THE UNIVERSITY OF CHICAGO

REFLECTIVE TECHNIQUES IN EXTENSIBLE LANGUAGES

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
JONATHAN RIEHL

CHICAGO, ILLINOIS
AUGUST 2008

To Leon Schram

ABSTRACT

An extensible programming language allows programmers to use the language to modify one or more of the language's syntactic, static-semantic, and/or dynamic-semantic properties. This dissertation presents Mython, a variant of the Python language, which affords extensibility of all three language properties. Mython achieves extensibility through a synthesis of reflection, staging, and compile-time evaluation. This synthesis allows language embedding, language evolution, domain-specific optimization, and tool development to be performed in the Mython language. This work argues that using language-development tools from inside an extensible language is preferable to using external tools. The included case study shows that users of an embedded differential equation language are able to work with both the embedded language and embedded programs in an interactive fashion — simplifying their work flow, and the task of specifying the embedded language.

ACKNOWLEDGMENTS

In keeping with my dedication, I'd like to begin by thanking Leon Schram, my A.P. computer science teacher. Mr. Schram took a boy who liked to play with computers, and made him a young man that could apply reason to the decomposition and solution of problems. In turn, I'd like to thank Charlie Fly and Robin Friedrich for helping extend that reason to encompass large scale systems and language systems. Thanks to David Beazley for bringing me back into academia and allowing me to continue to pursue the research agenda Charlie fathered. I am grateful both to and for my advisor, John Reppy. He has not only helped me through the later half of my graduate studies, but challenged me to expand the scope of my agenda and the depth of my understanding far beyond their original bounds.

My dissertation committee, Robert Bruce Findler, Ridgway Scott, and Matthew Knepley, have all been helpful and supportive of my work. I thank them for their time, attention, and guidance.

A good portion of this work has built upon contributions by both the Python and Stratego communities. Thanks to Guido van Rossum, the inventor of the Python language, as well as the Python and PyPy communities. I have benefited from discussions I've had with the Stratego community, specifically Eelco Visser, Martin Bravenboer, and Rob Vermaas.

The case study presented in this dissertation is built on work by the FEniCS community. Andy Terrel and Peter Brune have been capable guides into the finite element method and scientific computing.

I would not have been able to accomplish this without the strong moral support of my friends and family. Specifically, I'd like to thank Sarah Kaiser, Katrina Riehl, Jacob Matthews, and John Overton.

Finally, I'd like to thank the people of Earth, because they (mostly) help me.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGMENTS iv

LIST OF FIGURES ix

LIST OF TABLES xi

Chapter

1 INTRODUCTION 1
- 1.1 Some Programming Language Implementation Terminology 3
- 1.2 Some Programming Language Implementation Scenarios 6
  - 1.2.1 Language Evolution 6
  - 1.2.2 Language Embedding 8
  - 1.2.3 Domain-specific Optimizations 11
  - 1.2.4 Language Tools 13
  - 1.2.5 Other Possible Scenarios 14
- 1.3 Language Extensibility is Program Extensibility 15
  - 1.3.1 Reflection Implies Extensibility 15
  - 1.3.2 Glue Languages and Program Extensibility 17
  - 1.3.3 Superglue Languages and Language Extensibility 19
- 1.4 Increasing Extensibility in Languages 20
  - 1.4.1 Components in Language Implementations 20
  - 1.4.2 Syntactic Extensibility 21
  - 1.4.3 Semantic Extensibility 23
- 1.5 Overview of the Dissertation 26

2 THE SELF-ASSIMILATION OF STRATEGO/XT 28
2.1 Stratego and MetaBorg 28
2.1.1 Composable Syntax Definition in SDF 28
2.1.2 Program Transformation in Stratego 32
2.1.3 MetaBorg and Language Assimilation 33
2.2 Embedding Language Embedding Tools (or, is resistance really futile?) 34
2.2.1 Assimilating MetaBorg in Another Language 34
2.2.2 Making Assimilation a Language Feature 37
2.2.3 Generalizing Implementation Hooks for Fun and DSOs 39
2.2.4 Staged, Multilanguage Programming 41
2.3 Preliminary Experiences 42
2.4 Conclusions 44

3 THE MYTHON LANGUAGE 45
3.1 Language Origins 47
3.1.1 Staged, Multilanguage Programming, Redux 47
3.1.2 Staged, Multilanguage Programming, Fixed 49
3.1.3 Domain-Specific Optimizations 51
3.1.4 Python 53
3.2 Language Syntax 54
3.2.1 Lexical extensions 55
3.2.2 Concrete syntactic extensions 56
3.2.3 Abstract syntax extensions 57
3.3 Language Semantics 58
3.3.1 Compile-time computations 59
3.3.2 Compile-time environments 61
3.4 Implementation 62
3.4.1 Mython Standard Library 62
3.4.2 MyFront 66
3.5 Applications 67
3.5.1 Domain-specific Optimization Basics 67
3.5.2 Shortcut Deforestation 69
3.5.3 High-level Operators 72
3.6 Summary of Mython 72

4 DOMAIN-SPECIFIC OPTIMIZATIONS 74
4.1 Motivations for Domain-Specific Optimizations 74
4.1.1 The Cost of Domain Abstraction 74
4.1.2 Strength Reduction 75
4.1.3 Shortcut Deforestation 76
4.2 User Specification of Domain-Specific Optimizations 78
4.3 Rewriting Methods for Domain-Specific Optimization 78
4.3.1 Greedy Match and Replace 80
4.3.2 Bottom-up Rewriting 82
4.3.3 Tiered Rewriting 87
4.3.4 Rewrite Strategies 89
4.4 Quantitative Analysis 92
4.4.1 An Analysis Framework 92
4.4.2 Rewrite Performance 93
4.4.3 Rewrite Depth 96
4.5 Implementations 98
4.5.1 Implementation in GHC 98
4.5.2 Implementation in Manticore 99
4.5.3 Implementation in Mython 99
4.6 Conclusions 100

5 CASE STUDY: AN EMBEDDED DOMAIN-SPECIFIC LANGUAGE . . 102
5.1 Introduction 102
5.2 Background 103
5.2.1 Automating Scientific Computing 103
5.2.2 The Finite Element Method 105
5.3 The MyFEM Domain-specific Language 106
5.3.1 MyFEM Syntax 107
5.3.2 Types in MyFEM 109
5.3.3 MyFEM Semantics 113
5.4 Implementing MyFEM 116
5.4.1 Methodology 117
5.4.2 The Front-end 119
5.4.3 The Back-end 120
5.5 Results and Conclusions 121
5.5.1 A Comparison to Other FEniCS Front-ends 121
5.5.2 Future Work 121
5.5.3 Conclusion 122

6 RELATED WORK 123
6.1 Related Interfaces 123
6.1.1 Compiler Tools 123
6.1.2 Operator Overloading 125
6.1.3 Syntax Macros 126
6.1.4 Compiler Pragmas 128
6.2 Related Methods 129
6.2.1 Staging 129
6.2.2 Extensible Compilers 129
6.3 Extensible Languages 129
6.3.1 Caml p4 129
6.3.2 Converge 130
6.3.3 eXTensible C 130
6.3.4 F-sub 130
6.3.5 FLEX 131
6.3.6 Fortress 131
6.3.7 Lisp and Scheme 132
6.3.8 OMeta/COLA 132
6.3.9 Perl 6 132
6.3.10 Template Haskell 132

7 CONCLUSIONS 134

Appendix

A REWRITE DATA 137

B MYFEM DEFINITIONS 139
B.1 MyFEM Concrete Syntax 139
B.2 MyFEM Intermediate Language 141

REFERENCES 143

LIST OF FIGURES

1.1 A traditional method of language implementation. 2
1.2 Example of an interpreted program or subprogram using T-notation. 4
1.3 A hybrid implementation of a high-level language. 5
1.4 Toppling the reflective tower. 16
1.5 A traditional method of program extension. 18

2.1 The Stratego approach to language extension. 29
2.2 A host grammar, an extension grammar, and a composition grammar. 31
2.3 An example of multilanguage staging. 41

3.1 The Mython approach to language extension. 46
3.2 A formalization of rewrite specifications. 52
3.3 Example of a Mython program. 54
3.4 Abstract syntax for the Mython example. 55
3.5 Part of the lexical stream for the Mython example. 56
3.6 Transformation functions from Mython concrete syntax to abstract syntax. 58
3.7 Example of how the compilation environment flows through a Mython program. 61
3.8 Adding an optimization phase to MyFront. 68
3.9 The `simple_rw_opt()` function. 69
3.10 An example of using embedded DSL's in Mython to define an optimization. 71

4.1 Algebraic simplifications for the Pan DSL. 76
4.2 Shortcut deforestation for `map/map`. 77
4.3 Example rewrites for shortcut deforestation of string concatenation. 80
4.4 Example of top-down versus bottom-up traversal for greedy rewriting. 81
4.5 A relation on rewrites. 87
4.6 Relationships for partitioning rewrites into tiers. 89
4.7 A greedy rewriter in Stratego. 91
4.8 Rewrite times, in microseconds (μs), given matching input terms. . 94
4.9 Rewrite times, in microseconds (μs), given non-matching input terms. 94

5.1 Example of a MyFEM code generation program. 103
5.2 Structure of the MyFEM type environment. 110
5.3 A partial set of type rules for MyFEM. 111
5.4 Example type derivation for a MyFEM program. 112

5.5 MyFEM target pseudocode. 114

7.1 Example of an x86 assembly function. 135
7.2 Example of embedding machine code in Python. 136

LIST OF TABLES

4.1 Strategy run-times, broken into translation times. 95
4.2 Number of times underlying rewrite algorithm was applied, by optimizer. 96
4.3 Size of the output terms for input terms purely in the rewrite domain, in number of constructor terms, by optimizer and input term associativity. 97
4.4 Counts of string constructors for input terms in the rewrite domain, in number of string constructor terms, by optimizer and input term associativity. 98

5.1 MyFEM operator precedence. 108
5.2 Abstract syntax for MyFEM operators. 109

A.1 Rewrite times, in microseconds (μs), for matching term inputs (with slowdown relative to the greedy optimizer in parenthesis). 137
A.2 Rewrite times, in microseconds (μs), for non-matching input terms (with slowdown relative to the greedy optimizer in parenthesis). . . 137
A.3 Rewrite times, in microseconds (μs), for greedy and strategy based greedy optimizers (with slowdown relative to the greedy optimizer in parenthesis). 138

CHAPTER 1
INTRODUCTION

Traditional approaches to programming-language implementation fix many syntactic and semantic features of a language before it can be used. Many language implementors continue to use a static compilation environment, where the majority of language features are constrained at compiler-compile time. This environment is illustrated in Figure 1.1. In this method of language development, one or more language implementors create source code that describes fixed features of the intended language. The source code is input to a compiler-compiler tool-chain that creates a language implementation. The resulting implementation becomes part of a second tool-chain that accepts user source code and outputs a program.

These traditional approaches have the benefit of providing stakeholders with custom environments that cater to their needs. The implementors are able to use a suite of both domain-specific languages (DSLs) and general-purpose languages. This mixture affords the implementors flexibility in how their language and its features are specified. Language users are presented the corresponding implementation as a program, and are not bothered with its methods or details.

This separation of concerns breaks down at the point where the language users wish to customize programming-language features. Users may want to customize the language for many purposes, but most often they want to change a language to better suit their problem domain. It is not practical for language implementors to anticipate all user needs, and in some cases user needs may conflict with the needs of others or reduce the generality of the language. In the strictest interpretation of the methods presented above, language users are not permitted to make their own changes because they are not expected to require access to the language's source code. Even in cases where the language's source is available, it may not be practical for users to acquire the domain expertise necessary to understand and modify the language.

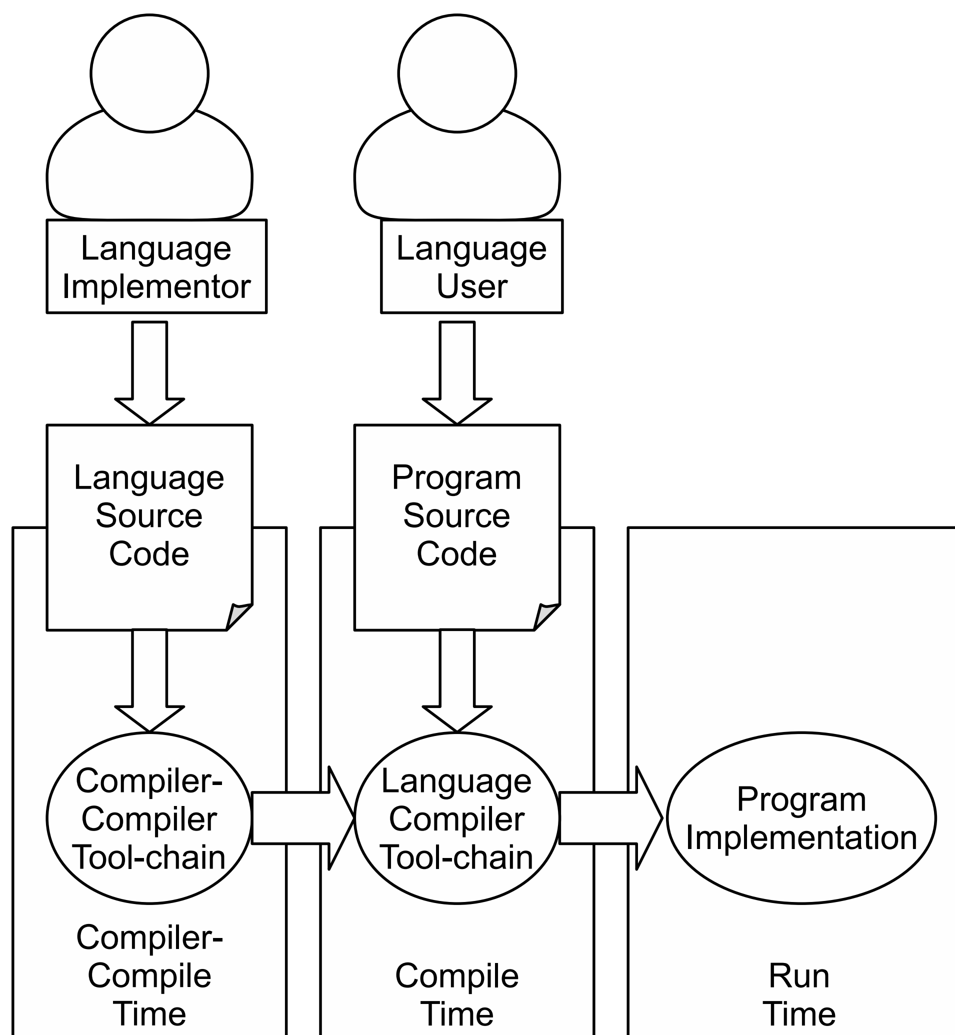


Figure 1.1: A traditional method of language implementation.

A solution to this problem is to make the language extensible. An implementor makes a language extensible by providing the means for users to change the language's features. A language's features can include some or all of its syntactic and semantic properties. Language implementors can maintain the original separation between language modification and language use by ensuring the extensibility features are optional. A feature is optional when the use of any other language feature does not require the use of the optional feature. This means that the resulting language is optionally extensible, and can still be used as before without requiring users to learn how to use the extension features.

This dissertation argues that making a language extensible is practical and accelerates user work flow when dealing with common development scenarios. It shows how language extensibility can be added to a language using existing approaches to language embedding. It then extends the Python programming language with a set of features that afford user extensibility, resulting in a variant language called Mython. Also, it describes methods for using extensibility to implement domain-specific optimizations. Finally, it gives a case study in how the extensibility features of the Mython language improve the usability and performance of an embedded partial-differential equation language.

The remainder of this chapter expands upon the initial framework described above, showing how traditional methods slow down language evolution, language embedding, domain-specific optimization, and tool development. It then presents the foundational ideas behind language extensibility. This chapter concludes by describing the organization of the remaining chapters.

1.1 Some Programming Language Implementation Terminology

In their book on partial evaluation, Jones, Gomard, and Sestoft provide a graphical notation for illustrating language implementations [33]. The following section generalizes Jones, Gomard, and Sestoft's notation, which this dissertation calls T-notation.

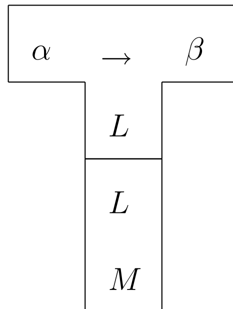


Figure 1.2: Example of an interpreted program or subprogram using T-notation.

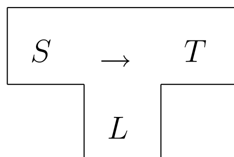
Later chapters and sections use T-notation to illustrate the architecture of various metaprogramming systems and language implementations.

Figure 1.2 shows two programs: a program that is written in a high-level language, L, and a program that interprets that high-level language. The top "T" shape represents a program written in the implementation language, L. This diagram generalizes the input type and output type of the program, using α for the program input, and β for the program output. The lower "I" shape represents a program that interprets the implementation language L. This interpreter is in turn implemented in machine code, M. The machine code is the native encoding of a program for either a physical or virtual computer.

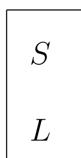
A compiler or translator is a special case of the above program abstraction, where both the input and output types are code. Both Jones et al. and this work use the following terms for compiler and translator programs:

- The source language, S, is the language input into the compiler or translator.
- The target language, T, is the language output by the compiler or translator. If the target language is machine code, $T = M$, the program or subprogram is a compiler. Otherwise, if the target language is the source code for some other language implementation, $T = S'$, the program or subprogram is a translator.
- The definition of implementation language, L, is unchanged from the generalized version, above.

The following diagram plugs the variables used above into the original T-diagram:



The diagram and definitions for an interpreter implementation reuses the compiler and translator terminology, but lacks a target language. Therefore, an interpreter for source code S that is implemented in language L appears as follows:



Many language implementations, such as CPython[13] and UCSD Pascal[78], use a "hybrid" implementation that has features of both compilation and interpretation. Figure 1.3 shows how a hybrid language implementation expands the programs originally from Figure 1.2 into several compilation passes.

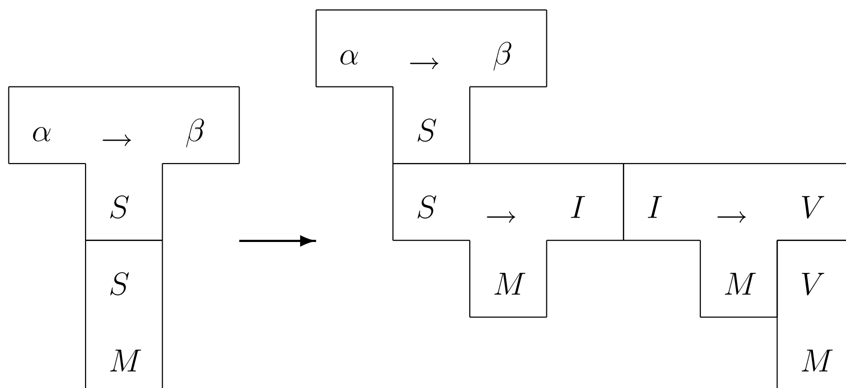


Figure 1.3: A hybrid implementation of a high-level language.

The first pass implements the language front-end. The front-end is a compiled subprogram that translates from source code, S, to an intermediate language, I. The compiler feeds the front-end's output to the implementation's back-end. The back-end compiles the intermediate language, I, into virtual machine code, V. The

virtual machine implementation makes this implementation a hybrid of compilation and interpretation. While the input program is compiled, the virtual machine implementation is an interpreter. In this instance, the interpreter is a machine language program, M, that interprets virtual machine code, V.

1.2 Some Programming Language Implementation Scenarios

This section looks at four scenarios where the status quo slows down users: language evolution, language embedding, domain-specific optimization, and finally language tool development. Each of these scenarios are given their own subsection below. These scenarios are important because they serve as the motivation behind this research, illustrating cases where extensibility, or one of its necessary features, improve user work flow. Later chapters and sections will relate their subject matter back to these following subsections, showing how various methods and systems serve or fail to serve one or more of these use cases. This section concludes with other scenarios that extensibility may either improve or enable, but are not directly addressed in this work.

1.2.1 Language Evolution

Language evolution is the process by which language features are added, removed, or modified over time. Typically, both language users and language implementors participate in this process by proposing new language features to the language community, which is managed by a committee or language implementation lead. When a community member wants to argue for a new language feature, that member floats some hypothetical syntax, and then persists in argument about semantics until the proposal obtains a wide enough acceptance. Communities will then go back to arguing over the surface syntax, keeping the extension from being adopted for another year, or even longer.

Supporting evidence for the difficulty of this scenario (albeit circumstantial) is provided in one of Phil Wadler's blog entries [73]:

> My experience on the Haskell committee caused me to formulate 'Wadler's Law': time spent in debating a feature doubles as one moves down the following list:
>
> - Semantics
> - Syntax
> - Lexical syntax
> - Lexical syntax of comments

Some of the impediments to language evolution are social in nature, but others are technological. Technological obstacles to language evolution include the following: the language implementation is not modular, the modules of the language implementation are separated by functional boundaries instead of feature boundaries, an implementation of the language is not available in the language itself, and custom tools may have been added to the compiler-compiler tool-chain.

The CPython implementation of the Python language provides a good example of many of these obstacles. Python is modular, but the modules are separated across several functional boundaries: the parser, the bytecode compiler, the virtual machine implementation, and the standard library. A single language feature may require modification of one or more modules across these boundaries. A rudimentary extension system does allow the development of "extension modules", but these are C libraries that commonly have no affect on the parser or bytecode compiler, and little interaction with the virtual machine. The CPython source exists primarily as a set of C modules. These modules freeze the language's syntax and much of the language's semantics at compiler-compile time, per Figure 1.1. Finally, CPython uses its own parser generator, `pgen`, which has its own input syntax and idiosyncrasies. Attempts to popularize the CPython parser generator have not been well received by the Python community. Other implementations, such as Jython, have abandoned `pgen` in favor of more popular, `yacc`-like parser generators.

Extensibility improves this situation by allowing prototypes of language extensions to be rapidly developed and argued about from inside the language being extended. One may also argue that this is an anti-use case; a certain inertia in language design is good. Nobody wants to write code that will break by the next release of the language implementation. However, languages that support syntax macros, specifically Lisp and Scheme, show that modular language extensions are manageable [23]. Solutions to the language evolution scenario should provide the ability to extend or modify a language's surface syntax, intermediate representations, optimization pass(es), and possibly the language's code generator (or virtual machine).

1.2.2 Language Embedding

This scenario focuses on language embedding in the sense of formal languages. Formal languages are infinite sets of strings, where strings are ordered collections of characters or symbols in an alphabet. A formal language embedding consists of a host language and a embedded language. Assuming a common alphabet, a host language embeds another language when there are prefix and postfix strings that delimit all strings in the embedded language. The result of concatenating a prefix, a string in the embedded language, and a postfix is a string in the host language.

Formally, given an alphabet, Σ, and two languages that are subsets of the Kleene closure [1] of that alphabet, $L_h \subset \Sigma^*, L_e \subset \Sigma^*$, the host language L_h embeds L_e if the following holds:

$$\forall s_2 \in L_e . \exists s_1, s_3 \in \Sigma^* . s_1 \cdot s_2 \cdot s_3 \in L_h \tag{1.1}$$

Some of the following subsections use the terms formal language and concrete syntax interchangeably. In the terminology of language implementation, a formal language defines the concrete syntax of an implementation.

Motivations

Users have various motivations for formal language embedding. Implementors of an application programmer interface (API) may want to build a formal embedded language, adding domain-specific notation. In other cases, users may want to use code for an existing language from inside another language. Users can use an embedding of an existing (or variant) formal language for metaprogramming and/or generative programming.

Using a formal, domain-specific language typically lets experts express knowledge in a much more succinct form than general-purpose languages. A formal, domain-specific language has several advantages over the use of a library or API. These advantages include the introduction of notation, constructs and abstractions that are not present in the implementation language of the library [43] (Section 1.2.3 explores the additional advantage of domain-specific optimization, which considers not just embedded syntax, but semantics). Once the domain knowledge is encoded, it becomes useful to process and reason about the domain abstractions from within a general purpose language.

There are several examples of formal language embeddings. The JavaSwul example used in several of the MetaBorg papers [12, 11] associates a concrete syntax with the Java Swing API for graphical user interfaces. Other formal language embeddings add existing database query syntax to a general purpose language. The ESQL/C language [31] is an example of an embedding of the SQL database query language in the C general purpose language.

Embedded Strings

Programming languages often provide a prefix and postfix delimiter for encoding arbitrary strings in the alphabet. Using these delimiters, users can trivially encode strings in the embedded language as substrings in a host language. For example, the formal language of C uses the double quote character (`"`) as both the prefix and postfix delimiter for encoding arbitrary strings. Embedding another language via this method only requires the user to escape special characters in the alphabet, which

he or she does most often to avoid confusing embedded characters with either the trailing delimiter or other formatting. Given this limitation, using host language strings almost satisfies the formal embedding definition given in Equation 1.1.

What embedded strings actually satisfy is the following:

$$\forall s_5 \in \Sigma^*.\exists! s_5'.\exists s_4, s_6 \in \Sigma^*.\texttt{unescape}(s_5') = s_5 \wedge s_4 \cdot s_5' \cdot s_6 \in L_h \quad (1.2)$$

Equation 1.2 assumes the availability of a function, `unescape()`. The `unescape()` function maps from strings that contain escape sequences to arbitrary strings in the alphabet. Escape sequences reduce the readability of embedded code. Readability is especially difficult when the embedded code has a similar set of escape notation, requiring escape notation to be applied to itself.

Ignoring the readability and mathematical equality problems of escape sequences, Equation 1.2 satisfies Equation 1.1 only when the embedded string is in the embedded language, $s_5 \in L_e$. This condition is possible since $s_5 \in \Sigma^*$ and $L_e \subset \Sigma^*$. However, Equation 1.2 is too general, and accepts embedded strings not in L_e. Being overly general, the embedded string method implies that a recognizer or parser for the host language can not identify syntactic errors in the embedded code.

Language Assimilation

Assimilation is another approach to language embedding [12] that overcomes some of the problems of using embedded strings. The end product of the assimilation method are two things: a formal assimilation language that satisfies Equation 1.1, L_h, and a subprogram or program that maps from strings in L_h to strings in some target language, L_t. The formal assimilation language is commonly, but not always, a superset of the target language, $L_h \supset L_t$, with the target also being a general-purpose language. The corresponding subprogram or program is also known as a source-to-source translator.

The MetaBorg pattern is a specific instance of the assimilation method that uses the syntax definition formalism (SDF) [64] and Stratego [10] languages. The

MetaBorg method benefits from these tools because of their modularity and composition features. For example, SDF input consists of modular formal language definitions. An assimilation implementor can create an assimilation language once given two SDF modules that define a pair of formal languages, $L_1, L_2 \subset \Sigma^*$. When the formal languages are mutually exclusive, $L_1 \cap L_2 = \emptyset$, the assimilation implementor can define a new formal language that recognizes both L_1 and L_2 by creating a module that imports the given two modules. The assimilation implementor can then create an assimilation language, embedding L_2 in L_h and making L_h a superset of L_1, by adding some delimitation syntax and defining the start symbol to be in L_1.

Chapter 2 looks at the means and issues of assimilation in greater depth. Specifically, Chapter 2 shows how a language implementor can apply MetaBorg to its own tool set, SDF and Stratego. The result of self-assimilation begins to break down the distinction between compile-compile time and compile time as depicted in Figure 1.1.

1.2.3 Domain-specific Optimizations

A domain-specific optimization (DSO) is a high-level program transformation that is known to be true for some domain-specific language. For example, a graphics domain-specific language may define operations for both performing rotations on geometry and composing these rotations. Given some unary rotation operator, rot, a transformation composition operator, $\oplus$, and some construct that performs no transformation, id, it is possible to show:

$$\forall \alpha . rot(\alpha) \oplus rot(-\alpha) = id$$

This equality allows a user to reason that any object language term of the form $rot(\alpha) \oplus rot(-\alpha)$ is redundant, and can be eliminated. Eliminating the term from an object language program avoids a potentially costly composition operation when that program is run. It is not practical for a host language implementation to arrive at the same conclusion. Optimizers typically do not allow users to provide appropriate hints or facts about the object language semantics. This constrains optimizers to reasoning

about the object language in the semantics of the host language, or an intermediate language which is even less abstract.

When users implement domain-specific languages as host language interpreters, domain-specific optimizations may be added to the interpreter specification as fast-path term simplifications. This technique does not scale well for several reasons. First, the logic required to detect these opportunities slows down the overall performance of the interpreter. Second, the same DSO detection logic perturbs the interpreter's control flow, increasing complexity and decreasing the interpreter's readability and maintainability. Regardless of the sophistication of an interpreter's domain-specific optimizations, their inclusion will typically be unable to fully address the overhead of interpretation.

When users implement their object language by composing it with a host language, the language composer has several options for specifying domain-specific optimizations. If the language composition involves a translation from object language syntax to the host language's syntax, they can add logic to the translator that detects and applies optimizations at compile time. This approach limits the language composer to applying domain-specific optimizations without the benefit of interacting with the host language optimizer. If a language composer understands the host optimizer's specification, the composer has the option of extending that specification. To accomplish this, the composer can preserve some or all of the object language terms in the optimizer's language representation. Alternatively, the composer can manually translate object language rewrites into transformations of the optimizer's language.

Approaches to simplifying this scenario for DSO implementors involves allowing the same implementors to add high-level optimizations that rewrite object language terms. Unlike the kind of extensibility proposed in Section 1.2.1 and Section 1.2.2, supporting language user specified rewrites only requires extension of the host language optimizer. By accepting domain-specific optimizations, the host language's optimizer would be able to detect and apply these simplifications in object language programs at compile time. The Glasgow Haskell Compiler (GHC) provides a domain-specific optimizer that allows library developers to specify domain-specific optimiza-

tions as compiler pragmas [47]. Language implementors can make their optimizers more programmable and user-friendly by providing general semantic extensibility.

1.2.4 Language Tools

Language users sometimes want to do more with their programs than just compile and run them. Sometimes users want to transform their program specifications, perform style checks, or reason about the properties of their programs without running them. Automated tools can either enable or assist the user with performing these tasks. Example tasks and tools are listed below:

- Lexically or syntax directed editors, such as the editors in Visual Studio and Eclipse, or the language modes found in Emacs.
- Foreign function interface tools, such as SWIG [6] or `nlffigen` [8].
- Object modeling tools, such as Rational Rose [50].
- Model checkers, such as the BLAST checker for the C language [7].
- Coding style checkers, such as lint [32].
- Simple code transformation tools, including instrumenters and coverage analysis tools.

Each of these tools require access to some or all of an input language's specification, which is needed to understand a user's program specification. Specifically, the lexical and syntactic specification for a language is required to understand the form of program specifications input into the tool. In cases where a syntax specification is available, parser generation tools such as Bison encourage coupling of the source language grammar with actions. Coupling parser actions in a grammar specification can complicate or confound instrumentation that could be used to build models of an input program specification. For example, the developers of the Simplified Wrapper Interface Generator, SWIG, abandoned using an actual grammar for the C++

language. Instead, the tool writers developed a simplified syntax that was capable of handling a subset of the C++ language.

Language introspection is one feature that can assist tool implementors. Using introspection, a program is able to reason about parts of its structure at run time. Languages such as Java and C# allow some degree of introspection, but not necessarily enough for all kinds of analysis. In these cases, introspection provides tools the ability to determine the interface of a separate compilation units, but not necessarily the form of the original specification, nor details of the units that satisfy an interface.

One possible language implementation method that helps tool development involves exposing the language machinery used in compilation to the surface language itself. This exposed machinery could allow analysis of the lexical, syntactic and type properties of the language implementation as well as the source code it accepts. Previous subsections have already proposed exposure of compiler internals for the purpose of modifying or composing a host language. It follows that such extensible languages would already have or could easily add hooks for using the existing compiler to analyze source code. The "visible compiler" feature of the SMLNJ implementation is one example of this kind of reflection [4]. This feature enabled the development of a compilation management tool, CM, for SMLNJ [9].

1.2.5 Other Possible Scenarios

As described in Section 1.2.2, general-purpose languages are not always a perfect fit for capturing programmer intent. Besides simplifying specialized language embedding and development, language extensibility can help in scenarios where programmers want to annotate source code. Many of these scenarios are related to checks of program correctness. Such scenarios include clarifications of how a program interacts with subprograms written in other programming languages, supporting add-later types [56], and embedding machine-verifiable proofs of program correctness.

1.3 Language Extensibility is Program Extensibility

This section outlines the meaning of language extensibility and describes a method for making a language extensible. First, it defines reflection and shows how reflection implies extensibility. It then discusses glue languages and how they are applied to create extensible programs. This section concludes by describing application of the extension method to a language implementation. The result of applying extensibility methods to a language is an extensible language.

1.3.1 Reflection Implies Extensibility

Malenfant, Jacques, and Demers define behavioral reflection as "the integral ability for a program to observe or change its own code as well as all aspects of its programming language (syntax, semantics, or implementation), even at run-time"[42]. They state this definition is more a goal of reflection than an achievement. This work would call a language with behavioral reflection a fully extensible language. As argued in Section 1.4, and demonstrated in the following chapters, implementors make languages more extensible by applying reflection. However, both what is reflected and how it is reflected greatly affects the flexibility of a language implementation. With these goals in mind, the remainder of this section looks at the methods and terms used in reflection.

Brian Cantwell Smith's dissertation demonstrated a method for building a reflective tower of interpreters [57]. A T-notation version of Smith's method appears in the left-hand side of Figure 1.4. This method of stacking interpreters on top of one another does permit reflection of the form defined above. However, building successive interpreter implementations does not scale well. As a rule of thumb, each layer of interpretation generally slows a program down by an order of magnitude [33].

The middle diagram of Figure 1.4 shows the result of using bootstrapping, or embedded compilation, to improve the performance of a language tower. The horizontal layout in the diagram indicates that each new language level is an output of the previous level's implementation. In order to avoid successive layers of interpretation, bootstrapping requires that each implementation level include a compiler for itself:

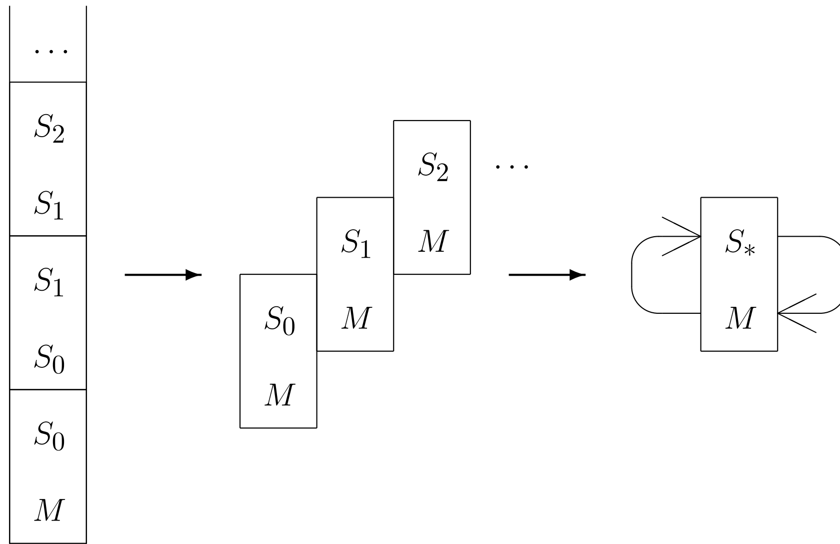
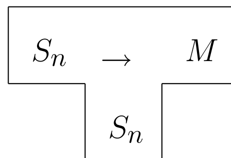


Figure 1.4: Toppling the reflective tower.



The language implementor uses this compiler to translate a S_{n+1} interpreter from S_n source to the native machine language, M.

The right-hand diagram in Figure 1.4 flattens the language tower further via reification. Reification is the process of exposing values in one language to another language. The implementor achieves reflection by reifying the implementation in the surface language. Figure 1.4 illustrates reification as an arrow leading from the M language implementation to the S_* language. The opposite arrow, leading from the S_* language to its underlying implementation, represents the ability for reified values to be modified, and possibly rebound. In the presence of an embedded compiler, a properly reified implementation achieves behavioral reflection without paying the cost of interpretation.

1.3.2 Glue Languages and Program Extensibility

A glue language is a programming language with the ability to reify machine language values at run time. Most glue languages implement this feature using parts of a system's dynamic linker. The glue language uses the dynamic linker to load machine language language values that are stored in a dynamic library or shared object file. As observed by John K. Ousterhout, a number of interpreted languages have the ability to reify machine values [46]. Such a language earns the name "glue language" by providing the ability to replace the high-level logic of an application that has been decomposed into a library.

The motivation behind moving parts of an application into a library is extensibility. Statically compiled applications have many program properties fixed at compile time. As application developers add more flexibility to their programs, they often create one or more configuration file formats. When adding further flexibility to these configuration files, the file formats begin to assume the properties of a small programming language. Embedding an interpreted language into an application takes the extensibility process even further, replacing the configuration file parsers with an interpreter. Reifying parts or all of the application in the interpreter transforms the application into an "application library" [43]. At the point that the interpreted language can replace the high-level logic of an application, it becomes a glue language, allowing both developers and users that ability to change application behavior without recompilation.

Figure 1.5 illustrates a scenario where a programmer uses a glue language to make a program extensible, allowing program users to modify or extend the program via modules. Using the resulting application library, the user can change a large degree of program behavior at run time. Based on how the developer reifies the various parts of the application and its data structures, the library can form an embedded domain-specific language, as described in Section 1.2.2.

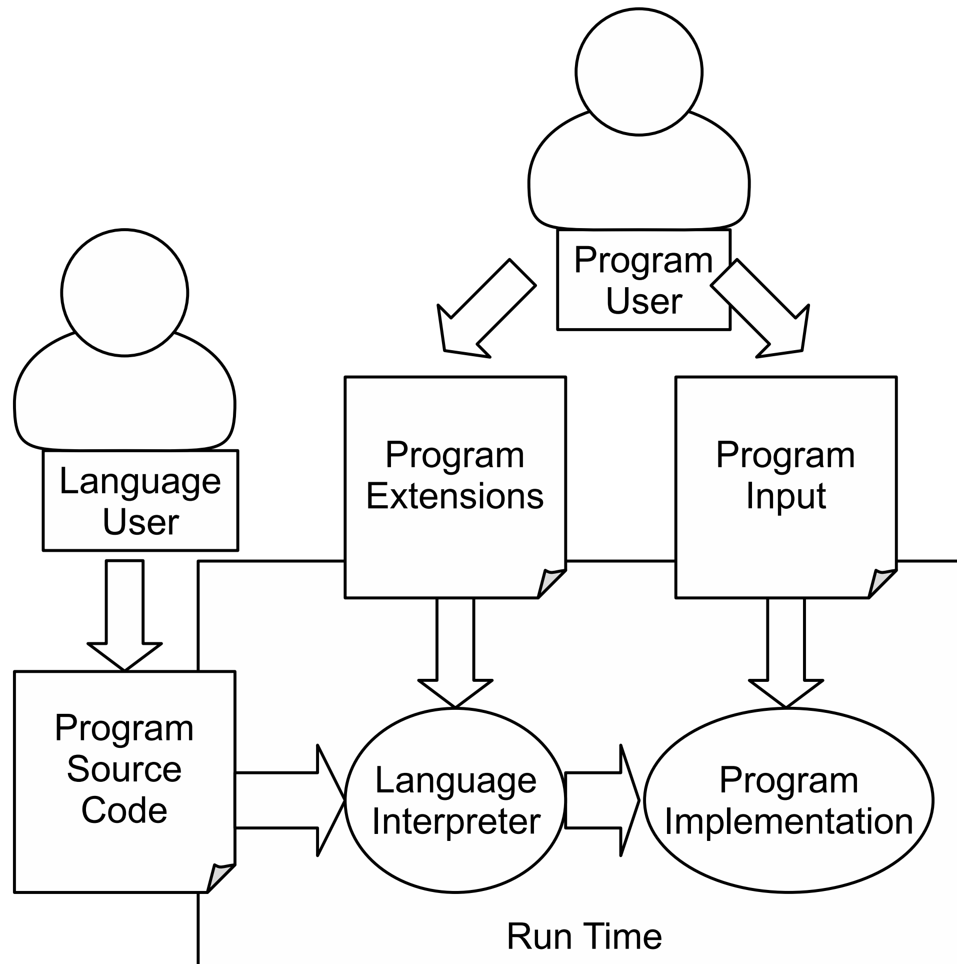


Figure 1.5: A traditional method of program extension.

1.3.3 Superglue Languages and Language Extensibility

A superglue language is a glue language that can both compose language implementations and recompose itself. Users can use the resulting extensibility to paste other language implementations into the superglue language. This kind of language embedding is similar to language assimilation, introduced in Section 1.2.2. Unlike language assimilation, a superglue language should incorporate more than just the properties of the embedded formal language. Ideally, the superglue language should be able to enforce and extend all static (compile-time) properties of the embedded language implementation. By honoring the static semantics of the embedded language, the superglue implementation overcomes many of the problems identified in Section 1.2.2.

A language implementor can create a superglue language from a glue language by first applying the extensibility method described in Section 1.3 to its own implementation. The result of this self-application is a visible or open compiler [4, 16]. The implementor then adds compile-time metaprogramming [61] to the language. Compile-time metaprogramming gives users the means to explicitly perform arbitrary computations during compilation (many interpreted language implementation use some form of a compiler, see Section 1.1). Users can use compile-time metaprogramming to embed or modify the current language implementation.

Both glue and superglue languages can embed the underlying tools used in language implementation. Through compile-time metaprogramming, superglue language users can avoid compiler-compile time by embedding the tool inputs (commonly a domain-specific language in its own right) in their libraries and programs. Superglue languages that incorporate these tools should allow users to create, embed, and experiment with languages much more easily than can be done with current tools. Chapter 2 specifically looks at tool embedding, and how users might control the resulting dynamism.

This reflective method of building superglue and extensible languages comes full circle, resulting in the kind of behavioral reflection described in Section 1.3.1. This method also has the effect of providing a spectrum of extensibility. The means of

reification are important in determining what behaviors a user can modify. For example, can users use arbitrary language tools or are they constrained to using the same parser generator as the original implementation? What about code generation? Is the underlying machine a virtual machine? Can the virtual machine itself also be extended? The following section, Section 1.4, identifies key parts of language implementation, and the affect that reifying those parts has on the extensibility of the language.

1.4 Increasing Extensibility in Languages

This section decomposes and defines the parts of common language implementations. Language implementors often decompose their languages into two halves: one half is responsible for handling the language syntax, while the second half is responsible for handling the language semantics as they relate to some means of evaluation. Later subsections look at these separate halves and common features of their specifications. The degree and kind of language extensibility available to language users is determined by how the language implementation is exposed back to itself.

1.4.1 Components in Language Implementations

This subsection decomposes common language implementations into two combinators. The language itself becomes a combinator from characters to code, and is composed as:

$$lang = back \circ front$$

Language implementors commonly refer to the first, syntactic combinator as the front-end. The front-end is responsible for translating input characters into data representative of the input program. The front-end's output is referred to as an intermediate representation. The second part is called the back-end. The back-end is responsible for translating the intermediate representation into code that is meaningful to a machine or evaluator. More sophisticated systems further decompose the

language implementation into more combinators with multiple intermediate representations. These can still be grouped roughly by function, and composed into a single syntactic and semantic combinator.

The front-end typically determines the syntactic properties of a language. How the syntactic combinator is reflected to the surface language determines the degree of syntactic extensibility of the language. The next subsection describes various forms of syntactic extensibility. The back-end defines the meaning of the language implemented by translating from the intermediate representation into code that can be evaluated by an actual machine. Reflecting the back-end combinator to the surface language allows users to not only change the language semantics, but also change how the language is translated. The concluding subsection looks at some common forms of semantic extensibility.

1.4.2 Syntactic Extensibility

Syntactic extensibility is the ability for a language user to change the syntax of the language. Language implementations support syntactic extensibility by reflecting some or all of the syntax specification to programmers. Language specifications often describe the syntax specification as a composition of a lexical specification, concrete grammar specification, and a mapping from concrete parses to abstract syntax. Language implementors may add various forms of syntactic extensibility by reflecting some or all of these syntactic specifications to the language user.

Lexical Extensibility

Languages support lexical extensibility by reflecting portions of their lexical specification to developers. User-defined type declarators, found in the C programming language, are one example of lexical extensibility [35]. The C programming language includes the `typedef` statement, which allows users to extend the set of strings that can declare a type. Following a `typedef` statement, a string that would have originally been lexically classified as an identifier is classified as a type name instead. This form of lexical extensibility is less of a convenience to the language user, and more

a means of preventing ambiguity in the grammar specification. Other imperative languages, such as variants of BASIC, permit users to extend the class of keywords that begin statements. In these constructs, the user extends the keyword lexical class and associates the keyword with a subroutine. This allows the language user to do a very simple form of abstraction and lexical binding. Standard ML provides another example of lexical extensibility, allowing users to define new operators at the lexical and syntactic level of the language [44].

Syntactic Sugar and Macros

Syntactic sugar is a programming language feature that exposes shortcut syntax for performing a computation that could still be expressed in the programming language without the "sugar" syntax. Syntactic or syntax macros allow users to create these shortcuts from inside the programming language. Support for syntax macros can be found in languages such as Lisp and Scheme. Syntax macros differ from simple or parameterized string-replacement macros. String replacement macros use string substitution, matching a user-specified string and replacing it with another string before the language performs lexical analysis. Syntax macros modify the syntactic specification of a language. When the language's parser recognizes a syntax macro, the parser expands the macro expression into syntactic structures defined in the original programming language. Syntax macro implementations take special care to avoid binding and name capture issues during macro expansion. Avoid these issues make syntax macros "hygienic" as opposed to string replacement macro systems [17, 21].

Source-to-source Translation and Extensible Parsers

Source-to-source translation is the translation from the concrete syntax of an object language into the syntax of a host language. Like syntax macros, a source-to-source translator outputs the abstract syntax of the host language. Unlike syntax macros, a source-to-source translator will often not extend nor use the concrete syntax of the host language. Source-to-source translation requires replacement of both lexical and concrete syntax, while syntax macros tend to fix the lexical conventions, and only

extend the underlying grammar. Concrete syntax macros are syntax macros that allow lexical extension and modification, and therefore source-to-source translation.

An extensible parser is a parser that allows users to modify or replace the concrete syntax it parses. A host language that reflects an extensible parser allows users to embed object languages (per the scenario described in Section 1.2.2). A user embeds an object language by first specifying a source-to-source translator as an extension to the parser. Following this step, the extended parser will recognize object language syntax and automatically translate it to abstract syntax in the host language.

Often the concrete syntax of the object and host language are not mutually exclusive. The result of this is ambiguity. For example, many languages support an infix addition operator using the plus symbol, "+". When one such language is embedded in another, the extended parser is unable to handle expressions such as "a + b" since it can't determine if it is an object language or a host language expression. One way the user can avoid this ambiguity is to use a delimiter pair to signal transition into and out of the object language. A delimiter pair consists of two lexical classes, such as the open and close parenthesis. Using a delimiter pair in a language cues the parser that a change in syntax occurs upon entry of the first delimiter and reverts upon encountering the second delimiter.

1.4.3 Semantic Extensibility

Semantic extensibility is a language feature that allows a user to change a program's meaning, its target encoding, or both. When the user can alter a program or subprogram's meaning, the language can support code reuse in ways that differ from functional abstraction. Very often there are multiple ways to map a high-level program to a low-level representation that a machine can evaluate. By making the machine code translator programmable, a language implementation allows its users to optimize their program to a particular operating environment. Examples of semantic extensibility include dynamic binding, staging, and custom optimizers.

Polymorphism and User-Programmable Types

Polymorphism is one form of code reuse. Polymorphism differs from subprogram abstraction in that not only can the value of inputs change, but the side-effects of a subprogram, as well as the types of the inputs and outputs can change. In statically typed languages, this involves type abstraction and specialization. Type abstraction is similar to function abstraction, but instead of abstracting over an open expression, a user either implicitly or explicitly abstracts a type expression over an unbound type. When an abstract, or polymorphic, type is bound to a concrete type, the language specializes the resulting code. Code reuse occurs when polymorphic code can be specialized multiple times, such as the C++ template system.

Taking the idea of type abstraction further leads to programmable type systems. In a system with programmable types, the user can perform computation at compile time. Type abstraction and specialization correspond to abstraction and application in the simply-typed λ-calculus. The types of types in this correspondence are called kinds. One approach to increasing type system programmability involves allowing user defined kinds. User defined kinds in the presence of algebraic data types increases the type safety of embedded domain-specific languages (EDSL's, Section 1.2.2). This increased type safety follows because the language's type checker will only admit programs that construct provably correct terms in the EDSL [15, 55].

Late Binding

Language implementations sometimes use late binding as another means of implementing polymorphism. Late binding, or dynamic binding, is the postponement of the determination of which value is bound to a piece of syntax (most often a global variable) until either code-generation time, link-time, load-time, or run-time. When binding is done later than code-generation time, a program's meaning or behavior can be modified by changing the link-time or run-time environment of a program. Dynamically typed languages with late binding achieve polymorphism via dispatch instead of specialization. These languages are commonly characterized as having a single static type. Their run-time implementations associate a type tag with all

language values. When the implementation performs an operation or references an attribute of a value, the runtime looks at the value's type tag and dispatches the value to a handler function.

Staging and Quotation

Staging permits code reuse by reflecting an intermediate representation of a program or subprogram as a value. One method of implementing staging involves a quotation construct and a reflected interpreter. The quotation construct causes any contained syntax to evaluate to its intermediate representation. At run time, the user's program can evaluate the intermediate representation value using the reflected interpreter. Staging not only supports code reuse, but offers a roundabout means of semantic extensibility. When the reflected intermediate representation can be decomposed into language primitives, users can change the meaning of quoted code by implementing their own interpreter for the intermediate representation. Both late binding and staging are temporal forms of semantic extensibility, since they can affect not just how code behaves, but when that behavior occurs. Later chapters use staging to contain extensibility to an explicit evaluation time.

Optimization and Partial Evaluation

The most basic form of semantic extensibility is found in the form of common compiler flags and pragmas. These are often not a part of the formal language specification and either appear as command line options to a language's implementation or a special form of inline comment. These flags and inline comments still control important semantic features of the output encoding. For example, the GNU compiler collection (GCC) provides compiler flags that can change the optimizer's behavior, allowing programmers to bias the optimizer for space or run-time efficiency, among other options. GCC also supports inline pragmas that change how data structures are represented in memory. These pragmas can modify the address alignment of both data structures, and the fields contained by the data structures. Compiler flags are often used to change the semantic combinator's output type, which is desirable when

the language user wants to create code for a different machine architecture than the compiler architecture (compiling to a different architecture is also called cross compilation).

Instead of deferring computation and specialization until run-time, optimizers seek to specialize and evaluate programs at compile-time. Lacking the inputs of a program user, the optimizer uses various methods of program transformation to simplify known values and avoid redundant computation. One approach to optimization follows a two step method. In this approach, the optimizer first inlines functions. Inlining involves taking known abstractions and substituting calls with the actual abstraction expression, specialized for the inputs provided by the caller. Then, the optimizer simplifies expressions. Constant folding is one means of expression simplification. An expression that passes a constant to a known primitive operator can be evaluated by the compiler itself. The result is turned back into a value in the intermediate representation, and substituted for the original. This method of partial evaluation is repeated until the optimizer cannot find further opportunities for inlining and simplification, or it reaches some user-specified bound.

1.5 Overview of the Dissertation

This dissertation argues that the process of reflecting a language's implementation into the language itself either enables or improves on the development scenarios outlined in Section 1.2. The following chapters first build a set of methods, then apply those methods to build systems, and finally demonstrate those systems as a means of validating this hypothesis.

Chapter 2 introduces the idea of staged multilanguage programming. It accomplishes this by looking at previous work done in language embedding. It then uses the reflective method of self-application. This creates the kind of syntactic and semantic extensibility that both language developers and users can apply to solve problems in language evolution, extension, optimization and analysis.

Chapter 3 serves to provide a concrete example of a staged multilanguage programming language. Building upon the ideas introduced in Chapter 2, the chapter

describes the Mython language. It illustrates both how Mython reflects its implementation to the surface language, and how users can apply these features.

Chapter 4 goes into further depth on techniques for implementing extensible optimizers. Building on work done for the Glasgow Haskell Compiler, Chapter 4 frames optimization as a term rewriting problem. It then looks at a variety of methods for term rewriting. The chapter concludes by comparing several implementation of these methods, and looks at systems that either currently or could easily apply rewrites as means of domain-specific optimization.

Chapter 5 provides a case study in the application of Mython. The chapter develops MyFEM, a domain-specific language for specifying partial-differential equations that scientists and engineers use in the finite element method (FEM). It illustrates how MyFEM applies Mython and its techniques to arrive at a system that is both more flexible and safer than other approaches.

Chapter 6 looks at the theory, methods and systems that are related to this work.

This work concludes with Chapter 7. The conclusion first reviews the methods and contributions of this dissertation. It then looks to the future of this work, identifying both incremental improvements to the Mython and MyFEM languages, as well as new directions and methods for these systems to take and apply.

CHAPTER 2
THE SELF-ASSIMILATION OF STRATEGO/XT

This chapter begins by describing the SDF and Stratego languages and how Stratego enables assimilation, as defined in Section 1.2.2. It then looks at assimilating SDF and Stratego with another language, and describes applications of the result in the presence of reflection. It concludes by looking at preliminary work.

2.1 Stratego and MetaBorg

Figure 2.1 shows a modified version of the story in Figure 1.1. This illustration adds modularity to the compiler-compiler tool-chain. Given the source code for a modular language implementation, users of that implementation can extend or embed that language by defining or composing new modules at compiler-compile time.

Both the syntax definition formalism (SDF) and Stratego are modular languages with applications towards language implementation. When language or tool implementors provide partial or full language implementations in SDF and/or Stratego, they allow users to extend and embed those parts using additional SDF or Stratego modules. MetaBorg is a method of using SDF and Stratego modules to both embed a formal language, and transform any embedded source into host source. The following subsections describe the SDF language, the Stratego language, and the MetaBorg pattern in more depth.

2.1.1 Composable Syntax Definition in SDF

The syntax definition formalism (SDF) is a modular language for describing context-free grammars [64]. The SDF implementation accepts a syntax definition module and outputs a parse table for a scannerless, generalized-left-right parser (SGLR parser) [69]. Users employ the parse table using a generalized parsing program. The parser

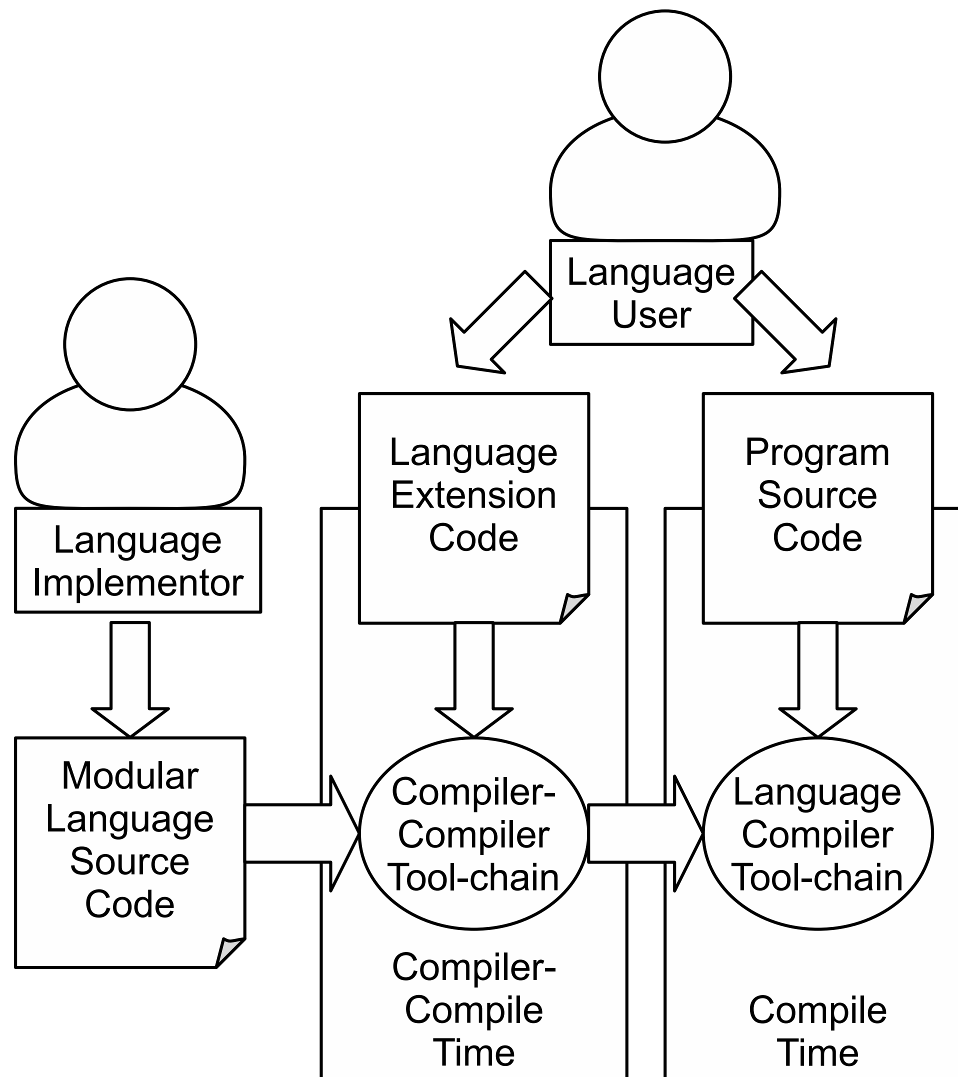
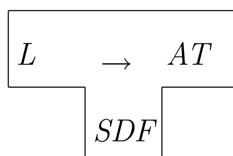


Figure 2.1: The Stratego approach to language extension.

accepts a file or string input and a parse table, and either fails to parse, or outputs a syntax tree. The syntax tree is output as an ATerm [63] data structure, which can be expressed as either a string or encoded in a binary format.

In T-notation, the SDF source is a transformer from the strings in the defined formal language, L, to ATerms, AT:



The SDF parser can either output concrete syntax or abstract syntax trees, based on flags passed to the parser. The SDF parser can also handle ambiguous grammars, output a special ATerm constructor, `amb`, as a container for multiple parses. The abstract syntax for a language definition is initially based on the nonterminal symbols of the grammar, called sorts. Language implementors can associate specific constructors with grammar productions. SDF includes tools for building regular-tree grammars, which are useful for describing and enforcing abstract syntax compliance in ATerms.

SDF users can compose formal languages by first using module composition, and then either adding or overriding syntax to form a hybrid language. Module composition simply requires the SDF user to define a hybrid module, and import the given language modules. The implementor then provides additional syntax and rules to mix, reuse, or embed the languages, and possibly resolve ambiguities.

Figure 2.2 provides an example of a trivial composition of two languages. Readers familiar with Backus-Naur form (BNF) grammars will find that productions are reversed in SDF, with the nonterminal result of a reduction appearing on the right-hand side. This example is trivial because two new tokens, "`%(`" and "`)%`", are used to delimit the embedded language. These new lexical tokens prevent ambiguities between the host and embedded languages, because substrings in the embedded language can only occur between these delimiters.

```
module HostLanguage
exports
  sorts HostExpr
  context-free syntax
    HostExpr "+" HostExpr -> HostExpr
    ...
```

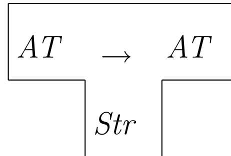
```
module AssimilatedLanguage
exports
  sorts AssimilatedExpr
  context-free syntax
    AssimilatedExpr "*" AssimilatedExpr -> AssimilatedExpr
    ...
```

```
module HybridLanguage
imports HostLanguage
        AssimilatedLanguage
exports
  context-free syntax
    "%(" AssimilatedExpr ")%" -> HostExpr
    ...
```

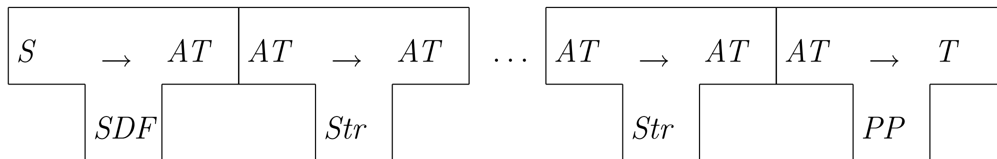
Figure 2.2: A host grammar, an extension grammar, and a composition grammar.

2.1.2 Program Transformation in Stratego

Stratego is a language for defining term traversal and transformation strategies [70, 10]. A strategy is essentially a function that accepts a term and an environment, and either fails, or returns a term and an environment. The Stratego compiler outputs a standalone program that accepts ATerm input, and either fails or outputs another ATerm instance. In T-notation, a Stratego transformer appears as the following:



Stratego users create a program transformer by composing two or more programs. The first program is the SDF parser, and is responsible for translating from the source language, S, to an ATerm representation of the program, AT. Once a source program is in the ATerm format, zero or more Stratego passes transform the source term. The implementor sequentially composes these using pipes, with the output of one pass being fed to the input of the next pass. The final transformer is a pretty-printer program[1], PP, that translates from ATerms, AT, to the target language, T.



Using this framework, developers can create a large variety of tools, including full language implementations. For example, given a set of optimization transformations, a user could create an application-specific or environment-specific optimizer. By experimenting with different optimization passes and optimizer orderings, the user could ensure the transformed program is optimal for their specific application or environment.

1. Stratego provides a pretty-printer language, as well as tools for generating pretty-printer boilerplate from a syntax specification.

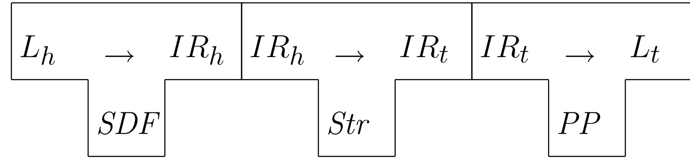
2.1.3 MetaBorg and Language Assimilation

MetaBorg is a usage pattern of the Stratego/XT tool set [12, 11]. A user follows the MetaBorg pattern by providing a grammar that composes two formal languages, and a transformation strategy for translating from composite abstract syntax trees to a target abstract syntax trees. The user applies the SDF and Stratego tool set to the grammar and strategy, and composes the resulting parser and transformer. The result is a source-to-source translator that specifically addresses the language embedding scenario presented in Section 1.2.2. As with older language implementation tools, the MetaBorg pattern is applied at compiler-compile time, generating a translator from a hybrid language to a target language (the output of which would still have to be fed to another language implementation). This precludes a certain degree of dynamism at compile time and run time, including introspection and the ability to leverage the compiler-compiler tools for run time code generation or run time assimilation of a new language.

This chapter uses some non-standard terminology. Assimilation is the process of applying the MetaBorg pattern, or a similar process, for the purpose of language extension or embedding. The developer that applies the assimilation method is the assimilator. The result of the assimilation is a translator. The translator translates from a hybrid language to a host language. Developers that use the hybrid language are users. As already demonstrated, the "$\oplus$" symbol represents an assimilator given composition of syntax or Stratego code, and is not a primitive operator.

Given these conventions, an assimilator follows the MetaBorg pattern by specializing the program transformation framework in Section 2.1.2. The assimilator provides two pieces of source code. The first piece of code is a SDF grammar for the hybrid language, $L_h = L_t \oplus L_e$. The SDF grammar generates a front-end for the translator, outputting a composition of the two abstract syntaxes, $IR_h = IR_t \oplus IR_e, IR_t \subset IR_h \subset AT$. The second piece of code is a strategy that traverses the hybrid terms, transforming terms from the embedded abstract syntax, IR_e, into terms in the target abstract syntax, IR_t. This example assumes there is already a pretty printer for the target language. In T-notation, this example results in the following program

transformer:



Assuming that assimilation is not something one would want to do at run or compile time, MetaBorg has already demonstrated itself as being useful for language embedding. Language evolution is served by the same process, where a language is simply extended by creation of a modular definition that is integrated with the target language syntax. Since Stratego/XT decouples syntax and semantics, tool development for static analysis can be done by writing a simple set of rewrites against an existing syntax. By virtue of exposing a comprehensive rewriting framework, Stratego/XT provides a good mechanism for permitting domain-specific optimizations. Actually leveraging the rewrite framework is difficult unless the compiler's implementation language is C. It is conceivable that domain-specific optimizations could be provided as patterns in the assimilation code, but this does not allow interaction with the native language optimizer.

2.2 Embedding Language Embedding Tools (or, is resistance really futile?)

This section begins with a discussion of the self-application of the MetaBorg pattern. It then provides a description of how to use the result to apply the MetaBorg pattern from within a single hybrid language. It then gives a generalization of necessary features of the hybrid language, and reapplies these features in order to enable domain-specific optimizations. Finally, it examines how techniques used in staged programming can assist developers in managing the resulting complexity.

2.2.1 Assimilating MetaBorg in Another Language

The assimilation of MetaBorg into a host language involves self-application of the MetaBorg pattern to the Stratego/XT tool set. First, the assimilator must compose

the host language syntax with the Syntax Definition Framework (SDF) syntax and the Stratego syntax. Second, the assimilator must provide transformation strategies for translating from SDF and Stratego syntax into host language constructs. This process yields a front end that allows users to embed syntax and rewrite strategies in the host language.

As first described in Section 2.1.1, the complexity of composing two grammars is managed in Stratego/XT by encapsulating them into separate modules. The assimilator proceeds by taking modular definitions of the extension syntax and the host language and importing them into a third module. This third module then adds context free reduction rules that reduce (possibly escaped) extension syntax into nonterminals found in the host language. Recursive embedding is accomplished by providing reductions from host language syntax into nonterminals of the embedded language (allowing something similar to anti-quote). The compositional syntax module may also provide additional syntax, disambiguation rules and lexical conventions that modify and refine the resulting hybrid language.

The Stratego/XT and SDF distributions provide SDF syntax specifications for their front ends. Both the Stratego and SDF grammars use modules as a top level unit of rewrite and syntax encapsulation, respectively. Before making a commitment to the details of host language representation, it is plausible that an assimilator would represent these modules as some kind of first class object. Therefore, a good hook for the SDF and Stratego grammars is a reduction that makes an SDF or Stratego module into a host language value, hooking into the host syntax at either an atom nonterminal, or the more generic expression nonterminal. Modulo some renaming and additional disambiguation, the composition syntax definition could be as simple as the following:

```
module MetaBorgAssimilation
imports Sdf2Renamed StrategoRenamed HostLanguage
exports
  context-free syntax
    "sdf" Sdf2ModuleName Sdf2ImpSection* Sdf2Sections -> HostAtom
```

```
    "stratego" StrategoModName StrategoDecl* -> HostAtom
...
```

Once the details of the composition syntax are resolved, the assimilator only needs to provide a host syntax and a module that renames key host language constructs into a "HostLanguage" module. This example is not without challenges. The SDF and Stratego module names are left in the example grammar since these names are used by the import declaration syntax intrinsic to both module grammars. Without allowing the host language namespace to extend into the tool set module namespace, these values will most likely have two names: a host language variable name, and an internal SDF/Stratego module name. This limitation could be sidestepped by making the module grammars hook at the declaration or statement level, but fewer potential host languages have declaration or statement syntax. More sophisticated couplings that would permit embedding host expressions in the SDF or Stratego code would certainly require more extensive composition rules.

The assimilator completes an assimilation by defining rewriting strategies from the hybrid syntax to the host syntax. In the specific case of the self-application of MetaBorg, there are a large number of possible terms (one for each symbol in the SDF and Stratego grammars) that need an explicit rewrite from their ATerm representation to their corresponding host language construct. In the case where the host language supports the ATerm library, self-assimilation of Stratego and SDF could use an exhaustive traversal strategy that translates the in-memory ATerm forms into host language calls to the proper ATerm constructors. Creation of this strategy could be automated by writing another traversal strategy on SDF grammars.

An assimilator might also consider two other alternatives to ATerm constructors, including using a container library or other first class objects. Native containers would provide users with a familiar means of inspecting and using embedded syntax. One of the problems with this approach is the initial abandonment of the Stratego/XT framework. The assimilator using native containers could write relatively straightforward container traversal functions that translate from the host data structures to ATerm forms. Another approach would involve the assimilator embedding the Strat-

ego/XT framework or something similar into the host language. In this approach, the assimilator could make syntax and rewrites first class objects in the host language. For example, a parse tree object could explicitly link the syntax that was used for parsing. Rewrite strategies could also statically report the set of terms they operate on, allowing some strategy failures to be determined at compile time.

2.2.2 Making Assimilation a Language Feature

Once syntax and semantics are integrated into a host language, assimilating other languages into the host language requires two additional steps. First, the host language must embed and expose the underlying machinery of assimilation. This machinery consists of a parser generator and a rewrite engine. Second, the host language must provide users with the ability to modify the parser being used by the host-language implementation. Once these steps are taken, users may become assimilators, and the original assimilator could possibly be considered a meta-assimilator.

The first piece of assimilation machinery needed by the host language is a parser generator. For the sake of simplicity, one should assume the parser generator is available as a function, *pgen*, with the following type:

$$pgen : Syntax \rightarrow String \rightarrow Tree_{opt}$$

This definition relies on the following assumptions: The *Syntax* type contains all the lexical and syntactic conventions necessary to parse an input string. Second, the parser outputs the parse tree as a generic tree with tagged nodes. Finally, if there is a syntax error, no tree is output. All of these assumptions are satisfied by a simple wrapping of the Syntax Definition Framework's SGLR parser generator.

The second assimilation mechanism required in the host language is a rewriting engine. Here, one should assume the engine is exposed as a transformation function, *transformer*, with the following type:

$$transformer : Strategy \rightarrow Tree_{opt} \rightarrow Tree_{opt}$$

The *transformer* function curries a general rewrite strategy with a wrapper function that passes `NONE` values through, and translates from the `fail` strategy result (which indicates failure of the overall strategy) to the `NONE` value. The result is a function that accepts trees or a failed parse and returns either a rewritten tree or a failure value. This typing is not ideal since it fails to capture the constraint that nodes in the input tree may be symbols in the hybrid language, while any output tree should only contain symbols in the host language.

A final piece needed for assimilation is the host language syntax.

$$syntax_{host} : Syntax$$

The host language provides an assimilator with the *pgen* and *transformer* functions, and the base syntax, $syntax_{host}$. To complete an assimilation, assimilators must also provide extension syntax, $syntax_{ext}$, and a rewriting strategy from the hybrid syntax to the host syntax, *ext2host*. Composing these pieces allows the development of an assimilation as follows:

$$syntax' = syntax_{host} \oplus syntax_{ext}$$

$$assimilation = (transformer\ \ ext2host) \circ (pgen\ \ syntax')$$

Hence the *assimilation* combinator defined here is a function from string data to a tree, or `NONE` when an error occurs.

$$assimilation : String \rightarrow Tree_{opt}$$

In the original MetaBorg pattern (Section 2.1.3), the assimilation is composed with a pretty printer that outputs resulting host syntax tree as code. The host language code is then be re-parsed and compiled by the original program. An alternate option is for the host language implementor to extend the host language with a construct for changing the language's parser from inside the language. The resulting construct might look something like this:

$\texttt{use}(assimilation);$
$stmt_{syntax'};$
$\ldots$

In this example, the `use` statement accepts functions of type $String \rightarrow Tree_{opt}$. Immediately following this statement, the parsing function passed to `use` replaces the previous parser. Therefore, any input following the `use` statement would be in the hybrid language. Unless the `use` statement is explicitly removed or overridden from the hybrid language, users may switch back to the initial host language by inputting a `use` statement with the initial parsing object as an argument ($pgen\ syntax_{host}$). Other variants of this form might constrain the hybrid syntax to a block statement.

2.2.3 Generalizing Implementation Hooks for Fun and DSOs

The rewrite capabilities of Stratego/XT fail to allow interaction between a host language's optimizer and the domain-specific optimization capabilities provided by rewrite strategies. The language extensions described in the previous subsection (Section 2.2.2) bring the rewrite engine into the language at compile time, but still only allow transformation in the implementation's front-end. One could apply a tactic similar to the previous section, but with each of the stages of a language implementation exposed to the language and fully replaceable.

For example, assume one wants to extend an implementation L, which is the composition of the following combinators:

$$L = codegen \circ optimizer \circ typech \circ parser$$

$$L : String \rightarrow Code_{opt}$$

The *parser* stage is similar to the one seen in Section 2.2.2, and maps from a string to an optional tree. In this example, the parser is joined by a type checking stage, $typech : Tree_{opt} \rightarrow Tree_{opt}$, where the output tree may contain additional type

annotations. The type annotated abstract syntax tree is then sent to an optimization phase, $optimizer : Tree_{opt} \rightarrow Tree_{opt}$. The optimizer might include constant folding, inlining, and strength reduction transformations[1]. The final stage is a code generator, *codegen*, that returns code for some machine.

An assimilator of a domain-specific language would employ a slightly modified `use` statement that would allow L to be entirely replaced. The assimilator would first provide a new parser, $parser'$, as in Section 2.2.2. The assimilator would then use a new optimizer, $optimizer'$, that composes the domain specific optimizations with the existing optimizer, *optimizer*. The result is a new language compiler, L':

$$L' = codegen \circ optimizer' \circ typech \circ parser'$$

Once defined, L' can be an argument to the new `use` statement:

$$\begin{array}{l} \texttt{use}(L'); \\ stmt_{L'}; \\ \ldots \\ \texttt{use}(L); \end{array}$$

An assimilator composes the domain-specific grammar with the host syntax (in the same manner as shown in the previous subsection).

$$parser' = pgen(syntax_{host} \oplus syntax_{ext})$$

To build an optimization phase that allows interaction with the native optimizer, a wrapper strategy must be introduced, *optwrap*. The *optwrap* combinator simply adapts the tree rewriting optimization function to the rewrite engine.

$$optwrap : (Tree_{opt} \rightarrow Tree_{opt}) \rightarrow Strategy$$

Using the wrapping strategy, the assimilator explicitly composes the native optimizer with domain-specific optimization strategies, *dsopts*. The assimilator may then

```
>>> mlsyn = <|syntax| ... |>
>>> mlrw = <|rewrites : mlsyn -> pysyn| ... |>
>>> mlfact = <|mlsyn| fun fact 0 = 1
                        | fact n = n * fact(n-1)
              |>
>>> mlfact_cobj = compile(transform(mlrw, mlfact))
>>> exec mlfact_cobj
>>> fact(4)
24
```

Figure 2.3: An example of multilanguage staging.

create a new optimizer, *optimizer′*, by applying the *transformer* combinator to the new strategy composition:

$$optimizer' = transformer(dsopts \oplus (optwrap\ \ optimizer))$$

The host language could also expose specific optimization phases as transformation strategies that can be composed with the user defined optimization passes in an explicit fashion. Other variants that do not specifically rely upon Stratego/XT might expose the language lexer and/or use different parser generators and parsing functions.

2.2.4 Staged, Multilanguage Programming

Staged programming uses a mechanism that embeds abstract (or sometimes concrete) syntax trees into a language [59]. A straightforward extension of the staging mechanism could make arbitrary syntax trees first class objects in the host language. Specifically, one could add a parameter to the quotation syntax, where users could specify the grammar used to parse the quoted material, or possibly a grammar and start symbol pair. An example of a parameterized quote construct is given in Figure 2.3.

In the example code, the parser generator input grammar is bound to the `syntax` name, and used as an argument to the (elided) quotation on the first line. The result is a parse tree of the syntax description appearing between the “`|`” and “`|>`” tokens.

The rewrite engine input grammar is bound to the `rewrites` name, and is used in the second line of the example to express a rewrite from the new syntax, `mlsyn`, to the host syntax, `pysyn`. The third line illustrates the application of the new syntax. At this line the interpreter would first ensure the argument is a syntax tree that conforms to the parser generator input grammar. The interpreter would then use the embedded parser generator to create a parser, and parse the quoted code. The expression would return a syntax object that is bound to `mlfact` in the assignment statement. The next statement shows application of the rewrites to the syntax object and the compilation of the resulting tree. The remaining statements execute the code object returned by the compiler, and apply the defined ML function using a host language call.

The example in Figure 2.3 elides a large amount of the complexity involved in defining a grammar and more importantly the rewrite semantics that map from an embedded language into the host language syntax. On the other hand, the constructs introduced in this example meet challenges not addressed in previous subsections. Previous subsections focus on how one can self-apply the MetaBorg pattern, but do not provide guidelines for managing the resulting complexity. For example, composition of an EBNF syntax with a C-like host language would present a heavy overloading of the "|", and "*" tokens. Coupling the right hand side of a production to expressions in the host language would result in heavy syntactic ambiguity that the assimilator would have to explicitly resolve. Parsing issues aside, the resulting grammar would most likely confuse most human readers as well. The staging constructs of quote and anti-quote are already familiar to some users, are useful for explicitly delimiting language transitions, and avoid ambiguity.

2.3 Preliminary Experiences

Section 2.2 outlines self-application of the MetaBorg pattern, but remains neutral with respect the host language. The outline could be applied equally to either a compiled or interpreted language. Self-application with a compiled host language is in conflict with the initial intent of this research, which was to allow incremental and

interactive assimilation. This intention constrained the choice of host languages to languages that provided a read-eval-print loop (REPL). The remainder of Chapter 2 considers the use of Python as a host language. Python is a good choice because of its interactivity, clean syntax, and easy interoperability with C code at the application binary interface (ABI) level. This choice was not without complications.

The first complication involves translation of the Python syntax into an SDF module. Python uses whitespace sensitive block delimitation that is detected by the lexer. The Python lexer detects changes in indentation, and at these transitions passes either an `INDENT` or `DEDENT` token to the parser. SDF allows lexical conventions to be specified in grammar definitions, but it is not clear how to specify this whitespace convention so that tokens are matched only at indentation transition points. This limitation might necessitate the development of a preprocessor that inserts explicit indentation tokens that would be matched to a non-invasive keyword pair. It is also possible that a better understanding of SDF could result in a grammar specification that avoids this issue.

Another complication involves interoperability with the ATerm library. Python's C implementation allows relatively straightforward interoperability with other libraries that support C calling conventions. Furthermore, shared object libraries may be accessed and called directly at runtime using an embedding of the C dynamic linker library. ATerm (and with trivial modifications, the SGLR library) is available as a shared object. Using the Python `ctypes` extension module, the interpreter can call into the ATerm library without development and compilation of a wrapper. The problem with this approach is that much of the ATerm typing and traversal code is implemented as C pre-processor macros, and are not exported as symbols in the shared object. These macros, such as `ATgetType()`, must be wrapped by C functions for them to be visible from the Python interpreter.

While not a technical issue, the lack of existing tool support does underscore the need for grammar engineering tools, or what Klint et al call Grammarware tools [39]. Specifically, a grammar specification adapter would be very useful, saving the time needed to translate the Python grammar specification (written for a custom parser generator) to a SDF module. Additional automation could assist with grammar

composition, identifying name conflicts, and possible ambiguities on a sort by sort basis.

2.4 Conclusions

Assimilation of the MetaBorg-tool set into an interactive host language with concrete syntax should serve language embedding, language extension, and the development of other tools better than existing approaches. The embedding, extension and syntactic analysis tools of the resulting system may be applied across multiple stages of development, including compiler-compile time, compile time, run time, and even later stages (in the case of staged programming). Self-assimilation with an extensible language further enables the development of domain-specific optimizations that can interact with one or more of the native optimization passes used in a given implementation. This chapter has identified how self-assimilation can be performed, and presented how current systems do not fully address a wide set of language development problems. While self-assimilation is still only theory at this time, the details presented in Section 2.2 should be specific enough for language implementors to embed the Stratego/XT framework, and add language extensibility features to their languages. The resulting feedback loop could result in white noise, but it could also provide language users with much more flexibility and autonomy while accelerating language development.

CHAPTER 3
THE MYTHON LANGUAGE

This chapter describes the Mython language. Chapter 2 introduced a methodology for creating composable language implementations. Mython refines the ideas presented in Chapter 2 by introducing compile-time metaprogramming [61], first mentioned in Section 1.3.3. Compile-time metaprogramming in Mython involves adding syntax that tells the implementation to interpret parts of the source at compile time. This approach provides a more uniform means for run-time language extension than the mechanisms in Section 2.2. Figure 3.1 illustrates how Mython changes the story first shown in the introduction.

Mython is an extensible variant of the Python programming language. Mython achieves extensibility by adding a quotation mechanism. The quotation form in Mython includes both quoted source and an additional parameter. The Mython compiler evaluates the additional quotation parameter at compile-time. The compiler expects the result of the compile-time parameter to be a function. Mython uses this function to both parse the quoted source code, and extend or modify the top-level bindings used for compile-time interpretation.

These identifier bindings form a dictionary or name table that this chapter will sometimes refer to as a compilation environment or compile-time environment, since it differs from the run-time environment that the implementation uses during interpretation of the compiled code. By explicitly passing the compilation environment to compile-time quotation functions, Mython's parameterized quotation allows users to inject code into the language compiler. Injected code can extend the language by modifying the compilation phases, which are visible to the compilation environment.

The Mython language is realized by the MyFront compiler, a tool for translating Mython into Python byte-code modules. This chapter introduces the Mython language, describes the implementation and usage of the MyFront tool, and shows how MyFront can be used to implement domain-specific optimizations for Python code.

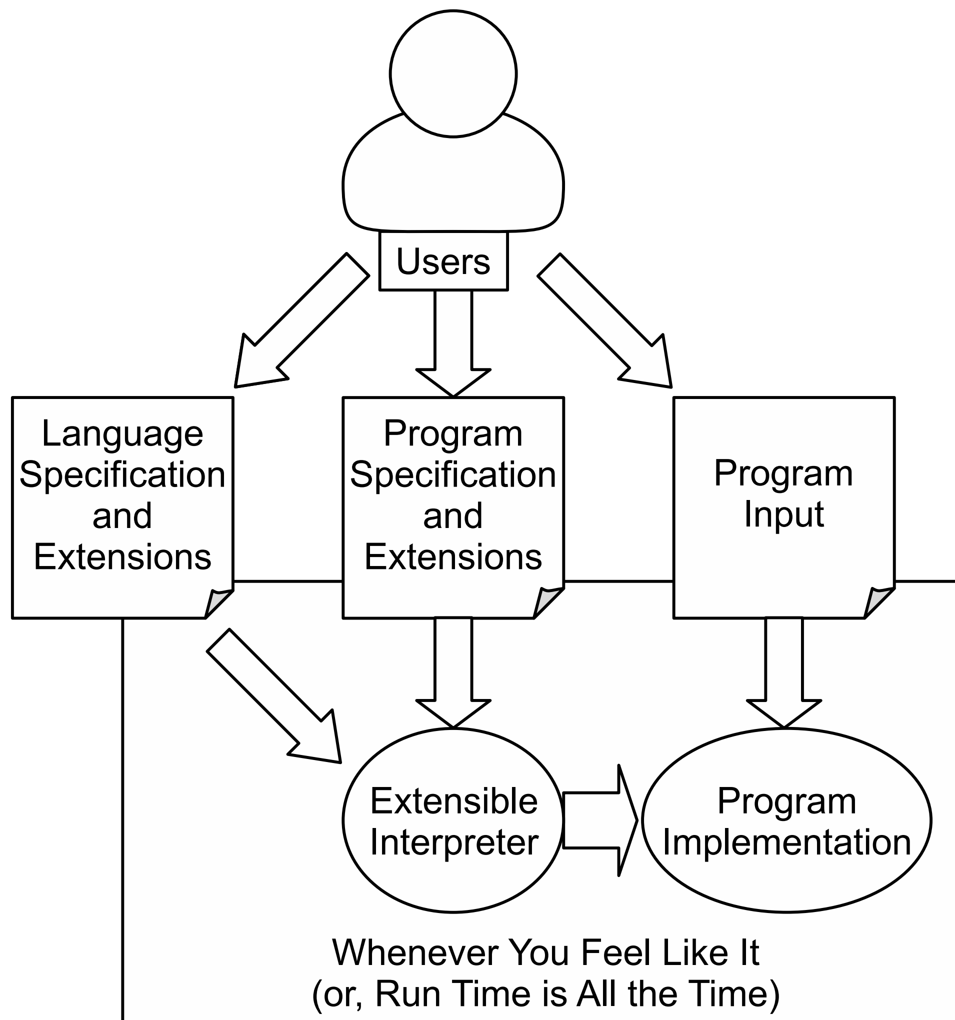


Figure 3.1: The Mython approach to language extension.

Mython lets you "make Python yours" in two senses. First, users can embed domain-specific languages using the Mython extensions. Mython employs parameterized quotation to not only provide a uniform means of inserting domain-specific code, but also to find errors in the embedded code. Mython can detect errors earlier in the development cycle than if the embedded code was simply inserted as a string. Second, the embedded code can be optimized by using the same Mython extensions to specify domain-specific optimizations. These optimizations can perform custom simplification of code, which can also provide further optimization opportunities to the native optimizer.

The remainder of this chapter has four parts. The first part, consisting of Section 3.1 and its subsections, reviews the details of parameterized quotation, how it has been extended to support compilation, and gives an overview of domain-specific optimizations. Sections 3.2, 3.3, and 3.4 provide the second part, defining both the Mython language, and discussing the MyFront compiler that implements the language. The third part, found in Section 3.5, provides an example of how a user-defined framework can be used to extend the Mython language, adding some of the optimizations described in the background material. The fourth and final part of this chapter gives a brief conclusion.

3.1 Language Origins

This section reviews staged multilanguage programming via parameterized quotation [52]. It refines earlier ideas by making evaluation stages explicit, defining acceptable quotation arguments, and explaining how the various evaluation environments are handled. This section concludes with an overview of domain-specific optimizations, a potential application of the quotation mechanism.

3.1.1 Staged, Multilanguage Programming, Redux

Figure 2.3 (page 41) communicated an earlier vision of staged, multilanguage programming. In that example, the language delimits parametric quotations using angle

and vertical bar braces (“`<|,|>`”), with the parameter separated from the rest of the quotation by another vertical bar (“`|`”). The arguments to quotation are expressions that evaluate to some syntax or parsing object, which is used to build a parse tree of the quoted string. The full quotation expression evaluates to the resulting parse tree. The example elaborates some parsing objects to include optional type annotations, and uses an interpreter, `transform()`, to compile and apply a set of rewrites to transform the embedded ML code into something the Python compiler can turn into a code object. The example dodges two key issues: the actual order of evaluating the quotations, and how environments are handled, especially in the presence of abstraction.

Since the example occurs in a hypothetical read-eval-print loop (REPL) of a Python-like language, each new prompt, signified by “`>>>`”, reflects a later evaluation stage. This fails to give insight into the various phases of the interpreter, and does not distinguish how the example differs from simply embedding code as string arguments to a parsing function. The vision behind staged multilanguage programming requires syntax errors in embedded code to be reported in the parsing phase. Using embedded code strings would not throw a syntax error until the evaluation phase, where the parsing function would run. This has implications on module compilation, where compile-time is clearly distinct from run-time.

All expressions in the example are evaluated in a global environment. The example does not show how quotation would work in the presence of local variables. If the permissible set of arguments to a quotation can be any expression in the environment, then abstraction can force evaluation of the parsing function to a later stage. Even if an implementor constrains the quotation parameter to a global environment, it would require the host language to have a notion of a static environment (an environment at compile-time). This would preclude highly dynamic languages (like Python) where global environments may be modified at run-time.

3.1.2 Staged, Multilanguage Programming, Fixed

This section fixes several of the ambiguities in the original presentation of parametric quotation. First, it requires the quotation argument to be explicitly evaluated at compile-time. The result of the argument expression is then applied explicitly to the quoted string. The result should be abstract syntax for an expression that will be evaluated at run-time. Second, it requires the quotation argument to be evaluated in a separate, compile-time environment. This requirement is forced in part by the first requirement. Finally, it allows the compile-time environment to be dynamic (at compile-time). This requires the environment to be explicitly threaded through the quotation argument expression.

Using subscripts to identify stage numbers, the resulting expression abstract syntax appears as follows:

$$e_1 ::= quote_1(e'_0, e''_2)$$

Where e_1 is a run-time expression, and e'_0 is a compile-time expression. Here e''_2 is a quoted expression, but is represented as a string at stage zero. This section and parts of this chapter refer to expressions evaluated at compile-time as stage-zero expressions, expressions evaluated at run-time as stage-one expressions, and quoted expressions as stage-two expressions (though these may be evaluated at stage-one via an interpreter function).

In a typed language, the addition of staging requires a new unary type constructor:

$$\tau_0 ::= Quote(\tau_1)$$

The following relates types at one stage to types at a lower stage number:

$$Quote(\tau_n) \in \tau_{n-1}$$

The result of evaluating e'_0 in some stage-zero environment Σ_0 should be a function that accepts a string and a stage-zero environment (which maps from identifiers to stage-zero values), and returns a value of some type α_1, and a stage-zero environment:

$$eval_0(e'_0, \Sigma_0) : string \times env_0 \rightarrow \alpha_1 \times env_0$$

Before leaving stage zero, the compiler rewrites the $quote_1$ expression to the result of applying e'_0 to the string representing e''_2 and the stage-zero environment. For stage zero to not result in an error state, this result should be a stage-zero value that represents a stage-one expression.

For example, a compiler encountering the expression "$quote_1(e'_0, e''_2)$" would first evaluate and bind the results of evaluating e'_0:

$$(f, \Sigma'_0) := eval_0(e'_0, \Sigma_0)$$

The compiler would then apply the resulting function, f, to the stage-two expression, e''_2, and the possibly modified stage-zero environment, Σ'_0:

$$(e'''_1, \Sigma''_0) := f(e''_2, \Sigma'_0)$$

The result is that the quotation expression is rewritten to e'''_1, or $quote_1(e'_0, e''_2) \mapsto e'''_1$. Ideally, e''_2 is of type α_2, and e'''_1 is of type $Quote(\alpha_2)$. This is not a requirement, since removing such a restraint allows the users to compile or partially evaluate quoted expressions.

The environment returned from the stage-zero evaluation of the quotation arguments is passed to later quotation expressions as they are found in the code. Therefore, the environment returned from the above stage-zero computation, Σ''_0, would be passed on to the next quotation expression found in the code. Environments chain over a prefix ordering of the expression tree. Quotation expressions may be rewritten to new quotation expressions, but they are not evaluated in the same stage. In this sense, quotation expressions cannot contain quotation expressions.

Constructs such as conditionals, abstraction, recursion, and other expressions that commonly affect the static environment are considered solely stage-one syntax, and do not change the stage-zero environment. Besides allowing extension of the stage zero environment for later quotation arguments, the environment may be used to

pass information to later compiler phases, such as the type checker, the optimization phases, and the code generator.

3.1.3 Domain-Specific Optimizations

The stage-zero environment allows implementation of domain-specific optimizations because it is visible to later compilation phases. Domain-specific optimizations (DSOs) are user-specified rewrites that perform algebraic optimization of domain-specific constructs (typically calls to functions in a library). Originally added to the GHC Haskell compiler by Peyton Jones et al. [47], domain-specific optimizations have been used to implement a faster Haskell string library [19] and perform stream fusion [18] in GHC programs.

A library writer can implement DSOs using algebraic rewrites. These rewrites perform optimizations such as strength reduction by matching an input program's abstract syntax to a pattern of function applications generalized over a set of metavariables. If a match occurs, the optimizer resolves metavariable assignments. The optimizer then constructs a piece of replacement abstract syntax, replacing metavariables with the terms bound during matching, and inserts it into the input program.

Figure 3.2 formalizes DSO rewrites as a set of metavariables, $\overrightarrow{x}$, a matching pattern over those metavariables, $p(\overrightarrow{x})$, and a replacement pattern, $p'(\overrightarrow{x})$. Matching patterns, which appear on the left-hand side of a rewrite, either match some known function call, $f()$, parameterized over matching subpatterns, $p_i(\overrightarrow{x})$, or bind to some metavariable, x. All metavariables in $\overrightarrow{x}$ must be bound at least once in the matching pattern, meaning that the set of free metavariables, $free(p(\overrightarrow{x}))$, should be equivalent to the set of generalized metavariables, $\overrightarrow{x}$. Metavariables that are used multiple times in a matching pattern should match equivalent terms. Replacement patterns, appearing on the right-hand side of a rewrite, either instantiate to a known function call, $f'()$, applied to a set of replacement subpattern arguments, $p'_i(\overrightarrow{x})$, or to a term bound to a metavariable, x'. The metavariable set, $\overrightarrow{x}$, coupled with the restriction all metavariables in that set must have at least one occurrence in the matching pattern, constrains replacement patterns to having no free metavariables.

$$\begin{array}{lcl} rewrite & := & \forall \overrightarrow{x}.p(\overrightarrow{x}) \rightarrow p'(\overrightarrow{x}) \\ & & \text{where} \quad free(p(\overrightarrow{x})) = \overrightarrow{x} \\ p(\overrightarrow{x}) & := & f(p_1(\overrightarrow{x}), ..., p_n(\overrightarrow{x})) \\ & | & x \quad \text{where } x \in \overrightarrow{x} \end{array}$$

Figure 3.2: A formalization of rewrite specifications.

The formalism in Figure 3.2 has three ambiguities: what constitutes a known function, the definition of term equivalence, and an instantiation method for terms bound to a metavariable. In languages with dynamic environments, including Python, any name not bound to a function's local name is not guaranteed to refer any statically known function. Unless static semantics are added in conjunction with control flow analysis, simply matching a name used at a function application site does not work, even in languages with fixed run-time namespaces that have higher-order functions. Later sections will look at this problem in more detail. Term equivalence is another ambiguity where implementors have a variety of options, including term identity, syntactic equivalence (assuming terms are free of side effects), or even syntactic equivalence of a normal form of each term (where the metavariable is bound to the "reduced", normal form). Finally, instantiation of metavariable bindings may either use a reference to the bound term, assuming instantiation to a directed acyclic graph form, or reconstruct a full copy of the bound term.

Shortcut deforestation is one example application of DSOs. Philip Wadler introduced deforestation as a program transformation [72]. Shortcut deforestation skips the transformation process, directly eliminating the intermediate tree. In a call-by-value language, nesting the $map()$ function is a common means of creating unintentional intermediate values. The expression $map(x, map(y, z))$ will create an intermediate list for the $map(y, z)$ subexpression, adding unnecessary allocation, initialization and deallocation expenses to the computation (though other optimizations might notice the intermediate value is not used later, and either allocate the intermediate value in the stack or do the second map in-place). Transforming the expression to avoid the intermediate value results in $map(x \circ y, z)$. This can be accomplished by

applying the following rewrite:

$$\forall xyz.map(x, map(y, z)) \rightarrow map(x \circ y, z)$$

Adding constants and operators to the pattern language is a useful extension of the rewrite formalism. This extension allows even more partial evaluation of domain-specific terms. For example, given a function that multiplies two three dimensional transformation matrices, $mat3mul()$, a function that generates a three dimensional rotation about the Z axis, $rot3z()$, the negation operator, and a function that generates the three dimensional identity matrix, $id3()$, the following rewrite performs simplification that would otherwise require a large degree of inlining and algebraic simplification:

$$\forall x.mat3mul(rot3z(x), rot3z(-x)) \rightarrow id3()$$

3.1.4 Python

Python is an object-oriented programming language with a C/C++-like surface syntax, but a Lisp-like semantics [65]. Python maintains the C syntax's imperative dichotomy, where statements and expressions are distinct syntactic constructs. Python provides common C control flow constructs such as `if` and `while`. Python also uses C-like expression syntax, with similar operator precedence and associativity. One way Python breaks with C syntax is its use of whitespace; Python uses newlines as statement delimiters, and indentation instead of braces. Python is dynamically typed and eliminates most of C's type declaration syntax (class definition does resemble C++ class declaration, but all member functions are defined in the class definition block, similar to Java).

Besides adopting a dynamic type system similar to Lisp and Smalltalk, Python also provides runtime code generation, introspection, and evaluation. Python code objects can be created at runtime using the built-in `compile()` and `eval()` functions, the `exec` statement, or using the `code()` constructor. The resulting object is a first-class value that can be inspected, and evaluated using either the `eval()` built-in

```
1  quote [cfront] c_ast:
2      int incnum (int x) {
3        return x+1;
4      }
5  incnum = cback(c_ast, 'incnum')
6  print incnum(41)
```

Figure 3.3: Example of a Mython program.

function or the `exec` statement. Unlike Lisp, Python does not provide either quotation or quasi-quotation as a way of creating syntactic objects that may be compiled into code objects. This design decision means that syntax errors in embedded code will not be found until run-time.

3.2 Language Syntax

This section introduces the Mython language. Mython extends Python by adding parameterized quotation definitions. Figure 3.3 gives an example of a Mython program that defines a quotation block named `c_ast`, which is translated to stage-two abstract syntax by the stage-zero translation function, `cfront`. At run-time (stage one), the quoted abstract syntax (stage-two code) is compiled into run-time code by the `cback` function. The result is a mixed language program that outputs 42.

The following subsections begin with a brief overview of the Python language. It then explains how Mython is a variant by describing the syntactic and semantic extensions made to the Python programming language. The next subsection discuses additions to the standard library, focusing on the new built-in functions that support quotation and code injection in Mython. The final subsection concludes with a brief description of the MyFront compiler, which implements the Mython language.

Section 3.1 gave examples of parameterized quotation as being syntactic expressions. Mython takes a different approach to parameterized quotation by making quotation a syntactic statement.

Figure 3.4 gives an example of the Mython abstract syntax for the program in Figure 3.3. Mython extends Python syntax at all three phases of parsing: the lexical

```
Module([
  QuoteDef(
    Some(Name('cfront', Load())),
    Some(Name('c_ast', Store())),
    'int incnum (int x) {\n  return x+1;\n}\n'),
  Assign([Name('incnum', Store())],
         Call(Name('cback', Load()),
              [Name('c_ast', Load()), Str('incnum')],
              [], None, None)),
  Print(None, [Call(Name('incnum', Load()),
                    [Num(41)], [], None, None)], True)
  ])
```

Figure 3.4: Abstract syntax for the Mython example.

phase, the parsing machinery, and the abstract syntax transformer. The following three subsections describe how each of these phases have been extended, and give an idea of how the input text in Figure 3.4 translates to the given abstract syntax. Note that both Figure 3.4 and later parts of this chapter follow the Python and Mython abstract syntax convention of annotating identifiers with either a nullary `Load` or `Store` constructor. This convention lets later compilation passes know if the identifier use is a name lookup or a name binder, respectively.

3.2.1 Lexical extensions

Mython adds the new lexical token type `QUOTED` to the lexer. The `QUOTED` token fills a similar role as a string, and can either be restricted to a single line or allow multiline quotation (multiline strings are permitted in Python and delimited by triple quotation marks). Unlike strings, quotation tokens are whitespace delimited, and require some knowledge of the syntactic state. Also unlike strings, the lexer removes indentation whitespace from each quoted line. Removing indentation whitespace allows Mython to quote itself, since the parser would otherwise interpret the additional whitespace as a naked indent token, causing a parse error.

The example program in Figure 3.3 shows an example of a multiline quotation. In this example, the quotation begins following the four space indentation on line 2.

```
(NAME, 'quote, 1) # (Keyword)
(LSQB, '[', 1)
(NAME, 'cfront', 1)
(RSQB, ']', 1)
(NAME, 'c_ast', 1)
(COLON, ':', 1)
(NEWLINE, '\n', 1)
(INDENT, '    ', 2)
(QUOTED, 'int incnum (int x) {\n  return x+1;\n}\n', 2)
(DEDENT, '', 5)
...
```

Figure 3.5: Part of the lexical stream for the Mython example.

Quotation stops at the end of line 4, delimited by the lack of indentation on line 5. The resulting token stream is shown in Figure 3.5, and illustrates how the leading whitespace on lines 2 through 4 has been stripped from the quoted string.

3.2.2 Concrete syntactic extensions

The Mython concrete syntax extends the Python syntax by adding the `quotedef` and `qsuite` nonterminals. These new nonterminals are defined (in the syntax of Python's parser generator, `pgen`[1]) as follows:

```
quotedef := 'quote' ['[' expr ']'] [NAME] ':' qsuite
qsuite   := QUOTED
          | NEWLINE INDENT QUOTED DEDENT
```

This syntax mirrors class definitions, which have the following grammar:

```
classdef := 'class' NAME ['(' [testlist] ')'] ':' suite
suite := simple_stmt
       | NEWLINE INDENT stmt+ DEDENT
```

1. The `pgen` language loosely follows extended Backus-Naur form, but adds parenthesis for grouping, and square brackets to delimit optional substrings.

In the Mython concrete syntax, the `quotedef` nonterminal is a peer to the `classdef` nonterminal, both being options for the `compound_stmt` nonterminal:

```
compound_stmt := ...
               | classdef
               | quotedef
```

The `qsuite` nonterminal contains the actual `QUOTED` token. As shown above, `qsuite` has productions for both the single line and multiline variants of the that lexical class.

3.2.3 Abstract syntax extensions

Python 2.5 extended the Python compiler infrastructure to include an abstract syntax tree intermediate representation [13]. Python formalizes its abstract syntax using the Abstract Syntax Definition Language (ASDL) [76]. Mython extends the abstract syntax by adding a `QuoteDef` constructor to the `stmt` abstract data type. This is formalized in ASDL as the following:

```
stmt = ...
     | QuoteDef (expr? lang, identifier? name, string body)
```

The `QuoteDef` constructor has three arguments. The first argument, `lang`, is an optional stage-zero expression that should evaluate to a language implementation function. If no implementation expression is present, Mython assumes the `body` argument is quoted Mython code. The second argument, `name`, is an optional identifier that is handed to the language implementation function, allowing implementors to associate a name with a quotation, or as part of a binding expression in the generated stage-one code. The final argument, `body`, is required and should contain the string matched by the `QUOTED` terminal.

Python and Mython use a transformation pass to translate a concrete syntax tree into an abstract syntax tree. The transformation rules are straightforward, with each `quotedef` transforming into a `QuoteDef` constructor. The optional `expr` child

```
tr( quotedef('quote', ':', qs) ) = QuoteDef(None, None, tr(qs))
tr( quotedef('quote', NAME, ':', qs) ) =
 QuoteDef(None, Some(Name(text(NAME), Store())), tr(qs))
tr( quotedef('quote', '[', e0, ']', ':', qs) ) =
 QuoteDef(Some(tr(e0)), None, tr(qs))
tr( quotedef('quote', '[', e0, ']', NAME, ':', qs) ) =
 QuoteDef(Some(tr(e0)), Some(Name(text(NAME), Store())), tr(qs))

tr( qsuite(QUOTED) ) = text(QUOTED)
tr( qsuite(NEWLINE, INDENT, QUOTED, DEDENT) ) = text(QUOTED)
```

Figure 3.6: Transformation functions from Mython concrete syntax to abstract syntax.

nonterminal translates to the `lang` argument of the constructor. The optional `NAME` child token translates to the `name` argument. Finally, the text of the contained `qsuite` nonterminal is passed as the `body` argument. Figure 3.6 defines a partial function, `tr()`, that translates from concrete syntax (represented as a term constructor and its child parse nodes) to abstract syntax. The given definition assumes the existence of a `text()` function that returns the text of a token as a string.

3.3 Language Semantics

Mython's operational semantics do not differ from the Python language. The Mython implementation follows the method outlined in Section 3.1.2, adding compile-time semantics for handling quotation. This section defines Mython's quotation semantics as a pair of rewrite rules from Mython abstract syntax into Python abstract syntax:

$$\begin{gathered}(\texttt{QuoteDef(None}, name_{opt}, body, \Sigma_0) \mapsto \\ (\texttt{QuoteDef(Some(Name('mython', Load()))}, name_{opt}, body, \Sigma_0)\end{gathered} \tag{3.1}$$

$$
\begin{aligned}
(\texttt{QuoteDef}(\texttt{Some}(lang), name_{opt}, body), \Sigma_0) \mapsto (stmts, \Sigma_1) \\
\text{where} \\
myeval = \Sigma_0[\texttt{'myeval'}] \\
(f, \Sigma_0') = myeval(lang, \Sigma_0), \\
(stmts, \Sigma_1) = f(name_{opt}, body, \Sigma_0')
\end{aligned}
\tag{3.2}
$$

Rule 3.1 replaces empty translation expressions with a reference to the default translation function, `mython()`, which is defined for stage zero in the initial environment (see Section 3.4.1). This rule means that a parameter-less quotation will quote Mython code. Rule 3.2 handles either the user-specified stage-zero expression, or the stage-zero expression inserted by Rule 3.1.

One should note that Rule 3.2 inserts abstract syntax that looks up the `mython` name in the compile-time environment. The rule does not insert a constant reference to the initial `mython()` built-in function (closure). This allows extension of the Mython language (for quotation) by rebinding the `mython()` quotation function in the compiler environment.

The final result of the rewrites should be a list of Python abstract syntax statements, and a new stage-zero environment. If the statement list is empty, the `QuoteDef` is simply removed from the stage-one AST. Otherwise, the list is inserted in place in the containing statement list.

3.3.1 Compile-time computations

Rule 3.2 performs two stage-zero computations. The first computation is the evaluation of the stage-zero expression, *lang*. The second computation is application of the first result to the quoted name, string and environment.

The compile-time expression is evaluated by a built-in function, `myeval()`, which is looked up in the stage-zero environment. This compile-time dynamic lookup allows extension of the Mython evaluator. The stage-zero expressions are still constrained, at time of writing, to the initial Mython syntax, since the semantic rewrites defined here are applied after the syntactic phases of the language. The built-in `myeval()`

function takes two arguments: the Mython abstract syntax for the expression, and an environment. The evaluator compiles the abstract syntax into Python byte-code, and then evaluates that byte-code in the given environment. The result is a Python object that should be callable (i.e., it should either be a function or an object that implements the `__call__()` special method). The dynamic type signature of `myeval()` is a logical extension of the *eval*() function described in Section 3.1.2, and is defined as follows:

$$\begin{aligned} quotefn &\equiv string\ opt \times string \times env \rightarrow \\ &\quad\ stmt_{pyast}\ list \times env \\ \texttt{myeval} &: expr_{myast} \times env \rightarrow quotefn \end{aligned}$$

The second computation applies the resulting callable object to the quotation name, the quotation string and the evaluation environment. If the result of the stage-zero expression fails either due to some internal error or it fails to conform to the type contract, a compile-time error is generated, and compilation is halted. Otherwise, the expected output of this second computation is a list of abstract syntax statements and a new environment.

At the time of writing, language semantics are constrained by the requirement that the output of stage-zero computations should consist of Python abstract syntax. This constraint precludes the case where the output of the second computation contains further stage-one `QuoteDef` statements. By unifying compile-time metaprogramming and staging, Mython could confuse users that are already versed in staged programming. Such users may forget that when using quotation to generate abstract syntax, they must explicitly transform it to Python abstract syntax before returning it to the compiler.

In future work, the constraint could be relaxed to permit full Mython abstract syntax output and handle nested quotation in one of two ways. In one instance the new quotations could be further expanded using the given rewrite rules, but this could allow unintentional non-termination of the compiler (not that the stage-zero expression or its result are guaranteed to halt either). In another implementation approach, the stage-one abstract syntax could simply be rewritten into a run-time

```
quote [X] x:
    xCode
# myeval('X', E_0)('x', 'xCode', E_0) => ast0, E_1
def spam ():
    quote [Y] y:
        yCode
    # myeval('Y', E_1)('y', 'yCode', E_1) => ast1, E_2
# Users might expect E_2 to be popped here...
quote [Z] z:
    zCode
# myeval('Z', E_2)('z', 'zCode', E_2) => ast2, E_3
```

Figure 3.7: Example of how the compilation environment flows through a Mython program.

call to the compiled expression code object, applied to the optional name, quotation string and the global stage-one environment. The second approach would allow syntax errors to be hidden at stage-one, passing them to a later computation stage.

3.3.2 Compile-time environments

Section 3.1.2 explained how the stage-zero environment was passed between quotation expressions. Mython quotation statements use the same strategy, namely prefix traversal of the quotation expressions, with the environment output of one quotation computation being passed to the next quotation. Figure 3.7 shows a partial Mython program, with environment inputs and outputs identified explicitly in the comments. Users may find this traversal strategy to be somewhat confusing, as the quotation-removal pass ignores lexical scope. Using the example in Figure 3.7, stage-zero symbols bound inside the `spam()` function will be visible to other stage-zero code following `spam()`. This behavior may change in later Mython implementations.

By providing compile-time metaprogramming, the Mython semantics are flexible enough to leave handling nested quotation and anti-quotation as implementation details for the domain-specific language implementor. Nested quotation ideally performs a double escape of quoted abstract syntax, making the resulting code

evaluate in "stage three". However, Mython does not currently define a compilation environment at stage one, which is the evaluation stage where the nested quotation should be compiled and escaped into abstract syntax. Anti-quotation at higher stages can be implemented by emitting code that evaluates at the next lower stage. For example, anti-quotation from stage two to a stage-one variable reference can be done by simply not escaping the name reference abstract syntax (in this case, it would generate `Name('yourname', Load())` instead of `Call(Name('Name', Load()), [Str('yourname'), Call(Name('Load', [], [], None, None))], [], None, None))`. Mython does not currently provide a mechanism for escaping from stage one into the stage-zero environment without going through the quotation mechanism.

Many Mython stage-zero functions fit into one of two patterns: pragma functions, and quotation functions. Pragma functions emit no stage-one code, but rather extend or enhance the compilation environment. Quotation functions escape stage-two code into abstract syntax objects at run-time, and either do not extend or "minimally" extend the stage-zero (compile-time) environment.

3.4 Implementation

3.4.1 Mython Standard Library

The Mython standard library must include several built-in functions to support native quotation, as well as provide support for extension of the compilation environment. The following subsections describe some of the functions that are defined in the initial stage-zero environment.

The `myparse()` built-in function

$$\texttt{myparse} : \mathit{string} \times \mathit{env} \rightarrow \mathit{mod}_{\mathit{myast}}$$

The `myparse()` function exposes the Mython syntactic front-end to the compiler environment. This function accepts a segment of Mython code, represented as a

string, and outputs a Mython abstract syntax tree (specifically returning a `Module` object, corresponding to the ASDL constructor of the same name). Additional information may be passed through an environment that maps from Mython identifiers to Python objects. By convention, the file name and the line number are represented in the environment, bound to the `'filename'` and `'lineno'` identifiers. If an error occurs during parsing, the front-end throws a syntax error exception, and includes file and line location information. The Mython front-end makes no attempt to perform error recovery and identify later syntax errors. This strategy reflects how the Python front-end handles syntax errors, with the detection and resolution of multiple syntax errors requiring multiple passes through the compiler.

Both the stock quotation and pragma functions, defined below, use the `myparse` function to parse their quoted text arguments.

The `mython()` built-in function

$$\texttt{mython} : quotefn$$

The `mython()` function acts as a stage-zero quotation function (see Section 3.3), parsing the quoted string using the Mython front-end. The parser outputs an abstract syntax tree, which is escaped into stage-two code using the `myescape()` built-in function (note the AST passed to the escape function is a module, but the resulting abstract syntax tree is an expression). If no name is given, the resulting stage-two code is emitted as an expression statement. If a name is given, the `mython()` function constructs stage-one abstract syntax that assigns the expression to the name argument, and the function extends the stage-zero environment with the original abstract syntax tree. This compilation environment extension allows the possibility of inlining or specialization in later compilation phases.

The following is the Python definition of the `mython()` built-in:

```
def mython (name, code, env0):
    env1 = env0.copy()
    ast = myparse(code, env0)
```

```
    esc_ast = myescape(ast)
    stmt = Expr(esc_ast)
    if name is not None:
        env1[name] = ast
        stmt = Assign(Name(name, Store()), esc_ast)
    return [stmt], env1
```

The `myescape()` *built-in function*

$$\texttt{myescape} : mod_{myast} \rightarrow expr_{pyast}$$

The `myescape()` function takes a Mython abstract syntax tree and outputs a Python abstract syntax tree that, when evaluated, constructs the input tree. The result is stage-two code, represented as a set of stage-one calls to the Mython abstract syntax constructors. For example, the Mython abstract syntax for a call to some function `foo()` would appear as follows:

```
Call(Name('foo', Load()), [], [], None, None)
```

The `myescape()` function would translate this into the following (where ellipsis are used for the empty list and `None` arguments to the `Call()` constructor):

```
Call(Name('Call', Load()),
    [Call(Name('Name', Load()),
          [Str('foo'),
           Call(Name('Load', Load()), ...)],
          ...)], ...)
```

Since the output of the escape function is only the Python subset of Mython abstract syntax, the Mython language does not initially allow quotations to compile to further quotations. Therefore, the standard library does not initially encounter the restrictions mentioned in Section 3.3.1.

The `myeval()` built-in function

$$\texttt{myeval} : mod_{myast} \times env \rightarrow \texttt{PyObject}$$

The purpose of the `myeval()` function is to provide a stage-zero Mython interpreter, accepting Mython abstract syntax and an environment as inputs and returning a stage-zero value as an output. The choice of calling this function `myeval()` may confuse native Python users, since this embedded Mython interpreter actually has both the effect of the Python `exec` statement and `eval()` built-in. The input type to `myeval()` is the *mod* type, which may be either an expression or a collection of statements (in the Python ASDL definition the *mod* type corresponds to the `Module`, `Interactive`, and `Suite` constructors). In the case where an expression is input, `myeval()` behaves similarly to the Python `eval()` function. Otherwise, `myeval()` behaves similarly to the Python `exec` statement, where assignments and definitions in the input code will extend the input environment argument in-place. While the `mython()` function expects a certain dynamic type from calling `myeval()`, the evaluation function is capable of returning any Python object. The lack of type checks in `myeval()` make the caller responsible for any type checking.

The `myfront()` built-in function

$$\texttt{myfront} : quotefn$$

The `myfront()` function is a stage-zero pragma function. The `myfront()` function is similar to the `mython()` function in the sense that the quoted text is parsed using the Mython front-end. Instead of escaping the resulting abstract syntax tree, `myfront()` evaluates it in the current compile-time environment using the `myeval()` function. The Python definition of `myfront()` is as follows:

```
def myfront (name, code, env0):
    ast = myparse(code)
    env1 = env0.copy()
    if name is not None:
```

```
        env1[name] = ast
    myeval(ast, env1)
    return ([], env1)
```

3.4.2 MyFront

MyFront is the initial Mython language implementation, and inherits its name from Cfront, the original C++ to C translator [58]. While it is possible to translate from Mython to Python using the simple rewrites defined in Section 3.3, the implementor decided enabling optimization and especially domain-specific optimizations would be better handled using a compiler. MyFront is similar in design to the actual Python compiler. In both compilers, the input source is a file that defines a module, which the compiler translates into a byte-code object. The Python compiler either evaluates the byte-code object into a module instance using the same virtual machine (VM) as the compiler, or the compiler marshals the module to disk, which can be loaded in a later run of the VM. At the time of writing, MyFront does not evaluate the module code object, and only supports marshalling of the module into a compiled Python module (typically using the same base-name of the module, but with the `.pyc` extension).

MyFront itself is a Python 2.5.1 program, and can be used from a UNIX command line as follows:

```
% MyFront.py my_module.my
```

This example would output a compiled Python module named "`my_module.pyc`", which can be imported directly into Python using the Python import statement, "`import my_module`". Flags input to MyFront from the command line will be placed in the compile-time environment as a list referenced by the "`argv`" name.

MyFront has three main phases of compilation. Each phase accepts some code representation and the compile-time environment. The output of each phase is some transformed code representation and a possibly modified environment. The phases are named in the compile-time environment as follows:

- `myfrontend()`: The `myfrontend()` function accepts Mython code as a string, expands any quotation statements as described in Section 3.3, and returns Python abstract syntax. The compiler uses the output environment to lookup the remaining compilation functions.

- `mybackend()`: The `mybackend()` function bound in the initial compilation environment accepts the Python abstract syntax representation of a module and returns a Python code object. By default, it does not change the output environment.

- `mymarshall()`: The `mymarshall()` function takes the output module code object and returns a marshalled string representation. It does not extend the environment.

3.5 Applications

Section 3.1.3 gave a high-level overview of domain-specific optimizations. This section describes how some of the ideas presented in Section 3.1.3 can be implemented in Mython. Section 3.5.1 begins with a discussion of how the MyFront-specific internals can be extended to add one or more optimization passes. Section 3.5.2 then gives an example of using Mython to perform shortcut deforestation of list comprehensions. This section concludes with a discussion of high-level operators, which can extend the Mython semantics to allow inlining, call specialization, and user-specified domain-specific optimizations.

3.5.1 Domain-specific Optimization Basics

As discussed in Section 3.4.2, MyFront's code generation back-end is composed into a stage-zero function named `mybackend()`. An optimization pass can plug into MyFront by composing the optimization function with the existing back-end and rebinding the result as the new language back-end. For example, the code in Figure 3.8 rebinds `mybackend` to a composition of the `dummy_opt()` function and the previous value of `mybackend`. The resulting back-end intercepts abstract syntax coming from the

```
quote [myfront]:
    def dummy_opt (tree, env):
        print tree
        return (tree, env)
    def compose_passes (p1, p2):
        return lambda tree, env : p2(*p1(tree, env))
    mybackend = compose_passes(dummy_opt, mybackend)
```

Figure 3.8: Adding an optimization phase to MyFront.

MyFront front-end, prints the tree, and then passes the (still unoptimized) tree on to the code generator.

MyFront also allows user-specified byte-code optimizers, which could plug into the compiler in a similar fashion. Unlike the abstract syntax tree optimizers, byte-code optimizers would have to intercept the output of the `mybackend()` function instead of its input.

The `simple_rw_opt()` function, shown in Figure 3.9, is an example of a tree transformation function that allows lower level tree rewrites to be added by further extension of the compile-time environment. For each tree node visited, the `simple_rw_opt()` function checks the environment for a rewrite function. In this scheme, rewrite functions are nominally indexed by the abstract syntax node constructor, so for some constructor named ***Node()***, the environment would be checked for a function named `rw_`***Node()***. If the rewrite function, `rw_`***Node()***, is not found in the environment, or the result of the rewrite function is `None`, then the transformer is recursively applied to the current root's child nodes. The root node is then reconstructed and returned. If the rewrite function succeeds, returning some tree instance, the transformer function returns that result.

As an example, one could use the example transformer to plug in a constant folding optimization for binary operations as follows:

```
quote [myfront]:
    def fold_binary_op (tree):
        if ((type(tree) == BinOp) and
```

```
def simple_rw_opt (tree, env):
    rw_result = None
    node_type_name, ctor, kids = explode(tree)
    rw_fn = env.get("rw_%s" % node_type_name, None)
    if rw_fn is not None:
        rw_result, env = rw_fn(tree, env)
    if rw_result is None:
        opt_kids = []
        for kid in kids:
            opt_kid, env = simple_rw_opt(kid, env)
            opt_kids.append(opt_kid)
        rw_result = ctor(opt_kids)
    return rw_result, env
```

Figure 3.9: The `simple_rw_opt()` function.

```
            (type(tree.left) == Num) and
            (type(tree.right) == Num)):
            op_fn = get_op(tree.op)
            return Num(op_fn(tree.left.n, tree.right.n))
        return None
    def rw_BinOp (tree, env):
        return fold_binary_op(tree), env
```

This example does not attempt to compose the new rewrite function with any previously existing rewrite, and will override any previous optimization specific to binary operation nodes. A user can fix this by using a guarded-choice strategy [70] to compose rewrites and rebinding the composed rewrite in the compile-time environment.

3.5.2 Shortcut Deforestation

The first example optimization in Section 3.1.3 eliminated the intermediate data structure generated by a nested use of *map*. While Python has a built-in `map()` function, the optimizer cannot ensure that the name is not rebound at run-time without changing the language semantics. Python's list-comprehension syntax provides an

opportunity to both avoid the issue of name capture and use abstract syntax rewrites to perform shortcut deforestation. Since list comprehensions have their own abstract syntax, the optimization machinery does not have to match function names, but only has to be extended to match abstract syntax. This subsection outlines how a user can build a set of domain-specific languages that help with the definition of rewrites.

To start, the user must extend Mython with two additional domain-specific languages. The first domain-specific language is a modification of Mython itself which allows users to create patterns of Mython abstract syntax. The `mypattern()` quotation function constructor defines this language, where the pattern language is parameterized over a set of metavariables. The second domain-specific language, implemented by the `myrewrite()` quotation function, composes two abstract syntax patterns already present in the compile-time environment, creates a rewrite function, and extends the rewrite infrastructure defined in Section 3.5.1.

Figure 3.10 gives some example Mython code that uses both `mypattern()` and `myrewrite()` to define a rewrite optimization. The example rewrites nested list comprehensions into a single comprehension, avoiding the construction of an intermediate list. The first quotation uses the pragma function `myfront()` to bind a tuple of names to `metavars` in the stage-zero environment. The `metavars` identifier appears as an argument to later calls of the `mypattern()` quotation function constructor. The stage-zero calls to `mypattern()` return a specialized Mython parser that generalizes Mython code into patterns over the set of passed metavariables. The next two quotation blocks define patterns for matching list comprehension abstract syntax. The final quotation defines a rewrite that uses the two patterns. The remainder of this section outlines more implementation details in support of this example.

Values in the pattern language consist of patterns, Mython expression syntax, $expr_{myast}$, or environments that map from metavariable names to Mython abstract syntax. The `mypattern()` function constructs pattern values by currying its argument of metavariables into a quotation function. The resulting quotation function uses a parser for Mython expressions to construct a tuple container for the metavariable list and the abstract syntax of the pattern expression. The quotation function then binds the pattern values to the quotation name in the stage-zero environment. In

```
quote [myfront]:
    metavars = ('x', 'y', 'z', 'i0', 'i1')
quote [mypattern(metavars)] p0:
    [x(i0) for i0 in [y(i1) for i1 in z]]
quote [mypattern(metavars)] p1:
    [x(y(i1)) for i1 in z]
quote [myrewrite]: p0 => p1
```

Figure 3.10: An example of using embedded DSL's in Mython to define an optimization.

Figure 3.10, the code constructs two patterns, binding them to `p0` and `p1`. If no name is given, Mython constructs the pattern, but the value immediately becomes garbage.

The `myrewrite()` quotation function defines the rewrite language, extending the pattern language by actually applying its operations. The surface syntax of the rewrite language is a pair of identifiers, which are delimited by a "double-arrow", '`=>`'. The quotation function performs the following operations:

- It interprets quoted code by looking up both identifiers in the compile-time environment, binding these patterns in its own name space.
- It then constructs a rewrite closure that matches the first pattern and constructs the second pattern upon a successful match.
- It extends the rewrite environment, using the method described at the end of Section 3.5.1.
- Finally, it binds the rewrite in the compile-time environment, if a name is given in the quote definition.

The rewrite function constructed by `myrewrite()` forms a closure, capturing the left-hand and right-hand patterns, originally bound in `myrewrite()`. When called by the rewrite infrastructure (at optimization time), the rewrite closure uses a function

to match the input expression to the left-hand pattern (bound during quotation expansion). If the match fails, the match function returns `None` and the rewrite closure passes this along to the rewrite infrastructure. Otherwise, the closure sends the match environment to a constructor function, along with the right-hand pattern. Finally the closure passes the resulting expression to the rewrite infrastructure, and the original input expression is replaced before being sent to the byte-code compiler.

3.5.3 High-level Operators

High-level operators (HLOPs) are a language construct introduced in the Manticore compiler [24]. In the context of Manticore, HLOPs are externally defined functions that the compiler loads and inlines at compile-time. Library implementors define HLOPs in one of the compiler's intermediate representations, BOM. The Manticore HLOPs are inlined during optimization of the BOM representation.

Mython allows users to add a similar facility by extension of the pattern matching and rewriting infrastructure described in Section 3.5.2. Users can modify the Mython semantics to use the compile-time environment as a static name space. To do this, the user changes the language semantics by injecting a custom optimizer. When MyFront runs the custom optimizer, it matches name references against the static HLOP environment. If a match is found, the HLOP expression is expanded. Ideally, the optimizer repeats this process until no new HLOP references are found, but the custom optimizer could also prohibit HLOPs from referencing HLOPs (this also removes the burden of dealing with recursion from the HLOP implementation). These approaches would solve some of the problems with domain-specific optimizations caused by having a dynamic run-time environment (see Section 3.1.3), since HLOP references are removed before code generation.

3.6 Summary of Mython

Developers have used Python as a glue language since its inception. By using C as a host language, Python has been able to provide users with language extensibility

and speed by allowing dynamic linkage of separately compiled C extension modules. Mython has the potential to take this extensibility further, allowing concrete syntax to be associated with either inline or separately defined extension code. This chapter has focused on the key mechanism used to provide this extensibility: parameterized quotation. Starting with the proposal in Section 2.2, this chapter has extended parameterized quotation to clearly define its interaction with a compile-time environment. It then defined the Mython language, extending Python to support parameterized quotation. It finally described how the the compiler for the resulting language can enable domain-specific optimizations. By developing several custom optimization related languages, later sections have shown that Mython can quickly provide both extensibility at the front-end, as well as optimize code on the back-end. As Mython provides more front-end support and better defines optimization conventions, it will provide developers the ability to both quickly develop new embedded languages that run faster and/or present a lower memory footprint. This hopefully leads to a virtuous cycle, where ease of language development speeds tool development, and faster tools accelerate further language development.

CHAPTER 4
DOMAIN-SPECIFIC OPTIMIZATIONS

This chapter describes domain-specific optimizations (DSO's). It begins by elaborating on why language implementors would want to include domain-specific optimizations in their languages. Section 4.2 then discusses how language implementors can modify their language front-ends to allow domain-specific optimization specification. Section 4.3 covers several methods for applying user-defined DSO's in an implementation's optimizer. Section 4.4 presents an analysis of the rewriting techniques. Section 4.5 looks at two existing domain-specific optimization implementations, concluding with a discussion about DSO's in Mython.

4.1 Motivations for Domain-Specific Optimizations

This section first explains the motivation behind adding domain-specific optimization to a language. It then gives several examples of domain-specific optimization and the resulting benefit the DSO's provide.

4.1.1 The Cost of Domain Abstraction

A domain-specific language often has a set of high-level, algebraic properties that exist independent of its underlying implementation. The majority of host languages fail to provide a means for expressing these properties. Host language optimizers are unable to reconstruct the high-level properties of a domain abstraction, and therefore lose an opportunity for using these properties to improve program performance. The result is either suboptimal program performance, or losing the development benefits of domain abstraction by performing hand optimization.

A host language that supports domain-specific optimization provides a means for expressing the high-level properties of an abstraction and sharing these properties

with the host optimizer. The resulting host optimizer uses these high-level properties to reason about domain-specific code both in terms of its abstraction layer and the details of its implementation. When used properly, a host language with domain-specific optimization allows abstraction implementors to hide complexity from their users, but do so without negatively impacting program performance.

DSO's can improve the run-time performance of domain-specific programs both in terms of speed and memory usage. The remaining two subsections show example problem domains and optimizations that improve program performance. Section 4.1.2 looks at a domain-specific language for two-dimensional graphics images. It then shows a set of algebraic properties that an optimizer can use to reduce the run-time complexity of domain-specific programs. Section 4.1.3 examines the high-level interactions of higher-order functions and the list domain. It then shows how a domain-specific optimization that captures this interaction can reduce a program's run-time memory consumption.

4.1.2 Strength Reduction

This subsection looks at algebraic simplification as a means for strength reduction. Strength reduction is the removal or simplification of a computation so that it takes fewer computation steps. This optimization technique ideally improves a program's run-time performance. Many optimizer implementations already contain a set of algebraic simplifications for reducing the strength of computation involving base types. For example, common optimizer implementations will commonly be able to recognize numerical simplifications and identities, such as addition by zero or multiplication by one.

Figure 4.1 shows a set of algebraic simplifications for the domain-specific language Pan [22, 54]. The first two simplifications give an example of how the empty image, `empty`, acts as an identity value for the `over` operator, which composes two images. Pan uses higher-order types, defining the image type to be the functions from points to colors. The transformation type, which `over` inhabits, is defined as the functions from images to images.

$$
\begin{aligned}
\texttt{over}(\texttt{empty}, i) &= i \\
\texttt{over}(i, \texttt{empty}) &= i \\
\texttt{fromPolar}(\texttt{toPolar}(f)) &= f \\
\texttt{toPolar}(\texttt{fromPolar}(f)) &= f \\
\texttt{rotate}(a_1, \texttt{rotate}(a_2, i)) &= \texttt{rotate}(a_1 + a_2, i) \\
\texttt{scale}((x_1, y_1), \texttt{scale}((x_2, y_2), i)) &= \texttt{scale}((x_1 * x_2, y_1 * y_2), i) \\
\texttt{dist0}(\texttt{fromPolar}((r, a))) &= r
\end{aligned}
$$

Figure 4.1: Algebraic simplifications for the Pan DSL.

A general-purpose optimizer may still be able to infer some of the simplifications in Figure 4.1. For example, the first two rewrites follow from the compile-time use of call specialization, inlining of the `empty` image function and the point-wise composition operator, and constant folding. Other optimizations such as the `rotate/rotate` simplification present too many obstacles for common general-purpose optimizers.

4.1.3 Shortcut Deforestation

Shortcut deforestation also relies on algebraic simplification, but as a means of eliminating intermediate data structures [72]. In functional languages, programs often modify the content of large data structures by duplicating the structure of the container and constructing new values. When composing two transformations over a container, the container structure must be duplicated first to store an intermediate value, and then a second time to hold the result. Construction of the temporary value adds to both the time required to evaluate the program due to allocation overhead, as well as the amount of memory required to run the program.

Figure 4.2 gives an example of the data structures involved during the evaluating a nested application of the `map` list transformation. Without deforestation optimization, an evaluator or compiler constructs a temporary value, `tmp`, that is used only once and is not visible to the programmer or the rest of the program. Using `map`/`map` fusion (shown as a dotted line), the optimizer can shortcut generation of the temporary value by performing a single `map`. This single `map` over the input list use the composition of the two functions, `f o g`, instead of evaluating each transformation on a separate pass.

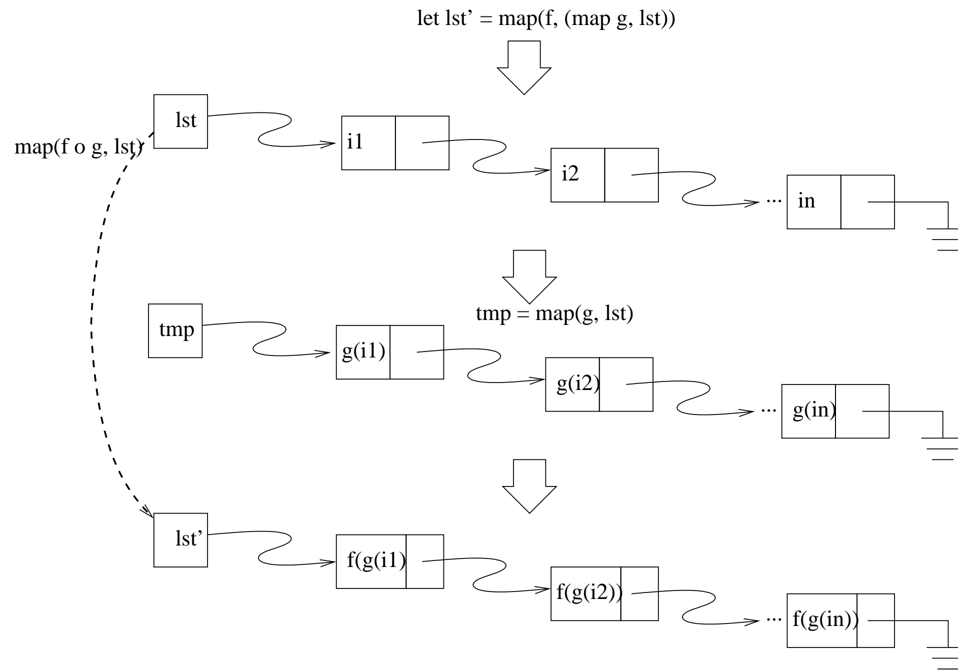


Figure 4.2: Shortcut deforestation for `map`/`map`.

Even if a host language implementation supports deforestation of standard containers, common embedded domain-specific language implementations can not take advantage of these optimizations. This limitation follows from the common practice of defining the abstract syntax of the embedded language as an algebraic data type (ADT). Programs in the embedded language represent their abstract syntax as a tree data structure that uses recursion in the ADT to relate parent and child tree nodes. Embedded language implementors must then define a new set of traversal and transformation operations, since the ADT is a novel data type. The unassisted optimizer can not apply shortcut deforestation to the resulting implementation, since abstraction has made both the syntax tree data type and its operators opaque. When available, embedded language implementors can use domain-specific optimizations to do shortcut deforestation, making their evaluators and compilers faster and require less memory.

4.2 User Specification of Domain-Specific Optimizations

Language implementors have several means for allowing users to input domain-specific optimizations. This section looks first at compiler pragmas, which are inline declarations in user code that modify compiler behavior. It then discusses using an embedded domain-specific language as a means of describing domain-specific optimizations. This section concludes by considering an external file or specification as a means of DSO input.

Compiler Pragmas Compiler pragmas use embedded syntax that is either defined as having syntax in the surface language, but compiler specific semantics, or embedded as comments (see also Section 6.1.4).

Embedded Domain-Specific Languages Language implementors have the option of extending their language's syntax to include syntax for specifying domain-specific optimizations. In the previous chapter, Figure 3.10 gave an example of using an embedded domain-specific optimization language in Mython.

External Libraries In contrast to embedded domain-specific languages, language implementors also have the option of using a separate domain-specific language or possibly a special subset or superset of the host language. Under this method, developers write their optimizations in separate files. The host language implementation or some build tool would then integrate the separate source, applying the user-defined optimizations where appropriate.

4.3 Rewriting Methods for Domain-Specific Optimization

Once a user defines a set of DSO's, it remains for a language's compiler to apply them. This subsection looks at several methods for applying domain-specific optimizations. The majority of these methods assume that implementors represent optimizations as rewrites of source terms. One means of representing these rewrites is as a set of atomic rewrites:

$$RW = \{rw | rw = \forall \overline{x}.p_{lhs}(\overline{x}) \rightarrow p_{rhs}(\overline{x})\}$$

Each rewrite universally quantifies a set of meta-variables, $\overline{x}$, which are distinct from variables in the source term language. These meta-variables parameterize a pair of patterns. Given a set of meta-variables, the pattern language expands to:

$$\begin{array}{rcl} p(\overline{x}) & := & t(p_1(\overline{x}), ..., p_n(\overline{x})) \\ & | & x \quad (\text{where } x \in \overline{x}) \end{array}$$

In this language, patterns either correspond to source terms which may contain subpatterns based on the arity of the source term, $t(...)$, or a meta-variable, x. Rewrites constrain the set of meta-variables, $mvs : p(\overline{x}) \rightarrow \mathcal{P}(\overline{x})$, appearing in the left-hand side to be exactly the set of meta-variables it is parameterized over, $mvs(p_{lhs}(\overline{x})) = \overline{x}$. Additionally, the right-hand pattern should be the same as or a subset of the set of meta-variables in the left-hand side, $mvs(p_{rhs}(\overline{x})) \subseteq mvs(p_{lhs}(\overline{x}))$. Depending on the rewrite method and the source term language, rewrites may further require that meta-variables in the left-hand pattern appear once and only once.

Figure 4.3 provides an example of a set of domain-specific optimizations. This example assumes that the source term language has at least form constructors, `concat`, `concatl`, `::` (the cons constructor), and `[]` (the empty list). The `concat` constructor, of arity two, corresponds to an operator that concatenates two strings. The operator creates and returns a new string value that is the concatenation of the two input strings. The `concatl` constructor, of arity one, corresponds to a primitive operator that concatenates a list of strings. This example assumes that the implementation of `concatl` constructs a new string by first summing the lengths of the input string list, creating a new string of the resulting length, and then performing concatenation in place on the new string.

For the concatenation of more than two strings, `concatl` is more efficient than nested `concat` operations. By virtue of constructing a new string value upon evaluation, nesting these two string operations in an expression creates a unnecessary temporary value, similar to the example in Section 4.1.3. All three example rewrites

$$
\begin{aligned}
rw_0 &= \forall x, y, z.\ \texttt{concat}(x, \texttt{concat}(y, z)) \rightarrow \texttt{concatl}(x :: y :: z :: [])\\
rw_1 &= \forall x, y, z.\ \texttt{concat}(\texttt{concat}(x, y), z) \rightarrow \texttt{concatl}(x :: y :: z :: [])\\
rw_2 &= \forall x, y.\ \texttt{concat}(x, \texttt{concatl}(y)) \rightarrow \texttt{concatl}(x :: y)\\
RW_0 &= \{rw_0, rw_1, rw_2\}
\end{aligned}
$$

Figure 4.3: Example rewrites for shortcut deforestation of string concatenation.

demonstrate a form of shortcut deforestation on strings. The first rewrite, rw_0, avoids creating a temporary string for the nested evaluation of `concat`. The second rewrite, rw_1, is a left-associative form of rw_0, resulting in the same right-hand side. The final rewrite, rw_2 allows the optimizer to chain successive string concatenations into a single list, avoiding not just one, but many temporary values.

4.3.1 Greedy Match and Replace

The simplest method for applying a set of rewrites is to greedily match and replace source terms. This method visits the source terms in either a bottom-up or top-down traversal. For each term visited, it checks to see if the term and its sub-terms match the left-hand pattern of a rewrite, $p_{lhs}(\overline{x})$. If the term matches, this algorithm constructs a matching environment that maps from meta-variables to source terms, $\sigma : MV \rightarrow \mathit{term}$. It then constructs new source terms using the right-hand pattern in the rewrite. During construction, it replaces any meta-variables in the pattern, x, with the terms found in the matching environment, $\sigma(x)$. Finally, the resulting source term replaces the initial matching term.

The greedy match and replace method gets its name by virtue of applying a rewrite as soon as an opportunity to do so has been identified. Hence the algorithm is greedy, looking only at local information, and not taking into account the possibility of occluding a more beneficial rewrite.

Figure 4.4 illustrates greedy rewriting using two term traversal strategies and the rewrites defined in Figure 4.3. The top transformation uses a top-down traversal,

$$\begin{array}{rc} & \underline{\texttt{concat}(v_0, \texttt{concat}(v_1, \texttt{concat}(v_2, v_3)))} \\ \rightarrow_{rw_0} & \texttt{concatl}(v_0 :: v_1 :: \texttt{concat}(v_2, v_3) :: []) \end{array}$$

$$\begin{array}{rc} & \texttt{concat}(v_0, \underline{\texttt{concat}(v_1, \texttt{concat}(v_2, v_3))}) \\ \rightarrow_{rw_0} & \underline{\texttt{concat}(v_0, \texttt{concatl}(v_1 :: v_2 :: v_3 :: []))} \\ \rightarrow_{rw_2} & \texttt{concatl}(v_0 :: v_1 :: v_2 :: v_3 :: []) \end{array}$$

Figure 4.4: Example of top-down versus bottom-up traversal for greedy rewriting.

greedily applying rewrites starting with the outer term. The top-down traversal successfully matches the outer term with the left-hand side of rw_0, transforming it into the second line. It then visits the nested sub-terms of the resulting expression, none of which match any left-hand side patterns in RW_0. Having checked that no further rewrites apply, the top-down visitor terminates.

The bottom transformation uses a bottom-up traversal strategy, first trying to match and apply any inner (child) terms to the rewrite set before examining an outer (parent) term. Using this visitation strategy, the optimizer first matches the child `concat` term of the top-most term (after failing to match v_0, v_1, v_2, v_3, and $\texttt{concat}(v_2, v_3)$.) Applying rw_0 to the child term results in the second line. The optimizer then checks the parent term, which matches rw_2. The resulting term appears on the bottom line, and avoids construction of two temporary string values in comparison with the original term. Having visited all the terms, the bottom-up strategy terminates.

Figure 4.4 demonstrates how greedy matching can occlude rewrites. In the top case, greedily applying rw_0 to the parent term prevented the optimizer from applying rw_0 to the child term and then rw_2 to that result. By not rewriting the parent immediately, the optimizer have arrived at the more globally optimal result shown in the lower example. This example also demonstrates how the traversal strategy can interact with a set of rewrites.

4.3.2 Bottom-up Rewriting

Bottom-up rewriting attempts to overcome some of the limitations of the greedy match and replace method. In its simplest form, this algorithm performs two passes on source terms. The first pass labels each source term with a state. This state identifies the various rewrites that could apply to a term. The second pass then applies rewrites based on the set of states that represent an optimal rewrite of all analyzed terms under some metric. Determination of this metric requires some measure of rewrite benefit, $b : RW \rightarrow \mathbb{N}$. Language implementations that employ this algorithm must allow users to define the benefit of a rewrite, and either require this for each rewrite, or have some default benefit.

This section presents a bottom-up rewriting algorithm that is a variation of the IBURG algorithm [25]. Fraser et al. originally designed the IBURG algorithm for instruction selection on complex instruction-set computer (CISC) architectures. The following algorithm modifies several features of IBURG. First, the source and target languages are identical. Second, it uses a wild-card nonterminal that matches all terms. Finally, it uses a benefit model, where the matching algorithm attempts to maximize benefit instead of minimizing cost.

To determine the matching state of a source term, bottom-up rewriting requires the creation of a matching grammar. The matching grammar, G_{match}, is a special form of context-free grammar (CFG) which defines a set of nonterminal tokens, N_{match}, and a set of productions, P_{match}. As mentioned above, this modified version of the IBURG matching grammar extends the context-free grammar with a special nonterminal token, "$\star$", that matches any term.

Productions in the matching grammar take the following form:

$$nt_0 ::= \texttt{T}\ nt_1\ ...\ nt_n\ (b)$$

Each nt in the production is a nonterminal in the matching grammar, $\forall 0 \leq i \leq n.nt_i \in N_{match}$. The `T` corresponds to a source term, and the number of nonterminals to its right should be equal to the source term's arity. The b is a measure of benefit, and corresponds to the benefit metric associated to a rewrite, and should either be

positive or zero. If the benefit is non-zero, the production should correspond to matching the left-hand pattern of a rewrite with the same benefit, $p_{lhs}(\overline{x})$.

Bottom-up matching requires that the pattern terms in the rewrite language be transformed into a set of productions in the matching grammar. The pattern transformer must first assign a nonterminal to each pattern term. The following function satisfies this requirement:

$$\begin{aligned} \mathtt{nt}(\mathtt{T}(p_1(\overline{x}),...,p_n(\overline{x}))) &= nt \text{ where } nt \text{ is fresh.} \\ \mathtt{nt}(x) &= \star \end{aligned}$$

Using the set of new nonterminals, the transformer must then create a set of new productions for each pattern term that matches a source term, $p_t(\overline{x}) = t(p_1(\overline{x}),...,p_n(\overline{x}))$:

$$\mathtt{tr}(p_t(\overline{x}), b) = \left\{\mathtt{nt}(p_t(\overline{x})) ::= \mathtt{T}\prod_{i=1}^{n}\mathtt{nt}(p_i(\overline{x}))(b)\right\} \cup \bigcup_{i=1}^{n}\mathtt{tr}(p_i(\overline{x}), 0)$$

This function uses the product of nonterminals, $\prod$, as shorthand for concatenation of the nonterminals for all the sub-patterns in $p_t(\overline{x})$.

The following example demonstrates transforming a set of rewrites given in Figure 4.3 into a matching grammar, and then using this to transform the term shown in Figure 4.4. This example assumes that a user provided the following benefits for the rewrites in RW_0:

$$\forall rw_i \in RW_0.\ b(rw_i) = 1$$

Using transformation defined above, this method first generates the set of nonterminals in the matching grammar. A production implementation will most likely generate nonterminals on the fly. The example rewrite transformer traverses the left-hand patterns of the given rewrites to arrive at the following nonterminal associations:

$$
\begin{aligned}
\texttt{nt}(\texttt{concat}(x, \texttt{concat}(y, z))) &= nt_0 \\
\texttt{nt}(\texttt{concat}(y, z)) &= nt_1 \\
\texttt{nt}(\texttt{concat}(x, \texttt{concat}(y, z))) &= nt_2 \\
\texttt{nt}(\texttt{concat}(y, z)) &= nt_3 \\
\texttt{nt}(\texttt{concat}(x, \texttt{concatl}(y))) &= nt_4 \\
\texttt{nt}(\texttt{concatl}(y)) &= nt_5
\end{aligned}
$$

Once the set of nonterminals is decided, the transformer constructs productions for each nonterminal:

$$
\begin{aligned}
nt_0 &::= \texttt{concat} \star nt_1 && (1) \\
nt_1 &::= \texttt{concat} \star \star && (0) \\
nt_2 &::= \texttt{concat}\ nt_3 \star && (1) \\
nt_3 &::= \texttt{concat} \star \star && (0) \\
nt_4 &::= \texttt{concat} \star nt_5 && (1) \\
nt_5 &::= \texttt{concatl} \star && (0)
\end{aligned}
$$

The productions for nt_1 and nt_3 have identical right-hand sides, and do not bind to a rewrite. For these reasons, an implementation of this method can merge both nonterminal symbols and productions into a single nonterminal and production. This also results in a time and space savings during the course of bottom-up matching, since the state space is exponential in the number of nonterminals. Further examples below assume that nt_1 replaces all right-hand occurrences of nt_3 in the matching grammar and the production for nt_3 is elided from the matching grammar.

Once the rewriter creates a matching grammar, it next labels each source term with a state. The state of some term, $\texttt{state}(t)$, is a set of pairs that define a partial relation from nonterminals in the matching grammar to the maximal benefit obtained by interpreting the given term as the given nonterminal. The state labelling algorithm must traverse terms in a bottom-up fashion since the state of child terms must be known before a parent term's state can be determined.

The state labelling algorithm determines the state of a source term by comparing the set of candidate strings the term could match with productions in the matching

grammar. The set of candidate strings for some term t, $\texttt{cands}(t)$, is the Cartesian product of a singleton term constructor set and the sets of nonterminals matched by sub-terms. Given a term, $t = \texttt{Term}(t_1, \ldots, t_n)$, the following defines the set of candidate strings for that term:

$$\begin{aligned}\texttt{cands}(t) \quad = \quad \big\{\texttt{Term}\; subnt_1 \; \ldots \; subnt_n \quad \big| \quad & subnt_1 \in \texttt{nonterms}(t_1), \\ & \ldots, \\ & subnt_n \in \texttt{nonterms}(t_n)\big\} \\ \text{where } \texttt{nonterms}(t) = \big\{nt \mid (nt, x) \in \texttt{state}(t)\big\}\end{aligned}$$

The algorithm uses $\texttt{cands}(t)$ to determine if any candidate string matches the right-hand side of a production. For each production that applies, the algorithm assesses the benefit of applying that production. This benefit is the sum of the child benefits, based on the nonterminal and benefit pair corresponding to interpreting the child as the nonterminal in the string, and the benefit associated with the production in the matching grammar. Finally, the system adds the left-hand nonterminal and benefit to the term state.

Many language implementations represent term expressions using a static single assignment (SSA) representation. This representation flattens the tree structure of source terms, binding each "node" in the tree to a unique identifier, and ordering these bindings in a bottom-up or postfix order. For example, a hypothetical compiler using a SSA representation would transform the source term from Figure 4.4 into the following:

$$\begin{aligned} temp_0 &= \texttt{concat}(v_2, v_3) \\ temp_1 &= \texttt{concat}(v_1, temp_0) \\ temp_2 &= \texttt{concat}(v_0, temp_1) \end{aligned}$$

The compiler then applies the state labelling algorithm sketched out above, but modified to apply to both the terms themselves and their bindings, since the bindings are unique and therefore synonymous with the terms. Instead of a depth-first, postfix

traversal of the source terms, the compiler linearly labels the binding occurrences, resulting in the following state map:

$$\begin{aligned}
\texttt{state}(temp_0) &= \{(\star, 0), (nt_1, 0)\} \\
\texttt{state}(temp_1) &= \{(\star, 0), (nt_1, 0), (nt_0, 1)\} \\
\texttt{state}(temp_2) &= \{(\star, 0), (nt_1, 0), (nt_0, 1)\}
\end{aligned}$$

The compiler then performs a top-down application of rewrites. In the SSA form used above, top-down rewrite application corresponds to starting at the last binding and moving upward. For each term, the optimizer applies rewrites based on the maximal benefit nonterminal found in the term's state. This results in the following output from the extended optimizer:

$$\begin{aligned}
temp_3 &= ::(v_3, nil) \\
temp_4 &= ::(v_2, temp_3) \\
temp_5 &= ::(v_1, temp_4) \\
temp_1 &= \texttt{concatl}(temp_5) \\
temp_2 &= \texttt{concat}(v_0, temp_1)
\end{aligned}$$

The result is essentially the same output as the greedy top-down optimizer, but in SSA form. This result is still one rewrite short of the following globally optimal term:

$$\texttt{concatl}([v_0, v_1, v_2, v_3])$$

This example illustrates a shortcoming of the presented state labelling algorithm. The algorithm only considers child terms as they exist at labelling time. It fails to consider what the child terms could become following rewrite application. The optimizer may be able to overcome this limitation by speculatively applying rewrites as a part of determining term state. This modification would apply applicable rewrites at labelling time, label the resulting transformed terms, but ascribe the resulting nonterminal and benefit to the original top-level term. This approach would result in the following state labelling for the previous example:

$$
\begin{array}{lcl}
\texttt{terms}(t(p_1(\overline{x}, ..., p_n(\overline{x})) & = & \{t\} \cup \bigcup_{1 \leq i \leq n} \texttt{terms}(p_i(\overline{x})) \\
\texttt{terms}(mv) & = & \emptyset \\
\texttt{dom}(rw) & = & \texttt{terms}(p_{lhs}) \\
\texttt{range}(rw) & = & \texttt{terms}(p_{rhs}) \\
rw_i \sqsupseteq_{rw} rw_j & \stackrel{def}{=} & \texttt{range}(rw_i) \cap \texttt{dom}(rw_j) \neq \emptyset
\end{array}
$$

Figure 4.5: A relation on rewrites.

$$
\begin{array}{lcl}
\texttt{state}(temp_0) & = & \{(\star, 0), (nt_1, 0)\} \\
\texttt{state}(temp_1) & = & \{(\star, 0), (nt_5, 1), (nt_1, 0), (nt_0, 1)\} \\
\texttt{state}(temp_2) & = & \{(\star, 0), (nt_4, 2), (nt_1, 0), (nt_0, 1)\}
\end{array}
$$

The result of applying rewrites given these state labels would result in the smallest number of intermediate strings for the example source term.

4.3.3 Tiered Rewriting

Tiered rewriting is another means of delaying the application of rewrites with the intent of increasing optimality at a global level. This method partitions a set of rewrites into ordered tiers. Each tier of rewrites will only rewrite to terms matched by rewrites in the current and any lower tier.

Figure 4.5 defines a relation on rewrites, $\sqsupseteq_{rw}$. The relation uses two sets. The $\texttt{dom}(rw)$ set is the set of term constructors that appear in the left-hand pattern of a rewrite. The $\texttt{range}(rw)$ set contains the term constructors that appear in the right-hand pattern of a rewrite. Given these sets, a rewrite, rw_i, has the potential to expose additional optimization opportunities for another rewrite, rw_j, when the terms rw_i constructs, $\texttt{range}(rw_i)$, is not mutually exclusive with the set of terms that rw_j rewrites, $\texttt{dom}(rw_j)$. The relation notes this relationship by ordering application of rw_i before application of rw_j, $rw_i \sqsupseteq_{rw} rw_j$.

The following gives the left-hand and right-hand term sets for the example set of string rewrites, RW_0:

$$\begin{array}{lcl|lcl} \mathtt{dom}(rw_0) & = & \{\mathtt{concat}\} & \mathtt{range}(rw_0) & = & \{\mathtt{concatl}, ::\} \\ \mathtt{dom}(rw_1) & = & \{\mathtt{concat}\} & \mathtt{range}(rw_1) & = & \{\mathtt{concatl}, ::\} \\ \mathtt{dom}(rw_2) & = & \{\mathtt{concat}, \mathtt{concatl}\} & \mathtt{range}(rw_2) & = & \{\mathtt{concatl}, ::\} \end{array}$$

Based on these sets, the rewrites in the RW_0 example have the following relationships:

$$rw_0 \sqsupseteq_{rw} rw_2, rw_1 \sqsupseteq_{rw} rw_2, rw_2 \sqsupseteq_{rw} rw_2$$

Tiers partition a set of rewrites, RW, with each tier being a non-empty subset of the rewrite set, $\emptyset \subset T_i \subseteq RW$, having a number, i, and being mutually exclusive with the other tiers, $\forall T_i \forall T_j, i \neq j, T_i \cap T_j = \emptyset$. The ordering of the tier numbers respect the $\sqsupseteq_{rw}$ relationship, such that the following holds:

$$\forall T_n.\forall i < n.\forall rw_i \in T_i, rw_n \in T_n.rw_n \sqsupseteq_{rw} rw_i$$

The following algorithm partitions a set of rewrites into tiers with the properties given above. First, the partitioning algorithm creates a directed graph where the vertices are sets of rewrites and the edges reflect the relation, $\sqsupseteq_{rw}$, generalized to sets, $\sqsupseteq_{rws}$. Figure 4.6 defines the digraph structure and relationships. This constructs the initial digraph, G^0_{RW}, by defining each initial vertex as the singleton set of a rewrite in the rewrite set. Second, it constructs G^1_{RW} by collapsing each strongly connected component in the directed graph. This eliminates any cycles in the graph and ensures the next step will terminate. Third, the algorithm iterates over G^i_{RW}, constructing T_i and G^{i+1}_{RW} as shown. This step starts with $i = 1$ and ends when $i = j$ such that $V^j_{RW} = \emptyset$. The i-th sub-step determines the set of vertices that do not have any (remaining) in-edges, $\mathtt{noins}(G^i_{RW})$, adds the rewrites they contain to the current tier, T_i, and removes these vertices from the graph used in the next sub-step, G^{i+1}_{RW}.

Applying the partitioning algorithm to the set of example rewrites, RW_0, yields the following two tiers:

$$T_1 = \{rw_0, rw_1\}, T_2 = \{rw_2\}$$

$$
\begin{array}{lcl}
G_{RW} & = & (V_{RW}, E_{RW}) \\
V_{RW} & = & \{rws | rws = \{rw | rw \in RW\}\} \\
V_i \sqsupset_{rws} V_j & \stackrel{def}{=} & (\bigcup_{rw_i \in V_i} \mathtt{range}(rw_i)) \cap (\bigcup_{rw_j \in V_j} \mathtt{dom}(rw_j)) \neq \emptyset \\
E_{RW} & = & \{(V_i, V_j) | V_i \sqsupset_{rws} V_j\} \\
G^0_{RW} & = & (V^0_{RW}, E^0_{RW}) \\
V^0_{RW} & = & \{\{rw\} | rw \in RW\} \\
\mathtt{noins}(G) & = & \{V_i | G = (V, E), V_i \in V, (\nexists V_j.(V_j, V_i) \in E)\} \\
T_i & = & \bigcup_{V_j \in \mathtt{noins}(G^i_{RW})} V_j \\
V^{i+1}_{RW} & = & V^i_{RW} / \mathtt{noins}(G^i) \\
E^{i+1}_{RW} & = & E^i_{RW} / \{(V_j, V_k) | V_j \in \mathtt{noins}(G^i)\} \\
G^{i+1}_{RW} & = & (V^{i+1}_{RW}, E^{i+1}_{RW})
\end{array}
$$

Figure 4.6: Relationships for partitioning rewrites into tiers.

A rewriting optimizer may use this partition information in conjunction with other rewrite algorithms to create an optimizer. For example, an optimizer may accept rewrites, partition these, and use a separate rewrite pass for each tier. Ideally, by running each rewrite pass to a fixed point, the optimizer exposes a maximal set of rewrite opportunities to the following tiers. In the running example, this information does not help much, since both the example term and set of rewrites are so small.

4.3.4 Rewrite Strategies

The Stratego language, first encountered in Chapter 2, is a domain-specific language for term traversal and rewriting. Language implementors may choose this method over the others because it permits more expressive rewrites, and this flexibility can be reflected back to the DSO language used by developers. The following subsection explains how rewrites can be specified as strategies, how these strategies can be composed, and how using additional strategy combinators make rewrites more extensible

The Stratego language provides syntactic sugar for defining term rewrites. Using the rewrite syntax, a user can write the DSO "$rw = \forall \overline{x}.p_{lhs}(\overline{x}) \rightarrow p_{rhs}(\overline{x})$" as "$rw$: $p_{lhs}(\overline{x})$ `=>` $p_{rhs}(\overline{x})$". The concrete syntax for rewrites has the following pattern:

```
label : lhs => rhs
```

The term pattern language, used for the `lhs` and `rhs` sub-strategies, is similar to the syntax developed at the beginning of this section, but does not quantify the pattern variables. When `label` is unique, the rewrite strategy syntax expands into the following strategy syntax:

$$\texttt{label} = \mathit{scope}(?\texttt{lhs}; !\texttt{rhs})$$

The ? combinator converts the left-hand term pattern into a matching strategy, where term constructors must match the current term and meta-variables extend the environment. The ! combinator converts the right-hand term pattern into a aggregate term constructor. A meta-variables term constructor looks for the meta-variable in the current environment, failing if the environment does not contain the meta-variable. The ; combinator sequentially composes the matching strategy with the constructor strategy, applying the construction strategy only in the event that the matching strategy succeeds. The sequential composition combinator preserves the environment between strategies, ensuring that meta-variables bound by the matching strategy can be referenced in the constructor strategy. Finally, the outer scoping combinator keeps the meta-variables bound in the matching strategy from escaping the rewrite strategy.

DSO implementors can encode each of the rewriting methods described earlier in this section as a composition of rewrite strategies. For example, an implementor can implement a greedy bottom-up rewriter by specifying the rewrites, composing them into a single strategy, and finally wrapping the top-level rewrite strategy into a traversal strategy. Figure 4.7 illustrates a translation and composition of the rewrites from Figure 4.3.

A Stratego developer can either compose rewrites implicitly or explicitly. Stratego allows implicit rewrite composition when a rewrite's label is already defined. When a rewrite's label is not unique, Stratego composes the new rewrite with the old rewrite using the non-deterministic choice combinator, +. Figure 4.7 composes the rewrites explicitly using the non-deterministic choice combinator. A developer could achieve

```
module string-dsos

rules
  rw0 : concat(x, concat(y, z)) -> concatl(x :: y :: z :: [])
  rw1 : concat(concat(x, y), z) -> concatl(x :: y :: z :: [])
  rw2 : concat(x, concatl(y)) -> concatl(x :: y)

strategies
  RW0 = rw0 + rw1 + rw2
  greedy-rw = repeat(oncebu(RW0))
```

Figure 4.7: A greedy rewriter in Stratego.

the same composition as the explicit form in the illustration by using the "`RW0`" label for all three rewrites, and omitting the second to last line.

The example handles term traversal using the `repeat` and `oncebu` combinators, which are a part of the Stratego library. The `repeat` combinator recursively applies the inner strategy until it fails, returning either the last successful strategy result or the initial term, if the inner strategy failed on the first try. The `oncebu` combinator performs a bottom-up traversal on the current term, succeeding if the inner strategy succeeds on the current term or a sub-term.

Strategies are more expressive than the other rewrite methods considered in this section. This follows by virtue of permitting a "where" clause in the rewrite syntax. This "where" clause allows insertion of arbitrary computation into the evaluation of a rewrite.

```
label : lhs => rhs where inter
```

This desugars into the following:

$$\texttt{label} = \mathit{scope}(?\texttt{lhs}; \mathit{where}(\texttt{inter}); !\texttt{rhs})$$

DSO writers can use the intermediary strategy, `inter`, both as an additional predicate for matching and a means of extending the meta-variable environment of the term constructor. The *where* combinator discards the term and preserves the environment as modified by the inner strategy, but will fail if the inner strategy fails.

4.4 Quantitative Analysis

This section looks at the quantitative characteristics of five different approaches to rewriting: greedy pattern matching (top-down to fixed-point), bottom-up, tiered greedy pattern matching, tiered bottom-up, and strategies. It presents a framework for comparative analysis of these methods, as well as a domain for these strategies to be applied. Section 4.4.2 presents and summarizes the time performance of these algorithms. Section 4.4.3 attempts to show that a tiered approach to rewriting can yield more rewrite opportunities.

4.4.1 An Analysis Framework

This subsection defines the framework used to analyze the five rewrite algorithms. The test framework implements each rewrite algorithm as a curried combinator that accepts a set of rewrites and a set of terms. The set of rewrites is similar to the example rewrite set, RW_0, found in Section 4.3. As discussed in that section, these rewrites perform tree deforestation, replacing a binary operator with a n-ary operator that accepts a list. This reduces the amount of memory required to hold intermediate results.

The test framework combines three kinds of input to generate the set of input test terms. First, term size determines the number of term constructors used in generating the test input. Second, the framework accepts a pair of term constructors, a null-ary constructor for leaf terms, and a binary constructor for parent terms. Finally, the framework can generate either balanced terms, left-associative terms, or right-associative terms.

The framework collects a variety of metrics for the rewrite algorithms. First, it times how long each algorithm took to process an input term. Second, it counts the number of output terms, and the number of remaining constructors in the domain of the rewriter. It also counts how many times the underlying rewrite function is called before the framework arrives at a fixed point output term. For the strategy-based optimizer, the framework also timed the amount of time required to translate from the native optimization terms to the term format used by the strategy module and

back again. Each of these metrics are used in the next subsection to answer a set of questions.

4.4.2 Rewrite Performance

This subsection looks at the following questions:

- What is the run-time complexity of the considered algorithms?
- Does the overhead added by state determination in bottom-up rewriting only increase the time spent in the optimization pass by some constant factor (hopefully less than two)?
- Can strategies that do not use a “where” clause have comparable run-time performance as the other rewrite techniques?

The algorithms described in Section 4.3 should have linear time performance, $O(n)$, in the number of the input terms, n. Where the greedy algorithm performs a single match and possibly rewrite pass, the bottom-up algorithm performs two passes: a labeling pass and a rewrite pass, which only changes the linear performance by a constant factor. When these rewrite algorithms are run until the framework obtains a fixed point output term, they have a worst case performance of $O(n^2)$. Tiers add additional passes of the $O(n^2)$ fixpoint algorithms, but this multiplier is negligible when the number of rewrites is less than the number of input terms (where the number of tiers is bound by the number of rewrites).

Figure 4.8 graphs the run-times in microseconds for the various rewrite algorithms. The input terms were balanced, and in the domain of the rewrite set. The greedy optimizer performed best in most cases. In some cases, the tiered greedy optimizer outperformed the non-tiered greedy optimizer. This comes at a cost in quality of output, with the simple greedy algorithm also resulting in the fewest intermediate results (see Section 4.4.3).

Figure 4.9 traces the run-time performance of the five rewriting optimizers on input terms that do not match any rewrites. The times given reflect the cost of term

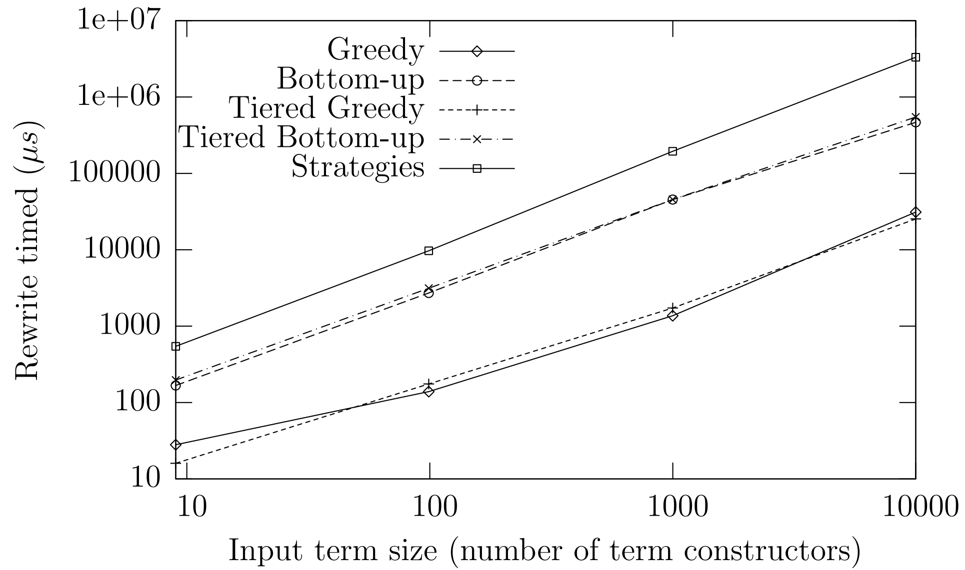


Figure 4.8: Rewrite times, in microseconds (μs), given matching input terms.

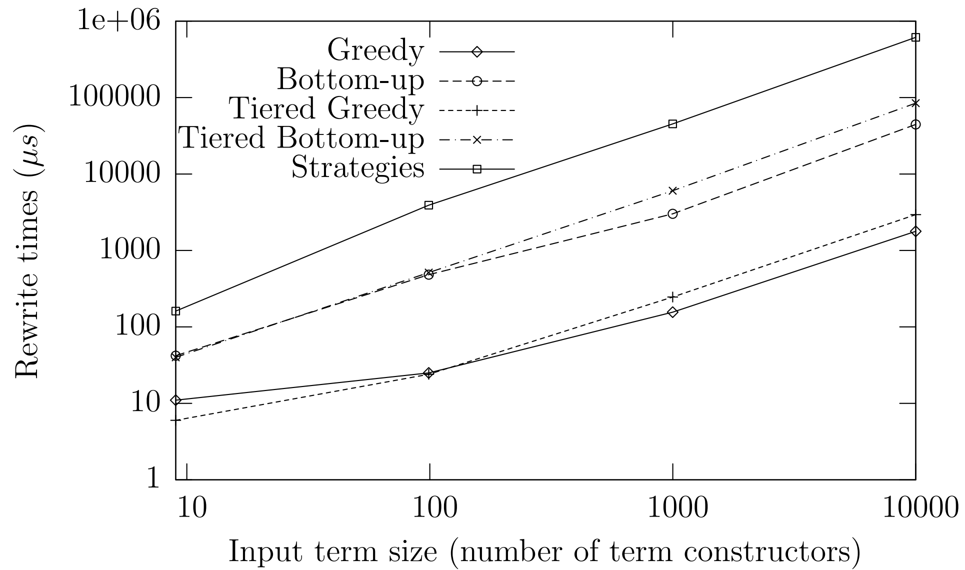


Figure 4.9: Rewrite times, in microseconds (μs), given non-matching input terms.

Table 4.1: Strategy run-times, broken into translation times.

Term size	Input translation	Rewrite	Output translation
9 (in-domain)	16	521	7
99 (in-domain)	83	9593	26
999 (in-domain)	17159	177147	336
9999 (in-domain)	79652	3231309	5641
9 (out-of-domain)	17	143	1
99 (out-of-domain)	87	3820	5
999 (out-of-domain)	3550	41660	7
9999 (out-of-domain)	29376	584293	7

traversal and match failure for each of the optimizers because none of the input terms were in the domain of the rewrite set. These times are roughly an order of magnitude less than the times in Figure 4.8, exposing a relatively high cost of term construction. Appendix A provides detailed data for both Figure 4.8 and Figure 4.9.

The data shown supports the linear complexity estimate of the underlying rewrite algorithms. When divided by the number of terms, the time per input term of each optimizer roughly remains in the same order of magnitude (sometimes decreasing as input term size increases). The overhead of using bottom-up rewrite techniques and/or tiering the rewrite set is also relatively stable across several orders of magnitude in the input size.

The test framework employs a strategy combinator library that attempts to remain true to Stratego. This requires terms to be translated to a term language native to the strategy combinator, passed to the rewriting strategy, and the result translated back to test framework terms. The secondary term language is based on the ATerm library [63], which is the native term language for Stratego. The remainder of this subsection looks at the performance of a strategy based greedy optimizer.

Table A.3 shows the run times, in microseconds, for a hand coded greedy optimizer and a greedy optimizer specified using rewrite strategies. The data shows that times for the strategy based operator do not stay within a constant bound across different orders of magnitude. The data for the out-of-domain input terms shows an even more marked increase in traversal times.

Table 4.2: Number of times underlying rewrite algorithm was applied, by optimizer.

Term size	Greedy	Bottom-up	Tiered Greedy	Tiered bottom-up
Balanced input terms:				
9	2	3	3	4
99 to 9999	2	4	4	5
Left-associative input terms:				
9 to 9999	2	3	3	4
Right-associative input terms:				
9	2	4	3	4
99 to 9999	2	4	4	5

Table 4.1 shows that the translation of terms into the strategy terms does not account for the majority of the additional overhead of the strategy rewriter. The strategy combinators use higher-order functions to compose strategies. Without aggressive inlining or other optimizations, these functions use the call stack much more often than the recursive matching function used in the greedy optimizer. Another possible detriment is the strategy optimizer's use of the non-deterministic choice combinator for composing rewrites. The non-deterministic choice combinator is not sophisticated enough to partition rewrites based on the top-level constructor in rewrite patterns. This also means that when an input term is outside the rewrite domain, all rewrites are tried before failing.

4.4.3 Rewrite Depth

This subsection looks at higher-level properties of the rewrite algorithms. First, it measures convergence rates of the various algorithms. The convergence rate is the number of times the algorithm must be applied before it reaches a fixed-point. Second, it looks at the size of the output terms. Finally, it gives the number of constructor terms output by each optimizer.

Table 4.2 shows the number of times the fixpoint loops call into the underlying rewrite routine. These measurements do not distinguish between tiers, showing the total number of times the tiered optimizer performed an optimization pass on either

Table 4.3: Size of the output terms for input terms purely in the rewrite domain, in number of constructor terms, by optimizer and input term associativity.

Term size	Greedy	Bottom-up	Tiered Greedy	Tiered bottom-up
Balanced input terms:				
9	15	15	15	15
99	147	153	162	159
999	1383	1437	1509	1491
9999	15783	16659	16806	16770
Left-associative input terms:				
9	15	15	15	15
99	171	150	171	150
999	1746	1500	1746	1500
9999	17496	15000	17496	15000
Right-associative input terms:				
9	12	12	15	15
99	102	147	171	150
999	1002	1497	1746	1500
9999	10002	14997	17496	15000

the input or an already rewritten term. For the bottom-up rewriter, the data does indicate that tiers can reduce the number of passes necessary to achieve a fixpoint output term. Since the test rewrite set contains two tiers, the number of passes to achieve fixpoint has a worst case of doubling the optimization passes. In the case of the bottom-up optimizer, using tiers only adds at most one additional pass. However, using tiers with the greedy optimizer does double the number of optimization passes for some inputs.

Table 4.3 and Table 4.4 show measurements of the output term size and the number of string constructors in the output terms, respectively. Together, this data shows that the greedy optimizer is comparatively optimal at performing shortcut deforestation, with the lowest number of output string constructors for all three association patterns for input terms. This implies the fewest intermediate results and the results include the globally optimal result where a single string constructor is generated for right-associative input terms.

The non-tiered greedy optimizer maintains its lower string constructor count even

Table 4.4: Counts of string constructors for input terms in the rewrite domain, in number of string constructor terms, by optimizer and input term associativity.

Term size	Greedy	Bottom-up	Tiered Greedy	Tiered bottom-up
Balanced input terms:				
9	2	2	2	2
99	18	20	23	22
999	244	262	286	280
9999	2048	2340	2389	2377
Left-associative input terms:				
9	2	2	2	2
99	25	32	25	32
999	250	332	250	332
9999	2500	3332	2500	3332
Right-associative input terms:				
9	1	1	2	2
99	1	16	24	17
999	1	166	249	167
9999	1	1666	2499	1667

for left-associative input terms. In the case of left-associative input terms, both bottom-up optimizers generate smaller output terms. The corresponding string constructor counts for these output terms implies that the bottom-up optimizers are still using more intermediate strings, and that the greedy optimizer is outputting longer lists that feed into list based string constructors.

4.5 Implementations

This section looks at several languages with DSO implementations, comparing how each explores the design space described in Section 4.3, and analyzed in Section 4.4.

4.5.1 Implementation in GHC

The Glasgow Haskell Compiler, GHC, uses compiler pragmas to specify domain-specific optimizations. Using this front-end, user optimizations are embedded in comments, and are not used by other Haskell implementations.

4.5.2 Implementation in Manticore

The domain-specific optimizer in Manticore uses two separate domain-specific languages for the specification of domain-specific terms and rewrites on those terms. Domain-specific terms are distinguished terms in the first level Manticore intermediate representation (BOM). These terms are called high-level operators (HLOps), and present a form of hygienic macros that are commonly not explicitly available in the Manticore surface language. The Manticore compiler dynamically loads HLOp definition files as HLOp's are encountered during compilation. The HLOp definition can include a reference to one or more rewrite definitions, which are also loaded by the Manticore compiler. Manticore uses these high-level rewrites on HLOps to permit extensible domain-specific optimization.

At time of writing, the back-end uses a mixture of two things: a beta-redex expander, and a bottom-up rewriter.

Future work will add tiered rewriting to the Manticore domain-specific optimizer. Manticore high-level rewrite tiers will not be as granular as those presented in Section 4.3.

4.5.3 Implementation in Mython

The design of the Mython language (which is in turn inherited from Python) presents a challenge for domain-specific optimizers. Out of the three implementations discussed here, Mython is the only language that has dynamic binding. This greatly complicates determining if a term corresponds to a domain-specific term. This problem applies to most dynamic languages, including Perl (see Section 6.3.9), for example. Perl 6 gets around this problem by using glyphs to specify a binding time for a term [74].

Mython's front-end extensibility allows users to embed domain-specific languages for rewrite specification in the language itself. Mython developers have the ability to associate concrete syntax with optimization specification languages. Using compile-time metaprogramming, these optimizations can then create optimization routines that are integrated with the Mython compiler's back-end.

Mython's back-end extensibility affords the ability to chose optimization algorithms based on their input domains. As shown in Section 4.4, the quality of an optimization can depend on the input terms. Users have the ability to customize the optimizer to their code, or even override the code generator to output several versions of their algorithm, and adaptively select an algorithm optimal to their application users' inputs.

Future work will embed the Stratego language, and allow users to perform an optional strategy application pass on the Python intermediate representation. The results for strategy based optimizers obtained in Section 4.4.2 indicates a Stratego implementation requires care to scale appropriately. One possibility for improving these results is to create a set of domain-specific optimizations for strategies.

4.6 Conclusions

This chapter has provided both the motivation and a method for adding domain-specific optimizations to language implementations. Section 4.1 discussed how approaches to abstraction can capture the form, but not necessarily the high-level semantics of a domain-specific language. Section 4.2 and Section 4.3 explored the design space of DSO specification and application, with Section 4.3.2 and Section 4.3.3 presenting novel methods for applying and organizing rewrites.

Section 4.4 presented a quantitative analysis of the rewriting methods from Section 4.3. Section 4.4.2 showed that all the rewriting methods scale linearly with the number of input terms. These time results mean that there are few practical barriers to adding programmable and sophisticated rewriting engines to a compiler. Section 4.4.3 attempted to show that more compilicated rewriting methods could benefit user programs. However, the data only shows there were no penalties. This analysis used an admittedly shallow term domain. Future work will look at more complicated domains, specifically domains that present greater opportunities for code motion and interactions at multiple levels of abstraction.

User-defined optimizations fit into the larger framework of extensibility by allowing users to modify a compiler's optimizer. These user-defined optimizations allow

high-level properties of user languages to be input and used in program transformation. The result is domain abstraction with greater program optimality than a host optimizer can provide alone.

CHAPTER 5
CASE STUDY: AN EMBEDDED DOMAIN-SPECIFIC LANGUAGE

5.1 Introduction

The scientific computing community seeks to develop more reliable, maintainable, and optimal scientific software. These three goals are at odds with one another. At the heart of this conflict is a kind of complexity similar to the sort that troubles domain-specific optimization (Section 4.1). For example, hand optimized code is often complex and sensitive to slight modifications, decreasing maintainability and reliability. Developers can increase code reliability through the use of both static and dynamic checks of correctness, but at the cost of increased learning curves for complex type systems, and increased run times as the code checks dynamic assertions. Both code development and maintenance may be simplified by using a domain-specific language (DSL), but again at the cost of optimality. As argued in Section 4.1, using a domain-specific language without the means for domain-specific optimization can result in sub-optimal code.

The FEniCS project [30] seeks to provide software for scientific simulation that addresses the issues of reliability, maintainability, and optimality. The FEniCS project deals with these problems through robust, automatic code generation and optimization. DSL's form a high-level interface to this automation. Figure 5.1 presents an example of the FEniCS embedded domain-specific language, MyFEM. This program generates a C++ function that is 64 lines of code, with multiple nested loops that are up to six deep.

The rest of this chapter describes the purpose and implementation of MyFEM. It serves as a case study of embedded domain-specific language implementation in an extensible host language. Section 5.2 reviews background material, including a brief

```
 1: from basil.lang.fenics import FEniCS
 2: from basil.lang.fenics.bvpir import *
 3: quote [myfront]: from basil.lang.fenics import FEniCS
 4:
 5: quote [FEniCS.bvpFrontEnd] Laplace:
 6:    TestFunction v
 7:    UnknownField u
 8:
 9:    <grad v, grad u>
10:
11: print "void Jac_Laplace (const Obj<ALE::Mesh>& m,"
12: print "    const Obj<ALE::Mesh::real_section_type>& s,"
13: print "    Mat A, void * ctx)"
14: print FEniCS.bvpIRToCplus(Laplace)
```

Figure 5.1: Example of a MyFEM code generation program.

review of the finite element method and the Mython programming language. Section 5.4 describes the MyFEM compiler design and how it is used to extend Mython. This paper concludes by identifying future and related work.

5.2 Background

This work is a fusion of two disciplines in computer science: scientific computing and programming languages. This section provides some background on scientific computing and embedded domain-specific language implementation. It begins with an overview of the motivations and goals of automating scientific computing. It then describes the finite element method, a computational tool for simulation. Finally, it provides an introduction to the Mython programming language, which uses compile-time meta-programming to embed domain-specific languages as concrete syntax.

5.2.1 Automating Scientific Computing

Simulation grants scientists and engineers the ability to do experiments that are not feasible in the real world. Some experiments are not feasible because of resource constraints, and include such domains as high energy physics. Other experiments may

be prohibitive because they pose real risks to the experimenters, their environment, or the public. For this very reason, one of the first and still common applications of the digital computer is the simulation of atomic physics. Even when resources are available and sufficient precaution can be taken, researchers can use simulation as a tool for guiding and validating both theory and experimentation.

In simulations people provide mathematical models of systems and a set of boundary conditions, which may include initial conditions. Frequently, these models employ partial-differential equations (PDE's) to relate the change in the system as a function of the system's instantaneous state [41]. An example of this might be an equation of the form:

$$u' = f(u)$$

In this model, the function f presents a means of calculating an instantaneous rate of change of a state vector, u', using only the current state vector u. Simulation involves the calculation some unknown, which can either be u' (which is trivial in this case), u, or f.

One of the most important aspects of automating scientific computing is the management of error. Solving problems using digital simulation necessitates discretization of the problem domain. Since many of these models are continuous, discretization introduces error into the calculations. These calculations are often then repeated by the simulation, which can cause the error to compound. This compounding nature of error can limit a simulation's usefulness.

Iterative methods allow management of a simulation's error, but they also affect the simulation's portability and maintainability. Iterative methods track error, iterating computations to decrease error when the error exceeds expectations. This iteration can add both space and time complexity to the underlying program. Therefore, automation of scientific computing includes a quantification of acceptable error as an input, in addition to a model and boundary conditions.

5.2.2 The Finite Element Method

The finite element method (FEM) is a means of discretization for the purpose of solving partial-differential equations. Typically the finite element method, particularly its application to boundary value problems, has three inputs: a mesh, boundary conditions, and partial differential equations (PDE's) that govern the relationship between field values. A mesh is typically a discretization of space into a triangulation, but this abstracts to arbitrary manifolds. The discretization also determines the relationship between field values and vertices in the mesh.

The MyFEM implementation allows engineers and scientists to specify boundary and conditions and PDE's, generating C++ code that interfaces with the Sieve [40] and PETSc [5] libraries. The Sieve library handles user mesh inputs. Sieve lets users represent arbitrary meshes, both in dimension and shape. This representation allows MyFEM to emit integration loops independent of the mesh, the finite element, and the solver. The PETSc library provides a variety of solvers which are used to solve the algebraic systems that result from discretization.

The remainder of this section focuses on MyFEM's role in the finite element method: the specification of partial-differential equations and boundary conditions. For example, an engineer may want to find a field u that satisfies the following condition:

$$-\nabla \cdot (k\nabla u) = f(x, y) \quad \text{in} \quad \Omega \tag{5.1}$$

This equation states that the divergence (represented as "$\nabla\cdot$") of the gradient of the field u is equal to a known function, f, in the domain Ω. In order to find the unknown field u, the engineer must also provide an essential boundary condition, such as the following:

$$u = 0 \quad \text{on} \quad \partial\Omega \tag{5.2}$$

To use MyFEM, the engineer must convert these equations to a weak-form expression. Deriving the weak form first involves introducing a test function, v, to both

sides of (5.1). The user integrates over the region Ω, and applies Green's identity. Green's identity splits the integral into an integral over Ω added to an integral over the boundary region $\partial\Omega$. The boundary integral is zero by (5.2), providing an essential boundary condition. Substitution results in the following:

$$\int_\Omega \nabla v \cdot k \nabla u - \int_\Omega vf = 0 \tag{5.3}$$

Details of this process and the reasoning behind its nomenclature are given in Chapter 2 of Gockenbach [27].

Once the user has a weak-form equation for his or her problem, the problem can be expressed in MyFEM as:

```
TestFunction v
UnknownField u
CoordinateFunction f

<grad v, grad u> - <v, f>
```

This example elides the constant k, and illustrates how MyFEM uses inner-product notation (`<,>`) to denote integration over the product of two subexpressions (details appear in Section 5.3.3). The result is a concrete syntax that is closer to actual mathematical notation, while still being machine readable. The MyFEM compiler translates the source program into target code that can interface with various equation solvers.

5.3 The MyFEM Domain-specific Language

This section describes the high level design of the MyFEM domain-specific language. It begins with the surface (concrete) and abstract syntax of the language. Section 5.3.2 discusses types in the MyFEM language. Finally, Section 5.3.3 explains the language's semantics.

5.3.1 MyFEM Syntax

MyFEM programs have two components: declarations and constraints. Declarations bind MyFEM identifiers to various values present in a runtime environment, which is external to MyFEM. The code that MyFEM generates typically abstracts over the bound variables, making them inputs to the output program. Section 5.3.1 discusses the various kinds of MyFEM variables. Constraints are equality expressions, which MyFEM combines to form a system of equations. Section 5.3.1 describes the concrete and abstract syntax of these constraints.

Appendix B.1 provides the full concrete syntax of MyFEM as a extended Backus-Naur form (EBNF) grammar.

Declarations

MyFEM has four kinds of declarations: unknown fields, test functions, coordinate functions, and fields. Unknown fields represent an unknown vector field and are represented using functions. Test functions correspond to the function, v, introduced as part the weak form expression of boundary value problems (see Section 5.2.2).

Coordinate functions allow variables to be associated with the input problem's coordinate system. Once bound, these variables allow a MyFEM user to specify scalar valued fields based on coordinates. These variables are bound in a first-come, first-bound nature. For example, given the following:

```
CoordinateFunction y, x
```

MyFEM binds the variable `y` to the first coordinate component, and `x` to the second component of a coordinate. This example runs counter to typical 2 dimensional coordinate systems where given a vector representation of a coordinate, v, the coordinate's projection onto the x-axis is $v[0]$.

Fields are arbitrary fields, and are reserved for future use in an extended MyFEM implementation.

Highest	Atoms, Dual pairing: `<,>`, Grouping: `(...)`
	Power operator: `^`
	Unary operators: (unary)`+`, (unary)`-`, `grad`, `div`, `trans`
	Multiplication: `*`, `/`
	Adding/subtracting: `+`, `-`
	Vector operations: `cross`, `dot`
Lowest	Equality: `=`

Table 5.1: MyFEM operator precedence.

Constraints

MyFEM inherits both its lexical syntax and parts of its concrete grammar from Python (see also Section 5.4.2). Constraints are newline delimited expressions, with parenthesis or brackets subexpressions allowed to span multiple lines. MyFEM and its back-end infrastructure assume that constraints that do not explicitly give an equality expression are equal to zero.

The expression syntax provides common Algol-style arithmetic operators, but adds several vector operators, and a bilinear integration operator, $<,>$. MyFEM reuses Python's stratified concrete expression syntax to enforce operator precedence. Table 5.1 lists the MyFEM operators from highest precedence to lowest.

The concrete syntax collapses to a simple abstract syntax, removing single child nodes that are not unary operators from parse trees. Table 5.2 shows the expression abstract syntax terms, the concrete representation of the operator, and the intended mathematical operation (with more detail given in Section 5.3.3). MyFEM removes some concrete operators from the abstract syntax, specifically subtraction, which is replaced by addition and negation.

MyFEM adds subscript syntax that allows parts of subexpressions to only be computed over parts of a region, or a region boundary. Constraining integration

Add(e_0, e_1)	`+`	Addition.
Cross(e_0, e_1)	`cross`	Cross product.
Div(e_0, e_1)	`/`	Division.
Diverge(e_0)	`div`	Divergence.
Dot(e_0, e_1)	`dot`	Dot product.
Eq(e_0, . . . , e_n)	`=`	Equality.
Grad(e_0)	`grad`	Gradient of a field.
Int(e_0, e_1)	`<,>`	Dual pairing (integration).
Mul(e_0, e_1)	`*`	Multiplication.
Neg(e_0)	`-`	Unary negation.
Pow(e_0, e_1)	`^`	Exponentiation.
Trans(e_0, e_1)	`trans`	Transpose of a matrix.

Table 5.2: Abstract syntax for MyFEM operators.

uses an underscore followed by the region's name. Users will be able to constrain a subexpression by grouping it with parenthesis, followed by a pipe and the region's name. At time of writing, this feature is not used, but present in the concrete syntax.

5.3.2 Types in MyFEM

The MyFEM type system is simple relative to the type systems of general-purpose languages. MyFEM has four base types and no type constructors. These base types include: scalar values, vectors, functions and functionals. Scalars are floating-point numbers ($\mathbb{R}$). Vectors are fixed-size, ordered collections of floating-point numbers ($\mathbb{R}^n$). The dimensionality of vectors, n, is a property of the input mesh, and is an arbitrary fixed constant as far as MyFEM is concerned. Functions are maps from vectors to scalars. Functionals are composable maps from functions to functions.

The type environment is built from the global declarations seen prior to a given constraint. Figure 5.2 illustrates the structure of the MyFEM type environment.

Figure 5.3 gives some type of MyFEM's type checking rules. The rules are stated as inference rules, with the premise appearing on top (with multiple premises joined implicitly by logical-and), and the conclusion appearing on bottom. The type judgement "$\Gamma \vdash e : \tau$" means that the expression e has type τ, given the type environment

$$
\begin{array}{rcl}
\Gamma & := & \mathit{decl}\ \Gamma \\
 & | & \cdot \\
 & & \\
\mathit{decl} & := & (x : \mathit{declspec}) \\
 & & \\
\mathit{declspec} & := & \texttt{TestFunction} \\
 & | & \texttt{UnknownField} \\
 & | & \texttt{CoordinateFunction} \\
 & | & \texttt{Field}
\end{array}
$$

Figure 5.2: Structure of the MyFEM type environment.

Γ. The rules sometimes use a comma as a notational shortcut. Where commas separate expressions, the type judgement means that the multiple expressions share the same type under the type environment.

The typing rules in Figure 5.3 are not complete, meaning they are not sufficient to infer the types of all well-typed MyFEM programs. These rules allow the following MyFEM program to type check:

```
TestFunction v
UnknownField u
CoordinateFunction x, y

<grad v, grad u> - <v, x + y>
```

Processing the declarations yields the following type environment:

$$
\begin{array}{rl}
\Gamma = & (y : \texttt{CoordinateFunction}) \\
 & (x : \texttt{CoordinateFunction}) \\
 & (u : \texttt{UnknownField}) \\
 & (v : \texttt{TestFunction}) \cdot
\end{array}
$$

Under this environment, a MyFEM type checker can verify that the constraint is in the functional type. Figure 5.4 gives the full proof of this type derivation.

$$\mathrm{T-Function}\quad \frac{(x,y)\in\Gamma \quad y\in\{\mathtt{TestFunction},\mathtt{UnknownField},\mathtt{Field}\}}{\Gamma\vdash \mathrm{Var}(x):\mathtt{function}}$$

$$\mathrm{T-Scalar}\quad \frac{(x,\mathtt{CoordinateFunction})\in\Gamma}{\Gamma\vdash \mathrm{Var}(x):\mathtt{scalar}}$$

$$\mathrm{T-Grad}\quad \frac{\Gamma\vdash e:\mathtt{function}}{\Gamma\vdash \mathrm{Grad}(e):\mathtt{vector}}$$

$$\mathrm{T-Int-1}\quad \frac{\Gamma\vdash e_0,e_1:\mathtt{function}}{\Gamma\vdash \mathrm{Int}(e_0,e_1):\mathtt{functional}}$$

$$\mathrm{T-Int-2}\quad \frac{\Gamma\vdash e_0,e_1:\mathtt{vector}}{\Gamma\vdash \mathrm{Int}(e_0,e_1):\mathtt{functional}}$$

$$\mathrm{T-Int-3r}\quad \frac{\Gamma\vdash e_0:\mathtt{function}\quad \Gamma\vdash e_1:\mathtt{scalar}}{\Gamma\vdash \mathrm{Int}(e_0,e_1):\mathtt{function}}$$

$$\mathrm{T-Comp-1}\quad \frac{\mathit{op}\in\{\mathrm{Add},\mathrm{Mul},\mathrm{Div}\}\quad \Gamma\vdash e_0,e_1:\mathtt{functional}}{\Gamma\vdash \mathit{op}(e_0,e_1):\mathtt{functional}}$$

$$\mathrm{T-Comp-2r}\quad \frac{\begin{array}{c}\mathit{op}\in\{\mathrm{Add},\mathrm{Mul},\mathrm{Div}\}\\ \Gamma\vdash e_0:\mathtt{functional}\\ \Gamma\vdash e_1:\mathtt{function}\end{array}}{\Gamma\vdash \mathit{op}(e_0,e_1):\mathtt{functional}}$$

$$\mathrm{T-Neg}\quad \frac{\Gamma\vdash e:\mathtt{function}}{\Gamma\vdash \mathrm{Neg}(e):\mathtt{function}}$$

$$\mathrm{T-Arith}\quad \frac{\mathit{op}\in\{\mathrm{Add},\mathrm{Mul},\mathrm{Div}\}\quad \Gamma\vdash e_0,e_1:\mathtt{scalar}}{\Gamma\vdash \mathit{op}(e_0,e_1):\mathtt{scalar}}$$

Figure 5.3: A partial set of type rules for MyFEM.

$$(1)\left[\dfrac{\dfrac{\dfrac{(v:\texttt{TestFunction})\in\Gamma}{\Gamma\vdash \mathrm{Var(v)}:\texttt{function}}}{\Gamma\vdash \mathrm{Grad(Var(v))}:\texttt{vector}} \qquad \dfrac{\dfrac{(u:\texttt{UnknownField})\in\Gamma}{\Gamma\vdash \mathrm{Var(u)}:\texttt{function}}}{\Gamma\vdash \mathrm{Grad(Var(u))}:\texttt{vector}}}{\Gamma\vdash \mathrm{Int(Grad(Var(v)),Grad(Var(u)))}:\texttt{functional}}\right.$$

$$(2)\left[\dfrac{\dfrac{(x:\texttt{CoordinateFunction})\in\Gamma}{\Gamma\vdash \mathrm{Var(x)}:\texttt{scalar}} \qquad \dfrac{(y:\texttt{CoordinateFunction})\in\Gamma}{\Gamma\vdash \mathrm{Var(y)}:\texttt{scalar}}}{\Gamma\vdash \mathrm{Add(Var(x),Var(y))}:\texttt{scalar}}\right.$$

$$(3)\left[\dfrac{\dfrac{\dfrac{(v:\texttt{TestFunction})\in\Gamma}{\Gamma\vdash \mathrm{Var(v)}:\texttt{function}} \quad (2)}{\Gamma\vdash \mathrm{Int(Var(v),Add(Var(x),Var(y)))}:\texttt{function}}}{\Gamma\vdash \mathrm{Neg(Int(Var(v),Add(Var(x),Var(y))))}:\texttt{function}}\right.$$

$$\dfrac{(1)\quad(3)}{\begin{array}{ll}\Gamma\vdash \mathrm{Add(} & \mathrm{Int(Grad(Var(v)),Grad(Var(u))),} \\ & \mathrm{Neg(Int(Var(v),Add(Var(x),Var(y))))}\quad\mathrm{)}:\texttt{functional}\end{array}}$$

Figure 5.4: Example type derivation for a MyFEM program.

The type of a constraint is ideally a higher-order function that takes a function and returns a boolean value. MyFEM programs take a function intended to approximate the unknown field, and return the residual for that function over the mesh. MyFEM transforms constraints into functionals. The output functional transforms an approximating function into a function that returns scalar values that are ideally zero. The magnitude of the output of the transformed function provides a measure of error for the approximating function.

The type system not only ensures the correctness of inputs, but also identifies inputs that would not generate proper target code. For example, the constraint `<grad v, u>` would fail to type check since there is no means to integrate a vector and a function. In this example the `grad v` subexpression would type check as a `vector` and `u` would type check as a `function`. While syntactically correct, MyFEM will reject integration of these terms since there is no corresponding type rule for such an operation.

5.3.3 MyFEM Semantics

This subsection describes the semantics of MyFEM and how MyFEM constraints map onto programs for measuring error. It begins with a discussion of the representation of functions and functionals. It then discusses how the MyFEM operators map into the underlying representation. Finally, it provides the operational semantics of several of the key operators in MyFEM, specifically the gradient and integration operators.

Vector Spaces of Functions

The surface language of MyFEM deals with vector spaces of real numbers. In its translation to a target language, MyFEM uses vector spaces to also represent the function and functional values. In this representation, functions are vectors that represent linear combinations of the underlying basis functions. In turn, functionals are matrices that transform one function to another.

For example, a simple vector space for quadratic functions includes the functions $\{x^2, x, 1\}$. This vector space maps the function "$2x^2 + 3x - 10$" to the vector $< 2, 3, -10 >$. In general, the vector $< a, b, c >$ represents the quadratic function $ax^2 + bx + c$. In this vector space, differentiation in the x coordinate is a functional represented by the following matrix:

$$\frac{\partial}{\partial x} = \begin{pmatrix} 0 & 2 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

The function vector space is an input of the MyFEM program, and is composed with the input mesh data structure. Each function that corresponds to a dimension in the input vector space is called a basis function. Paired with the provided basis function space is a function space for the derivative of the basis functions. Together, the basis function space and any needed derivatives form a jet space. The FIAT [36] system provides the jet space. MyFEM uses the FIAT inputs for computing the gradient of a function.

```
1: for cell in cells:
2:   for quad in quads:
3:     for f in local basis functions:
4:       for g in local basis functions:
5:         M[ind(f),ind(g)] = ([| e0[f,g] |] *
6:                             quadWeights[quad] * detJ)
7:       v[ind(f)] = [| e1[f] |]
```

Figure 5.5: MyFEM target pseudocode.

The remainder of this subsection uses the variable n to represent the number of basis functions in the discretization. This should not be confused with the dimensionality of the input FEM manifold. The number of dimensions in the FEM manifold is represented as dim.

High Level Semantics in the Target Language

As mentioned at the end of Section 5.3.2, the result of parsing and compiling a MyFEM constraint is a functional. MyFEM constraints calculate a functional over the basis space, which is represented by a $n \times n$ matrix. This can also include a constant term, which is constant with respect to the unknown field. This constant term is represented here as v, where $|v| = n$. Calculation of these values is done inside several loops.

MyFEM handles constraints of the form $<$ `v`,$e_0 + e_1$$>$, where e_0 is linear in the unknown field and e_1 is constant in the unknown field. These distinctions are approximated by e_0 being typed as a function and e_1 being typed as a scalar. The resulting functional and function are computed within a set of nested loops in the target language. The target code evaluates the pseudocode shown in Figure 5.5. In Figure 5.5, `[| e0[f,g] |]` represents the calculation of e_0 at the local basis functions f and g and `[| e1[f] |]` represents the calculation of e_1 at the local basis function f.

Much like estimating the integration of a function is done by creating boxes that approximate the area under the curve, the finite element method estimates the integration of an arbitrary manifold by creating discrete cells that cut up the manifold.

The outermost loops traverse through the input discretization, building a global functional. Many of the global basis functions are defined as being non-zero only within a small portion of the global region. As an optimization, the inner loops iterate over local basis functions. The local basis functions are both non-zero in the current subregion, and together form a small subset of the global basis functions. The sample code uses the `ind` function to map from local basis functions to the global basis functions.

At the time of writing, the meaning of constraints that do not fit the form given above is undefined.

Operator Semantics

To complete an overview of MyFEM semantics, the following considers the operational semantics of two MyFEM operators: the gradient operator and the integration operator.

The gradient operator takes a function and creates a vector field representing the slope of the function. Mathematically, the gradient of a function f is defined as the following:

$$\nabla f =< \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_{dim}} >$$

The discretization provides a vector space for the derivative of the local basis functions. MyFEM uses this to calculate the gradient vector field for the current location in the manifold. The result is stored in an intermediate vector, v^{inter}, whose elements are calculated as follows:

$$v_i^{inter} = \sum_j^{dim} invJ[i,j] * f'_{in}[j]$$

The integration operator performs a full tensor contraction of its arguments. The computational semantics of the contractions map to different operations based on the types of its operands. For example, when both operands of an integration expression are vectors, the integration operator integrates over the dot product of the vectors.

$$< v_0, v_1 >= \int_\Omega v_0 \cdot v_1$$

This translates to the following target language computation:

$$[| < v_0, v_1 > [f, g]|] = \sum_i^{dim} v_{0,i}^{inter} * v_{1,i}^{inter}$$

When the integration operands are functions, the integration operator is intended to calculate the integral of the product of the two functions:

$$< f_0, f_1 >= \int_\Omega f_0 f_1 \tag{5.4}$$

Under the assumption that these functions are linear and unknown, MyFEM's target language calculates the integration functional by multiplying the values of the local basis functions f and g. Computation of this product translates to the target code as follows:

$$[| < f_0, f_1 > [f, g]|] = f * g$$

When one of the integral operands is a scalar expression, the semantics of the integral do not change from equation 5.4. At the current position in the manifold, the scalar expression is constant, and can be removed from the sum. Therefore the integral can be computed outside of the second local basis function loop:

$$[| < f_0, f_1 > [f]|] = f * [|f_1()|]$$

5.4 Implementing MyFEM

This section describes the implementation of MyFEM and how this implementation interoperates with Mython. The architecture of MyFEM is typical for a compiler, where a front-end handles translation of the concrete syntax into some intermediate representation and a back-end manages optimization and code generation for the intermediate representation. Section 5.4.1 looks at the general methods used to embed

a domain-specific language in Mython. Section 5.4.2 covers the front-end implementation details of MyFEM. Finally, Section 5.4.3 discusses the code optimization and generation portions of the language implementation.

5.4.1 Methodology

The most straightforward means of embedding a language in Mython is to build a parser using a parser generator. One problem with this approach is the verbosity of the resulting concrete parse tree. Therefore, it is often convenient to transform the concrete parse tree into an intermediate representation that is more closely representative of the underlying semantics. Abstract syntax trees are one such intermediate representation, but developers can find creating abstract syntax tree constructors and utilities tedious due to a duplication of a lot of boilerplate code. The following subsections identify how Mython and its libraries help developers address parser generation, tree traversal, and generating abstract syntax trees.

Parsing

Part of Mython's reflective nature is how it exposes its parser generator to the surface language. The `pgen` parser generator is an LL(1) parser generator that takes extended Backus-Naur form (EBNF) grammars and generates parsing automata. Originally developed in C by Guido van Rossum for the implementation of Python, Mython offers a reimplementation of `pgen` in Python. Mython uses this version of `pgen` to generate parsers for Python, Mython and MyFEM. For example, the EBNF grammar found in appendix B.1 is a valid `pgen` input.

This approach has two drawbacks. First, `pgen` has no integration with its lexical syntax. Parsers generated by `pgen` are constrained to using either Python's lexer or some adaptive lexer that outputs Python-compatible lexical tokens. This isn't too great a limitation, since keywords are not given specific lexical classes in Python, but passed as identifiers to the parser. MyFEM uses keywords for several of its new operators, sidestepping this issue.

The second drawback involves the robustness of `pgen`. Its heritage as a Python-specific tool mean it has not been hardened for general-purpose parser generation. It does not provide good error detection for ambiguity, and grammars that are not in the family of LL(1) languages. Finally, `pgen` does not include error recovery in the generated parsers. Both `pgen` and the parsers it generates will halt upon detection of the first syntax error.

The output of `pgen` generated parsers is a concrete syntax tree. These trees have the following structure:

$$\begin{aligned} node &:= (payload, children) \\ payload &:= ID \\ &\mid \; token \\ token &:= (ID, string, lineno, \ldots) \\ children &:= [node_1, \ldots, node_n] \end{aligned}$$

The Visitor Pattern in Mython

Mython's compiler uses an implementation of the visitor pattern [26] to walk and transform tree data structures. The approach described by this section closely follows other implementations in Python, including one found in Python `compiler` package [66]. Mython reflects this implementation up to the surface language as an abstract base class, `Handler`. Mython uses subclasses of the `Handler` class to both transform concrete syntax trees to abstract syntax tress, as well as generate code from abstract syntax trees.

The `Handler` class implements a generic visitation method, `handle_node()`. The `handle_node()` method first looks at the current tree node being visited, and uses the `get_handler_name()` method to determine a method name that specifically handles the given node type (by default, this is "`handle_`*NT*`()`", where *NT* is the name of the nonterminal symbol for the node). The `handle_node()` method then uses introspection to see if the handler method exists in the current `Handler` instance. If the specific method exists, `handle_node()` calls it, otherwise it calls a generic handler method, `handle_default()`.

Mython users can follow the same pattern as Mython itself, specializing the `Handler` class to walk both concrete and abstract syntax trees for domain-specific languages. This requires the user to specialize several methods. These include the `get_nonterminal()`, which `get_handler_name()` uses to generate the dispatch name, `get_children()`, which the visitor uses to determine a node's children, and finally `handle_default()`, which is the default handler method.

Abstract Syntax

Mython reflects the abstract syntax definition language (ASDL) to its surface language [76]. ASDL is a domain-specific language for describing abstract syntax data structures. Mython inherits its use of ASDL from Python, where both languages use ASDL to define their abstract syntax. While not part of the standard library, Python implements an ASDL parser and a utility for translating ASDL. Mython builds on the Python utility by providing a tool that translates from ASDL to a Python module. The module defines a class hierarchy for representing abstract syntax. Users can in turn introspect the output Python modules to create pretty-printers, visitors and other utilities.

Mython has a second visitor base class, `GenericASTHandler`, that is designed specifically for ADSL syntax trees. The `GenericASTHandler` class simplifies the number of methods needed to implement the visitor pattern, directly using a node's type name to determine its handler method, and providing a default handler that performs a depth-first traversal of the tree. The Mython compiler specializes this class into several subclasses, including one for lexical scoping calculations, and one for code generation.

5.4.2 The Front-end

The MyFEM front-end applies the methods described in Section 5.4.1 to create a front-end function that Mython users can use to embed MyFEM programs. The example code in Figure 5.1 shows such an embedding, where line 3 imports the MyFEM lan-

guage definition module into the compiler. Line 5 then uses the `FEniCS.bvpFrontEnd` quotation function to parse the MyFEM code on lines 6–9.

The `bvpFrontEnd()` function is a composition of several functions:

```
def bvpFrontEnd (name, text, env):
    global bvpParser, bvpToIR, bvp_escaper
    cst, env_1 = bvpParser(text, env)
    ir = bvpCSTToIR(cst)
    esc_ir = bvp_escaper(ir)
    stmt_lst = [ast.Assign([ast.Name(name,
                   ast.Store())], esc_ir)]
    return stmt_lst, env_1
```

The first function, `bvpParser()`, is a parser generated using the `pgen` tool. This outputs a parse tree. The front-end then send the parse tree to a transformer function, `bvpCSTToIR()`. This function translates the parse tree into several intermediate representations, finally resulting in an abstract syntax tree for an imperative intermediate representation. The imperative abstract syntax tree is translated into Python abstract syntax by the `bvp_escaper()` function, completing the quotation.

5.4.3 The Back-end

A back-end is responsible for performing any user specified optimizations and generating target code from the intermediate representation. Users can implement domain-specific optimizations as transformations of abstract syntax using the techniques described in Section 5.4.1. A user defines a custom back-end by composing his or her optimizers with a code generator. This section describes a simple back-end that generates C++ target code from the intermediate representation output by the front-end documented in the previous section.

The example in Figure 5.1 uses the front-end to generate the abstract syntax tree for an imperative intermediate language. As part of the Mython quotation mechanism, the Mython program reconstructs this tree at run-time and binds it to the

`Laplace` identifier. The script then passes the reconstructed tree to the back-end function, `bvpIRToCplus()` (in the `FEniCS` module), on line 14. The back-end function constructs a visitor for the intermediate representation, and uses the visitor object to generate a string containing the target C++ code.

MyFEM uses ASDL to define the imperative intermediate language. Appendix B.2 gives the ASDL specification of this language. MyFEM follows the ASDL AST visitation methodology, specializing the `GenericASTHandler` class. The resulting subclass, `IRToCplusHandler`, contains custom handler methods responsible for translating each of the abstract syntax nodes into C++ strings.

5.5 Results and Conclusions

The following section concludes this chapter by looking at related front-ends in the FEniCS effort and reviewing future work.

5.5.1 A Comparison to Other FEniCS Front-ends

There are several existing packages that serve the same role as MyFEM. The FEniCS form compiler, or FFC, is a Python based compiler that translates from Python code to C++ [37]. SyFi is a C++ library for building finite elements [3].

FFC expresses partial-differential equations using operator overloading.

SyFi uses the GiNaC symbolic computation library to express partial-differential equations [3, 71]. SyFi still requires users to assemble matrices by hand.

5.5.2 Future Work

MyFEM and Mython would benefit from several improvements, such as moving to a modular language definition, using a higher-level intermediate representation, more code generators, and more domain-specific optimizations. MyFEM will include domain-specific optimizations for finite elements, following related work in the FErari system [38].

5.5.3 Conclusion

This chapter presented a case study of using the Mython extensible language to embed another domain-specific language, MyFEM. The MyFEM language enhances the maintainability, reliability, and optimality of scientific software. MyFEM achieves maintainability by offering domain-specific notation. MyFEM's high-level notation reduces both code size and complexity. Figure 5.1 illustrated a 14 line Mython program, which generates a 65 line C++ function. The embedded MyFEM code, consisting of four lines, shows how MyFEM can reduce code size by an order of magnitude. Section 5.3.2 described how MyFEM's type system ensures the reliability of its target code. Even in the absence of domain-specific optimization, MyFEM's target code uses scientific libraries that provide speed, scalability, and portability across problem domains.

MyFEM differs from previous work in the scientific computing community by providing both concrete syntax and the opportunity for user-level domain-specific optimization. This work also provides a case study of using a recent class of extensible languages. These languages allow developers to embedded languages with concrete syntax and permit integration of some or all of their compiler with the host language compiler.

CHAPTER 6
RELATED WORK

6.1 Related Interfaces

There are many existing approaches to solving the problems posed in Section 1.2. These solutions include building custom compilers and translators, overloading or adding operators, syntax macros, and even the ever helpful compiler pragma. Each of these are described in more detail in the following subsections.

6.1.1 Compiler Tools

There has been so little innovation in the mainline parser and compiler generator tool chain that description of most of its usage and data flow could be copied verbatim from the literature of the mid-1980's [1], if not earlier. Yacc and yacc-like tools usually accept a single input file consisting of a custom version of the extended Backus-Naur form (EBNF) for specifying context-free grammars and output a parser in some target language (C in the case of bison/yacc, and Java in the case of ANTLR or JavaCC). Parser generators are typically coupled with a lexer generator, such as lex. The lexer generator accepts a set of regular expressions, and outputs a routine for segmenting and classifying strings of text into syntactically atomic substrings. This kind of framework has become canonical enough that even new parser generators and parsing algorithms are typically provided with a similar or even backwards compatible interface. These kinds of frameworks are brittle for several reasons, including the following: syntax and semantics are often coupled in these parsers, the syntax specifications are monolithic, and parsers are typically locked into one class of parsing algorithms (which were designed when time and space considerations were much more important than they are today).

While coupling syntax and semantics is important in some language designs, replacing actions in a syntax specification is not usually straightforward. Other approaches for obtaining the parse tree are still available, including instrumenting the parser's state machine (peering a tree node with the current top of stack on a shift, taking the current set of peers and making them children of a new node on a reduce, for instance). However, this solution causes extra work to be done, and that work is thrown away by consumers of the parse tree. Second, grammar specifications are monolithic, and composing two grammars requires ad hoc hand insertion of one specification into another. There are also lexical issues that can complicate composition. While it should be possible to handle lexical integration using lexical states, this assumes the target lexer generator supports different lexical states. Finally, even when one has two grammars for two different languages, and one wants to compose them, they are often written for different, incompatible parsing algorithms (such as LALR(1) or LL(n)). Furthermore, even simple extension of a syntax can introduce ambiguities or parser state conflicts, not to mention interfere with actions still necessary for successful parsing.

These criticisms readily map to the use cases presented in Section 1.2, some of which have already been mentioned. Language embedding hits most of these hurdles, the largest theoretical issue being seen as the grammar composition problem [39]. Assuming the two language specifications are for the same tool chain, it would be difficult enough to unambiguously merge two grammars and lexical conventions, much less merge actions that generate anything more complicated than an abstract syntax tree. Language extension is in a similar situation, but depending on the scope of the extension, the actions may not be as brittle as they are in full blown language embedding. Semantic coupling was already indicted in Section 1.2.4 as being a key obstacle in tool creation. Finally, domain-specific optimizations from a compiler-compiler perspective would most likely be initially formalized as a translating preprocessor for an extended syntax, facing a combination of the issues faced by the language extension and translation tool. Closer integration of domain-specific optimization features with the compiler optimizer open a larger class of issues (such as back-end integration) that are orthogonal to the compiler-compiler tool chain, unless one considers an optimizer

generator such as twig as being included in the tool chain [25].

6.1.2 Operator Overloading

Operator overloading is a popular method for allowing domain-specific libraries to be employed using notation more familiar to experts of that domain. For example, if one was writing a matrix library, allowing "A*B" to replace "A.multiply(B)" is more compact, and considered to be more readable by some. While languages like C++ and Python only allow existing operators to be overloaded, SML and Haskell allow custom operators to be defined and used as infix notation.

While operator overloading and extensions can improve the readability of domain-specific code by domain experts, they often can not be stretched far enough to cover all the notation one might want to implement. For example, neither Haskell nor SML allow the definition of postfix operators. These languages only support prefix operators by virtue of allowing the operator to be defined as a unary function, binding at the fixed precedence level of function application. Definition of other forms, such as the absolute value notation of $|exp|$ (or other syntactic constructs such as quote), are also not supported. Operator overloading has been used in languages such as Python to support a syntax object language, but common EBNF constructs such as the Kleene closure are not supported in the operator language. The Kleene closure is an unary postfix operator, typically represented by an asterisk, which can not be supported because the asterisk operator is immutably defined as being an infix binary operator.

The considerations above greatly limit the ability of operator dynamism to address most of the scenarios in Section 1.2. With aggressive extension of the operator set, some languages may embed a subset of another language, but embedding a language with a radically different syntax is precluded. Examples would include embedding SQL in C++, XML in Haskell (not that this hasn't been done using other methods, or approximated using combinators [75]), or even EBNF in Python. Operator dynamism does allow some syntactic sugar to be associated with language evolution proposals, but typically weakens any argument for change. Typical counter arguments are of

the following form: if the feature can be supported using operator overloading or extension, why not just make it a library? Operator overloading and extension in the best case is simply orthogonal to the concerns of tool writers. In the worst case, operator dynamism can require more sophisticated type analysis in order to determine the binding of a polymorphic operator. Operator dynamism is also typically orthogonal to domain-specific optimizations. In the best cases, aggressive inlining and partial evaluation can implicitly cover some domain optimizations. In other cases, domain-specific optimizations can be explicit with the library implementor providing fast execution paths for cases that could have been statically optimized.

6.1.3 Syntax Macros

Syntax macros are user defined tree rewrites that are typically constrained to translating from abstract syntax to abstract syntax. As mentioned in Section 1.2.1, this suits the Lisp and Scheme languages, which have concrete syntax that readily implies the underlying abstract syntax. In other languages the abstract syntax remains a tree, but code does not uniformly delimit its syntax hierarchy. For example, Java and other C like languages use braces and semicolons at the declarator, definition and statement level, but then switch to parentheses, implicit associativity and precedence rules at the expression level. Only recently have concrete syntax macros been developed for C, as embodied in Robert Grimm's xtc [28]. Aspect oriented programming provides a slightly different approach to navigating and transforming abstract syntax, or a subset thereof, but does not allow custom syntax extension.

As they are implemented in Lisp and Scheme, syntax macros provide the kind of interactive flexibility not present in static language development systems. Macros allow language implementors, library writers and users to make changes to the language's syntax at all stages of development (as identified in the abstract, those stages are: compiler-compile time, compile time, load time, and run time). Changes to the syntax are still constrained since additional syntax must use parenthesis to delimit hierarchy, or revert to using a custom parser. Macros also have strange interactions

with the static and dynamic properties of some module system implementations, and cross-module macro interactions require explicit management [23].

The concrete syntax macros in xtc are specialized to only permit language embeddings in C. Thus, xtc precludes the arbitrary mixtures of language that are allowed by Stratego/XT while still only allowing language embeddings at compile time. Lisp style syntax macros specialize for parenthesis languages, but do not constrain embedding to compile time. Lisp macros do require the embedded language to be a parenthesis language. Otherwise, a parser must be used and syntax checking embedded code can only be done at load time or run time. The macro systems identified here go a long way towards assisting with language extension. Typical Lisp-like language communities do not propose language extensions, they simply submit macros that implement their language extension and let the community decide if the change is worthwhile or not. Writing tools for xtc extensions is not straightforward unless the tool writer is either a downstream consumer of the C code it generates, or the tool embeds the xtc system itself. Lisp macros either work orthogonally to tool development or complicate it. Lisp macros let macro writers switch out semantics from underneath the tool, possibly invalidating hard coded assumptions (one specific instance is type checkers in PLT Scheme [49]). Finally, both kinds of syntax macros permit the development and deployment of domain specific optimizations. In xtc, domain-specific optimizations can not interact with the downstream C compiler's optimizations. Cross DSO interaction is possible with Lisp macros, but only up to the point where byte code is generated (meaning there are no interactions with machine optimizations).

The introduction of user-specified compile-time computation into a Python compiler follows a lot of existing work done in other languages. While related, string macro and template languages, including the C preprocessor [35], or the Cheetah template language [53], which uses Python, do not include the kind of syntax guarantees that Mython attempts to provide metaprogrammers. The C++ template language permits metaprogramming [67] and partial evaluation [68] to be done at link time (depending on when templates are instantiated). This approach constrains the C++ metaprogrammer to the arcane C++ template syntax, whereas Mython attempts to

permit arbitrary concrete syntax.

The macro systems of various Lisp and Scheme implementations often permit both staged syntax checking and general-purpose computation to be done before compilation [21, 17]. These systems often accomplish similar goals to those of the HLOP mechanism described in Section 3.5.3, but do so in a much safer fashion. This safety comes at a price, however, since the macro languages tend to favor balanced parenthesis languages and tend to abandon either safety (as in the Common Lisp defmacro system) or expressiveness (as with syntax-rules macros or metaocaml's system), or require complicated protocols as with syntax-case macros.

6.1.4 Compiler Pragmas

Compiler pragmas are compiler specific flags and inputs that are embedded in source code. Pragmas are typically not a part of a language standard, nor well integrated with the host language (though popular pragmas do sometimes get moved into a language). Some languages do contribute special syntax for identifying and isolating pragmas. Even in these cases, pragma semantics remain compiler specific.

Compiler pragmas do not provide a good solution to the use cases under consideration, especially since they are compiler specific and by necessity avoid too much coupling with the underlying programming language. Pragmas are only examined here since this mechanism was used by Simon Peyton-Jones et al to add domain-specific optimizations to the Glasgow Haskell Compiler (GHC) [47]. Specific compilers could support language embedding and extension via pragmas, but any extension mechanisms would be specific to a compiler. Pragma extension mechanisms would also not be likely accessible from the host language as first class objects. In the best case, compiler pragmas may be ignored by tool writers. In the worst case, pragmas may change language semantics (such as data alignment and packing), requiring the tool writer to specialize tools for popular compilers. As mentioned earlier, a compiler pragma extension was used to implement domain specific optimizations in the Glasgow Haskell Compiler. The resulting mechanism was used to provide a proof of concept implementation that was able to fully interact with the compiler's other

optimizations. If DSO pragmas become widely popular, they would most likely be moved into the Haskell language (pending surface syntax), and not left as a compiler pragma.

6.2 Related Methods

6.2.1 Staging

The ideas behind parameterized quotation of concrete syntax originate with the MetaBorg usage pattern of Eelco Visser's Stratego/XT framework [12], but have staging [59] mixed in as a means of containing extensibility. Mython's approach to language extension can be contrasted to the possibly arbitrary extensibility proposed by Ian Piumarta [48]. Mython's goals of allowing user-specified optimization originate from Peyton Jones et al. [47], which can be traced further back to Wadler's paper on shortcut deforestation [72]. In his paper on deforestation, Wadler also introduces "higher-order macros", which bear more than a passing resemblance to the HLOP work done for both Manticore [24] and Mython.

6.2.2 Extensible Compilers

Norman Ramsey's work on C−− is related to Mython. Ramsey introduces a compile-time dynamic language, Lua, into the C−− compiler [51]. Ramsey uses the interpreter as a glue and configuration language [45] to compose various phases of the C−− compiler. Similarly, MyFront lacks command line options because users can stage front-end extensions, and they can control and extend the back-end using the `myfront()` quotation function.

6.3 Extensible Languages

6.3.1 Caml p4

Caml p4 is a preprocessor that allows users to define LL(1) grammars and map these onto Ocaml code [20]. By virtue of being a preprocessor, p4 only offers a limited

form of front-end extensibility. Caml p4 also limits the forms of grammars users can define to the LL(1) family of formal languages. In contrast, a language such as Fortress (Section 6.3.6) offers a similar preprocessing pass for user-defined syntax, but the Fortress compiler can evaluate user-defined computations allowing even context sensitive parses.

6.3.2 Converge

This subsection describes Laurence Tratt's Converge programming language [62, 61]. Converge is like Mython in many ways, since Converge's syntax seems to borrow heavily from Python's syntax. Tratt's primary contribution is a language that supports compile-time metaprogramming (CTMP). Mython requires CTMP to achieve embedded syntax, a use case demonstrated by Tratt in the Converge documentation.

6.3.3 eXTensible C

Robert Grimm's eXTensible C (xtc) language adds concrete syntax macros to C [28, 29]. The concrete syntax macros in xtc are specialized to only permit language embeddings in C. The xtc tool precludes the arbitrary mixtures of language that are allowed by Stratego while still only allowing language embeddings at compile time. Writing tools with xtc or using tools that process xtc extensions are not straightforward unless the tool writer is either a downstream consumer of the C code that xtc generates, or the tool embeds the xtc system itself. In xtc, domain-specific optimizations can not interact with the downstream C compiler's optimizations.

6.3.4 F-sub

Luca Cardelli provided syntactic extensibility in his language, F<: [14]. F<: appears to be capable of supporting both language embedding and extension. Grammars are first class objects in the language, constructs exist to allow new productions to be added to existing non-terminals, and the core language's grammar is exposed as a

part of the language. F<: restricts grammars to LL(1), using an extensible recursive-descent parser, and does not allow lexical conventions to be modified.

6.3.5 FLEX

Alan Kay's FLEX computation environment provided a programming language by the same name [34]. The FLEX programming language allowed for the definition of both new operators and syntax. These claims are made in Kay's master thesis:

> New binary and unary operators may be declared giving the programmer powerful control over the language itself. [...] FLEX may also be extended by either modifying itself via the compiler-compiler contained in the language or a wholly new language may be created using the same tools.

Some of these ideas have persisted in the Smalltalk language, and feed into Ian Piumarta's work on Squeak.

6.3.6 Fortress

The Fortress programming language [2] supports user embedding of domain-specific languages via *syntax expanders*. Fortress users define a syntax expander by defining a pair of delimiters, a binding identifier, and a subexpression. Users can then escape into domain-specific concrete syntax by using the new delimiter pair. The Fortress compiler evaluates the expander subexpression as part of a preprocessor which is run before any of the traditional front-end. The preprocessor binds the delimited source to the identifier in the evaluation environment, and expects the subexpression to evaluate to Fortress abstract syntax. Syntax expanders are similar to quotation functions in Mython (Section 3.3), but only provide front-end extensibility. Unlike Mython, Fortress avoids name capture in syntax expanders by rewriting binding and subsequent use forms in the output syntax. This approach is similar to how some Scheme implementations handle hygenic macros [21].

6.3.7 Lisp and Scheme

As they are implemented in Lisp and Scheme, syntax macros provide the kind of interactive flexibility not present in static language development systems. Macros allow language implementors, library writers and users to make changes to the language's syntax at all stages of development (including compiler-compile time, compile time, load time, and run time). Changes to the syntax are still constrained since additional syntax must use parenthesis to delimit hierarchy, or revert to using a custom parser. Macros also have strange interactions with the static and dynamic properties of some module system implementations, and cross-module macro interactions require explicit management [23].

6.3.8 OMeta/COLA

Ian Piumarta presented an extensible Smalltalk kernel in his invited talk at DLS 2006 [48]. The kernel fully exposes the language's implementation to itself and allows users to change all aspects of the language from inside the language. The result is closely related to the Stratego modification proposed in Chapter 2 The result to date is OMeta/COLA [77].

6.3.9 Perl 6

Perl 6 extends the native regular expression support in Perl 5 with support for recursive descent parsing. Specifically, the rule primitive allows an identifier to be associated with an EBNF-like definition of a production. Perl 6 also exposes the language implementation, allowing custom parsers to replace the Perl parser at run-time [60, 74].

6.3.10 Template Haskell

This extension to Haskell has been used for doing embedded domain-specific languages (EDSL's) [54]. Seefried argues that Template Haskell enables "extensional metaprogramming", where both the language front end and back end are available

to a domain specific language implementation. Template Haskell still supposes that the domain specific language will use the Haskell concrete syntax, extended using operator extensions and combinators.

CHAPTER 7
CONCLUSIONS

This chapter concludes this work by first reviewing the thesis, and the contribution made by each chapter towards validating it. It then looks at the future of this work, providing a plan for building increasingly extensible and capable languages.

Increasing the reflectivity of languages improves or enables language evolution, language embedding, domain-specific optimization, and language tool development. Chapter 1, and specifically Section 1.2, looked at each of these scenarios in some detail. It then described reflection in languages, program extensibility, and how decomposition of language implementations can achieve language extensibility. This approach to reflection and extension characterizes the methods advocated and demonstrated in the remainder of the dissertation.

Chapter 2 began by examining existing language embedding and evolution techniques. It then showed how self-application of the embedding techniques enhances language extensibility. Chapter 2 also provided some ideas for how to both apply and contain the resulting language extensibility.

Chapter 3 introduced a platform for experimenting with the techniques developed in Chapter 2. Mython solves several technical problems with the vision of Section 2.2.4 by using compile-time metaprogramming in conjunction with a visible compiler. As a result, Mython affords users both syntactic and semantic extensibility. Chapter 3 concluded by outlining how a user can implement domain-specific optimizations in Mython without the benefit of prior support for such extension.

Chapter 4 focused on rewrites as an approach to implementing domain-specific optimization. It described several existing and new algorithms for term rewriting, and showed that the new algorithms will not add prohibitive overhead to an optimizer. Domain-specific optimizations serve as an example of the back-end extensibility described in Section 1.4.3. A language implementor can use the techniques in

```
00000000 <myproc>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   81 ec 40 00 00 00       sub    \$0x40,%esp
   9:   8b 45 08                mov    0x8(%ebp),%eax
   c:   40                      inc    %eax
   d:   c9                      leave
   e:   c3                      ret
```

Figure 7.1: Example of an x86 assembly function.

Chapter 4 to incrementally increase extensibility without requiring a fully open or extensible compiler.

Chapter 5 provided a concrete example of using Mython to implement and embed a domain-specific language. The MyFEM domain-specific language provides a high-level interface for specifying differential equations and boundary conditions. Chapter 5 illustrates how domain-specific languages can reduce source code complexity, achieving an order of magnitude reduction in line count. MyFEM also demonstrates how users can leverage parts of Mython to create a domain-specific language.

Mython provides a platform for experimentation with multilanguage systems. Future work will include further experimentation with language embedding. A keystone of this effort will be the integration of a C compiler. This will dispose of the need for external foreign function interface tools, since the C code can be inspected directly. The machine code generating back-end of the compiler can be used to perform just-in-time compilation.

The following is an example of one interaction between a statically compiled language and an interpreted language. Figure 7.1 provides an example machine language subroutine. Figure 7.2 illustrates a Python script that allocates memory in the heap using the `array` module, and then builds a dynamic foreign function interface to the machine code. Future work will build upon this demonstration using the open C compiler, automating the generation of the assembly code and the foreign function interface. Mython will further extend the compilation infrastructure to build safer and higher-level interfaces.

```
import array, ctypes
fpitoi = ctypes.CFUNCTYPE(ctypes.c_int, ctypes.c_int)
inc_code = [0x55, 0x89, 0xe5, 0x81, 0xec, 0x40, 0x00, 0x00,
            0x00, 0x8b, 0x45, 0x08, 0x40, 0xc9, 0xc3]
inc_array = array.array("B", inc_code)
inc = fpitoi(inc_array.buffer_info()[0])
```

Figure 7.2: Example of embedding machine code in Python.

Continued development of MyFEM will show the usefulness of the Mython platform, and increase Mython's utility. MyFEM serves as a useful demonstration of building an embedded compiler in Mython. This demonstration includes the addition of an embedded static type system. Future work will bridge MyFEM's back-end with the just-in-time compilation framework. Adding just-in-time compilation allows users to quickly build and modify scalable scientific simulations.

In a similar fashion to MyFEM, Mython allows users to quickly build and modify programming languages. This dissertation has shown how reflection makes Mython possible and increases the extensibility of programming environments. In turn, extensible programming environments enable faster language evolution, language embedding, domain-specific optimization, and tool development.

APPENDIX A
REWRITE DATA

The following tables provide detailed data graphed in Figure 4.8 and Figure 4.9.

Table A.1: Rewrite times, in microseconds (μs), for matching term inputs (with slowdown relative to the greedy optimizer in parenthesis).

Term size	Greedy	Bottom-up	Tiered Greedy	Tiered bottom-up
9	28	167 (5.96×)	16 (0.57×)	197 (7.04×)
99	139	2718 (19.55×)	175 (1.26×)	3136 (22.56×)
999	1368	45418 (33.2×)	1728 (1.26×)	45493 (33.26×)
9999	31159	466740 (14.98×)	25343 (0.81×)	546482 (17.5×)

Table A.2: Rewrite times, in microseconds (μs), for non-matching input terms (with slowdown relative to the greedy optimizer in parenthesis).

Term size	Greedy	Bottom-up	Tiered Greedy	Tiered bottom-up
9	11	42 (3.82×)	6 (0.55×)	40 (3.64×)
99	25	482 (19.28×)	24 (0.96×)	513 (20.52×)
999	156	3013 (19.31×)	246 (1.58×)	6041 (38.72×)
9999	1781	44539 (25.01×)	2942 (1.65×)	84492 (47.44×)

Table A.3: Rewrite times, in microseconds (μs), for greedy and strategy based greedy optimizers (with slowdown relative to the greedy optimizer in parenthesis).

Term size	Greedy	Strategies
9 (in-domain)	28	544 (19.43×)
99 (in-domain)	139	9702 (69.8×)
999 (in-domain)	1368	194642 (142.28×)
9999 (in-domain)	31159	3316602 (106.44×)
9 (out-of-domain)	11	161 (14.64×)
99 (out-of-domain)	25	3912 (156.48×)
999 (out-of-domain)	156	45217 (289.85×)
9999 (out-of-domain)	1781	613676 (344.57×)

APPENDIX B
MYFEM DEFINITIONS

The following two sections define the MyFEM domain-specific language's concrete syntax and abstract syntax, respectively.

B.1 MyFEM Concrete Syntax

The MyFEM concrete syntax is given below:

```
# Top level

start : line* ENDMARKER

line : (decl | constraint) NEWLINE

# Declarations

decl : ( 'TestFunction' id_list
       | 'UnknownField' id_list
       | 'CoordinateFunction' id_list
       | 'Field' id_list
       )

id_list : NAME (',' NAME)*

# Constraints

constraint : expr ('=' expr)*
```

```
expr : arith_expr (vec_op arith_expr)*

vec_op : 'cross' | 'dot'

arith_expr : term (('+'|'-') term)*

term : factor (('*'|'/') factor)*

factor : ('+' | '-' | 'grad' | 'div' | 'trans') factor
       | power

power : atom ['^' factor]

atom : ( '(' expr ')' ['|' bounds]
       | '<' expr ',' expr '>' [bounds]
       | NAME
       | NUMBER
       )

bounds : '_' NAME
```

B.2 MyFEM Intermediate Language

The following is an ASDL definition of the MyFEM intermediate language:

```
module BVP version "$Revision$"
{
    closure  = BVPClosure (decl * decs,
                           stmt* body)

    decl = VDec(identifier id, string ty, string? dim, string? init)

         attributes (int lineno)

    stmt = Loop(identifier? loop_var, identifier  loop_iter,
                stmt * body)
         | Assign (lvalue lhs, expr rhs)
         | SumAssign (lvalue lhs, expr rhs)
         | Special (identifier sid, object? options)

         attributes (int lineno, expr result)

    expr = Index (expr iexpr, expr index)
         | Pow (expr lexpr, expr exp_expr)
         | Mult (expr* exprs)
         | Add (expr* exprs)
         | Sub (expr* exprs)
         | Sum (identifier? loop_var, identifier loop_iter,
                expr sexpr)
         | Var (identifier vid)
         | Const (object val)
         | SpecialExpr (identifier sid, object? options)
```

```
        attributes (int lineno)

    lvalue = LIndex (lvalue expr, expr index)
           | LVar (identifier lvid)

           attributes (int lineno)
}
```

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification, Version 1.0α. Technical report, Sun Microsystems Inc., September 2006.

[3] Martin Alnæs and Kent-Andre Mardal. SyFi User Manual, September 2007. Available at `http://www.fenics.org/wiki/Documentation`.

[4] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 13–23, New York, NY, USA, 1994. ACM.

[5] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.3.2, Argonne National Laboratory, September 2006. Available at `http://www.mcs.anl.gov/petsc`.

[6] David M. Beazley. SWIG: An Easy-to-Use Tool for Integrating Scripting Languages with C and C++. In *Proceedings of the Fourth USENIX Tcl/Tk Workshop*. USENIX Assoc., 1996.

[7] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with Blast. In M. Cerioli, editor, *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005, Edinburgh, April 2-10)*, LNCS 3442, pages 2–18. Springer-Verlag, Berlin, 2005.

[8] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[9] Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Trans. Program. Lang. Syst.*, 21(4):813–847, 1999.

[10] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 2008. (To appear).

[11] Martin Bravenboer, Rob Vermaas, Jurgen Vinju, and Eelco Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In Robert Glück and Mike Lowry, editors, *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *Lecture Notes in Computer Science*, pages 157–172, Tallin, Estonia, September 2005. Springer.

[12] 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.

[13] Brett Cannon. Python Enhancement Proposal PEP 339 – Design of the CPython Compiler, 2005. Available at `http://www.python.org/dev/peps/pep-0339/`.

[14] Luca Cardelli. An Implementation of F<:. Technical report, Digital Equipment Corporation, February 1993.

[15] James Cheney and Ralf Hinze. Phantom types, 2003. Available at `http://www.informatik.uni-bonn.de/~ralf/publications/phantom.pdf`.

[16] 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.

[17] William Clinger and Jonathan Rees. Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–162, New York, NY, USA, 1991. ACM Press.

[18] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, April 2007.

[19] Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting Haskell Strings. In *Practical Aspects of Declarative Languages 8th International Symposium, PADL 2007*. Springer-Verlag, January 2007.

[20] Daniel de Rauglaudre. Camlp4 - Reference Manual, September 2003. Available at `http://caml.inria.fr/pub/docs/manual-camlp4/index.html`.

[21] R. Kent Dybvig. Writing hygienic macros in Scheme with syntax-case. Technical Report 356, Indiana University, Bloomington, Indiana, USA, June 1992.

[22] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[23] Matthew Flatt. Composable and compilable macros:: you want it when? In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 72–83, New York, NY, USA, 2002. ACM Press.

[24] Matthew Fluet, Nic Ford, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Status Report: The Manticore Project. In *Proceedings of the 2007 ACM SIGPLAN Workshop on ML*, October 2007.

[25] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[27] Mark S. Gockenbach. *Understanding and Implementing the Finite Element Method*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[28] Robert Grimm. xtc: Making C Safely Extensible, August 2004. Available athttp://www.cs.nyu.edu/rgrimm/papers/dslopt04.pdf.

[29] Robert Grimm. Systems Need Languages Need Systems! In *Second ECOOP Workshop on Programming Languages and Operating Systems*, July 2005.

[30] J. Hoffman, J. Jansson, C. Johnson, M. Knepley, R. C. Kirby, A. Logg, and L. R. Scott. The FEniCS Project. Available at http://www.fenics.org/.

[31] International Business Machines. *IBM Informix ESQL/C Programmer's Manual, Version 9.53*. International Business Machines, Armonk, NY, USA, March 2003.

[32] Stephen C. Johnson. Lint, a C program checker. Technical Report 65, Bell Labratories, 1977.

[33] Niel D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[34] Alan C. Kay. FLEX - a flexible extendable language. Master's thesis, University of Utah, 1968.

[35] Brian W. Kernignhan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, USA, 1988.

[36] Robert C. Kirby. Algorithm 839: Fiat, a new paragigm for computing finite element basis functions. *ACM Trans. Math. Softw.*, 30(4):502–516, 2004.

[37] Robert C. Kirby and Anders Logg. A compiler for variational forms. *ACM Trans. Math. Softw.*, 32(3):417–444, 2006.

[38] Robert C. Kirby and Anders Logg. Benchmarking domain-specific compiler optimizations for variational forms. *ACM Trans. Math. Softw.*, 35(2):1–18, 2008.

[39] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, 2005.

[40] Matthew G. Knepley and Dmitry A. Karpeev. Mesh Algorithms for PDE with Sieve I: Mesh Distribution. *Scientific Programming*, 2008. (To appear).

[41] Anders Logg. *Automation of Computational Mathematical Modeling*. PhD thesis, Chalmers University of Technology, 2004.

[42] Jacques Malenfant, M. Jacques, and F.-N. Demers. A Tutorial on Behavioral Reflection and its Implementation. In *Proceedings of Reflection '96*, April 1996.

[43] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[44] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[45] John K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the Winter USENIX Conference*, pages 133–145, January 1990.

[46] John K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, March 1998. Available at `http://home.pacbell.net/ouster/scripting.html`.

[47] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop (HW '01)*, pages 203–233, Firenze, Italy, September 2001.

[48] Ian Piumarta. Open, extensible dynamic programming systems – or *just how deep is the 'dynamic' rabbit hole?*, October 2006. Presented at the Dynamic Languages Symposium (DLS) 2006. Slides available at `http://www.swa.hpi.uni-potsdam.de/dls06/`.

[49] The Programming Languages Team. PLT Scheme. Available at http://www.plt-scheme.org/.

[50] Terry Quatrani. *Visual Modeling with Rational Rose 2002 and UML, 3rd Edition*. Addison-Wesley Professional, Indianapolis, IN, USA, 2002.

[51] Norman Ramsey. Embedding an interpreted language using higher-order functions and types. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 6–14, New York, NY, USA, 2003. ACM Press.

[52] Jonathan Riehl. Assimilating MetaBorg: Embedding language tools in languages. In *Proceedings of the Fifth International Conference on Generative Programming and Component Engineering (GPCE'06)*, October 2006.

[53] Tavis Rudd. Cheetah – The Python-Powered Template Engine, 2007. Available at http://www.cheetahtemplate.org/.

[54] Sean Seefried, Manuel Chakravarty, and Gabriele Keller. Optimising Embedded DSLs using Template Haskell. In *Proceedings of the Third Internaltion Conference on Generative Programming and Component Engineering (GPCE'04)*, volume 3286 of *Lecture Notes in Computer Science*, pages 136–167. Springer, October 2004.

[55] Tim Sheard. Languages of the future. *SIGPLAN Not.*, 39(12):119–132, 2004.

[56] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *ECOOP'07: 21st European Conference on Object-Oriented Programming*, 2007.

[57] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, January 1982.

[58] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[59] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.

[60] Audrey Tang. Pugs: an implementation of Perl 6, October 2006. Presented at the Dynamic Languages Symposium (DLS) 2006. Slides available at http://www.swa.hpi.uni-potsdam.de/dls06/.

[61] Laurence Tratt. Compile-time meta-programming in a dynamically typed OO language. In *Proceedings of the Dynamic Languages Symposium*, pages 49–64, October 2005.

[62] Laurence Tratt. The Converge programming language. Technical Report TR-05-01, Department of Computer Science, King's College London, February 2005.

[63] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Softw. Pract. Exper.*, 30(3):259–291, 2000.

[64] M. G. J. van den Brand and P. Klint. *ASF+SDF Meta-Environment User Manual, Revision: 1.149*, January 2005.

[65] Guido van Rossum. Python Reference Manual (2.5), September 2006. Available at `http://www.python.org/doc/2.5/ref/ref.html`.

[66] Guido van Rossum. Python Library Reference (2.5.2), February 2008. Available at `http://www.python.org/doc/2.5.2/lib/lib.html`.

[67] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[68] Todd L. Veldhuizen. C++ templates as partial evaluation. In *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ed. O. Danvy, San Antonio, January 1999.*, pages 13–18. University of Aarhus, Dept. of Computer Science, January 1999.

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

[70] 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.

[71] Jens Vollinga. GiNaC: Symbolic computation with C++. *Nucl. Instrum. Meth.*, A559:282–284, 2006.

[72] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science, (Special issue of selected papers from 2nd European Symposium on Programming)*, 73(2):231–248, 1990.

[73] Philip Wadler, January 2006. Available at `http://wadler.blogspot.com/2006/01/bikeshed-coloring.html`.

[74] Larry Wall. Programming is Hard, Let's Go Scripting..., December 2007. Available at `http://www.perl.com/pub/a/2007/12/06/soto-11.html`.

[75] Malcolm Wallace and Colin Runciman. Haskell and XML: generic combinators or type-based translation? In *ICFP '99: Proceedings of the fourth ACM SIGPLAN*

international conference on Functional programming, pages 148–159, New York, NY, USA, 1999. ACM Press.

[76] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Chris S. Serra. The Zephyr Abstract Syntax Description Language. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 213–228, October 1997.

[77] Alessandro Warth and Ian Piumarta. OMeta: an Object-Oriented Language for Pattern Matching. In *Proceedings of Dynamic Languages Symposium '07*, October 2007.

[78] Niklaus E. Wirth. Recollections about the development of pascal. In *History of programming languages—II*, pages 97–120. ACM, New York, NY, USA, 1996.