Language Embedding and Optimization in Mython

Jonathan Riehl
University of Chicago
jriehl@cs.uchicago.edu

Abstract

Mython is an extensible variant of the Python programming language. Mython achieves extensibility by adding a quotation mechanism that accepts an additional parameter as well as the code being quoted. The additional quotation parameter takes the form of a Mython expression. Unlike other user code, Mython evaluates the quotation parameter at compile-time. The result of the compile-time expression is a function that is used to both parse the quoted code, and extend the compile-time environment. 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 paper 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 using a little rewrite language.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.6 [*Software Engineering*]: Programming Environments—Interactive environments

General Terms Design, Languages

Keywords extensible languages, open compilers, compile–time metaprogramming

1. Introduction

Mython lets users extend the language in multiple 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 using the same Mython extensions. Using side effects in their compile-time code, users can specify domain-specific optimizations. These optimizations can perform custom simplification of code, which can also provide further optimization opportunities to the native optimizer.

DLS'09, October 26, 2009, Orlando, Florida, USA.
Copyright © 2009 ACM 978-1-60558-769-1/09/10...$10.00

The Mython language is realized by the MyFront compiler, a tool for translating Mython into Python byte-code modules. This paper 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. This paper's primary contribution is description of a working synthesis of compile-time metaprogramming and open compilation. This paper then provides a demonstration of how this synthesis enables domain-specific optimization.

The remainder of this paper has four parts. The first part, consisting of Section 2 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. Section 3, and Section 4 provide the second part, defining both the Mython language, and describing the MyFront compiler that implements the language. The third part, found in Section 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 paper discusses related work, future work, and gives a brief conclusion.

2. Background

This section reviews the motivations behind and methods used in the Mython language. The first subsection discusses language embedding and applications of extensible languages. This section then refines earlier ideas by making evaluation stages explicit, defining acceptable quotation arguments, and explaining how the various evaluation environments are handled. It then gives an overview of domain-specific optimizations, one application of the quotation mechanism. It concludes with a brief discussion of the Python language.

2.1 Language Embedding

The Mython developers seek a platform for experimenting with multilanguage systems. This platform should have the flexibility of a dynamic language, while providing users with the option of working with static languages. Many tools address language interoperability using middleware or a common application-binary interface. Fewer tools assist in interoperability at the syntactic level, where code for multiple languages can appear in the same source file. This subsection describes several different methods for syntactic language embedding, arriving at Mython's approach.

This paper uses the following terminology to describe language embedding. A language embedding implementation realizes a *hybrid language*. The concrete syntax of a hybrid language consists of a *host language*'s syntax, with additional (or re-purposed) rules that include some or all of the concrete syntax of the *embedded language*. In many cases the embedding implementation is a translator program that outputs code in a *target language*. Translators commonly focus on translating embedded code constructs to the host language, with the target and host languages being identical.

```
c_source = """int incnum (int x) {
  return x + 1;
}"""
c_ast = cfront(c_source)
incnum = cback(c_ast, 'incnum')
print incnum(41)
```

Figure 1. Example of embedding C in Python as a string.

```
cquote c_ast:
    int incnum (int x) {
      return x + 1;
    }
incnum = cback(c_ast, 'incnum')
print incnum(41)
```

Figure 2. Example translator source for a C and Python hybrid.

```
quote [staticcfront] c_ast:
    int incnum (int x) {
      return x + 1;
    }
incnum = cback(c_ast, 'incnum')
print incnum(41)
```

Figure 3. Example of a Mython program.

2.1.1 Embedded Strings

The simplest method for language embedding is for programmers to use strings in the host language to contain embedded code. This method fails to be a proper syntactic embedding for two reasons. Common string syntax requires that certain characters, such as formatting characters or string delimiters, be escaped using a special lexical syntax. The host's special lexical syntax takes precedence over the embedded code's concrete syntax, requiring developers to encode the embedded language using escape sequences, and reducing readability. Second, embedded strings can hide syntax errors that would be caught earlier by a syntactic embedding.

Figure 1 gives an example of embedding C in Python using Python's multiline strings. This and later examples demonstrate variations of a multilanguage program that outputs 42. These examples assume the availability of two functions. The `cfront` function parses C source code, returning an abstract-syntax tree (AST) for the input source, or raises an exception in the presence of a syntax error. The `cback` function is a (hypothetical) code and wrapper generator, accepting abstract syntax and a string label for a C symbol to wrap, and returning a Python function that calls the underlying C function.

Figure 1 uses Python's multiline string syntax to overcome some readability issues. Multiline strings assist embedding code by allowing unescaped formatting characters and single quotation marks. This approach still breaks down when seeking to embed nested strings that use a similar or identical escape syntax. Readability aside, what happens in this example if a syntax or type error is put into the embedded C source? The static checks handled by the front-end, `cfront`, are deferred to run time. The dynamic host language discards the static error detection benefits the developer would gain from just using separate, static compilation.

2.1.2 Separate Translation

As discussed above, another approach to language embedding is for a programmer to create a hybrid language implementation. Tools such as Stratego/XT [1] assist developers in creating translators, providing tools and methods for syntactic embedding as well as source-to-source translation. Developers that follow this methodology will build translators capable of true syntactic embedding, but have complicated the compile-time pipeline of their users. This method assumes that there is a distinct compilation time for user programs, failing to fully accommodate cases where the host language is either interpreted or compiled at run time. The resulting hybrid language will also be fixed at compiler-compile time to supporting a single embedded language.

Figure 2 is an example of a hypothetical C and Python hybrid language. This example assumes that a tool developer extended the Python syntax, adding a new `cquote` keyword that marks the beginning of a multiline block with embedded C code. This new `cquote` block follows Python's syntactic conventions by lexically delimiting multiline blocks using whitespace. When the translator recognizes a `cquote` block, it calls the `cfront` function, either reporting a static error, or emitting Python source that will rebuild the AST at run time and bind the resulting data structure to the name following `cquote`. The tool's users gain static checks of the embedded code, but lose the flexibility of being able to decompose and recompose the C front-end function. The tool's users also have to perform separate translation, which is a similar complication to the development cycle as separate compilation.

2.1.3 Parameterized Quotation

Previous work proposed building a language environment that was capable of moving the compiler-compile-time and compile-time computations of the separate translation method to either compile or run time [13]. The proposal advocated the creation of a dynamic host language that used a parameterized form of quotation to embed other languages. It further argued that including the same domain-specific languages used to generate the host language would accelerate user contributions toward language evolution, embedding, tools, and optimization.

Mython is a realization of this proposed hybrid language. Figure 3 gives an example of a Mython program with analogous form and identical function to previous examples. Mython's `quote` block has similar syntax to the `cquote` block in Section 2.1.2, but supports an additional expression. Delimited by square brackets, this additional expression allows the user to specify a function for translating the embedded code into Python. This example assumes the compile-time definition of the `staticcfront` function, which adapts the `cfront` function from previous examples to conform to Mython's translation protocols.

Implemented as a compiler from Mython source code to Python bytecode, Mython can eliminate the need for separate compilation or translation. In this case, Mython replaces Python's in-process compiler with a programmable translation and compilation function. This work shows how generalization of the quotation mechanism restores dynamism while affording programmers precise control over how embedded code is handled both during compilation and run time.

2.2 Language Extension

Section 2.1.3 introduced syntax and motivation for generalizing quotation with a parameter. The previous subsection does not explain *when* and *how* the new parameter is evaluated. This section discusses Mython's approach to compile-time metaprogramming (CTMP). Compile-time metaprogramming involves adding an evaluator to the compiler or translator, with syntactically scoped expressions and statements evaluated at compile time instead of run time. These compile-time expressions and statements allow users to generate code without using a separate tool.

In the example from Figure 3, the compiler evaluates the "`staticcfront`" expression at compile time, using an evaluator that handles names and side-effects by explicitly accepting and returning an environment. This compile-time environment is distinct from the run-time environment used in evaluation of the bulk of the host code. The compiler expects the compile-time expression to evaluate to a translation function. Translation functions should accept the run-time name associated with the block, a string with the embedded source, and the compile-time environment. Translation functions return host language abstract syntax, and the possibly modified compile-time environment. The compiler passes the environment returned from the compile-time evaluation of the quotation arguments to later quotation expressions as they are found in the code. Users can bind new names in the compile-time environment using a special translation function provided by the compiler.

Constructs such as conditionals, abstraction, recursion, and other expressions that commonly affect environments used in static analysis are considered solely run-time syntax, and do not change the compile-time environment. Besides allowing extension of the compile-time 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.

2.3 Domain-specific Optimization

The compile-time environment allows users to implement domain-specific optimizations (DSOs). Domain-specific optimizations 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. [10], domain-specific optimizations have been used to implement a faster Haskell string library [5] and perform stream fusion [4] 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 4 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.

The formalism in Figure 4 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

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

Figure 4. A formalization of rewrite specifications.

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 [21]. 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:

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

2.4 Python

Python is an object-oriented programming language with a C/C++-like surface syntax, but a Lisp-like semantics [17]. 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, intro-

```
Module([
  QuoteDef(
    Name('cfront', Load()),
    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 5. Abstract syntax for the Mython example.

spection, and evaluation. Python code objects can be created at run-time 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 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. The Mython Language

This section provides a brief overview of how Mython syntax and semantics differ from the Python language.

3.1 Language Syntax

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

Figure 5 gives an example of the Mython abstract syntax for the program in Figure 3. Mython extends Python syntax at all three phases of parsing: the lexical 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 5 translates to the given abstract syntax. Note that both Figure 5 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.

3.1.1 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

```
(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 6. Part of the lexical stream for the Mython example.

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 shows an example of a multi-line quotation. In this example, the quotation begins following the four space indentation on line 2. 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 6, and illustrates how the leading whitespace on lines 2 through 4 has been stripped from the quoted string.

3.1.2 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
```

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.

3.1.3 Abstract syntax extensions

Python 2.5 extended the Python compiler infrastructure to include an abstract syntax tree intermediate representation [2]. Python formalizes its abstract syntax using the Abstract Syntax Definition Language (ASDL) [22]. Mython extends the abstract syntax by adding a `QuoteDef` constructor to the `stmt` abstract data type. This is formalized in ASDL as the following:

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

```
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 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 7 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.

3.2 Language Semantics

Mython's operational semantics do not differ from the Python language. The Mython implementation follows the method outlined in Section 2.2, 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:

$$\begin{array}{c}(\texttt{QuoteDef}(\texttt{None}, name_{opt}, body), \Sigma_0) \mapsto \\ (\texttt{QuoteDef}(mython, name_{opt}, body), \Sigma_0) \\ \text{where} \\ mython = \texttt{Name('mython', Load())}\end{array} \quad (1)$$

$$\begin{array}{c}(\texttt{QuoteDef}(lang, name_{opt}, body), \Sigma_0) \mapsto (stmts, \Sigma_1) \\ \text{where} \\ myeval = \Sigma_0[\texttt{'myeval'}] \\ (f, \Sigma_0') = myeval(lang, \Sigma_0), \\ (stmts, \Sigma_1) = f(name_{opt}, body, \Sigma_0')\end{array} \quad (2)$$

Rule 1 replaces empty translation expressions with a reference to the default translation function, `mython()`, which is defined for compile time in the initial environment (see Section 4.2). This rule means that a parameter-less quotation will quote Mython code. Rule 2 handles either the user-specified stage-zero expression, or the stage-zero expression inserted by Rule 1.

One should note that Rule 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.

3.2.1 Compile-time computations

Rule 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 evaluator briefly described in Section 2.2, and is defined as follows:

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

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 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.

3.2.2 Compile-time environments

Section 2.2 explained how the compile-time 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. At the entry of a lexical scope, Mython pushes the environment on a stack. At the end of a lexical scope, Mython discards the current compile-time environment, popping and using the environment defined just prior to entry of the lexical scope. Figure 8 shows a partial Mython program, with environment inputs and outputs identified explicitly in the comments. Using the example in Figure 8, stage-zero symbols bound inside the `spam()` function are not visible to other stage-zero code following `spam()`.

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

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

Figure 7. Transformation functions from Mython concrete syntax to abstract syntax.

```
quote [X] x:
    xCode
# X, E_1 = myeval('X', E_0); ast0, E_2 = X('x', 'xCode', E_1)
def spam ():
    quote [Y] y:
        yCode
    # Y, E_3 = myeval('Y', E_2); ast1, E_4 = Y('y', 'yCode', E_1)
# Note that E_4 is dropped at the end of spam()'s lexical scope.
quote [Z] z:
    zCode
# Z, E_5 = myeval('Z', E_2); ast2, E_6 = Z('z', 'zCode', E_2)
```

Figure 8. Example of how the compilation environment flows through a Mython program.

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 an environment for compilation at run time, 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 of a run-time 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 into the compilation environment without going through the quotation mechanism.

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

4. The MyFront Compiler

This section describes the MyFront compiler, an implementation of the Mython language. This section begins with a high-level description of the largely orthodox compiler design, and then shows how the design is reified in the surface language. Ideally, the built-in compiler functions illustrate the designer's expectations for the form and function of the end products of a modular language definition.

4.1 The Compiler Implementation

MyFront is the initial Mython language implementation, and inherits its name from Cfront, the original C++ to C translator [15]. While it is possible to translate from Mython to Python using the simple rewrites defined in Section 3.2, 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 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 3.2, 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.

4.2 The 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.

4.2.1 The `myparse()` built-in function

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

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.

4.2.2 The `mython()` built-in function

$$\texttt{mython} : \mathit{quotefn}$$

The `mython()` function acts as a stage-zero quotation function (see Section 3.2), 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
```

4.2.3 The `myescape()` built-in function

$$\texttt{myescape} : \mathit{mod}_{myast} \rightarrow \mathit{expr}_{pyast}$$

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 3.2.1.

4.2.4 The `myeval()` built-in function

$$\texttt{myeval} : \mathit{mod}_{myast} \times \mathit{env} \rightarrow \texttt{PyObject}$$

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.

4.2.5 The `myfront()` built-in function

$$\texttt{myfront} : \mathit{quotefn}$$

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)
```

```
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 9. Adding an optimization phase to MyFront.

5. Applications

Section 2.3 gave a high-level overview of domain-specific optimizations. This section describes how some of the ideas presented in Section 2.3 can be implemented in Mython. Section 5.1 begins with a discussion of how the MyFront-specific internals can be extended to add one or more optimization passes. Section 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.

5.1 A Little Language for Rewrites

As discussed in Section 4, 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 9 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 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 10, 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
            (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 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 10. The `simple_rw_opt()` function.

```
    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 [20] to compose rewrites and rebinding the composed rewrite in the compile-time environment.

5.2 Shortcut Deforestation

The first example optimization in Section 2.3 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 5.1.

Figure 11 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

```
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 11. An example of using embedded DSL's in Mython to define an optimization.

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 Figure 11, 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 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.

5.3 High-level Operators

High-level operators (HLOPs) are a language construct introduced in the Manticore compiler [7]. 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 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 2.3), since HLOP references are removed before code generation.

6. Related Work

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 [8], or the Cheetah template language [14], which uses Python, do not include the kind of syntax guarantees that Mython attempts to provide metaprogrammers. The C++ template language permits metaprogramming [18] and partial evaluation [19] 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 [6, 3]. These systems often accomplish similar goals to those of the HLOP mechanism described in section 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.

Norman Ramsey's work on C−− is related to Mython. Ramsey introduces a compile-time dynamic language, Lua, into the C−− compiler [12]. Ramsey uses the interpreter as a glue and configuration language [9] 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.

The ideas behind parameterized quotation of concrete syntax originate with the MetaBorg usage pattern of Eelco Visser's Stratego/XT framework [1], but have staging [16] 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 [11]. Mython's goals of allowing user-specified optimization originate from Peyton Jones et al. [10], which can be traced further back to Wadler's paper on shortcut deforestation [21]. 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 [7] and Mython.

7. Future Work

Section 4.2 focuses on the bare set of quotation functions needed to stage Mython itself and escape into the compiler environment. Future work will attempt to expand the Mython standard library to support a variety of domain-specific language description languages, including Stratego and the Syntax Definition Framework. This support will speed assimilation of front-ends, further Mython extensions, as well as the development of back-end tools for a variety of popular languages. The goal of adding these tools is to make creating multilanguage programs similar to the example in Figure 3 a reality.

At the time of writing, there have been several Python 3.0 releases. MyFront's host language, concrete syntax, and abstract syntax either uses or extends Python version 2.5. Moving Mython to support and extend Python 3.0 would not only ensure it survives into the next Python development series (Python 2.X code will not always work in Python 3.X), but provide some new features that the Mython back-end is suited to handle. Specifically, Python 3.0 adds optional type annotations. These type annotations are not used by the Python 3.0 compiler. MyFront's extensibility should allow users to quickly add semantics to their annotations, either performing run-time type checking (replacing decorators) or start prototyping a static type system.

Section 5 quickly built an example optimization framework with the intention of showing that Mython can support user-defined domain-specific optimizations. However, this framework serves only as a proof of concept, ending with an unsafe and hard to comprehend macro system inspired by work in Manticore that is not exposed to the surface language. To properly support DSOs based on static call-site matching, Mython should look at adding a proper macro system and making the HLOP compile-time environment behave and integrate better with the run-time environment.

8. Conclusion

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 paper has focused on the key mechanism used to provide this extensibility: parameterized quotation. 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.

Acknowledgments

The author would like to acknowledge the helpful feedback of John Reppy, the anonymous reviewers, Martin von Löwis, and the pointers into the literature provided by members of the Manticore team and Jacob Matthews.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

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

[10] 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.

[11] 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/`.

[12] 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.

[13] 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.

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

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

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.