transforming Source 176
9.4.4 Transformation Hooks 176
9.5 Implementation 176
9.6 Related Work 177
9.7 Summary 178

10 Extending Compilers with Transformation and Analysis Scripts 179
10.1 Introduction 179
10.2 Scriptable Domain-Specific Analysis and Transformation 181
10.2.1 Architecture 182
10.2.2 MetaStratego as a Scripting Engine 183
10.3 Examples of Domain Support Scripting 184
10.3.1 Project-Specific Code Style Checking 184
10.3.2 Custom Data-Flow Analysis 188
10.3.3 Domain-specific Source Code Transformations 190
10.4 Implementation 191
10.4.1 Analysis Architecture 191
10.4.2 Transformlet Repositories 192
10.5 Related work 192
10.6 Discussion 193
10.7 Summary 194

11 Code Generation for Axiom-based Unit Testing 195
11.1 Introduction 196
11.2 Expression Axioms in Java 197
11.2.1 JUnit Assertions 197
11.2.2 Java Specification Logics 198
11.3 Structuring the Specifications 201
11.3.1 Associating Axioms with Types 202
11.3.2 Optional and Inherited-only Axioms 204
11.4 Java API caveats 206
11.4.1 Override and Overload 206
11.4.2 `clone` and Other Protected Methods 207
11.4.3 The `equals` “congruence” relation 208
11.5 Testing 208

11.5.1 Test Data Generator Methods 209
11.5.2 Determining Test Set Quality 210
11.5.3 Running the Tests 210
11.5.4 Interpreting Test Results 211
11.6 Test Suite Generation 211
11.6.1 Generating Tests from Axioms 212
11.6.2 Organising Generated Tests 215
11.6.3 Executing Tests 215
11.7 Implementation 216
11.8 Discussion and Related Work 217
11.9 Summary 218

VI Conclusion 221

12 Discussion 223
12.1 Techniques for Language Independence 223
12.1.1 Abstracting over Data 223
12.1.2 Expressing Generic Algorithms 224
12.1.3 Adapting Generic Algorithms 225
12.1.4 Modular Language Descriptions 225
12.2 Other Approaches to Software Evolution 226
12.3 Availability of Research Systems 226

13 Further Work 227

14 Conclusions 229

15 Summary 231

Acknowledgements

Most dissertation prefaces start with a sentence saying that the work would not have been possible if it had not been for the supervisor(s). That's the third sentence in this preface. If it had not been for Magne Haveraaen and Eelco Visser, this dissertation would not exist. I am extremely grateful to both for allowing me to play around with what I consider to be really fun stuff for the last three and half years – and getting paid for it! The different styles of Eelco and Magne have been a source of inspiration, and sometimes a little bit of frustration. In some odd way, it's like having two parents who don't always agree on certain parts of your upbringing. Magne's socractic style of teaching and counseling together with his insistence on thorough (algebraic!) analysis before anything else has taught me to think a lot harder about the assumptions I would normally make without noticing. Eelco's style of exploration and research through construction has in many ways reinforced my natural tendency to build things in order to understand how they work. The two styles can certainly be made to mix, as Eelco and Magne have both done and shown themselves. I've come to notice, however, that finding the best mix requires a good amount of patience and experience.

Magne must bear most of the blame for luring me into computer science research in the first place. His sneaky trick was: "If you're not planning on going into research, there's no need for me to spend any time commenting on your master thesis, since you won't be needing good marks on it anyway." It worked. (I was young and naive.) I agreed to applying for a PhD scholarship, and the journey since has been most enjoyable. This journey took me to Eelco's lab at Utrecht University, The Netherlands for the academic year 2004/2005. Escaping the rain in Bergen for ten months was great. It was better still to work with Eelco and his band of merry hackers. Most of the Stratego-specific work, such as GraphStratego, AspectStratego and MetaStratego, has its origins from my time in Utrecht. Martin Bravenboer, Eelco Dolstra, Rob Vermaas, Rui Guerra, Karina Olmos, Armijn Hemel, Iris Reinbacher, Andres Löh, Peter Verbaan, Peter Lennartz and the rest of the gang in Utrecht made my stay there truly memorable.

My journey did not end in Utrecht, however. I've also had the pleasure of visiting the lab of Krzysztof Czarnecki at the University of Waterloo, Canada. I greatly appreciate the insights provided by Krzsysztof. While there, I conducted most of the work that went into the software transformation systems survey of this dissertation. Again, I was extremely fortunate to spend time with some great lab mates: Barry Pekilis, Sean Lau, Michal Antkiewicz, Krzysztof Pietroszek, Chang Hwan Peter Kim, Abbas Heydarnoori, Igor Ivkovic and other passers by.

But wait. There's more. I spent last summer at IBM's T.J. Watson Research Center in New York, USA. While there, I worked with Norman Cohen, Paul Chou and Vijay Saraswat. Though not directly relevant to my thesis project, I got to hack

on compilers for three months straight, and was happy as a clam. I met heaploads of nice people there as well, including my lab mates Marco Zimmerling and Young-Ri Choi, and fellow interns Ilona Gaweda and Shane Markstrum. I also got very useful feedback on my research from Bob Fuhrer, Frank Tip and John Field. My time at Watson enforced my motivation for plugging transformation systems into existing language infrastructures (of which they have a lot).

During my occasional stays Bergen, I've been surrounded by rain and supportive friends. My brother Arne Kristian and Eva Kamilla have always had a warm meal and a mattress ready. Tilde Broch Østborg, the person apart from my family who knows me the best, has been a constant support. Lately, Tormod Haugen has pleasantly invaded my privacy and offered his dry wit and juicier cooking. At the office, Anya Bagge has always had a solution to my LaTeX problems and there is nothing worth knowing about building Stratego programs that Valentin David hasn't mastered. Paul Simon Svanberg has been my link to Real Life in the last few months and Eva Succi has made me believe that I know a little bit about LaTeX, too.

Throughout the entire process, whenever something went awry, or whenever something went allright, my mother has been but a phone call away. And in the way only mothers can, she's put things in perspective when I've been suffering from the compulsory fits of PhD despair.

During the writing of this dissertation I have received thorough input from Eelco and Magne, but I've also had great help from some very good friends. In alphabetical order (because any rating would be inaccurate and unfair): Martin Bravenboer, Tormod Haugen, Barry Pekilis.

Part I

Introduction

– I don't see how you in ten pages can do the whole thing completely wrong!

– Barry Pekilis

Introduction

Software is a crucial part of the modern infrastructure on which we all rely, and, therefore, it must be reliable, robust, correct and be able to evolve over time with our changing needs. Ensuring these properties for the massive amounts of software in use is considered one of the grand challenges in computer science. This social and technical challenge is often referred to as "dependable systems evolution" [Som00], "the software maintenance challenge" [Art88], "the software crisis" [DT96] and "trustworthy computing" [MdVHC03].

Better tools and techniques for processing and manipulating software are likely to be part of any solution to this challenge. Development of software processing tools and techniques is studied in the field of program transformation [PS83]. Many results from this field have proven to be highly applicable for software evolution. A frequently encountered drawback, however, is that implementations of program transformation and analysis techniques are often language-specific; they tend to be tied to the front-end or grammar they were written against, even when the underlying algorithms are general. This significantly impairs reuse of transformation code and systems.

This dissertation addresses the reuse limitation by introducing novel techniques for constructing reusable, language-independent program analyses and transformations. The proposed techniques include a versatile approach for easily plugging transformation systems into existing language infrastructures, such as compilers, and a declarative, aspect-based approach for software practitioners to express transformation programs for language families, rather than just for a single language. With these techniques in hand, the dissertation demonstrates how automatic software maintenance tasks can be increasingly expressed in a reusable manner. Case studies illustrate their applicability to encoding of architecture and design rules as executable program analyses, expressing control- and data-flow transformations, and interactive code generation of unit tests from user-written axioms.

1.1 Software Evolution

Software maintenance and evolution is by far the most expensive and time-consuming part of the software life-cycle [Pfl05]. The trend during the last 30 years shows that maintenance is an increasing part of the total software cost. Reports from the 1970s suggest that 60-70% of the total cost went into maintenance and evolution [ZSJG79]. In the 1980s, this figure crept closer to 70% [McK84] and, during the 1990s, it reached around 90% [Moa90, Erl00]. About 50% of the maintenance time is spent understanding the existing software [FH83].

Organisations with an investment in software are perhaps affected by this fact the most when they need to effect substantial changes. The sheer size of the code bases make radical changes and redesigns prohibitively and increasingly expensive. Ulrich [Ulr90] estimated that 120 billion lines of code was maintained in 1990. In 2000, the number was at 250 billion lines according to Sommerville [Som00]. Estimates by Müller suggest that the doubling happens around every 7 years [MWT94].

Software is becoming a limiting factor for progress in all kinds of organisations. To escape this situation, software needs to be constructed differently, and in ways which make it possible for small teams of programmers to understand, maintain and change large projects with millions of lines of code. Large parts of maintenance need to be done with (semi-) automatic software processing tools. Automation is key, but automation cannot work until the substrate being processed, the software, is easily managed by the tools. This means inventing better techniques for analysing and transforming large code bases.

1.2 Program Transformation

The field of program transformation is concerned with developing theories, tools and techniques for the analysis and transformation of programs. Typical applications in this field include transformation of programs to improve a certain metric such as execution speed, class cohesion or memory footprint; translation between languages, e.g. compilation, code generation and interpretation; analysis and verification of program properties such as absence of deadlocks, information leakage or buffer overflows. Each of these examples constitutes a transformation problem or a transformation task. A fuller discussion of program transformation is given in Chapter 2.

Program transformation techniques aid in the development of robust language infrastructures which in turn provide the basic components required for all forms of language processing. On top of these infrastructures, scalable analyses and transformations have been realised for many problems such as searching for code defects and security vulnerabilities. These analyses can handle multimillion line projects. How-

ever, while these analyses and transformations generally consist of algorithms and data types that are language independent, their implementation are usually specific to a given infrastructure. This makes them very difficult to reuse across different infrastructures, even for the same language. Presently, they are only accessible by a handful of specialists and have not gained widespread acceptance. This effectively reduces reuse of both knowledge and tools, and seriously lessens the promise of program transformations as an approach for dependable software evolution.

This dissertation focuses on:

- methods for constructing versatile program transformation environments which aid developers in implementing reusable, language-independent transformation programs;
- how to express transformation programs, and how to design transformation languages such that transformations can become reusable across subject languages and between transformation tasks;
- how to capture subject language constructs, and other entities found in software, using transformation functions and abstract data types in the transformation language; and
- how to manage these transformation functions and data types so that they are convenient to use by programmers of transformation programs.

This work reuses and expands upon promising techniques that encourage language independence and reuse of transformations. The paradigm of strategic programming has a central part in this dissertation.

1.2.1 Strategic Programming

Strategic programming [VB98, Vis99, LVV03] is a generic programming technique for processing tree- and graph-like object structures. The technique separates two concerns: object transformations and traversal schemes. Strategies are built using traversal combinators and provide complete control for expressing generic traversal schemes. These strategies are parametrised with transformations that are responsible for supplying the problem-specific transformations.

This separation is a particularly powerful approach for building reusable program transformations. The strategies can be reused across transformation problems and subject languages, whereas the transformation parameters, expressed as rewrite rules, are used to adapt the generic strategies to a particular language and problem.

Relatively few programming languages have been built with strategic programming in mind. One example of a "strategic" language is Stratego [BKVV06], a

domain-specific language for program transformation based on a sub-paradigm of strategic programming called strategic term rewriting.

In (strategic) term rewriting approaches to program transformation, programs are described as terms which in most respects may be considered analogous to trees. Using terms and rewriting allows the succinct expression of many transformation problems, but the terms are sometimes also a limitation. The choice of model used to describe programs in a given transformation system has consequences for which transformation tasks that system is best applicable to.

1.3 Program Models

The effectiveness and applicability of a software transformation system depends to a large extent on how its underlying program model has been formulated. The model determines which transformation tasks will be easy and which will be difficult or impossible. Particularly, the "abstractness" of the representation determines which analyses and transformations are possible – if the model is too abstract, refactoring is not possible, and if the model is too detailed, many analyses become too expensive.

Common representations include Prolog-style fact databases, relational databases, various forms of graphs, lists of tokens and concrete syntax trees. All of these are discussed in Chapter 2. One representation, which is noteworthy because it relates very closely to the representation of programs as terms, is the abstract syntax tree.

Abstract Syntax Trees

Abstract syntax trees (ASTs) contain the essence of programs. They are a minimal and precise form of syntax trees (sometimes called parse trees). Syntax trees are constructed by parsing the source code text. The resulting tree contains all the lexical tokens of the original source code, possibly also including whitespaces, represented as a tree according to a[1] subject language grammar.

For most transformation and analyses tasks, both the tokens and whitespaces are redundant. Stripping them away is desirable, for efficiency reasons. This stripping yields an AST which contains the essence of the original textual representation [2].

The AST has numerous appealing advantages:

- it is a high-level, as opposed to machine-level, representation;

[1]A previous version of this manuscript erroneously used the definite article here. As Peter Mosses kindly pointed out, multiple variants (implementations) of a language grammar usually exist. Furthermore, it is desirable to keep the AST interface decoupled from the underlying grammar as much as possible, so that clients to the AST API are insulated from incidental (implementation-specific) grammar changes.

[2]McCarthy, the father of Lisp, is generally credited with inventing the term AST.

- ASTs capture the essence of the language;
- everything in the source code that contributes to the executed program is in the AST;
- using maximally shared, directed acyclic graphs, ASTs can be stored and exchanged very efficiently [vdBdJKO00];
- there are a number of established techniques for augmenting ASTs with extra information such as layout, line number information and traceability.

For these reasons, most of the examples in this dissertation will revolve around ASTs — since an AST captures the essence of a subject language, abstracting over languages implies abstracting over ASTs. ASTs also have their limitations. Some of these will be addressed in Chapter 7 where strategic graph rewriting is discussed. It is important to keep in mind that the techniques developed herein are not bound to just ASTs; most will work for any tree or graph-like structures which may be arbitrarily more or less abstract than ASTs.

1.4 Language Abstractions for Program Transformations

The strategic programming paradigm is an attractive starting point for expressing reusable, language-independent transformations. This paradigm, and in particular strategic term rewriting, provides an attractive level of genericity in the formulation of transformation programs. Certain obstacles remain, however, many of which are shared with other approaches to program transformation. These must be addressed if substantially better levels of reuse and language independence are to be achieved.

One of these limitations is the inability of transformation systems to abstract over its program model implementation. It would be attractive to separate the transformation engine logic from the program model representation. It should be complemented with a versatile technique for adapting transformation engines to external program models. This would make it possible to combine transformation engines with any software development framework that provides a suitable program model.

Another limitation is the severely restricted ability of modern transformation systems to cope with cross-cutting concerns in transformation programs. Related to this is the ability to adapt existing transformation programs to new subject languages, or to changing program models.

A final limitation, particular to strategic term rewriting, is the poor support for program models that are graph-like in nature, such as program flow graphs.

The strategic programming paradigm has been extended in this work to address the above limitations using the following abstractions:

Program Object Model Adapters A program object model (POM) adapter is a technique for abstracting over implementation details of the program model in a given language infrastructure. The transformation system is written against the POM adapter interface. It is a minimal interface for navigating and manipulating tree and graph structures. By supplying infrastructure-specific adapters that translate operations on this interface to operations on the internal object model, transformation engines can be freely reused across language infrastructures, e.g. across compiler front-ends. A notable feature of the technique is that the majority of the adapter code can be automatically generated by analysing the object model interface of the language infrastructure.

Aspects Aspects extend the strategic programming paradigm with a general approach to capturing cross-cutting concerns and deal with properties such as traceability, type checking and unanticipated extensibility. Using aspects, it becomes easier to express generic transformation algorithm skeletons and to adapt these to specific program object models and to specific subject languages.

References References provide an extension to the strategic term rewriting paradigm for rewriting on graph-like structures. This allows the strategic term rewriting machinery to be applied to computing on control- and data flow graphs. References provide a way to turn some global-to-local rewriting transformations into local-to-local.

It should be noted that these abstractions can be recast for other transformations languages and programming language paradigms. This will be discussed in the respective chapters.

It must also be noted that the field of software verification and validation, which is also an important direction for dependable systems evolution, largely falls outside the focus of this thesis. Software verification and validation typically uses abstract models of the underlying software. These models are partially or fully extracted from the existing software using a variety of different tools. The techniques and tools described in this dissertation can thus complement these approaches.

1.4.1 Extensible Languages

When expressing program transformations, one needs to handle domain abstractions with cross-cutting properties such as scoping rules, variable bindings and state propagation. The behaviour of these domain abstractions may be very complex. While manipulating domain abstractions using functions and abstract data types is possible, it is often notationally inconvenient. They frequently exhibit a cross-cutting nature which results in cross-cutting concerns in the transformation program.

In some cases, these concerns can be handled using techniques borrowed from aspect-oriented programming. By extending the transformation language with support for aspects, one can modularise some of the cross-cutting concerns arising from domain-abstractions into libraries. However, not all cross-cutting concerns are expressible in aspect-languages and many that are suffer from complicated notations. Some of the proposed abstractions, such as the ones providing graph rewriting, are therefore realised as active libraries [VG98]. Libraries in this form can interact with the compiler to provide detailed, library-specific error messages when the abstractions are misused and may also come with library-specific optimisations and notation.

Active libraries with notation extend the host transformation language with new language constructs. Each new library thus becomes a small domain-specific embedded language (DSEL). Those libraries with cross-cutting properties are termed domain-specific aspect languages (DSALs). The extensible transformation language framework called MetaStratego supports both forms of language extensions. The framework allows Stratego developers to implement their own active transformation libraries. To a certain extent, MetaStratego follows the approach to language extension described in [Vis05b].

1.5 Method

The method employed for arriving at each of the results in this dissertation followed a simple, four step process:

1. *Identify Problem* – A specific limitation preventing language independence or reusability was identified.

2. *Formulate Solution* – An analysis was conducted to describe the characteristics of the problem, and then a design was formulated which sought to solve it.

3. *Implement Solution* – The formulated solution was implemented as a computer program. In some cases, this led to language extensions, in other cases, it led to transformation libraries or new infrastructure.

4. *Demonstrate Applicability* – One or more prototype applications demonstrating the applicability of the implemented solution were constructed.

This process has been applied to each the proposed abstractions presented herein.

1.6 Contributions

The contributions of this dissertation are:

- a novel technique for plugging transformations into arbitrary language infrastructures;
- a novel extension of the strategic programming paradigm with aspects for handling cross-cutting concerns;
- demonstrating how aspects can be used to adapt strategies and rule sets after-the-fact, i.e. grey box reuse;
- a novel extension of the strategic programming paradigm for graph structures;
- the construction of a modern, interactive development environment for development of and experimentation with interactive strategic programming;
- a state-of-the-art survey of design and architectural features found in contemporary program transformation systems;
- the design and implementation of an infrastructure for an extensible program transformation language;
- a validation of the proposed techniques and abstractions through the construction of several prototypes:
  - a language extensions for alerts;
  - an interactive development environment for Stratego;
  - a compiler scripting for framework-checking; and
  - an interactive generator of unit tests from axioms of algebraic specifications.

1.7 Outline

This dissertation is divided into five parts, as follows.

1. *Software Transformation Systems* – provides background material from the field of program transformation. This introduction chapter is in part based on the paper *Stratego: A Programming Language for Program Transformation* [Kal06]. Chapter 2 gives a detailed discussion of the state-of-the-art in software transformation system design and architectural features, with a focus on the capabilities for language independence. In Chapter 3, the basic notions from universal algebra and term rewriting are given along with a formulation of the System S calculus for strategic term rewriting. The Stratego language is an implementation of the System S calculus.

2. *Abstractions for Language Independence* – contains the main contributions of this dissertation. Chapter 4 introduces the program object model adapter technique and shows how it allows plugging transformation systems into existing language infrastructures. This enables large-scale reuse of entire transformation environments. The chapter is based on the paper *Fusing a Transformation Language with an Open Compiler* [KV07a] written with Eelco Visser. In Chapter 5, a language extension for capturing cross-cutting concerns in strategic programming languages is introduced based on the paper *Combining Aspect-Oriented and Strategic Programming* [KV05] written with Eelco Visser. The chapter describes a flexible and declarative technique for adapting and extending general transformation algorithm skeletons to specific problems and subject languages.

3. *Supportive Abstractions for Transformations* – provides additional abstractions which augment the main abstractions proposed in the previous section. Chapter 6 introduces the Stratego programming language and MetaStratego, an extensible variant Stratego language and its compiler infrastructure. This chapter is partly based on *Stratego/XT 0.16. A Language and Toolset for Program Transformation* [BKVV07] and *Stratego/XT 0.16: Components for Transformation Systems* [BKVV06], both written with Martin Bravenboer, Rob Vermaas and Eelco Visser. The MetaStratego infrastructure forms the basis for all the language abstractions proposed in this dissertation. Chapter 7 shows an extension to Stratego that supports a particular form of graph rewriting and motivates its use by computations on control flow graphs. It is based on the paper *Strategic Graph Rewriting: Transforming and Traversing Terms with References* [KV06] written with Eelco Visser. This extension allows strategic term rewriting techniques to be applied to other program models than (syntax) trees.

4. *Case Studies* – discusses several prototypes where the abstractions from the previous parts have been tested in practise. Chapter 8 gives an application of the language extension techniques explored in this dissertation to a domain-specific aspect language for mouldable failure handling. It is based on the paper *DSAL = library+notation: Program Transformation for Domain-Specific Aspect Languages* [BK06] written with Anya Bagge, but the alert extension was first explored in *Stayin' Alert: Moulding Failure and Exceptions to Your Needs* [BDHK06] written with Anya Bagge, Valentin David and Magne Haveraaen. This chapter is included to demonstrate that the language extension techniques employed in this dissertation are more generally applicable. Chapter 9 introduces an interactive development environment for (Meta)Stratego called Spoofax, based on the paper *Spoofax: An Extensible, Interactive Development Environment for Program Transformation with Stratego/XT* [KV07b] written with Eelco Visser. Parts of the Spoofax infrastructure have served as a testbed

for many of the other case studies. Chapter 10 demonstrates the applicability of the proposed abstractions with a case study demonstrating how the Stratego transformation system may easily be plugged into an existing development framework for Java. This allows library-specific analyses and transformation to be written by developers of Java frameworks and libraries. Chapter 11 shows how the transformation infrastructure and language abstraction may be applied to interactive program generation. A code generator for unit tests from axioms is presented, based on a testing methodology proposed by Magne Haveraaen.

5. *Conclusion* – contains some general reflections over language-independence as well as the concluding remarks. Chapter 12 is devoted to a summary and general discussion of the results obtained in this work. Chapter 13 discusses further work. Chapter 15 summarises. Chapter 14 contains the conclusion.

1.8 Summary

Dependable software evolution is one of the grand challenges in computer science. Automating maintenance tasks is one key way to tackling this challenge. Program transformation provides scalable and robust techniques for automatic maintenance, but is hindered by poor reuse and language-dependence. This dissertation claims that better reuse and language-independence can be found by abstracting over program models and by using aspects to adapt transformation algorithms to specific subject languages and program models. The rest of this dissertation serves to substantiate this claim.

Part II

Software Transformation Systems

2 Programmable Software Transformation Systems

This chapter gives an overview of the state-of-the-art in architectures and designs for programmable software transformation systems. This is highly warranted because unlike for business systems, compilers and web applications, no books exist which propose best practises for design and implementation of software transformation systems. In fact, even the research literature is to a large extent lacking in such information.

Architectural features and design considerations for these systems are explored using a formal notation called feature models, and further illustrated with examples taken from a careful selection of a dozen concrete research systems. The feature models [Bat05] are used to compare and contrast the design of both architectures and transformation languages. They give a sense of the complexity and breadth of the design space for software transformation systems. Special focus is placed on the program models found in transformation systems, and how these interrelate with the transformation languages.

2.1 Software Transformation Systems

A *software transformation system* is an application that takes a *source program* written in a *source language* and transforms this into an *target program* in a *target language*, according to instructions of a *transformation program*, written in a *transformation language*. The source language can be any formal language. What some refer to as (code) generators are included in the definition. In cases where distinguishing between the source and target language is not necessary, the term *subject language* will be used. It is meant to subsume both. The transformation is implemented by a *transformation programmer* and is always designed to preserve certain semantics. The exact semantics to be preserved are specific to the transformation, however. The goal of a transformation T is to reduce some cost $C_m(p)$ of some metric m on a program

p: we want $C_m(T(p)) < C_m(p)$, i.e. the transformed program should be "better", according to some metric [PP96, Pai96, CC02].

A traditional application area for software transformation is *transformation-oriented programming* [Par86, Fea87]. In this approach to software development, an executable implementation in the target language is derived mostly automatically from a formal, non-executable specification in the source language. Each transformation step is proved correct, either by only applying transformations guaranteed to preserve the desired semantics, or by manually filling in proof obligations the transformation system cannot automatically resolve. Here, the metric is executability – eventually an executable program is obtained, and the property being preserved is the correctness of the behaviour of the program, with respect to the source specification.

Another important application is *source-to-source* transformations, where the target and source language is the same. Typical applications in this area include program optimisation, where execution speed is the metric; re-engineering, where certain notions of maintainability are used as metrics; and refactoring [Opd92], where (often very loose) metrics for design quality are used. Software transformation techniques and systems have also been used to create compilers, source code documentation systems and program analysers. The survey by Partsch and Steinbrüggen [PS83] contains additional examples of applications for transformation systems.

A note about compilers is warranted. While the general definition above also treats compiler as software transformation systems, the subject of this survey – *programmable* transformation systems – differs from compilers in one crucial aspect: the transformation programmer can extend and adapt the software manipulation facility by supplying new transformations. A programmable software transformation system may be seen as a programming environment built specifically to manipulate programs, i.e. to implement transformation programs. It is therefore more natural to compare programmable transformation systems to compiler construction kits, so-called compiler compilers, rather than directly to compilers. Conceivably, transformation systems could be built directly on top of compilers, however. This is the subject of Chapter 10.

This chapter will show that software transformation systems are available in many variants, ranging from extensions to general purpose programming languages, to fully self-contained and stand-alone transformation environments.

2.1.1 Anatomy of a Transformation System

A common way to think about transformations is to divide them into *stages*. All stages taken together is considered a *pipeline*. The syntax of the input and output languages are specified by *source-* and *target grammars*, respectively. For source-to-source transformation systems, as illustrated in, Figure 2.1, the source and target language is the same.

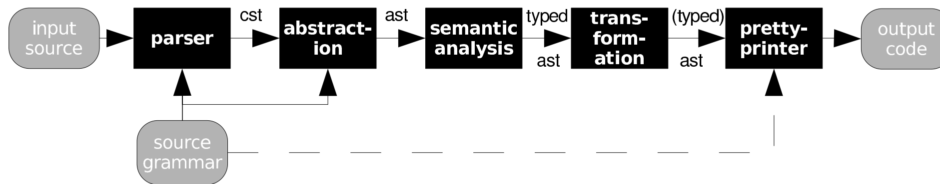


Figure 2.1: Conceptual pipeline for a source-to-source transformation system.

The process indicated in Figure 2.1 starts with the system parsing the source of the input language. The format of the input language is described by a *source language grammar*. The parsing stage constructs a parse tree, or *concrete syntax tree* (CST), from the input text. Layout and unnecessary lexical elements such as parentheses and keywords are removed from this tree in the abstraction stage, and an *abstract syntax tree* (AST) is derived. Semantic analysis is performed and the AST is annotated with type information. In practise, the AST may be constructed while parsing, and in some implementations, type checking is also done concurrently with parsing. The transformation rewrites the AST. After modifications are complete, the tree will be serialised back to source code, using a code formatter, or *pretty printer*.

This model is highly conceptual. Many source-to-source transformation systems, such as TXL [Cor04], transform the CST directly. ASTs are never derived. Some systems do not support type contexts and the AST in these systems will not contain type information. Others construct a higher level program model, or an abstract syntax graph, which is then subjected to graph rewriting techniques.

A complete transformation, from program code to program code, is called a *run*. Each of the boxes in Figure 2.1 represents a well-delineated transformation, and is called a *stage*. Each stage may internally be split into *phases*. Each phase consists of a sequence of rule application *steps*. A step, or rule application, is the smallest unit of transformation. They represent the atoms from which transformations are built.

Other architectural models for transformation systems also exist. A common example is the *incrementally updating* system. In these systems, the output of one run is the input to the next. A human operator is usually involved in adjusting the transformation parameters between each run.

2.1.2 Features of Software Transformation Systems

A software transformation system may be decomposed into three closely related parts: a *program representation* holding the program the system manipulates, a *transformation language* for expressing these manipulations, and an *environment* which is used to interact with the developer. Figure 2.2 shows a feature model fragment which visualises this decomposition. Details of each of these features will be described in the

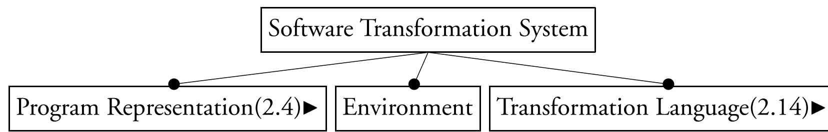


Figure 2.2: Top-level features of software transformation systems.

following sections. The numbers in parentheses refer to figure numbers for additional diagrams which elaborate on a particular feature. Not all features will be discussed in full detail. This dissertation is largely concerned with the interplay between abstract models for programs and transformation languages used to manipulate these. A full discussion of the user interface, i.e. *environments*, of transformation systems is therefore out of scope. Before continuing, the feature model notation is explained.

2.2 Feature Models

Symbol	Explanation
F	Solitary feature with cardinality [1..1], i.e., *mandatory* feature
F	Solitary feature with cardinality [0..1], i.e., *optional* feature
[n..m] F	Solitary feature with cardinality $[n..m]$, $n \geq 0 \wedge m \geq n \wedge m > 1$, i.e., *mandatory clonable* feature
F	Grouped feature with cardinality [0..1]
F▸	Feature model reference *F*
	Feature group with cardinality [1..1], i.e. *xor*-group
	Feature group with cardinality $[1..k]$, where k is the group size, i.e. *or*-group

Figure 2.3: Symbols used in cardinality-based feature modeling

Feature models [Bat05] provide a graphical notation for describing variation points found in the design of software systems. The notation is well suited for visualising the relationship between features using the precise and general kernel language described in Figure 2.3. Organising the feature space into hierarchical contexts helps guide discussions. The application of feature models spans from the purely conceptual, at the domain concept level, to implementation detail, at the design and architectural level. This chapter mainly uses feature models for describing architectural variation points.

By saying that feature models describe the essential variability of software transformation systems, it is meant that they describe the *characteristic concepts* and features for these systems, and that the models show the relationships and interactions between these. The characteristic concepts and features are described using a *design vocabulary*, which is introduced in the boxes of the feature diagrams. It is important to point out that this chapter is guided by the notion of *"characteristicness"*: a discussion of features which are also commonplace outside software transformation is avoided; features which pertain to software systems in general will not be discussed.

Alternative formalisms for describing design knowledge are ontologies [Gru93]. Feature models were chosen here because they are better suited to visualise the variability and configuration aspects of software designs. For a discussion of the relation between feature models and ontologies, refer to [CKK06].

2.3 Program Representation

Software transformation systems operate on formal documents which have a precise syntax definition and sometimes a detailed, formal semantics. These documents may be programs or specifications, or simply structured specification documents with little semantics. Both specifications and program source are commonly referred to as *program code* or *subject code* in the rest of this chapter. Though programs are formal documents, models representing programs are referred to as *program object models* (sometimes just program models) throughout this dissertation, to distinguish them from general document object models as found in the field of document processing. This dissertation takes the stance that subject code usually has an a priori defined semantics which operations on the program object model must preserve.

Due to its formal nature, program code has a clear structure, but this structure does not necessarily match how the transformation system represents program code internally. The choice of internal data structure used to represent programs affects the ease with which various operations can be performed. For example, if the program is represented as a control-flow graph, control flow analysis becomes easy, but structural or syntactic changes, such as refactoring is all the more difficult. The choice of representation significantly affects the possible applications of a transformation systems. Specific design and implementation choices for the representation further influence both performance characteristics and the difficulty of expressing different kinds of analyses and transformations. This argument also works in reverse: the intended transformations of a system will to a large extent dictate the choice of internal representation.

As an example, consider software transformation systems intended for source-based re-engineering. These usually employ a parse tree representation that accurately captures source code details. This may include layout and comments. On the other

hand, systems intended for software modelling mostly use graph-like representations that are far removed from the concrete syntax of the source language.

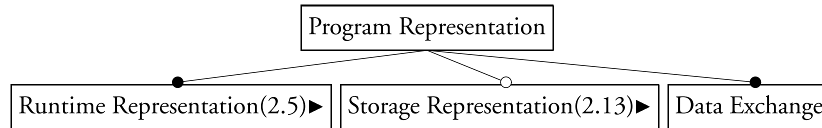


Figure 2.4: Feature decomposition for program representation.

Figure 2.4 shows a decomposition of the feature space for *program representations*.

- *Runtime representation* – refers to the data structure and features of how the program code is represented at runtime. Of all the features related to program representation, the choice of runtime representation has the largest impact on the expressiveness and performance of a transformation system, see p. 20.
- *Storage representation* – refers to the facilities for storing program code on disk at intermediate transformation stages. Choices pertaining to intermediate storage on disk affects the interoperability and modularity of a system, see p. 30.
- *Data exchange* – refers to facilities for loading source code into the system and produce target code as output. This might be features for parsing and pretty-printing, used with source-to-source transformations. These features fall mostly outside the scope of this dissertation.

The following sections discusses each feature in turn.

2.3.1 Runtime Representation

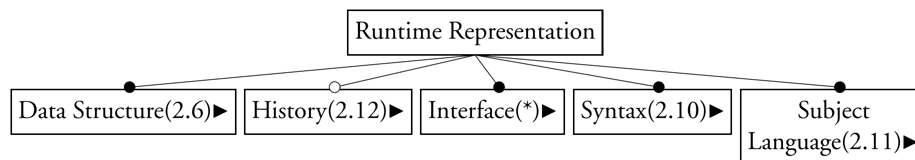


Figure 2.5: Feature decomposition for runtime representation.

Subject programs are contained in a runtime representation when the software transformation system executes. This may for example be an abstract syntax tree, a graph model, or a database. Collectively, these are called *program object models*, and may be described by the following features.

- *Data structure* – refers to the choice of (principal) abstract data type used for the program object model. This is arguably the most important aspect of the runtime representation. Common choices are trees and graphs, with various invariants on the well-formedness of the subject program, see the next section.

- *History* – the representation may optionally support the notion of transformation history by keeping a modification history of the program code, see p. 29.

- *Interface* – refers to the programming interface available for the runtime representation. In many systems, the interface is available as language constructs in the transformation language. That is, the transformation language is specifically designed with primitive constructs for manipulating the program object model. For this reason, the interface feature is discussed together with the other language features, in Section 2.4.

- *Syntax* – refers to the types of syntaxes available – abstract or concrete – for writing and reading program code when implementing transformation programs, see p. 27.

- *Subject language* – The language in which the program code must be expressed, i.e. the supported source and target languages, see p. 29.

Data Structure

The feature model for the data structure in Figure 2.6 describes which data types are used to represent the program code at runtime, and which support exists for maintaining well-formedness of the program code structure.

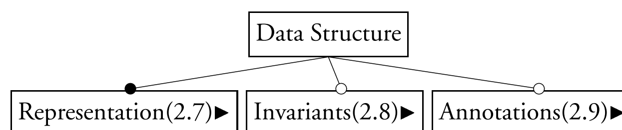


Figure 2.6: Feature decomposition for data structure.

- *Representation* – details the choice of abstract data type for the program code, ranging from strings and lists to relational databases, see the next section.

- *Invariants* – describes how structural and semantical invariants of the program code can be placed and enforced on the representation, see p. 24

- *Annotations* – refers to the ability of the representation to handle meta-information not part of the program code structure, see p. 26.

Representation

Figure 2.7 describes the features of the data type used to represent the program code. Under each principal choice (list, tree, graph, etc), the most common variants are shown.

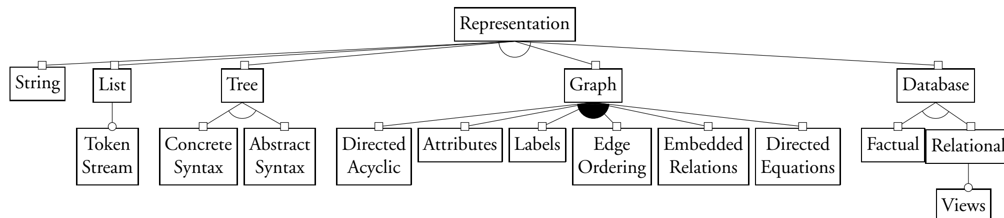


Figure 2.7: Variants of data structure representations.

- *String* – The simplest choice of data structure for representing program code is a text (or even binary) string. In this case, the transformation system amounts to a string rewrite engine, as in the theory of formal languages and automata. The C/C++ preprocessor is one example of such a "transformation system". The trio `sed`, `grep` and `awk` [DR97] of Unix tools is another, based on regular expressions. Representing programs as strings fails to capture the grammatical structure inherent in the program code. This quickly leads to subtle bugs for any non-trivial transformation. String rewriting engines can hardly be called software transformation systems.

- *List* – A slightly more structured representation than the string is the token stream output by a lexer, i.e. a *list* of tokens. Each token is marked with a type, such as keyword, identifier, string literal or parenthesis, e.g:

  ```
  ["if":keyword, "(":left_paren, ..., ")":right_paren]
  ```

  The ANTLR parser toolkit [PQ95] supports rewriting on token lists. Both the string and token list representations of program code are useful for limited, layout preserving rewriting. As long as no context or grammatical information is needed, the matching can be done reliably at the lexical level.

- *Trees* – Most practical transformations need at least grammatical structure and most often also context knowledge such as variable binding or type information. Extracting the grammatical structure from program code text can be automated using syntax analysis, i.e. parsing. Syntax analysis produces trees. Representing program code as trees dates back to the earliest compilers, and multiple variants are possible. In the case of *concrete syntax trees*, the tree contains a faithful representation of the source code, possibly excluding

non-essential whitespace. In the case of *abstract syntax trees*, all non-essential nodes, such as whitespace, parentheses, statement and expression separators, have been removed. These can be automatically regenerated. Normally, comments and documentation are also left out. Trees are often given a textual syntax, in the form of terms, e.g.:

```
If(False,Int(1),Int(2))
```

The maximal sharing [vdBdJKO00] technique is a variation of the tree representation which improves execution time of matching and memory efficiency is. The tree is represented as a directed, acyclic graph (DAG), where equal subtrees occurring multiple times in the tree are stored only once. This technique improves the efficiency of term matching significantly. It has been used in several transformation systems, including ASF+SDF [vdBvDH+01], Stratego, ELAN [BKK+04], Tom [MRV03] and derivates of these. It is important to note that the maximal sharing technique hides the sharing, making the DAG behave as a tree. This is required for the term rewriting theory. Rewriting on DAGs, sometimes referred to as term graph rewriting, has different termination and confluence properties [Plu99, BEG+87]. An example of transformation system based around term graphs is HOPS [Kah99, KD01], an interactive program transformation and editing environment that ensures syntactic and semantic correctness. HOPS may also be described as a syntax directed editor.

- *Graph* – Plain trees are not sufficient for explicitly capturing some important types of context information, such as typing and variable binding. Section 2.4 discusses how tree-based transformation systems deal with this problem. The program code may be expressed as a *graph*. This allows additional links (edges) to be added from, say, a variable use to its definition, or from an identifier to its type, thus capturing context information. *Labelled* edges are handy for distinguishing between kinds of relationships between two nodes, for example, between a `use-def` and a `type-of` relationship. *Attributes* are named properties of nodes that contain values. In most graph-based systems, a node may have a set of named attributes. These can be matched on during rewriting. Some systems, such as the modelling system MetaEdit [SLTM91], also allow attributes on edges. Attribute grammar systems are capable of declaring dependencies between attributes across nodes using *directed equations*. Nodes may be related using *embedded relations*, as in PROGRES [Sch04]. These features are closely tied to transformation language features and are discussed in Section 2.4. None of the transformation systems known to the author employs hypergraphs directly, i.e. graphs where edges connect more than two nodes. The GAMMA multiset rewriting system [BM91, BM93], seems to come closest in terms of hypergraph semantics. Other works have been derived from this,

such as [CFG96]. Neither of the systems is used for program transformation.

- *Database* – Linking together nodes in a graph or subterms of a term can be done using a *relational* database. Transformation systems based on this approach are comparatively sparse. APTS is the only relational database system that allows arbitrary transformation. In [SNDH04], the authors describe an interactive system focused on the refactoring of Clean programs. The system described in [CNR90] extracts facts from C code into a database, but only allows subsequent analysis, not transformation. In all cases, the program code is expressed in tables with relations between them. Transformations and analyses are expressed as relational queries, in styles similar to normal relational databases. A feature unique to the database approach is the ability to declaratively construct custom *views* of the program code and do manipulation on these. In all other approaches, similar functionality must be provided by the developer, and is highly non-trivial. A related approach is the *factual* databases used in logic programming languages such as Prolog. This is employed by JTransformer [Win03, KK04]. The structure of the program code is stored as facts in a database. Questions (queries) may be asked. These are automatically resolved against the database by the Prolog inference engine. A discussion of the finer points of different database approaches is beyond the scope of this chapter, save to point out that while the Prolog model is based on the theory of predicate calculus, the relational database approach has its roots in relational algebra.

Perhaps the principal tradeoff in the selection of a suitable representation is between expressiveness and efficiency. Low-level and simple abstract data types such as lists and trees are very efficient to transform, but it often becomes difficult to embed analysis results in flexible ways. That is why annotations (discussed later) are only found as additions to the more "primitive" representations. The elaborate representations, such as general graphs and relational databases, are mainly used for high-level concepts. Models are first extracted from the source code. Queries and computations are performed on these models. The results are later used to guide transformations on the low-level representations.

Relational databases are often used for various types of code querying and analysis, as in the case of CodeQuest [HVdMdV05]. The program object models used for this are removed from the primary grammar structure, because encoding recursive data structures into relational databases (tables) is generally inefficient.

Invariants

The syntax and semantics of the program code, no matter how it is represented, give rise to a large amount of invariants, see Figure 2.8. These must kept throughout the

transformation run, but may be lifted temporarily during transformation steps, or even phases.

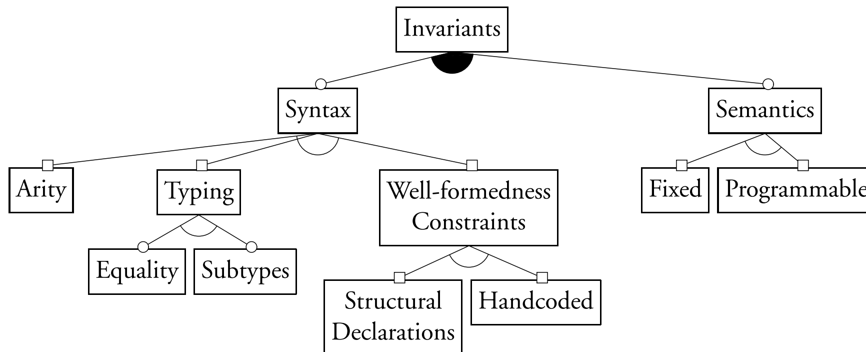


Figure 2.8: Feature decomposition for data structure invariants.

Syntactical Invariants

- *Arity* – A weak variant of typing, found in the term-based system Stratego. For each type, only the numeric arity, i.e. the number of arguments is enforced. An if-then-else node may look like *If : Expr * Stmt * Stmt*, thus declaring terms on the form *If*(e,st_0,st_1) where e must be an *Expr* and st_0, st_1 must be *Stmt*. Stratego only requires that three subterms be attached to *If*. It does not verify their types.

- *Typing* – The most common way for ensuring grammatical well-formedness on the subject code is to use the type system of a strongly typed transformation language (not to be confused with the type system of the subject language). The variants include basic *equality*-based systems (only terms of type T may be used where T is expected) and systems which support the notion of *subtyping* (any subtype of T may be used where T is expected). In either case, the syntactical correctness of the program code with respect to a grammar can be ensured. It is worth noting that although token lists discussed above are by definition typed, they rarely offer any grammatical correctness guarantees. TXL [Cor04] operates on concrete syntax trees. The language ensures that when a subtree is replaced on a given node, the new subtree must be of a compatible type, i.e. the new subtree must parse to the same production as the old. The types are defined by the grammar for the language.

- *Well-formedness constraints* – A more powerful approach is to provide a declarative language for expressing *structural constraints*. It is then possible to either verify that a given transformation will never violate these constraints, or to insert constraint checking between transformation phases, at declared *safe spots*.

AGG [Tae04] is a graph transformation system which provides structural constraints on its graphs. The constraints are specified as part of the graph grammar.

One could consider the structural constraint feature an extension or variant of user-defined types, but there are some essential differences. Ensuring that each rewriting step respects a given grammar is computationally feasible, because grammatical constraints map relatively easily to most strongly typed languages. Checking structural constraints after a rewrite step may not terminate in the general case (for example, if the constraint is given in a Turing-complete formalism). Even when the constraint language offers termination guarantees, the computational complexity may be prohibitive.

Semantical Invariants In addition to syntactical well-formedness, facilities may exist for defining parts of the semantics of the program code.

- *Fixed* – The system comes with a fixed implementation that preserves (possibly a subset of) the semantics of the subject language. In the case where the language of the program code is fixed, a complete enforcement of the semantical invariants is possible thanks to a priori hand coded logic in the system. JTransformer provides a library of conditional transformations for Java, many of which are guaranteed to preserve Java semantics. The FermaT [War02, War89] transformation library guarantees semantics preservation for FermaT's fixed subject language, WSL.

- *Programmable* – The system supports the programmer in implementing language semantics constraints, for example by providing suitable generic libraries or language constructs for capturing language semantics. Varying degrees of support for this is present in most transformation systems. Notable features are discussed in Section 2.4.

Few transformation systems enforce type-correctness of the subject code or similar forms of semantical correctness on their transformations. It may be exceedingly difficult to check for these during the runtime of the transformation, and it is also an open problem how to efficiently encode semantics for the subject language into the type system of the transformation language. For this reason, it is not uncommon to "outsource" questions of correctness to theorem provers.

Annotations

The structure used to represent the program code should be precise and minimal. This reduces the complexity of the transformation programs: fewer node types means

fewer patterns for the rules. A minimal structure sometimes conflicts with extensibility, however. It is often necessary to store intermediate computation results and relate these to elements in the program code. A common way to handle this at the program representation level is to store such information as annotations, see Figure 2.9.

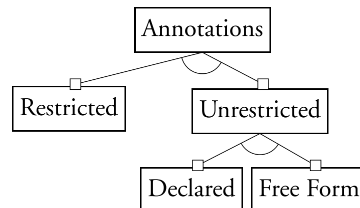


Figure 2.9: Feature decomposition of annotations on the program representation.

Annotations are (temporary) pieces of meta information that may result from analyses such as type inferencing, variable scoping or source code metric calculations. Annotations are also used to retain comments and layout information, without declaring these as part of the primary program code structure.

- *Restricted* – Only a limited, pre-defined number of annotations may be placed on the program code in the runtime structure.
- *Unrestricted* – Annotations can be freely defined by the programmer. In the case of *free-form* annotations, arbitrary meta information is allowed. This is the case for ASF and Stratego. In the case of *declared* annotations, syntactical (and optionally, semantical) restrictions are placed on the annotations. These must be declared in advance.

Annotations are different from (tree or graph) attributes in several ways. Since annotations are pieces of meta-information, they can be discarded at any time without changing syntactic or semantic validity. Moreover, even declared annotations are not part of the program code grammar, so one cannot expect that all transformations will respect and update them.

Most transformation system support annotations in one way or another. It is generally the case that strongly typed transformation languages necessitate declared, as opposed to free-form, annotations.

Syntax

Developers reading and writing fragments of subject language program code do so in the program code *syntax*. This syntax may be quite different from that of the subject language, and is is often influenced by the choice of program representation.

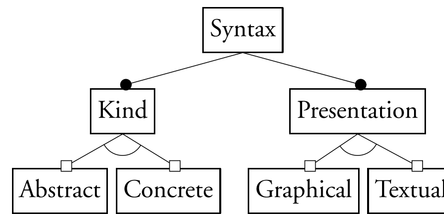


Figure 2.10: Features for supporting subject syntax in transformation programs.

- *Kind* – In source-to-source transformation systems such as TXL [CCH05] and ASF+SDF [vdBvDH+01], subject program code fragments are often written in the syntax of source language called the *concrete syntax*. The concrete syntax for the program code is embedded in the transformation program. When concrete syntax support is not present, program code must be written using the data types of the transformation language, that is, in an *abstract syntax*. This is the case for Tom and ANTLR, where tree nodes and trees are built like any Java data structure, using object instantiation.

  Stratego supports both concrete and abstract syntax, demonstrated in the functionally identical rules shown next, where the concrete syntax fragment is enclosed in "semantic" brackets:

  ```
  EvalIf: |[ if(true) ~e0 else ~e1 ]| → |[ ~e0 ]|
  EvalIf: If(BooleanLiteral("true"),e0,e1) → e0
  ```

- *Presentation* – For the systems mentioned above, the syntaxes were all *textual*. Another variation is to represent the program code using a *graphical* notation, irrespective of whether the source language is visual or not. This is done in AGG and PROGRESS which both offer abstract graphical syntax and presentation.

The primary tradeoff between concrete and abstract syntax is readability versus preciseness. Concrete syntax patterns are mostly easier to read and write for programmers. However, extra care must be taken to ensure that the pattern (and the meta variables) match exactly the types of AST nodes intended. Consider the following concrete syntax pattern:

```
|[ boolean equals(Object ~n) { ~stm }]|
```

It does not match the following declaration, because of the visibility modifier `public`.

```
public boolean equals(Object o) { return false; }
```

The pattern, as written, specifies that only declarations without any visibility modifiers should be matched. Writing exact pattern matches in abstract syntax is often easier, though significantly more verbose. On the other hand, code generation usually

benefits significantly from concrete syntax templates, since such templates are generally easier to write and maintain compared to equivalent templates in an abstract syntax.

Subject Language

The possible choices for subject language clearly defines the applicability of a given transformation system for a concrete problem.

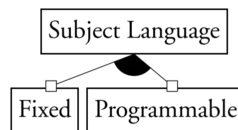


Figure 2.11: Feature decomposition for subject language.

- *Fixed* – The subject language is fixed to a particular language.
- *Programmable* – The subject language can be freely defined by the programmer.

Both JTransformer and FermaT are fixed to one subject language. This fixedness gives the systems an advantage in providing a convenient and robust transformation library. However, FermaT's subject language is WSL, a wide-spectrum language designed to capture a large set of source languages. It contains a small kernel of constructs to specify (non-deterministic) choice and iteration. Various assembler dialects have been transformed into it [War99]. Both C and COBOL code is in turn produced from WSL. The basic transformations in the FermaT library guarantee both syntactic and semantic correctness. JTransformer also comes with a library of basic transformations for its subject language, Java. Many of these preserve the Java semantics and syntax.

The choice of language may be programmable, as is the case for TXL [Cor04], ASF+SDF [vdBvDH+01], Stratego/XT [BKVV06] Tom [MGR05], DMS [BPM04], and Elegant [Aug93]. In these systems, the developer must supply all syntactic and semantics-preserving logic, using whatever support the transformation system provides for this. For realistic languages, this is a considerable undertaking. In many cases, separate projects exist which specialise general systems for a particular language. These aim to achieve the best of both worlds: a sound library of basic transformations with full flexibility, e.g. CodeBoost [BKHV03] specialises Stratego for C++.

History

Support for history as part of the runtime representation provides a low-level way of keeping track of changes to the program code. It complements execution traces, discussed in Section 2.4.

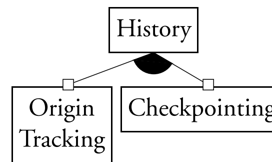


Figure 2.12: Features for history support.

- *Origin tracking* – A feature that retains tracking information with the program code elements throughout a complete transformation. It is used to determine how a code element in the final product relates to the code elements in the source input, i.e. where a given code element in the result program came from in the source program. Earlier prototypes of ASF had this feature [vDKT93].
- *Checkpointing* – Runtime representation support for transaction-like operations. With checkpointing, a snapshot can be taken of the program code so that this state can be restored if a transformation sequence fails. Stratego offers full checkpointing support due to the (local) backtracking feature of the language, as does Tom when rewriting functional terms (Tom also supports non-functional terms and graphs, where the backtracking is not available).

2.3.2 Storage Representation

Many transformation systems provide special support for storing intermediate forms of the program code, see Figure 2.13. The code may be stored in a special, efficient storage format, or as source code in the source or target language. The choice between a special storage format versus language source code affects how auxiliary information can be added.

Special storage of the internal data allows bundling of analysis results and constraints with the data. This may in turn be used to minimise costly analyses, such as parsing and type checking, by caching results on disk between executions of the system. Having a standardised internal storage facility opens up for interchange of analysis results between components of the transformation system: fragments of code can now easily be serialised and sent between separate processes, or over a network.

Aside from the size benefit offered by good storage formats, extra information such as accumulated transformation history can be added in the form of origin tracking or execution traces. This is not possible (or at least rather difficult) when the interchange format is fixed to the source code of the source (or target) language.
The storage representation feature from Figure 2.13 is decomposed into the following features:

- *Extensibility* – Either the storage representation is *fixed* for the transformation system, or it is *programmable*. This allows the programmer to extend it. Strat-

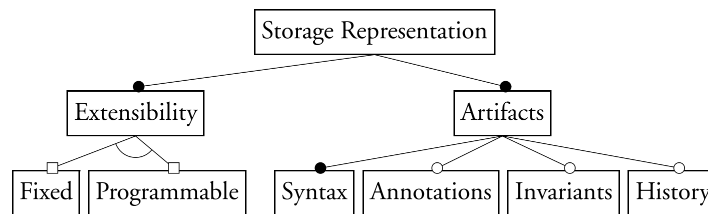


Figure 2.13: Feature decomposition of storage representation.

ego/XT and ASF use a fixed format called ATerm. In the case of Stratego/XT, additional formats may be added by the user. AGG has a fixed XML-based format for its graphs.

- *Artifacts* – The choice of *syntax* is the most influential design choice of the storage representation. When concrete syntax (of either the source or target language) is used for storage, auxiliary information is considerably more difficult to encode. By using an extensible abstract syntax, a transformation system provides the transformation programmers with more freedom.

  In the case of an abstract syntax, custom *invariants* concerning the data may accompany the program code. User-extensible invariants allows the transformation programmer to express additional invariants that must be respected by other programs and components processing this program code.

  Depending on the choice of syntax, the storage format may support storing *annotations*. There is usually a correspondence between how the runtime representation language handles annotations and how these are stored: the runtime typing and structural constrains must usually be respected.

  History – The stored files may contain checkpointing information which may allow backtracking across saved sessions. Such information allows mid-transaction saves and rewind. Additionally, origin traces may be included in the saved file This aids in origin tracking between sessions and between tools.

Storing of concrete syntax captures layout, even for visual languages, where the graphical objects in saved visual programs retain their user-edited placements. The GXL [HWS00] language encodes this information in special graph attributes in the stored files. Storing additional, custom transformation invariants along with the program is required if other transformation components are to know about these additional constraints. A caveat is that the formats used to store such constrains, and their meaning, must be known to all components. The AGG system preserves these constraints by coding both the program model and the constraints into the same unit.

2.4 Transformation Language

The transformation language is the centrepiece of any programmable transformation system. It is the main vehicle for expressing transformations, and should therefore easily express the kinds of transformations desired by its user. As with any programming language, the degree of expressiveness provided by the language is a double-edged sword: Having many language features generally increases expressiveness, so does avoiding usage restrictions on individual features. There is a tension between expressiveness and how easy proofs of transformations can be done. Usage restrictions on individual language features, and careful consideration of feature combinations are required if good provability is desired. However, not all program transformation approaches are concerned with provability. This has allowed a rich set of transformation language features to evolve.

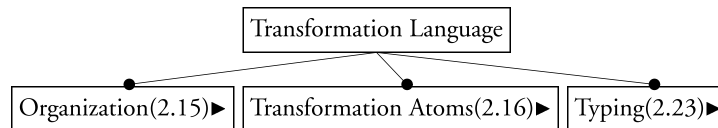


Figure 2.14: Feature decomposition of transformation languages.

Given the rich literature and existing surveys on the details of particular feature sets, such as [Fea87, PS83, Vis05a, vWV03], this section focuses on the broad lines and the relationship between transformation languages and program models. The features being considered are shown in Figure 2.14.

- *Organisation* – refers to the organisation of the rule and data declarations, see the next section.
- *Transformation Atoms* – refers to the properties of the units of transformations, i.e. the functions, rewrite rules, queries and strategies, see p. 36.
- *Typing* – describes characteristic features of how typing is realised in transformation languages, see p. 47.

2.4.1 Organisation

Features for organising the language declarations are shown in Figure 2.15. This organisation is necessary for managing the complexity of the transformation program itself. As transformation programs grow in size, they are subjected to the same issues of scale which are already seen in constructing other types of software applications.

- *Grouping* – The feature model suggests the hierarchical organisation of transformation expressions or statements into applications of *operations*. Operations

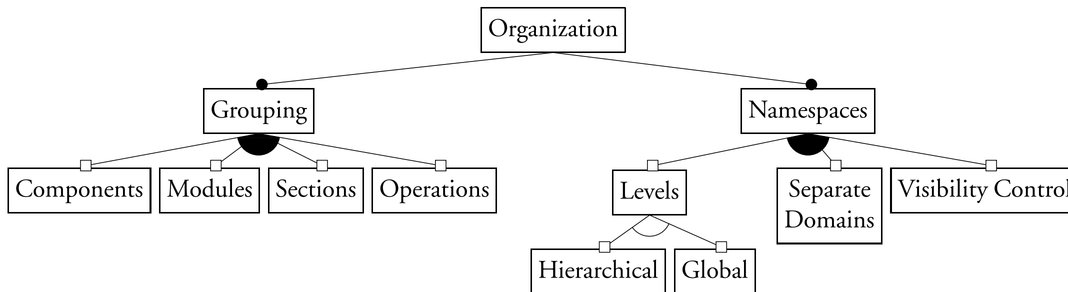


Figure 2.15: Language features for organization of declarations.

correspond to transformation atoms. For textual transformation languages, operations may be placed into *sections* inside their declaring file. Sections are in turn grouped into named *modules*. A module may be a file, or multiple files may make out one module. Modules may again be composed into *components*.

- *Namespaces* - The organisation of namespaces is related to grouping. Scoping may be done in *levels* which follow a *hierarchy*, usually the one provided by the grouping. Another alternative is no levelling at all. This gives one global name space for all declarations. In either case, the different types of declarations may be organised into *separate domains*, allowing both a rule and a type with the same name to exist at the same time without causing confusion. A final consideration is encapsulating names into their respective scope by restricting *visibility*.

The basic organisation features are combined in a plethora of ways. The class-based systems JastAdd and Tom group methods (operations) into named sections (classes) which become one level in the namespace. The methods reside in a different namespace domain than the variables and the types. It is possible to have both a type and a method with the same name without confusion.

For PROGRES, hierarchical visibility is only available for global graph constraints, on a per-section basis. For Elegant, a component is either a scanner, transformer or code generator, i.e. a phase of a compiler. For APTS, all definitions are maintained as entries in databases. You can load and store databases, composing them by merging two databases, thus emulating the concept of components. For Stratego, each file is a module, divided into sections. Rules and strategies are the only two types of operations, and both live in the same, global namespace. There is no visibility control, so rules and strategies with the same name may interfere.

One drawback of the numerous realisations of these basic features is that learning transformation languages might be daunting. The nuances and novel combinations serve to increase the learning curve for new developers. Another drawback is that there are few, if any, standardised ways of organising transformation programs. The

closest to a de facto standard, at least at the macro level, is perhaps those which mimic compiler pipelines. This is (mostly) the case for JastAdd, Elegant and Stratego.

A number of noteworthy characteristic organisation features are discussed next.

Rule/Dependency Separation – Attribute grammar systems combine the dependencies between nodes and the directed equations used to compute derived attributes into one construct. In PROGRES, the rule for computation of the derived attribute is kept separate from the dependence declaration.

Inheritance – In Tom and JastAdd, both of which are embedded domain-specific languages for Java, inheritance is used to encode the grammar structure of the subject language in the type system of the host language. Consider the following the grammar fragment:

$$\begin{aligned} expr &::= literal \,|\, binexpr \,|\, \ldots\,; \\ literal &::= string_literal \,|\, integer_literal \,|\, \ldots\,; \end{aligned}$$

This translates into the following (Java) type declarations for JastAdd (the class `ASTNode` is always the root of such type hierarchies in JastAdd):

```
abstract class Expr extends ASTNode { ... }
abstract class Literal extends Expr { ... }
class StringLiteral extends Literal { ... }
```

The situation is similar with Tom, but the programmer may choose the root class freely.

In PROGRES, the graph grammar declaration uses subtyping to declare the types (and attributes) of nodes in the graph, e.g.

```
node class Root; intrinsic a : = 1; end
node class Child1 is a Root; redef b : = 2; end
node class Child2 is a Root; redef c : = 3; end
```

Transcripts – Transcripts are an organisational unit only found in APTS. A transcript implements either a rewrite or an inference rule for a relation. A transcript for a relation contains one or more inference rules which are used to analyse the CST and maintain a database of program properties. The rules inside the transcript are applied non-deterministically until no relations in the program database can be changed, i.e. until a fixpoint has been reached. Consider the following example, included to provide some flavour of the APTS language. The example defines the notion of free variables in a SETL-like subject language [Pag93].

```
1 transcript freevar();
2  rel freevar:  [node, tree];
3     free:  [tree];
4  prompt free:  [1, ' is a free variable '];
```

```
5    external bvar:  [node, tree];
6            op:  [node, node];
7    key free:  [1];
8  begin
9    freevar(root(), .x) -> free(.x);
10   match(% expr, .x % ) | isavar(% expr, .x% )
11      -> freevar(% expr, .x% , % expr, .x% );
12   match(% lexpr, .x % ) | isavar(% lexpr, .x% )
13      -> freevar(% lexpr, .x% , % lexpr, .x% );
14   op(.x, .y) and freevar(.y, .z) and not bvar(.x, .z) -> freevar(.x, .z);
15 end;
```

This transcript, named `freevar()`, defines the relations `freevar` and `free`. It depends on the external relations `bvar` and `op` (defined in other transcripts). The `prompt` definition specifies how tuples of the relations in this transcript are displayed. The inference rules of this transcript are specified between `begin` and `end`. The rules on line 10-13 specify that any variable that is an expression on a left or right hand side of an assignment, is a free variable. `match` and `isavar` are builtins of APTS. `match` supports non-linear pattern matching (discussed later); here, `.x` is a pattern variable. The main inference rule for free variables is given on line 14, and states that a free variable `.z` of term `.y` is a free variable of term `.x` iff `.x` contains `.y` as an immediate subterm (the `op(.x, .y)` part) and `.z` is not a bound variable of `.x` (the `not bvar(.x, .y)` part). These rules are applied non-deterministically until none are applicable any more. This completes the update of the program database.

Rewrite rule transcripts are similar to relation transcripts. They consist of one or more rewrite rules, which rewrite the CST, as opposed to the program database.

Transcripts have some properties of modules. There is a simple kind of namespacing and visibility for transcripts: rules inside a transcript are not by default visible outside the transcript. Rules from other transcripts can only be invoked indirectly, by invoking their transcripts. Transcripts also enforce a special evaluation semantics. All external relations must have been evaluated before a transcript can be evaluated. Cyclic dependencies between relations are only allowed within transcripts. Multiple transcripts may be defined in the same file.

Parametrisation

The different organisation units, such as types, rules, functions and strategies may for practically all transformation languages always be parametrised with values. In the transformation languages Elegant, Tom and JastAdd types may be parametrised with types. Stratego and Elegant offer higher-order operations.

Higher-Order Operations – A catch-all feature for higher-order rules, strategies, functions and queries. The known benefits from higher-order functions also apply to

rules, strategies and queries: they aid in parametrisation and subsequent composition of code, thereby allowing a very flexible, precise and familiar notation for expressing operations. The following Stratego strategy definition defines a top-down (pre-order) term (tree) traversal, where the strategy s is applied at every subterm (tree node) before its children are visited:

```
topdown(s) = s; all(topdown(s))
```

Module Parametrisation – Parametrisation of modules, as offered by the ML-family of languages, is seen in very few of the domain-specific languages provided by any of the transformation languages. JastAdd and Tom (both based on Java, which offers parametrised classes) are the only known exceptions. Also, no transformation system currently offers parametrised components. The absence of parametrisation at higher levels, and the absence of higher levels of organisation, may be taken as a sign that issues common to programming in the large have not been addressed for transformation systems yet.

2.4.2 Transformation Atoms

Transformation atoms are the fundamental building blocks of transformations, see Figure 2.16. For rule-based languages, they are the rewrite rules. For functional languages, they are the functions. For relational languages, they are the queries. In style with modern science, transformation atoms are not indivisible: functions are made from expressions, rewrite rules from patterns and conditions, and relational queries from path expressions and statements.

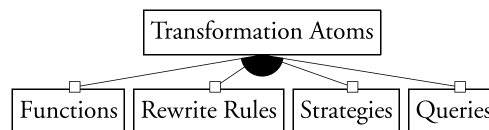


Figure 2.16: Feature decomposition of transformation atoms.

The following discussion will focus on characteristic properties of the rewrite rules, as this is arguably one of the most characteristic features of transformation languages. Functions and queries found in in transformation languages are familiar from general purpose and relational query languages.

Relations – Multiple variants of the relation feature exist. In APTS, the program database stores relations extracted from the CST using inference rules. In PROGRES, relations between nodes and node types are declared using graph queries and path expressions. In JTransformer, the Program Element Fact (PEF) database contains relations extracted from the Java AST, e.g.:

```
1 importT(10000, 30001, 20003).
2 importT(10001, 30001, 'java.util').
```

The fact on line 1 represent a Java import statement. Each fact has a unique id. The fact on line 1 has id 10000. It states that the class corresponding to fact id 20003 is imported by the compilation unit of id 30001. On line 2, another fact states that the same compilation unit also imports `java.util.*`, i.e. all classes of the package `java.util`.

In all systems, queries can be done on the relations; the relations often encode "refined facts" that are extracted and analysed from the CST and AST, i.e. information that is only implicit in the AST representation, such as the binding from a name to the actual definition for that name.

Relation Functions – Elegant provides a kind of function with a special semantics, called a relation. In contrast to functions, relations can have an arbitrary number of input and output arguments. The arguments are updated by the body in any order. The effect of a relation is to synchronise all the output domains with the input domains, [JAM99].

```
relations
MakeFunctions (NIL : List(Func), out {}, {}) { }
MakeFunctions (funcs : List(Func) out signs, decls) {
  MakeOneFunc (funcs.head out s : VOID, d : VOID)
  MakeFunctions(funcs.tail out ss : VOID, ds : VOID)
local
  signs : VOID = { s "\n" ss }
  decls : VOID = { d "\n" ds }
}
```

The relation `MakeFunctions` is used to traverse a list (`funcs`) of functions, and for each element, call the relation `MakeOneFunc` to compute its signature and its complete declaration. This results in two separate lists which are both returned from `MakeFunctions`, one for the signatures, in `signs`, and one for the declarations, in `decls`. The abstraction `MakeFunctions` therefore returns two values, whereas a function would only return one. It is possible for the return values to be declared as lazy values. In this case, they will only be computed if they are used by the caller.

Congruences – Congruences are a language construct for defining data structure specific traversals. They are described in Chapter 3 in the context of Stratego.

Queries – Queries are expressions for navigating, analysing or modifying the program code. In the case of graph queries, the queries are usually only used for analysis. Modification is done using graph rewrite rules. Queries on relational databases also allow database updates, which amounts to program code modification. The following PROGRES code illustrates a query, [Sch04].

```
1 query AllConsistentConfigurations(out CNameSet : string [0: n]) =
2   use LocalNameSet, ResultNameSet : string [0: n] do
3       ResultNameSet : = nil
```

```
4      & GetAllConfigurations(out LocalNameSet)
5      & for all LocalCName : = elem(LocalNameSet) do
6         choose
7           when ConfigurationWithMain(out LocalCName)
8           and not ConfigurationWithUselessVariant(LocalCName)
9           and for all LocalMName : = elem(LocalCName.-has->.=needs=>) do
10              ModuleInConfiguration(LocalCName, LocalMName)
11              end
12          then ResultNameSet : = ResultNameSet or LocalCName
13        end
14       end
15     & CNameSet : = ResultNameSet
16   end
17 end
```

This query computes all consistent configurations of a software package. It uses another query, `GetAllConfigurations`, to obtain its starting point. This is looped over. For each configuration, a few sanity checks are performed. The inner loop on lines 9-11 checks all variants that are targets of `has` edges, and sees if all necessary modules of these variants are part of the configuration currently selected by line 6 from the set iterated in line 5.

Closures – Closures are a common feature in functional programming languages, such as Haskell, ML and Elegant. They combine well with data structure navigation features for writing tree transformations. Dynamic rules, discussed later, share many properties of closures, but come with some unusual semantics for scoping and visibility.

Editing Operations – The FermaT language does rewriting using editing operations such as cut, copy, paste and delete. There is a requirement placed on how these operations are used. This allows FermaT to guarantee that any editing on the program code will always result in a syntactically and semantically valid result, though not behaviourally equivalent. A few examples of editing operations:

`@Cut` *// delete the current item and store it in the cut buffer*

`@Paste_Over(I)` *// replaces the current item with I*

`@Rename(old, new)` *// renames a variable throughout the current tree*

`@Delete` *// deletes the current item*

`@Splice_Over(L)` *// replaces the current with with the list L of items*

Path Expressions – Path expressions are declarations that express paths through the program code structure. In a sense, it provides a small declarative sublanguage for navigation and matching. The feature is mostly found in program transformation systems with graph representations. These are also found in some tree rewriting languages, such as XSLT. The following PROGRES path expression defines a path

(i.e. a relation) named needs from one or more ATOM nodes to one or more MODULE nodes.

```
path needs : ATOM [0: n] -> MODULE [0: n] =
    ( instance of VARIANT & -v_uses-> )
 or ( <-has- & instance of MODULE & -m_uses-> )
end;
```

It states that there is a needs path from an ATOM a to a MODULE m if a is a VARIANT (a subclass of the ATOM node type), and there is a v_uses edge from m to a, or if there is a has edge from m to a, a is a MODULE and there is also an m_uses edge from a to m.

Logic Predicates, Assertions and Retractions Predicates are used express queries on the program element fact (PEF) base. A predicate consist of one or more patterns which will be attempted matched against the facts database using unification. Logic assertions are used to enter facts in the PEF base. The facts are terms, expressing relations. Retractions are used in JTransformer to remove facts from the PEF base.

```
1 fullQualifiedName(20003, ?Fqn)
2 importT(10000, 30001, 20003).
3 retract(importT(10000, 30001, 20003)).
```

The predicate on line 1 instantiates the variable Fqn with the fully qualified name of the declaration with unique id 20003, which may for example be a class. Line 2 is an assertion of the relation importT between its three constant values. Its meaning in JTransformer was discussed in earlier in this section. Line 3 removes the fact asserted by line two from the PEF database.

A general tradeoff common to many of these features is that of expressiveness versus efficiency. For example, allowing existential quantification and universal quantifiers in queries may quickly result in even small queries which become prohibitive to compute on moderately sized graphs.

Rewrite Rules

A rewrite rules is a function r which takes a (fragment of a) program f_0 to another (fragment of a) program f_1, i.e.: $r : f_0 \rightarrow f_1$. f_0 is referred to as a *left-hand side* pattern and f_1 a *right-hand side* pattern. Determining which f_0 a rule is applicable to, and what kind of computational expressiveness is allowed in computing f_1, are fundamental considerations.

- *Declaration* – refers to properties of the rule declaration. A declaration of a rewrite rule may keep the *domains separate*, i.e. the left and right hand side may be visually separate in the example above. Alternatively, they may be mixed together, as in the case for congruences. A rewrite rule normally has one left-hand side and one right-hand side, i.e. two *domains*, see p. 42. In Elegant,

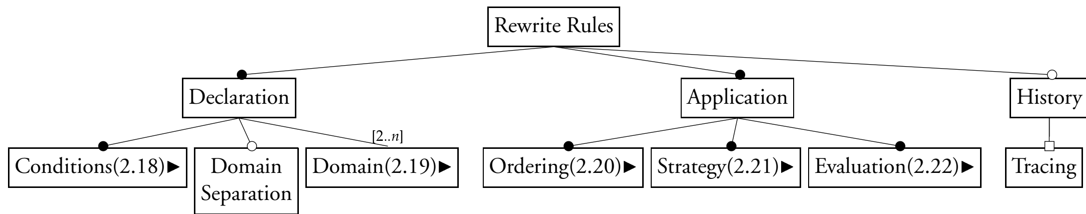


Figure 2.17: Feature decomposition of rewrite rules.

a third variation exists where an arbitrary number of domains can be combined into what is called a relation. Most transformation languages with rewrite rules support *conditions*, see p. 40.

- *Application*– Describes how the application of rules are *ordered*, see p. 43 , and also how the programmer can express *strategies*, see p. 44, for rule application on top of the *evaluation* mechanics, see p. 45, provided by the language.
- *History* – refers to features where the execution *trace* can be recorded.

The rendition of rewrite rules also varies considerably. The previous sections have illustrated examples from both APTS and Stratego. The following rewrite rule is from JastAdd:

```
rewrite Use {
  when(decl() instanceof TypeDecl)
  to TypeUse new TypeUse(getName());
}
```

Transactions – Transactions provide concurrency and consistency guarantees to a sequence of transformation operations. The concurrency guarantees allow multiple, simultaneous accessors to the program code. The consistency guarantees that the program code is consistent with respect to a set of invariants after the sequence of operations inside the transaction have been applied. The PROGRES language offers consistency. Concurrency is also supported by PROGRES at the runtime representation level but the language is not concurrent.

Dynamic Rules – Dynamic rules are described in Section 3.3.3 in Chapter 3.

Conditions Variation of application conditions for rules, see Figure 2.18, exist in abundance. For purposes of discussion, the feature spaces is divided into four parts, described next.

- *Predicates* – predicates are declarative questions evaluated against the *structure* of the code, or against *relations* constructed from the code.

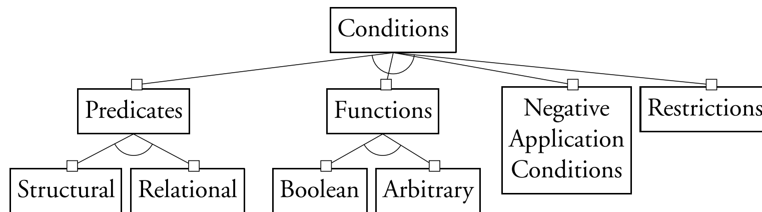


Figure 2.18: Feature decomposition of rule application conditions.

- *Functions* – functions are user-defined algorithms. They may return *arbitrary* results and often allow a more flexible way of encoding predicates into several computational steps.

- *Negative application conditions* – negative application conditions (NACs) are worth mentioning in relation to graph rewriting. Positive graph patterns only pose restrictions on which edges must be exist between nodes. Negative application conditions are used to express which edges *may not* exit. They may also be considered as a variant of structural predicates.

- *Restrictions* – restrictions are a special kind of a pattern found in PROGRES. Restrictions can be named and reused by rewrite rules. When evaluated, they can in turn call out to functions (called queries in PROGRES parlance), which can do arbitrary computations and graph traversals.

Functions and predicates are the primary variation points for conditions. These general concepts take many shapes, such as the negative application conditions and restrictions.

Unification – Unification is a generalisation of basic pattern matching. A query with multiple concurrent patterns can contain reoccurring variables which must be instantiated to the same value for each pattern, i.e. they must be unified. Unification is equivalent to instantiation in logic. In logic languages such as Prolog, unification is done against a set of terms, all stored in a facts database. Pattern matching with non-linear patterns can be considered a restricted form of unification; the matching is done against one term, using one pattern, but the recurring variable(s) in the pattern must be instantiated to the same value in all places.

Node Folding – Node folding provides a unification-like capability to graph pattern matching. It is found in graph rewriting systems where every pattern match is attempted across all nodes of the graph. In some systems, it is by default required that two different nodes, n_1 and n_2, in the left-hand side pattern match different nodes in the graph. Node folding allows specifying that n_1 and n_2 may match the same node.

Reference Attributes – Reference attributes allow placing cross-node links in an abstract syntax tree, i.e. links which do not go directly to a parent or a child, turning

it into a abstract syntax graph. In JastAdd, directed equations may be subsequently be expressed on top of the abstract syntax graph, whereas other attribute grammar systems such as Elegant only allows directed equations on the AST. The following JastAdd fragment declares the synthesised (lazily evaluated) attribute `booleanType()` on the `Program` node, which references the definition for the builtin type `boolean`.

```
syn lazy PrimitiveDecl Program.booleanType() =
  (PrimitiveDecl) localLookup("boolean");
```

Overlays – Overlays are described in Chapter 3.

Domains Figure 2.19 describes the domains used for pattern matching in rewrite rules.

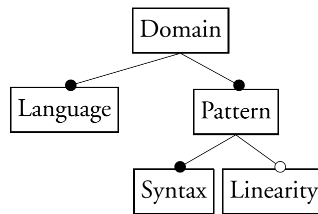


Figure 2.19: Feature decomposition of rule domains.

- *Language* – Specifies which subject language the pattern must be written in.
- *Pattern* – The pattern of the domain is expressed using either *abstract* or *concrete* syntax, with either a graphical or textual presentation, as discussed in Section 2.3.1. When the pattern variables instantiated non-linearly, the semantics is the same as for unification.

The choice of language may be fixed by the transformation system, or it may be user-definable. FermaT (fixed to WSL) and JTransformer (fixed to Java) are examples of fixed systems. Tom, Stratego and Elegant are examples of systems supporing user-definable subject languages.

List Comprehension – List comprehension is a language feature that improves syntax for list matching, list iteration and list transformations. The list comprehension syntax is very close to the mathematical syntax and semantics of list (or set) comprehension. This feature is also often found in functional programming languages.

Pattern Matching – Pattern matching offers structural matching on program code, either using abstract or concrete syntax. The patterns may contain pattern variables which will be bound during the matching process. Transformation programs using pattern matching on the program model often become tied to the structural details of that model. For example, rewrite rules in term rewriting systems often become closely

tied to the signature they were written against. This makes it difficult to switch or modify signatures, i.e. change or evolve the subject languages, while keeping the rewrite rules.

Embedding Clauses – Embedding clauses specify how to rearrange the edges in a graph during a rewrite step once a match has been found. The clauses declare how edges will be changed in the transition from the left-hand side to the right hand side in terms of copy, redirect and remove operations.

Ordering Features for ordering rules are shown in Figure 2.20.

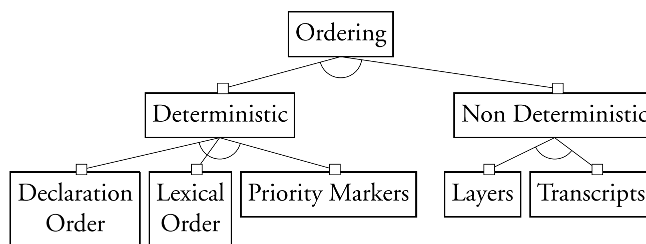


Figure 2.20: Feature decomposition of rule ordering

- *Deterministic* – The selection of rules is completely deterministic.
- *Non-deterministic* – The selection of rules is non-deterministic.

Deterministic languages mostly use the *declaration order* of rules to determine the order, e.g. Elegant, JastAdd . Another alternative is to require explicit *priority markers* on the rules, as for example in XSLT.

Directed Equations – Directed equations declare how a given attribute of a node must be computed from attributes on other nodes in the graph or tree. They give both a declaration of the attribute dependencies and the expression for computing the derived attribute value. The following JastAdd fragment declares the attribute `isValue()` to be a synthesised attribute of type `boolean`, and that its value is constantly `true`.

```
syn boolean Exp.isValue();
eq Exp.isValue() = true;
```

The equation may be any expression (which results in a compatible) type. For example, the type of a `VarDecl` may be computed from the type of the declaration of the type of the current variable declaration node, or more succinctly:

```
eq VarDecl.type() = getType().decl().type();
```

Traversal Strategies – Traversal strategies are declarations for how to traverse trees, and how rewrite rules should be applied to the tree during traversal, see [Vis05a].

Backtracking – Backtracking provides the ability to unroll (a series of) changes made to the runtime representation during transformation, thus reverting to a previous state. As such, backtracking relates to transactions, discussed later. Efficiency of implementation rests on how much data needs to be duplicated for rollback to be possible, whether rollback is local or global, and also the runtime complexity of the rollback algorithms. The performance can be be improved by use of maximal sharing techniques [vdBdJKO00] and lazy evaluation.

Rule Set Layering – Rule set layering is a feature for imposing application ordering on a set of rules. The rule set is divided into layers. Each layer will be evaluated with a fixed evaluation strategy, such as fixpoint, until no more rules in that layer apply. At this point, the next layer will be evaluated in the same fashion. Effectively, this divides the application of a set of rules into phases. Layering retains the declarative approach to expressing rewriting systems. It combines well with critical pairs analysis to prove confluence: confluence must be proven on a per-layer basis.

Tree Cursor – The editing operations of FermaT always take place at the current position in the tree, maintained by a tree cursor. The cursor can be moved around with navigation commands such as up, down, left and right. For example, the function `@Parent` provides the parent of an item (node), and `I^n` will give the `n`-th child of an item `I`.

Strategy The application strategy, Figure 2.21, determines how the rewrite rules will be applied to the runtime representation. Application strategies are very much related to ordering and scoping; they determine the location in the runtime representation an atom is applied, in which order, and how application failures should be handled.

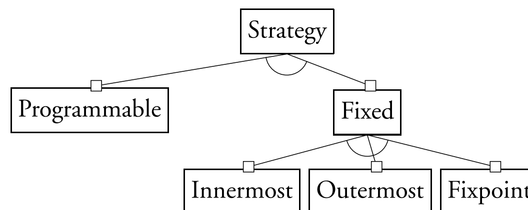


Figure 2.21: Feature decomposition of rule application strategies.

- *Programmable* – The application is programmable by the transformation programmer. Even when strategies are *programmable*, a library of ready made strategies may be available. This is the case with Stratego. Its library provides over a substantial collection of different application strategies.

- *Fixed* – The application strategy is pre-programmed into the transformation language and cannot be changed. Common alternatives are *innermost*, *outer-*

most and *fixpoint*, but the variation is immense. Refer to [Vis05a] for a broader catalogue of common evaluation strategies.

There is a tension between provability and flexibility. Having a fixed of a limited number of evaluation strategies makes analysis of the code possible, for example, critical pairs analysis. Allowing programmers to freely define custom strategies comes with Turing completeness. In general, this removes the ability for automatically proving or guaranteeing termination. It also removes automatic guarantees of confluence. A substantial survey of strategies in rule-based program transformation systems is given in [Vis05a].

Relation Calls – Embedded relations provide a limited relational-like functionality in graph rewriting systems. An embedded relation is placed on a node type to tie it to a set of other node types. Inferred links are encoded by path expressions which will be evaluated every time the link is accessed, allowing the members of the relation to change.

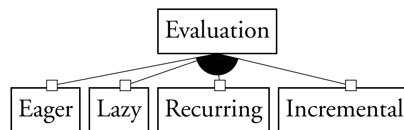


Figure 2.22: Feature decomposition of rule evaluation.

Evaluation

- *Eager* – expressions are computed in the order they are seen by the interpreter
- *Lazy* – expressions are not computed until their result is needed. Once evaluated, the result is memoized and used for all future evaluations of this expression.
- *Recurring* – Similar to lazy expressions; the expression is reevaluated every time the result is needed, taking updated values for all involved variables into account. Recurring evaluation is equivalent to lazy evaluation without memoization.
- *Incremental* – attaching a recurring evaluation to a variable gives incremental evaluation: Whenever such a variable is read, the evaluator is run, potentially recomputing all dependent variables which are also incremental.

Many transformation systems seem to be rather sensitive to how transformation algorithms are formulated. As with many high-level languages, developers may inadvertently write sound and clean transformation programs with prohibitive execution times. A number of optimisation features have been proposed for improving

the efficiency of the recommended ways of formulating transformation problems. It may therefore be no surprise that many of the characteristic features discussed in this chapter come from the PROGRES language. PROGRES was designed to make graph transformation practical. It offers a wide range of architectural and language features that aid in writing general graph transformations efficiently.

Conditional Path Iteration – Conditional path iterations are user-definable iterations over paths, similar to the mathematical notion of transitive closures on a set of predicates. Conditional paths are found in graph languages, such as PROGRES. Instead of returning all visited nodes, they return all possible termination points. This feature is also found in the tree rewriting language XSLT [Cla99]. An example of this feature was shown in the `needs()` example, under *path expressions* in Section 2.4.2.

Memoization Markers – Memoization makers allow programmers to declare that results of computations should be stored and reused whenever the same expression is reevaluated. The feature is found in graph systems with paths and attribute grammar systems, and is used to control recomputation of dependent values. When rules and functions are marked with a memoization marker, it implies that they are referentially transparent. The following JastAdd fragment declares the attribute `x()` of (node) class `A` to be a lazy, synthesised attribute, i.e. that its value should be memoized.

```
syn lazy A.x();
```

Cycle Detection – In attribute grammar systems, detecting cycles in the dependencies between attributes is necessary for correct evaluation. The job of cycle detection is to determine whether a given equation directly or indirectly depends on its own value.

Cycle Breaking – This feature is dependent on cycle detection. Once cycles are detected, various schemes are possible for breaking them. The simplest is to disallow the cycle altogether by refusing to compile grammar declarations with cycles. Another alternative is to ask the user to manually insert lazy evaluation where appropriate. In some systems, such as JastAdd, cycles are broken with a fixed, but automatic strategy. The following JastAdd attribute declaration specifies that the value for an attribute which turns out to be circular should be `true`.

```
syn lazy boolean ClassDecl.hasCycleOnSuperclassChain() circular [true];
```

Derived Attributes – Derived attributes are variables (attributes) inside nodes whose values depend on the value of other attributes. Updating the value of a dependent variable automatically recomputes the value of all its dependents. The dependencies are practically always expressed using directed equations. A typical attribute grammar system will define its attributes using equations, making all attributes derived attributes (except the ones which are defined by constant expressions). Both synthesised and inherited attributes are kinds of derived attributes.

Finite Differencing – Finite differencing is a transformation for replacing costly, repeating calculations with less expensive differential and semantically equivalent counterparts. The transformation is independent of the subject language, and mostly useful for algorithms with repeated calculations. A special case of finite differencing is the strength reduction optimisation found in most compilers. A detailed example is beyond the scope of this chapter, but refer to [PK82] for an explanation of finite differencing support in APTS.

2.4.3 Typing

The structure of the program code must be captured by the transformation language type system, see Figure 2.23. Transformation languages are primarily meant to work on a restricted domain of data. This opens up the opportunity for custom, or domain-specific, type systems. These may sometimes be simpler than ones found in general-purpose languages.

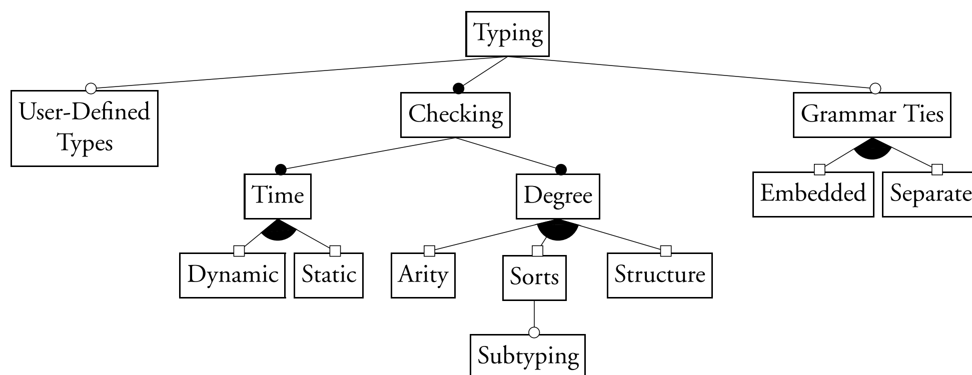


Figure 2.23: Feature decomposition of typing.

- *User-defined types* – The system allows the programmer to define new types.
- *Checking* – Refers to which features exist for checking type correctness.
  - *Time* – Determines when the type checking takes place. For solely *dynamic* type checking, all type checks are performed at runtime, and this may incurs a performance hit. For solely *static* type checking, the transformation program is guaranteed at compile time to maintain type consistency. Most languages fall in between.
  - *Degree* – Describes the nature of the type checking. The type checking may ensure structural validity of the program object model, type correctness or other semantic properties. FermaT, for example, ensures that

every transformation results in a semantically valid, executable subject program.

- *Grammar ties* – For many transformation systems, the data types for the subject code is derived directly from a grammar. In these cases, it is common for the type declarations to be *embedded* in the grammar. For other systems, the definition of the subject code structure is *separate* from the grammar, and grammar-independent.

A characteristic trait of the advanced type systems for transformation languages is that they offer flexible and powerful features for maintaining data structure consistency. The grammar-dependence of the types for the subject code is a characteristic feature of both tree- and graph-based systems. In Stratego, the term structure definition for subject-program terms is usually derived directly from the syntax declaration of a subject language. Some systems completely separate the subject language grammar from the type declaration of the internal program representation of subject programs. It is the programmer's responsibility to convert between the parser output and the type declaration for the subject code. This is the case for JastAdd, where any parser may be used, as long as it builds objects from the types declared in a separate, user-defined JastAdd AST declaration file.

Transformation Invariants – Transformation invariants are invariants on the program code which are guaranteed by the transformations. They are encoded as pre and post conditions on the transformation atoms or transactions. Such invariants are very useful for conducting proofs on the transformation program. Often, there is a clear correspondence between the transformation invariants and the data structure invariants discussed in Section 2.3. The PROGRES language can specify graph invariants in its graph grammar, such as the absence of cycles, which must be respected during graph rewriting:

```
constraint ACyclicAggregation = not (self in self.-contains-> +)
```

Meta Attributes – Meta attributes are attributes on node types, offered by the PROGRES language. They allow parametrisation of grammar declarations and are similar to (type) parameters on types. Meta attributes confer the ability to compose types at compile time, much like generic types. Consider the following container node class, defined in PROGRES.

```
1 node class CONTAINER;
2   meta ElementType : type in ELEMENT;
3   intrinsic contains:  ELEMENT [0: n];
4   constraint self.contains.type = self.type.ElementType;
5 end;
```

A `CONTAINER` node holds a list of elements of a given type `type`. It may be instantiated for a specific type, such as `Stmt` in the following way:

```
node type StmtContainer : CONTAINER;
  redef meta ElementType : = Stmt;
end;
node type Stmt : ELEMENT end;
```

This defined the container `StmtContainer` which may only contain elements of type `Stmt`. This constraint is ensured by line 4, above, and will be checked at runtime.

2.5 Discussion

This survey described and discussed numerous features characteristic to transformation system. Its main focus was on the program models and representations used in transformation systems, and how these relate to the transformation language used to manipulate the models.

The analysis undertaken behind this survey indicates that high-level program models support language-independence well. They often achieve this by replacing language-specific information present at the source code level – such as the difference between `for` and `while` loops in the C-language family – with more general concepts – e.g. bounded/unbounded loop. The abstracted model may often be easy to transform, but translating the result of a high-level transformation back to the underlying program is often difficult. As a consequence, if language-independence via abstract program models is required, many classes of transformations may have to be given up because required information is not present in the abstract model. Abstract models are therefore best suited for capturing problem-specific views on software.

There is a second observation related to the use of abstract program representations. The problem of general graph matching (determining an isomorphism between two graphs) is in NP. It is therefore common for program transformation systems based on graph to extract smaller, more abstract models from a code base. Additionally, general graph rewriting systems provide numerous optimisation features and language constructs for making graph rewriting computationally tractable. Some of these were discussed in Section 2.4.

A similar observation may be made for databases. The author has only found a handful of transformation systems based on relational databases. On the other hand, many analysis frameworks have been constructed by using databases to represent programs.

A remark on the use of meta information (annotations) might be in order. The introduction of meta information often makes transformations easier to write. By separating the logic for computing the meta information from the logic using this information, it is sometimes possible to formulate transformation algorithms in a

more language-independent way. Much of the logic for computing meta-information remains language-specific. By standardising on a given meta information format, large parts of the transformation algorithms may be reusable, however. The tradeoff is that the meta-information may have to be refreshed or recomputed throughout a transformation. Depending on the nature of the annotation, this may very expensive. Many systems, especially those based on attribute grammars, employ lazy evaluation to partly circumvent this problem.

2.6 Summary

This chapter presented a detailed survey of the state-of-the-art in software transformation systems and showed that this is a very feature-rich domain where many novel language features have been invented. The survey contained feature models describing central parts of the design space for transformation system. The models were supplemented with examples taken from about a dozen research systems.

The survey indicated that several features for abstracting over subject languages exist, especially for systems with very high-level program representations. A problem with these models is that transformations are difficult to translate back to concrete programs. There is therefore a rather clear case for additional abstraction facilities which provide good language abstraction facilities while simultaneously supporting easy rewriting of programs. In particular, the program model and language constructs for manipulating it are the central components that need good abstraction facilities if one is to attain transformation reuse and language-independence.

– Is it easy for humans to write code using this syntax?
– It depends on how you define "human".
– Magne Haveraaen asking Valentin David

3 Strategic Term Rewriting

This chapter recalls some basic elements of term rewriting theory and some supporting parts of universal algebra. It proceeds by discussing a programming paradigm called strategic programming which supports the separation of data traversal concerns from data processing logic – allowing each part to be implemented and reused separately – and how strategic programming, in the form of strategic term rewriting, helps expressing reusable term rewriting systems. The chapter describes a calculus for strategic term rewriting called System S calculus. This calculus provides the basic abstractions of tree transformations and term rewriting: matching and building terms, term traversal, combining computations, and failure handling. The strategic term rewriting language Stratego, that implements the System S calculus, is described.

3.1 Term Rewriting

The field of term rewriting studies methods for replacing subterms of terms with other terms. Techniques from this field are attractive for program transformation and analysis because every computer program can be represented as a term. The (abstract) syntax tree of a program can be directly treated as a term. The mathematical machinery of term rewriting may be brought to bear on analysis and transformation problems.

Term rewriting theory [Ter03] makes use of basic notions known from universal algebra [Coh81], a field of mathematics which seeks to describe any mathematical object by its operations. Objects and operations are described formally using signatures. In term rewriting, one talks of sorts and constructors in lieu of objects (types) and operations.

3.1.1 Algebraic Signatures and Language Signatures

In both universal algebra and term writing, terms are defined over signatures. Signatures may be considered analogous to the context-free grammars used to describe the structure of text. Both context-free grammars and signatures describe properties of

(potentially) recursively defined tree structures. A standard definition of an algebraic signature is given below.

Definition 1 Algebraic Signature.

An algebraic signature Σ *is a pair* (S, Ω) *of sets, where* S *is a set of* sorts *and* Ω *a set of* operations. *Each operation is a* $(k+2)$*-tuple,* $k \geq 0$*, on the form*

$o : s_1 \times \ldots \times s_k \rightarrow s$

where $s_1, \ldots, s_k, s \in S$*,* o *is the* operation name *and* $s_1 \times \ldots \times s_k \rightarrow s$ *its* arity. *The sorts* $s_1, \ldots, s_k$ *are* argument sorts*, and* s *the* target sort. *When* $k = 0$*,* $o :\rightarrow s$ *is a* constant symbol*, or just* constant.

The following example of an algebraic signature declares the four basic arithmetic operations.

```
signature Arithmetic
sorts Int
ops
  plus : Int × Int → Int
  minus : Int × Int → Int
  divide : Int × Int → Int
  times : Int × Int → Int
```

In this dissertation, algebraic signatures will be used to describe abstract data types. For example, the above signature partially describes the data type *Int* and some of its operations (*plus*, *minus*, *divide* and *times*). All operations (and terms involving operations) will be written in *italics* in the main text.

In several traditions of program transformation based on term rewriting there is second role for signatures: they may be used to declare the abstract syntax of programming languages, akin to document type definitions commonly found for markup languages like XML [BPSM+] and SGML [sgm86]. Signatures used in this capacity are referred to as *language signatures* in this dissertation. They have some minor and subtle differences compared with the algebraic signatures.

The language signatures described here follow the tradition introduced by the Stratego rewriting language. Operations are referred to as *constructors*. In the main text, constructors (and terms involving constructors) will be written in `MixedCase`. Constructors must always start with an uppercase letter. A more important difference between the two uses of signatures is that in signatures describing languages, the argument sorts of constructors follow the abstract grammar of the subject language they define. Consider the signature definition for a minimal language L that supports variables, assignment and addition operations on floating point and integer numbers:

```
1 signature L
2 sorts Var Exp Stmt String
```

```
3 constructors
4   Var : String → Var
5         : Var → Exp
6   Int : String → Exp
7   Float : String → Exp
8   Plus : Exp × Exp → Exp
9   Assign : Var × Exp → Stmt
```

Line 4 declares that variable terms are of sort `Var`. Line 5 is an injection which declares that every term of sort `Var` is also a term of sort `Exp`, i.e. `Var` is a subsort of `Exp`. The `Int` and `Float` constructors describe literals of integers and floats, respectively. In the abstract syntax, a `Plus` term is constructed from two terms of sort `Exp`. Assignments are statements (of sort `Stmt`) which assign the result of expressions to variables.

3.1.2 Patterns and Terms

Universal algebra defines the notion of terms over signatures, a traditional definition of which is given in Definition 3. These terms may contain variables.

Definition 2 (Variables).
Given a signature $\Sigma = (S, \Omega)$ with an associated family $V = (V_s)_{s \in S}$ of disjoint infinite sets, an element $x \in V_s, s \in S$ is a variable *x of sort s.*

Algebraic terms may be recursively constructed from variables and the application of operations to the result of operations or to variables.

Definition 3 (Algebraic Terms).
Given a signature $\Sigma = (S, \Omega)$ and an associated set of variables X, the set of (algebraic) terms for Σ, $(T_{\Sigma(X),s})_{s \in S}$ are defined by simultaneous induction:

1. $X_s \subseteq T_{\Sigma(X),s}$
2. *if* $o :\to s \in \Omega$, *then* $o \in T_{\Sigma(X),s}$
3. *if* $o : s_1 \times \ldots \times s_k \to s \in \Omega$, $k \geq 0$ *and if* $t_i \in T_{\Sigma(X),s_i}$ *for* $1 \leq i \leq k$, *then* $o(t_1, \ldots, t_k) \in T_{\Sigma(X),s}$.

An element in $T_{\Sigma(X),s}$ is called a $\Sigma(X)$-term of sort s, or just a term. $\mathrm{Var}(t)$ *denotes all variables occurring in the $\Sigma(X)$ term t. If* $\mathrm{Var}(t) = \emptyset$, *$t$ is called a* ground term.

Every valid algebraic term for a given signature must respect the sorts of the signature, i.e. the arity of each operation. Algebraic terms may contain variables. The terms for language signatures, and their nomenclature, behave slightly differently from algebraic terms.

p ::=	$c(p,\ldots,p)$	constructor application
|	str	string literal
|	r	real number
|	i	integer number
|	x	variable
c ::=	identifier	constructor name
x ::=	identifier	variable name
|	_	wildcard

Figure 3.1: Syntax definition for Stratego (language) patterns. The number of patterns p in a constructor application must correspond to the numeric arity of the constructor named c. Wildcards are "open holes" in patterns, akin to nameless variables.

The syntax for Stratego language terms is described in Figure 3.1. When language terms, or just terms, are constructed, the language signature is assumed to be single-sorted. Only the numeric arity must be respected, i.e. only the number of arguments, irrespective of the sorts. This is done for practical convenience. Term rewriting approaches, including that of Stratego, use step-wise substitution of subterms when going from one signature to another. It is useful to allow intermediate terms which are not valid according to either the source or the target signature, without having to explicitly declare a "super-signature" which defines all possible constructor combinations.

Another difference between universal algebra and the nomenclature used in strategic rewriting is the meaning of the word "term". Language terms are always ground terms. A language term containing variables will be referred to as a *pattern*, often written p. Variables in patterns always start with lower case letters, e.g. `x`. Consider the example term and pattern:

`Plus(Int("0"),Int("1"))`	`Plus(x,y)`
(term)	*(pattern)*

The kind of term expression – pattern or ground term – is easily recognised from the syntax since all constructors start with an uppercase letter and all variables start with a lowercase letter.

A pattern p may be matched against a term t. This matching is purely syntactical. It succeeds if and only if there exists a valid variable substitution $\sigma(p) \equiv t$. The variables $Var(p)$ of p will be bound to their corresponding subterms in t, e.g:

$$\langle \text{match}\, \texttt{Plus(x,y)} \rangle\, \texttt{Plus(Int("0"),Int("1"))} \Rightarrow \sigma : [x \mapsto \texttt{Int("0")}, y \mapsto \texttt{Int("1")}]$$

Conversely, a pattern p may be instantiated into a term t, by replacing all its variables x with terms:

$$[x \mapsto \texttt{Int("0")}, y \mapsto \texttt{Int("1")}] : \langle \text{build}\,\texttt{Plus(x,y)} \rangle \Rightarrow \texttt{Plus(Int("0"),Int("1"))}$$

Patterns are used in program transformations to check for structural (syntactic) properties and to construct new program fragments. By combining pattern matching and pattern instantiation into one (potentially named) unit, the rewrite rule is obtained.

3.1.3 Rewrite Rules

Rewrite rules are the units of transformation – or the atomic building blocks, if you will – in term rewriting systems. Each rewrite rule describes how one term can be derived from another term in a single step.

Definition 4 (Rewrite Rule).
A rewrite rule $R : p_l \rightarrow p_r$, with name R, left-hand side pattern p_l, right-hand side pattern p_r, and $p_l, p_r \in T_{\Sigma(X)}$, reduces *the term t to t' if there exists a $\sigma : X \rightarrow T_\Sigma$ such that $t = \sigma(p_l)$ (p_l matches t) and $t' = \sigma(p_r)$ (p_r instantiates to t'). The term t is called the* redex *(reducible expression) and t' the* reduct.

In the context of System S and Stratego, the term variables are variables in the Stratego program, and the substitution σ corresponds to a variable environment ε. This is clarified in the next section. A set of rewrite rules R is said to induce a *one-step rewrite relation* on terms, written as follows:

$$t \rightarrow_R t'$$

This says that t reduces to t' with one of the rules in R. Composing these in sequence, i.e. $t_0 \rightarrow_R t_1 \rightarrow_R \ldots$ gives a *reduction sequence* with $\rightarrow_R$, where R is repeatedly applied to the root of a term.

Definition 5 (Conditional Rewrite Rule).
A conditional rewrite rule $R : p_l \rightarrow p_r$ where c, with c being a logical expression in some logic, specifies that R is only applicable if, for some σ, p_l matches t with σ and $\sigma(c)$ holds (evaluates to true).

3.1.4 Rewriting Strategies

The rewrite sequence, as defined above, repeatedly applies the rules of R to the root of a term, i.e. to the top-level constructor and its subterms. The definition does not describe how rules may be applied to subterms. Nor does it say anything about the

order in which the rules in R of are applied for each step – it may be the case that multiple rules are applicable.

Other definitions for rule application exist in term rewriting theory, but for program transformation, a flexible and precise way of programming both the application location (inside a term) and the order of (rule) application is necessary. In this dissertation, the System S calculus is used for this purpose.

3.2 System S – Strategic Term Rewriting

Strategic term rewriting extends basic term rewriting with additional constructs that accurately control the application strategies for sets of rules. These constructs are used to control the order of rule application, traversal over term structures, and how to handle rule application failures.

The System S core calculus is a formalism for strategic term rewriting. It provides the basic abstractions of tree transformations and term rewriting: matching and building terms, term traversal, combining computations and failure handling. It was first introduced by Visser and Benaissa [VBT98, VB98]. The programming language Stratego is directly based on this calculus.

This section contains a slightly modified formulation of the same core calculus which is more in the style of [BvDOV06]. The definitions given herein are only those necessary for later chapters. Compared to the original description, non-deterministic choice, $s_0 + s_1$ and the `test` operator have been dropped. These are now replaced by a guarded choice combinator. The `some(s)` traversal primitive has been eliminated. A syntax of System S is shown in Figure 3.2. For the rest of this section, the word “program” is taken to mean the transformation program. Terms are used to represent subject programs.

In Chapter 5 and Chapter 7, the System S calculus and Stratego is extended with additional constructs that improve the capacity for expressing language independent transformation programs.

Basic Definitions

The operational semantics of System S is specified using the notation described below. The semantics describes the behaviour of strategies. Rewrite rules are encoded as strategies (shown later), but are provided with syntactic sugar to give them their familiar notation.

The domain of strategy applications is the set of terms extended with a special failure value $\uparrow$. The notation $\bar{t}$ is used to indicate terms from this extended domain; the notation t still refers to terms. Consider the following assertion:

$$\Gamma, \varepsilon \vdash \langle s \rangle\, t \Rightarrow \bar{t}'(\Gamma', \varepsilon')$$

s ::=	`id`	identify
|	`fail`	failure
|	$?p$	match term
|	$!p$	build term
|	$s; s$	sequential composition
|	$s < s + s$	guarded choice
|	`where`(s)	where
|	$\{x,\ldots,x\text{: } s\}$	new variable scope
|	`one`(s) | `all`(s)	generic traversal operators
|	$f(f,\ldots,f\|p,\ldots,p)$	strategy invocation
x ::=	identifier	variable names
f ::=	identifier	strategy names
c ::=	identifier	constructor names

Figure 3.2: Syntax for System S. The definition of term patterns p was given in Figure 3.1. The semantics of strategy invocation is defined in [BvDOV06].

It states that the strategy s applied to term t in context of the system state Γ (used to model dynamic rules) and variable environment ε evaluates to the term t' in a new system state Γ' and a new environment ε'. The variable environment takes on the role of the σ substition previously described for rewrite rules.

Strategies may fail. This is noted with the following assertion:

$$\Gamma, \varepsilon \vdash \langle s \rangle\, t \Rightarrow\uparrow (\Gamma', \varepsilon')$$

Changes to state and variable bindings are preserved in the case of failure.

Variables A variable environment ε is a finite ordered map $[x_1 \mapsto \overline{t_1}, \ldots, x_n \mapsto \overline{t_n}]$ from variables to terms or failure. A variable x may occur multiple times in ε, in which case the first (leftmost) binding is applicable. The application of an environment – a variable lookup – is defined as picking out the first binding for x (if any):

$$[x_1 \mapsto \overline{t_1}, \ldots x_n \mapsto \overline{t_n}](x) \begin{cases} t'_i & \text{if} \quad x_i \equiv x \wedge \forall j < i : x_j \not\equiv x \\ \uparrow & \text{if} \quad \forall j \leq n : x_j \not\equiv x \end{cases}$$

The variables in ε fulfil the role of algebraic term variables. The instantiation $\varepsilon(p)$ of the pattern p yields a (language) term, i.e. a ground term, by replacing every variable x in p with its bound term from ε. This is identical to variable substitution with σ with the exception that the pattern variables are variables of the System S calculus (i.e. variables in the Stratego language).

Environments ε are used in the matching process of patterns p. It is convenient to have a notation stating that the only difference between environments ε and ε' are the bindings for the variables of p. The notation $\varepsilon' \sqsupseteq \varepsilon$ declares that the environment ε' is a refinement of the environment ε. This means that if $\varepsilon = [x_1 \mapsto \overline{t_1}, \ldots, x_n \mapsto \overline{t_n}]$, then $\varepsilon = [x_1 \mapsto \overline{t'_1}, \ldots, x_n \mapsto \overline{t'_n}]$ and for each $i : 0 \leq i \leq n$, $\varepsilon(x_i) = \varepsilon'(x_i)$ or $\varepsilon(x_i) = \uparrow$ and $\varepsilon'(x_i) = t$ for some term t. $\varepsilon' \sqsupseteq_p \varepsilon$ declares that the environment ε' is the *smallest refinement* of the environment ε with respect to a term pattern p if $\varepsilon' \sqsupseteq \varepsilon$ and for all x not in p, $\varepsilon'(x) = \varepsilon(x)$.

Algebraic Properties The notation $e_1 \equiv e_2$ is used to describe algebraic properties of the defined constructs and to define syntactical shorthands. These equations are universally quantified unless otherwise stated.

3.2.1 Primitive Operators and Strategy Combinators

System S provides a handful of *primitive operators* on terms. The most basic of these are identity (`id`) and failure (`fail`) operators. Applying the identity operator to a term leaves the term unchanged; applying the failure operator signals a failure:

$$\Gamma, \varepsilon \vdash \langle \texttt{id} \rangle t \Rightarrow t(\Gamma, \varepsilon) \qquad\qquad \Gamma, \varepsilon \vdash \langle \texttt{fail} \rangle t \Rightarrow \uparrow (\Gamma, \varepsilon)$$

The operators, such as `id` and `fail`, are combined into expressions using *strategy combinators*. The purpose of the combinators is to describe control flow. Strategy expressions are built from primitive operators and combinators. The combinators are used to express application – evaluation – strategies of transformations in terms of how strategy application failures are handled. Any System S operator (except identity) may fail. Strategy combinators are used to specify what should happen when failures occur.

Sequential Composition The sequential application of two strategies s_1 and s_2 is expressed using the sequential composition combinator, $s_1; s_2$.

$$\frac{\Gamma, \varepsilon \vdash \langle s_1 \rangle t \Rightarrow t'(\Gamma', \varepsilon') \quad \Gamma', \varepsilon' \vdash \langle s_2 \rangle t' \Rightarrow \overline{t''}(\Gamma'', \varepsilon'')}{\Gamma, \varepsilon \vdash \langle s_1; s_2 \rangle t \Rightarrow \overline{t''}(\Gamma'', \varepsilon'')}$$

$$\frac{\Gamma, \varepsilon \vdash \langle s_1 \rangle t \Rightarrow \uparrow (\Gamma', \varepsilon')}{\Gamma, \varepsilon \vdash \langle s_1; s_2 \rangle t \Rightarrow \uparrow (\Gamma', \varepsilon')}$$

The assertions describe that strategy s_1 is first applied to the current term t. If it succeeds, s_2 is applied to its result; the result of the combination is the result of s_2. If s_1 fails, the combination fails. The following equations are consequences of the definitions above. They show that the `id` operator is a unit for sequential composition and that `fail` is a left zero.

$$\texttt{id}; s \equiv s \quad s; \texttt{id} \equiv s \quad \texttt{fail}; s \equiv \texttt{fail}$$

Not that in the general case, $\exists s : s; \texttt{fail} \not\equiv \texttt{fail}$. This follows from the way the state and the environment propagates over s: any environment ε before s will in general be ε' after s, whereas `fail` preserves the environment. Because of this, `fail` is not a right zero for sequential composition.

Guarded Choice The *guarded choice* (sometimes referred to as just the choice combinator) $s_1 < s_2 + s_3$ resembles an if-then-else expression, e.g.:

$$\texttt{id} < s_2 + s_3 \equiv s_2 \quad \texttt{fail} < s_2 + s_3 \equiv s_3$$

First, s_1 is applied. If s_1 succeeds, s_2 is applied and the result of s_2 is the result of the combined expression; if s_2 fails, the combination fails. Should s_1 fail, s_3 is applied and the result of s_3 is the result of the combination; if s_3 fails, the combination fails.

$$\frac{\Gamma, \varepsilon \vdash \langle s_1 \rangle t \Rightarrow t'(\Gamma', \varepsilon') \quad \Gamma', \varepsilon' \vdash \langle s_2 \rangle t' \Rightarrow \overline{t''}(\Gamma'', \varepsilon'')}{\Gamma, \varepsilon \vdash \langle s_1 < s_2 + s_3 \rangle t \Rightarrow \overline{t''}(\Gamma'', \varepsilon'')}$$

$$\frac{\Gamma, \varepsilon \vdash \langle s_1 \rangle t \Rightarrow\uparrow (\Gamma', \varepsilon') \quad \Gamma', \varepsilon \vdash \langle s_3 \rangle t \Rightarrow \overline{t'}(\Gamma'', \varepsilon'')}{\Gamma, \varepsilon \vdash \langle s_1 < s_2 + s_3 \rangle t \Rightarrow \overline{t'}(\Gamma'', \varepsilon'')}$$

An important feature of the guarded choice is that if s_1 fails, both the effects due to s_1 on the term t are *and* to the environment (but not the state Γ) are undone. This means that the choice combinator implements a notion of (local) backtracking.

Negation, Left and Right Choices For notational convenience, the operators *not*, *left choice*, and *right choice* may be defined using guarded choice:

left choice	$s_0 \texttt{<+} s_1 \equiv$	$s_0 < \texttt{id} + s_1$
right choice	$s_0 \texttt{+>} s_1 \equiv$	$s_1 < \texttt{id} + s_0$
not	$\texttt{not}(s) \equiv$	$s < \texttt{fail} + \texttt{id}$
try	$\texttt{try}s \equiv$	$s \texttt{ <+ id}$

3.2.2 Primitive Traversal Strategies

The combinators in the previous section addressed the first of the two concerns of rule application: how rule application failure may be handled. The second concern – where in a term rules should be applied – is addressed by *primitive traversal strategies*. There are two primitive traversal strategies: `one` and `all`. They enable term traversal by local navigation into subterms.

`topdown(s) =`	`s; all(topdown(s))`	*top-down traversal*
`bottomup(s) =`	`all(bottomup(s)) ; s`	*bottom-up traversal*
`repeat(s) =`	`try(s; repeat(s))`	*apply s until it fails*
`oncetd(s) =`	`s <+ all(oncetd(s))`	*apply s once, start at the top*
`oncebu(s) =`	`all(oncebu(s)) <+ s`	*apply s once, start at the bottom*
`innermost(s) =`	`bottomup(try(s; innermost(s)))`	*innermost traversal*
`outermost(s) =`	`repeat(oncetd(s))`	*outermost traversal*

Table 3.1: A selection of frequently used traversal and application strategies.

All Subterms The `all(s)` strategy applies the strategy expression s to each subterm of the current term, potentially rewriting each. `all(s)` succeeds if and only if s succeeds for all subterms.

$$\frac{\Gamma_0, \varepsilon_0 \vdash \langle s\rangle\; t_1 \Rightarrow t_1'(\Gamma_1, \varepsilon_1) \quad \ldots \quad \Gamma_{n-1}, \varepsilon_{n-1} \vdash \langle s\rangle\; t_n \Rightarrow t_n'(\Gamma_n, \varepsilon_n)}{\Gamma_0, \varepsilon_0 \vdash \langle \texttt{all}(s)\rangle\; c(t_1, \ldots, t_n) \Rightarrow c(t_1', \ldots, t_n')(\Gamma_n, \varepsilon_n)}$$

$$\frac{\Gamma_0, \varepsilon_0 \vdash \langle s\rangle\; t_1 \Rightarrow t_1'(\Gamma_1, \varepsilon_1') \quad \ldots \quad \Gamma_{i-1}, \varepsilon_{i-1} \vdash \langle s\rangle\; t_n \Rightarrow\uparrow (\Gamma_i, \varepsilon_i)}{\Gamma_0, \varepsilon_0 \vdash \langle \texttt{all}(s)\rangle\; c(t_1, \ldots, t_n) \Rightarrow\uparrow (\Gamma_i, \varepsilon_i)}$$

The strategy `all(s)` behaves as follows with respect to failure, identity and constant terms:

$$\texttt{all(id)} \equiv \texttt{id} \quad \texttt{<all}(s)\texttt{>}c() \equiv c() \quad \texttt{<all(fail)>}c(t_1, \ldots, t_n) \equiv \texttt{fail}\ (\text{if } n > 0)$$

One Subterm The traversal strategy `one(s)` is similar to `all`, but applies s to exactly one subterm. It fails if s does not succeed for any of the subterms.

$$\frac{\Gamma, \varepsilon \vdash \langle s\rangle t_1 \Rightarrow\uparrow (\Gamma_1) \;\ldots\; \Gamma_{i-2}, \varepsilon \vdash \langle s\rangle t_{i-1} \Rightarrow\uparrow (\Gamma_{i-1}) \;\; \Gamma_{i-1}, \varepsilon \vdash \langle s\rangle t_i \Rightarrow t_i'(\Gamma_i, \varepsilon_i)}{\Gamma, \varepsilon \vdash \langle \texttt{one}(s)\rangle c(t_1, \ldots, t_n) \Rightarrow c(t_1, \ldots, t_{i-1}, t_i', t_{i+1}, \ldots, t_n)(\Gamma_i, \varepsilon_i)}$$

$$\frac{\Gamma, \varepsilon \vdash \langle s\rangle t_1 \Rightarrow\uparrow (\Gamma_1, \varepsilon_1) \quad \ldots \quad \Gamma_{n-1}, \varepsilon \vdash \langle s\rangle t_n \Rightarrow\uparrow (\Gamma_n, \varepsilon_n)}{\Gamma, \varepsilon \vdash \langle \texttt{one}(s)\rangle c(t_1, \ldots, t_n) \Rightarrow\uparrow (\Gamma_n, \varepsilon_n)}$$

The `one(s)` strategy backtracks (undoes) all modifications to the variable environment made by failing applications of s, but changes to the system state are kept.

Generic Traversal Strategies

An important feature of System S (and Stratego) is its ability to define signature-independent (and thereby language-independent) traversal strategies. This support is

the result of mixing primitive traversal operators and strategy combinators. The mix yields the notion of *generic traversal strategies*. Examples of generic traversal strategies are given in Table 3.1.

Each generic traversal strategy $s_t(s)$ is parametrised with a strategy s that is applied throughout a term according the traversal scheme specified by s_t. The argument strategy s is used to insert language-specific rewriting logic, thereby instantiating the generic strategy for a specific subject language and signature.

3.2.3 Building and Matching Terms

System S supports two complementary operations for applying patterns to terms: match and build. Patterns are matched against terms using the match operator (`?`). Variables in the pattern are bound to their respective subterms. Terms are instantiated from patterns using the build operator (`!`). Variables are replaced by the terms they have previously been bound to.

Term Matching The assertions for term matching are given below:

$$\frac{\exists \varepsilon' : \varepsilon' \sqsupseteq_p \varepsilon \wedge \varepsilon'(p) \equiv t}{\Gamma, \varepsilon \vdash \langle ?p \rangle\, t \Rightarrow t(\Gamma, \varepsilon')} \qquad \frac{\nexists \varepsilon' : (\varepsilon' \sqsupseteq_p \varepsilon \wedge \varepsilon'(p) \equiv t)}{\Gamma, \varepsilon \vdash \langle ?p \rangle \Rightarrow\uparrow (\Gamma, \varepsilon)}$$

The semantics is compatible with the previously defined notion of match with variable substitution σ, with one exception: variables in p may already be bound. These variables are not rebound, but the corresponding subterms of t must match the terms bound by the variables of p. For example, a match of the pattern `Plus(x,y)` against the term `Plus(Int("0"),Int("1"))` (attempts to) bind the variable `x` to the term `Int("0")`. The match fails if the variable `x` is already bound to a term that is not `Int("x")`.

Term Building Term building is, in a sense, the inverse of matching. The build semantics is defined as:

$$\Gamma, \varepsilon \vdash \langle !p \rangle t \Rightarrow \varepsilon(p)(\Gamma, \varepsilon)$$

With the environment $\varepsilon = [x \mapsto$ `Int("0")`, $y \mapsto$ `Int("1")`$]$, the expression `!Plus(x,y)` will result in the the term `Plus(Int("0"),Int("1"))`.

3.2.4 Variable Scoping

The static scoping of term variables $x_1, \ldots, x_n$ can be controlled using the scope operator $\{x_1, \ldots, x_n : s\}$. Given $\varepsilon_0 = [y_1 \mapsto\uparrow, \ldots, y_n \mapsto\uparrow]$ and $\varepsilon_1 = [y_1 \mapsto \overline{t_1}, \ldots y_n \mapsto \overline{t_n}]$:

$$\frac{\Gamma, \varepsilon_0\varepsilon \vdash \langle [y_1/x_1, \ldots, y_n/x_n]s\rangle\; t \Rightarrow \overline{t'}(\Gamma', \varepsilon_1\varepsilon')}{\Gamma, \varepsilon \vdash \langle \{x_1, \ldots, x_n : s\}\; t \Rightarrow \overline{t'}(\Gamma', \varepsilon')} (y_1, \ldots, y_n \text{ fresh})$$

The operator introduces a new scope in which the strategy s is evaluated where the variables $x_1, \ldots, x_n$ have been replaced by fresh copies. This results in the usual notion of variable scoping: After s finishes, any binding for x_i, $1 \leq i \leq n$ introduced by s is removed from the environment. The scope operator succeeds if s succeeds and fails if s fails.

A useful syntactical abstraction over the scope operator is the `where` clause, later used for defining conditional rewrite rules. A `where(s)`-clause temporarily saves the current term, applies s to it, then restores the current term:

$$\texttt{where}(s) \equiv \{x : ?x; s; !x\}$$

It follows from the previous definitions that all variable bindings due to s are kept if s succeeds, and that `where(s)` fails iff s fails.

3.2.5 Rewrite Rules

The System S calculus can express rewrite rules with or without conditions, R_c and R_u, respectively:

$$R_u : p_l \rightarrow p_r \equiv ?p_l; !p_r$$
$$R_c : p_l \rightarrow p_r \text{ where } s \equiv ?p_l; \text{ where}(s); !p_r$$

The following is an example of a rewrite rule, named `Simplify`, defined in Stratego:

```
Simplify:
  Add(Int(x), Int(y)) → Int(z)
where <addS> (x,y) ⇒ z
```

The condition of this rule consists of the application of the strategy `addS` to the tuple `(x,y)`. (This tuple is the application of a nameless constructor with numeric arity two.) The result is "assigned" to the variable `z` using another syntactic abstraction, the $\Rightarrow$ operator, defined as follows:

$$s; ?p \equiv s \Rightarrow p$$

3.2.6 Additional Constructs

This section defined the core constructs of the System S calculus which are necessary for describing the language extensions proposed later in this dissertation. System

Strategy Expression	Meaning — *(basic constructs)*
$!p$	*(build)* Instantiate the term pattern p and make it the current term
$?p$	*(match)* Match the term pattern p against the current term
$s_0 < s_1 + s_2$	*(left choice)* Apply s_0. If s_0 fails, apply s_1. Else, roll back, then apply s_2.
$s_0 ; s_1$	*(composition)* Apply s_0, then apply s_1. Fail if either s_0 or s_1 fails
`id, fail`	*(identity, failure)* Always succeeds/fails. Current term is not modified
`one(`s`)`	Apply s to one direct subterm of the current term
`all(`s`)`	Apply s to all direct subterms of the current subterm

Figure 3.3: Basic language constructs.

S has several additional language constructs. These are presented informally using examples in the next section. Each of the explained constructs is used in some of the examples containing Stratego code throughout the following chapters, but understanding their precise and detailed semantics is not required. For a complete introduction to all of Stratego, refer to the Stratego/XT manual [BKVV05]. Specific caveats and considerations are noted along with the examples where pertinent.

3.3 Stratego

Stratego is a domain-specific language for writing program transformation libraries and components. It is based on the System S rewriting calculus. The language provides rewrite rules for expressing basic transformations, programmable rewriting strategies for controlling the application of rules, concrete syntax for expressing the patterns of rules in the syntax of the object language, and dynamic rewrite rules for expressing context-sensitive transformations, thus supporting the development of transformation components at a high level of abstraction. The program object model used for representing subject programs are terms.

In the next sections, the parts of Stratego which are relevant for comprehending the remainder of this dissertation are explained in detail. A short description is given in Figure 3.3 and Figure 3.4 of the core Stratego language constructs offered to the programmer. The following sections informally describe additional features of Stratego.

3.3.1 Signatures, Patterns and Terms

Stratego supports the declaration of signatures for describing the abstract (or concrete) syntax of subject languages. Stratego signatures are very close to the concept of language (as opposed to algebraic) signatures described previously. Consider the following example:

Strategy Expression	Meaning — *(syntacic sugar)*
`\`p_l `->` p_r`\`	Anonymous rewrite rule from term pattern p_l to p_r
`?`x`@`p	Equivalent to `?`x `;` `?`p; bind current term to x then match p
`<`s`>` p	Equivalent to `!`p `;` s; build p then apply s
s `=>` p	Equivalent to s `;` `?`p; match p on result of s

Figure 3.4: Syntactic sugar.

```
1 signature
2 sorts Exp Stmt
3 constructors
4   Var : String → Var
5       : Var → Exp
6   Int : String → Exp
7   Float : String → Exp
8   Plus : Exp × Exp → Exp
9   Assign : Var × Exp → Stmt
```

This example illustrates the following differences between Stratego and algebraic signatures:

- Stratego signatures are not named. A program written in Stratego may have several signature declarations. The sorts and constructors from all of these declarations will be combined into one implicit "super signature".
- Only the arity of constructors is guaranteed by the Stratego language, i.e. it is a one-sorted system which allows synonym names for its sort. Given the signature above, the constructor `Plus` may be applied to any two subterms. Their sorts are never checked. Additionally, sorts need not be declared before they are used in constructor definitions, e.g. lines 7–8 above, where the sort `Var` is undeclared. It is considered good form to declare all sorts, however. A separate tool, called `format-check`, can be applied to a term to check if it is valid with respect to a given signature.
- Stratego has builtin (primitive) sorts and special term syntax for strings (`String`), lists (`List(x)`), tuples (`Tuple(x)`), integer (`Int`) and real (`Real`) numbers. The sort `Term` is used (by convention) to indicate an "any" sort. That is, any term may be inserted where a `Term` is expected.
- Nameless constructors of arity one are allowed, and these are called injections. Injections declare the terms of the argument sort may be placed wherever the target sort is allowed. In effect, injections declare their argument sort to be a subsort of the target sort, and are used by the `format-check` tool.

Strategy	*Meaning*
`rules(`rd_1 `...` rd_n`)`	define rules $rd_1, \ldots, rd_n$
`{|`r_1`, ...,` r_n`: s|}`	start new scope for rule names $r_1, \ldots, r_n$
s_1 `/`$r_1, \ldots, r_n$`\` s_2	fork rule sets $r_1, \ldots, r_n$, apply s_1 then s_2, intersect rule sets
`/`$r_1, \ldots, r_n$`*` s	apply s until rule sets $r_1, \ldots, r_n$ reach fixpoint
Rule definition (rd)	*Meaning*
`R :` $p_1 \rightarrow p_2$ where s	introduce rule R
`R :+` $p_1 \rightarrow p_2$ where s	extend rule R with another left-hand side p_1 (and r.h.s. p_2)
`R :-` p	undefine rules R with left-hand side p

Table 3.2: Essential basics of dynamic rules.

3.3.2 Congruences

A feature of System S (but not unique to it) is the combination of term traversals and rewriting into one compact construct, called congruences. Consider the following constructor:

$$C : S1 \times \ldots \times Sn \rightarrow S$$

A congruence for this constructor is defined as the following rewrite rule with higher-order parameters $s_1, \ldots, s_n$:

$$c(s1, \ldots, s_n) : c(x_1, \ldots, x_n) \rightarrow c(y_1, \ldots, y_n) \text{ where } \langle s_1 \rangle x_1 \Rightarrow y1; \ldots; \langle s_n \rangle x_n \Rightarrow y_n$$

Given the above definition of a congruence and the previous definition of a rewrite rule, the expression

```
Plus(s0, s1)
```

syntactically expands to the following:

```
?Plus(x0,x1) ; where(<s0> x0 => x0' ; <s1> x1 => x1') ; !Plus(x0',x1')
```

While congruences are syntactically succinct, they mix data traversal strategies and term rewriting logic. This ties rewriting programs to very specific signatures and impairs reuse across subject languages.

3.3.3 Scoped, Dynamic Rules

Stratego supports the notion of dynamic rewrite rules that may be introduced and removed dynamically at runtime. These rules are used to capture and propagate context through the rewriting strategies. Figure 3.2 gives a brief summary of the dynamic rule basics.

The expression `rules(`*R*`: t -> r)` creates a new rule in the rule set for *R*. The scope operator `{| R : s |}` introduces a new scope for the rule set *R* around the strategy *s*. Dynamic rule scopes are dynamic – they follow the flow of the program. Variable scopes, on the other hand, are static – they follow the grammatical structure of the program text. Changes (additions, removals) to the rule set *R* done by the strategy *s* are undone after *s* finishes (both in case of failure and success of *s*). Sometimes, multiple rules in a rule set *R* may match. For example, the rule extension `rules(`*R* `:+ t -> r)` may be used several times with overlapping left hand sides. To get the results of all matching rules in *R*, one may use `bagof-`*R*. The additional operations relating to dynamic rewrite rules will be explained in the context of constant propagation, in Chapter 5.

The following example illustrates an application of dynamic rules to the problem of propagating variable constants. This example will be expanded upon in later chapters. The rule `PropConstAssign` must be applied to terms representing variable assignments in the subject language. If the right hand side of the assignment is a constant, the dynamic rule `PropConst` is added. This dynamic rule maps a given subject language variable to its known constant.

```
PropConstAssign:
Assign(Var(x), e) → Assign(Var(x), e')
where
   prop-const> e ⇒ e'
 ; if <is-value> e' then rules( PropConst : Var(x) → e' )
                 else rules( PropConst :- Var(x) ) end
```

If the constant is not known, i.e. the term `e` is not a value, any previous mappings for this subject language variable is removed.

Concrete Syntax Patterns

Concrete syntax patterns supplement term patterns and may sometimes result in more succinct transformation programs. Syntax patterns are by convention enclosed in "semantic brackets" (`|[ ]|`). They will be expanded in-place by the Stratego compiler to their equivalent AST term patterns.

```
?|[ e0 := e1 + e2 ]| ≡ ?Assign(e0, Plus(e0), Plus(e2))
```

The grammar used to parse the concrete syntax must be specified to the compiler. The grammar is defined using a parser from the XT collection of transformation components described below.

3.3.4 Overlays

Overlays may be thought of as "term macros" and are used to abstract pattern matching over terms. Consider the following overlay declaration:

```
PlusOne(x) = Plus(x, Int("1"))
```

When compiling a program where this overlay is defined, the Stratego compiler will substitute every occurrence of the term `PlusOne(`$\mathcal{X}$`)` with the term `Plus(`$\mathcal{X}$`,Int("1"))`, for example:

$$\texttt{?PlusOne(Int("42"))} \xrightarrow{\textit{overlay expansion}} \texttt{?Plus(Int("42"),Int("1"))}$$

The $\mathcal{X}$ in this case is *not* a Stratego variable. Overlay substitution may be considered a "meta-rewriting" pre-processor step where all constant terms and patterns in a given Stratego program are expanded. After this pre-processing is finished, "normal" compilation resumes.

3.3.5 Modules

Stratego programs are organised into modules. Each module corresponds to a file, and is divided into typed sections. A module may import any number of other modules. A module import is (almost) equivalent to textual inclusion of the imported module[1]. Circular dependencies are allowed. Each section type, e.g. `strategies`, `overlays` and `rules`, specifies which declarations are allowed within that section. One exception exists: both strategies and rules may be declared freely within both `rules` and `strategies` sections.

3.3.6 Stratego/XT

A short note on the name "Stratego/XT" is necessary. The Stratego language was designed to support the development of transformation components at a high level of abstraction. It is distributed together with XT, a collection of flexible, reusable transformation components and declarative languages for deriving new components. Complete software transformation systems are composed from these components. The composition of Stratego and XT is named Stratego/XT.

The traditional usage pattern of Stratego/XT is illustrated in Figure 3.5. The developer starts by constructing or reusing a syntax definition for the subject language *L*. This definition is used to automatically derive a language infrastructure, such as a parser, pretty printer and a signature declaration for the abstract syntax of *L*. The developer may then write transformations using the derived infrastructure against the language *L*. The robustness and quality of the infrastructure is to a large extent

[1]The module name and the import declarations are removed.

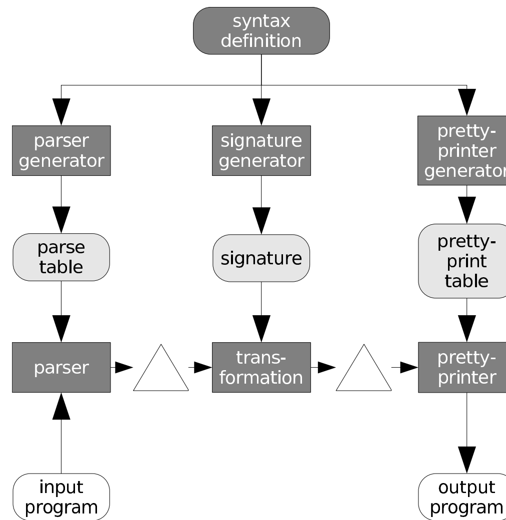


Figure 3.5: Derivation of language infrastructures from syntax definitions (grammars).

determined by the accuracy and quality of the grammar. For many mainstream languages, constructing a solid grammar is highly non-trivial. Consequently, robust and practical mechanisms for easily reusing existing language infrastructures is therefore desirable.

3.4 Summary

This chapter discussed the strategic programming methodology, a programming approach where data traversal patterns are separated from the data processing logic. It described (a subset of) the System *S* core calculus which applies the principles of strategic programming to term rewriting. The result is a clear separation between rewrite rules, which perform data processing, and generic traversals with combinators, which are used to encode data traversals. In the context of program transformations, the separation enables independent reuse of language specific rewrite rules and rule application strategies. This promotes language independence by allowing generic strategies to be reused across language specific rule sets. Basic elements of term rewriting theory were also introduced, together with their relation to universal algebra.

Part III

Abstractions for Language Independence

4

Program Object Model Adapters

Software transformation systems provide powerful analysis and transformation frameworks with concise languages for language processing, but instantiating them for every subject language is an arduous task, often resulting in half-completed front-ends. A lot of mature front-ends with robust parsers and type checkers exist, but few of them expose good APIs to their internal program representations. Expressing language processing problems in general purpose languages without the benefit of transformation libraries is usually tedious. Reusing these front-ends with existing transformation systems is therefore attractive. However, for this reuse to be optimal, the functional logic found in the front-end should be exposed to the transformation system – simple data serialisation of the abstract syntax tree is not enough, as this fails to expose important compiler functionality such as import graphs, symbol tables and the type checker.

This chapter introduces a novel design for a *program object model adapter* that enables program transformation systems to rewrite directly on compiler program object models such as ASTs. The design is reusable across language front-ends and also across program transformation systems based on the term rewriting paradigm. It provides an efficient and serialisation-free interface between the language-general software transformation system and the language-specific front-end infrastructure.

Chapter 10 illustrates the applicability of this design using a prototype framework based on MetaStratego and the Eclipse Compiler for Java. The prototype allows scripts written in Stratego to perform framework and library-specific analyses and transformations.

Much of the content of this chapter has been presented in the paper *"Fusing a Transformation Language with an Open Compiler"* written with Eelco Visser [KV07a].

4.1 Introduction

Software transformation systems are attractive candidates for implementing program analyses and transformations because their high-level domain-specific languages and

their supporting infrastructure allow precise and concise formulations of transformation problems. Unfortunately, transformation systems rarely provide robust and mature parsers and type analysers for a given subject language. Open compilers are also attractive because they provide solid parsers and type analysers, but they are mostly implemented in general-purpose languages. This means that the analyses and transformations must also be implemented in a general-purpose language without the benefit of the transformation features covered in Chapter 2. A consequence of this is that even relatively simple transformation tasks may quickly become time-consuming to implement.

The design introduced in this chapter aims to obtain the best of both worlds by combining the expressive power provided by transformation languages with the maturity and robustness of open compilers using a program object model (POM) adapter. The POM adapter welds together the transformation system runtime and the abstract syntax tree (AST) of the compiler by translating rewriting operations on-the-fly to equivalent sequences of method calls on the AST API. This obviates the need for data serialisation. The technique can be applied to most tree-like APIs and is applicable to many term-based rewriting systems. Using this adapter, transformation languages become compiler scripting languages. Their powerful features for analysis and transformation, such as pattern matching, rewrite rules, tree traversals, and reusable libraries of generic transformation functions, are offered to developers. By instantiating this design with a concrete transformation language and a concrete compiler, as is shown in Chapter 10, a powerful platform for programming domain-specific analyses and transformations is obtained. Depending on the transformation language used, the combined system can be wielded by advanced developers and framework providers because large and interesting classes of domain-specific analyses and transformations may often be expressed by reusing the libraries provided with the transformation system.

The contribution of this chapter is a general technique for fusing domain-specific languages for language processing with open compilers without resorting to data serialisation. When instantiated, this design brings the analysis and transformation capabilities of modern compiler infrastructure into the hands of advanced developers through convenient and feature-rich transformation languages. The technique can help make transformation tools and techniques practical and reusable both by compiler designers and by framework developers since it directly integrates them with stable tools such as the Java compiler. Developers can write interesting classes of analyses and transformations easily and compiler designers can experiment with prototypes of analyses and transformations before committing to a final implementation. In Chapter 10, the system's applicability is validated through a series of examples taken from mature and well-designed applications and frameworks.

The rest of this chapter is organised as follows: In Section 4.2, the design of the POM adapter is explained. Section 4.3 discusses the implementation details of the

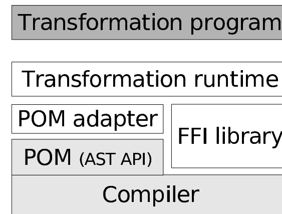


Figure 4.1: Program object model adapter archicture.

design. Section 4.4 discusses related work. Section 4.5 discusses tradeoffs related to the proposed technique. Section 4.6 summarises.

4.2 The Program Object Model Adapter

The program object model adapter fuses together a compiler and a software transformation language. The term *program object model* is used in this dissertation for referring to the object model representing a program inside the compiler. This is typically an AST with symbol tables and other auxiliary data structures such as import graphs. The POM adapter translates the primitive rewriting operations of the rewriting engine to function calls of the POM API.

4.2.1 Architecture Overview

Consider Figure 4.1 which shows the principal components of the design. There are three distinct layers in the figure, coded with different shades of grey. At the bottom, the compiler provides an API for modifying and inspecting its internal program object model. It may also provide additional functionality that should be exposed to the transformation programs such as the ability to manipulate its include paths and output directories. The language used to implement the compiler will be referred to as the *compiler language*. (The source language will, as usual, be the language of the input programs fed into the compiler.) At the top of the figure, transformation programs are written in the *transformation language*. The middle parts of the figure (white boxes) make out the runtime of the transformation language, also referred to as the transformation engine. This part is written in (potentially) another language: the implementation language of the transformation systems, referred to as the *runtime language*. This will in practise always will be the same as the compiler language.

The POM adapter design explained in this chapter is discussed using examples of ASTs implemented in a traditional object-oriented style, but this is not a requirement. Other styles can also be used, however, a large portion of modern language infrastructures are implemented in a object-oriented style. Refer to [Jon] for additional abstract

syntax tree implementation idioms. Irrespective of the implementation language, the essential requirement for the adapter to work is that there are operations on each node for obtaining its children, and, if modification is required, to construct new nodes from existing children.

In the object-oriented style, each node type in the AST, such as `CompilationUnit`, is represented by a concrete class. Children of a node can be retrieved using `get`-methods and replaced using `set`-methods. New nodes are typically constructed using factories such as the method `newCompilationUnit()` of an `AST` factory. Constructing nodes using a `new` operator is also supported by the adapter technique.

The runtime of the transformation system must execute on the same platform, and in the same process, as the compiler. Remote procedure calls, possibly across platforms, is an obvious and a relatively straightforward extension to POM adapter design, but its performance overhead will most likely be prohibitive. A requirement on the transformation runtime is that it has a clear interface to its term representation. Investigations of term rewriting systems suggests that this is the case for a good number of systems, including ASF+SDF [vdBvDH+01], Tom [MRV03] and Stratego [BKVV06]. Provided such a term interface, the task of the POM adapter is to translate operations on the term interface to equivalent operations on the AST API. Any data structure that can provide a suitable interface can be treated as terms and rewritten. This is done by wrapping a POM in the term interface required by the interpreter. The adapter translates term rewriting operations to POM API method calls. These are executed directly on the POM without any intermediate data serialisation.

The transformation runtime would also benefit from a facility for calling foreign functions, i.e. functions implemented outside the transformation language, but this is not strictly necessary. Most transformation systems seem to have such a facility. If a foreign function interface (FFI) facility exists, it may be used to expose native AST API functions as library functions in the transformation language. For example, type analysis, type lookup and import graph queries may be exposed to transformation programs through an FFI.

The design does not place any restrictions on the mode of operation of the transformation system. As was discussed in Chapter 2, several architectures exist for transformation systems, such as pipeline-based and incrementally updating. The POM adapter design does not dictate any one model.

4.2.2 Design Overview

In binding a transformation runtime to a compiler, two interfaces need to be connected: the term interface of the transformation system and the POM interface of the compiler. The algebraic concept of a signature is a very good tool for this task. Based on the signature, adapters can be generated that provide a term interface to the

POM implementation by translating term interface operations to POM operations.

Signatures for AST Classes

Signatures are fundamental for describing formal languages and very appropriate formalisms for describing the abstract syntax of programming languages. The approach taken by the POM adapter design is to extract an exact signature from the AST class hierarchy. This is possible because the AST class hierarchy essentially expresses the signature of the abstract syntax of the language.

Consider the AST class hierarchy in Figure 4.2. The root of this type hierarchy is the abstract class `ASTNode` The abstract classes `Expression` and `Type` derive from it. Concrete types like `ArrayType` and `BooleanType` exist under `Type`. Expressions like `ArrayAccess` exist under `Expression`. This design follows the typical AST implementation idiom found in many object oriented compilers [Jon].

A signature $\Sigma = (S, \Omega)$ can be generated from a class hierarchy using the following algorithm. Assume the root node type of the AST class hierarchy to be r, the function $s(c)$ which maps classes c to sorts and the function $c(c)$ which maps classes c to constructor names.

1. For every abstract class c_a, add a sort $s(c_a)$ to S.
2. If c'_a is a direct subclass of c_a, then $s(c'_a)$ is a direct subsort of $s(c_a)$. Alternatively, injections may be used: Given c'_a, a direct subclass of c_a, add an injection $s(c'_a) \to s(c_a)$ to Ω.
3. For every concrete class c_c, add a constructor with name $c(c_c)$ to Ω. For every parameterless method in c_c returning a subtype c'' of r, add an argument of sort $s(c'')$ (corresponding to a class c'') to the parameter list of $c(c_c)$. The result sort of $c(c_c)$ is the sort of the direct superclass of c_c.

Applying this algorithm to Figure 4.2 gives the following excerpt of a signature with injections at the bottom.

```
signature EclipseJava
sorts
  ASTNode, Expression, Annotation, Type
constructors
  MarkerAnnotation : Name → Annotation
  ArrayAccess : Expression × Expression → Expression
  ArrayCreation : ArrayType × List(Expression) × ArrayInitializer → Expr'n
  CastExpression : Type × Expression → Expression
  PostfixExpression : Operator × Expression → Expression
  PrefixExpression : Operator × Expression → Expression
```

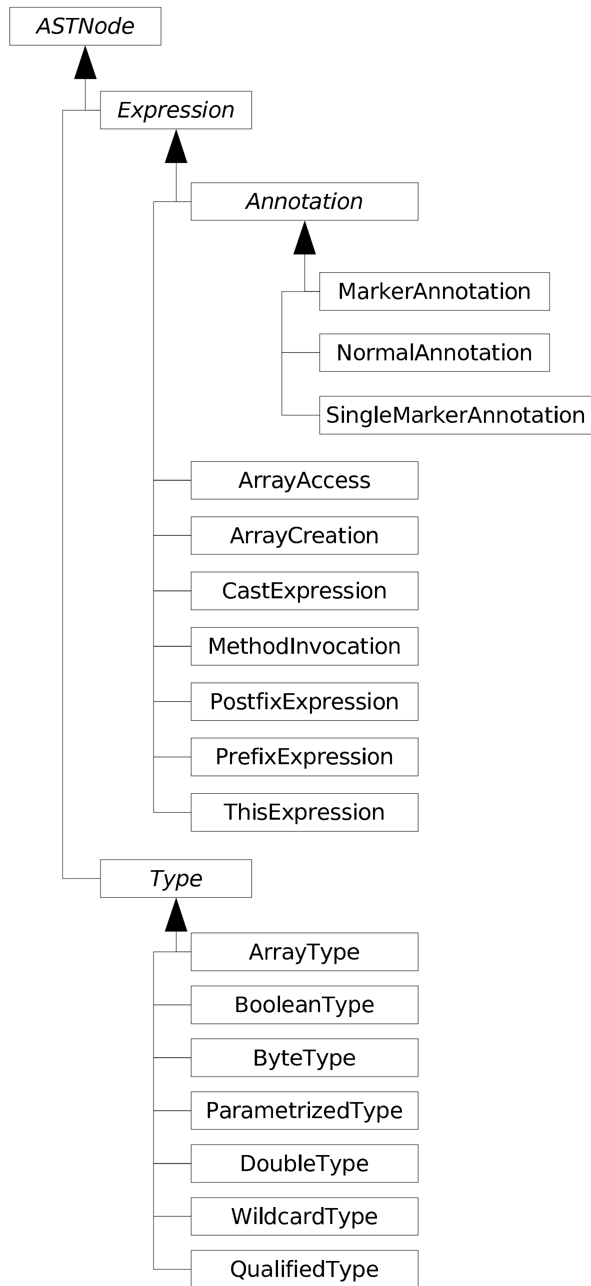


Figure 4.2: Excerpt from the AST type hierarchy of the Eclipse Compiler for Java. Other Java compilers, such as Polyglot and Sun's `javac`, have structurally similar ASTs. Filled arrows indicate inheritance. Names in *italics* indicate abstract classes.

```
ThisExpression : Name → Expression
ArrayType : Type × Int × Type → Type
BooleanType : Type
ParametrizedType : Type × List(Type) → Type
WildcardType : Type → Type
QualifiedType : SimpleName × Type → Type
               : Expression → ASTNode
               : Annotation → Expression
               : Type → ASTNode
```

Consider the class `MarkerAnnotation` in Figure 4.2 and its corresponding constructor definition in the signature above. `MarkerAnnotation` derives from the abstract class `Annotation`, making `Annotation` the result sort of the `MarkerAnnotation` constructor. Further, the class `MarkerAnnotation` has one method returning a subclass of `ASTNode`, `Name getName()` (not shown), and this gives rise to the single constructor argument of sort `Name`.

By processing the source code of the POM, a starting signature is automatically generated. While immediately usable, the initial result is only a proposal. The order of the argument list for each constructor may need manual tuning for consistency among the various constructors. The extracted signature will remain stable as long as the POM implementation changes only rarely. For many mature compilers, the AST designs seem to change rather slowly and mostly in response to changes in the subject language.

Using algebraic signatures, as shown above, is sufficient for capturing precisely and concisely the relationships between the AST node types (as sorts) and their allowed subnodes (in the constructor declarations). In principle, other abstract syntax description languages, such as the Zephyr [WAKS97] abstract grammar language, may be used to represent the abstract syntax. Grammatical formalisms may offer additional expressiveness, but for the scheme illustrated above, this is not necessary. For some readers, grammars may present a more familiar notation than signatures, however.

Term Hierarchy

Terms are recursively built from constructor applications, but as Chapter 2 showed, term rewriting systems frequently have additional primitive term types used for expressing transformation algorithms such as integers and real numbers, lists and tuples, and strings. The terms are available as primitive types in the transformation language, but the machinery behind the terms is written in the runtime language. That is, the term library is implemented in the runtime language.

Below is given a signature for term manipulation. It is a generalisation of the ATerm interface which is used by ASF+SDF, Tom, Stratego and other term rewriting

systems. The remainder of this chapter will explain how to map operations from this signature to AST method calls. The mapping goes from functions on objects in the runtime language to functions on objects in the compiler language. (The compiler and runtime languages are often the same.)

The term interface is separated into two complementary parts: the inspection interface and the generation interface. The former provides operations for traversing and decomposing terms and is a read-only interface to the underlying POM. The generation interface provides operations for constructing POM objects from the ground up, i.e. from the leaves up. The clear separation into two interfaces is very useful because not all POM implementations allow modification. Some front-ends only support inspection of their POM and, using the inspection interface, these front-ends may be reused for expression analysis, but not for transformations. When implementing a POM adapter, one can therefore decide whether read-only access to the POM is sufficient for the problems at hand, or if a full read/write solution must be instantiated.

Inspection Interface Figure 4.3 shows the type hierarchy of all primitive term types. The operations defined on these types comprise the term inspection interface. It illustrates that numerous subsorts of *Term* may exist. It is not required that the term rewriting engine supports all these subtypes. A minimal, but still useful, set would include *TermAppl*, *TermInt*, *TermList* and *TermString*. This is sufficient for representing ASTs of many, if not most, compilers.

At the root of Figure 4.3 is the sort *Term* which has the following operations defined for it:

```
signature TermInspection
sorts Term Integer TermCtor
ops
  get-primitive-type : Term → Integer
  get-constructor : Term → TermCtor
  get-subterm-count : Term → Integer
  get-subterm : Term × Integer → ITerm
  is-equal : Term × Term → Term
  hash-code : Term → Integer
```

The *get-primitive-type* operation returns an integer enumerating which primitive type a given term is, i.e. whether it is an application term, a string, a list, a tuple, an integer, a real or a constructor. The *get-constructor* returns the constructor for its given term. Constructor sorts are described later. The *subterm-count* and *get-subterm* operations are used to inspect and decompose a term. The *is-equal* and *hash-code* operations are used to compare terms. They are also used when terms are placed in collections such as sets and queues.

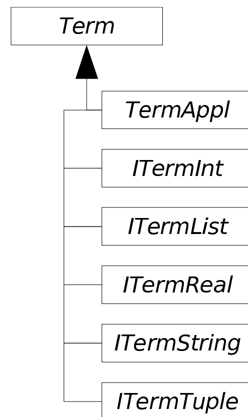


Figure 4.3: Hierarchy of sorts for the low-level term interface.

The various specific term types, such as string, integer and real have a *get-value* operation which returns a string, integer or real number, respectively, using the appropriate type in the runtime language. List term types have operations for obtaining the head and tail of the list, and a predicate signalling if a list is empty.

Objects of the sort *TermCtor* describe constructors of a signature. An *TermCtor* object has a name and an arity. The arity can be a list of sorts, or a number. Since many term rewriting systems are single-sorted, a number (of argument sorts) suffices to describe the arity of a constructor. This "one-typedness" presents some problems when rewriting on fully typed structures such as ASTs implemented in strongly typed languages. It necessitates some form of translation between the type system of the compiler language and that of the transformation language. This topic will be returned to later.

Generation Interface A separate generation interface exists which complements the inspection interface described previously. It consists of the following operations:

```
signature TermGeneration
sorts
  String Int List(s)
  { TermList TermAppl TermInt TermString } < Term // subsorts of Term
ops
  make-appl : TermCtor × List(Term) → TermAppl
  make-int : Integer → TermInt
  make-list : List(Term) → TermList
  make-string : String → TermString
  make-real : Real → TermReal
```

```
make-tuple : List(Term) ⟶ TermTuple
```

The *make-appl* operation is used to instantiate application terms, e.g. Plus(t_0,t_1), from a term constructor object (of sort TermCTor) and a list of terms, i.e. [t_0,t_1]. The operations *make-int*, *make-real* and *make-string* are used to create terms which will be available as values in the transformation language from integers, reals and strings, respectively, in the runtime language. The parametrised sort *List(s)* is assumed to be a builtin or library type of the runtime language.

For a given POM, some of the term subsorts may be unused. As a result, not all of the generator operations need to be defined. For example, typical ASTs are constructed from named nodes (constructor applications), lists, strings and sometimes integers. In these cases, the operations *make-real* and *make-tuple* are irrelevant.

Translating Operations

Given a formal signature for the POM and the term interfaces described previously, a POM adapter may be generated. Based on the previously extracted signature, code generation templates may be used to instantiate the necessary adapter code.

Adapting Inspection The crux of the inspection interface is the *get-subterm* method. In an object-oriented setting, this method will dynamically dispatch on its first argument. Assume a class AdaptedCompilationUnit with a field actualCompilationUnit of type CompilationUnit (from the AST implementation). The *get-subterm* method amounts to a switch:

```
meth get-subterm(this : AdaptedCompilationUnit, i : integer) =
 switch i:
  case 0:  adapt(this.actualCompilationUnit.get-package())
  case 1:  adapt(this.actualCompilationUnit.get-imports())
  case 2:  adapt(this.actualCompilationUnit.get-types())
  default:  raise ArrayIndexOutOfBounds
```

The *adapt* method is overloaded on ASTNode types. For each subclass C of ASTNode, it instantiates a term adapter object of type Adapted$C$. The type of this new object has *get-subterm* defined on it similar to the one just shown. The remaining methods of the term interface are automatically generated using the code templates; for each Adapted$C$ class, the corresponding constructor C defines the return values of *get-subterm-count*, *get-constructor* and *get-primitive-type*. These functions can be generated automatically from the constructor definitions. The two remaining operations, *is-equal* and *hash-code* are discussed in the next section.

Adapting Generation The *make-appl* is the core of the generation interface. It forwards calls to the relevant factory methods of the AST, or instantiates subclasses

of `ASTNode` itself, say, via `new`, if the POM does not provide a node factory.

Based on the extracted signature, a map is constructed which goes from constructor names (and arity) to constructor methods. When the *make-appl* receives a request to construct `CompilationUnit`, the map is consulted and the request is forwarded to the (generated) method *make-compilation-unit*:

```
fun make-compilation-unit(t : ITermConstructor, kids : List(ITerm)) =
  if (is-package(kids[0])
     and is-import-list(kids[1])
     and is-type-list(kids[2]))
    astFactory.newCompilationUnit(as-package(kids[0]),
                                 as-import-list(kids[1]),
                                 as-type-list(kids[2]))
  else
    raise InvalidArguments
```

The responsibility of *make-compilation-unit* is to ensure that the term arguments are type correct before invoking the relevant factory method (or new expression) in the POM interface.

With this generation scheme in place, practically all of the adapter is boilerplate code that can be automatically generated based on two artifacts: the signature declaration and the compiler-specific code templates. Only the order of signature sorts must be verified and potentially fixed up by hand.

4.3 Implementation

The POM adapter design has been instantiated for the MetaStratego runtime (Stratego/J). This section describes the details of this implementation and how it fuses MetaStratego with the Eclipse Compiler for Java (ECJ).

4.3.1 Term Interface

The term interface described in the previous section has been implemented rather straightforwardly in Java. A basic, extensible implementation of the various interfaces is provided by Stratego/J. Its classes and their corresponding sorts (abstract classes) are shown in Figure 4.4. This implementation provides *basic* terms: the `Basic`-terms in the figure. By deriving from the basic implementation, POM adapters may supplement the primitive term types provided by a POM with the full range of primitive term types supported by Stratego. This allows reusing generic analysis transformation algorithms which assume the presence of certain term types, such as tuples or reals, even though the POM itself does not provide one. This is made possible because the basic terms can be mixed with the adapted POM terms. More on this later.

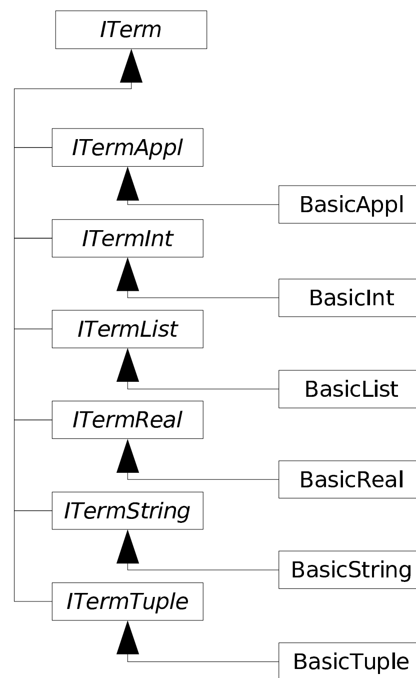


Figure 4.4: The Stratego/J runtime provides a default implementation for the sort hierarchy in Figure 4.3.

Inspection Interface The Java interface of the ITerm type is given below:

```
public int getPrimitiveTermType();
public ITermConstructor getConstructor();
public int getSubtermCount();
public ITerm getSubterm(int index);
public boolean isEqual(ITerm rhs);
public int hashCode();
```

The basic term implementation always forwards the equals() method to isEquals(). Because of this, terms may be freely used in Java collection classes. As an optimisation, the method public ITerm[] getAllSubterms() was added to the ITerm interface. In some POM implementations, children are kept in linked lists. If getSubterm() is used to traverse these lists, the traversal time will be quadratic in the number of children. This presents a significant slowdown. The problem occurs whenever the MetaStratego interpreter needs to traverse all children of a term, for example when it evaluates a one or an all. By extracting all children at the beginning of the traversal, this extra cost is avoided.

The following is another excerpt of the signature extracted from the AST class hierarchy of the Eclipse Java compiler.

```
signature EclipseJava
sorts Annotation Javadoc Name
constructors
  ...
  PackageDeclaration : Javadoc × List(Annotation) × Name → ASTNode
  ...
```

The following shows the final code for the corresponding PackageDeclaration adapter:

```
class AdaptedPackageDeclaration implements AdaptedECJAppl {
  private final PackageDeclaration adaptee;
  private static final IStrategoConstructor CTOR =
        new ASTCtor("PackageDeclaration", 3);

  protected WrappedPackageDeclaration(PackageDeclaration adaptee) {
    super(CTOR);
    this.adaptee = adaptee;
  }

  public ITerm getSubterm(int index) {
    switch(index) {
    case 0:  return ECJFactory.adapt(adaptee.getPackage());
    case 1:  return ECJFactory.adapt(adaptee.imports());
```

```
    case 2:  return ECJFactory.adapt(adaptee.types());
    } throw new ArrayIndexOutOfBoundsException();
  }

  public PackageDeclaration getAdaptee() {
    return adaptee;
  }
}
```

In the current implementation, AST nodes are wrapped lazily, thus wrapping only occurs when needed. When AST nodes are traversed by the rewriting engine, the AST node children are wrapped progressively, as terms are unfolded.

The `isEqual()` method performs a deep equality check, but will not result in a recursive adaptation of child objects. Recall that Stratego allows pattern matching with variables. All the code for handling variable bindings is kept inside the interpreter implementation. This keeps the POM adapter interface minimal.

Generation Interface The POM adapter technique does not require an implementation of the generation interface. If one is not provided, only analysis can be done. For rewriting to be possible, the following factory methods must be available.

```
public interface ITermFactory { ...
  public ITerm makeAppl(ITermConstructor ctor, ITerm[] args);
  public ITerm makeString(String s);
  public ITerm makeInt(int i);
  public ITerm makeList(ITerm[] args);
}
```

Default implementations exist for strings, lists and integers, provided by a term factory for `Basic` terms called `BasicTermFactory`. Only the `makeAppl` method must be supplied by hand. In the prototype, this method forwards constructor requests to the appropriate factory methods of the ECJ AST. When a request for constructing, say, a `PackageDeclaration` node is seen, the request is forwarded to `newPackageDeclaration()` of the ECJ `AST` factory.

```
1 public class ECJFactory implements ITermFactory {
2   ...
3   public ITerm makeAppl(ITermConstructor ctor, ITerm[] args) {
4     switch(constructorMap.get(ctor.getName())) {
5     ...
6     case PACKAGE_DECLARATION:
7       return makePackageDeclaration(args);
8     ...
```

```
9       }
10    }
11
12    private ITermAppl makePackageDeclaration(ITerm[] args) {
13      if((!isJavadoc(kids[0]) && !isNone(kids[0]))
14          || !isAnnotations(kids[1])
15          || !isName(kids[2]))
16        return null;
17
18      PackageDeclaration pd = ast.newPackageDeclaration();
19      if(isNone(kids[0]))
20        pd.setJavadoc(null);
21      else
22        pd.setJavadoc(getJavadoc(kids[0]));
23      pd.annotations().addAll(getAnnotations(kids[1]));
24      pd.setName(getName(kids[2]));
25      return adapt(pd);
26    }
27  }
```

The method `makePackageDeclaration` first ensures that `args` contains the correct number and types of arguments, on lines 13–16. Next, it calls `AST.newPackageDeclaration()` on line 18, uses `set` methods on the resulting `PackageDeclaration` object on lines 19–24 to complete the construction. Line 4 is a performance trick for mapping constructor names to constructor methods. Java does not support function pointers directly. A lookup table (`constructorMap`) is statically initialised when the factory is created. This map allows rapid dispatch to the corresponding construction logic for a given constructor.

Using the Adapter

The Stratego/J interpreter is a Java object of type `Interpreter`. When it is instantiated, one of its constructors allows the user to specify which term representations should be used for its programs (the compiled Stratego scripts) and which factory to use for the data (the terms which it will process). The following code snippet initialises an interpreter instance that will accept compiled scripts as ATerms and can rewrite on the Eclipse compiler ASTs.

```
ITermFactory data = new ECJFactory();
ITermFactory program = new WrappedATermFactory();
Interpreter intp = new Interpreter(data, program);
intp.addOperatorRegistry(new ECJLibrary());
```

The Eclipse Compiler FFI (discussed below) is added to the registry of known foreign functions on the last line.

4.3.2 Design Considerations

Functional Integration – The POM adapter for ECJ provides an FFI library that provides type checking support for Java subject code. This library exposes type checking strategies, such as `type-of`, to the transformation programs written in Stratego. These strategies will call the ECJ type checker, through the FFI mechanisms provided by Stratego/J. Invoking `type-of` on, say, an `InfixExpression` term, will result in a call to `resolveTypeBinding()` on the `InfixExpression` object wrapped by this term. Stratego is a single-sorted rewriting language typed, and only the arity of terms is statically guaranteed. If, say, a `SimpleName` term is passed to `type-of`, the FFI stub for `type-of` will detect this and fail, just like any expression in Stratego can fail.

Imperative and Functional Data Structures – The rewriting engine assumes a functional data structure. In-place updates to existing terms are not allowed. The generation interface is designed so that existing terms will never be modified – there simply are no operations for modifying existing terms. This makes wrapping imperative data structures in such a functional dress relatively straightforward – compilers need not provide one. The only restriction is that AST nodes must not change behind the scenes; the rewriting engine must have exclusive access while rewriting. For in-place rewriting systems, e.g. TXL [Cor04], a slight modification of the `ITerm` interface would be necessary so that subterms of existing terms can be modified in place. Instead of a `makeAppl` method, one would need a `ITerm replaceAppl(ITerm t, int index, ITerm newKid)` method. (Support for this already exists in the BasicTerm implementation, but is not used by Stratego/J.)

In imperative data structures, the state of the system may change during matching because traversal over the data may change the state of the traversed objects – data may for example be loaded from the disk during traversal. State change always occurs when the program object model is wrapped lazily: new POM adapter objects will (in general) be instantiated during matching. Strictly speaking, matching therefore has "side-effects". While the POM adapter may technically speaking force a state change, it will never result in observable differences of terms: all previously bound terms will remain unchanged (so ε remains unchanged). Since it is extremely rare for visitor interfaces to have harmful state changing behaviour, potential side-effects are of little practical concern for building and matching.

Efficiency Considerations – Using a functional data structure provides some appealing properties for term comparison and copying. As described in [BV00], maximal sharing of subterms (i.e. always representing two structurally identical terms by the same object) offers constant-time term copying and structural equality checks as these reduce to pointer copying and comparisons, respectively. This is important for

efficient pattern matching because term equivalence is deep structural equality, not object (identifier) equality. The ECJ AST interface provides deep structural matching, but this is not constant-time.

Quick, deep structural matching can be provided in the POM adapter, but then lazy wrapping must be given up. In this case, hash codes must be computed deeply, because the hash must reflect the entire structure of the term and not just the object identity of the AST node. Ideally, only two objects that are structurally equal should have the same hash code. Once a hash code has been computed, it can be memoized since the subterms will never change.

Efficiency – The memory footprint of the wrapper objects is small. Each object has only two fields. By keeping a (weak) hash table of the AST nodes already wrapped, the overhead is reduced even further. The current implementation takes just over four minutes on a 1.4GHz laptop with 1.5GB of RAM to run a simple bounds checking idiom analysis described in Chapter 10 on the entire Eclipse code base (about 2.7 million lines of code). Complicated transformations are limited by the efficiency of the current Stratego interpreter, not the adapter. Compiling the scripts to Java byte code, instead of the abstract Stratego machine, should significantly improve performance for complicated scripts.

Type System Interaction

The type system of Stratego is significantly more dynamic than that of Java. Many of the usual caveats of integrating a dynamically typed scripting language with a statically typed "host" language apply. However, a few additional considerations specific to the current context warrants further discussion.

Strongly vs Weakly Typed ASTs The ECJ AST is strongly typed and the term rewriting system needs to respect this. Stratego is dynamically typed and would normally allow the term `InfixExpression(1,BooleanLiteral(True),3)` to be constructed, even though the subterms must be `String` and `Expression` as declared previously (making `1`, `3` invalid subterms). `ECJFactory` has two modes for dealing with this. In strict mode, the factory bars invalid `EclipseJava` terms from being built. As a result, the build expression `!InfixExpression(1,BooleanLiteral(True),3)` fails. Terms without any `EclipseJava` terms, such as `(1,2,3)`, can be built freely. These will not be represented as `EclipseJava` terms, but by the default internal term library of the interpreter. Terms without `EclipseJava` constructors are referred to as basic terms.

In lenient mode, mixed terms consisting of basic and `EclipseJava` terms are allowed, such as `InfixExpression(1,BooleanLiteral(True),3)`. The `BooleanLiteral` subterm remains an `EclipseJava` term, but `1` and `3` are basic terms. The root term, `InfixExpression`, becomes a mixed term and is also handled by the basic term library. Since all terms are constructed from their leaves up (`ITermFactory` forces

this), `ECJFactory` can determine inside its `makeAppl()` method when it can build an `EclipseJava` term: if and only if all subterms are `EclipseJava` terms and are compatible with the requested constructor, an `EclipseJava` term is built. Otherwise, a mixed term must be constructed. ECJ FFI functions will fail if they are passed mixed terms. Java programs, such as Eclipse plugins, may embed the Stratego/J interpreter for rewriting ASTs. The embedding Java code will receive an `ITerm` as the result from the interpreter. The actual runtime type of this object can be any subtype of `ITerm`. Therefore, if the embedding Java expects an `AdaptedASTNode`, it must perform a dynamic type check to ensure this before proceeding.

Subject Language Semantics Rewritings can result in structurally valid but semantically invalid ASTs, for example, by removing from a class a method which is called elsewhere. Neither Stratego nor the ECJ AST API checks for this. However, a subsequent type reanalysis will uncover the problem. If the type analysis functions are used as transformation post-condition checks, one can ensure that a transformation is always type correct.

4.4 Related Work

Language processing is what software transformation systems like Tom [MRV03], TXL [Cor04], ASF+SDF [vdBvDH+01] Stratego [BKVV06] were designed for. Except for Tom, these systems were not designed to work with more than one term representation. Retrofitting the POM adapter into existing implementations should not be too difficult provided that there is a clean interface to the terms. In the cases where the contract of the term interface is similar to that described in Section 4.2, many details of the implementation used for Stratego/J should be reusable. This is the case for the ATerm-based approaches such as ASF+SDF and Tom.

Open compilers such as SUIF [WFW+94], OpenJava [TCIK00], OpenC++ [Chi95] and Polyglot [NCM03] are natural candidates for integration. They have well-defined APIs to many parts of their pipeline, often including the backend. It is not necessary for the compiler to be designed as an open platform, however. As long as the AST API is accessible, a POM adapter can be generated for it. If one accepts greybox reuse, this is possible for most compilers.

A key strength of Stratego is generic traversals (built with `one`, `some` and `all`) that cleanly separate the code for tree navigation from the actual operations (such as rewrite rules) performed on each node. The JJTraveler visitor combinator framework is a Java library described by van Deursen and Visser [vV02] that also provides generic traversals. Generic traversals and visitor combinators go far beyond traditional object-oriented visitors. The core term interface required by both approaches is very similar. Comparing the `Visitable` interface of JJTraveler, the ATerm inter-

face found in ASF+SDF and the Stratego C runtime, suggests that the POM adapter should be reusable for all of these systems, implementation language issues notwithstanding (C for ASF+SDF, and Java for JJTraveler and the MetaStratego runtime).

A related approach to rewriting on class hierarchies is provided by Tom [MRV03]. Tom is a language extension for Java. It provides features for rewriting and matching on existing class hierarchies. This is done by using a declaration language called Gom to describe existing classes as abstract data types. Using these descriptions, a preprocessor will expand matching operations in the Tom language into the appropriate method calls according to the Gom declaration. Recent versions of Tom also support generic traversals in the style of JJTraveler, but its library of analyses is still rather small. Gom and the POM adapter technique are both based on the idea of obtaining an abstract declaration of specific class hierarchy and adapting a term rewriting program to operate on the class hierarchy. The approach described in this chapter can extract these descriptions automatically. The POM adapters enable the plugging into any program model at runtime – the binding between a given transformation runtime and a given program object model may be deferred until it is needed. The Tom program is specialised for a given class hierarchy at compile-time. The POM approach makes very few assumptions about the rewriting language; the term interface provided by the POM adapter can form the basis for most rewriting languages, including Tom.

4.5 Discussion

Recent research has provided pluggable type systems, style checkers and static analysis with scripting support. This indicates that there is demand for high-level languages for expressing both analyses and transformations. The languages should be directly usable by software developers. The experiences gained in the field of program transformation, and that have gone into the language design for software transformation systems, are directly applicable for problems of this kind. The tradeoff with using a domain-specific language for expressing transformations and analysis is that of any high-level domain-specific language: the same language features that make the language powerful and domain-specific also make it more difficult to learn. This can be offset in part by good documentation and a sizable corpus of similar code to learn from. In conjunction with the Spoofax development environment, discussed in Chapter 9, the fusion of Stratego and ECJ described in this chapter becomes easily accessible to developers. This will become more apparent through the case studies presented in Part V of this dissertation.

The POM adapter implementation discussed in this chapter, and which is the basis for Chapter 10, was generated using a custom Stratego program and a collection of hand-written Java templates. Careful inspection of the AST implementations

of the Sun Java Compiler, the prototype Fortress compiler, the extensible Java compiler Polyglot [NCM03] and the JastAdd [HM03] compiler compiler, suggests that the POM adapter technique is applicable to a wide number of compilers written in the object-oriented style. Additionally, implementation platform not-withstanding, investigation of the GNU Compiler Collection (GCC) tree representation API indicates that this technique should also be applicable to GCC. Furthermore, it seems feasible to write a more general POM adapter generator based on the current prototype program which developers with basic knowledge of Stratego should be able to adapt this generator to process new AST hierarchies – at least those implemented in Java.

4.6 Summary

This chapter introduced a novel design of a program object model adapter and demonstrated how this design can fuse rewriting language systems with existing compilers and front-ends. This fusion enables language independence through large-scale reuse: entire transformation systems may be plugged onto existing language infrastructures, such as compiler front-ends. The stability and robustness of mainstream front-ends is thereby immediately available to transformation programmer who may express their analysis and transformation problems using high-level transformation languages which support precise and concise formulations.

This chapter demonstrated that even a relatively small degree of extensibility on the part of the compiler is sufficient for plugging in a rewriting system. It motivated that the POM adapter can be reused for other, tree-like data structures and that its design is also applicable to other rewriting engines. In Chapter 10, the applicability of the design will be demonstrated through a series of analysis and transformation problems taken from mature and well-designed frameworks.

– I must say, cracking is much like acupuncture. It's about finding the right spots to insert some NOPs.

– Håvard Sørbø.

5 Modularising Cross-Cutting Transformation Concerns

Properties such as logging, persistence, debugging, tracing, distribution, performance monitoring and exception handling occur in most programming paradigms and are normally very difficult or even impossible to modularise with traditional modularisation mechanisms because they are cross-cutting. Recently, aspect-oriented programming has enjoyed recognition as a practical solution for separating these concerns.

This chapter describes an extension to the Stratego term rewriting language for capturing such properties. It demonstrates how this aspect extension offers a concise, practical and adaptable solution for dealing with unanticipated algorithm extension for forward data-flow propagation and dynamic type checking of terms. The chapter describes and discusses some of the challenges faced when designing and implementing an aspect extension for and in a rule-based term rewriting system.

The aspect language described in this chapter was first presented in the paper *"Combining Aspect-Oriented and Strategic Programming"* written together with Eelco Visser [KV05].

5.1 Introduction

Good modularisation is a key issue in design, development and maintenance of software. Software should be structured close to how one wants to think about it [Par72] by cleanly decomposing the properties of the problem domain into basic function units, or *components*. These can be mapped directly to language constructs such as data types and functions. Not all properties of a problem decompose easily into components. Some turn out to be non-functional and these frequently cross-cut the module structure. Such properties are called *aspects*. The goal of aspect-oriented software development [KLM+97] is the modularisation of cross-cutting concerns. By making aspects part of the programming language, one is left with greater flexibility in modularising software. The cross-cutting properties need no longer be scattered across the components. Using aspects, they may now be declared entirely in separate units, one

for each property. Examples of general aspects include security, logging, persistence, debugging, tracing, distribution, performance monitoring, exception handling, origin tracking and traceability. All these occur in the context of rule-based programming in addition to some which are domain-specific such as rewriting with layout. Existing literature predominantly discusses aspect-based solutions to these problems for object-oriented languages and the documentation of paradigm-specific issues and deployed solutions for the rule-based languages is scarce.

This chapter describes the design and use of aspects in the context of rule-based programming. It introduces the AspectStratego language which enables modular declaration of many separate cross-cutting concerns encountered in rule-based transformation languages. A discussion of the joinpoint model underlying AspectStratego is provided. In addition, the practical usefulness of the extension is demonstrated by three small case studies motivated by the problem of constant propagation. The contributions of this chapter include:

1. The description of a novel aspect language extension implemented for and in a rule-based programming language.

2. An example of its suitability for adding flexible dynamic type checking of terms in a precise and concise way.

3. A demonstration of its application to unanticipated algorithm extension by showing how aspects can help in generalising a constant propagation strategy to a generic and adaptable forward propagation scheme using principles of invasive software composition [Ass03].

This chapter is organised as follows. The next section describes the running example of this chapter: a simple constant propagator. Section 5.3 introduces an extension to Stratego which allows separate declaration of cross-cutting concerns and shows how this extension facilitates declarative code composition. Section 5.4 discusses three cases where the aspect extension is found to be highly useful: logging, type checking of terms and algorithm adaptation. Section 5.6 discusses previous, related and future work. Section 5.7 summarises.

5.2 Constant Propagation

The code in Figure 5.1 shows an excerpt of a strategy for propagating constants applicable to an imperative language with assignment, `While` and `If` constructs. The example in Figure 5.2 illustrates the application of the constant propagator to a short program.

```
1  module prop-const
2  signature
3   constructors
4    Var : Id → Var
5          : Var → Exp
6    Int : String → Exp
7    Plus : Exp × Exp → Exp
8    If : Exp × Exp × Exp → Exp
9    While : Exp × Exp → Exp
10   Assign : Var × Exp → Exp
11 rules
12   EvalBinOp : Plus(Int(i), Int(j)) → Int(k)
13             where <addS>(i,j) ⇒ k
14   EvalIf : If(Int("0"), e1, e2) → e2
15   EvalIf : If(Int(v), e1, e2) → e1 where <gtS> (v, "0")
16 strategies
17   prop-const =
18     PropConst <+ prop-const-assign <+ prop-const-if
19     <+ prop-const-while <+ (all(prop-const) ; try(EvalBinOp))
20   prop-const-assign =
21     Assign(?Var(x), prop-const ⇒ e)
22     ; if <is-value> e then rules( PropConst : Var(x) → e )
23                      else rules( PropConst :- Var(x) ) end
24   prop-const-if =
25     If(prop-const, id, id)
26     ; (EvalIf ; prop-const <+
27        (If(id, prop-const, id) /PropConst\ If(id, id, prop-const)))
28   prop-const-while =
29     ?While(e1, e2)
30     ; (While(prop-const, id)
31        ; EvalWhile
32         <+ (/PropConst\* While(prop-const, prop-const)))
```

Figure 5.1: An excerpt of a Stratego program defining an intraprocedural conditional constant propagation transformation strategy for a small, imperative language.

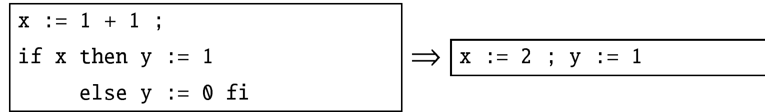


Figure 5.2: Example showing the constant propagation strategy.

The principle of the constant propagation algorithm is straightforward: A traversal through the abstract syntax tree (AST) of a program is done. Whenever an assignment of a constant value to a variable is encountered, this is recorded using a dynamic rewrite rule set named `PropConst`. If the variable is subsequently assigned a non-constant value, the corresponding rule is deleted. This is done in `prop-const-assign` defined on lines 19–22. The process results in rule set mapping variable names to their constant values.

The dynamic rule set is subsequently used to replace every variable with its value where this is known. This replacement opens up for the elementary evaluation rules `EvalBinOp` and `EvalIf` defined on lines 12–14. These rules simplify some expressions involving `Plus` and `If`, respectively.

The `prop-const` strategy on lines 16–18 is the top level driving strategy which takes care of recursively applying the constant propagation throughout a term. It works by calling the rule `PropConst` to replaces any variable term for which the value is known to be constant. If the `PropConst` rule fails, the current term is not a variable with a known constant. In that case, the strategy `prop-const-assign` is attempted. If the `prop-const-assign` is applied to an `Assign` term and the right hand side of the assignment evaluates to a constant, a new `PropConst` rule is generated. If the right hand side of the assignment is not a constant, any previously defined `PropConst` rule for this variable is removed since its value is no longer known. Should the strategy `prop-const-assign` fail, two other strategies are tried in order, namely `prop-const-if` and `prop-const-while`. These are described below. If all strategies fail, `prop-const` falls back to applying itself recursively to all subterms of the current term and finally try to apply the `EvalBinOp` rule (lines 12–13) on the result. This ensures that all language constructs, such as `Plus`, are traversed.

The `prop-const-if` strategy on lines 23–26 matches an `If` construct using the congruence operator while at the same time applying `prop-const` to the condition expression. If the congruence succeeds, the `prop-const-if` strategy proceeds by either (1) simplifying the `If` using the `EvalIf` rule and then recursively continuing the `prop-const` algorithm on the result, or (2) applying `prop-const` recursively to the then-branch and else-branch in turn, keeping only `PropConst` rules which are valid after both branches, i.e. those rules that are defined and equal in both branches.

Recall that the dynamic rule intersection operator `s1 /PropConst\ s2` used on line 26 applies both strategies `s1` then `s2` to the current term in sequence while distribut-

ing (clones of) the same rule set for the dynamic rule `PropConst` to both strategies. Afterwards, only those rules which are equal in both branches are kept. A similar explanation applies to `prop-const-while`, defined on lines 27–31, where the fixpoint operator `/PropConst* s` is used. This operator applies `s` repeatedly until a stable rule set is obtained. Each iteration will apply `s` to the *original* term and the result of the final iteration is kept as the new term.

Generalisation and Adaptation

As written, the algorithm already has some extension points the user of the constant propagator may plug into without modifying the algorithm itself. For example, adding another evaluation rule for `EvalIf` that deals with non-zero constants is trivially possible. There are also other extensions and adaptations users may want to apply to this algorithm which are impossible to do without reimplementing the algorithm from scratch. Section 5.4.1 demonstrates that pervasive logging is one such example. Using aspects, it is possible to extend the code with logging capabilities to record all rule invocations. Section 5.4.2 shows another problematic case where pervasive (dynamic) type checking of terms to ensure the result is a correct term is desired. This is also handled easily with aspects. Finally, Section 5.4.3 shows how the algorithm can be refactored into a more generalised schema for forward propagating data-flow transformations. All extensions and adaptations are performed with the help of the aspect extension to the Stratego language described next.

5.3 AspectStratego

AspectStratego is an extension to the Stratego language which addresses the problem of declaring cross-cutting concerns in a modular way. The language extension bears some resemblance to the AspectJ language [KHH+01].

The reader is not assumed be familiar with AspectJ, but for readers familiar with AspectJ, some differences and similarities are remarked: Much of the terminology in this chapter is inherited from AspectJ. The joinpoint model of Aspect Stratego is somewhat similar to that of AspectJ, but has been adapted to fit better within the paradigm of rule-based rewriting systems. Both AspectJ and AspectStratego provide the programmer with expressions called pointcuts. In AspectStratego these are boolean predicates on the program structure unlike the set theoretic approach taken in AspectJ. Pointcuts are used to pick out well-defined points in the program execution, called joinpoints, and are available in advice to pinpoint places to insert code before, after or around. The inserted code is declared as part of the advice. Advice are in turn gathered in named entities called aspects. The act of composing a program with its aspects is called weaving.

```
1 module prop-const-logger
2 imports logging prop-const
3 aspects
4  pointcut call = strategies(prop-const)
5  aspect prop-const-logger =
6   before : call = log(|Debug, "Invoking constant propagator")
```

Figure 5.3: Aspect extending the constant propagation module with logging. The `log` is from `logging` module, part of the Stratego library.

Figure 5.3 shows how one may use an aspect to extend the constant propagator with trivial logging. It declares a pointcut, `call`, on line 4 that identifies all strategies named `prop-const`. The aspect on line 5–6 declares that before every joinpoint identified by `call`, the code fragment `log(...)` shall be inserted.

Section 5.4 will define the give a more advanced example of logging. Next, the new terminology and language features introduced in this example will be defined.

5.3.1 Joinpoints

A *joinpoint* is a well-defined point in the program execution through which the control flow passes twice: once on the way into the sub-computation identified by the joinpoint and once on the way out. The purpose of the aspect language is allowing the programmer to precisely and succinctly identify and manipulate joinpoints.

5.3.2 Pointcuts

A *pointcut* is a boolean expression over a fixed set of predicates, defined in Table 5.1, and the operators `;` (*and*), `+` (*or*) and `not`. Pointcuts are used to specify a set of joinpoints. There are two kinds of predicates in a pointcut: joinpoint predicates and joinpoint context predicates [1]. A *joinpoint predicate* is a pattern on the Stratego program structure used to pick out a set of joinpoints. A *joinpoint context predicate* is a predicate on the runtime environment which can be used in a pointcut to restrict the set of joinpoints matched by a joinpoint predicate.

Table 5.1 lists the supported joinpoint and joinpoint context predicates. A *pointcut declaration* is a named and optionally parametrised pointcut intended to allow easy sharing of identical pointcuts between advice. The parameters are used to expose details about the pointcut to the advice. The declaration `pointcut call = strategies(prop-const)` from Figure 5.3 shows a parameterless pointcut named `call`

[1] This terminology and implementation differs from the AspectJ language which provides *primitive pointcut designators* instead, see [KHH⁺01].

Joinpoint	*Matches*
`calls(`*name-expr* $\Rightarrow$ *n*`)`	strategy or rule invocations
`strategies(`*name-expr* $\Rightarrow$ *n*`)`	strategy executions
`rules(`*name-expr* $\Rightarrow$ *n*`)`	rule executions
`matches(`*pattern* $\Rightarrow$ *t*`)`	pattern matches
`builds(`*pattern* $\Rightarrow$ *t*`)`	term constructions
`fails`	explicit invocations of fail
Joinpoint context	*Matches*
`withincode(`*name-expr* $\Rightarrow$ *n*`)`	joinpoints within a strategy or rule
`args(`$n_0, n_1, \ldots, n_n$`)`	joinpoints with given arity
`lhs(`*pattern* $\Rightarrow$ *t*`)`	rule left-hand sides
`rhs(`*pattern* $\Rightarrow$ *t*`)`	rule right-hand sides
Advice	*Action on joinpoint*
`before`	run advice before
`after`	run advice after
`after fail`	run after, iff code in pointcut failed
`after succeed`	run after, iff code in pointcut succeeded
`around`	run before and after
Cloning	*Action on declaration*
`clone` *kind name-expr* $\Rightarrow$ *name-expr*	clone and rename a named declaration

Table 5.1: Synopsis of the AspectStratego joinpoints, joinpoint context predicates and advice variants. The *name-expr* can either be a complete identifier name, such as `EvalBinOp`, a prefix, such as `prop-*`, a suffix, such as `*-assign` or an infix, such as `*-const-*`. The result of a *name-expr* is a string, and may optionally be assigned to a variable using the `=>` *x* syntax. The *kind* is either of the keywords `strategies`, `rules` or `overlays`. The *pattern* is an ordinary Stratego pattern, which may contain both variables and wildcards. When a *name-expr* is used for cloning, the literal parts may be replaced. I.e., `clone *-prop-* as *-myprop-*` will rewrite the middle part of the identifier.

with the joinpoint predicate `strategies` and no joinpoint context predicates. It picks out all definitions of strategies named `prop-const`.

5.3.3 Advice

An *advice* is a body of code associated with a pointcut. There are three main kinds of advice: *before*, *after* and *around*. The different forms of advice specify where the body of advice code should be placed relative to the joinpoint matched by its pointcut. Table 5.1 lists the available advice types for AspectStratego. The declaration on line 6 in Figure 5.3 is an example of a before advice. The strategy `log` is provided by the library, and will be discussed later. This code will be inserted – weaved – into the `prop-const` strategy from Figure 5.1 as follows:

```
prop-const = log(|Debug, "Invoking const....")
  ; (PropConst <+ prop-const-assign <+ prop-const-if
  <+ prop-const-while <+ (all(prop-const) ; try(EvalBinOp)))
```

Composing code by inserting advice like this opens up the possibility for manipulating the current term. Recall that all strategies and strategy expressions in Stratego are applied to the current term unless they are specifically applied to a variable or pattern with the application operator (`<s> x`). Exactly how the strategy or rule invocations inside the advice body changes the current term can be controlled in two ways: the advice body is a strategy expression and may be wrapped (entirely or partially) in a `where` to control how and if the current term is modified. In the above case, `log` only takes term arguments and is designed to leave the current term untouched, making `where` superfluous.

This manipulation of the current term turns out to be very useful in `around` advice where the implementer of the advice has full control over how the pointcut should be executed. The placeholder strategy `proceed` is available for this purpose. By placing the `proceed` within a `try` or as part of a choice (+), it is trivially possible to add failure handling policies. The flexibility of `around` allows the aspect programmer to completely override and replace the implementation of existing strategies and rules by not invoking `proceed` at all. This can even be applied to strategies found in the Stratego standard library.

The usefulness of current term manipulation stems from terms normally being passed via the current term from one strategy to another, not as term arguments. E.g., in the following example, `strat2` will be applied to the current term left behind by `strat1`:

```
  strat1 ; strat2
```

An alternative, more imperative formulation of the same would be:

```
  strat1 ⇒ r ; strat2(|r)
```

This is not within the style of Stratego as it becomes cumbersome to use when one replaces sequential composition (;) with the other strategy combinators, such as left choice (<+). Current term manipulation is thus mostly analogous to manipulating input parameters and return values in AspectJ.

A note about rule and strategy priority is warranted. Aspects may be used to modify an existing rule (or strategy), but there is no mechanism by which aspects can directly change the priority of a rule or an aspect.

5.3.4 Cloning

A very useful feature provided by AspectStratego is the ability to clone existing named definitions, such as rules, strategies or overlays. For example, the declaration below will clone all the strategies starting with the name `prop-*` and, for each, create an identical copy with the `my-prop-*` name prefix (the matching value of `*` will be expanded, of course).

```
clone strategies prop-* ⇒ my-prop-*
```

The flat structure of Stratego, with only one global name space for all definitions, makes cloning straightforward. It is allowed, but generally discouraged, to give the clone a name which conflicts with existing definitions. In Stratego, multiple strategies or rules may have the same name, so cloning with a conflicting name must remain allowed – it is sometimes what the developer intends.

Using the cloning feature, it becomes possible to rewrite the pointcuts to apply to clones strategies, i.e. to `my-prop` instead of `prop`. Cloning enables aspects to instantiate multiple *concurrent* variants of existing library functionality in the same program. This allows existing language-specific transformations to be adapted for new subject language signatures. New signatures may introduce additional language constructs. Support for these constructs may be added to an existing (potentially cloned) transformation using the techniques for unanticipated algorithm adaptation discussed later.

The cloning feature was, to the knowledge of the author, first proposed (for Java) in [BBK+05].

5.3.5 Weaving

The pointcuts are designed to be evaluated entirely at compile-time. All cloning declarations are evaluated and resolved before any pointcuts are matched. Given an advice declaration, the compiler will interpret its pointcut declaration on the Stratego abstract syntax tree (AST) to find the location where the advice body must be weaved. The code in the advice body is then inserted into the AST before, after or around the joinpoint.

The body of the advice has a rudimentary reflective capability, which is also resolved at compile-time. Table 5.1 indicates that the advice body has access to rule and strategy names. The Stratego runtime has no reflective nor code-generating capabilities so these names are mostly useful for logging purposes. Advice body code also has access to patterns from match expressions, and may evaluate these patterns at runtime. This is demonstrated in Section 5.4.2.

5.3.6 Modularisation

All AspectStratego code, including aspects, must reside in modules. This seems sensible, since the goal of aspects is to modularise cross-cutting concerns. An `aspect` or `pointcut` can only be declared within an `aspects` section of a module. This is similar to how for example overlays must reside in the `overlays` section. While `aspects` sections may be interleaved with the other Stratego sections (e.g., `strategies`, `rules`, `signature`), it is encouraged that each aspect is declared in a separate module. First, this helps keep aspects – separate, cross-cutting concerns – truly separate, both in design and implementation. Second, this also allows them to be selectively enabled or disabled using compiler flags without any code modification at all. Modules may be substituted in the build system without source code modification. The mechanisms and language features required for controlling the application of aspects on the module level are still subject to research; it is currently not possible to restrict the application of aspects based on the module of a joinpoint.

AspectStratego keeps the pointcut declarations outside the aspect declarations, to signify that pointcuts may be shared between aspects. In object-oriented renditions of aspects, such as AspectJ, sharing of pointcuts between aspects is captured using inheritance: a subaspect inherits all pointcuts from its superaspect. The usefulness of shared pointcuts are demonstrated in Section 5.4.3.

5.4 Case Studies

This section motivates the use of AspectStratego with three case studies relevant to rule-based programming. The first example is a simple logging aspect which is included to show similarities and differences with the AspectJ language. The second is a dynamic type checker of terms realised entirely as an aspect. It shows how aspects may sometimes be used as an alternative to compiler extensions. The final case is a discussion of how aspects may be useful in expressing variation points when implementing generalised adaptable algorithms.

```
module simple-logger
strategies
 invoked(|s) = ![ "Rule '", s, "' invoked" ]
aspects
 pointcut log-rules(n) = rules(* => n)
 aspect simple-logger =
  before : log-rules(r) = log(|Debug, <invoked(|r)>)
  after fail : log-rules(r) = log(|Debug, <failed(|r)>)
  after succeed: log-rules(r) = log(|Debug, <succeeded(|r)>)
  after : log-rules(r) = log(|Debug, <finished(|r)>)
```

Figure 5.4: A complete logging aspect in AspectStratego. The definitions of `failed`, `succeeded` and `finished` are similar to `invoked`. The direction of information flow through the pointcut declaration arguments is somewhat uncommon: they specify information going *out* of the declaration.

5.4.1 Logging

Logging of program actions is often useful when developing software and is therefore a problem one wants to encode in a concise fashion. The program points one wants to trace frequently follow the program structure, for instance, the entry and exit of functions. In these cases, the established solution is to wrap the function definitions in syntactical or lexical macros which perform simple code composition. The numerous shortcomings of this technique, such as decreased code readability, lack of flexibility, interference with meta-tools (especially for documentation and refactoring) and typographic tedium, are all addressed by aspects. The aspect language also allows pervasive insertion of logging code in locations unanticipated by the original implementer such as inside rule conditions and failures deep inside the Stratego library.

The code in Figure 5.4 shows an aspect, called `simple-logger`, that may be used to insert logging code around all rules in a program by adding it to the `imports` list of the main module. The code transformations induced by the weaving are detailed in Section 5.5.

While the built-in `log` strategy provides the ability to set the logging level at runtime (e.g. only errors, and no warning and debug messages), a program with explicit `log` calls inserted into its strategy and rule definitions will always take a slight performance hit. Stratego, where the coding style encourages many and small rules and strategies, is sensitive to any such overhead even with aggressive inlining. Consequently, it is desirable to have the ability to easily remove most or all `log` calls before final deployment. Aspects make this trivial.

The application of one aspect may open up for further adaptation by another aspect. For example, the strategy `invoked` in Figure 5.4 may be the target for further aspects. Note that these second level — or "meta" — aspects pose a few potential problems with respect to weaving order that have not been resolved in the implementation yet. In the current implementation, aspects are weaved in the order of declaration. Consider the following definition of `ext-invoked`:

```
aspects
 pointcut invoked = calls(invoked)
 aspect ext-invoked =
  before : invoked = ...
```

If this aspect were to be weaved before `simple-logger`, it would have no effect, as `invoked` is not called anywhere at the time `ext-invoked` is weaved. As long as the user is aware of this, and manually linearises the dependency chain between aspects by declaring `ext-invoked` after `simple-logger`, the result will match the intention of the user.

5.4.2 Type Checking

Terms in Stratego are built with constructors from a signature, but the language does not enforce a typing discipline on the terms: it is a single-sorted rewriting language. Given the signature in Figure 5.1, a Stratego program may construct an invalid term, e.g. `!Plus(Int("0"), "0")`. As the normal mode of operation for Stratego is local and piecewise rewriting of terms, possibly from one signature to another, invalid intermediates cannot be forbidden. To debug such problems, it is common to manually insert debug printing, or weave in a logger to generate a program trace for manual inspection. This form of manual verification is highly error-prone.

The Stratego/XT environment comes with format checking tools for this purpose. The tools can be applied to the resulting term of a Stratego program, checking it against a given signature. While all signature violations are caught by these tools, they cannot help in telling where in your program the actual problem is present as the check happens entirely after program execution. It is possible to use aspects to weave the format checker into the rules of our program at precisely the spots where one would like the structural invariants to hold. The `typechecker` aspect in Figure 5.5 makes use of the format checker functionality in Stratego/XT to pervasively weave format checking into all rules of a Stratego program. By modifying the `typecheck-rules` pointcut, the user can control the exact application of the type checker. Its usage is similar to the `simple-logger`: it must be imported, but, in addition, a `typecheck` strategy for the relevant signature must be declared in a `strategies` section:

```
typecheck(|t) = format-check-Imp(|t)
```

```
module typecheck-example
aspects
 pointcut typecheck-rules(n, t) = rules(n) ; rhs(t)
 aspect typechecker =
  around(n, t) : typecheck-rules(n, t) =
  proceed ; (typecheck(|t)
    <+ ( log(|incorrect-term(n) ; fail ))
```

Figure 5.5: An aspect for weaving simple dynamic type of terms into rules.

Given the signature in Figure 5.1, the Stratego/XT format checker tools generate a Stratego module containing a complete format checker for that signature. The top level strategy for this format checker is named `format-check`. It may be applied to a term and checks if it is a valid (sub)term of that signature.

As with logging, introducing the checking aspect provides the user with a quick and concise mechanism to decide which parts (if any) of a program should be type checked. Its usage can be toggled both at compile- and runtime (the latter always incurs a small performance hit as previously discussed).

The aspect Figure 5.5 invokes the `typecheck` strategy. The argument *t* to `typecheck` is the pattern matched by the `typecheck-rules` pointcut, i.e. the pattern is extracted from the right-hand side pattern of a rule. In the case that `t` is a term (no variables), it can in theory be entirely checked at compile time as both the signature and the term are completely known to the compiler. In the case that *t* contains variables, the static parts may be checked at compile time, but the variable part must be evaluated at runtime.

The type checking aspect is only a partial replacement for a built-in type system. It performs no type inferencing and can therefore not eliminate redundant checks. The topic of typed, strategic term rewriting is discussed in [Läm03].

5.4.3 Extending Algorithms

The algorithm in Figure 5.1 is an instance of the more general data-flow problem of forward propagation examples of which are common subexpression elimination, copy propagation, unreachable code elimination and bound variable renaming. The algorithm can be factored into a language-specific skeleton and problem-specific extensions. The language-specific skeleton must account for control-flow constructs and scoping rules specific to a given language. In some cases, it is possible to abstract over subject language differences. Using aspects, additional flexibility is provided and the skeletons may more easily be reused for similar subject languages. Cases for additional language constructs may easily be added to the skeleton using before advice,

and non-applicable cases may be voided using around advice.

A *variation point* is a concrete point in a program where variants of an entity may be inserted. By providing clearly defined variation points, the skeleton is made adaptable to the specific propagation problem at hand.

Expressing Adaptable Algorithms

There are many well-known techniques for expressing adaptable algorithms. When providing an algorithm intended for reuse and adaptation by other programmers (users), the following properties of the technique become important:

- *adaptability*; one would like maximal freedom in which variation points one exposes to the users.
- *reuse*; the users of the algorithm should need to reimplement as little code as possible. This is especially important in the face of maintenance.
- *traceability*; when errors (either in the design or the implementation) are discovered in the algorithm, users should be offered an easy upgrade path. Ideally, the users should only need to replace the library file wherein the algorithm resides. This may not always be feasible, but, at the very least, the users should know which parts of their system may be affected by the error.
- *evolution*; one must be able to change the internals of the algorithm without disturbing the users.

Boilerplates One of the most popular, but least desirable, techniques for adaptation is *boilerplate* adaptation. In this approach, a code template is manually copied then modified to fit the situation at hand. The approach suffers from high maintenance costs due to inherent code duplication. It is especially problematic if the original template is later found to contain grave (security) errors since there is no traceability of where it has been used. On the other hand, it offers a very high degree of flexibility as all variation points may be reached. At its most extreme, boilerplate adaptation allows the applicant to gradually replace the entire algorithm.

Design Patterns Another, popular technique for reuse is the use of *design patterns* [GHJV95]. A design pattern is a piece of reusable engineering knowledge. For every case where a design pattern is applicable, it must be implemented from scratch by the programmer. In the recent years, much research has been into improving reuse of design patterns, either by providing direct language support [Bos98, Hed98] or by placing them in reusable libraries [AC98, HK02].

Hooks and Callbacks *Hooks* and *callbacks* are well-known techniques for exposing variation points through *overridable* stubs the user of a library or algorithm can extend. By calling registration functions, the user may add callbacks and hooks which are called at pre-determined locations in the algorithm or upon particular events in the program. As long as the contract between the algorithm and its callbacks is maintained, the algorithm internals may evolve separately from the adapted hooks and therefore offers good maintenance properties. Its drawbacks include the fact that not all variation points may be expressed as hooks, and that it is difficult to adapt an algorithm with different sets of hooks in multiple contexts within the same program. In Stratego, this can to some degree be solved using scoped dynamic rules. For other paradigms, function pointers, closures and/or objects allow multiple contexts to exist.

Higher-order Parameters In functional languages, it is common to expose variation points through *higher-order parameters*. The paper [OV05] describes an adaptable skeleton for forward propagation using this approach. The technique provides a precise way for exposing variation points which is both easy to use and allows the user to adapt the algorithm on a per-context basis within the same program. One drawback is the issue of "parameter plethora", i.e. the number of parameters users must deal with. In cases where the problem space is large, the algorithm often has many variation points yielding a long parameter list. A common solution to this problem is providing multiple entry points into the algorithm, each with an increasing number of parameters. Another is having parameters with default values where the language supports this.

Limitations

Boilerplates and design patterns are not really desirable given their poor support for code reuse and traceability. While the last two solutions discussed above offer both good reuse and traceability, they suffer from a few additional drawbacks. Over time, experience with the use of an algorithm may expose a need to extend it with further variation points unanticipated by the original implementer. Exposing a new variation point frequently results in a change in the algorithm interface, through the adding new higher-order parameters, hooks or new parameters to existing hooks. Backwards compatibility can normally be handled by writing wrappers mimicking the old interface which forwards to the new. The price is the burden of maintaining multiple versions of the same interface.

Another consideration when extending an algorithm is how to propagate the new variation point through its internals. Suppose in `prop-const` (Figure 5.1), one wanted to add the ability to transform the current term before recursively descending into the children. With a solution based on higher-order parameters, this transform parameter would have to be "threaded" through all `prop-*` strategies as a higher-order parameter,

and thus result in an intrusive rewrite.

A final consideration is who should be able to perform adaptation and extension of existing algorithms. It is normally not possible for the user of the algorithm to extend it outside the exposed variation points even if they can be clearly identified, unless the user has access to the source code, in which case the boilerplate technique may be resorted to.

Dealing with Evolution

This section demonstrates a solution to the extensibility problem for handling *unanticipated* variation points that is complementary to hooks and higher-order parameters. It uses the declarative features of aspects to clearly identify and name the variation points in the algorithm. The code in Figure 5.6 shows how some of the variation points already discussed have been exposed through pointcuts. The algorithm provider may decide to add some trivial points, `fail` in `forward-prop` and `id` in `prop-assign` to allow the pointcuts and advice to be expressed more clearly. These are not strictly necessary. The same joinpoints can be identified and used with only slightly more complicated pointcuts and advice and also by a user of the skeleton without involving the provider nor changing the code.

The `forward-prop` pointcut may be used to insert the transformation code before and after the propagator visits subterms of a given term. The `prop-*` pointcuts may be used similarly for inserting code before and after recursive descent into subterms of their respective language constructs. The `prop-rule` pointcuts are used for declaring which dynamic rule(s) to use for intersections and during traversal. Note that the pointcuts have the same names as the strategies they match inside. This makes it very clear to the user where the advice is applied. Admittedly, this is also a potential source of confusion as the same identifier may refer to either an aspect or a strategy/rule, depending on context. The pointcut namespace is kept separate from the other namespaces in Stratego (one for rules and strategies, another for constructor names) because the namespaces in Stratego are global and one-level. There is no hierarchy of namespaces (c.f. Chapter 2, Section 2.4.1).

By using these variation points exposed through aspects, the code in Figure 5.7 demonstrates how the skeleton may be instantiated with advice to obtain a constant propagator. After weaving, the result is the exact algorithm presented in Figure 5.1. `around` advice is used instead of `after` advice to properly parenthesise the expressions and control operator precedence. Consider the weaving of the `around` advice for `prop-while` pointcut. The pointcut matches the following joinpoint code:

```
/\* While(forward-prop,forward-prop)
```

By using the around advice, this expression is replaced with:

```
(While(forward-prop,id); EvalWhile <+ proceed)
```

```
module forward-prop
strategies
 forward-prop =
   fail <+ prop-assign <+ prop-if <+ prop-while
     <+ all(forward-prop)
 prop-assign =
   Assign(?Var(x), forward-prop => e) ; id
 prop-if =
   If(forward-prop, id, id)
   ; (If(id,forward-prop,id) /\ If(id,id,forward-prop))
 prop-while =
   ?While(e1, e2)
   ; (/\* While(forward-prop, forward-prop)))
aspects
 pointcut prop-rule(r) =
   (calls(dr-fork-and-intersect) ; args(_, _, r))
   + (calls(dr-fix-and-intersect) ; args(_, r))
 pointcut prop-rule = fails ; withincode(forward-prop)
 pointcut forward-prop = calls(all) ; withincode(forward-prop)
 pointcut prop-assign = calls(id) ; withincode(prop-assign)
 pointcut prop-if =
   calls(dr-fork-and-intersect) ; withincode(prop-if)
 pointcut prop-while =
   calls(dr-fix-and-intersect) ; withincode(prop-while)
```

Figure 5.6: Skeleton for forward propagation with variation points exposed as pointcuts. For a real language, the skeleton is often quite large and often difficult to construct. `s1 /Rule\ s2` is syntactic sugar for `dr-fork-and-intersect(s1, s2 | [ "Rule" ])`, and `/Rule*` is sugar for `dr-fix-and-intersect`. In the above code, the `Rule` will be filled in later by aspects, thus the empty fork (`/\`) and fix (`/*`) in `prop-if` and `prop-while`, respectively.

Since `proceed` invokes the original (matched) joinpoint code, the end result is the same code as found in `prop-const-while` in Figure 5.1, modulo the fact that the driving strategy is now named `forward-prop`. Using cloning and renaming, it is possible to derive a practically identical implementation. Additionally, the patterns and traversals may be adapted for additional signatures, thus easily instantiating the forward propagator for a family of subject languages.

Evaluation

The proposed solution is now evaluated based on the criteria set out above.

Adaptability Exposing variation points through pointcuts is more adaptable than higher-order parameters and hooks because it can be done without changing the algorithm itself. As long as the variation point can be picked out using a pointcut, an advice may be used to insert a callback into the algorithm at that point. This is easier with AspectStratego than many other aspect extensions since the data normally is passed through the algorithm as the *current term*. It is possible to use pointcuts to modify the current term before or after any strategy or rule invocation in the algorithm implementation. Aspects can be viewed as a complementary extension mechanism to callbacks/hooks since it may be used to add these. Similarly, the aspect technique is complementary to higher-order parameters. It is also possible to wrap the entry point to the algorithm in a reparametrised strategy.

Different levels of adaptability may be exposed using aspects. These these levels are expressed separately from the algorithm skeleton. By choosing between the available adaptation aspects, the user may select which sets of variation points he intends to deal with. Assuming white-box reuse, the user may add new variation points to the algorithm in this fashion.

Reuse Compared to design patterns and boilerplates, much better reuse is obtained. With a properly designed skeleton, the amount of code needed to adapt the algorithm is proportional to the extra functionality added.

Traceability It is directly evident from the code both which version of the skeleton that has been used and how it has been adapted (using which aspects). Traceability is therefore better than for boilerplates and patterns, and at the same level as parameters and callbacks.

Evolution As time goes by, new callbacks and higher-order parameters may easily be added to the skeleton using aspects. Further, aspects may be used internally to propagate the parameters to all sub parts of the algorithm implementation. Arguably, extra care must be taken to ensure that the semantics of the pointcuts are kept after

```
module forward-prop-usage
imports forward-prop
aspects
 aspect prop-const =
  around : prop-rule(r) = proceed([ "PropConst" ])
  around : prop-rule = PropConst
  around : forward-prop = (proceed ; try(EvalBinOp))
  before : prop-assign-ext =
    ?Assign(Var(x), e)
    ; if <is-value> e
       then rules(PropConst: Var(x) → e)
       else rules(PropConst:- Var(x)) end
  around : prop-if = EvalIf ; forward-prop <+ proceed
  around : prop-while =
    (While(forward-prop,id) ; EvalWhile <+ proceed)
```

Figure 5.7: Instantiation of the forward-prop to make the constant propagator in Figure 5.1, using aspects. Admittedly, the example is somewhat contrived, as these are variation points we normally would anticipate and explicitly parameterize easily.

an algorithm revision since they now are declared separately. This problem is no different from variation points exposed through higher-order parameters or hooks as long as the pointcuts are known to the revising party.

In the case where users have identified and extended variation points through their own pointcuts, the situation is more precarious. This is a known drawback of white box reuse.

A particularly attractive feature of aspects in the context of this dissertation is the way they enable the expression of language independent transformations. General algorithm skeletons may be implemented and adapted invasively when they are instantiated for new subject languages. To some extent, existing language-specific transformations may in some cases also be adapted to other, similar languages.

5.5 Implementation of the Weaver

The aspect weaver for AspectStratego is realised entirely inside the Stratego compiler as one additional stage in the front-end. It operates on the normalised AST where the module structure has been collapsed. All definitions from all included modules are thereby available for weaving. The weaver is implemented as traversals on the AST. The full pipeline for compiling – or weaving – the aspect extension into Stratego is

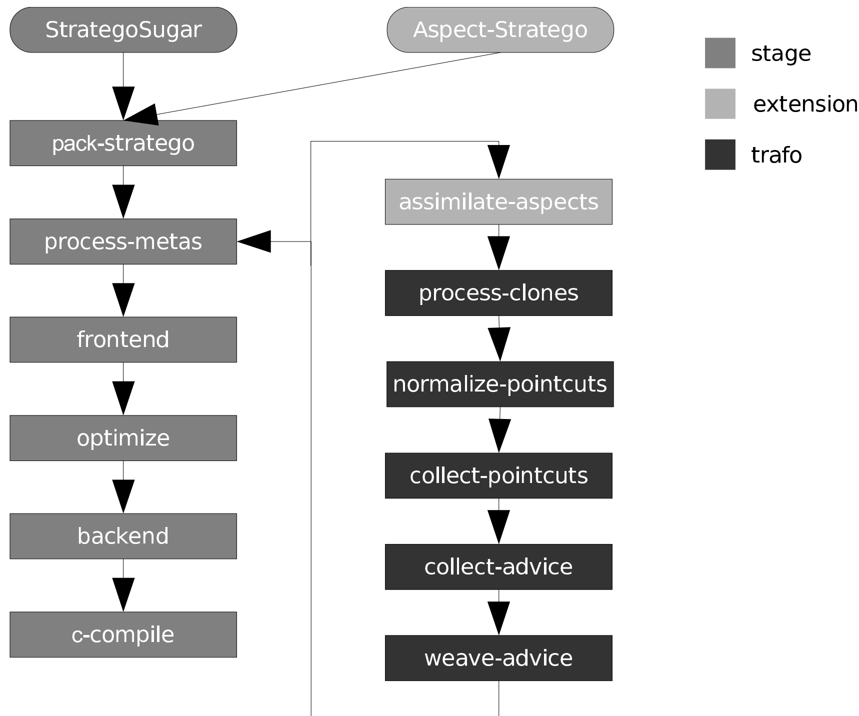


Figure 5.8: Pipeline for assimilating AspectStratego into StrategoCore

shown in Figure 5.8.

Cloning – All clone expressions are collected and the relevant definitions are duplicated and renamed in two-pass top-down traversal called `process-clones`. First, all the clone expressions are collected and then to all matches are found and duplicated.

Pointcut collection and evaluation (`collect-pointcuts` in the figure) is a top down traversal that collects all pointcut declarations. Every pointcut encountered is essentially a simple logical expression. These expressions are decomposed into conjunctive normal forms called *fragments*. A fragment contains one joinpoint and an arbitrary set of joinpoint context predicates all separated by logical `and`. For example, the pointcut `(rules(n) + strategies(n)) ; args(y)` is split into the two fragments `rules(n) ; args(y)` and `strategies(n) ; args(y)`. For each named pointcut, a dynamic rule is generated and used as a map from pointcut names to the fragment set for that pointcut.

Advice collection and evaluation (`collect-advice`) is a top down traversal that collects all advice declarations. When an advice is encountered, its associated pointcut is looked up and one dynamic rule is generated for each fragment of that pointcut. In a generated rule, the left-hand side matches the term in the Stratego AST corresponding to the fragment's joinpoint predicate. For example, `rules` match against the AST term for rule declaration, `RDefT`. The generated rule evaluates all joinpoint

```
EvalBinOp =
  log(|Debug, invoked("EvalBinOp")) ;
  if shadowed-EvalBinOp then
    if log(|Debug, succeeded("EvalBinOp")) then
      try(log(|Debug, finished("EvalBinOp")))
      else
      log(|Debug, finished("EvalBinOp")) ; fail
    end
  else
     // identical to the then-clause,
     // with succeeded replaced by failed
  end
```

Figure 5.9: The declaration of `EvalBinOp` from Figure 5.1 after weaving in the `simple-logger` aspect.

context predicates in its condition. These rules are applied later by the weaver. When a rule succeeds, it provides the weaver with the context information identified by its pointcut fragment and the advice body associated with that pointcut.

Weaving – The actual weaving is a bottom up traversal of the AST (`weave-advice`). It exhaustively attempts to apply all generated advice rules from the previous step. On any term where one or more rules match, their associated advice bodies are collected and applied in place.

5.5.1 A Weaving Example

By weaving the `simple-logger` aspect (Figure 5.4) into the module in Figure 5.2, all executions of `EvalBinOp` and `EvalIf` are logged. Weaving of this aspect on the rule `EvalBinOp` proceeds as follows.

First, the weaver shadows both declarations by adding a new unique prefix to the existing rule name. Then, a wrapper strategy from the template in Figure 5.10[2] is instantiated. It has the name of the original rule (`EvalBinOp`). The final result of this weaving for `EvalBinOp` is shown in Figure 5.9. The wrapper first executes the code from the before advice followed by the shadowed (original) code. If the shadowed code fails, the `after fail` advice is run followed by the `after` advice. The enclosing `try` and `if-then-else` are there to allow `after fail` and `after succeed` advice to change a failure into success or success into failure, respectively. `after` advice may not change failure/success but may replace the current term.

[2]Technically, the actual implementation uses the guarded choice operator. For readability reasons, the `if-then-else` is shown in the examples.

```
before ;
if pointcut-code then
  if after-succeed then try(after) else after ; fail end
  else
  if after-fail then try(after) else after ; fail end
end
```

Figure 5.10: Template for advice weaving. Cursive identifiers are insertion sites for advice code. If a particular advice is not present in a joinpoint, it is replaced by an `id` (`after-fail` is replaced by `fail`).

5.5.2 Aspects as Meta Programs

When evaluating the pointcuts in the aspect compiler, there is a need to do interpretation of the pointcut expressions. This is realised as interpretive dynamic rules in the current implementation. Unfortunately, this leads to a rather rigid and tangled implementation where extending the language with new joinpoint (context) predicates becomes needlessly complex. It is conceptually much more appealing to view each advice as a small meta program to be executed on the AST. This meta program must be constructed at compilation time and can therefore not be a fixed part of the compiler. The current implementation can be seen as a manual specialisation of such a meta program where the dynamic parts are captured by dynamic rules.

Instead of inventing and maintaining a new interpreter for such meta programs, it is desirable to generate a small Stratego program for every meta program. This would be possible in a rewriting language with an open compiler or in a flexible, multi-staged language. The weaver would generate compiler extensions (meta programs), then execute these as part of the compilation process. This is now possible with the MetaStratego infrastructure, but the weaver has not yet been updated to take advantage of these developments.

5.6 Discussion

There are several documented examples of cross-cutting concerns found in the domain of rule-based programming. For example, the problem of origin tracking is documented in [vDKT93] and the problem of rewriting with layout in [vdBV00]. Both papers present interpreter extensions as the solution to their respective problems.

In [KL03], it is argued that both the above cases are instances of the more general problem of propagating term annotations – a separate concern which should be adaptable by the programmer. The solution proposed in [KL03] is to provide the programmer with declarative *progression methods* expressed as logic meta-programs.

It is realised as a research prototype in Prolog. The aspect extension also provides a mechanism for specifying cross-cutting concerns in a declarative and adaptable way, but the style proposed herein is very similar to the popular AspectJ language, although recast for Stratego.

Many other aspect extensions have been documented. The AspectS system for the Squeak dialect of Smalltalk [Hir03] describes a weaver which works entirely at runtime using the reflective features of the Smalltalk runtime environment. The Casear aspect extension for Java [MO03] brings runtime weaving to Java. In [LK97], the authors describe a small object-oriented language Jcore and its extension Cool for expressing coordination of threads. The two are composed using an aspect weaver. AspectC++ [SGSP02] is an aspect extension to the C++ language. An aspect extension for the functional language Caml is described in [HT05]. In [MRB+], the authors document an aspect extension to the GAMMA transformation language for multiset rewriting and demonstrates its use to express timing constraints and distribution of data and processes. AspectStratego attempts to solve many of the same problems as the languages above because these problems are found in many languages. In addition, this chapter also motivates how problems specific to rule-based programming, such as language independence, may have solutions based on aspects.

The implementation of aspect-weavers using rewriting has been documented in [AL00] for the context of graph rewriting and in [GR04] using term rewriting. In both cases, the subject languages were object-oriented. In [Läm99], the authors detail an aspect language for declarative logic programs with formally described semantics, and a weaver based on functional meta-programs. Reflective languages with meta programming facilities such as Maude [CDE+03] are alternative implementation vehicles for aspects. The appeal of aspects is their concise, declarative nature with their clearly defined goal: identify joinpoints for inserting code. This contrasts with the flexibility and complexity offered by general meta programming. The "general-purposeness" of meta programming may in fact often be a hindrance to users. Distilling the power of general meta programming into a concise, declarative aspect language may therefore be worthwhile. While this chapter also describes the implementation of an aspect weaver using a term-rewriting system, the subject language, Stratego, is not a declarative logic nor an object-oriented language. This gives rise to a different set of joinpoints than considered by the above references.

The algorithm extension technique described in Section 5.4.3 is an example of compile-time code composition and is thus somewhat related to techniques such as templates in C++. Unlike C++ templates, the AspectStratego composition language is purely declarative and new variation points can be exposed retroactively without reparameterizing.

5.7 Summary

This chapter presented an aspect extension for the Stratego term-rewriting language, combining the paradigms of aspect-oriented programming and strategic programming. The implementation of this language was discussed. Several examples of is applicability were given, including a flexible dynamic type checker of terms as a practical example of aspects as an alternative to the interpreter extensions in [vDKT93] and [vdBV00]. The chapter also demonstrated how aspects may be helpful in handling unanticipated algorithm extension using the technique of invasive software composition. This form of (potentially retroactive) parametrisation increases the genericity of (existing) implementations, and thereby improves language independence. Aspects may be regarded as a declarative mechanism for adding support for subject language families to transformation libraries and are therefore an attractive language abstraction for language independent transformations.

Part IV

Supportive Abstractions for Transformations

6

An Extensible Transformation Language

This chapter introduces new compiler and runtime infrastructure that turns Stratego into an extensible transformation language. The extensible variant of Stratego is used throughout this dissertation as a platform for experimenting with language extensions and abstractions for language-independent transformations. The extensible compiler reuses most of the Stratego compiler and the Stratego library. It is a strict – or conservative – extension of the Stratego infrastructure. The compiler is complemented by a versatile transformation runtime designed to abstract over different term representations. The runtime supports dynamic loading of transformation components during execution. These components are expressed using a new and light-weight component architecture for self-contained transformation components.

6.1 An Extensible Compiler

The primary design goal of the extensible compiler, called the MetaStratego compiler, is to support the development of language extensions for Stratego. Figure 6.1 illustrates examples of language extensions under development with this compiler. Developers of transformations may add their own extensions (illustrated with the stippled box containing `your-stratego`) as part of a specific transformation project.

6.1.1 Declaring Syntax and Assimilator

The extensible compiler enables programmers to plug in language extensions at compile time. A programmer may do this in either of two ways. Each Stratego module is implemented in a `.str` file, for example `foo.str`. By creating a so-called meta file, it is possible to give instructions to the compiler about how to treat the contents of `foo.str`. The following is an example meta file, `foo.meta`:

```
Meta([
  Syntax("AspectStratego"),
  Assimilator("assimilate-aspects")
```

Figure 6.1: Language extensions for Stratego implemented using MetaStratego. This dissertation discusses GraphStratego and AspectStratego. The reason for the depicted MetaStratego extension is explained in Section 6.1.1.

```
  ])
```

This file must always accompany `foo.str`. It declares to the MetaStratego compiler that the `AspectStratego` syntax must be used to parse the file `foo.str`. The meta file also declares that the AspectStratego language extension is to be translated (assimilated) into plain Stratego using a transformation component called `assimilate-aspects`.

When the file `foo.str` is compiled, the MetaStratego compiler will search the include path for the `AspectStratego` syntax and the `assimilate-aspects` assimilator program. For convenience, it is also possible to specify the language extension inside the module. This is done in a separate meta section:

```
module foo
meta
  syntax = "AspectStratego"
  assimilator = "assimilate-aspects"
imports
  ...
```

The meta section must always appear immediately after the module declaration. A small pre-processing step will read the top of the file and extract the meta information. The choice between meta files and meta sections is a matter of programmer preference.

Extending Stratego with a given language extension results in an *extended* Stratego language. For example, extending Stratego with support for aspects results in AspectStratego. The term *plain Stratego* will be used to refer to Stratego without any language extensions.

A short note on the meta section may be warranted. Meta sections are not part of plain Stratego. Support for meta sections is due to a tiny syntax extension provided by the MetaStratego compiler. It will by default parse all source files using this MetaStratego syntax. Technically speaking, all language extensions are therefore extensions of MetaStratego, not plain Stratego.

6.1.2 Language Extensions

The language extensions presented in this dissertation are all composed of two parts; a *notational component*, which extends the Stratego syntax, and a *transformation component* which provides the semantics of the extension. A language extension may be deployed separately from the compiler. The build system of a given project must declare the relevant paths of all necessary syntax extensions to the MetaStratego compiler.

The notational component of a language extension reuses the syntax extension mechanisms provided in [Vis02, BV04]. By composing grammar modules for the syntax extensions with the base grammar for Stratego[1], an extended Stratego language is obtained. Programs in the extended language are parsed with the parse table generated from this extended grammar. The compiler front-end will produce a corresponding extended AST which contains extension-specific nodes.

The semantics of the extension is expressed as a transformation from the extended AST into the plain Stratego AST. These transformation are called assimilators [BV04] because they assimilate (embed) the extension into plain Stratego. Developers implementing assimilators make use of the standard Stratego library, the Stratego compiler library and the MetaStratego compiler library as shown in Figure 6.2.

6.1.3 Compiler Pipeline

The MetaStratego compiler pipeline is depicted in Figure 6.3. The MetaStratego compiler reuses most of the standard Stratego compiler, but supplements it with a some new functionality and extension points. The standard compiler provides a mechanism for embedding concrete syntax patterns into the transformation program [Vis02]. It essentially provides the compiler user with an option to specify which grammar should be used to parse a given source file. MetaStratego relies on this mechanism for providing the additional syntax offered by the language extensions.

Many assimilators may be formulated so that they only interact with the compiler at one point: in the `process-metas` stage. All the language extensions proposed

[1]Strictly speaking, the grammar for MetaStratego which, except for meta section support, is identical to the Stratego grammar.

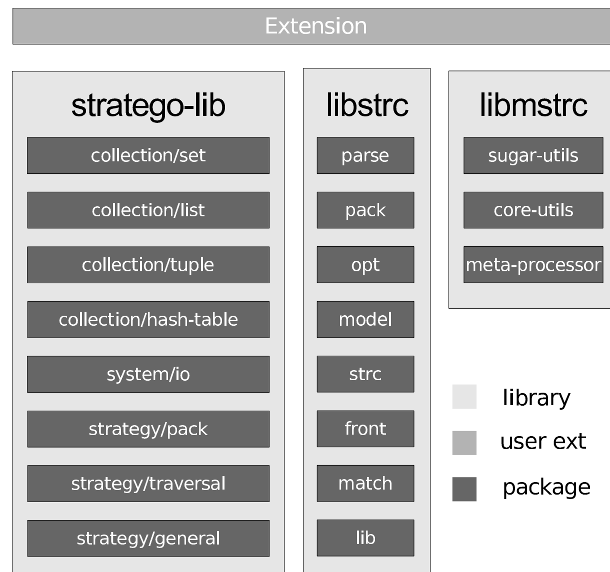


Figure 6.2: Transformation libraries available for implementing language extensions. `stratego-lib` is the Stratego standard library, `libstrc` is the Stratego compiler library and `libmstrc` is the MetaStratego compiler library.

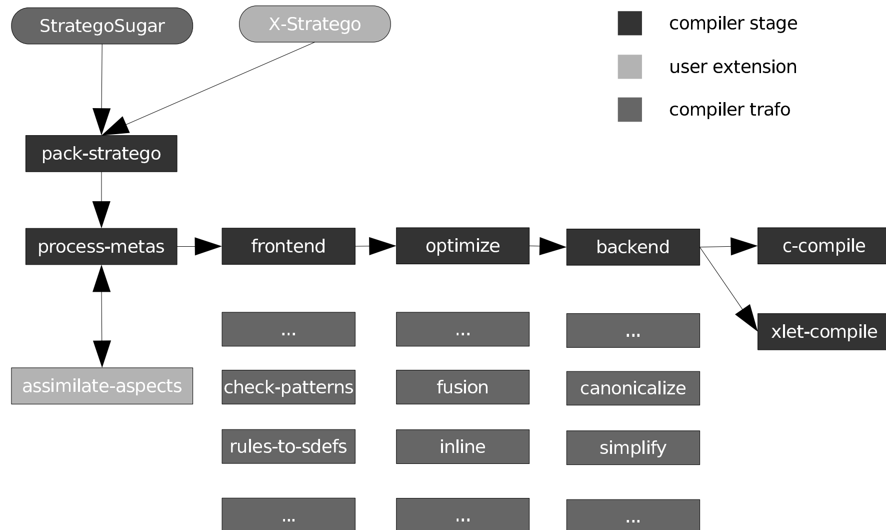


Figure 6.3: The extensible MetaStratego pipeline. The component `assimilate-aspects` is the assimilator for the AspectStratego language described previously. Using the extension points, the assimilators can hook into the various stages of the compiler.

```
signature StrategoCore
sorts Env Strat Var Pat
ops
  GChoice : Env × Strat × Strat × Strat → Env
  Match : Env × Pat → Env
  Build : Env × Pat → Env
  Scope : Env × List(Var) × Strat → Env
  Seq : Env × Strat × Strat → Env
  One : Env × Strat → Env
  All : Env × Strat → Env
  Call : Env × Var → Env
  Fail : Env → Env
  Id : Env → Env
```

Figure 6.4: Signature for the StrategoCore language. The semantics of most operators were described in Chapter 3.

in this dissertation are able to translate the entire extension into plain Stratego before the front-end stage. This may not always be possible. For example, additional optimisation opportunities may arise as a result of the extension. Plugging into the `optimize` stage may therefore be desired. For this reason, MetaStratego pipeline exposes new extension points into the compiler pipeline. New transformations may be added before or after each compiler stage shown in Figure 6.3.

A given extension may attach several assimilators to the various stages in the compiler. When a given extension point is reached, all registered assimilators for that stage will be executed. A limitation of the current extension scheme is that all extensions must be serialisable. That is, the order of all assimilators must be linearised and they must be executed sequentially. Multiple and co-existing language extensions are still possible, but they are difficult to use because the user must ensure that the assimilators are listed in the correct order in the meta file (or section).

6.1.4 StrategoCore

The output at the end of the backend stage in Figure 6.3 is a program in a minimal core language called StrategoCore. This language, specified in Figure 6.4, is the very close to the barest minimum required for implementing System S. Translating plain Stratego into StrategoCore is a stepwise process. It is complete at the end of the front-end. Various optimisations in the `optimize` stage, such as optimisation of pattern matching, are performed on the core format.

Both the MetaStratego and the Stratego compiler have an option to output pro-

grams in the core format. The core format is independent of any operating environment or hardware architecture. It is therefore a good candidate format for representing Stratego programs in a portable manner which can be loaded dynamically on a sufficiently capable transformation runtime.

6.2 An Extensible Runtime

The extensible compiler is complemented with an extensible runtime for Stratego. This runtime is designed around the general term library interface described in Chapter 4. It allows the runtime to perform rewriting on any data structure which can be mapped to this interface. The primary motivation for constructing this runtime was to enable large-scale reuse of transformation systems by plugging them into development environments, compilers and other language infrastructures.

To support these experiments, the runtime has been implemented in Java. Java is the *lingua franca* of modern software development and a substantial collection of front-ends and compilers for various languages have been implemented in Java, for C, C++, SQL, Python, Ruby, Fortress and others. Integrated development environments, like NetBeans and Eclipse, are frequently implemented in Java.

An additional feature of the runtime is its ability to load Stratego programs (in the StrategoCore format) dynamically at runtime. This feature makes it possible to extend a transformation system dynamically with new functionality. This is used in the Stratego development environment described in Chapter 9.

6.2.1 Design

The runtime provides an interpreter for StrategoCore files and extension facilities for plugging in new program object models and foreign function libraries. The interpreter is called Stratego/J, but is sometimes referred to as the MetaStratego runtime. An example instantiation of this runtime is shown in Figure 6.5.

In this diagram, two program object model adapters have been plugged into the interpreter (`org.spoofax.interpreter`). Program object models for the Eclipse Compiler for Java are adapted by the `org.spoofax.interpreter.adapter.ecj`. The component `org.spoofax.interpreter.adapter.aterm` adapts the ATerm library. Additional foreign functions for calling into the Eclipse platform – e.g. for opening windows in Eclipse and hooking into menus – are provided by `org.spoofax.library.eclipse`.

The runtime is supplemented with a scannerless GLR parser implemented in Java by the author. This parser, called jsglr, is compatible with the SGLR [Vis97] parser, developed at CWI, The Netherlands. It enables complete systems with the complete transformation cycle (parse-transform-unparse) to run on the Java platform and be plugged into development environments.

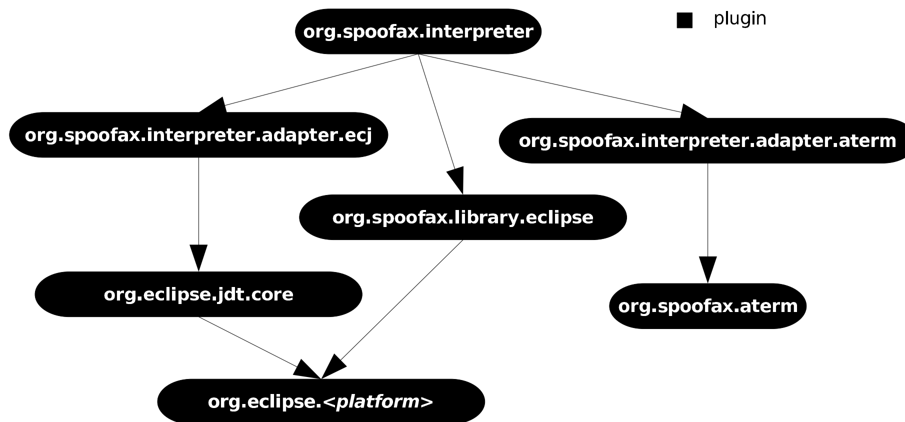


Figure 6.5: Components of the Stratego/J runtime.

6.2.2 Implementation

The implementation scheme of the interpreter engine is largely unremarkable. It follows the semantics of System S [BvDOV06] closely. A choice point stack [Mor98] is used to support backtracking. Some minor optimisations have been added to keep the stack depth at a minimum by throwing away stack frames whenever possible. This is an important consideration because Stratego programs are often deeply recursive.

6.2.3 Performance

The current performance of the interpreter is significantly slower, around 10–20 times, than that of natively compiled Stratego code. Additional optimisations are possible, especially in the pattern matching code, but it is unlikely that the interpreted code will ever perform on par with the natively compiled output of the Stratego compiler. The interpreter is capable of executing the Stratego (and the MetaStratego) compiler, thereby completely hosting Stratego on the Java Virtual Machine (JVM).

An attractive future direction for the Stratego compiler would be to add support for compiling strategy definitions to Java bytecode. It may also be worthwhile to experiment with an incremental (just in time) compilation scheme so that compilation only happens for frequently used strategies.

6.3 Light-Weight and Portable Transformation Components

The Stratego/XT environment provides a conceptually attractive component system called XTC [BKVV06], but this system suffers from some rather severe performance issues. It is based around the concept that every component runs in its own process. These processes are composed using (UNIX) pipes. Each component is a console application that reads its input from the standard input stream and returns its result on the standard output stream. All data is thus serialised from one component to another via a the operating system file API and this incurs considerable performance penalties both because of serialisation and because of process startup. A significant advantage of the model is that composition can be done at deployment time, instead of at compile-time – components may easily be swapped in and out after they are compiled.

Because of its performance limitations, XTC is currently being phased out of Stratego/XT and replaced by more traditional, dynamically linked libraries. This resolves the performance issues, but all components must be linked into the transformation pipeline statically. Also, the build processed is complicated slightly because of this fact since all components must be accounted for ahead-of-time. There is still a place for easily deployable transformation components, however, and especially when integrating transformation systems with interactive environments. This is why the light-weight transformlet component system presented next was designed. It should not be considered a full replacement for XTC, however.

6.3.1 Transformlets

A *transformlet* is a small, self-contained transformation component produced from a Stratego module (program) using the MetaStratego compiler. Figure 6.6 shows an example transformlet. A transformlet must declare information about itself, called meta information. It may also declare that it is capable of extending specific hooks in the environment that it will be loaded into.

The meta information must be declared using the strategy `xlet-meta-info`, as shown in the example. Deployment of the transformlet requires this meta information for packaging the transformlet into a deployable transformation component called an `.xlet` file.

When a transformlet (an `.xlet` file) is loaded into the runtime environment it undergoes *activation*. The runtime actives a transformlets by querying the transformlet for the presence of hooks. This is done by calling a strategy provided by the transformlet named `xlet-define-hooks`. The developer of a transformlet must know the names of the available hooks in the deployment environment when the transform-

lets is written. Extension of hooks are declared in the strategy `xlet-define-hooks`. This strategy will be invoked when the transformlet is loaded into its environment and must produce a list of tuples. Each tuple contains the name of a hook and the strategy that should be invoked when the hook is invoked.

Using this hook mechanism, it is possible to express open-ended callbacks. This is useful for interactive environments. When the runtime is asked to execute actions for a given hook, all registered strategies will be executed. This mechanism is used internally by the interactive development environment described in Chapter 9 to provide extensibility via user-written scripts.

6.3.2 Implementation

Each transformlets is composed of two parts: the *package descriptor* and the *core program*. It is represented as a (possibly compressed) ATerm. Consider the code in Figure 6.6. A valid transformlet is obtained by first compiling this module into a StrategoCore file. This file is a term representing the entire transformation program. A program called `xlet-make` is then applied to this term. `xlet-make` extracts and removes the meta information embedded in the core file. That is, the strategy `xlet-meta-info` is removed from the StrategoCore file. Finally, the extracted meta information and modified StrategoCore file are composed into a term corresponding to the signature in Figure 6.7. The root of the transformlet is an `XLet` term. The `Specification` subterm is the top-level StrategoCore term; this is where the modified program is placed. The meta information becomes the package descriptor.

6.4 Summary

This chapter described an extensible compiler for the Stratego transformation language together with a versatile and extensible transformation runtime. The platform supports dynamic loading of light-weight and portable transformation components called transformlets.

The basic infrastructure introduced in this chapter has been used as an experimentation platform for all the abstractions and case-study prototypes presented in this dissertation.

```
module example
imports
  spoofax/transformlet/-

strategies

  xlet-meta-info =
  !XLet(
      PackageDescriptor(
          Name("example")
        , Version("0.1.0")
        , [APIVersion("0.1.0")]
        , Dependencies([PackageRef(Name("spoofax"), APIVersion("0.1.0"))])
        , [ License("GPL-2")
          , Author(
              AuthorName("Karl Trygve Kalleberg")
            , AuthorEmail("karltk@stratego.org")
            )
          ]
        )
    , None
    )

  xlet-define-hooks = ![("hello-action", "hello")]

  hello = <debug> "Hello, World"
```

Figure 6.6: Example transformlet. The meta information is defined by `xlet-meta-info`. The strategy `xlet-define-hooks` defines that if the hook `hello-action` is invoked, the `hello` strategy should be called.

```
signature
  sorts
   XLet Name Version APIVersion Dependencies VersionRange
   MetaInfo AuthorName PackageDescriptor
  constructors
    XLet : PackageDescriptor × Specification → XLet
    PackageDescriptor: Name × Version × APIVersion × Dependencies ×
List(MetaInfo) → PackageDescriptor
    Name : String → Name
    Version : String → Version
    APIVersion : String → APIVersion
    Dependencies : List(PackageRef) → Dependencies
    PackageRef : Name × List(APIVersoin) → PackageRef
    VersionRange : Version × Version → VersionRange

    Author : AuthorName × AuthorEmail → MetaInfo
    AuthorName : String → AuthorName
    AuthorEmail : String → AuthorName

    License : String → MetaInfo
    Homepage : String → MetaInfo
    BugTracker : String → MetaInfo
```

Figure 6.7: Signature for the transformlet programs.

7

Strategic Graph Rewriting

This chapter introduces GraphStratego, a prototype extension to the Stratego programming language for rewriting on terms with references. The main motivation behind GraphStratego is to extend the Stratego language with support for the notion of cycles in its program model. Many abstract program models require support for circular structures. Perhaps the most commonly used are the various flow graphs, such as for control- and data-flow, employed in compilers and program analysers. These models are often language independent.

GraphStratego improves the language independence for transformations in the sense that the abstract program models mentioned above may now be captured natively and processed using new language constructs. The extended language allows succinct formulations of transformation programs which traverse and rewrite graph-like models.

This chapter is a verbatim reprint of the paper *Strategic Graph Rewriting: Transforming and Traversing Terms with References*[KV06] written with Eelco Visser, with the exception of some minimal formatting changes.

7.1 Abstract

Some transformations and many analyses on programs are either difficult or unnatural to express using terms. In particular, analyses that involve type contexts, call- or control flow graphs are not easily captured in term rewriting systems. In this paper, we describe an extension to the System S term rewriting system that adds references. We show how references are used for graph rewriting, how we can express more transformations with graph-like structures using only local matching, and how references give a representation that is more natural for structures that are inherently graph-like. Furthermore, we discuss trade-offs of this extension, such as changed traversal termination and unexpected impact of reference rebinding.

7.2 Introduction

Strategic programming is a powerful technique for program analysis and transformation that offers the separation of local data transformations from data traversal logic. The technique is independent of language paradigm and underlying data structure, but is perhaps most frequently found in functional and term rewriting languages. Much of its power comes from the ability to define complex traversal strategies from a small set of traversal combinators. Traversal strategies are often used to traverse terms.

Terms, when implemented as maximally shared, directed acyclic graphs, have many desirable properties for representing abstract syntax trees (ASTs) as used in program transformation. In the ATerm model [vdBdJKO00], maximal sharing of subterms means that all occurrences of a term are represented by the same node in memory. Consequently copying of terms is achieved by copying pointers, and term equality entails pointer comparison. The model ensures *persistence*; modifying a term means creating a *new* term, the old term is still present. This makes it easy to support backtracking. Destructive updates of term references are not permitted, which allows efficient memory usage.

A consequence of the DAG representation is that terms cannot have explicit backlinks, i.e. *references* to arbitrary subterms elsewhere in the same term. Such references to other parts of the AST, including an ancestor of a term, are useful for expressing the results of semantic analyses as local information for rewrite rules. Turning ASTs into terms with references can turn some transformation problems from global-to-local into local-to-local. By adding explicit references in the terms, we arrive at a variation of term graphs [Plu01]. The added expressivity gained from term references allows us to encode graphs rather succinctly, and therefore also express structures that are inherently graph-like more naturally, such as high-level program models and many intermediate compiler representations, including call-, control flow- and type graphs, thus setting the stage for implementing transformations such as constant and copy propagation, type checking and various static analyses. We conserve the ability to express program transformations using local rewrite rules together with generic traversal strategies, obtaining a form of strategic graph rewriting. By adding explicit references, we also give up some of the desirable properties of terms mentioned above, as the references allow destructive updates, change the termination criteria for traversals, alter the matching behavior of rewrite rules and exhibit side effects due to reference rebinding.

In this paper, we explore the design of a strategic rewriting language on terms with references, and discuss the tradeoffs found in this design space. We will show that we can arrive at a practical and useful variation of term graphs that allows us to express graph algorithms and rewriting problems rather precisely.

The paper is organized as follows: In Section 7.3, we introduce basic term rewrit-

ing with Stratego, show new primitives for manipulating references and how existing constructs extend to handle terms with references. In Section 7.4, we show how the new language features are used to compute various common graph representations for programs from ASTs. In Section 7.5, we implement two basic graph algorithms: depth first search and strongly connected components, and their application to finding sets of mutually recursive functions. We also discuss an implementation of lazy graph loading. In Section 7.6, we discuss notable aspects of our implementation. In Section 7.7, we discuss related work. In Section 7.8, we discuss design tradeoffs and future work. We conclude in Section 7.9.

7.3 Extending Term Rewriting Strategies to Term Graphs

We now present the extension of the strategic term rewriting language Stratego with term references. First, we give an overview of the basic concepts of Stratego. Next, we extend terms to term graphs, i.e. terms with references, by introducing primitives for references. Finally, we discuss rewrite rules and generic traversals on term graphs.

7.3.1 Term Rewriting Strategies

The Stratego program transformation language [BKVV06] is an implementation of the System S [VB98] core language for term rewriting. We will not discuss all the features of System S and Stratego in this paper, but restrict ourselves mainly to conditional rewrite rules, strategies, traversals, and scoped, dynamic rules.

Terms A term t is an application $c(t_1,\ldots,t_n)$ of a constructor c to zero or more terms t_i. There are some special forms of terms such as lists ($[t_1,\ldots,t_n]$) and integer constants, but these are essentially sugar for constructor terms. A term pattern is a term p with variables x, written on the form $p(x)$.

Rewrite Rules A conditional rewrite rule, R : $p_l(x)$ -> $p_r(x)$ `where` c, is a rule named R that transforms the left-hand side pattern p_l to an instantiation of the right-hand side pattern p_r if the condition c holds. The following rule can be used to simplify addition expressions.

```
Simplify: Add(Int(x), Int(y)) -> z where <add> (x, y) => z
```

When applied to the term `Add(Int(1),Int(2))`, it will execute the condition `<add> (x,y)`, which is a strategy expression for computing the sum of two integers. The resulting term, 3, will be bound to the variable `z` using the operator => as assignment.

Strategies Strategies are used to implement rewriting algorithms. A *strategy* is built from primitive traversals (`one(s)`, `all(s)`, `some(s)`), combinators (`s1 <+ s2` (left choice), `s1 ; s2` (strategy composition)), identity (`id`), failure (`fail`) and invocations of rewrite rules or other strategies. The following strategy will attempt to apply the rule `Simplify` to all subterms of the current term. If `Simplify` fails, either because the left hand side pattern does not match, or because the condition does not hold, the strategy `id` will be applied instead. `id` always succeeds, and returns the identity (i.e. same term).

```
try-simplify = all(Simplify <+ id)
```

When applied to `Sub(Add(Int(1),Int(2)),Int(4))`, the first subterm of `Sub` will be rewritten to `3`, but `Simplify` will fail for the second subterm (`Int(4)`), and `id` will be applied instead. The end result is the term `Sub(Int(3),Int(4))`.

Strategy Expression	Meaning
`!`$p(x)$	*(build)* Instantiate the term pattern $p(x)$ and make it the current term
`?`$p(x)$	*(match)* Match the term pattern $p(x)$ against the current term
s_0 `<+` s_1	*(left choice)* Apply s_0. If s_0 fails, roll back, then apply s_1
s_0 `;` s_1	*(composition)* Apply s_0, then apply s_1. Fail if either s_0 or s_1 fails
`rec` $x(s(x))$	*(recursion)* Strategy x may be called in s for recursive application
`id, fail`	*(identity, failure)* Always succeeds/fails. Current term is not modified
`one(`s`)`	Apply s to one direct subterm of the current term
`some(`s`)`	Apply s to as many direct subterms of the current term as possible
`all(`s`)`	Apply s to all direct subterms of the current subterm
`\`$p_l(x)$ `->` $p_r(x)$`\`	Anonymous rewrite rule from term pattern $p_l(x)$ to $p_r(x)$
`?`x`@`$p(y)$	Equivalent to `?`x `;` `?`$p(y)$; bind current term to x then match $p(y)$
`<`s`>` $p(x)$	Equivalent to `!`$p(x)$ `;` s; build $p(x)$ then apply s
s `=>` $p(x)$	Equivalent to s `;` `?`$p(x)$; match $p(x)$ on result of s

Generic Traversal Strategies The primitive traversals `one(s)`, `some(s)` and `all(s)` can be composed using the strategy combinators to obtain *generic traversal strategies*, which are used to control the order of rewrite rule applications throughout a term.

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

The strategy `topdown(s)` will apply the strategy `s` to the current term before recursively applying itself to `all` the subterms of the new current term. `bottomup(s)` works similarly: `s` is applied to all subterms before the current term (with freshly rewritten subterms) is processed by `s`.

Build and Match Pattern matching is also available independently of rewrite rules by the match operator strategy, written `?`. The expression `?Add(Int(`x`),Int(`y`))` when

applied to the term `Add(Int(1),Int(2))`, will bind `x` to 1 and `y` to 2. The inverse operator of match is build, written `!`, which will instantiate a pattern. Given the previous bindings of `x` and `y`, `!Add(Int(x),Int(y))` will instantiate to `Add(Int(1),Int(2))`. Figures 7.1(a), and 7.1(b) show simple ground terms and their corresponding ATerms.

7.3.2 References

Thus far, the language we have presented only operates on plain terms. We now extend terms with *references*. That is, in addition to a constructor application, a term can now also be a term reference r. A reference can be thought of as a pointer to another term. It is a special kind of term that our language extension knows how to distinguish from other terms (refer to Section 7.6 for implementation details).

We conceptually extend the language with three new operators for operating on references: *create new reference*, *dereference*, and *bind reference*. The creation of a new reference produces a fresh, unbound reference with a unique identifier. The identifier is used to compare references with each other. Like terms, references may be passed around as parameters, copied, matched, and assigned to variables. Additionally, references may be bound. Binding a term to a reference is similar to binding a value to a variable. After binding, a reference may be dereferenced. A deference will produce the term which was previously bound. These conceptual operators are available inside pattern expressions in three different forms, giving us *term graph patterns*.

Build and bind: `!`r~$p(x)$ will instantiate the term pattern $p(x)$ and bind the resulting term to a new, unique reference which will be bound to the variable r. If the variable r is already bound to a reference, a new reference will not be created. Instead, the reference of r will be rebound to the new term.

Bind or match: `?`r~$p(x)$ matches a reference r bound to a term that matches the term pattern $p(x)$. More specifically, to succeed, this expression must be applied to a reference r', r' must have the same reference identifier as r, and r' must be bound to a term which matches the term pattern $p(x)$. If the variable r is unbound, r will be bound to r' before the matching starts.

Dereference: `^r` will dereference the reference r, i.e., if `r` is bound to the term `t`, `^r` will produce `t`. With `r` bound to `t`, `?^r` is equivalent to `?t` and `!^r` is equivalent to `!t`.

With these operations, we can instantiate the term graph in Figure 7.1(c): `!r~Int(2); !Sub(r,r)`. Or more succinctly as one term graph pattern: `!Sub(r~Int(2),r)`. In the following we use some idioms: When we need a reference, but do not yet have its term, we use the expression `!r~()` to create a new reference bound to the "dummy" term `()`. If we want to match a reference, but do not care what it is bound to, we write `?r~_`.

General Graphs Term references may also be used to construct more general graphs, such as those shown in Figures 7.1(d) and 7.1(e). When constructing mutually de-

pendent graphs, such as the `f()` and `g()` nodes in Figure 7.1(d), term graph construction must always be split into two stages. First, one half must be built with an unbound reference, e.g. `!f~FunDef("f",[g~()])`, then the graph is connected when the other half is built: `!g~FunDef("g",[f])`. Note the use of the idiom `g~()` to break cycles.

7.3.3 Rewrite Rules and References

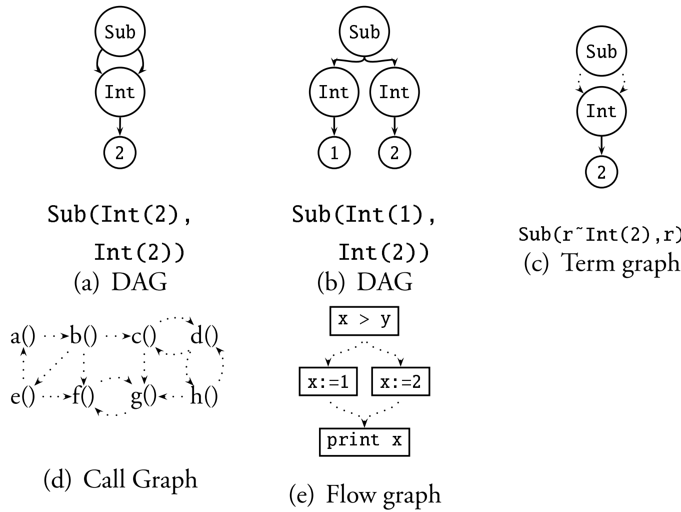
Conditional rewrite rules on term graphs, R: $g_l(x)$ `->` $g_r(x)$ `where` c, mirror rewrite rules on terms. $g_l(x)$ and $g_r(x)$ are term graph patterns, as described previously and c is the rule condition. `Simplify` can now be reformulated to work on term graphs:

```
Simplify: r0~Add(r1~Int(x~_), r2~Int(y~_)) -> r0~Int(r3)
  where !r3~Int(r4~<add> (^x,^y))
```

Rewrite rules on term graphs will not maintain maximal sharing unless the programmer takes explicit care. This leads to differences in the equality checking of term graphs compared to equality checking of term. For efficiency, the built-in comparison of term graphs only exists in a "shallow" form, i.e. identity checking: Two terms with references are equal iff all subterms are structurally equal, and all references have the same identity. This means that terms may now in fact be structurally equal, but differences in their reference identities will prevent the shallow equality test from uncovering this.

Rebinding of References For terms, maximal sharing and constant time equality checking is always guaranteed by the ATerm library. When matching a regular variable against a term, the pointer to that term gets copied when it is used in a build. If the original term is later modified, copy-on-write is performed behind the scenes to ensure referential transparency. For term graphs, this is no longer the case, as references may be rebound at any time. Consider the graph building expression `!Sub(r~Int(2),r) => a ; !r~Int(3)`. Here, we assign the graph from Figure 7.1(c) to the variable `a`, but subsequently change the binding of the references contained in the term graph of `a`. Effectively, this will change the value of `a` after `a` was bound. This may seem dangerous, as it opens up for problems related to lack of referential transparency. Certainly, these issues must be managed, but it is important to note that the binding of terms to references is always done explicitly. It is not possible to retroactively create a reference to a subterm of another term. E.g., if the term `Sub(Int(2),Int(2))` from Figure 7.1(a) is bound to the variable `v`, it can never change, as it does not contain any references.

If side effects are unwanted, in the sense that references in the term of an already bound variable should never change, assignments of term graphs should be coupled



(a) DAG (b) DAG (c) Term graph (d) Call Graph (e) Flow graph

Figure 7.1: Examples of graphs supported by our language. References are shown as stippled edges.

with a call to `duprefs`, e.g. `!r~Int(2); !Sub(r,r); duprefs => a; !r~Int(3)`. Here, the references in the term graph `Sub(r,r)` will be replaced with new, unique references before the assignment, so that subsequent rebindings cannot affect the value of a.

7.3.4 Term Graph Traversal

We will now show how generic traversals are adapted to work on term graphs through an example of term graph normalization. Our goal is to use `Simplify`, shown earlier, to simplify the term graph `Sub(r~Add(r'~Int(1),r'), r)`. Specifically, we only want to simplify each referenced term once. This illustrates how "side-effects" can be used beneficially, and how term graph rewriting can be more efficient than term rewriting: we only need to consider identical terms once, because we can recognize them by their reference identifiers. This argument is only valid once a proper term graph has been constructed, however. Our current implementation makes no attempt at maintaining such term graph properties globally during arbitrary rewriting sequences.

Phased Traversals To manage termination of graph traversals, we introduce a concept of *phases*. Phases are used to ensure that each reference is only visited once, so that loops in the graph do not result in non-termination. We do this by introducing a new primitive strategy, `phase(s)`, and new variants of the primitive traversal operators: `wone(s)`, `wsome(s)` and `wall(s)`.

When applied to a reference r, `wall(s)` will first dereference r, obtaining the term t, then apply s to all subterms of t. The resulting term is rebound to r. If any subterm of t is also a reference, it will be dereferenced before s is applied, and rebound afterwards. `wone(s)` and `wsome(s)` are similar.

`phase(s)` will internally instantiate a new, globally unique marker and then apply `s` to the current term. Any invocations of `wone`, `wsome` and `wall` in `s` will take the marker into consideration. As each reference is dereferenced during the traversal, using `wone`, `wsome` or `wall`, the marker is placed on the reference. Any subsequent attempt at dereferencing will not yield a term result, thus making the reference untraversable. When the phase is exited, all markers for that phase are removed. It is possible to nest phases. The inner phase will instantiate a new, unique marker and can revisit references already visited by its enclosing phase.

During a phased traversal, it is sometimes necessary to control whether the dereferences due to matching or the ^ operator should be marked with the current phase marker or not. This can be controlled by using `wrap-ref(s)`, which dereferences, applies `s`, then rebinds, irrespective of any phase markers. Analogously, `wrap-phase-ref(s)` can be used to only visit unmarked references, and mark the reference after a visit.

Reference Expression	Meaning
`!`r~$p(x)$	*(Build and bind)* Instantiate term pattern $p(x)$ and bind result to r.
`?`r~$p(x)$	*(Bind or match)* See text in Section 7.3.2.
^r	*(Dereference)* Look up term for r. Fail iff r is unbound.
`duprefs`	Replace all references in the current term with new, unique ones
`phase(`s`)`	For traversals done by s, visit each reference at most once
`wrap-ref(`s`)`	Apply s to reference and rebind, irrespective of phase
`wrap-phase-ref(`s`)`	Apply s to reference and rebind, while respecting phase

Generic Graph Traversals The following traversal strategies are adaptations of the generic traversals for terms. They use phase markers to avoid visiting the same reference more than once. These strategies use `wall` and will rebind references they encounter to rewritten terms.

```
wtopdown(s)     = phase(rec x(s ; wall(x)))
wbottomup(s)    = phase(rec x(wall(x) ; s))
wdownup(s1,s2)  = phase(rec x(s1; wall(x); s2))
```

Term Graph Normalization With the phasing and generic graph traversals in place, we can now express term graph normalization simply and precisely as:

```
simplify = wbottomup(try(Simplify))
```

When applied to the term `Sub(r~Add(Int(1),Int(2)), r)`, the result of `simplify` is `Sub(0~Int(3), 0~)`. Here, `0~t` indicates that the reference 0 is bound to the term `t`. Bindings of references are only shown once, the first time they are encountered.

For more general rule sets, it may be necessary to exhaustively apply rules using a fixed-point strategy. The following strategy follows the same definition pattern as `innermost` for plain terms [JV01, JV03]:

```
winnermost(s) = phase(rec x(wall(x); try(s; x)))
```

The difference is that the `phase` mechanism ensures that a node in a term graph (where all subterms are references) is visited only once. Thus the strategy performs a bottom-up traversal and tries to apply the normalization strategy `s` to each node. If that succeeds, the result is transformed by a recursive call to itself. This would entail a complete bottom-up traversal of the resulting term. However, the subterms that have already been visited, i.e., normalized, will not be visited again. This property ensures efficient implementation of the strategy, a result that was obtained in the term case only through a specialization of the strategy to its argument rules [JV01, JV03].

7.4 From Terms to Term Graphs

In this section, we describe how ASTs can be turned into various types of graphs commonly found in compilers, such as use-def chains, call graphs and flow graphs. First, however, we turn our attention to the problem of computing term graphs from terms with maximal sharing. This is done using dynamic rules.

Dynamic Rules A dynamic rule *S* is a rewrite rule which is defined and possibly undefined at runtime, see [BKVV06]. The expression `rules(`*S*`: t -> r)` creates a new rule in the rule set for *S*. The scope operator `{| S : `*s*` |}` introduces a new scope for the rule set *S* around the strategy *s*. Changes (additions, removals) to the rule set *S* done by the strategy *s* are undone after *s* finishes (both in case of failure and success of *s*). Sometimes, multiple rules in a rule set *S* may match. To get the results of all matching rules in *S*, we can use `bagof-S`.

Computing Term Graphs The following strategy implements a top down traversal with a memoization scheme to efficiently construct term graphs from terms. For each term it encounters, the strategy checks if this term has been memoized in the dynamic rule *S*. If so, the term is replaced with its corresponding reference. If not, all its subterms are replaced with references recursively, then a new reference is created and recorded in *S*.

```
term-graph = {|S: rec x(S<+all(term-graph); ?t; !r~t; rules(S: t->r)|}
```

Applied to `A(A(B),A(A(B),A(B)))`, we get `3^A(1^A(0^B),2^A(1^,1^))`.

7.4.1 Use-Def Chains

The use-def chain is a data representation found in most compilers for recording links from the use of a variable to its closest definition or assignment. Such data flow information is the basis for many program optimizations, in particular constant and

copy propagation. A variable is said to be *used* when its value is read; it is said to be *defined* when it is assigned to, either at its declaration or by a later assignment statement. Def-use chains are links from the definition of a variable to all its uses. The algorithm we present below will record both def-use and use-def chains.

```
use-def       = {| Use, Def: def-to-use ; use-to-def |}
def-to-use    = Var <+ VarRef <+ Assign <+ If <+ wall(def-to-use)
use-to-def    = topdown(try(add-ref-to-var <+ add-ref-to-assign))
new-def(|v,r) = rules(Def : v -> r)
add-use(|d,u) = rules(Use :+ d -> u)

add-ref-to-var    = ?r~Var(v,e,_); !r~Var(v,e, Uses(<bagof-Use> r))
add-ref-to-assign = ?r~Assign(v,e,_); !r~Assign(v,e,Uses(<bagof-Use> r))

If: If(c,t,e) -> If(c',t',e') where <def-to-use> c => c'
  ; <def-to-use> t => t' \Def/ <def-to-use> e => e'

Var: Var(v, e) -> r where <def-to-use> e => x
  ; !r~Var(v,x,Uses([])); new-def(|v, r)

VarRef: VarRef(v) -> r where <bagof-Def> v => defs
  ; !r~VarRef(v, Defs(defs)); <map(add-use(|<id>, r))> defs

Assign: Assign(v, e) -> r where <def-to-use> e => x
  ; !r~Assign(v, x, Uses([])); new-def(|v,r)
```

The `use-def` algorithm assumes the existence of the following term constructors: `If`, for `if` constructs, `Var`, for variable definitions, `VarRef` for variable (de)references and `Assign` for assignments. It is divided into two parts, `def-to-use` and `use-to-def`. For `def-to-use`: If a definition of a variable is seen, i.e. an `Assign` or `Var` term, this term is replaced with a reference to itself, and a mapping from the variable name to the reference is recorded in the dynamic rule `Def` using `new-def`. This is done in the `Var` and `Assign` rules. When a variable use is subsequently seen, it is also replaced by a term to itself by the `VarRef` rule. Its name is looked up in the `Def` rule, and references to the closest definitions are added using `bagof-Def`, see `VarRef`. The `Use` rule is updated to record the reference to this use, using `add-use`. Special care must be taken in the case of control constructs. We only show the case for `If`. Here, one rule set is computed for each branch, and the rule sets are joined afterwards, using the rule set union operator, `\Def/`. This ensures that new definitions from both branches are kept.

For `use-to-def`: In this pass, each `FunDef` is updated to contain references to the uses recorded by the previous pass, in the `Use` rule. When applied to a term for the program

```
var x := 0; if (x > 0) { x := 1 + x } else { x := 2 + x } ; print x
```

we get (read `Asn` as `Assign` and `VRef` as `VarRef`):

```
Block([~5,If(Int(0),Block([~1]),Block([~3])),Print(~0)])
~0 = VRef("x",Defs([~3,~1]))  ~1 = Asn("x",Add(Int(1),~2),Uses([~0]))
~2 = VRef("x",Defs([~5]))     ~3 = Asn("x",Add(Int(2),~4),Uses([~0]))
~4 = VRef("x",Defs([~5]))     ~5 = Var("x",Int("10"),Uses([~4,~2]))
```

7.4.2 Call Graphs

Another common program representation in modern compilers is the call graph. It records the interrelationships between the functions of a program, i.e. which functions call which, and is used for various static analyses such as reachability analysis, optimizations such as dead code removal and by documentation generation tools. The following code transforms an AST into a call graph by introducing references from all call sites (`Call` terms) to the corresponding function definition (`FunDef`) terms, and by adding a reference from the `FunDef` being called (callee) to the `FunDef` of the calling function (caller). Figure 7.1(d) illustrates the forward direction.

```
compute-call-graph   = {| FunLookup:  add-refs ; add-call-markers |}
introduce-references = topdown(try(AddFunRef))
with-fundefs(s)      = Program(map(s), id)
register-fun         = ?r; ?r~FunDef(_,_,_,_); rules(CurFun: _ -> r)

AddFunRef: x@FunDef(n,_,_,_) -> r where !r~x; rules(FunLookup: n -> r)

add-call-markers = {| CalledBy, CurFun:
    with-fundefs(wdownup(try(register-fun), try(AddCallRef)))
  ; with-fundefs(wrap-ref(AddCalledByRef)) |}

AddCallRef: Call(n, xs) -> Call(n, xs, r)
where <FunLookup> n => r ; CurFun => z ; rules(CalledBy :+ n -> z)

AddCalledByRef: FunDef(n,a,t,b) -> FunDef(n,a,t,ns,b)
where <bagof-CalledBy> n => ns
```

Three dynamic rules are used in this algorithm. `FunLookup` is used to map names of functions to their corresponding `FunDef`. `CurFun` is used to keep a reference to the `FunDef` we are currently inside. `CalledBy` is used to accumulate a set of callees for a given function name.

The algorithm works as follows. First, we replace every `FunDef` n node with a reference to n, and record the function name in the `FunLookup` rule set. This is done

by `add-refs`. Second, we do a downup traversal, where the current function is kept in the dynamic rule `CurFun` on the way down. On the way up, we add a reference to the destination `FunDef` f for any `Call` encountered, and register the current function in the `CalledBy` set for f. This is the first part of `add-call-markers`. Third, we place the callee sets collected in the `CalledBy` dynamic rule set on the corresponding `FunDefs`, finally obtaining a bidirectional call graph.

7.4.3 Flow Graphs

Flow graphs are used to represent the control flow of a program, analogously to the way use-def chains represent data flow. Flow graphs, along with use-def chains, are at the heart of many flow-sensitive optimizations, such as constant folding, loop optimization, and jump threading. We can compute a flow graph from the various statements in the AST as follows. In `If(c,t,e)`, flow goes from the condition c to both branches, t and e, which in turn go to the successor block of `if`. In `While(c,b)`, flow goes from the condition c to the body b, and from c to the successor block. The body b always flows back to the condition c. All other statements correspond to basic blocks: the flow from one statement goes directly to the successor block.

We show a three pass algorithm, `ast-to-flow-graph`. First, we do rewrites of control flow constructs locally, as described above, with the `MarkControlFlow` rule set. In the case of `If` and `While`, temporary `FlowT` blocks are inserted with dummy references, since the successor block is not known locally yet. Second, AST statement blocks are split into basic blocks, with `SplitBlocks`. Each non-control statement is rewritten to a Flow block. Third, the `FlowT` blocks are connected to the `Flow` blocks produced in (2), and rewritten to `Flow`, resulting in a flow graph, as seen in Figure 7.1(e).

```
ast-to-flow-graph = bottomup(try(MarkControlFlow))
  ; bottomup(try(SplitBlocks))
  ; wbottomup(try(\ FlowT(x,y) -> Flow(x,y) \))

SplitBlocks: Block(xs) -> r where
  <map(?FlowT(_,_) <+ {r: \ t -> Flow(t, r) where !r~() \})> xs => xs'
; foldr(\ (f@Flow(t1, n), t2) -> f where !n~t2 \
     <+ \ (f@FlowT(t1, n), t2) -> f where !n~t2 \ |None)
; <Hd> xs' => hd ; !r~hd

MarkControlFlow: If(cond, thn, els) -> FlowT(If(cond', thn', els'), next)
where !next~(); ; !thn'~Flow(thn,[next])
  ; !els'~Flow(els,[next]); !cond'~Flow(cond,[thn', els'])

MarkControlFlow: While(cond, body) -> FlowT(r, next)
```

```
where !body'~(); !next~(); !cond'~Flow(cond, [next, body'])
  ; !body~Flow(body, [cond']); !r~While(cond', body')
```

7.5 Graph Algorithms and Applications

In this section, we show how some basic graph algorithms can be implemented using the reference mechanism we have described in Section 7.3.

Depth First Search Our depth first search implementation works on graphs where each node is a term. The algorithm takes two parameters, `l` and `es`. `es` will be used to compute the outgoing edges from each node. `dfs` keeps track of the current depth during visits. On a visit to a node, the strategy `l` will be called with the current depth value as parameter, so that it can be used to compute the label for the current node, or for other transformations.

```
dfs(l : a * a -> a, es)     = phase(wall(dfs(l, es | 0)))
dfs(l : a * a -> a, es | n) =
    wrap-phase-ref(where(es => edges)
  ; where(l(|n) => label)
  ; where(<wall(dfs(l, es | <inc> n))> edges) ; !label)
```

The traditional depth first search, as described in for example [CLR97], is applied initially to the set V of a graph $G = (V, E)$. We get the same behavior by applying `dfs` to a list of references to all nodes in the graph. We will demonstrate the use of the `dfs` strategy next, when we discuss strongly connected components.

Strongly Connected Components The basic algorithm for strongly connected components (SCC) is also described in [CLR97], and consists of four steps: First, call `DFS(G)` to compute finishing times `f[u]` for each vertex `u`. Second, compute the transposed graph `GT=transpose(G)`. Third, call `DFS(GT)`, but in the main loop of `DFS`, consider the vertices in order of decreasing `f[u]`. Fourth, produce as output the vertices of each tree in the `DFS` forest formed in point 3 as a separate strongly connected component.

In our implementation of SCC, shown below, we avoid actual graph transposition by requiring one strategy, `es` for computing forward edges from a node, and another, `res`, for computing reverse edges. We also combine the third and fourth step by using a modified `dfs`, called, `dfs-collect`, which collects each set of SCCs into a list during the third step.

```
dfs-collect(l : a * a -> a, es) =
  phase(all({|C: dfs-collect(l,es|0) ; bagof-C|}))
```

```
dfs-collect(l : a * a -> a, es | n) = ?r~_
  ; wrap-phase-ref(where(es => edges) ; where(l(|r) => label)
  ; where(<all(dfs-collect(l, es | <inc> n))> edges) ; !label)

sort-fundefs =
  sort-list(LSort(where((?r;!^r; FinishTime,?r';!^r'; FinishTime); gt)))
collect-components(|r) = rules(C :+ _ -> r)
inc-time = (Time <+ !0) => n ; where(inc => n'; rules(Time: _ -> n'))
time-count(|n) = ?x; where(inc-time => n'); rules(FinishTime: x -> n')

scc(l : a * a -> a, es, res) = {|FinishTime, Time:
    dfs(l, es)
  ; sort-fundefs
  ; dfs-collect(collect-components, res)
  ; filter(not(?[])) |}
```

The current time is maintained in the `Time` dynamic rule, and the finishing time in `FinishTime`. Our `scc` should normally be called with the `time-count` strategy as its first argument, but the user is free to adapt this.

7.5.1 Finding Mutually Recursive Functions

Suppose we want to use SCC to compute sets of mutually recursive functions. Then, each node in the graph is a function f. The outgoing edges of f are references to the functions *called by* f. The incoming edges of f are references to the functions *calling* f. This graph is what was computed by `call-graph`, discussed in Section 7.4.2. The following strategies may used for edge computations.

```
calls-as-outbound    = collect(\ Call(_,_,x) -> x \)
calledby-as-outbound = collect(\ FunDef(_,_,_,x,_) -> x \) ; concat
```

Applying `scc(time-count, calls-as-outbound, calledby-as-outbound)` to a list of references to all functions in a program, say Figure 7.1(d), will produce the cliques (a, b, e), (f, g) and (c, d, h).

7.5.2 Lazy Graph Loading

Instead of binding terms to references, strategies may be bound instead. When a reference r with the strategy s bound to it is dereferenced, s is invoked, and the resulting term is taken as the term value for r. We call this an *active reference* since it has a strategy (i.e., function) attached to it that is activated and executed upon dereference. Active references are useful for term (graph) rewriting of larger terms,

especially when doing sparse analyses on larger bodies of program code. With active references, terms for programs can be loaded as skeletons. For example, all bodies of functions or classes may be left out, and be parsed and loaded, or even generated, on demand.

7.6 Implementation

We have implemented a prototype of the language extension described in this paper. Our implementation is a conservative extension to the existing Stratego infrastructure: Every valid Stratego program retains its behavior and terms without references are still represented entirely as ATerms. References are introduced as a special kind of term, `Ref`, and we have modified the language implementation to recognize and treat terms of this type specially. `Ref`s are closely related to pointers, as found in C, and to references, as found in Java. Unlike pointers and Java references, a Stratego `Ref` always starts out as bound. It may subsequently be rebound to another term. The bindings from references to terms are maintained in a global table, or more precisely, in a global, dynamic rule set. When a new reference is bound, a new rule is added to the set. When an existing reference is rebound, its corresponding rule is changed. Using dynamic rule sets aids in implementing backtracking behavior. For left- and guarded choice, references rebound or introduced by a failed strategy should be backtracked before the program proceeds. This is implemented in our compiler by rewriting every left choice operator to

```
start-ref-cs ; s1 ; commit-ref-cs <+ discard-ref-cs ; s2
```

Here, `start-ref-cs` will make a change set for the global rule set. If `s1` succeeds, the change set is committed and changes are kept. If `s1` fails, all changes to the reference rule set by `s1` are undone.

Managing the revisitation of references in term graphs is crucial for ensuring termination. The `wrap-phase-ref` mentioned earlier is responsible for this.

```
wrap-phase-ref(s) = ?r@Ref(_) < seen-before < id
+ where(<phase-deref> r; s; bind-ref(|r)) + s
```

`wrap-phase-ref` is implemented using guarded choice $s_1 < s_2 + s_3$, which works as follows. If s_1 succeeds, s_2 will be applied to the resulting term. If it fails, s_3 will be applied to the initial term. If `wrap-phase-ref(s)` is applied to term, `s` is applied and we are done. When at a reference r, we first use `seen-before` to check if we have seen r before. If so, we ignore `s` (by applying `id`, then returning). If not, r is dereferenced and marked, using `phase-deref`, `s` is applied to its term, and r is rebound by `bind-ref`.

Using `wrap-phase-ref`, we can now implement new traversal primitives on references. Let us consider `wall(s)`:

```
wall(s) = is-ref
  < wrap-phase-ref(all(wrap-phase-ref(s))) + all(wrap-phase-ref(s))
```

If we are not at a reference, `all` will be applied to the current term and any references it has as direct subterms will be marked. If we are at a reference, we will mark, transform then rebind it. The markers used by `wrap-phase-ref` can be managed using `phase(s)`, given next:

```
phase(s) = where(local-phase-ctr => pc; inc-phase-ctrs)
  ; start-seen-cs; s; discard-seen-cs
  ; where(restore-local-phase-ctr(|pc))
```

`phase(s)` works as follows: Before *s* is applied, a new, unique, internal phase marker is produced using `local-phase-ctr`, then the counter is increased, preparing for the next invocation. `start-seen-cs` enters a new "scope" for this marker. The counter is maintained in a dynamic rule defined inside `increase-phase-ctrs`, and is later used by `phase-deref` and `seen-before`. Once *s* completes, all markers will be discarded.

Our implementation has not yet been tuned for performance. While we have used Stratego's dynamic rules for implementation convenience, we only rely on the ability of dynamic rules to provide hash tables with change sets. In the current implementation, reference lookup time is linear in the depth of choices on the stack. A more efficient implementation of hash tables with change sets is likely to improve performance.

7.7 Related Work

Term graph rewriting theory is an active field. For an introduction and summary, see [Plu01]. A calculus for rewriting on cyclic term graphs has been proposed by Bertolissi [Ber05]. Many systems exist for general graph rewriting, such as PROGRES [Sch04] and FUJABA [NNZ04]. Few term graph rewriting systems for practical applications exist. HOPS [Kah99] and Clean [PvE98] are a notable exceptions. Claessen and Sands [CS99] describe an extension to the Haskell language which adds references with equality tests. Their goal is to better describe circuits, which are graph structures with cycles, in a purely functional language. Their references are immutable once created, unlike ours, making rewriting more difficult express. Lämmel et al [LVV03] discusses how strategic programming relates to adaptive programming, a technique found in aspect-oriented systems for traversing object structures. They show how traversal strategies may be implemented for cyclic structures, such as graphs, by keeping record of visited nodes. Our phased traversals expand upon this by allowing nested, overlapping traversals and fine-grained control of visitation marking. Our implementation shares some features with monadic programming. Monads are sometimes described as patterns for using functions to transmit state

without mutation, and are described by Wadler [Wad92]. In our implementation, dynamic rules are the functions used to transmit the state, namely the internal graph references. Some functional languages, such as Clean [PvE98], are implemented as rewriting systems with implicit term graphs. Functions in Clean are graph rewrite rules on the underlying term graph. In our language, the choice between term and term graph rewriting and their corresponding tradeoffs is not fixed, but rather left to the programmer. An important goal for our language extension is to better capture graph-like program representations, and to offer convenient transformation capabilities for these. Many excellent and general graph libraries exist, and we are not aiming to replace these.

To the best of our knowledge, no other term graph rewriting system supports strategic term graph rewriting, using rewriting strategies and generic traversals.

7.8 Discussion and Further Work

The construction of use-def chains, call- and flow graphs shows how global-to-local problems are now local-to-local, as the remote context is available locally for rules to match on. The addition of references for this purpose also introduces the problem of traversal non-termination in the presence of cycles. We have shown how this can be managed by phases. Another issue of term references is the unexpected impact of reference rebinding, in the loss of referential transparency. The code `!Sub(r~Int(2)) => v ; !r~Int(3)` will alter the value of `v` after it is bound. Judicious use of `duprefs` can control this. Comparison of term graphs is currently done using weak equality; i.e., comparison references in terms is done based on identity, not structure, which allows constant time comparison. Deep comparison is available through the library, and is linear in the size of the term graphs. The pattern-based language constructs introduced in this paper for reference manipulation came about after trying to program with only the primitive operators create reference, bind reference and dereference. While these primitives are still at the heart of the implementation, the notation presented in this paper make them more convenient to use. Further exploration of the design space is warranted. One attractive extension is matching modulo references, which allows term patterns to be matched directly on terms with references, by implicitly visiting references during matching.

7.9 Conclusion

We have presented the design and implementation of an extension to the Stratego term rewriting language for rewriting on terms with references, and demonstrated its practical application through the construction of several common graph-based program representations found in compilers. The contributions of this paper include

the introduction of language abstractions for dealing with references within a rule-based term rewriting language, a demonstration of how term matching, building and rewrite rules can be combined with term references, how benefits of generic term traversal can be kept by using phased traversals to deal with non-termination due to cyclic graphs, and how backtracking can be combined with destructive graph updates to retain the strategic programming flavor of Stratego. We showed how our language can be used to implement some basic graph algorithms and how these can be applied to graph-based program representations. We discussed design tradeoffs related to introducing references in terms, including traversal termination and impact of reference binding.

Part V

Case Studies

8 Language Extensions as Transformation Libraries

The language abstractions proposed on this dissertation are provided in form of extensions to Stratego that were built as transformation libraries using the MetaStratego framework. There is little specific to Stratego that makes it an inherently extensible language. It is, however, extremely well suited for implementing language extensions.

This chapter contains a case study which serves to illustrate that some of the experience gained while conducting the primary investigation – language-independent software transformations – is also easily applicable to language extension in general. The language extensions proposed in this dissertation are formulated as transformation libraries complemented with a convenient notation, in the form of a syntax extension to Stratego, using some of the techniques introduced in [BV04]. The same techniques can be applied easily to any language. To further illustrate this, the author implemented a small language extension to the small toy language called TIL [Vis05b] for handling alerts.

The underlying extension technique was subsequently presented and discussed in the paper *DSAL = library+notation: Program Transformation for Domain-Specific Aspect Languages* [BK06] written with Anya Bagge. The paper distills and discusses the principal details of this approach to language extension. It follows the tradition of language embedding suggested in [Vis02, BV04] but focuses the non-local effects of the language embeddings due to the cross-cutting nature of the embedded language.

This chapter is a verbatim reprint of the above-noted paper with the exception of some minimal formatting changes.

8.1 Abstract

Domain-specific languages (DSLs) can greatly ease program development compared to general-purpose languages, but the cost of implementing a domain-specific language can be prohibitively high compared to the perceived benefit. This is more

pronounced for narrower domains, and perhaps most acute for domain-specific aspect languages (DSALs).

A common technique for implementing a DSL is writing a software library in an existing programming language. Although this does not have the same syntactic appeal and possibilities as a full implementation, it is a technique familiar to most programmers, and it can be done cheaply compared to developing a full DSL compiler. Subsequently, the desired notation may be implemented as a simple syntactic preprocessor. The cross-cutting nature of DSALs, however, makes it difficult to encapsulate these in libraries.

In this paper, we show a technique for implementing a DSAL as a *library+notation*. We realize this by implementing the library in a program transformation system and the notation as a syntactic extension of the subject language. We discuss our experience with applying this technique to multiple kinds of DSALs.

8.2 Introduction

The implementation of domain-specific abstractions is usually done by way of libraries and frameworks. Although this provides the semantics of the domain, it misses out on good notation and many optimisation opportunities. Implementing domain specific languages by adding notation (syntax) to a library, and then programming a simple compiler that translates from the notation into equivalent library calls is an easy and powerful technique, which is cost-effective in many larger domains. Both the libraries and the simple compiler can be implemented in general purpose languages without too much effort, and it is important to note that the library need not be implemented in the same language as the compiler. If the "library" language supports syntax macros, like Scheme [DHB92], or has a sufficiently powerful metaprogramming facility, like C++ templates [AG05], the translation task may be accomplished through the inherent meta-programming constructs of this language. Otherwise, a stand-alone preprocessor is commonly used. For example, adding complex numbers or interval arithmetic to Java, with an appropriate mathematical notation, can be accomplished by writing or reusing a Java library, and writing a simple translator from the mathematical notation into OO-style calls. The approach of adding notation to (object-oriented) libraries was explored in the MetaBorg project [BV04], where the subject language Java was extended in various ways using Stratego as the meta-programming language.

For domain-specific aspect languages, the translation story is different. Behind the notation visible to the programmer lie cross-cutting concerns which may reach across the entire program, possibly requiring extensive static analysis to resolve. The straight-forward translation scheme into library calls for the subject language is not applicable as we are no longer dealing with basic macro expansion. Instead, we shall

view aspects as meta-programs that transform the code in the base program. These meta-programs may be implemented with transformation libraries in a transformation language (which may be different from the subject language). This allows us to consider DSALs as syntactic abstractions over transformation libraries, analogous to the way DSLs are syntactic abstractions over base libraries in the subject language. That is, we do not translate the DSAL notation into library calls in the subject language, but rather to library calls in the transformation language. Provided that the transformation language has a sufficiently powerful transformation library for the subject language, writing a transformation library extension for a domain-specific aspect is an easy task. We will demonstrate this technique by example, through the construction of *Alert*, a small error-handling DSAL extension to the Tiny Imperative Language (TIL).

The main contributions of this article are: A discussion of how the *library + notation* method for DSLs can be applied to DSALs, if the library is implemented in a meta-language; an example of the convenience of employing a program transformation language in the implementation of DSALs, compared to implementation in a general-purpose language; and a discussion of our experience with this technique for several different subject languages and aspect domains.

The paper is organised as follows. We will begin by briefly introducing our DSAL example and the TIL language (Section 8.3), before we discuss the implementation of our DSAL using program transformation (Section ??). Finally, we discuss our experiences and related work (Section 8.5), then offer some concluding remarks (Section 8.6).

8.3 The Alert DSAL

Handling errors and exceptional circumstances is an important, yet tedious part of programming. Modern languages offer little linguistic support beyond the notion of exceptions, and this language feature does not deal with the various forms of crosscutting concerns found in the handling of errors, namely that the choice of how and where errors are handled is spread out through the code (with `if`s and `try/catch` blocks at every corner), leading to a tangling of normal code and error-handling code. Also, the choice of how to handle errors is dependent on the mechanism by which a function reports errors—checking return codes is different from catching exceptions, even though both may be used to signal errors. Confusingly, even the default action taken on error depends on the error reporting mechanism, from ignoring it (for return codes and error flags) to aborting the program (exceptions).

The Alert DSAL allows each function in a program to declare its *alert mechanisms*—how it reports errors and other exceptional situations that arise, and allows callers to specify how alerts should be handled (the *handling policy*), independent of

the alert mechanism. We use the word *alert* for any kind of exceptional circumstance a function may wish to report; this includes errors, but may also be other out-of-band information, such as progress reports. Typical examples of alert mechanisms are exceptions, special return values (commonly `0` or `-1`) or global error flags (`errno` in C and POSIX, for instance). Ways of handling alerts include substituting a default value for the alerting function's return code; logging and continuing; executing recovery code; propagating the alert up the call stack; aborting the program, or simply ignoring the alert.

The alert extension is a good example of a domain-specific aspect language. It allows separation of several concerns: the mechanism (how an alert is reported) is separated from the policy (how it is handled), and code dealing with alerts is separated from code dealing with normal circumstances. The granularity of the policies (i.e., to what parts of the code they apply) can be specified at different scoping levels, from expressions and blocks to whole classes and packages.

Separating normality and exceptionality has already been demonstrated with AspectJ [LL00], but the AspectJ solution is less notationally elegant, and fails to separate mechanism from policy (it only deals with exceptions).[1] Using domain-specific syntax makes the extension easier to deal with for programmers unfamiliar with the full complexity of general aspect languages. Our alert extension is described in full in [BDHK06]. Here, we will look at the implementation of a simplified version for the Tiny Imperative Language.

8.3.1 The TIL Language

The Tiny Imperative Language (TIL) is a simple imperative programming language used for educational [BKVV05] and comparison purposes in the program transformation community. The grammar for TIL is given in the appendix (Section 8.7). A TIL program consists of a list of function definitions followed by a main program. TIL statements include the usual `if`, `while`, `for` and block control statements, variable declarations and assignments. Expressions include boolean, string and integer literals, variables, operator calls and function calls. We will use the name TIL+Alert for the extended TIL language.

8.3.2 Alert Declarations and Handlers

An *alert declaration* specifies a function's alert mechanisms. Our simple extension allows two ways of reporting alerts; via a condition which is checked before a call, or via a condition checked after a call. The pre-checks allow a function to report invalid

[1] We are not experts on aspect orientation, but we believe that the full separation of concerns available with our alert system is difficult if not impossible to achieve with existing general aspect languages.

Alert declaration. *Alert declarations are given after the regular function declaration. Actual arguments and the function's return value are available in the alert condition expressions. Pre-alerts have a condition that is checked before a call to the function and typically involve checks on the arguments; post-alerts are checked after the call has returned, and typically involve the return code (accessible as the special variable* `value`*, legal only in alert conditions and handlers.).*

```
FunDecl AlertDecl      -> FunDecl
"pre" Exp "alert" Id   -> AlertDecl
"post" Exp "alert" Id  -> AlertDecl
"value"                -> Exp
```

Figure 8.1: Grammar for TIL function declarations with alert extension.

parameters (before the call, avoiding the need for checks within the function itself), while the post-checks can be used for testing return values. The syntax for alert declarations is given in Figure 8.1. As an example, the following function definition declares that the function `lookup` raises the alert `Failed` if the return value is an empty string:

```
fun lookup(key : string) : string
  post value == "" alert Failed
begin ... end
```

The following declaration specifies that a `ParameterError` occurs if `f` is called with an argument less than zero, and that if the return value is `-1`, an `Aborted` alert was raised:

```
fun f(x : int) : int
  pre x < 0 alert ParameterError
  post value == -1 alert Aborted
```

A *handler declaration* specifies what action is to be taken if a given alert is raised in a function matched by its call pattern (the syntax is shown in Figure 8.2). The call pattern can be either `*` (all functions) or a list of named functions, possibly with parameter lists. This corresponds to the *pointcut* concept in AspectJ [KHH+01]. The handler itself is a statement; it can reference the actual arguments of the call (if a formal parameter list is provided in the handler declaration), names from the scope to which it applies, and `value`—the return value of the function for which the handler was called. For example, this handler declaration specifies that the program should abort with an error message in case of a fatal error:

```
on FatalError in * begin
```

Handlers. *A handler associates a statement with an alert condition; the statement is executed if the alert occurs. The* use *statement substitutes a value for the return value of the alerting function.*

```
"on" Id "in" {CallPattern ","} Stat -> Stat
"use" Exp ";"                       -> Stat
```

Call patterns. *A * matches a call to any function. The second form matches a call to a named function; the third form makes the actual arguments of the call available to the handler.*

```
"*"                    -> CallPattern
Id                     -> CallPattern
Id "(" {Id ","}* ")"   -> CallPattern
```

Figure 8.2: Grammar for handler declarations. The notation `{X Y}*` means `X` repeated zero or more times, separated by `Y`s.

```
  print("Fatal Error!");
  exit(1);
end
```

The `use` statement is used to "return" a value from the handler; this value will be given to the original caller as if it was returned directly from the function called:

```
on Failed in lookup(k) begin
  log("lookup failed: ", k);
  use "Unknown";
end
```

The `on`-declaration is a statement, and applies to all calls matching the call pattern within the same lexical scope. If more than one handler may apply for a given alert, the most specific one closest in scoping applies.

TIL+Alert does not add anything that can not be expressed in TIL itself, at the cost of less notational convenience. For example, given the above alert and handler declarations, a call

```
print(lookup("foo"));
```

would need to implemented somewhat like

```
var t : string;
t = lookup("foo");
if t == "" then t = "Unknown"; end
print(t);
```

This cumbersome pattern should be familiar to many programmers (programming with Unix system calls, for instance, or with C in general): save the result in a temporary variable, test it, handle any error, resume normal operations if no error was detected or if the error was handled. Exceptions alleviate the need to check for errors on every return, but writing `try/catch` blocks everywhere a handler is needed is still cumbersome, and changing handling policies for large portions of code is tedious and error-prone.

8.4 Implementation of TIL+Alert

We have several possibilities when faced with the task of implementing a DSAL, or a language extension in general:

1. Compile to object code—write an entirely new compiler for the extended language.
2. Compile to unextended language—write an aspect-weaving preprocessor for an existing compiler.
3. Compile to aspect language—write a preprocessor for an existing aspect weaver.

The first choice is typically the most costly, and therefore also the least attractive. The second option is a common technique for bootstrapping new languages, and was used for both C++ and AspectJ. The third option is only possible if the subject language we are extending already supports a form of aspects which can be suitably used for writing implementing (most of) the semantics of our DSAL. We will discuss this option in more detail in Section 8.5.

DSALs are almost by definition extensions of existing languages, and we can therefore expect to have at least some language infrastructure. In other words, we need only consider the latter two situations above. In our experience, implementing the aspect extension as library + notation in a program transformation system is a very efficient approach in terms of development time.

8.4.1 DSAL = library + notation

We have said that (alert handling) aspects are meta-programs, then showed the programmer notation for these in Section 8.3.2 where we discussed the alert grammar. This covers the “notation” half of our equation. Now we will discuss how the semantics are implemented as a transformation library written in a program transformation system.

Stratego/XT [BKVV06] is our implementation vehicle of choice. Stratego is a domain-specific language for program transformation based on the paradigm of

strategic programming [LVV03] and provides many convenient language abstractions for our problem domain. The language is bundled with XT, a set of reusable transformation components and generators—in particular a formalism for defining language syntax, called SDF [Vis97]—that support the development of language processing tools. In Section 8.5 we will discuss some of the benefits and drawbacks of using program transformation systems for implementing aspect weavers.

An existing language infrastructure for TIL exists that provides a grammar, a rudimentary compiler that does type checking and optimization, and finally a runtime that executes the compiled result. Together, these components make out a general-purpose transformation library for TIL. Using it, we can implement any program analysis and transformations on TIL programs [BKVV05]. The Alert grammar is implemented as a separate grammar module of about 30 lines of SDF code. Compositing this with the basic TIL grammar results in the complete syntax for the TIL+Alert language, c.f. the first step in Figure 8.3. We then use the TIL transformation library to implement a new Alert transformation library. Based on this, we can run meta-programs which perform the semantics of the alert constructs, i.e. the `on` and `pre`/`post` declarations: At compile-time, an abstract syntax tree for TIL+Alert is constructed and the corresponding meta-program for each alert construct is executed. Once all alert constructs in the program have been handled, the base program will have been rewritten. This completes the aspect weaving.

Ideologically, our approach can be considered an example of the "transformations for abstractions"-philosophy described by Visser [Vis05a] – we are effectively extending the open TIL infrastructure with transformations (our meta-programs) that provide new abstractions (the alerts). Next, we will describe the principles behind the implementation of the alert extension, and pay particular attention to the weaving done by the meta programs.

8.4.2 Type Checking

The constructs of the Alert language (`pre`, `post`, `on` and `use`) require their own type checking. To do this, we exploit the construction of the basic TIL type checker. It is a rule set. By adding new type checking rules to this set, we can easily extend its domain (i.e. the ASTs it can process), as we do here for `use`. The following is a Stratego rewrite rule:

```
TypecheckUse: Use(e) -> Use(e){t}
where  <typecheck-exp ; typeof> e => t
```

This rule, named `TypecheckUse`, says that if we are at a `Use` node in the AST with one subnode called `e` (this happens to be an expression), then we reuse the `typecheck-exp` function from the TIL library and annotate the `Use` node with the computed type `t`.

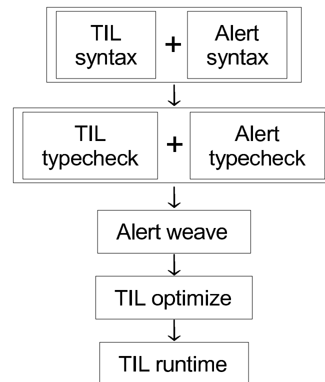


Figure 8.3: Implementation schematics. The *Alert weave* step implements the interpreter for the Alert meta-language and transforms a TIL+Alert program into a valid TIL program.

The ; operator works as function composition. The cases for `pre` and `post` are very similar. For type checking purposes, we define an `on` declaration to be a statement, thus having the `void` type. These few rules implement the "Alert typecheck" box in Figure 8.3.

8.4.3 Alert Weaving

The compilation flow in Figure 8.3 shows that after type checking, the DSAL meta-program parts of a TIL+Alert program are executed, effectuating the weaving. Once weaved, the Alert constructs are gone and the rest of the pipeline will process a pure TIL program. This program is optimised and compiled using unmodified steps of the TIL compiler.

The DSAL notation can be expanded using the simple translation scheme for we DSLs, described in the introduction, i.e. basic macro expansion, but with one crucial difference: whereas the DSL notation is expanded to library calls of a *subject* language library, the DSAL notation is expanded to library calls of a *transformation* language library, and the transformation language is generally different from the subject language. Here, TIL is our subject language and Stratego is our transformation language. Essentially, the DSAL notation is a syntactic abstraction over the Alert transformation library. This notation is embedded in the subject language (TIL), providing a distilled form of meta-programming inside TIL for managing the error handling concern.

When weaving Alert, we have to consider three constructs: the modified function definitions which now have `pre`/`post` conditions, the `on` handler declarations, and function calls. The code for the following cases are all part of the Alert transformation library where they are are implemented as Stratego rewrite rules. When the DSAL

notation is expanded, it results in calls to these rules.

Pre/Post Conditions on Function Definitions Pre/post conditions are easy to process. They are merely markers, or annotations, on the functions. The expression of a pre/post condition can only be activated by an `on`-handler, so the meta-program processing the pre/post conditions has two tasks: first, to store the alert declaration for later use, and second, to remove it from the AST so that we may eventually reach a pure TIL AST. The following rewrite rule, `WeaveFunDef`, does this:

```
WeaveFunDef:
FunDef(x@FunDeclAlert(fd@FunDecl(n, _, _), _), body) ->
  FunDef(fd, body)
where rules( Functions: n -> x )
```

It takes a function definition (a `FunDef` node) that has a subnode which is a pre/post condition (a `FunDeclAlert`) and rewrites the `FunDef` node to a pure TIL `FunDef` by removing the `FunDeclAlert` node. Further, `WeaveFunDef` creates a new, dynamic rule called `Functions` that records a mapping from the name of this function to its complete pre/post alert declaration. A dynamic rule works exactly like a rewrite rule, but can be introduced at runtime, much like closures in functional programming languages. This is done with the `rules` construct. After `WeaveFunDef` has finished, the pre/post condition is removed, and the `Functions` rule can now be used as a mapping function from the name of a TIL+Alert function to its declaration.

In the code above, _ is the wildcard pattern (matches anything) and v@$p(x)$ means bind the variable v to the AST matched by the pattern $p(x)$.

On The processing of `on` itself is also easy. Its node is removed from the AST and we add it to the current set of active `on`-handlers, maintained in the dynamic rule `On`. `On` maps from the name of an alert to the call patterns and handler for it.

```
WeaveOn: On(n, patterns, handler) -> None
where rules(On : n -> (patterns, handler))
```

Function Calls Rewriting function calls to adhere to the new semantics is the crux of the Alert DSAL, and is done by `WeaveFunCall`. This rule implements the following translation scheme. Consider the pattern for functions f in the following form, where f is the function name, f_i are the variable names, t_i are the corresponding types, tr is the return type, and the precondition is as explained earlier:

```
fun f(f0 : t0, ...) : tr
  pre exp alert signal
begin ... end
```

Whenever we see the declaration of an on handler, we need to process the subsequent calls in the same (static) scope, since these may now need to be transformed. We are looking for patterns on the form:

```
on signal in pattern handler;
...
z := f(e0, ...);
```

When we encounter an instance of this pattern, we may need to replace the call to f by some extra logic that performs the precondition check and, if necessary, executes the relevant on-handler according to the following call template[2].

```
z := begin
  var r : tr;
  var a_0 : t_0 := e_0; ...
  if exp then handler
  else r := f(a_0, ...) end
  return r;
end
```

WeaveFunCall will perform the aspect weaving. We will now describe the principles behind it, but not present the full source code, as this is available in the downloadable source code for TIL+Alert (see Section 8.6).

The weaving of WeaveFunCall can only happen at FunCall nodes, i.e. nodes in the TIL+Alert AST that are function calls. Assume WeaveFunCall is applied to a function call of the function f. First, it will check that f signals alerts by consulting the Function dynamic rule that was produced by WeaveFunDef. If indeed f has a declared alert, then the set of active on handlers for the current (static) scope is checked by consulting the On dynamic rule that was initialized by WeaveOn. Multiple on handlers can be active, so another Alert library function is used to resolve which takes precedence (the closest, most specific). Once the appropriate handler is found, the function call to f is rewritten according to the call template shown above, i.e. the FunCall node is replaced by an expression block (an EBlock) which does the precondition check before the call.

Extra care must be taken in the handling of variable names during this rewrite. The precondition expression is formulated in terms of the formal variable names of f, so we cannot insert that subtree unchanged. We must remap the variables, and this is done by a function called remap-vars. As the call template shows, for each formal parameter f_i of f, we create a local variable a_i that is assigned the actual value from

[2]The begin/end block here is called an expression block. It is effectively a closure that must always end in a return. It will be removed by a later translation step that lifts out the variables contained within it, finally giving a valid TIL program.

the call site. We rename the variables in the precondition expression of f, from f_i to a_i, and insert the rewritten expression as exp in the call template.

8.4.4 Coordination

The meta-programs induced by the `on`, `pre` and `post` declarations are dispatched by a high-level strategy that can be likened to an interpreter for the Alert aspect extension. This strategy is implemented as a traversal over the TIL+Alert AST. It contains the logic responsible for translating the Alert notation into calls to the Alert transformation library, and in that capacity, it corresponds to the DSL macro expander. Its execution will coordinate the meta-programs for the various alert constructs. Once the traversal completes, all the Alert-specific nodes will have been excised from the tree, and the result is a woven TIL AST that can be optimized and run.

8.5 Discussion

While DSLs can often be implemented as rather simple macro expanders, the same translation scheme is apparently not applicable for DSALs. The cross-cutting nature of DSALs means that statements or declarations in a DSAL usually have non-local effects. A single line in the DSAL may bring about changes to every other line in the program, and this is not possible to achieve using macro expanders. However, the translation scheme offered by the macro expansion technique is appealing both because of its simplicity and its familiarity; we already have ample experience and tools which may be brought to bear if we could reformulate the DSAL implementation problem to be a DSL implementation problem. This is what our technique offers, by using a program transformation system to implement the library (semantics) for the DSAL notation (syntax). Here, we perform a brief evaluation of our approach.

8.5.1 Program Transformation

Program transformation languages are domain-specific languages for manipulating program trees. Stratego and other transformation language such as TXL [CHHP91] and ASF [vdBHKO02] all have abstract syntax trees as built-in data types, rewrite rules with structural pattern matching to perform tree modification, concrete syntax support and libraries with generic transformation functions. The advantage to using such languages for program transformation is that the transformation programs generally become smaller and more declarative when compared to implementations in general-purpose languages, be they imperative, object-oriented or functional.

High-level Transformations In our experience, when doing experiments with aspect language and aspect weaving, working on high-level program representation

such as the AST is often preferable to lower-level represention traditionally found in compiler-backends. The AST provides all the information from the original source code and is together with a symbol table a convenient and familiar data structure to work with. When working with ASTs, it is important for the transformation language to have good support for both reading and manipulating trees and tree-like data structures.

Generic Tree Traversals Many program transformation languages and functional languages, especially members of the ML family, have linguistic support for pattern matching on trees. We have already seen pattern matching in Stratego in the rewrite rules in Section 8.4. Using recursive functions and pattern matching, tree traversals are relatively simple to express, e.g.:

```
fun visit(Or(e, e)) = ..
  | visit(And(e, e)) = ..
```

In object-oriented (OO) languages, the Visitor pattern is a common idiom for tree traversal, but compared to pattern matching with recursion, it is very verbose. Both techniques perform poorly when the AST changes, however. Introducing a new AST node type requires changes to all recursive visitor functions, or in the OO case to the interface of the Visitor (and thus all classes implementing it). There is, however, an aspect-oriented solution to the cross-cutting-concern part of this problem [?].

Generic programming [LVV03] in functional languages and generic traversals, as offered in Stratego, provide a solution. Generic traversals also allow arbitrary composition of traversal strategies.

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

This defines `bottomup` (post-order traversal) of a transformation `s` as "first, apply `bottomup(s)` recursively to all children of the current node, then apply the transformation `s` to the result". Once defined, this function can be used to succinctly program the variable renaming needed by the `WeaveFunDef` in Section 8.4.3:

```
remap-vars(|varmap) =
  bottomup(try(\ Var(n) -> Var(<lookup> (n, varmap)) \))
```

Syntax Analysis Support Program transformation languages typically come with parsing toolkits and libraries for manipulating existing languages, reducing the effort needed to create a language infrastructure. Also, there is often a tight integration between the parser and the transformation language in transformation systems. Among other things, this allows expressing manipulations of code fragments from the subject language very precisely, using concrete syntax.

Rewriting with Concrete Syntax Another important task is tree manipulation. Rewrite rules provide a concise syntax and semantics for tree rewriting, but rewriting on ASTs can of course be expressed in any language. In program transformation languages, rewriting with concrete syntax, i.e. using code fragments written in the subject language is often provided, and this may improve the readability of rewrite rules considerably, e.g.:

```
Optimize: |[ if 0 then ~e0 else ~e1 end ]| -> |[ ~e1 ]|
```

Here, `~e0` and `~e1` are a meta-variables, i.e. variables in the transformation language (Stratego) and not the subject language (TIL).

Generic Transformation Libraries Libraries for language processing are not unique to program transformation systems, but transformation libraries often contain quite extensive collections of tree traversal and rule set evaluation strategies not found elsewhere. Also, some transformation systems provide generic, reusable functionality for data- and control-flow analysis, as well as basic support for variable renaming and type analysis. However, the libraries of transformation systems are often less complete than that of general purpose languages, when it comes to typical abstract data types.

Maturity and Learning Curve A clear disadvantage of contemporary program transformation systems is their relative immaturity when compared to implementations of mainstream, general-purpose languages. The compilers are usually slower, the development environments are not as advanced, and fewer options for debugging and profiling exist. Further, the same domain abstractions that make domain-specific transformation languages effective to use, also make them more difficult to learn, a tradeoff that must be evaluated when considering the use of a transformation language.

8.5.2 Program Transformation Languages for Aspect Implementation

The stance we take in this paper is that a aspect languages are a form of domain-specific transformation language; they provide convenient abstractions (join points, pointcuts, advice) for performing certain kinds of transformations (aspect weaving—dealing with cross-cutting concerns). They hide the full complexity of program transformation from programmers. Domain-specific aspect languages are even more domain-specific, and hide the complexities of general aspects from their users.

As domain-specific transformation languages, DSALs are conveniently implemented as libraries in a program transformation language. We make this claim based

on our experience with the DSAL = library+notation method from constructing the following systems:

- A domain-specific error-handling aspect language [BDHK06]—a simplified version of this is used as an example in this paper. Our current implementation is for C, and is implemented in the Stratego program transformation language [BKVV06] using the C Transformers framework [BDD06].
- A component and aspect language for adaptation and reuse of Java classes. An early version of this is described in [BBK+05]; it is implemented by translation to AspectJ [KHH+01], using Stratego.
- AspectStratego [KV05]—an aspect-language extension to the Stratego program transformation language; implemented in Stratego itself, by compilation to primitive Stratego code.
- CodeBoost [BKHV03]—a transformation system for C++ that provides *user-defined rules*; an aspect language that allows users to declare library-specific optimization patterns inside the C++ code. The patterns are simple rewrite rules, executed at compile-time. User-defined rules is implemented with the library+notation technique, with the library written in Stratego.

Part of the design goals for many of these experiments was harnessing the expressive power of general program transformation systems into "domain-specific transformation languages" that the programmers of the subject languages could benefit from. In a word, these domain-specific transformation languages are DSALs. For most of our systems, the transformations underlying these extensions, i.e. the implementation of the DSAL semantics, are reusable Stratego libraries, and form the basis for further extensions and experiments.

Experiences One lesson learned from the construction of these DSALs is that good infrastructure for syntax extensions of the subject language is important. Reusing frontends from existing compilers usually precludes extending the syntax, as that would require massive changes to the frontend itself (and for mainstream languages, this is a substantial task). Implementing robust grammars for complicated languages like C++ and Java is infeasible, so language infrastructures provided by program transformation systems were of great help to us. Another lesson is that familiarity with language construction is crucial. Extending a subject language with an arbitrary DSAL may be very complicated, depending on what the DSAL is supposed to achieve. It may therefore be premature to expect regular developers to be able to design their own DSAL language extensions. This is often in more due to the complex semantics of the subject language itself, than the complexity of the DSAL.

8.5.3 Related Work

JTS, the Jakarta Tool Suite [BLS98] is a toolkit for developing domain-specific languages. It consists of *Jak*, a DSL-extension to Java for implementing program transformation, and *Bali*, a tool for composing grammars. Jak allows syntax trees and tree fragments to be written in concrete syntax within a Java program, and provides abstractions for traversal and modification of syntax trees. Bali generates grammar specifications for a lexer and parser and class hierarchies for tree nodes, with constructor, editing and unparsing methods. Bali supports composition of grammars from multiple DSLs. DSL development with JTS is much like what we have described here; an existing language is extended with domain-specific syntax (in Bali), and a small tool is written (in Jak), translating the DSL to the base language.

XAspects [SLS03] is a system for developing DSALs. It provides a plug-in architecture supporting the use of multiple DSALs within the same program. Declarations belonging to each DSAL are marked syntactically, picked up by the XAspects compiler and delivered to the plug-ins. The plug-ins then perform any necessary modification to the visible program interface (declared classes and methods). Bytecode is then generated by the AspectJ compiler; the plug-ins then have an opportunity to perform cross-cutting analysis and generating AspectJ code which is woven by the AspectJ compiler. Thus, implementation of a new DSAL is reduced to creating a plug-in which performs the necessary analyses and generates AspectJ code. Our method, with program transformation, can either complement XAspects, as a way of implementing XAspects plug-ins, or replace it, by developing a libraries for AspectJ manipulation in a program transformation language. The plug-in architecture of XAspects is appealing, as it forces possibly conflicting DSALs to conform to a common framework, making composition of DSALs easier. Both XAspects and our implementation can be seen as library+notation approaches. However, since domain-specific aspects in XAspects can only modify existing code using AspectJ advice and intertype declarations, there are limits to the invasiveness of the DSAL expressed with XAspects. Our implementation strategy has no such constraint since Stratego supports any kind of code modification.

The AspectBench Compiler [AAC+05] provides another open-ended aspect compiler, but is more focused on general aspect languages. It implements the AspectJ language, but is also intended as research platform for experimenting with aspect language extensions generally.

Logic meta programming (LMP) is proposed as a framework for implementing DSALs in [BMV02], because expressing cross-cutting concerns using logic languages is appealing. We believe that our approach could be instantiated with an LMP system as well: the DSAL notation may be desugared into small logic meta-programs which perform the actual weaving. Depending on the logic language, constructing and compositing logic-based transformation libraries may be possible.

In [CBE+00], the authors argue that AOP is a general discipline that should be confine itself in a domain-specific language, but rather be addressed with a general, open framework for composing all kinds of aspects. Such an infrastructure, should it be constructed, would be an interesting compilation target to expand DSAL notation to.

Gray and Roychoudhury [GR04] describe the implementation of a general aspect language for Object Pascal using the DMS program transformation system. They conclude that since transformation systems often provide good and reusable language infrastructure for various subject languages, they are good starting points when developing new aspect extensions. We are of the same opinion, and advocate a disciplined approach where the aspect extensions themselves are implemented as reusable transformation libraries that may in turn be used a substrate for later extensions.

Assman and Ludwig [AL00] describe the implementation of aspect weaving using graph rewrite systems. The authors express the weaving steps in terms of graph rewrite rules, similar to how we describe them as tree rewrite rules. In principle, transformation libraries could be constructed from the sets of graph rewriting rules, but the rule set appears to always be evaluated exhaustively. This makes rule set composition (i.e. library extension) problematic, since two rule sets that are known to terminate may no longer terminate when composed. In Stratego, there is no fixed normalization strategy; the transformation programmer may select one from the library or compose one herself, which in practice adds a very useful degree of flexibility.

8.6 Conclusion

In this paper, we have discussed the *library+notation* method for implementing DSLs: building a library that implements the semantics of the domain, a syntax definition for the desired notation, and a simple translator that expands the notation into library calls. We showed how this method can also be used effectively for implementing DSALs by writing the library part in a program transformation system, expressing the notation as a syntax extension to a subject language, and translating the notation of the DSAL into library calls in the transformation system. This makes the DSAL a meta-program that is executed at compile-time, and that will rewrite the subject program according to the implemented DSAL semantics. Our illustrating examples were based around a small imperative language with an aspect extension for separately declaring error handling policies.

We argued that, based on our experience, program transformation systems are ideal vehicles for implementing such libraries because they themselves come with domain-specific languages and tools for doing language processing, which greatly reduces the burden of implementation when compared to general purpose languages.

The complete implementation of TIL+Alert is available at `www.codeboost.org/alert/til`.

8.7 TIL Grammar

Programs. A program is a list of function definitions, followed by a main program (a list of statements).

```
FunDef* Stat* -> Program
```

Functions. A function definition defines is function with a given signature (`FunDecl`) and body (a list of statements).

```
"fun" Id "(" {Param ","}* ")" ":" Type -> FunDecl
FunDecl "begin" Stat* "end"            -> FunDef
Id ":" Type                            -> Param
```

Statements:

```
"var" Id ";"                                  -> Stat
"var" Id ":" Type ";"                         -> Stat
Id ":=" Exp ";"                               -> Stat
"begin" Stat* "end"                           -> Stat
"if" Exp "then" Stat* "end"                   -> Stat
"if" Exp "then" Stat* "else" Stat* "end"      -> Stat
"while" Exp "do" Stat* "end"                  -> Stat
"for" Id ":=" Exp "to" Exp "do" Stat* "end" -> Stat
Id "(" {Exp ","}* ")" ";"                     -> Stat
"return" Exp ";"                              -> Stat
```

Expressions:

```
"true" | "false"      -> Exp
Id                    -> Exp
Int                   -> Exp
String                -> Exp
Exp Op Exp            -> Exp
"(" Exp ")"           -> Exp
Id "(" {Exp ","}* ")" -> Exp
```

Lexical syntax:

```
[A-Za-z][A-Za-z0-9]*    -> Id
[0-9]+                  -> Int
"\"" StrChar* "\""      -> String
~[\"\\\n] | [\\][\"\\n] -> StrChar
```

There's kind of a drop&drag interface.

– Eelco Visser

9 Interactive Transformation and Editing Environments

Many programmable software transformation systems are based around novel domain-specific languages (DSLs) with a long and successful history of development and deployment. Despite their reasonable maturity and applicability, these systems are often discarded as esoteric research prototypes partly because their languages are frequently based on less familiar programming paradigms such as term and graph rewriting or logic programming, and partly because modern development environments are rarely found for these systems. The basic and expected interactive development aids, such as source code navigation, searching, content completion, real-time syntax highlighting and error checking, are rarely available to developers of transformation code.

This chapter describes Spoofax, an interactive development environment based on Eclipse for developing program analyses and transformations with Stratego/XT. The chapter illustrates how the new language and system abstractions introduced in Part III and Part IV of this dissertation are useful when constructing interactive editing and transformation environments. Spoofax provides, in addition to the aids mentioned above, a code outliner and incremental building of projects. This significantly eases the development of language processing tools using Stratego/XT. Moreover, Spoofax is extensible with scripts written in Stratego that can be executed within Eclipse and allow live analyses and transformations of the code under development.

This chapter is based on the the paper *"Spoofax: An Interactive Development Environment for Program Transformation with Stratego/XT"*, written together with Eelco Visser [KV07b].

9.1 Introduction

Developing and maintaining frameworks and libraries is at the core of any modern software development project and the development of transformation programs is no different. When the code size creeps over a certain limit, it becomes difficult to keep track of, and navigate the source, without reasonable editor support. Unfortunately,

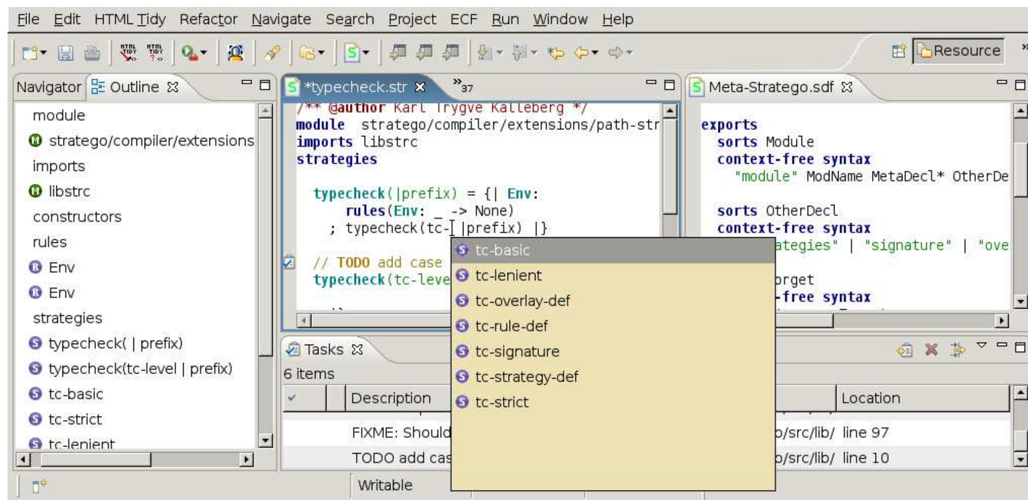


Figure 9.1: Screenshot showing Syntax Definition Formalism (SDF) and Stratego editors with outline view.

most editing tools for domain-specific software transformation languages offer little assistance for editing larger programs. The basic and expected interactive development aids, such as source code navigation, content completion, syntax highlighting and continuous error checking, are rarely available to developers of transformation code.

This lack of development aids keeps the entry barrier for new developers high; DSLs for program transformation use their own syntax and language constructs which may be unfamiliar to many. In addition, most editing environments support these languages rather poorly, providing only limited syntax highlighting. Even skilled developers may be less effective because errors are reported late in the edit-compile-run cycle; that is, only after compiling. It is generally held that errors should be reported immediately after a change has been made while the human programmer is still in a relevant frame of mind. Also, error reporting should ideally be customisable and check project-specific design rules, where possible. This problem also exists with Stratego/XT. Until recently, a good editing environment did not exist for Stratego. This made development with Stratego/XT harder than necessary.

This chapter describes Spoofax, an extensible, interactive environment based on Eclipse for developing program transformation systems with Stratego/XT. Spoofax supports Stratego/XT by providing modern development aids such as customisable syntax highlighting, code outlining, content completion, source code outlining and navigation, automatic and incremental project rebuilders.

Spoofax is a set of Eclipse plugins – a Stratego and an Syntax Definition Formalism (SDF) editor, a help system and the Stratego/J interpreter from Chapter 6.

It supplements Stratego/XT, which must be installed separately, by providing an extensible, interactive development environment. Figure 9.1 shows an example session with an SDF editor (top right), a Stratego editor (top middle), a list of pending tasks extracted from all project files (bottom), and a code outline view (left) displaying all imports, rules and strategies defined in the edited file. The popup is a content completer showing alternatives for the `tc-` prefix.

In this dissertation, the contributions of the Spoofax environment include user extensibility with scripts written in Stratego that allow live analyses and transformations of the code under development; syntax highlighting, navigation and content completion that eases the learning curve for new users of Stratego; and, integration into a mainstream tools platform that is familiar to developers and that runs on most desktop platforms.

9.2 Core Functionality

The Spoofax environment is built around a program model of a Stratego project. This model is called a *build weave* and is discussed below. An important task of the core functionality of Spoofax is to maintain this build weave as users and tools modify the various artifacts that make up the Stratego project including: Stratego files, syntax definitions and build files. Another task of the core functionality is to provide support for user preferences and project-specific settings, and make certain that these are saved across editing sessions in the Eclipse preference store.

9.2.1 Architecture

As depicted in Figure 9.2, Spoofax is divided into a handful of separate components called plugins. Each plugin provides a specific piece of functionality. Some plugins depend on others as indicated in the figure by directed arrows. The full composition of all plugins provides the Spoofax "feature"; that is, an Eclipse-specific term for a set of plugins that together provide a well-defined tool or application.

The editor plugin (`org.spoofax.editor`) provides the interactive editors for Stratego and SDF as well as other interactive capabilities such as configuration menus and various views (shown later). The Stratego/J plugin (`org.spoofax.interpreter`) provides the execution engine for running all the user-provided Stratego scripts. The jsglr parser plugin (`org.spoofax.jsglr`) is used to produce ASTs from the editor buffers. Both the compiled scripts and the ASTs are represented as terms using a slightly modified version of the ATerm [vdBdJKO00] library (`org.spoofax.aterm`). The plugin `org.spoofax.help` provides a manual for Stratego/XT. It may be accessed and read through the Eclipse help system.

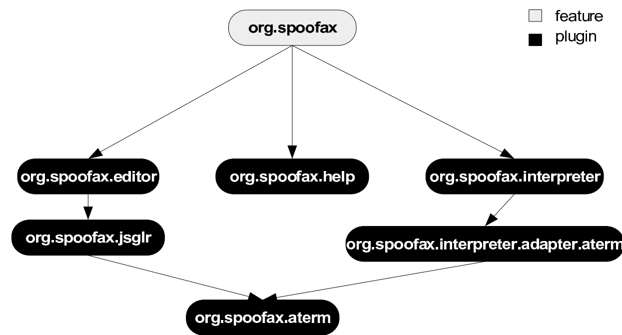


Figure 9.2: Relationship between the plugins that make up Spoofax.

9.2.2 Build Weave

The build weave is essentially a module dependency graph. Each graph node corresponds to a Stratego module. A module, in this case, is either a Stratego source file with the `.str` suffix or a compiled module with the `.rtree` suffix. Each directed edge corresponds to an import declaration in a module. The build weave also contains information about how to resolve import names to actual files on disk using the include paths defined in the project build system. The main purpose of the build weave is to provide searching capabilities to the editor so that the user can search a project for definitions, e.g. find definition locations for a given identifier. It is also used for incremental project building, as discussed below.

The weave is constructed by parsing all `.str` files of a Stratego project and the build system (in the form of `Makefiles`). The build system declares all include paths. There are for resolving module import names to actual files from the file system. Once constructed, the build weave activates logic which listens for events pertaining to the modules or build system files. When a relevant change event is seen, the affected parts of the build weave are marked dirty. These parts will be lazily updated when subsequent requests require updated information. For example, the source code navigation feature of the editor may ask for the list of all visible definitions from a given module. If this module is marked dirty (or depends on nodes marked dirty), then the dirty nodes will be reparsed (possibly adding new nodes and edges in the graph). The weave will be marked as up to date. Only after this update process is finished will the list of visible definitions be computed.

9.2.3 Project Rebuilding

Whenever a module is changed that is referenced by the build system, i.e., it contributes to the final deployable program, the build weave will signal the project builder to commence a rebuild of the project. The current Stratego compiler is a

whole-program compiler. This means that a project rebuild may take several minutes. For convenience, it is possible to turn off automatic project rebuilding.

9.3 Editor

The principal user-interface component provided by Spoofax is the Stratego editor. It is built on top of the Eclipse editor framework and provides a range of editor features from syntax highlighting to source code navigation.

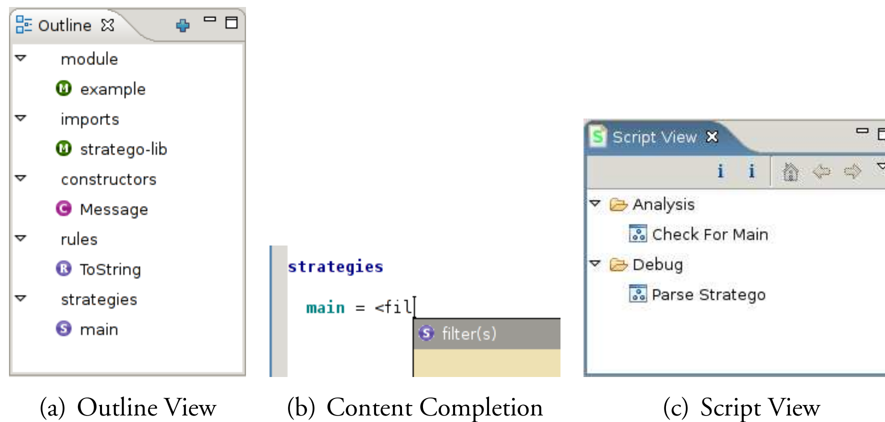
9.3.1 Content Completion

The purpose of content completion is to help the developer writing source code by suggesting possible textual completions based on the surrounding source code context. For example, if a developer asks for a completion for the prefix "`fil`" in a location where a strategy or rule is applicable, then the strategy `filter(s)` may be suggested, provided that the current module imports the standard library where the `filter(s)` strategy was defined. See Figure 9.3(b).

The Spoofax content completer is not perfectly context-aware. For example, it cannot always guess whether a strategy, variable or constructor name is expected. In these cases, it will suggest all possible choices. However, the completer is aware of sections in a Stratego module. If completion is requested in the `imports`-section, only valid module names are suggested. Further, in the `rules` and `strategies` sections, constructor, strategy, rule or overlay names are in general possible. Only closer inspection of the context can determine which is applicable. The Spoofax content completer will automatically suggest overlays and constructors if the prefix before the cursor is a `!` (a build operator), since that uniquely identifies the expression which follows as a term. Similarly, if the immediate prefix before the cursor is a <, the following expression must be a strategy expression, so strategies and rules are the only possible choices.

9.3.2 Syntax Highlighting

Perhaps the most basic development aid expected by an editor is the proper syntax highlighting of source code. Spoofax provides syntax highlighting for both SDF and Stratego. The user can configure the visual attributes, such as slant, boldness and colour, of the different syntactical categories, which includes keywords, rule or strategy declarations, comments, documentation and built-in primitives. Due to the flexible syntax definition formalism used for Stratego, some uncommon corner cases must be dealt with. One of these is the multiple meanings of the character `'`. It may be the start(and end) of a character literal, e.g. `'a'`. It is also a valid suffix of an identifier, e.g. `decl'`. Fortunately, it is possible, though a bit complicated, to



(a) Outline View (b) Content Completion (c) Script View

Figure 9.3: Outline and script views, and a content completion popup.

distinguish between these cases purely at the lexical level. Figure 9.1 shows syntax highlighting for both Stratego and SDF.

9.3.3 Parenthesis Highlighter

The job of the parenthesis highlighter is to find a matching parenthesis for any parenthesis next to the cursor. By convention, most editors (including Spoofax) will first check to see if there is a closing parenthesis before the cursor and, if so, find the first opening parenthesis of the same type at the same nesting level. If there is no closing parenthesis in front of the cursor, an opening parenthesis is looked for immediately after the cursor. This logic accounts for the different allowed parenthesis types in Stratego ({, [, (and <), and will indicate a mismatch by colouring the offending match red. In the case of a good match, a pink outline is used to highlight the matching parenthesis.

```
nasty = id < id + id  ; <id> < <id> + <id ; <id> +> id >(); !("(",")")
```

A noteworthy complication is the meaning of the character <, which is allowed as both a (nested) application operator, e.g., <s1 ; <s0> > and in the choice, e.g. s0 < s1 + s2 as well as part of the left (<+) and right (+>) choice.

9.3.4 Outline

The Outline View, Figure 9.3(a), is a helper view which, when open, always applies to the currently active editor. It shows all definitions found in the module being

edited. The definitions are sorted under their respective types including rules, strategies, module, imports and constructors. The purpose of the Outline View is to allow quick navigation inside larger modules, and to show the structure of a module at a glance. By clicking on any of the defined names, the cursor is moved to the corresponding definition inside the module.

9.3.5 Source Code Navigation

In any non-trivially-sized project, developers spend a lot of their time navigating the source code to find, understand and fix existing definitions or to add new definitions. Cross-module source code navigation is supported by Spoofax due to the build weave discussed in Section 9.2.2. By placing the cursor under an identifier, the "go to definition" action can be invoked. This action will compute the possible definition sites for the identifier requested and, if multiple applicable locations are found, produce a popup window to ask the user to select which definition location to go to. This mechanism allows the user to easily navigate to rule, strategy, overlay and constructor definitions.

A complementary action, "open definition", allows the user to open a popup listing all definitions visible from within a module. This list may be searched by prefix and wildcard strings, e.g. `fi*ter`. It is only populated with definitions which would be visible through the import graph of the current module. A final action, "open module", allows the user to open any module visible from the import graph of the current module.

9.3.6 Build Console

Whenever a project (re)build is started, all output from the build process is redirected to a special Build Console as shown in Figure 9.4. This console keeps the history of all output messages from all previous builds until the user explicitly resets the log.

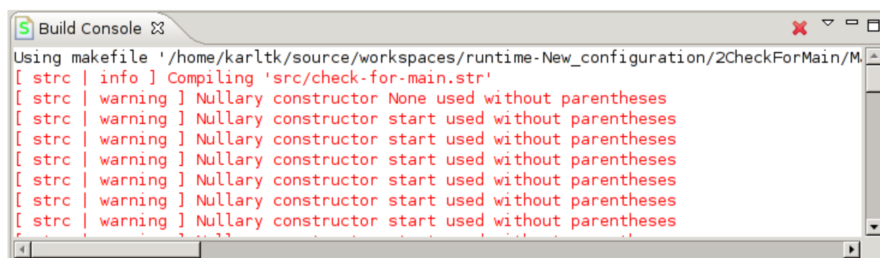


Figure 9.4: The build console shows an example output from the Stratego/XT build system.

9.4 Scripting

Software transformation systems such as Stratego/XT are powerful vehicles for implementing program analyses and transformations. Spoofax offers this power to Stratego programmers in the form of user-supplied editor scripts. By writing a script in Stratego using a few hooks provided by Spoofax, users can extend Spoofax with custom functionality. These scripts run on the Stratego/J execution engine and are used to rewrite ASTs obtained from Stratego source code using the jsglr parser. Scripts may be loaded into Spoofax and executed later by invoking them from a catalogue of scripts found in the scripts view, as discussed below.

Consider the example script in Figure 9.5. Line 4 imports the Stratego syntax and lines 5 and 6 are required for hooking into Spoofax. The strategy `main` on line 15 defines a trivial Stratego program which first obtains the AST for the current editor buffer (line 17), then traverses this AST and collects all definitions of a `main` strategy using the rule from line 22 (here, concrete syntax is used). The resulting list is printed to the script console on line 19. On line 20, a popup is displayed if the list is empty.

9.4.1 Script View

All scripts loaded by the user are visible in the Script View, as shown in Figure 9.3(c), and may be invoked from this view by double-clicking on the script's name. Each script is classified under a category. This aids in organising the script collection. Referring back to Figure 9.5, line 12 declares a user-visible name, which will be shown in the Script View. On line 13, the category is declared. Line 16 declares to the transformlet infrastructure that the strategies `xlet-script-name` and `xlet-script-category` are meta information, not part of the script logic. Additional meta-information definitions, such as author and license, are also possible and will be extracted and placed in a cache. This means that all script definitions need not be reloaded upon every restart of Eclipse for the Script View to be populated.

9.4.2 Script Console

All scripts executing inside Spoofax will have their output redirected to the Script Console. For editor scripts, this console is mainly useful for debugging and simple logging of non-essential information. On line 18 in Figure 9.5, the current term (a [possibly empty] list of strategy definitions) will be printed to the Script Console. This console is visually very similar to the Build Console that was shown in Figure 9.4.

```
1  module check-for-main
2  imports
3    stratego-lib
4    Stratego-Sugar
5    org/spoofax/editor/editor-common
6    org/spoofax/editor/transform
7    org/spoofax/xlet/core
8    org/spoofax/bindings/eclipse/eclipse-ui
9
10 rules
11
12   xlet-script-name = !"Check For Main"
13   xlet-script-category = !"Analysis"
14
15   main =
16      xlet-meta(xlet-script-name ; xlet-script-category)
17    ; spoofax-current-ast
18    ; collect(FindMain)
19    ; where(spoofax-debug)
20    ; try(?[] ; <eclipse-ui-show-popup> ("Malformed", "Missing main"))
21
22   FindMain = ?|[ main = s ]|
```

Figure 9.5: Simple analysis script that checks for the presence of `main` in a module

9.4.3 Analysing and Transforming Source

Analysis and transformation of Stratego code takes place on the AST associated with the current program being edited. In other editors, such as Emacs, code transformation is purely textual, which severely limits the possible transformations and analyses. An AST is obtained from the current editor buffer using the strategy `spoofax-current-ast`. Internally, the source code is parsed using the jsglr parser and, if successful, the AST is returned. A limitation of the current implementation is that the source code must be syntactically correct, since the standard grammar for Stratego does not have any error-correcting productions. The resulting AST can be analysed and modified by the user scripts. Once modification is done, the result can be pretty printed back to the buffer. Layout is not currently preserved well enough for everyday use, but known techniques, such as those explained in [BV00] could be applicable.

9.4.4 Transformation Hooks

In addition to user-initiated execution, scripts can be triggered to automatically execute when certain events in Eclipse are seen. Currently, three events are supported.

OnLoad – Whenever a Stratego file is loaded, a script can execute before the editor displays the file. It is generally recommended, though not required, that the script only performs analysis and the results of this analysis be made visible in the default Eclipse Problems View with other compiler errors and warnings.

OnSave – Scripts can also execute when files are saved. This is frequently useful for (re)generating dependent – or derived – files.

OnTimer – At intervals determined by the script meta information, a script may be executed. These scripts cannot expect any editors to be open, but may query for their existence and should gracefully fall asleep if none are found.

9.5 Implementation

Spoofax is implemented as an Eclipse plugin, written in Java and Stratego. The interfacing between Java and Stratego is handled using the Stratego/J interpreter introduced in Chapter 6. The user-provided extensions are written as transformlets, using the transformlet infrastructure also introduced in Chapter 6.

In the current release, most of the core functionality from Section 9.2, and the editors in Section 9.3, are written in pure Java. The scripting support is mostly written in Stratego (except for the Stratego/J interpreter). Since the editor code is heavily dependent on the libraries and abstractions provided by Eclipse (especially for actions and callbacks using inner classes) it is unlikely that Stratego can provide a worthwhile alternative here. The build weave implementation, however, would clearly benefit from being reimplemented using GraphStratego presented in Chapter 7. The current

implementation predates Stratego/J, and was therefore written in Java. The propagation rules for change events and the graph navigation code are prime candidates for a rewrite.

Parsing – The editor is built on top of three different parsers of Stratego. The ones used for syntax highlighting and code outlining are hand-written in Java because they must work work well for syntactically incorrect programs. A scannerless GLR parser (jsglr) is used to extract the abstract syntax tree from source files and are available for user scripts to inspect. Modification is also possible, but layout is not (yet) always properly preserved. Most contemporary syntax highlighters are written using a mishmash of regular expressions and state-keeping helper code. The Stratego syntax highlighter is, sadly, no exception. It is especially important that the highlighting works well when the program is syntactically incorrect, thereby aiding the programmer when needed the most. The Stratego syntax highlighting is context-dependent, so using a purely declarative tokeniser is not feasible.

Help – The help system is a packaging of the official Stratego/XT reference manual, along with the official Stratego tutorial, and a collection of detailed examples [BKVV05].

9.6 Related Work

Many program transformation systems provide some form of interactive environments. The paragraphs which follow briefly discuss some program transformation systems that are advanced and actively developed.

The *Meta-Environment* is an open and extensible framework for language development, source code analysis and source code transformation based on the ASF+SDF transformation system [vdBvDH+01]. The environment provides interactive visualisations, editors with error checking and syntax highlighting. *Tom* is a software environment for defining transformations in Java [MRV03] and comes with a basic Eclipse editor plugin that provides syntax highlighting, context-specific help, error checking and automatic compilation, but no source navigation. *JTransformer* is a Prolog-based query and transformation engine for Java source code, based on Eclipse. It provides a Prolog editor with syntax highlighting, auto-completion, code outlining, error checking and context-specific help. *HATS* is an integrated development environment for higher-order strategic programming [Win99]. *HOPS* is a graphically interactive program development and program transformation system based on term graphs [Kah99]. The environment is a mix between literal and visual programming. *ANTLRWorks* [BP] is a graphical development environment for developing and debugging ANTLR grammars, with an impressive feature list that includes code navigation, visualisations, error checking and refactoring.

All these systems have feature sets overlapping with Spoofax, but to my knowl-

edge, only the Meta-Environment was also designed to be extensible using a transformation language.

9.7 Summary

This chapter introduced and described an extensible, interactive development environment for Stratego/XT that provides modern development aids like content completion, source code navigation, customisable syntax highlighting, automatic and incremental project building. Users can extend the environment with scripts written in Stratego, and these can perform analysis and transformation on the code under development.

The transformlet techniques introduced in Chapter 6 enabled the extensible scripting features.

Spoofax is still evolving and maturing, but already over a dozen of active Stratego programmers are using it. The feedback so far suggests that the environment lowers the entry level for new users and makes existing developers more productive. Increased productivity comes both from offering source code navigation for Stratego and from the close integration with editors for other (subject) languages that are already available for Eclipse.

– Eventually, you'll be famous enough to ramble about stuff you don't know.
– I do that all the time, but the compiler is my audience...

– Øyvind Kolås replies.

10 Extending Compilers with Transformation and Analysis Scripts

Efficient and robust tool support for domain abstractions is crucial for effective software development, but implementing such domain-aware tools using current language infrastructures is very difficult. Expressing framework and library-specific analyses for design rules, bug pattern finding or protocol checking, as well as transformations for library-specific optimisation, is a goal strived for by implementers of pluggable type systems, defect checkers, code smell detectors and framework migration tools. Domain-specific transformation languages hold the promise that language processing problems may be expressed succinctly and precisely, but this depends on the availability of robust language front-ends that can parse and type check large amounts of source code robustly.

This chapter demonstrates how the general transformation language abstractions introduced in Part III may be applied to building a scriptable analysis and transformation framework. The framework consists of a compiler, a transformation language and a program object model adapter for the abstract syntax tree (AST) of the compiler. The adapter fuses the Eclipse Compiler for Java with the Stratego/J. This enables Stratego scripts to be written which rewrite directly on the compiler AST. The applicability of the system is illustrated with user-definable scripts that perform framework and library-specific analyses and transformations.

The case study found in this chapter is a significantly expanded version of the one found in the paper *"Fusing a Transformation Language with an Open Compiler"* written with Eelco Visser [KV07a].

10.1 Introduction

Stringent use of domain abstractions is key to efficient and maintainable software. The compiler cannot optimise, nor check the correct usage of, domain abstractions because the rules governing the abstractions are part of the application domain and

not the programming language. As a result, domain abstractions are often used incorrectly or inefficiently. Various techniques have been devised to combat this, such as typestate analysis [SY86, SY93], pluggable type systems [ANMM06], code smell detectors, defect analysers [Cop05] and other static analysis tools. For the most part, development and maintenance of these tools is so costly that their construction can only be afforded for the most used domains. For example, common static analysis tools for Java support only the standard library [HP04] and Enterprise JavaBeans [CNFP06]. The situation is similar for domain-specific optimisation: high-performance compilers may come with extensions and directives which improve performance for certain, general numerical computation problems [CDK+01]. Other domains receive little or no support.

The state-of-the-art is that existing analyses and optimisations only serve a very restricted set of domains and the needs of most other projects and frameworks are largely left unattended. The absence of solid and adaptable tools has led to the proliferation of ad-hoc techniques that are often brittle and text-based [DR97]. Fortunately, it has become more common to expose at least some API to the compiler internals in the recent years, in particular to the abstract syntax tree. This presents a significant opportunity for providing good domain support for a much wider range of domains. It is now possible to leverage the maturity and robustness of the parsers and type analysers available in mainstream compilers. Previously, such infrastructure could only be reused from selected, open research compilers [WFW+94, TCIK00, NCM03].

The framework presented in this chapter takes advantage of the recent opening of mainstream compilers. It is a fusion between the Stratego rewriting language and the Eclipse Compiler for Java (ECJ) based on the program object model adapter technique described in Chapter 4.

The composed system provides a powerful framework that allows framework developers to implement domain-specific transformations and analysis for Java frameworks in Stratego. Developers may take advantage of pattern matching, rewrite rules, generic tree and graph traversals as well as a reusable library of generic transformation strategies and data-flow analyses.

The contributions of this chapter include:

- Bringing the analysis and transformation capabilities of modern compiler infrastructure into the hands of advanced developers via a convenient and mature program transformation language.

- Making program transformation tools and techniques practical and reusable for framework developers by integrating directly with stable tools like the Java compiler. This lowers the entry barrier for developers wanting to write library-specific program analyses and transformations.

- A discussion of the design and implementation of a prototype tool for domain-specific analysis and transformation.
- A validation of its applicability through a series of examples taken from mature and well-designed applications and frameworks.

The remainder of this chapter is organised as follows: Section 10.2 motivates the need for domain-specific analysis and transformation and why scripting these with a language-independent transformation system is useful. Section 10.3 shows the practical applicability of the prototype on a series of commonly encountered framework-specific analysis and transformation problems. Section 10.4 discusses implementation details of the prototype. Section 10.5 covers related work. Section 10.6 discusses some tradeoffs related to the technique.

10.2 Scriptable Domain-Specific Analysis and Transformation

The main motivation for extending compilers with scripts is the lack of domain knowledge possessed by traditional compilers. This domain ignorance bars compilers from providing detailed errors and warnings about the usage of domain abstractions and from automatically optimising the usage of domain abstractions. For example, the compiler will not warn if a function is written so that a file may be read before it is opened nor will it remove a call to `close()` on a file that is known to be closed. This is reasonable, given that the semantics of library objects is, in general, not known to the compiler writer.

A compounding problem is the lack of any general facility of adding such knowledge by the user. As a consequence of the closedness of compilers, many domain-specific language processing tools are constructed from scratch, duplicating substantial parts of compiler infrastructure that could and should have been reused. Because implementing and maintaining robust language infrastructures for mainstream languages is very laborious, many stand-alone language processors are often brittle or incomplete. This is clearly an unfortunate situation. Finding good approaches to compiler infrastructure reuse of is a worthwhile topic of study, and relates closely to the topic of language-independent transformations explored in this dissertation.

All compilers have an internal program object model. For most compilers, the abstract syntax tree (AST) forms the core of this model and is supplemented with additional data structures (such as a symbol table) and functionality (such as type analysis, code searching, and pretty-printing). Some modern compilers, such as the Sun Java Compiler, expose AST programming interfaces that allow developers to implement custom language processing tools based on the compiler. Another example is the Eclipse compiler for Java, which has APIs that are used to implement the

interactive source code refactorings in the Eclipse Java development environment. Compared to implementations with general purpose languages like Java, implementing language processing with transformation languages usually results in smaller and more readable programs, which are quicker to change and maintain. The downside is that their infrastructure for processing mainstream subject language code is generally not as robust, optimised nor up to date as that available in mainstream compilers. This is not likely to change. The massive user base served by mainstream compiler serves to weed out bugs and easily justifies investments for hand-optimising core parts of the infrastructure.

Recent research and industry practise is rife with examples where domain-specific transformations and analyses play an important role, such as framework-specific refactoring, optimisation of library abstractions [GL00, Kal03], advanced style and type checking [ANMM06] and framework-specific code smells [vEM02]. These domain-specific language processing problems are different from their general counterparts in at least four ways.

1. They are often less performance-critical because they only apply to small amounts of code and they can be targeted and applied only to the relevant parts of the code, whereas the general compiler analyses and transformations are applied exhaustively.

2. They may have a much higher degree of variability. The conditions they check for, the locations they should be applied to and the point in the software life cycle they should be applied vary even between individual projects using the same domain abstractions.

3. They have a much higher rate of change. Whenever the framework changes or is rearchitected, the domain-specific analyses and transformations must follow suit.

4. They occur across programming languages, and any one system may involve the combination of many languages which have clear project-specific rules for how they should interoperate.

Being able to attack this problem with a high-level, language-general transformation system coupled with stable and robust language processing foundations is appealing because the combination is likely to provide an effective and expressive platform.

10.2.1 Architecture

The Java transformation framework proposed in this chapter is available as a standalone command-line application and as a reusable Eclipse plugin.

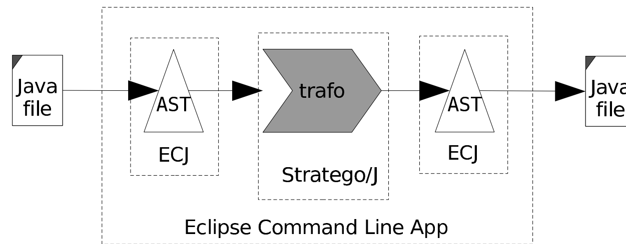


Figure 10.1: Command-line based transformation.

In stand-alone mode, as shown in Figure 10.1, the system performs fully automatic source-to-source transformation. The user supplies the path of a project and a script to execute, or the path to a single source file and a transformation script. The script may use a file API to traverse the project directories and to parse source files to obtain their AST. After rewriting, the script may use the file API to write modified ASTs back to disk in the form of formatted (pretty-printed) source code.

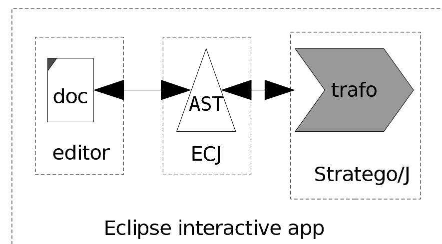


Figure 10.2: Interactive transformation.

In plugin mode, as shown in Figure 10.2, interpreter objects may be instantiated with arbitrary scripts. These objects may be handed individual ASTs obtained from live documents. Strategies may be invoked via the interpreter objects to perform AST rewrites. The editing framework provides logic for synchronising the text with the modified AST. This allows scripts to be used for very fine-grained source code queries and transformations on the source code alongside the programmer's manual editing of the source code.

10.2.2 MetaStratego as a Scripting Engine

The combination of the Stratego/J runtime and the transformlet lightweight component system is very handy for deploying interactive transformation scripts. The framework supports the loading and execution of transformation scripts in the form of transformlets. As each transformlet is loaded, it registers new actions with the framework. These actions become available through a separate menu in the user interface. The programmer may invoke the script actions from this menu. The

transformlets may be implemented with any of the language extensions introduced in previous chapters of this dissertation.

10.3 Examples of Domain Support Scripting

This section describes analyses and transformations that are specific to a given project or framework. Some analyses check that domain abstractions provided by the framework or library are used properly. They apply to the clients of the libraries, but not to the library code itself. Other analyses check for a consistent realisation of the domain abstractions. These apply to the implementation of the library. Additional examples will demonstrate how the results of these analyses may be used to perform source code transformations.

The examples have been chosen to demonstrate how the (extended) Stratego language can handle syntax-, type- and flow-based analyses, and what an advanced framework developer with a good working knowledge of language processing and Stratego could implement. However, Stratego is capable of performing significantly more advanced analyses and transformation than shown here. See [OV05, BKVV06, Kal03] for some examples. It is also important to note that with a program object model (POM) adapter for another front-end, for example a C or C++ compiler, the same techniques are directly applicable to libraries written in C or C++.

10.3.1 Project-Specific Code Style Checking

Software projects of non-trivial size always adopt some form of (moderately) consistent code style to aid maintenance and readability. Maintainers of such systems may be concerned with checking for proper implementation and proper use of domain abstractions. Consistency of implementation can be improved by encouraging systematic use of particular idioms. The following idioms are taken from the internal AST implementation of the Eclipse Compiler for Java and from the graphical user interface library Standard Widget Toolkit (SWT)[1].

Equality Test Idiom. A common idiom when implementing `equals()` methods on objects in Java is to start with an `instanceof` check. The internal AST classes of the Eclipse Java compiler follow this idiom. For the `CompilationUnit` class, the idiom looks like this:

```
public boolean equals(Object obj) {
  if(!(obj instanceof CompilationUnit))
    return false;
  ...
```

[1] Part of Eclipse.

```
}
```

The idiom is simple to implement, but consistently applying it requires a degree of fastidiousness best left to the computer. It is easy to miss a spot when performing maintenance, and new maintainers to an existing code base may not be aware of the idiom. The pattern matching capabilities of Stratego can be used to verify that the code for `equals()` is of the correct form:

```
1  check-equals-method =
2    ?MethodDeclaration(_,_,SimpleName("equals"),object(),_,Block(stmts))
3   ; where(not(<Hd> stmts ; ?IfStatement(e, _, _) ; <check-expr> e)
4         ; emit-warn(|"equals() does not start with instanceof check"))
5
6  check-expr =
7  ?PrefixExpression(
8       PrefixExpressionOperator("!")
9     , ParenthesizedExpression(InstanceofExpression(_, _)))
```

The `check-equals-method` strategy should be applied to the method declaration of an `equals()` method. It starts with a pattern match (on line 2) to verify this. In the process, it binds the variable `stmts` to the list of statements of the method body of `equals()`. The subterm `object` is an overlay that matches an argument list of in parameter of type `Object`. The first statement of the body is retrieved with `Hd` (on line 3) and matched against a pattern for `if`. On a successful match, the condition of the `if` is checked with `check-expr` which will only match expressions on the form `!(_ instanceof _)`, where `_` is any expression. Should the `if` be omitted, or not follow the required idiom, `emit-warn` (on line 4) will display a warning. When run in command-line mode, this warning will be displayed on the console. In interactive mode, it can appear in a window containing compile-time warnings. The code above is purely syntactical. It does not know anything about the Java type context in which it is applied. It would be reasonable to add additional context information so that the `instanceof` check was verified to check for an appropriate type, i.e. the type in which the `equals()` method is defined. To improve analysis accuracy and the expressive power, type checking can be employed, as shown next.

Field Type Restriction Idiom. The choice between `LinkedList`, `ArrayLists` or primitive arrays is often a source of contention. Once the selection has finally been made for a particular group of classes, it serves to be consistent. The internal AST of ECJ uses the primitive array type pervasively, e.g.:

```
class TypeDeclaration ... {
  ...
  public FieldDeclaration[] fields;
```

```
  ...
}
```

The following strategy can be applied to a type declaration and will verify that none of the fields are of the type `forbidden-type`, or any subtype thereof:

```
check-type-decl(|forbidden-type) =
   ?TypeDeclaration(_, _, _, _, _, body)
  ; where(<map(try(check-field(|forbidden-type)))> body)

check-field(|forbidden-type) =
    ?FieldDeclaration(_, field-tp, _)
  ; <type-of ; is-subtype-of(|forbidden-type)> field-tp
  ; emit-warn(|"Field is of illegal type!")
```

The strategy `check-type-decl` should be applied to a type declaration term, and will use `check-field` to iterate over its fields. The strategies `type-of` and `is-subtype-of` are used to retrieve the type of each field and test if these are subtypes of `forbidden-type`. Applying `check-type-decl(|"java.util.List")` to a type declaration with, say, a `List` field results in a warning.

Next, an example shows how generic traversals can control which AST nodes a strategy should apply to.

Context-Specific Visibility Idiom. Decomposing abstractions into namespaces may pose significant challenges for the visibility mechanism of the namespace system when constructing large libraries. Consider the placement of classes into Java packages for the graphical widgets found in SWT. Each graphical element, such as a button, text area, or check box, is encapsulated in its own subclass of `Widget`. Most widget classes in SWT are not intended to be subclassed by the user. However, for implementation purposes, the `final` keyword was not used and the subclassing prohibition is only mentioned in the source code documentation of the individual classes.

```
check-illegal-swt-subclass =
    ?TypeDeclaration(_, _, _, _, _, _)
  ; type-of ; supertype-of ; dotted-name-of ⇒ stp
  ; <list-contains(?stp)> restricted-swt-types
  ; emit-warn(|"Illegal subclass of org.eclipse.swt.widgets.Widget!")
```

This code snippet will check that a given type declaration is not a subtype of any of the “inheritance restricted” SWT classes. The list of these widgets is kept in the (global) variable `restricted-swt-widgets`. The strategy should be applied a type declaration term. If the initial pattern match succeeds, the dotted name (for example, `"java.lang.String"`) of the super type of this type declaration is computed and stored in `stp`. If the super type is contained in the `restricted-swt-types` list, the current type

declaration inherits a subclass-restricted widget and the warning at the bottom line is emitted. Arguably, with a sufficiently complicated regular expression, violations of the inheritance restriction may often be found using purely text-based approaches. Unfortunately, should programmers insert comments at unexpected places, this approach is sure to break.

Warnings will be emitted if the strategy is applied to the code of the SWT widget library. The strategy should therefore only be applied to code *using* SWT. It can be applied easily to a Java project using the following a two-level generic traversal scheme:

```
analyse-package(|project) =
  dir-topdown(parse-and-resolve(|project)
          ; topdown(try(check-illegal-swt-subclass)))
```

At the outer level of the traversal, `dir-topdown` will recurse through a directory structure and call `parse-and-resolve` to construct the corresponding compilation unit (AST) for each `.java` file. Once constructed, `topdown` will recurse over it, in pre-order, and apply the `check-illegal-swt-subclass` strategy to all terms. This ensures that all contained top-level, local, anonymous and inner type declarations inside each compilation unit are visited and checked. Multiple checks can easily be composed into one pass using the `topdown` generic traversal:

```
topdown(try(check-illegal-swt-subclass) ; try(check-equals-method))
```

Bounds Checking Idiom. Consider the following code for iterating over `x`:

```
for(int i = 0; i < x.size(); i++) { ... }
```

If `x` is a value object of type `T`, i.e. happens to be immutable, then the `size()` method will be invoked needlessly for every iteration. The JIT may possibly inline this call but only if the code is executed frequently enough. One might like to encourage a coding style that is also efficient with the bytecode interpreter:

```
{ final sz = x.size(); for(int i = 0; i < sz; i++) { ... } }
```

This idiom is used throughout the implementation of the internal AST classes of ECJ and may be checked for using the following strategy:

```
check-for =
    ?ForStatement(_, e, _, _)
  ; <topdown(try(call-to-immutable))> e

call-to-immutable =
    ?MethodInvocation(_, _, _, _, _, [])
  ; binding-of ⇒ MethodBinding(class-name, _, _, _)
  ; <list-contains(?class-name)> immutable-classes
```

```
  ; emit-warn(|"Call to method on immutable object in loop iteration")
```

The strategy `check-for` should be applied to a `for`-statement. If any of the condition expressions are calls to methods without parameters of objects of an immutable type, a warning is emitted. The list of known non-mutating methods is given in the list `immutable-classes`.

Using data-flow analysis, method calls on objects which are not immutable could also be considered. As long as the body of the `for`-loop does not invoke any mutating operation and does not pass `x` as an argument to another method, immutability can be assumed. By keeping (*typename*,*methodname*) pairs in an `immutable-methods` list, the immutability property can be looked up, much like the subclass restriction property was looked up.

The field type restriction and the bounds checking idioms show how analyses requiring type information can be expressed. The type analysis functionality is provided by ECJ, and made available to scripts through the strategies `subtype-of`, `supertype-of`, `is-subtype-of`. These strategies connect to the compiler via the foreign function interface introduced in Chapter 4.

10.3.2 Custom Data-Flow Analysis

Totem propagation is a kind of data-flow analysis where variables in the source code are marked with annotations called totems [Kal03]. These assert properties on the variables which are later used by other analyses and transformations. A meta-program will perform data-flow analysis and propagate the asserted totems throughout the code, following the same principles as constant propagation.

Totem propagation is in many ways similar to typestate analysis, which is "a data-flow analysis for verifying the operations performed on variables obey the typestate rules of the language" [SY93]. Typestate analysis is mostly concerned with verifying protocols such as ensuring that files are opened before they are read. Totem propagation uses the same data-flow machinery to discover opportunities for optimising away unnecessary calls (such as a call to `sort()` on a sorted list) or replacing costly operations with cheaper ones (such as binary search instead of linear search on sorted lists). Meta-programs performing these forms of data-flow analyses must be aware of the propagation rules for each kind of totem.

A totem propagator could be useful for removing dynamic boundary checks in a library for matrix computations. Consider the following matrix interface defined in the Matrix Toolkits for Java (MTJ) library [mtj06]:

```
public interface Matrix {
  public Matrix add(Matrix B, Matrix C);
  public Matrix mult(Matrix B, Matrix C);
  public Matrix transpose();
```

```
  ...
}
```

These operations have certain well-defined requirements. Two matrices, A and B, may only be added if they have the same dimensions, i.e. A has same number of rows and columns as B. Two matrices, A and B, may be multiplied and placed into C if the number of columns of A equals the number of rows of B. The dimensions of C must be equal to the number of rows of A and the number of columns of B. Transposition of a matrix swaps the row and column dimensions. These rules are violated by the following code:

```
Matrix m = new DenseMatrix(5,4);
Matrix n = new DenseMatrix(4,6), z = new DenseMatrix(5,6),
      w = new DenseMatrix(3,5);
m.mult(n,z); z.transpose(); z.mult(m,w); // m and w incompatible
```

In the above example, all matrix dimensions are compatible with respect to the first two operations but not for the final expression `z.mult(m,w)`. The matrix operations in MTJ will verify dimensions before calculating and throw exceptions if the preconditions are not met. Performance-wise, this is costly and latent mismatches may lurk in seldom used code.

To alleviate this problem a totem propagator may be applied which knows how to propagate and verify the dimension of matrix operations. Initial dimensions can be picked up from programmer-supplied assertions (in the form of an assert statement, `assert Matrix.dimensions(m,4,3)`) or from the variable initialisation. Whenever a dimension is asserted for a variable in the code, a new, dynamic rule `Dimensions:` *name* -> *dim* is created that remembers the asserted dimensions *dim* for a variable *name*. If an existing rule for the variable already exists, it is updated. This rule can then be applied (and updated) when propagating the dimension totem across a transposition:

```
PropTotem =
   ?MethodInvocation(src, SimpleName("transpose"), _, [])
 ; <type-of ; dotted-name-of> src ⇒ "no.uib.cipr.matrix.Matrix"
 ; <Dimensions> src ⇒ [rows, columns]
 ; rules(Dimensions : src → [columns, rows])
```

Here, the old dimensions (if they are known) will be swapped and the `Dimensions` rule updated. There are other (overloaded) `PropTotem` rules which deal with addition and multiplication. The propagator core is based on the general constant propagation framework proposed by Olmos and Visser [OV05] but is adapted to allow propagating arbitrary data properties and not just constants:

```
prop-totem =
    PropTotem
```

```
  <+ prop-totem-vardecl
  <+ prop-totem-assign
  ...
  <+ all(prop-totem)
```

The `prop-totem` strategy should be applied to a method body where it will recurse through the subterms. At each term, a series of strategies is tried in order. If all fail, the recursion continues into the children of the current term. The first strategy applied is `PropTotem`. This is a set of overloaded rules for the `add`, `mult` and `transpose` cases. The rule with the matching pattern will be applied. If none of the rules succeed, the current term is not method call to `add`, `mult` or `transpose`. In this case, the `prop-totem` strategy continues by calling the `prop-totem-vardecl` strategy. This will try to infer totems from variable declaration terms. If the current term is an assignment ($v = e$), the totem of e is inherited by v. This is handled by `prop-totem-assign`. Additional cases deal with control flow constructs like `if` and `while`, as described in [OV05]. These extra cases may retroactively be added to the algorithm using the aspect mechanism described in Chapter 5.

Once the correctness of the dimensions can be guaranteed, based on the user assertions and propagation, the runtime dimension checks can be removed by source code transformation.

10.3.3 Domain-specific Source Code Transformations

Results of analyses can be used to perform source code transformations either as part of the compilation process or as refactorings on the source code. Such code transformations could aid in framework migration and may perform pervasive style changes or remove code smells.

Optimising Matrix Dimension Checks Using totem propagation described previously, matrix operations can be rewritten to remove runtime dimension checks, provided that the matrix dimensions can be determined statically (at compile time) to be correct. In that case, the following substitution is applicable:

```
A.mult(B,C) -> A.uncheckedMult(B,C)
```

The following rewrite rule captures the necessary conditions and can be plugged directly into the totem propagator to achieve a correct substitution:

```
PropTotem:
    MethodInvocation(src1, SimpleName("mult"), x, [src2, dst])
 → MethodInvocation(src1, SimpleName("uncheckedMult"), x, [src2, dst])
 where
   <type-of ; name-of> dst ⇒ "no.uib.cipr.matrix.Matrix"
 ; <Dimensions> src1 ⇒ (s1r, s1c) ; <Dimensions> src2 ⇒ (s2r, s2c)
```

```
  ; <Dimensions> dst ⇒ (dr, dc); !s1c ⇒ s2r; !s2c ⇒ dc; !s1r ⇒ dr
```

The `where` clause is a rewriting condition which ensures that the `mult` call is on the correct data type and that the dimensions are compatible.

Optimising Loop Boundary Checks The bounds checking idiom from the previous section can also be turned into a code transformation:

```
OptimizeFor:
   ForStatement(init, cond, incr, body)
→ Block(<concat> [vdecls, [ ForStatement(init, cond', incr, body) ]])
where
    <collect(is-immutable-call) ; new-names> cond ⇒ call-var-pairs
  ; <map(\(e, v) → vardecl(<type-of> e, v, e)\)> call-var-pairs
      ⇒ vdecls
  ; <bottomup(try(RewriteImmutable(|vars)))> cond ⇒ cond'
```

The generic `collect` strategy is used with `is-immutable` to find all invocation of get-like methods in the condition expression. For each expression, a new uniquely named variable is created (by `new-names`) and a variable declaration for it is created that gets added before the `for` loop. Each expression is replaced with its corresponding, freshly named, temporary variable using the `RewriteImmutable` strategy. This avoids name capture in the generated code.

10.4 Implementation

The analysis framework presented in this chapter reuses the Eclipse Compiler for Java via a POM adapter. The compiler is available as a plugin for the Eclipse development platform [Ecl]. Although most users encounter Eclipse as a graphical application, it is possible to create so-called headless applications using the Eclipse infrastructure that have no graphical interface. The command line application depicted in Figure 10.1 is an example of a headless application.

10.4.1 Analysis Architecture

The principal components of the transformation framework are shown in Figure 10.3 The plugin `org.spoofax.eclipsetrafo` contains both the application which can be invoked from the command-line and a plugin class which may be used as part of a graphical application.

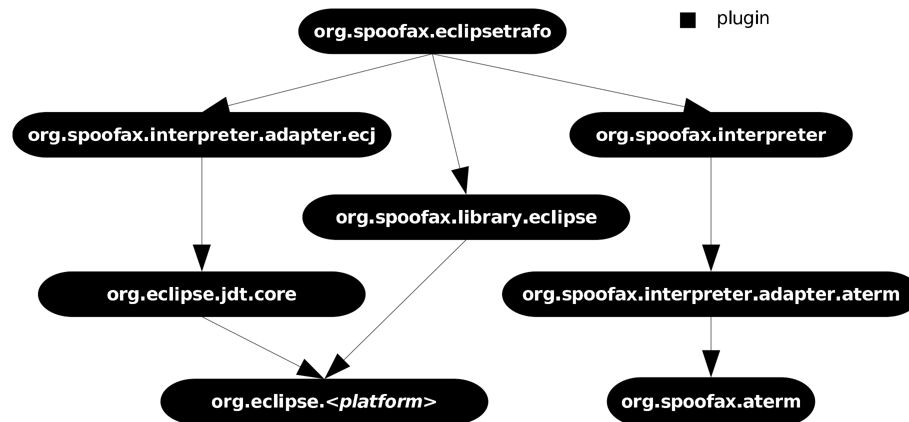


Figure 10.3: Principal components of the analysis and transformation framwork introduced in this chapter. The org.eclipse.jdt.core provides the Java compiler, and the org.eclipse.<platform> represents the collection of plugins that comprise the Eclipse runtime and necessary support plugins for the Java compiler.

10.4.2 Transformlet Repositories

The analysis and transformation scripts shown in the previous section can be compiled into standalone transformlets. This facilitates sharing of analyses and transformations between developers. Since transformlets are fully self-contained, it is easy to upload them into online repositories and distribute them to other users. However, there is currently no easy-to-use graphical interface for subscribing to repositories and downloading transformlets, but simple command-line tools are available.

10.5 Related work

Programmable static analysis tools, such as CodeQuest [HVdMdV05], PQL [MLL05], and CodeSurfer [AT01], all support writing various kinds of flow- and/or context-sensitive program analyses, in addition to (sometimes limited) queries on the AST. Pluggable type systems, an implementation of which is described by Andreae et al [ANMM06], also offer static analysis capabilities. Developers can express custom type checking rules on the AST that are executed at compile-time so as to extend the compiler type checking. Programmable analysis frameworks like PMD [Cop05] provides a good collection of standard analyses that are specific to a given programming language, in this case Java, and where the user can implement additional ones using an analysis API. Neither programmable static analysis tools nor pluggable type systems support source code transformations, however.

Languages for refactoring such as JunGL [VEdM06] and ConTraCT [KK04] provide both program analysis and rewriting capabilities. JunGL is hybrid between an

ML-like language (for rewriting) and Datalog (for data-flow queries) whereas ConTraCT is based on Prolog. JunGL supports rewriting on both trees and graphs, but is a young language and does not (yet) support user-defined data types. Stratego is a comparatively mature program transformation language with sizable libraries and built-in language constructs for data- and control-flow analysis, handling scoping and variable bindings, and pattern matching with concrete syntax (not demonstrated in this chapter). It comes with both a compiler and interpreter and has been applied to processing various other mainstream languages such as C and C++ [BDD06].

Open compilers, such as the SUIF [WFW+94] project, Polyglot [NCM03], OpenJava [TCIK00], and OpenC++ [Chi95], offer extensible language processing platforms. In many open compilers, the entire compiler pipeline, including the backend, is extensible. Constructing and maintaining such an open architecture is an arduous task. As demonstrated in this chapter, many interesting classes of domain-specific analyses and transformations require only the front-end to be open. Exposing just the front-end is less demanding than maintaining a fully open compiler pipeline. Transformation systems may be plugged into either of these open compilers using the POM adapter technique.

The research on active libraries [VG98] has largely focused on performance optimisation for example using pre-processors and library annotations [GL00, Kal03]. Compiler scripts are useful for exploring other topics of library design such implementation consistency, contract checking and sensible idiom or pattern usage.

10.6 Discussion

Program analysis tools have a long history, preceding even the venerable `lint` [Joh78]. Recent research has to a certain extent focused on scriptable frameworks for expressing extensible analysis tools. Scripts allow developers to adapt, for example, pluggable type systems, style checkers and static analysis to their specific frameworks or libraries.

The appealing feature of the system described in this chapter, and that of JunGL and ConTraCT, is that in addition to scripting syntax-, type- and flow-based analyses, it also allows scripting of source code transformations based on the analysis results. Transformation languages are good candidates for compiler scripting languages. New transformations and analyses may be expressed quickly due to their high-level domain-specific language constructs. This makes them an appealing part of a testbed for prototyping language extensions as well as new compiler analyses and optimisations.

The plethora of custom analysis and transformation tools suggests that compiler writers should cater for potential extenders in their infrastructure design. As this chapter shows, even the rather simple and minimal inspection interface of the POM adapter is sufficient for expressing powerful program analyses. General code transfor-

mation can be scripted if functionality for building AST nodes is also exposed by the compiler.

A limitation of ECJ, and of many other compilers, is that rewriting the AST will invalidate the type information. Type analysis is often done during or just after parsing. The typical usage scenarios for ASTs inside compilers do not make incremental reanalysis of types necessary. Incremental reanalysis would be very useful for POM clients that perform rewriting. Without such support, complete type reanalysis must be performed to restore accurate type information. This often involves unparsing and subsequent reparsing of text.

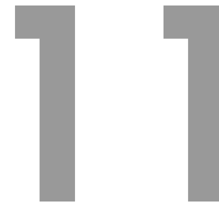
Stratego does not have any fundamental limitations on the types of analyses and transformations it can express. The language is Turing-complete, and can express both imperative and functional algorithms for program analysis and transformation. Special support exists, in the form of reusable strategy libraries and language constructs such as dynamic rules, for performing control- and data-flow analysis over subject programs represented as terms, i.e. abstract syntax trees. Refer to [OV05] for more details on these features. In practise, the current performance of the interpreter may be a limiting factor for particularly resource-intensive analyses and transformations. In these cases, the C-based Stratego/XT infrastructure [BKVV06] may be an alternative. Certain whole-program analyses may require very efficient implementations of specific data structures, such as binary decision diagrams (BDDs). Stratego does not currently have a library providing BDDs.

10.7 Summary

This chapter presented a powerful framework for scripting domain-specific analyses and transformations for Java based on the Stratego rewriting language and the Eclipse Compiler for Java. The examples, all taken from what is considered to be mature and well-designed frameworks, illustrate the usefulness of the domain-specific abstractions for program analysis and transformation provided by the MetaStratego system. The framework was made possible in particular by the program object model adapter technique (described in Chapter 4) which enabled the quick and large scale reuse of the Eclipse compiler front-end. The fusion between the Stratego/J runtime and the ECJ compiler is very efficient: it can apply the bounds checking idiom to around 2.7 million lines of Java code in just over four minutes on a low-end laptop.

I don't think you can break it.

– Martin Bravenboer



Code Generation for Axiom-based Unit Testing

This chapter presents a case-study of how the abstractions and infrastructure discussed in Part III and Part IV of this dissertation may be applied to source code analysis and interactive code generation. The motivation for the case study prototype, an interactive unit test generator, is that current unit testing methodologies often result in poor test coverage due to rather ad-hoc approaches.

Developers are encouraged to always write test cases, often as a driving force for new software features, or for systematic regression testing to avoid reintroducing known errors during software maintenance. This easily results in a development process focused around the individual test cases, rather than addressing the general requirements the cases are intended to represent. By expressing expected behaviours as axioms written in idiomatic Java code, it may be possible to improve the quality of the test code. Each axiom captures a design intent of the software as a general, machine checkable rule, rather than an individual case, and may be used to generate unit tests.

The quick and easy construction and integration of the test generator into an interactive development environment was made possible mainly due to the POM adapter technique described in Chapter 4 and the transformation runtime from Chapter 6. This chapter illustrates the applicability of the abstractions proposed in this dissertation, and, to a lesser degree, the workings of the generator tool. However, a detailed motivation and a discussion of the principal elements of the underlying test methodology are necessary before a detailed account of the transformation tool can be provided.

The results in this chapter were obtained in collaboration with Magne Haveraaen.

11.1 Introduction

Testing has gained importance over the past decade. Agile methods see testing as essential in software development and maintenance. Testing is given many different roles, from being the driving force of software development, to the somewhat more modest role of regression testing.

The most influential of these is probably Beck's extreme programming (XP) method [Bec98], giving rise to the mind-set of test-driven development (TDD) [Bec02]. TDD assumes that every software unit comes with a collection of tests. A new feature is added by first defining a test which exposes the lack of the feature. Then the software is modified such that all (both old and new) tests are satisfied. The process of extending software with a new feature can be summed up in five steps: (1) Add a test for a new feature which is written against the desired future API of the feature. (2) Run all tests and see the new one fail, but making certain that the unrelated features remain correct. (3) Write some code implementing the new feature, often just focusing on the minimal logic necessary to satisfy the tests written previously. (4) Rerun the tests and see them succeed. (5) Refactor code to improve the design.

A problem with a strict TDD approach is that each test is often casewise, i.e., it only tests one case in the data domain of a feature. While this opens up for implementing the logic of a new feature in small steps – or to insert dummy code for just passing the test – it does not to ensure that the full feature will be implemented. For these reasons, refactoring assumes a prominent place. Using refactoring techniques, the feature implementation may be incrementally generalised from the pointwise dependencies required by the tests to the full logic required by the intended application.

Many tools have been developed to support TDD. In Java, test-driven development is most often done using JUnit [BG], a Java testing framework for unit tests.

Arguably, the focus on agile methods has taken the focus away from formal methods at large. This is somewhat unfortunate, as a substantial amount work has been done on effectively using formal methods as a basis for testing. For example, in 1981, the DAISTS system [GMH81] demonstrated that formal algebraic specifications could be used as basis for systematic unit testing. DAISTS identified four components of a test: *Conditional equational axioms* that serve as the test oracle; the *implementation* which must include an appropriate equality function; the *test data cases*, and a *quality control* of the test data (at least two distinct values). The Daistish system [HS96] took these ideas into the object-oriented world by providing a DAISTS like system for C++. The notation used for the axioms were also conditional equational based, giving a notational gap between the specification functions and the C++ methods. A more recent approach which also maintains this distinction is JAX (Java Axioms) [SLA02], which merges JUnit testing with algebraic style axioms. An automation was later attempted in AJAX using ML as notation for the axioms. These experiments suggest that an algebraic approach to testing may result

in much more thorough tests than standard hand-written unit testing. The emphasis on general axioms rather than specific test cases is the key to better unit tests.

This chapter builds on these above-noted ideas by introducing and describing several novel improvements which lead up to a tool-assisted method for implementing modular axiom-based testing for JUnit and Java. The two main contributions of this chapter include:

- A case study of the practical application of program object model adapters, transformlets and the other infrastructure (Chapter 6) for expressing interactive program generation tools.
- The detailed discussion of a generator tool, JAxT (Java Axiomatic Testing) [KH], which automatically generates JUnit test cases from all axioms related to a given class.

Additionally, the tool construction resulted in several new techniques and developments related to the research in testing pursued by Haveraaen [HB05], including (1) a technique for expressing as reusable axioms the informal specifications provided with the Java standard API; (2) a flexible structuring technique for implementing modular, composable sets of axioms for an API, which mirrors specification composition, as known from algebraic theory; and, (3) a discussion of practical guidelines for writing test set generators that exercise the axioms.

11.2 Expression Axioms in Java

In the proposed approach, axioms are expressed as idiomatic Java code, not in a separate specification language, as is common with other approaches based on algebraic specifications. There are several benefits to this: First, the developers need not learn a new formal language for expressing the properties they want to test. Second, the axioms will be maintained alongside the code and restructuring of the code, especially with refactoring tools, will immediately affect the axioms. This reduces or prevents any "drift" between the implementation and the specifications. Third, code refactoring, source code documentation and source navigation tools may be reused as-is for expressing and developing axioms. Fourth, it becomes easy to write tools to automatically produce test cases from the axioms. This is important because axioms may easily contain errors. This makes early and frequent testing of axioms desirable.

11.2.1 JUnit Assertions

The approach discussed here uses axioms to express invariants – assertions – about desired properties for an abstraction. The Java `assert` mechanism allows these prop-

erties to be stated as boolean expressions. If the expression evaluates to false, the `assert` mechanism will fail the program by throwing an `AssertionError` exception.

For testing purposes, the JUnit system provides a wider range of assertions than what `assert` offers. A notable difference is that the failure of one assertion terminates the immediately surrounding test, but not the remainder of the test suite. This allows a full set of tests to run, even if the first test fails. In addition, the JUnit assertions provide a detailed account of the error if the assertion does not hold, making it significantly easier to trace down what the problem can be.

11.2.2 Java Specification Logics

In standard specification theory, such as that used in [GMH81], axioms are formed from terms (expressions) with variables (placeholders for values or objects). If a variable is not given a value in the axiom, e.g., by quantification, it is said to be free. Interpreting this in the context of a programming language, free variables of a term can be viewed as parameters to the term. This leads to the following definition:

Definition 6 *An* axiom method*, or* axiom*, is a* `public`*,* `static` *method of type* `void`*, defined in an axiom class. The method body corresponds to an axiom expression (term), and each method parameter corresponds to the free variables of that expression (term). When evaluated, an axiom fails if an exception is thrown, otherwise it succeeds.*

It is recommended, but not required, that axioms use assertion methods in the style of JUnit, e.g.:

```
public static void equalsReflexive(Object a) { assertEquals(a,a); }
```

This axiom states the reflexive property for any object of class `Object` (or any of `Object`'s subclasses). The method `assertEquals(Object a, Object b)` is provided by JUnit and checks the equality of the values of the two objects using `a.equals(b)`. The axiom will also hold if `a` is the `null` object, since `assertEquals` safeguards against this case.

Specifying the desired behaviour of exceptions is also straightforward, albeit significantly more verbose:

```
public static void equalsNullFailure(){
  Object a = null, b = new Object();
  try {
    a.equals(b); // calling equals on the null reference
    fail(); // exception should have been raised
  } catch (NullPointerException e) {
    // OK
  }
}
```

Here, the effect of applying equals to a `null` reference is written as an axiom, named `equalsNullFailure`, with no free variables. The axiom states that, for any `a` and `b` where `a` is the `null` object, the expression `a.equals(b)` must raise an exception of type `NullPointerException`. (This is the Java semantics for invoking methods on null references.)

Expressive Power

In specification theory, one often asserts the expressive power of a specification logic [MM95]. The simpler logics have less expressive power, but have better meta-properties than the more powerful logics, i.e. reasoning about the logic is less difficult. It is beyond the scope of this chapter to provide a detailed classification and comparison of Java versus other specification logics, except for the following brief remarks.

Equational Logic – The simplest logic for the specification of abstract data types – classes – is equational logic which asserts that two expressions are equal (for all free variables). This is intuitively captured using `assertEquals` based on the `equals` method of Java. However, there are several theoretical problems here.

First, in normal logic a term is composed of mathematical functions deterministically relating inputs to outputs. In stateful languages, such as Java, one may modify one or more arguments rather than returning a value. The function may be (semi-) non-deterministic or the result of a function may depend on an external state. Such methods are beyond standard equational logic. As long as a method is deterministic, it is mostly straight forward to reformulate the terms of an equation as a sequence of statements computing the two values to be checked against each other.

Second, the `equals` method, on which `assertEquals` is based, may not be correctly implemented. So one should treat `equals` as any other method, and hence, also make certain it satisfies certain properties: it should be deterministic; it must be an equivalence relation (reflexive, symmetric, transitive); and, it should be a congruence relation, i.e., every method should preserve equality with respect to the `equals` methods. Fortunately, these are properties that can be written as Java axioms, and then tested[1]. For instance, one may repeatedly evaluate `equals` on two argument objects and ascertain that the `equals` should always have the same result as long as one does not modify the objects. Interestingly, the first two of these requirements are formulated in the Java API [Jav]. The last requirement will be discussed in section Section 11.4.

Third, there may be no way of providing a relevant `equals` method for some class, e.g., a stream class. This is known as the oracle problem [GA95]. However, using properly configured test setups, these classes may be made mostly testable as well.

Conditional Equational Logic – A more powerful logic is conditional equational logic. This allows a condition to be placed on whether an equality should be checked

[1] Testing will never prove these properties, but will serve to instill confidence about their presence.

for. In Java axioms, this is done by using an if-statement to guard whether the `assertEquals` should be invoked. Axioms for symmetry and transitivity of the `equals` method use this pattern, e.g.:

```
public static void equalsSymmetry(Object x, Object y) {
  if (x != null && y != null)
    assertEquals(x.equals(y), y.equals(x));
}
public static void equalsTransitive(Object x, Object y, Object z) {
  if (x != null && y != null && z != null)
    if (x.equals(y) && y.equals(z))
      assertEquals(x, z);
}
```

Here, an explicit `null`-check is required to avoid problems with `null` references in the assertions.

Quantifier-free Predicate Logic – Quantifier free predicate logic is an even more powerful logic. It permits the expression of negations and conditionals anywhere in the logical expressions. This is trivially expressed using boolean expressions in Java.

Full Predicate Logic – Full predicate logic causes a problem with the quantifiers. A universal quantifier states a property for all elements of a type. There is no counterpart in programming, although supplying arbitrary collection classes and looping over all elements will be a crude (testing style) approximation. Existential quantifiers may be handled by Scholemisation – given that there are algorithms for finding the value the quantifier provides.

Java-style Axioms Java style axioms have a different expressive power and allow expressing properties not captured by the standard logic. For instance, the distinction between an object and its reference is easily handled by JUnit assertions. Exceptions and methods that modify their arguments, rather than returning a result, can also be dealt with easily. Further, statistical properties can be expressed, such as the well-distributedness requirement on the `hashCode()` method, or temporal properties related to processes and timings, even against physical clocks.

The drawback to this extra expressive power is that one cannot immediately benefit from the theoretical results from the more standard specification logics. That is, the general theoretical results from algebraic specifications are not directly applicable to specifications written in a "Java logic".

This chapter will stick to the more value-oriented aspects of Java as a formal specification language, hopefully giving an indication of the intuitive relationship between this style and the standard specification logics.

11.3 Structuring the Specifications

All classes in Java are organised in a strict hierarchy forming a tree with the type `Object` at the top. A class may implement several interfaces. An interface may inherit several other interfaces. Further, a given class should satisfy the (informally stated) assumptions and requirements of each of its supertypes[2]. This is illustrated by the following simple class `Position`, which is used to index the eight-by-eight squares on a chess board:

```
public class Position implements Comparable<Position> {
  private int x, y; // range 0<=x,y<8
  public Position(int a, int b) { x=a % 8; y=b % 8; }
  public int compareTo(Position q) {
    return x-q.x; // ordering only on X−component }
  public boolean equals(Object obj) {
    final Position q = (Position) obj;
    return x==q.x && y==q.y; }
  public int getX() { return x; }
  public int getY() { return y; }
  public int hashCode() { return 3*x+y; }
  public void plus(Position q){ x=(x+q.x) % 8; y=(y+q.y) % 8; }
}
```

The method `plus` gives movements on the board, e.g., `k.plus(new Position(1,2))` for moving a knight `k`. The position class is a subclass of `Object` (which is implicitly inherited) and it implements `Comparable<Position>`. The intent is that the `Position` methods should satisfy all requirements given by its supertypes, i.e., those from the class `Object` and those from the interface `Comparable<Position>`, as well as any requirements given for `Position` itself.

Institutions

It would be desirable to write the requirements of classes and interfaces as sets of axioms, in a modular fashion, and then allow these sets to be composed soundly. The notion of institution [ST88] provides the mathematical machinery permit expressing requirements in modules called specifications. Each specification provides a set of axioms. Operations exist for building larger, compound specifications from smaller ones. The specification for a type can be extended with new methods, thus dealing with the extension of classes or interfaces by for example using inheritance. Axioms may be added to a specification, for example to provide additional requirements for a subtype. The union of specifications may also be taken. This allows the construction

[2] In Java terminology a type encompasses both class and interface declarations.

of a compound specification for all supertype axiom sets.

The theory of institutions shows that one can safely accumulate any axioms from the supertypes as well as add new axioms for Position. More importantly, it shows that this accumulation will be consistent and not cause any unforeseen interaction problems, as often is the case when one considers inheritance among classes. In this sense, the modularisation and composition properties of specifications are a lot more well-behaved than that of software code. In addition, framework of institutions provides a significant freedom in organising axioms so that they become convenient to work with. The method described in this chapter uses this freedom to allow a flexible and modular organisation of axioms alongside the class and interface definitions.

11.3.1 Associating Axioms with Types

Axioms, in the form of static methods, are grouped into Java classes. This immediately integrates the axioms with all Java tools. During the development process, axioms will be refactored along the main code, e.g., when a method is renamed or the package hierarchy is modified. This is considerably more developer-friendly than using separate specification languages.

Definition 7 *An* axiom class *is any class A which implements a subinterface of* Axioms<T>*, contains only axiom methods, its* axiom set*, and where T specifies which type the axioms* pertain to.

The name of a class providing axioms may be freely selected and placed in the package name space, but it must be labelled with an appropriate *axiom marker*. Labelling is done by implementing one of the predefined subinterface of Axioms<T> to signify whether the axiom set is required or optional for *T* or its subtypes.

Definition 8 Required axioms *are defined using the* RequiredAxioms<T> *marker interface on an axiom class A, and states that all axioms of A must be satisfied by T and and all its descendants.*

Using this structuring mechanism, it is possible to group the required axioms for equals, introduced in Section 11.2.2, into a class EqualsAxioms which implements RequiredAxioms<Object>, by defining the methods equalsSymmetry, equalsTransitive and equalsNullFailure in this class (to complete the specification for equals(), axioms for testing reflexivity and determinism are also required). Similarly, hash code axioms may be captured as follows:

```
public class HashCodeAxioms implements RequiredAxioms<Object> {
  public static void congruenceHashCode(Object a, Object b) {
    if (a.equals(b)) assertEquals(a.hashCode(), b.hashCode());
  }
}
```

```
1  public class PositionPlusAxioms implements RequiredAxioms<Position> {
2    public static void associativePlus(Position p, Position q, Position r) {
3      // compute pc = (p+q)+r;
4        Position pc = new Position(p.getX(), p.getY());
5        pc.plus(q);
6        pc.plus(r);
7      // compute p = p+(q+r)
8        q.plus(r); // destructive update
9        p.plus(q); // destructive update
10     assertEquals(pc, p);
11   }
12   public static void commutativePlus(Position p, Position q) {
13     Position pc = new Position(p.getX(), p.getY());
14     pc.plus(q);
15     q.plus(p); // destructive update
16     assertEquals(pc, q);
17   }
18 }
```

Figure 11.1: Axioms requiring that `plus` is associative and commutative.

Figure 11.1 shows some axioms for `Position`. Since `plus` modifies its prefix argument, a separate object `pc` is necessary to not modify `p` before it is used as an argument in the second additions (lines 9 and 15). This is not a problem for `q`, as it is not modified in the first additions. These axioms are destructive on the test data: the values of some of the arguments have been modified when the axioms have been checked (lines 8, 9 and 15).

The axioms belong to the same package as the type the axiom is associated to, so `PositionPlusAxioms` should be in the same package as `Position`. When axioms are retrofitted to an existing API, this placement may not be possible. One solution is to place the new axiom classes in an identically named package, but with `jaxt` as a prefix to the package name. Then the axioms for `Object` would go in a package `jaxt.java.lang` and the axioms for the Java standard collection classes would end up in `jaxt.java.util`.

Figure 11.2 shows how all the axioms for the supertypes, together with the axioms for `Position`, specify the design intent for `Position`.

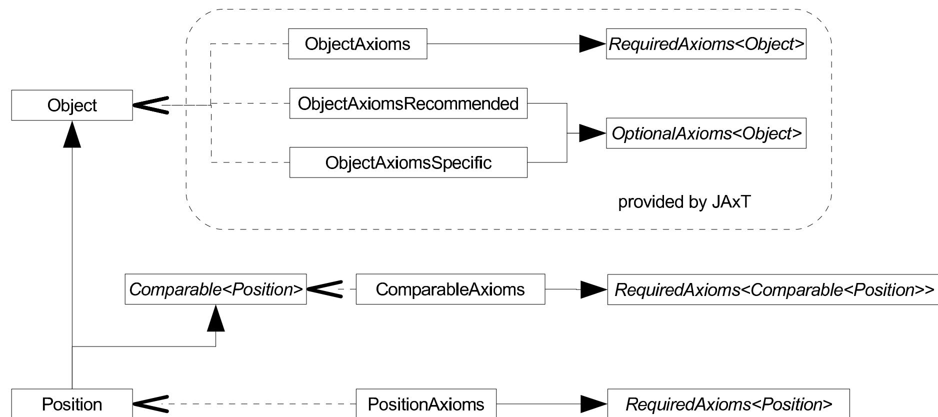


Figure 11.2: Organisation of axiom sets for `Position`. Boxes with italics represent interfaces. `XAxioms`-classes depend on, or pertain to, the classes they describe, marked by a stippled arrow. Filled arrows indicate inheritance.

11.3.2 Optional and Inherited-only Axioms

A close reading of the Java API documentation [Jav] shows that it not only contains requirements – the kind of axioms described in this chapter – but also a multitude of recommendations and class-specific descriptions. For instance, the description of `Comparable<T>` contains the phrase "strongly recommended" and other places use the phrase "recommended". In the class `Object`, and some of its subclasses, such as enumerations, it is specified that reference equality is the same as value equality.

Definition 9 Optional axioms *are defined using the* `OptionalAxioms`*<T> marker interface on an axiom class A, and states that all axioms of A pertain to T, but are optional for any descendant of T, unless the programmer of a subtype has requested these axioms specifically. If A is an optional axiom set of T, this set may be inherited to a descendant D of T by adding the marker interface* `AxiomSet`*<A> to an axiom class pertaining to D.*

The optional reference equality of `Object` is asserted by `equalsReference()` in the following axiom class, `ObjectEqualsReference`:

```
public class ObjectEqualsReference implements OptionalAxioms<Object> {
  public static void equalsReference(Object x, Object y) {
    if(x != null)
      assertEquals(x.equals(y), x == y);
  }
}
```

This axiom class implements the `OptionalAxioms<T>` interface, and therefore, the method `equalsReference()` will not be required automatically by the subtypes of

`RequiredAxioms<T>`	*required by type T and all descendants*
`OptionalAxioms<T>`	*required by type T, but not its descendants.*
`SubclassAxioms<T>`	*required by all subtypes of T but not by T*
`AxiomSet<Ax>`	*import axiom set* `Ax`

Table 11.1: Summary of axiom structuring mechanisms.

`Object` unless one of their axiom classes implements `AxiomSet<ObjectEqualsReference>`. Consider the following empty axiom class which states axioms pertaining to the class `EnumDemo` (not shown).

```
public class EnumDemoAxioms implements OptionalAxioms<EnumDemo>,
                                       AxiomSet<ObjectEqualsReference>
{ }
```

While the class `EnumDemoAxioms` does not specify any axioms itself (although it could), it does activate the axioms from the set `ObjectEqualsReference`; that is, the optional equality axioms are now required for `EnumDemo` (but none of its subclasses, since `EnumDemoAxioms` is itself marked optional). As expected, even though `Position` inherits directly from `Object`, the object equality axioms (`ObjectEqualsReference`) have no relevance since the axiom classes for `Position` do not implement the type `AxiomSet<ObjectEqualsReference>`.

It is sometimes necessary to state axioms that only pertain strictly to subclasses, and not originating from the base class which is exempt. This is done using subclass axioms.

Definition 10 Subclass axioms *are defined using the* `SubclassAxioms<T>` *marker interface on an axiom class A, and states that all axioms of A pertain to all subtypes of T, but not T itself.*

Consider a (possibly abstract) class `C` which contains declarations and common methods for its subclasses where one may want to check as much as possible of `C` and be certain that all subclasses satisfy all axioms. By marking some axioms with `SublassAxioms<C>`, these will not be tested on class `C` itself, but will be checked on all subclasses of `C`.

Table 11.1 summarises the structuring mechanisms for axioms. These are used in the small example class hierarchy depicted in Figure 11.3. It is important to note that as all these relationships are formally marked in the code, they can be discovered automatically by a tool, see Section 11.6.

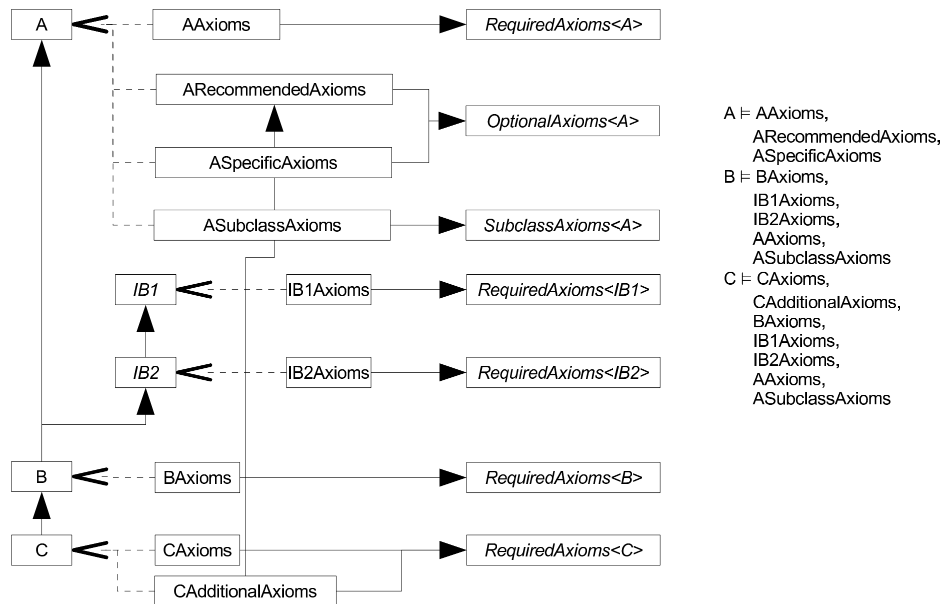


Figure 11.3: Organisation of axioms. The notation `X` $\models$ `AxL` means that class `X` satisfies all the axioms in the axiom list `AxL`.

11.4 Java API caveats

Capturing the informal requirements described in the Java API documents into machine-checkable axioms is a non-trivial exercise. This section documents some of the challenges encountered in this process.

11.4.1 Override and Overload

Most object-oriented languages, including Java, distinguish between overriding a method and overloading a method. Method overriding occurs when a subclass redefines a method defined in one of its superclasses, thus altering the behaviour of the method itself without changing the class interface. On the other hand, method overloading allows the same method name to be reused for different parameter lists, e.g.:

```
1 public class Test extends Object {
2   int x, y;
3   public boolean equals(Object obj){
4     return x==((Test)obj).x
5         && y==((Test)obj).y;
6   }
```

```
7    public boolean equals(Test obj){
8      return x==obj.y;
9    }
10 }
```

The class `Test` provides two overloaded methods where the first overrides the `equals` method from `Object`. Consequently, the axioms for `equals()`, defined in `Object`, will be applied to this method, while the equals method in lines 7-9 will not be tested. This might be surprising, but follows from the semantics demonstrated by the following method calls:

```
Object o1 = new Object();
Object c1 = new Test();
Test c2 = new Test();
o1.equals(c2); // from Object
c1.equals(c2); // from lines 3–6
c2.equals(c2); // from lines 7–9
```

The overloaded version will only be called if both arguments have the *declared* type `Test` (or a subclass thereof).

In a language with templates, like C++, one could use genericity to apply the axioms to any one of such overloaded methods. Unfortunately, this is not possible using Java generics (since the type erasure of a generic `<T> boolean equals(T o)` would be identical to `Object equals(Object o)` defined in `Object`. This is forbidden by the type system.) so the axioms for `equals` will have to be redeclared for class `Test` if they are to be applied to the second `equals` (line 7-9).

11.4.2 `clone` and Other Protected Methods

The `clone()` method is declared as a protected method of `Object`. This makes it problematic to specify axioms because a protected method is only accessible to subclasses. Since `clone()` is protected, it is not possible for an axiom method to invoke it on an object of type `Object` and, consequently, it cannot be adequately described. This is unfortunate, because the API documentation lists an important number of recommended (optional) axioms for `clone()`.

In principle, the same limitation holds for any protected method and is shared by any testing approach where the testing methods are defined in a class separate from the class being tested. In Java, it is possible to circumvent this limitation by declaring the testing class in the same package as the tested class – classes in the same package may invoke each other's protected methods without being in a subtype relationship.

For `clone()`, which is defined in the immutable package `java.lang`, axioms must be written specifically for every class that makes `clone()` public. The axioms will then apply to all subclasses of this class, but not to any other class that exposes the

`clone()` method. Following the previous remarks, it is not possible to circumvent this restriction using Java generic method declarations.

In the instance of `clone()`, another attractive possibility would be to state the axioms for the interface `Cloneable`, `Cloneable` is defined in the Java language specification as a marker interface used to enable the cloning algorithms encoded in the `clone()` method from `Object`. Unfortunately, `Cloneable` is an empty interface which does not (re)declare `clone()` as a public method.

11.4.3 The `equals` "congruence" relation

In the standard approaches to equational specifications, the equality test is a congruence relation. This is an equivalence relation which is preserved by all methods. Preservation means that, for any two argument lists to a method, if the arguments are pairwise equal, then the results of the two method calls also must be equal.

In Java, the `equals` method defined in `Object` is close to, but not entirely, a congruence relation. If it were, for any two objects `a` and `b`, `a.equals(b)`, then

- `a.hashCode()==b.hashCode()`, which is a required axiom in the Java API,
- `(a.toString()).equals(b.toString())`, but this is not an axiom in the Java API.

Unfortunately, this means that one of the central means for closely relating Java to equational specification theories is unavailable.

For `equals` to have the congruence property, it would have to be implemented as a form of "meta property", requiring axioms to be written for every new method. Remember that overridden methods inherit such properties from the superclass. A tool like JAxT could easily handle this by either explicitly declaring the needed congruence axioms, or tacitly assuming them, thus generating the necessary test code even without making the axioms explicit. Even without tool support in the current iteration of JAxT, maintaining a congruence relation is a *strongly recommend* practise wherever feasible. Future releases may add facilities for handling the congruence property.

11.5 Testing

The previous section described how to express and organise axioms in and for Java. These axioms can be used as test oracles for ensuring implementations exhibit the intended behaviour. For the testing setup to be complete, relevant test data must be provided. During testing, each data element from the test data set will be provided for the relevant free variables of the axioms.

11.5.1 Test Data Generator Methods

A test data set may be as simple as the data from a typical test case, as practised by typical agile or test-driven development. For the many cases where additional testing is desired, the JAxT framework has an infrastructure that opens up for a much more systematic approach to test data.

Creating the test is the responsibility of the developer. The first step towards creating a test set is to implement a test set generator. The JAxT generator wizard, explained in Section 11.6, will provide a test set generator stub on the following form, for a class `X`:

```
public class XTestGenerator {
  public Collection<X> createTestSet()
    { return null; }
}
```

The developer must fill in the `createTestSet()` method with code that produces a reasonable test set of `X` objects. This can be data from an existing test case, or a static collection of test values.

For `Position` objects, the following test set generator method, placed in the class `PositionTestGenerator`, produces a collection of random, but valid, `Position` objects:

```
public static Collection<Position>
createTestSet() {
 final int size = 200;
 List<Position> data =
   new ArrayList<Position>(size);
 Random g = new Random();
 for(int i = 0; i < size; i++) {
   data.add(new Position(g.nextInt(8),
                        g.nextInt(8));
 }
 return data;
}
```

The random number generator provided by the Java standard library is used to produce test data. Some authors, such as Lindig [Lin05], have reported that random test data may be more effective than most hand-crafted data sets in detecting deficiencies. In a similar approach, discussed by Claessen and Hughes [CH00], algorithms are used for deriving random test data sets based on abstract data types.

In the particular case of `Position`, the complete test data set of the 64 distinct position values could be provided, but for completeness, distinct objects with equal values due to the difference between equality on object references with equality on object values are also needed.

If the objects of a class `X` are very time-consuming to instantiate, one might consider implementing a test set generator method that extends `IncrementalTestGenerator`. This class provides a ready-made `createTestSet()` method that returns a collection which instantiates its elements on demand. The developer must implement the method `X generateNewValue(int index)`, which must produce random objects of type `X`. The `IncrementalTestGenerator` takes care of memoization.

Using `IncrementalTestGenerator` will not result in better total running times of the tests. It may, however, allow the developer to uncover errors earlier in the event of a failed axiom since no time is spent up front to generate a full test data set which will never be traversed entirely. A formulation using the `generateNewValue()` method may sometimes be cumbersome. For this reason, the use of `IncrementalTestGenerator` is optional.

11.5.2 Determining Test Set Quality

The JAxT library offers a few simple, but powerful checks for properties of test sets. For example, there is a method that checks whether the provided collection has at least two distinct data values, similar to the requirements in [GMH81]. Another method can test whether there are at least three distinct objects with equal values – necessary if a transitivity axiom is to be exercised.

The purpose of these checks is so developers can apply them to the test sets produced by their generators and ensure some degree of test set quality. The quality assurance of test sets usually occurs during the development of test set generators themselves. In general, it is not necessary to run test quality metrics as part of a test suite, provided that the developer has checked and acquired sufficient confidence in the test set generators.

Additional checks, in particular statistical metrics which can be used to judge distribution characteristics, are scheduled for inclusion into JAxT, but how to best integrate existing test generation approaches is still an open problem. There is a wealth of material to chose from, however, such as [DO91, TL02].

11.5.3 Running the Tests

When combining the test data with the axioms to run the tests, there are several issues to take into account. Some of the axioms may be destructive on the data sets, so each test data element must be generated for each use. This is normally handled by fixtures in unit testing tools, but for efficiency reasons, one may choose to have test data generation in the test methods themselves. While this entails more verbose unit test methods, since the test methods will be automatically generated from the axioms and the test data generator methods, it presents no extra burden on the user.

With automated test data generation, it becomes very easy to (accidentally) create large data sets. In general, larger test sets improve the quality of the testing, but run-times may become excessive when testing axioms with many arguments. In the example `Position` class, all tests for axioms with up to three free variables take less than a couple of seconds and are within the normal time-frame for repeated unit testing in the TDD approach. For more complicated axioms, such as checking that `equals` is a congruence for `plus`, a quadruple loop on the data set is required and takes about 30 seconds. While in the the edit-compile-run cycle, regular unit testing using large data sets is not ideal. The framework currently provides limited support for adjusting the data set sizes. Additional work independent of JAxT is required to allow the developer to flexibly tune the size, and to continuously vary the trade off between thorough testing versus short testing times.

11.5.4 Interpreting Test Results

Writing code and axioms, and their associated tests is error-prone. For this reason, early and frequent testing is valuable. When a test fails, it only states that there is some mismatch between the formulated axiom and the implementation of the methods used in the axiom. It is important to remember that, at least in the beginning, errors can just as easily be in the axiom as in the code. Therefore, both must be checked carefully. As always, newly written pieces of code, whether a new axiom or a new class, are typically more likely to contain errors than legacy pieces that have already been thoroughly tested.

11.6 Test Suite Generation

The techniques described in the previous sections are structured and formal enough for a tool to aid the developer in deriving the final unit tests and test set generators. The author has experimented with such automatic generation of unit tests from axioms by building a prototype testing tool called JAxT[KH][3]. This section describes the findings from this prototyping experiment.

Below is the test class generated by JAxT for `Position`, with comments removed:

```
public class PositionTest extends TestCase {
 private Collection<Position> testSetPosition;
 public PositionTest(String name)
  { super(name); }
 protected void setUp() throws Exception {
  super.setUp();
  testSetPosition =
```

[3] JAxT stands for Java Axiomatic Testing.

```
   PositionTestGenerator.createTestSet();
 }
 protected void tearDown() throws Exception {
  super.tearDown();
  testSetPosition = null;
 }
 public void testObjectReflexiveEquals() {
  for (Position a0 : testSetPosition)
   reflexiveEquals(a0);
 }
 public void testComparableTransitiveCompareTo() {
  for(Position a0 : testSetPosition)
   for(Position a1 : testSetPosition)
    for(Position a2 : testSetPosition)
     transitiveCompareTo(a0, a1, a2);
  }
}
```

The following sections will explain how this test case was derived.

11.6.1 Generating Tests from Axioms

The task of JAxT is to automatically derive unit tests for a given class `C` using those axioms from the set of all axioms associated with `C` – each axiom induces a new unit test. The set of associated axioms can be found by inspecting the axiom classes associated with `C`.

When the axioms were created, the programmer clearly specified which axioms directly pertained to `C` by placing `C`'s axioms into those classes implementing the interface `Axioms<C>`. By placing the marker `Axioms<C>` on the an axiom class `AX`, all (static) methods in `AX` are considered to be axioms for `C` and must therefore be fulfilled by an descendant of `C`.

This implies that `C` itself may have inherited axioms from a related superclass. For any (direct or indirect) superclass `P` of `C` or (direct or indirect) interface `I` of `C`, one or more axiom classes may exist with `Axioms<P>` or `Axioms<I>` markers. Methods in these classes are also considered to be axioms associated with `C`.

Computing Axiom Sets

In order to produce the final set of test methods for a class `C`, all applicable axiom methods must be found. As suggested in Figure 11.3, axioms for all named types provided by `C`, i.e. its superclasses, its or any of its supertypes' interfaces, are searched.

The algorithm compute-axioms() detailed in Algorithm 1 produces the final list of axiom methods for C, The resulting list is then fed into a test case generator. The test generator works as follows:

First, it computes the required axiom sets of C; that is, all classes which implement RequiredAxioms<C> or OptionalAxioms<C>. Next, the supertypes of C are traversed and, for each, the set of subclass and required axioms are collected. After this is done, an initial set of axiom classes (i.e. axiom sets) is given in Ξ. Note that the axiom classes themselves may pull in additional (optional) sets, via the implements AxiomSet<AX> mechanism. These optional sets are then added to Ξ and the final set of axiom sets is obtained. The methods of these axioms are the final product of compute-axioms().

Algorithm 1 compute-axioms(C)

Ξ := required-axiom-sets-of(C) $\cup$ optional-axiom-sets-of(C)
for $T \in$ supertypes-of(C) **do**
  $\Xi := \Xi \cup$ subclass-axiom-sets-of(T) $\cup$ required-axiom-sets-of(T)
end for
for $AX \in \Xi$ **do**
  $\Xi := \Xi \cup$ super-axiom-sets-of(AX)
end for
return $\bigcup_{AX \in \Xi}$ methods-of(AX)

User interaction

Not all the details of the generation can be inferred from the source code, such as which package the generated test class should be placed in, though reasonable defaults can be suggested. To support various working styles and project organisations, the prototype offers a graphical generator wizard that allows user to optionally specify corrections to the assumed defaults. The user accesses the GUI to select a single class or multiple classes or packages to invoke the test generator. A wizard appears that allows the user to select which classes to generate test set generators for and where to place them. Further, the user can select which package the generated test case should be placed in and whether to generate a test suite, if tests for multiple classes have been requested.

The user may also include additional axiom libraries that may contain relevant axioms for the type hierarchy at hand. For example, the JAxT library already provides axioms for Object and Comparable that may be reused as desired. This axiom inclusion feature supports reuse of axiom libraries that may be distributed separately from existing implementations. A major benefit of this design is that axioms can be easily retrofitted for existing libraries, such as the Java Standard Libraries, and such

axiom libraries can be incrementally developed without modification or access to the source code of the original library.

Thanks to JAxT, users can easily and incrementally update existing test classes, e.g., when axioms have been added or removed. By reinvoking JAxT, the test classes will be regenerated and additional test class stubs for new data types will be created. As Eclipse refactorings carry through to the axioms, the axioms will remain in sync with the code they test.

Generation of Tests

When the user requests axiom tests for a class `X`, a corresponding `XAxiomTests` is generated (name may be customised). This is a JUnit fixture, i.e. `XAxiomTests` derives from `junit.TestCase`, provides a set of test-methods and may provide a `setUp()` and a `tearDown()` method.

The `setUp()` method initialises all necessary test sets, as follows:

```
setUp() {
 testSetT0 = T0Generator.createTestSet();
 ...
 testSetTn = TnGenerator.createTestSet();
}
```

The `createTestSet()` methods return `Collections` which will be traversed by the tests. The exact set of T_i`Generator` calls is discovered from the argument lists of the axioms exercised by this test fixture.

For each axiom `ax(`T_0`, ..., `T_n`)` in axiom class A a `testAAx()` method is generated, on the following form:

```
/** {@link package.of.A#ax(T0, ..., Tn)} */
public void testAAx() throws Exception {
 for(T0 a0 : testSetT0)
  ...
   for(Tn an : testSetTn)
    A.ax(a0, ..., an);
}
```

This test will invoke the axiom A`.ax()` with elements from the (random) data sets `testSetT`i. The generated Javadoc for `testAAx()` will link directly to the axiom being tested.

After all tests have been run, the tearDown() method is executed, which takes care of releasing all test sets:

```
tearDown() {
 testSetT0 = null;
 testSetTn = null;
```

```
}
```

For convenience of this example, the pattern above has been idealised. The generator produces slightly different `setUp()` and test methods in cases where the data sets are not shared between all test methods. Consider the situation where a fixture has two test methods, `testA()` which uses the data set `testSetA` only, and `testB()` which uses `testSetB` only. Since `setUp()` is invoked before every test method, it is wasteful to initialise both `testSetA` and `testSetB` every time. Therefore, the generator will only put test sets which are shared among all test methods in `setUp()`. Local invocations to `createTestSet()` will be placed in the test methods for the other test sets.

11.6.2 Organising Generated Tests

The test generator produces two types of generated artifacts: the test set generator stubs which are meant to be fleshed out by the programmer, and the unit tests, which are meant to be executed through JUnit and never modified. For this reason, it is recommended that unit tests are generated into a separate package, such as `project.tests`, to place them separately from hand-maintained code. If there are additional, hand-written unit tests in `package.tests`, placing the generated tests into a separate package, such as `project.tests.generated`, is preferred. The test set generator stubs are clearly marked as editable in the generated comments and the test cases are marked as non-editable.

As previously stated, the generator can produce a suggested test suite based on a package or a project that lists all the tests requested in the same invocation of the test generator wizard. The purpose of these suites is to catalogue tests into categories and for the developer to be able to execute different categories at different times; some axioms may result in very long-running tests, others may not be interesting to test at each test suite execution and others still should be executed very frequently.

Both the test suites and test set generator stubs are freely editable. The generator will never overwrite these artifacts when they exist, even if the developer accidentally requests it.

11.6.3 Executing Tests

The current prototype tool supports two primary modes of test execution: from the command-line and through and interactive GUI.

The command-line mode is intended for integrating the tests into nightly build cycles, or other forms of continuous integration. We impose no restrictions on the management of test suites – the tool merely aids in producing suggested starting points – so varying degrees of testing may be decided on a per-project or per-build basis. By organising the generated tests into categories, it becomes easy to select which sets of axioms should be exercised at any given build.

The interactive mode reuses the JUnit framework. Once a test case (or test suite) has been generated, it can be immediately executed by the developer like any other JUnit test. Since there is a direct link in the Javadoc for every generated test method, it is trivial to understand which axiom is violated when a given test method fails.

JAxT does not provide any test coverage analysis itself, but existing solutions for JUnit, such as Cobertura [Cob] and NoUnit [NoU], are applicable.

11.7 Implementation

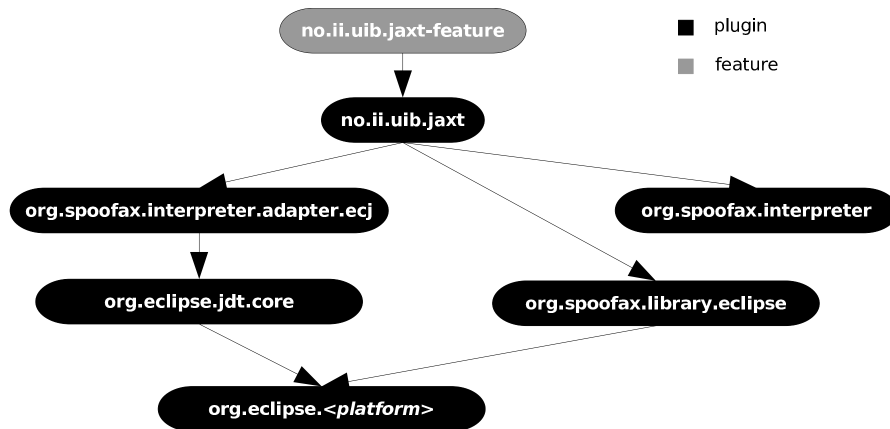


Figure 11.4: Components of the JAxT generator.

The JAxT tool is implemented as a plugin to the Eclipse development platform [Ecl]. It is divided into the component seen in Figure 11.4. The POM adapter technique, described in Chapter 4, and the Eclipse integration framework provided by Spoofax, as discussed in Chapter 9, were crucial for its construction.

The test generator wizard is very similar to the one provided by JUnit: The JUnit test generator is applied to a class *C* and produces a testing stub for each method in *C*, whereas JAxT, when applied to *C*, produces an immediately executable test for every axiom pertaining to *C*. The wizard interface collects all necessary user input. When the forms are complete, the control passes to a compiled MetaStratego transformlet which implements all the analysis and generation logic.

The script will use the Stratego/J FFI functionality to call into the Eclipse Compiler for Java and query for all necessary compilation units (`.java` and `.class` files). For compilation units that have source code, ASTs are extracted and adapted to terms using the POM adapter shown in Chapter 10. For "sourceless" compilation units, i.e. `.class` files, ECJ provides a read-only, high-level inspection interface. An additional POM adapter for this interface was implemented during the course of this work.

With this adapter, the JAxT transformation logic can traverse and inspect these compiler objects as well. This adapter provides a small signature for the notions of method and type bindings and by using it, many computations on the type hierarchy become very succinct and easy to express, e.g.:

```
ecj-all-supertypes-of =
    ?TypeBinding(_,_,_,superclass,superifaces,_,_)
  ; <collect-all(?TypeBinding(_,<id>,_,_,_,_,_))> [superclass | superifaces]
  ; map(!DottedName(<id>))
```

This three-line strategy will compute the transitive closure of all super classes and super interfaces (i.e. supertypes) of a given class (or interface), using the generic traversal strategy `collect-all` provided by the standard Stratego library. The strategy produces a list of `DottedName` terms, that contains the dotted names (fully qualified names) of the supertypes. Note the the super type hierarchy may in the general case be a directed acyclic graph and is rarely a tree. Even if the plain Stratego language is used, this presents no problems. Potential problems due to term building in graphs, discussed in more detail in Chapter 7, are avoided by only providing a read-only interface. Traversals will always terminate since there are no cycles.

Once the script has extracted the necessary information from the compilation units, it will will assemble ASTs for what will become the final JUnit test class and the test generator stub classes. Each of these ASTs will be written to a separate `.java` file. The Eclipse code formatter will be used to pretty-print the result. This ensures that the final result is in accordance with the user-configured settings for the Java project in which JAxT is applied.

11.8 Discussion and Related Work

The standard Java documentation, and that of many other software libraries, is rife with examples of formal requirements. These are typically not machine readable, nor machine checkable. As a result, these requirements are often inadvertently violated, often resulting in bugs which tend to be difficult to trace. The goal of program verification and validation techniques, including (unit) testing is to increase developer confidence in the correctness of their implementations. If a technique for formalising the library requirements was devised, and tests may be automatically generated from this formalisation, confidence that the library abstractions were used correctly would take a significant boost. This is what the JAxT, and other axiom-based techniques for test generation like JAX [SLA02], DAISTS [GMH81] and Daistish [HS96] do.

The testing approach sketched in this chapter is a continuation of the JAX tradition [SLA02] and is, in a sense, a combination of two techniques for ensuring robustness of software: test-driven development and algebraic specifications. The ideas of axioms and modular specifications are taken from algebraic specifications as

a way of succinctly describing *general* properties of an implementation, as opposed to the case-by-case approach normally advocated by TDD. Principles for integration into graphical development environments and the design of practical tools for test code generation and immediate execution of tests were inspired by TDD approaches. These principles allow the proposed method to bring instant feedback to the developer.

Any approach to (semi-)automatic test generation will eventually have to be implemented using some programming language. Experience from this case study suggest that constructing the generator in a language-general, domain-specific transformation language has several benefits compared to the author's previous experiences with implementing language processing using strictly general-purpose languages.

Succinctness – Compared to an implementation in a general programming programming, the transformation logic expressed in Stratego becomes compact and to the point (c.f. more detailed examples in Chapter 10). If care is taken to name the strategies and rules appropriately, most of the transformations read fairly well.

Infrastructure Reuse – The initial development of JAxT was very quick due to the reuse of the Eclipse Compiler infrastructure. Even if one accounts for the time taken to construct the additional POM (for type bindings), and the time spent constructing the AST POM itself, this was considerably less than the time necessary to construct a robust Java 1.5 parser from scratch, and much less than a stable type-checking front-end. The adapters were semi-automatically extracted from APIs in a matter of hours and, after a few more hours of plugging in the relevant FFI calls, the type checker was ready to be reused from within Stratego.

Development Tools – A weak point of most non-mainstream languages is the state of their development tools. This is also the case with Stratego which, for example, lacks an interactive debugger. The Spoofax development environment (Chapter 9) helped a lot, but additional work is required if the environment is to have a level of quality similar to that of the mainstream language environments.

Code Templates – Much of the generated code is composed from pre-defined templates. These are expressed using the abstract syntax of the ECJ, shown in Chapter 10. Both readability and maintainability would get a significant boost if these templates were expressed using concrete syntax, i.e. as concrete Java code. However, depending on the complexity of the templates, this would require the construction of a full Java 1.5 grammar which is embeddable into the transformation language.

11.9 Summary

This chapter presented a case-study of how the techniques proposed in Part III and Part IV of this dissertation are applicable to code analysis and interactive code generation. The study presented a tool-assisted approach for testing general properties of

classes and methods based on axioms and algebraic specifications expressed entirely in Java. It provided a detailed description of how desired program properties can be expressed as axioms written in an idiomatic Java style, including a rich and flexible mechanism for organising the axioms into composable modules (composable specifications). The proposed organisation mechanism is structured enough that the testing tool, JAxT, can automatically compute all axioms pertaining to a given class and generate JUnit test cases and test suites from the composition of these. By reexecuting JAxT periodically, the unit tests can trivially be kept in sync with the axioms, as these change.

The study also discussed design and implementation aspects of JAxT, illustrating how the techniques proposed in this dissertation may be applied in practise. The results of the study suggests that language-general, domain-specific transformation languages provide an attractive vehicle for expressing interactive language processing problems, but that additional tool support may be necessary before most programmers can be expected to benefit from the increased succinctness offered by these languages.

Part VI

Conclusion

Sometimes I want to run a garbage collector on the internet.

– Eelco Dolstra

12 Discussion

This chapter discusses different approaches for achieving language independent transformations and some fundamental tradeoffs related to these. The chapter also briefly discusses the relation of program transformation to other approaches for software evolution. Additionally, it contains a discussion on the place for open research systems in the pursuit of better practical transformation techniques.

12.1 Techniques for Language Independence

This dissertation is concerned with the development of techniques for expressing language-independent program transformations. Its motivation has been to find ways of making transformations applicable across different subject languages quickly and easily. The techniques proposed herein are by no means exhaustive. A discussion of alternative approaches is therefore warranted.

There are several possible directions for achieving language-independent program transformations. All of them must tackle the tradeoff between two opposing requirements: the need to hide language details versus the level of detail required by a given transformation. Abstracting over language details hides irrelevant differences in subject languages and enables higher-level program models. Consequently, transformations written for these models may be applied to languages which are abstractable into the higher-level models. The requirements placed on the model varies greatly between transformation tasks, however. The contents of a model used to analyse the module dependency graph of a program is quite different from one which is used for intra-procedural control-flow analysis. The combination of several approaches is therefore more likely to give good results.

12.1.1 Abstracting over Data

Capturing software in a high-level and general program object model (POM) is a data-centric approach to language independence. The goal of this technique is to abstract over irrelevant language details and provide a uniform model across subject

languages. Abstract syntax trees (ASTs) may be seen as the first step in this direction. ASTs abstracts over most of the syntactical "noise" in the subject language, but may still be used to reproduce the original program faithfully modulo layout and comments. POMs may be arbitrarily more abstract. The PROGRES example in Chapter 2, Section 2.4.2 illustrates how the concept of program configurations may be described abstractly. This abstract model could support transformations, but mapping these transformations from the abstract model back to equivalent operations on the original source code is generally a hard problem to solve.

The fundamental tradeoff for the data-centric technique is the choice of what should be considered irrelevant language details. This clearly depends on the transformation problem for which the abstract model should be used. Arriving at a final authoritative language independent program object model is therefore very likely to be infeasible. A more versatile approach is required which can account for the varied needs of the transformations.

12.1.2 Expressing Generic Algorithms

Formulating generic and parametrised transformation algorithms, where language specific components can be inserted, may be considered a "function-centric" approach to language-independence. The strategic programming paradigm supports this approach well by dividing transformation programs into general transformation logic and data processing rules. The principle is that the general transformation logic, in the form of strategies, is designed for the application of language-specific data processing rules. By replacing the rules, the same logic may be applied to different subject languages.

This approach works well for a good number of problems, but suffers from drawbacks discussed in Chapter 5, Section 5.4.3. An additional drawback, particular to program transformation, is that the designer has no abstract signature to write the algorithm against. The model abstracting over subject languages – what may be called an abstract language – is never defined explicitly in the transformation program. Instead, the algorithm is written against a mental model hidden in the rule set. The model is purely conceptual, but the transformation must nonetheless respect it. The lack of a clear abstract language definition often makes it very hard to reason about the transformation. At best, the model exists in the form of well-written documentation. Accidental violation of the model is easy because the transformation language compiler cannot check any of its rules. In the course of development, the transformations are often tested against a concrete selection of subject languages to raise confidence in their correct behaviour. This may easily introduce an unintended bias in the formulation favouring the example selection.

Describing abstract languages for language-independent program transformation is still an open research problem. It is possible that the answer may be found from

studying better and more flexible ways of describing computer languages in general.

12.1.3 Adapting Generic Algorithms

Aspects may further improve the genericity of an algorithm by exposing additional variation points for the algorithm user. In some cases, accidental implementation bias may be corrected without changing the algorithm.

12.1.4 Modular Language Descriptions

Finding a good formalism for describing computer languages in small, reusable modules has been and, many would claim, still is a long-standing goal of computer science. The approach taken in this dissertation is to describe the language semantics as transformation libraries that define how the various languages features are translated into a minimal core language and, eventually, (via a machine-specific compiler or interpreter) into executable machine code. A clear drawback of this approach is that the description is very tied to its eventual application: the compilation and execution of programs. Unless special care is taken during the design of the transformation library, reusing the implementation for other tasks, perhaps for program validation or defect checking, may be impossible. Research into modular language semantics holds some hope that "problem neutral" descriptions may eventually be possible.

The research towards modular language descriptions [Mos04a] is concerned with capturing the semantics of programming languages into small modules that each describe a particular language feature. These modules may be composed with other modules. The final composition describes the entire language and may form the basis of any language processing tool for the composed language such as a compiler, type checking front-end, refactorer or defect checker.

Various formalisms for defining modular semantics have been devised including monadic denotational semantics [Mog91], abstract state machine montages [KP97], action semantics [DM03] and modular structural operational semantics [Mos04b]. None of these formalisms have seen significant adoption, unfortunately.

A firm and general basis for describing subject language semantics could be of great use for transformation developers. For example, determining whether it is possible to find a reasonable mapping from a given subject language to a given abstract language could become easier. Additionally, it could improve the means available for validating (or verifying) that a given transformation respects certain semantics of a subject language.

12.2 Other Approaches to Software Evolution

Program transformation is not the only component in a solution toward good software evolution. Therefore, it is important that the techniques proposed in this dissertation for expressing reusable, language-independent program transformations work well with the (relatively) new and forthcoming programming and software evolution methodologies such as refactoring, generative programming, interactive development environments, extensible languages and aspect-oriented programming. The experimentation performed during the development of the case studies in Part V shows that the techniques apply well to generative programming and that integration into interactive development environments is significantly easier due to the POM adapter. An early prototype for refactoring of Java code has been constructed. This suggests that implementing refactorings in a "strategic" style using the abstractions proposed in this dissertation may be very powerful. Additional experiments are necessary in order to gain more experience before a final conclusion can be made. Experience from the development of the domain-specific aspect language described in Chapter 8 suggests that aspect-oriented subject languages are reasonably well supported. Extensible languages are discussed in Chapter 13.

12.3 Availability of Research Systems

The investigation and analysis leading to the survey of software transformation systems in Chapter 2 uncovered that practically all of the (still active) research systems described in the literature are freely available for download. With only a handful of exceptions, the full source code for these systems were also provided. In many cases, the availability of source code proved crucial to understanding the detailed workings of many features. The availability of source code was also important during the analysis leading up to the POM adapter (Chapter 4). During this analysis, it became necessary to consult concrete compilers and transformation systems to account for design decisions usually glossed over in the literature.

An important theme of the dissertation is to demonstrate that the techniques proposed herein are applicable in practise. For this reason, all the source code for the software constructed for this dissertation, including the case study prototypes, is available for download via `www.spoofax.org`[1].

[1]The name "Spoofax" was selected because available `.org` domains are hard to come by.

– What if we build this giant trebuchet and lay siege to the computer science department?

– Anya Helene Bagge

13 Further Work

This chapter discusses possible future research directions based on some of the unresolved problems encountered during this research.

The aspect language (Chapter 5) may benefit from additional pointcuts that identify program locations based on program flow in the style of [AAC+05]. An additional extension would be to add support for dynamically weaving aspects into transformlets as these are loaded. For this to be possible, the aspect meta program must be maintained alongside main program at runtime so that it can be invoked when new transformlets are loaded. When using the aspect language for algorithm extension and adaptation of transformation skeletons, the resulting aspect programs sometimes become difficult to read and understand. There are patterns in these aspect-based adaptation schemes which may be distilled into new, higher-level language concepts. It is not yet clear exactly which patterns form the best foundations for a more general adaptation language.

The current graph language extension (Chapter 7) is minimal enough to capture cyclic graphs. This is sufficient for capturing the flow-based program models found in compilers, but may be insufficient for other applications. It may be useful to extend the language with a more general notation of graphs thus arriving at a more general concept of strategic graph rewriting. If this path is pursued, a graph visualisation plugin for Spoofax would be highly beneficial.

The program object model adapter technique (Chapter 4) has been applied for plugging term libraries, compilers and front-ends into the Stratego transformation system. Experimentation on higher-level programs, for example, ones based on software modelling approaches involving UML, may be fruitful. Together with the graph extension to Stratego, experiments with combining strategic rewriting with diagrammatic modelling approaches is possible.

The current selection of case studies demonstrates that the proposed techniques are applicable to realistic scenarios both for fully automatic and interactive program transformation. They do not cover the full range of transformations nor the full range of subject languages, however. Additional experiments involving additional subject languages are warranted.

Another area of future investigation is the pursuit of better support for language-independent transformations applicable to extensible subject languages. Consider a non-extensible subject language L. The syntax and semantics of L are known to the transformation developer when a transformation T is adapted to fit L. For an extensible language E, both the syntax and semantics may later be extended by programmers of E. These extensions may require additional cases to be added to T. The question of how these cases may be added requires additional research to answer. This topic is discussed more thoroughly in [Vis05b].

Additional mechanisms for the modularisation and adaptation of transformations is likely to be a fruitful topic of further study. Some experiments with a declarative "view" mechanism, somewhat related to that of Wadler [Wad87], have been conducted in the course of this research. These experiments are based on the concept of (generalised) signature morphisms. Initial experiments are encouragning, but significantly more work (and pages) is required before a conclusion can be reached.

Perfection is achieved on the verge of collapse... so, don't aim for it.

–Nicolae Vintila

14 Conclusions

The main contributions of this dissertation are two novel and complementary techniques for improving language independence of program transformations: program object model adapters for decoupling the transformation engine from the program model and aspects for adapting generic transformation algorithm skeletons to specific subject languages.

Survey of Transformation Systems – A detailed survey of the state-of-the-art in software transformation systems was presented. It employed feature models and concrete examples taken from about a dozen research systems for describing central parts of the design space for transformation system. The survey indicated that good abstraction facilities for the program model are necessary for language independence.

Program Object Model Adapters – Large-scale reuse of transformations and transformation systems across subject language infrastructures is supported by the program object model adapters. They weld together transformation system runtimes with the abstract syntax tree of an existing language infrastructure such as the front-end of a compiler. This is done by on-the-fly translation of rewriting operations in the transformation system to sequences of equivalent method calls on the AST API. This obviates the need for data serialisation and thus enables efficient integration between transformation engines and front-ends. The technique can be applied to most tree-like APIs and is applicable for many term-based rewriting systems.

Aspects – The AspectStratego language offers a declarative mechanism for adding support for subject language families to transformation libraries. This improves the genericity and language-independence of transformations. The aspects also capture many cross-cutting concerns found in transformation programs. Well-defined points in the execution of a transformation program can be identified, parametrised and modified using aspects. By extracting and parametrising execution points, a transformations can be formulated, even retroactively in cases where grey box reuse is accepted.

Strategic Graph Rewriting – The System S rewriting calculus has been extended to handle graphs. Its implementation, the GraphStratego language, provides new language abstractions for capturing graph-like program models while still maintaining

the benefits from strategic rewriting: the separation of rewrite rules from traversal strategies, thus providing a language for strategic graph rewriting.

An Extensible Transformation Language Framework – The MetaStratego framework used to prototype the proposed language extensions is available as a general language prototyping framework. It is provided so that other developers may experiment with new transformation language extensions.

A Flexible Transformation Runtime – The Stratego/J interpreter provides a transformation engine capable of abstracting over term representations. It may easily be plugged into existing language infrastructures and is supported by jsglr, an implementation of a scannerless GLR parser.

The main contributions are supported by several case studies which serve to demonstrate their usefulness. The presented case studies include:

- An implementation of a domain-specific aspect language for alert handling.
- An interactive development environment for program transformations that has served as a test-bed for the techniques and language abstractions presented in this dissertation.
- A collection of examples of framework-specific transformation and analysis, showing that advanced framework developers may benefit from the above contributions since writing custom language processing tools is now fairly simple.
- A generator of executable unit tests from algebraic specifications that demonstrates how the proposed techniques may be used to extend development environments with new, interactive program transformations.

Spoofax – The Spoofax interactive development environment for strategic programming is useful in its own right. It provides a modern and effective editing environment for transformation developers. Scripts, written in Stratego (with any of the extensions discussed above), may be used to extend the development environment.

It's science. I'm not questioning it.

– Joshua Nichols

15 Summary

The motivation for this work has been to develop techniques and tools for expressing reusable, language-independent program analyses and transformations. It has been a goal that transformation programs should be reusable for language families, across language infrastructures and, when possible, across language paradigms. The following summarises the dissertation:

State of the Art First, a survey of existing software transformation systems is presented. The focus of the survey is on transformation languages and architectures. Attention is given to the program models, i.e. the facilities a transformation system has for representing and manipulating programs. The survey indicates that the program model is the central component that needs good abstraction facilities and a good abstract representation if one is to attain transformation reuse and language-independence.

Domain Analysis Next, an analysis of the program model is presented to find and describe the domain objects which must be abstracted over. This analysis suggests that the necessary abstractions behave in rather complicated ways: capturing them in traditional transformation libraries is not the best solution. It might be better to express the abstractions as new language features in the transformation language.

Design Subsequently, the above-noted abstractions are captured as abstract data types, and described using basic algebraic specifications. This formalises the abstractions and the rules governing them. Algebraic specifications work well as requirements for implementing the abstractions in transformation libraries. Algebraic axioms are a good source for deriving both optimisation rules and static checks for correct usage of the abstractions in the transformation library. This proved very useful in the implementation of the prototypes which were constructed for the Stratego transformation language. The abstractions are augmented with a domain-specific notation, making them full-fledged language features in Stratego. This provides a more convenient notation and raises the abstraction level for the programmer.

Implementation Finally, an extensible variant of Stratego called MetaStratego is presented. The prototypes of all the new abstractions were implemented on top of this extended system. The runtime of MetaStratego has some appealing features: it can operate on any tree- or graph-like program model, it executes on the Java Virtual Machine (JVM), it can be easily plugged into interactive development environments written in Java, and it can plug into compiler front-ends running on the JVM.

Proofs of Concept As proofs-of-concept, several prototypes were constructed, including:

- A modern, interactive development environment for (Meta)Stratego.
- A framework for interactive and automatic transformation and analysis of Java code.
- A code generator for axiom-based unit testing.
- A domain-specific aspect language for alerts.

The conclusion that this dissertation presents a significant step towards language independence for program transformation systems. It also indicates that there is more to be done before the goal is finally achieved.

Bibliography

[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Bruno Dufour, Christopher Goard, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, Julian Tibble, and Clark Verbrugge. abc: the AspectBench compiler for AspectJ – a workbench for aspect-oriented programming language and compilers research. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 88–89, New York, NY, USA, 2005. ACM Press.

[AC98] Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 134–143. ACM Press, 1998.

[AG05] David Abrahams and Aleksey Gurtovoy. *C++ Template Meta-Programming*. Addison-Wesley, Boston, MA, USA, 2005.

[AL00] Uwe Assmann and Andreas Ludwig. Aspect weaving with graph rewriting. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 24–36, London, UK, 2000. Springer-Verlag.

[ANMM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Proceedings of OOPLSA'06: Conference on Object-Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2006. ACM Press.

[Art88] Lowell Jay Arthur. *Software evolution: the software maintenance challenge*. Wiley-Interscience, New York, NY, USA, 1988.

[Ass03] U. Assmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., 2003.

[AT01] Paul Anderson and Tim Teitelbaum. Software inspection using CodeSurfer. In *Proceedings of WISE'01 (Itl Workshop on Inspection in Software Engineering)*, 2001.

[Aug93] Lex Augustueijn. *Functional Programming, Program Transformations and Compiler Construction*. PhD thesis, Eindhoven Technical University, The Netherlands, October 1993.

[Bat05] Don S. Batory. Feature models, grammars, and propositional formulas. In J. Henk Obbink and Klaus Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[BBK+05] Anya Helene Bagge, Martin Bravenboer, Karl Trygve Kalleberg, Koen Muilwijk, and Eelco Visser. Adaptive code reuse by aspects, cloning and renaming. Technical Report UU-CS-2005-031, Utrecht University, 2005.

[BDD06] Alexandre Borghi, Valentin David, and Akim Demaille. C-transformers: A framework to write C program transformations. *ACM Crossroads*, 12(3), April 2006.

[BDHK06] Anya Bagge, Valentin David, Magne Haveraaen, and Karl Trygve Kalleberg. Stayin' alert: Moulding failure and exceptions to your needs. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, Portland, Oregon, October 2006. ACM Press.

[Bec98] Kent Beck. Extreme programming: A humanistic discipline of software development. In *FASE*, pages 1–6, 1998.

[Bec02] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2002.

[BEG+87] H P Barendregt, M C J D Eekelen, J R W Glauert, J R Kennaway, M J Plasmeijer, and M R Sleep. Term graph rewriting. In *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*, pages 141–158, London, UK, 1987. Springer-Verlag.

[Ber05] Clara Bertolissi. *The graph rewriting calculus: properties and expressive capabilities*. Thèse de doctorat, Institut National Polytechnique Lorrain - INPL, October 2005.

[BG] Kent Beck and Erich Gamma. JUnit – Java Unit testing. `http://www.junit.org` and `http://junit.sourceforge.net/` per 2007-03-15.

[BK06] Anya Helene Bagge and Karl Trygve Kalleberg. DSAL = library+notation: Program transformation for domain-specific aspect languages. In *Proceedings of the Domain-Specific Aspect Languages Workshop*, October 2006.

[BKHV03] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In Dave Binkley and Paolo Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[BKK+04] P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. *Electronic Notes in Theoretical Computer Science*, 4(1):35–50(15), Jan 2004.

[BKVV05] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. *Stratego/XT Tutorial, Examples, and Reference Manual*. Department of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, August 2005. (Draft).

[BKVV06] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.16: Components for transformation systems. In Frank Tip and John Hatcliff, editors, *PEPM'06: Workshop on Partial Evaluation and Program Manipulation*. ACM Press, January 2006.

[BKVV07] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.16. a language and toolset for program transformation. *Science of Computer Programming*, 2007. Accepted for publication.

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 143, Washington, DC, USA, 1998. IEEE Computer Society.

[BM91] Jean-Pierre Banâtre and Daniel Le Métayer. Introduction to Gamma. In *Research Directions in High-Level Parallel Programming Languages*, pages 197–202, 1991.

[BM93] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Commun. ACM*, 36(1):98–111, 1993.

[BMV02] Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages with logic metaprogramming. In *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative*

Programming and Component Engineering, pages 110–127, London, UK, 2002. Springer-Verlag.

[Bos98] Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.

[BP] Jean Bovet and Terence Parr. ANTLRWorks: The ANTLR GUI development environment. Home page at www.antlr.org/works/ (visited 2007-04-15).

[BPM04] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program transformations for practical scalable software evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society.

[BPSM+] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. World-Wide Web Consortium.

[BV00] M.G.J. van den Brand and J.J. Vinju. Rewriting with layout. In Claude Kirchner and Nachum Dershowitz, editors, *Proceedings of RULE'00*, 2000.

[BV04] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[BvDOV06] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178, 2006.

[CBE+00] Constantinos A. Constantinides, Atef Bader, Tzilla H. Elrad, P. Netinant, and Mohamed E. Fayad. Designing an aspect-oriented framework in an object-oriented environment. *ACM Comput. Surv.*, 32(1es):41, 2000.

[CC02] Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on*

Principles of programming languages, pages 178–190, New York, NY, USA, 2002. ACM Press.

[CCH05] James R. Cordy, Ian H. Carmichael, and Russell Halliday. *The TXL Programming Language, Version 8*, January 2005.

[CDE+03] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The Maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[CDK+01] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[CFG96] P. Ciancarini, D. Fogli, and M. Gaspari. A Logic Language based on Gamma-like Multiset Rewriting. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proc. 5th Workshop on Extensions of Logic Programming*, volume 1050, pages 83–102, Leipzig, Germany, 1996. Springer-Verlag, Berlin.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM Press.

[CHHP91] James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. TXL: a rapid prototyping system for programming language dialects. *Comput. Lang.*, 16(1):97–107, 1991.

[Chi95] Shigeru Chiba. A metaobject protocol for C++. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, New York, NY, USA, 1995. ACM Press.

[CKK06] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. Feature models are views on ontologies. In *Proceedings of the 10th International Software Product Line Conference (SPLC'06)*, pages 41–51, Washington, DC, USA, August 2006. IEEE Computer Society.

[Cla99] James Clark. XSL transformations (XSLT) version 1.0. W3C recommendation, W3C, November 1999.

[CLR97] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1997.

[CNFP06] Paolina Centonze, Gleb Naumovich, Stephen J. Fink, and Marco Pistoia. Role-based access control consistency validation. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 121–132, New York, NY, USA, 2006. ACM Press.

[CNR90] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Trans. Softw. Eng.*, 16(3):325–334, 1990.

[Cob] Cobertura. `http://cobertura.sourceforge.net` per 2007-03-15.

[Coh81] Paul M Cohn. *Universal Algebra*. Springer, 1981.

[Cop05] Tom Copeland. *PMD Applied*. Centennial Books, Nov 2005.

[Cor04] James R. Cordy. TXL - a language for programming language tools and applications. *ENTCS*, 110:3–31, 2004.

[CS99] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *ASIAN '99: Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science*, pages 62–73, London, UK, 1999. Springer-Verlag.

[DHB92] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp Symb. Comput.*, 5(4):295–326, 1992.

[DM03] Kyung-Goo Doh and Peter D. Mosses. Composing programming languages by combining action-semantics modules. *Sci. Comput. Program.*, 47(1):3–36, 2003.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[DR97] Dale Dougherty and Arnold Robbins. *sed & awk (2nd Edition)*. O'Reilly Media, Inc., Mar 1997.

[DT96] Merlin Dorfman and Richard H. Thayer. *Software Engineering*. 1996.

[Ecl] Eclipse. `http://www.eclipse.org` per 2007-03-15.

[Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Pro*, pages 17–23, May/June 2000.

[Fea87] M. S. Feather. A survey and classification of some program transformation approaches and techniques. In *The IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation*, pages 165–195, Amsterdam, The Netherlands, 1987. North-Holland Publishing Co.

[FH83] R. Fjeldstad and W. Hamlen. Application program maintenance-report to our respondents. In *Tutorial on Software Maintenance*, pages 13–27. IEEE Press, 1983.

[GA95] Pascale Le Gall and Agnès Arnould. Formal specifications and test: Correctness and oracle. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *COMPASS/ADT*, volume 1130 of *Lecture Notes in Computer Science*, pages 342–358. Springer, 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[GL00] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. *SIGPLAN Not.*, 35(1):39–52, 2000.

[GMH81] John D. Gannon, Paul R. McMullin, and Richard G. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981.

[GR04] Jeff Gray and Suman Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 36–45, New York, NY, USA, 2004. ACM Press.

[Gru93] Thomas R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. Technical Report KSL93-04, Stanford University, Stanford, August 1993.

[HB05] Magne Haveraaen and Enida Brkic. Structured testing in Sophus. In Eivind Coward, editor, *Norsk informatikkonferanse NIK'2005*, pages 43–54. Tapir akademisk forlag, Trondheim, Norway, 2005.

[Hed98] Görel Hedin. Language support for design patterns using attribute extension. In *ECOOP '97: Proceedings of the Workshops on Object-Oriented Technology*, pages 137–140. Springer-Verlag, 1998.

[Hir03] Robert Hirschfeld. Aspects - aspect-oriented programming with Squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232. Springer-Verlag, 2003.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[HM03] Görel Hedin and Eva Magnusson. JastAdd – an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, 2003.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[HS96] Merlin Hughes and David Stotts. Daistish: systematic algebraic testing for OO programs in the presence of side-effects. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 53–61, New York, NY, USA, 1996. ACM Press.

[HT05] Akinori Yonezawa Hideaki Tatsuzawa, Hidehiko Masuhara. Aspectual Caml: an aspect-oriented functional language. In Curtis Clifton, Ralf Lämmel, , and Gary T. Leavens, editors, *FOAL 2005 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2005*, Chicago, IL, March 2005.

[HVdMdV05] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris de Volder. CodeQuest: querying source code with Datalog. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 102–103, New York, NY, USA, 2005. ACM Press.

[HWS00] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Toward a standard exchange format. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 162, Washington, DC, USA, 2000. IEEE Computer Society.

[JAM99] Paul Jansen, Lex Augusteijn, and Harm Monk. An introduction to Elegant. Technical report, Philips Research Laboratories, Eindhoven, The Netherlands, 1999.

[Jav] Java 5.0 api. `http://java.sun.com/j2se/1.5.0/docs/api/` per 2007-03-15.

[Joh78] S. Johnson. *Lint, a C Program Checker*. AT&T Bell Laboratories, 1978.

[Jon] Joel Jones. Abstract syntax tree implementation idioms. In Brian Marick, editor, *Proceedings of The 10th Conference on Pattern Languages of Programs (PLOP'03)*. Hillside Group. Online publication.

[JV01] Patricia Johann and Eelco Visser. Fusing logic and control with local transformations: An example optimization. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.

[JV03] Patricia Johann and Eelco Visser. Strategies for fusing logic and control via local, application-specific transformations. Technical Report UU-CS-2003-050, Institute of Information and Computing Sciences, Utrecht University, February 2003.

[Kah99] Wolfram Kahl. The term graph programming system HOPS. In Rudolf Berghammer and Yassine Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 136–149, Wien, March 1999. Springer. ISBN: 3-211-83282-3.

[Kal03] Karl Trygve Kalleberg. User-configurable, high-level transformations with CodeBoost. Master's thesis, University of Bergen, P.O.Box 7800, N-5020 Bergen, Norway, March 2003.

[Kal06] Karl Trygve Kalleberg. Stratego: A programming language for program manipulation. *ACM Crossroads*, 12(3), April 2006.

[KD01] Wolfram Kahl and Frank Derichhsweiler. Declarative term graph attribution for program generation. *J. UCS*, 7(1):54–70, 2001.

[KH] Karl Trygve Kalleberg and Magne Haveraaen. JAxT – Java Axiomatic Testing. `http://www.ii.uib.no/mouldable/testing` per 2007-03-15.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and Willian G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001: Object-Oriented Programming:*

15th European Conference, volume 2072 of *LNCS*, pages 327–353. Springer-Verlag, June 2001.

[KK04] Günter Kniesel and Helge Koch. Static composition of refactorings. *Sci. Comput. Program.*, 52(1-3):9–51, 2004.

[KL03] J. Kort and R. Lämmel. Parse-Tree Annotations Meet Re-Engineering Concerns. In *Proc. Source Code Analysis and Manipulation (SCAM'03)*. IEEE Computer Society Press, September 2003.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[KP97] Philipp W. Kutter and Alfonso Pierantonio. Montages specifications of realistic programming languages. *J. UCS*, 3(5):416–442, 1997.

[KV05] Karl Trygve Kalleberg and Eelco Visser. Combining aspect-oriented and strategic programming. In Horatiu Cirstea and Narciso Marti-Oliet, editors, *Workshop on Rule-Based Programming (RULE'05)*, Electronic Notes in Theoretical Computer Science, Nara, Japan, April 2005. Elsevier Science Publishers.

[KV06] Karl Trygve Kalleberg and Eelco Visser. Strategic graph rewriting: Transforming and traversing terms with references. In *Proceedings of the 6th International Workshop on Reduction Strategies in Rewriting and Programming*, Seattle, Washington, August 2006. Online publication.

[KV07a] Karl Trygve Kalleberg and Eelco Visser. Fusing a transformation language with an open compiler. In Tony Sloane and A. Johnstone, editors, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA'07)*, ENTCS, pages 18–31, Braga, Portugal, March 2007. Elsevier.

[KV07b] Karl Trygve Kalleberg and Eelco Visser. Spoofax: An interactive development environment for program transformation with Stratego/XT. In Tony Sloane and Adrian Johnstone, editors, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA'07)*, ENTCS, pages 47–50, Braga, Portugal, March 2007. Elsevier.

[Läm99] Ralf Lämmel. Declarative aspect-oriented programming. In Olivier Danvy, editor, *Proceedings PEPM'99, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM'99, San Antonio (Texas), BRICS Notes Series NS-99-1*, pages 131–146, January 1999.

[Läm03] Ralf Lämmel. Typed Generic Traversal With Term Rewriting Strategies. *Journal of Logic and Algebraic Programming*, 54, 2003. Also available as arXiv technical report cs.PL/0205018.

[Lin05] Christian Lindig. Random testing of C calling conventions. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 3–12, New York, NY, USA, 2005. ACM Press.

[LK97] C.V Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Xerox Palo Alto Research Center, 1997.

[LL00] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. *ICSE*, 00:418, 2000.

[LVV03] Ralf Lämmel, Eelco Visser, and Joost Visser. Strategic programming meets adaptive programming. In *Proceedings of Aspect-Oriented Software Development (AOSD'03)*, pages 168–177, Boston, USA, March 2003. ACM Press.

[McK84] J McKee. Maintenance as a function of design. In *Proceedings of the AFIPS National Computer Conference*, pages 187–193, 1984.

[MdVHC03] C Mundie, P de Vries, P Haynes, and M Corwine. Trustworthy computing. Technical report, 2003. White paper.

[MGR05] Pierre-Etienne Moreau, Julien Guyon, and Antoine Reilles. *Tom User's Guide*. Lorraine Laboratory of IT Research and its Applications, Nancy, France, Jul 2005.

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA '05: Proc. of the ACM SIGPLAN Conf. Object oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM Press.

[MM95] J. Meseguer and N. Martí-Oliet. From abstract data types to logical frameworks. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification: 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop*, volume 906 of *Lecture Notes in Computer Science*, pages 48–80. Springer, 1995.

[MO03] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99. ACM Press, 2003.

[Moa90] J. Moad. Maintaining the competitive edge. *Datamation*, 36(4):61–62,62,66, Feb 1990.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[Mor98] Pierre-Etienne Moreau. A choice-point library for backtrack programming. In Konstantinos Sagonas, editor, *Proceedings of the JICSLP-98 Workshop on Implementation Technologies for Programming Languages based on Logic*, pages 16–31, 1998.

[Mos04a] Peter D. Mosses. Modular language descriptions. In Gabor Karsai and Eelco Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, page 489. Springer, 2004.

[Mos04b] Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[MRB+] MohammadReza Mousavi, Michel Reniers, Twan Basten, Michel Chaudron, Giovanni Russello, Angelo Cursaro, Sandeep Shukla, Rajesh Gupta, and Douglas C. Schmidt. Using Aspect-GAMMA in the design of embedded systems. In *Proceedings of Seventh Annual IEEE International Workshop on High Level Design Validation and Test*, pages 69–74, Cannes, France. IEEE Computer Society Press, Los Alamitos, CA, USA, 2002.

[MRV03] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In *12th International Conference on Compiler Construction*, LNCS, pages 61–76. Springer, 2003.

[mtj06] Matrix Toolkits for Java. http://rs.cipr.uib.no/mtj/, 2006.

[MWT94] H. Müller, K. Wong, and S. Tilley. Understanding software systems using reverse engineering technology. In *The 62nd Congress of L'Association Canadienne Francaise pour L'Avancement des Sciences Proceedings (ACFAS)*, volume 4, pages 41–48, 1994.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, LNCS, pages 138–152. Springer, 2003.

[NNZ04] U. Nickel, J. Niere, and A. Zundorf. Tool demonstration: The FUJABA environment. In *The 22nd International Conference on Software Engineering (ICSE)*. ACM Press, 2004.

[NoU] NoUnit. `http://nounit.sourceforge.net` per 2007-03-15.

[Opd92] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, Champaign, IL, USA, 1992.

[OV05] Karina Olmos and Eelco Visser. Composing source-to-source dataflow transformations with rewriting strategies and dependent dynamic rewrite rules. In Rastislav Bodik, editor, *CC'05: 14th International Conference on Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 204–220. Springer-Verlag, April 2005.

[Pag93] Robert Page. APTS external specification manual (rough draft). Available at http://www.cs.nyu.edu/jessie/, 1993. Unpublished manuscript.

[Pai96] Robert Paige. Future directions in program transformations. *ACM Comput. Surv.*, 28(4es):170, 1996.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[Par86] H. Partsch. Transformational program development in a particular problem domain. *Science of Computer Programming*, 7(2):99–241, 1986.

[Pfl05] Shari Lawrence Pfleeger. *Software engineering: theory and practice (3rd ed)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[PK82] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, 1982.

[Plu99] D. Plump. *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*, chapter Term graph rewriting, pages 3–61. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[Plu01] Detlef Plump. Essentials of term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 51, 2001.

[PP96] Alberto Pettorossi and Maurizio Proietti. Future directions in program transformation. *ACM Comput. Surv.*, 28(4es):171, 1996.

[PQ95] T. J. Parr and R. W. Quong. ANTLR: a predicated-LL(k) parser generator. *Softw. Pract. Exper.*, 25(7):789–810, 1995.

[PS83] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, 1983.

[PvE98] M. Plasmeijer and M. van Eekelen. Language report: Concurrent Clean. Technical Report CSI-R9816, Computing Science Inst., U. of Nijmegen, Nijmegen, The Netherlands, 1998.

[Sch04] Andreas Schürr. *The PROGRES Language Manual Version 9.x*. Lehrstuhl für Informatik III, RWTH Aachen, Aachen, Germany, 2004.

[sgm86] *ISO 8879:1986 – Standard Generalized Markup Language*. International Organisation for Standardization (ISO), 1986.

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific*, pages 53–60. Australian Computer Society, Inc., 2002.

[SLA02] P. David Stotts, Mark Lindsey, and Angus Antley. An informal formal method for systematic junit test case generation. In Don Wells and Laurie A. Williams, editors, *XP/Agile Universe*, volume 2418 of *Lecture Notes in Computer Science*, pages 131–143. Springer, 2002.

[SLS03] Macneil Shonle, Karl Lieberherr, and Ankit Shah. XAspects: an extensible system for domain-specific aspect languages. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 28–37, New York, NY, USA, 2003. ACM Press.

[SLTM91] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. MetaEdit: a flexible graphical environment for methodology modelling. In *CAiSE '91: Proceedings of the third international conference on Advanced information systems engineering*, pages 168–193, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[SNDH04] Rozalia Szabo-Nacsa, Peter Divianszky, and Zoltan Horvath. Prototype environment for refactoring Clean programs. In *Proceedings of the Sixth International Conference on Applied Informatics (ICAI'04)*, 2004.

[Som00] I. Sommerville. *Software Engineering (6th Edition)*. Addison-Wesley, 2000.

[ST88] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.

[SY86] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[SY93] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Trans. Softw. Eng.*, 19(5):478–485, 1993.

[Tae04] Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *Proceedings of the 2nd International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE 2003)*, volume 3062 of *Lecture Notes in Computer Science*, pages 446 – 453. Springer, Jan 2004.

[TCIK00] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava: A class-based macro system for java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 117–133, London, UK, 2000. Springer-Verlag.

[Ter03] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[TL02] K. C. Tai and Y. Lie. A test generation strategy for pairwise testing. *IEEE Trans. Softw. Eng.*, 28(1):109–111, 2002.

[Ulr90] W. Ulrich. The evolutionary growth of software engineering and the decade ahead. *American Programmer*, 3(10):12–20, 1990.

[VB98] Eelco Visser and Zine-el-Abidine Benaissa. A core language for rewriting. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*, Pont-à-Mousson, France, September 1998. Elsevier Science Publishers.

[VBT98] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

[vdBdJKO00] M. G. T. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Softw. Pract. Exper.*, 30(3):259–291, 2000.

[vdBHKO02] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[vdBV00] M.G.J. van den Brand and J.J. Vinju. Rewriting with layout. In Claude Kirchner and Nachum Dershowitz, editors, *Proceedings of the Second International Workshop on Rule-based Programming (RULE'00)*, 2000.

[vdBvDH+01] Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF meta-environment: A component-based language development environment. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 365–370, London, UK, 2001. Springer-Verlag.

[vDKT93] Arie van Deursen, Paul Klint, and Frank Tip. Origin tracking. *J. Symb. Comput.*, 15(5/6):523–545, 1993.

[VEdM06] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a scripting language for refactoring. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 172–181, New York, NY, USA, 2006. ACM Press.

[vEM02] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE Computer Society Press, October 2002.

[VG98] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, Philadelphia, PA, USA, 1998. SIAM.

[Vis97] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[Vis99] Eelco Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, Trento, Italy, July 1999. Springer-Verlag.

[Vis02] Eelco Visser. Meta-programming with concrete object syntax. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[Vis05a] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.

[Vis05b] Eelco Visser. Transformations for abstractions. In Jens Krinke and Giulio Antoniol, editors, *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 3–12, Budapest, Hungary, October 2005. IEEE Computer Society Press. (Keynote paper).

[vV02] Arie van Deursen and Joost Visser. Building program understanding tools using visitor combinators. In *Proceedings 10th Int. Workshop on Program Comprehension, IWPC 2002*, pages 137–146. IEEE Computer Society, 2002.

[vWV03] Jonne van Wijngaarden and Eelco Visser. Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems. Technical

Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University, 2003.

[Wad87] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313, New York, NY, USA, 1987. ACM Press.

[Wad92] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM Press.

[WAKS97] Daniel C. Wang, Andrew W. Appel, Je L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 213–228, 1997.

[War89] M. Ward. Proving program refinements and transformations. Dphil thesis, Oxford University, 1989.

[War99] Martin Ward. Assembler to C migration using the FermaT transformation system. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 67, Washington, DC, USA, 1999. IEEE Computer Society.

[War02] M. Ward. Program slicing via FermaT transformations. In *COMPSAC'02: 26th Annual International Computer Software and Applications Conference*, Los Alamitos, California, USA, August 2002. IEEE Computer Society Press.

[WFW+94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.

[Win99] Victor L. Winter. An overview of HATS: A language independent High Assurance Transformation System. In *ASSET '99: Proceedings of the 1999 IEEE Symposium on Application - Specific Systems and Software Engineering and Technology*, page 222, Washington, DC, USA, 1999. IEEE Computer Society.

[Win03] Thomas (Rho) Windeln. LogicAJ – eine erweiterung von AspectJ um logische meta-programmierung. Master's thesis, University of Bonn, Bonn, Germany, 2003.

[ZSJG79] M. Zelkowitz, A. Shaw, and J J. Gannon. *Principles of Software Engineering and Design*. Prentice-Hall, 1979.