RubyWrite: A Ruby-embedded Domain-specific Language for High-level Transformations

Arun Chauhan Andrew Keep Chun-Yu Shei Pushkar Ratnalikar

School of Informatics and Computing, Indiana University

Bloomington, IN 47405, USA

{achauhan,akeep,cshei,pratnali}@cs.indiana.edu

Indiana University Technical Report TR704

January 2013

Abstract

`RubyWrite` is a Domain Specific Language (DSL), embedded within Ruby, with the goal of providing an extensible, effective, portable, and easy to use framework for encoding source-level transformations. Embedding within Ruby provides access to the powerful features of Ruby, including its meta-programming capabilities. Ruby's multi-paradigm programming model and flexible syntax drove our decision to use it as a host language. Easy integration with C interfaces lets us move performance critical operations to C, or link with external libraries. `RubyWrite` consists of three components, a tree builder, an unparser, and a tree rewriter. It has been used in multiple compiler research projects and as a teaching aid in a graduate-level compilers course. We expect `RubyWrite` to be widely applicable and a proof of concept in leveraging a modern language to write a portable compiler infrastructure core.

Contents

Chapter 1

Introduction

We describe a Domain Specific Language (DSL) embedded in Ruby, aimed at simplifying the task of source-level transformations through term-rewriting. The motivation for the DSL, called `RubyWrite`, is our need for a compiler development environment allowing easy integration of source-level transformations with program analysis.

After spending frustrating months trying to develop a compiler for MATLAB in C++ a few years ago, we made the decision to move to a domain-specific language for compiler development. The C++ development experience proved slow and error-prone causing us to seek a more suitable tool. After a pilot study, we began using Stratego/XT [2]. It provides powerful term-rewriting capabilities, clean and compact syntax, and built-in support for data flow analysis. Using Stratego/XT lead to an immediate order of magnitude productivity gain. It was also an effective teaching aid in a graduate-level compilers course, letting students implement advanced compiler techniques within class assignments—something that would be unthinkable in a general-purpose language like C++ without substantial preparatory efforts.

As the compiler development effort in our research group progressed, the analyses became more and more sophisticated. We needed static-single assignment form, dependence analysis, and advanced memory-behavior analysis, all of which were naturally expressed in terms of graphs. While the static-single assignment algorithm can be reworked to operate directly on the abstract syntax trees, other analyses resisted such adaptation unless algorithmic efficiency was compromised. For instance, a vectorizer for Octave [21], encodes dependence analysis within the functional paradigm of Stratego by giving up the efficiency of a graph-based algorithm that uses global lookup. As we escaped into C for more and more analysis and related transformation work, we found ourselves wanting a more complete and flexible language environment.

Using a special purpose language in graduate compilers class also had its limitations. Even though students were sufficiently mature and motivated to put in the effort of going through the somewhat steep learning curve, unfamiliar syntax, unusual semantics, and pure functional nature with which several of the students had no prior experience, all added to the hurdles that students had to cross.

`RubyWrite` addresses these issues. The flexible syntax and powerful meta-programming features of Ruby make it a good host language for an embedded DSL. Its unique combination of functional and object-oriented features enable leveraging the best of both worlds. `RubyWrite` provides a functional interface to writing rewriting-based source-level transformations. At the same time Ruby's object-oriented programming paradigm is closer to more familiar mainstream languages, lowering the barrier to using the language. Its large (and growing) collection of libraries provide a diverse set of components to aid in compiler development, for example, the

Ruby Graph Library [14] and Ruby bindings for LLVM [11]. The growing popularity of the language is resulting in a heightened interest in addressing performance bottlenecks [8, 15, 17]. A C-extension mechanism is also well integrated in Ruby. Combined with libraries, such as RubyInline [18], hotspots can be cleanly optimized by rewriting in C. Some of our own research is also aimed at improving Ruby performance.

`RubyWrite` has attracted overwhelmingly positive feedback as a teaching aid. While graduate students might be unfamiliar with Ruby, they are well versed in object-oriented programming and conversant with the C-like syntax used by many contemporary programming languages. There is a higher level of motivation in learning a modern general-purpose language, instead of a special purpose language that is unlikely to be used outside the classroom. Finally, it is a small incremental step learning `RubyWrite` once students familiarize themselves with Ruby.

Increasing our confidence in our approach is our experience with optimizing large programs written in MATLAB. Complex algorithms can be coded relatively quickly in MATLAB and then hotspots identified and optimized. We expect a similar strategy will enable us and others to achieve high programming productivity in developing compilers using `RubyWrite`, while ensuring it is not stymied by performance bottlenecks.

While the idea of embedded DSLs for compiler writing is not novel, we believe that using a language that supports multi-paradigm programming imparts unique advantages to `RubyWrite`. We have used Ruby to build our system, but it could be easily substituted by another similarly equipped language. More importantly, we have learned several important lessons in this process that could be valuable to others building or using similar tools.

1. Introducing new students or team members to `RubyWrite` is surprisingly easy, even when they are not familiar with Ruby, since its syntax borrows so heavily from other popular languages.

2. Although Ruby has a reputation for being slow, our projects have not run into performance walls.

3. Standard Ruby mechanism can be leveraged to use libraries like the Ruby Graph Library, and for writing C and C++ extensions, both of which have turned out to be handy.

4. Our overall productivity with `RubyWrite` is on par with the gains we saw using purely functional Stratego/XT, without the disadvantages of steep learning curve and awkwardness of incorporating imperative behavior, when it provides the most convenient mechanism to implement certain compiler functions.

5. Using a mature, broadly supported language makes all program development arsenal of debugging, profiling, and interactive sessions automatically to `RubyWrite` users, which is again something that turns out to be very valuable.

6. Finally, groups outside our own have expressed interest in using this tool for their own compiler projects and graduate level compiler courses.

`RubyWrite` is written entirely in Ruby. It can be downloaded as an anonymous user from its version control repository and can be installed as a "gem".

Chapter 2

Ruby

Ruby is a dynamically-typed, purely object-oriented language created by Yukihiro Matsumoto (Matz) [7, 20]. Ruby draws heavily from Smalltalk, but incorporates syntax for branching and looping. Ruby also provides facilities for functional programming, including a syntax for closures[1].

Similar to Smalltalk, Ruby uses single-inheritance for class hierarchies and provides a way to write mixins for shared functionality through modules. In addition to working as mixins, modules are also used to create namespaces. Class and module definitions can be nested, providing a way to group related functionality. Classes and modules are left “open”, allowing programmers to add, remove, or redefine methods. This facility along with a simple way to trap “missed” messages and evaluate code at runtime translate into powerful meta-programming capabilities. The bodies of class and module definitions are executable, so it is possible to write methods that are executed at definition time. Ruby uses a prefix format to indicate the types of variables; constants begin with a capital letter, global variables begin with `$`, class variables begin with `@@`, instance variables begin with `@`, and local variables begin with lower case letters. Classes and modules can define both “class” methods, similar to static methods in Java, and “instance” methods, similar to normal Java methods. Beyond the traditional alphanumeric characters for method names, `?`, `!`, and `=` can be used as suffixes for method names. Special symbolic names such as those to indicate equality (`==`) or array reference and assignment (`[]` and `[]=`) can be defined on any class or module.

In addition to standard looping constructs, Ruby also provides iterators, where a block of code may be passed to a method that can be invoked by the iterator using `yield`, or stored away as a `Proc` to be invoked later. These blocks are full closures and can be defined with either curly braces (`{}`) or the `do` and `end` keywords. Ruby allows parentheses to be dropped when the meaning of an expression is unambiguous. Embedded DSL writers utilize this feature to blur the line between keywords and method calls. Integers, floating point numbers, strings, arrays, hashes, ranges, and regular expressions all have a literal syntax. Ruby allows for interpolated strings using the `#{}` syntax and borrows “here” documents from Perl. `RubyWrite` uses these common Ruby idioms to provide a clean and succinct syntax.

The rest of this chapter provides a basic overview of the Ruby language, highlighting the main differences from C-like languages. More complete descriptions are available through several books and online resources [7, 20, 16].

[1] The new syntax is available as of Ruby 1.9.x, but a similar facility exists in previous versions of Ruby through the `lambda` method.

2.1 Basic Syntax

Ruby expressions follow the commonly used C-style syntax. In addition to the usual arithmetic and logical operators in C, Ruby also supports `**` (exponentiation), `<=>` (combined comparison, returning 1, 0, or -1 depending on whether the first operator is greater than, equal to, or smaller than the second), `===` (used to test equality within a `case` statement), `.eql?` (true if the receiver and argument have same value), and `.equal?` (true if the receiver and argument have the same object ID). Spelled-out logical operators, `and`, `or`, and `not` ma be used as synonyms for `&&`, `||`, and `!` for readabilty. Finally, the `..` and `...` operators build ranges, the former includes the end point and the latter excludes it.

Unlike C, the statements are not terminated by semicolons. Although, semicolons may be used as separators to write multiple statements on a single line. Indentation and white spaces are ignored.

Ruby treats each "statement" as an expression. The value of an assignment statement is simply the value of the left hand side after the assignment. One consequence of this is that assignments can be chained, e.g., `a = b = c`. Multiple values may be assigned using a multi-assignment, e.g., `x, y = 1, 2`.

Arrays may be created and referenced with `[]`, or by explicitly instantiating objects from the `Array` class. Similarly, hash tables are supported by the language as the `Hash` class or with the operator `{}`.

All names starting with uppercase letters may not be reassigned, including constant values, class names, and module names.

2.2 Control-flow Constructs

Functions or methods in Ruby are defined using the keywords `def` and `end`. For example, the following defines a function to print a string.

```
def helloWorld
  puts "Hello World"
end
```

The `if` construct in Ruby also includes an optional `elsif` clause, which eliminates the dangling-else ambiguity. For example:

```
if x < 0
  y = -x
elsif x > 0
  y = x
else
  y = 0
end
```

The `case` expression works similar to C `switch`, except that the `break` at the end of each case is implicit in Ruby. Each individual option is identified by the keyword `when` and the default case is identified by `else`. Following code is equivalent to the `if` example above.

```
x = case
    when x < 0 then -x
    when x > 0 then x
    else 0
    end
```

The example also illustrates the use of the `case` as an expression as the right hand side of an assignment.

Ruby supports a variety of looping constructs, including `while`, `for`, `until`, and `loop`. The examples in Figure 2.1 illustrate these constructs, each expressing the same computation using a different looping construct.

A convenient feature in Ruby is statement modifiers. These are optional conditions that can be attached at the end of a statement and could be one of `if`, `unless`, and `while`. For example, the conditional `break` statement in the rightmost code segment in Figure 2.1 could be written as `break if i== 10`. A C-style `do-while` loop can be written using `while` as a modifier.

The most common idiom in Ruby for writing loops, in fact, does not use any of these standard looping constructs. Instead, it makes use of Ruby blocks, which are closures that may be passed to any method. The following code performs the same computation as the code in Figure 2.1 using the `upto` method on `Fixnum` and passing a *block* to do the addition.

```
sum = 0
0.upto(9) do |i|
  sum = sum + i
end
```

The block is enclosed within `do` and `end` and arguments to the block may be specified using the vertical bars (`|`). Within the `upto` method this block may be invoked with `yield`. Note that any user-defined method may also invoke any block passed to it in this manner. The block may also be enclosed within curly braces ({ and }), instead of `do` and `end`.

Ruby provides several other convenient methods on standard classes as iterators, such as, `each` on `Array` that iterates over each element of an array, and `each_pair` for `Hash` that iterates over each key-value pair in a hash table.

```
sum = 0
i = 0;
while i < 10
  sum = sum + i
  i = i + 1
end
```

```
sum = 0
for i in 0..9
  sum = sum + i
end
```

```
sum = 0
i = 0
until i > 9
  sum = sum + i
  i = i + 1
end
```

```
sum = 0
i = 0
loop do
  sum = sum + i
  i = i + 1
  if i == 10
    break
  end
end
```

Figure 2.1: Looping constructs in Ruby.

```
# Reopen Fixnum class and define method factorial
class Fixnum
  def factorial
    (self == 0) ? 1 : (self * (self - 1).factorial)
  end
end
# Call factorial on the range 1..5 using map
(1..5).map { |n| n.factorial }  => [1, 2, 6, 24, 120]
# Define Factorial module to use on floats as well
module Factorial
  def factorial
    ...   # same code from above
  end
end
# Reopen Float to include the Factorial module
class Float
  include Factorial
end
# Make sure Float changed:
5.0.factorial  => 120.0
```

Figure 2.2: Example program in Ruby illustrating some of its object-oriented features.

2.3 Object-oriented Features

Ruby is a purely object-oriented language. All objects are derived from the base class `Object`. Ruby's "open classes" allow classes to be reopened at any time and methods to be added, redefined, or removed.

Ruby supports single inheritance, using the syntax `A < B` to indicate that class `A` extends class `B`. All instance variables start with `@` and all class variables start with `@@`. Methods and instance variables may be declared public (default), private, or protected. There are some subtle differences between Ruby's interpretation of private and protected from that of C++. In Ruby (unlike C++) private methods may be accessed from subclasses. However, private methods of another instance are not accessible within the class. In order to make such methods accessible to other instances of the class or its subclasses the method must be declared protected.

Ruby modules provide a way to work around the single-inheritance constraint (similar to Java interfaces). Including a module in a class makes all the methods exported from the module available within that class as if they are part of that class.

Figure 2.2 shows a complete example using some of Ruby's object-oriented features. The example defines the `factorial` function on `Fixnum` class by reopening it, then creates a `Factorial` module and uses it to define `factorial` for `Float` as well. In order to maintain the fully object-oriented paradigm, any code that is apparently outside any class definition gets added to the `Object` class.

2.4 Advanced Features

Ruby includes several advanced features, such as meta-classes, catching the missing method exception, and generating code on the fly. We refer the readers to other references for discussions of these features [7, 20, 16].

Chapter 3

Design

An appropriately designed tool can bring dramatic productivity gains to the task of compiler development. Chapter 8 describes several such tools. `RubyWrite` focuses on rewriting abstract syntax trees (AST) using pattern matching and transformation. It also provides other useful features and tools. `RubyWrite`'s nature as an embedded DSL means new features can be added by using the host language natively, allowing us to test and later incorporate features into the core library. It also means a growing list of libraries can be leveraged in the language, such as those for parsing, graph manipulation, and code generation. Further, Ruby provides a mechanism for writing wrappers for external C or C++ libraries, allowing us to utilize other code analysis and optimization libraries. The mechanism can also be used to rewrite any parts of `RubyWrite` in C that become a performance bottleneck, although we so far in our compilers perform well.

Embedding our DSL into a higher level-language addresses one of the challenges encountered using Stratego/XT [2]. In more complex analyses we wanted to use a control-flow graph (CFG) representation. Originally, we attempted to extended Stratego to use the Boost Graph library by creating a C wrapper for the library that could be called from Stratego. This approach proved difficult to implement and maintain, and was ultimately abandoned. Our ParaM project also needed a bridge to the C++ library provided by Octave, the open source MATLAB implementation. The Ruby Graph Library provides the same functionality as the Boost Graph Library in Ruby, and a simple C++ bridge to Octave was straightforward to implement.

Since the primary mechanism in `RubyWrite` is pattern matching and replacement it is suited to transforming code at relatively high levels where a graph-based intermediate representation is worthwhile, rather than linear intermediate representation. This is also similar to Stratego, but we decided to make pattern matching separate from pattern building. In Stratego, strategies perform matching and building as a single unit of work. This sometimes lead to misleading error messages when used in conjunction with the deterministic choice operator, as the stack trace seemed to indicate the pattern never matched, when actually the build had failed. `RubyWrite` separates these processes and makes it easier to pinpoint where things went wrong.

3.1 Features

`RubyWrite` is designed to agree with the idioms of its host language. For example, it encourages conformance to Ruby's convention of using "!" and "?" in method names to reflect side-effects and Boolean return values, respectively. Ruby's meta-programming features are leveraged at multiple levels, including automatic encapsulation of user-defined methods in wrapper code,

code-factoring based on functionality rather than classes, and unit testing. `RubyWrite` provides uniform semantics for almost all the operations, irrespective of whether they are user-defined or `RubyWrite` primitives[1].

Implemented as a Ruby module, `RubyWrite` lives in an isolated name space. Making `RubyWrite` available as a module, instead of a class, frees users to incorporate its functionality as a "mixin" within any user class and at any point within the class hierarchy.

`RubyWrite` lets users specify sequences of rewriting classes to implement arbitrary phase order. A compiler using `RubyWrite` may choose to construct the ordering dynamically based on the input. `RubyWrite` comes with several convenience methods to traverse the AST in different ways.

`RubyWrite` is also an easily extended and flexible framework. When one of our compiler projects needed source file and line number annotations we added a simple extension to support generic annotations. `RubyWrite` could then be used to write a tool to analyze the size of built procedures as feedback for our MATLAB projects. As another example, we easily added an interface to hardware counters as an extension.

`RubyWrite` consists of three main components:

1. *Building ASTs* The Ruby `Symbol` class is extended to support the `[]` operator, which enables creation of recursively specified trees with node types `Node`. ASTs built using this are the core representation that are used by the other components of `RubyWrite`.

2. *Unparsing* A submodule within `RubyWrite` provides a convenient and succinct way to convert the ASTs into concrete syntax, i.e., unparse the ASTs. It is implemented as a standalone embedded DSL, called ShadowBoxing. Designed after the "Generic Pretty Printer" based on the BOX language [4], ShadowBoxing can be used to naturally specify indentation and bracketing for programming constructs.

3. *Tree rewriting* Features to rewrite and traverse ASTs constitute the biggest component of `RubyWrite`. Some of these are motivated by Stratego/XT [2].

Chapter 4 describes each of the components in detail.

3.2 Syntax

Particular attention is paid to ensure `RubyWrite` fits well into its host language. For instance, although AST transformation methods are defined with a special syntax, they can be used like any other Ruby method. These methods can also contain arbitrary Ruby code. The user is also free to use Ruby variables to store references to subtrees. Avoiding special calling conventions helps programmers use the DSL naturally and allows for ease of tasks such as unit testing.

The parsing and unparsing features of `RubyWrite` may be used without invoking the rewriting module. There is no restriction on the overall organization of the code. However, any code needing access to re-writing features must be inside a class that include the `RubyWrite` module.

The core primitives in tree matching and rewriting are `match?`, `build`, `try`, and `call` of user-defined rewriting method. Several other convenience primitives are provided, but those can be constructed (and, are constructed) from the core primitives. `match?` matches a supplied AST node with a specified pattern, which may include identifiers to be bound to portions of the tree under the matched node. The `build` primitive instantiates an AST node, along with

[1]The only exception is `match`, since it must alter the matching environment of its caller.

Matching, with block

$$\frac{\mathcal{E} \vdash n.\texttt{match?} \Rightarrow n(\mathcal{E}') \qquad \mathcal{E}' \vdash n.\lambda \Rightarrow n'(\mathcal{E}'')}{\mathcal{E}, n.\lambda \vdash n.\texttt{match?} \Rightarrow n'(\mathcal{E}'')}$$

Building, with block

$$\frac{\mathcal{E} \vdash n.\lambda \Rightarrow n''(\mathcal{E}') \qquad \mathcal{E}' \vdash n.\texttt{build} \Rightarrow n'(\mathcal{E}')}{\mathcal{E}, n.\lambda \vdash n.\texttt{build} \Rightarrow n'(\mathcal{E}')}$$

Try

$$\frac{\mathcal{E} \vdash n.\lambda \Rightarrow n'(\mathcal{E}')}{\mathcal{E}, n.\lambda \vdash \texttt{try} \Rightarrow \texttt{n'}(\mathcal{E}')} \qquad \frac{\mathcal{E} \vdash n.\lambda \Rightarrow\uparrow(\mathcal{E}')}{\mathcal{E}, n.\lambda \vdash \texttt{try} \Rightarrow \texttt{n}(\mathcal{E})}$$

Method / Rewriter Call

$$\frac{\mathcal{E} \vdash n.\phi \Rightarrow n'(\mathcal{E}')}{\mathcal{E}, n.\phi \vdash \texttt{call} \Rightarrow n'(\mathcal{E})}$$

Figure 3.1: Semantics for basic `RubyWrite` primitives

its children (i.e., subtrees), possibly using identifiers that were matched before with the `match?` primitive. `try` provides a safe way to attempt a tree transformation and returns failure if the rewriting fails. Finally, `call` provides a way to invoke a user-defined tree-rewriting method on a supplied AST node. These are discussed in detail in Chapter 4.

3.3 Semantics

The behavior of `RubyWrite` core primitives is summarized in Figure 3.1. $\mathcal{E}$ is the matching environment binding Symbol names to AST nodes. λ is a code block passed as a Ruby block, ϕ is a user-defined `RubyWrite` method or rewriter. We use the notation $n.f$ to indicate method f is invoked on the AST node n. The rest of the notations are standard. Notice these semantics are only for the matching environment maintained by `RubyWrite` and do not account for Ruby's built-in variable environment.

Exceptions raised in a code block passed to `match` or `build` are raised to the caller. Any uncaught exceptions inside a `RubyWrite` method or rewriter appear as exceptions in the calling context. We omit the semantic description of failure modes for brevity, but they are easily derived.

Chapter 4

Using RubyWrite

RubyWrite consists of three major components:

1. AST builder: This component is used by including (with require) rubywrite. It makes a Node class available, which represts the AST nodes, and adds the operator [] within the Symbol class as a convenient mechanism to instantiate AST nodes.

2. Unparser: This component is used by including shadow_boxing. It makes the class ShadowBoxing available for writing unparsers.

3. Tree rewriting and traversal: This component is used by including rubywrite. It provides several pattern matching, tree traversal, and tree rewriting methods available in the form of a Ruby module RubyWrite. The features of this component are divided into three main submodules, Basic, PrettyPrint, and Traversals, which may be directly included, instead of rubywrite, if only a subset of functionality is needed.

The rest of this chapter describes each of these features in greater detail, with examples.

4.1 Representing ASTs

RubyWrite defines a Node class to represent ASTs internally. An AST node is a String, an Array, or a Node. A Node consists of a label, represented as a Symbol, and an arbitrary number of children. Internal AST nodes are either Node or Array objects. Leaf nodes are String objects.

Thus, 'some string', ['a','b',['c','d']], :Var['x'] are examples of syntactically valid representations of ASTs. This representation of trees is adequate to represent any AST. RubyWrite provides a convenience method [] on Symbol to construct ASTs. For example, the following is a simple "Hello World" program in C and one possible AST for it, represented using the syntax described above.

```
int main () { printf("Hello World\n"); }

:Function['int', 'main', :Args[[]],
          :Body[:StmtList[[:FunctionCall['printf',:Args[['"Hello World\n"']]]]]]]
```

Note that the [] operator does not enforce the types of its arguments. Therefore, it is up to the programmer to make sure that the ASTs are well-formed. It is expected that the ASTs will be built using an external tool, such as racc for creating bottom-up LR parsers. RubyWrite does not provide any parsing support other than primitives to build trees.

4.2 Unparsing using ShadowBoxing

`RubyWrite` supports unparsing[1] of an arbitrary AST, i.e., translating an AST into the concrete syntax of the target language, through the class `ShadowBoxing`. The syntax of the unparser defines a mini-DSL of its own, which is also called ShadowBoxing. This DSL is inspired by the BOX language that the *Generic Pretty Printer* (GPP) [4] also uses.

ShadowBoxing is enabled by including `shadow_boxing.rb` with `require 'shadow_boxing'`. An unparser is created by instantiating the `ShadowBoxing` class and passing the `new` method a block containing a sequence of "rules", one for each AST node type. Each such `rule` takes a node name and a block that specifies the printing rule. The printing rule consists of an expression that specifies the formatting of the node. Figure 4.1 shows a portion of an unparser for MATLAB.

When a rule is invoked, its block is called with the node's children passed as arguments. For example, the `:Function` rule expects four arguments. The printing format is specified with either `h` or `v` "boxes". The `h` box prints elements horizontally, while the `v` box prints elements vertically. Formatting primitives take a hash table of options followed by a list of arguments to format. The hash table specifies spacing options. Arguments can be `String`s or AST nodes. `String`s are reproduced in the output and AST nodes are processed recursively. Rules either

[1]We consistently use the term *unparsing* when we mean generating concrete syntax from abstract syntax, and *pretty-printing* when we mean printing out the AST in a human-readable form.

```
require 'shadow_boxing'

class UnparseMATLAB
  def unparse(node)
    boxer = ShadowBoxing.new do
      rule :Var do |var| var end
      rule :Const do |val| val end
      rule :Assign do |lhs, op, rhs|
        h({:hs => 1}, lhs, op, rhs)
      end
      rule :While do |test, body|
        v({}, v({:is => 2},
                h({:hs => 1}, "while", test),
                body), "end")
      end
      rule :Function do |retvals, name, args, body|
        v({:is => 2},
          h({}, "function ", "[",
            h_star({}, ", ", *retvals.children), "] = ", name,
                   "(", h_star({}, ", ", *args.children), ")"),
                   body)
      end
      ...
    end
    boxer.unparse_node(node).to_s
  end
end
```

Figure 4.1: A snippet of an unparser for MATLAB

return a value or call formatting primitives. In either case the final output is expected to be a `String`.

Rules can be as simple as the one for `:Var`, returning a variable name as a `String`, or express more complex layouts using primitives. For instance, the `:Assign` rule uses `h` with the `:hs => 1` option to specify horizontal layout with one-character spacing between the arguments. The `:is` option specifies the amount of indentation. The `:Function` rule demonstrates another formatting primitive, `h_star`. The `h_star` primitive uses a supplied *separator* to demarcate its arguments. The argument `*retvals.children` expands the `children` array into all its elements, thus passing all the MATLAB function's formal parameter names as distinct arguments to `h_star`. Note that `v_star` is redundant since `h_star` can also accept newline (`'\n'`) separator.

So far all we have done is create a `ShadowBoxing` object. Each instance of `ShadowBoxing` supports the `unparse_node` method. An AST is unparsed by passing it to `unparse_node`, which returns an unparsed object. To be able to print the unparsed object the method `to_s` converts it to a string.

The `unparse_node` accepts an optional Boolean argument, which enables support for arbitrary user-defined attributes associated with each AST node when set to true. These attributes can be accessed using the `attributes` method on `Node` and may be unparsed separately. This feature is useful in emitting information that is not strictly part of the AST, such as comments or back-end compiler directives.

4.3 Tree Rewriting and Traversal

The biggest component of `RubyWrite` consists of its tree rewriting and traversal functions. These features are accessed by defining a Ruby class and including the module `RubyWrite` in it, which becomes available once `rubywrite.rb` is included using `require 'rubywrite'`. However, specific features may be enabled by including just one of the required submodules, `Basic`, `Traversals`, and `PrettyPrint`.

4.3.1 Sub-module Basic

The `match?` and `build` primitives

The two most fundamental primitives for tree rewriting are `match?` and `build`. As the `?` at the end of `match?` suggests, the primitive returns a Boolean value—`true` if the match succeeds and `false`, otherwise. It takes two arguments, a pattern and the AST node to match. The pattern is specified using a syntax similar to that used for writing the AST. The difference is that pattern leaf nodes can be `Symbol` objects. The following example shows a simple application of `match?`.

```
if match? :BinOp[:a, :op, :b], node
   commuted = :BinOp[lookup(:a),lookup(:op),lookup(:b)]
end
```

The `match?` expression specifies the pattern consisting of a node name `:BinOp` with three children, and attempts to match it against an AST node, `node`. If `node` is indeed of this type then the symbols `:a`, `:op`, and `:b` are bound to the three children. The bound subtrees can then be accessed using the `lookup` method.

As a shortcut, a new AST node may be instantiated using previously bound symbols without having to write `lookup` for each symbol, using the `build` primitive. All free symbols in the expression passed to `build` are first replaced by the trees they are bound to and the resulting node is then instantiated. Thus, the following code is equivalent to the above.

```
if match? :BinOp[:a, :op, :b], node
   commuted = build :BinOp[:a, :op, :b]
end
```

Note that the pattern may be specified as a nested tree. In general, `RubyWrite` matches the pattern tree recursively, using the following rules.

1. A `String` in pattern matches an identical `String` in AST.
2. A `Node` in pattern matches `Node` in AST with the same label and an identical number of children, where the match succeeds recursively on each child.
3. An `Array` in pattern matches an `Array` in AST with the same number of elements, where the match succeeds recursively on each element.
4. A `Symbol` in pattern matches any subtree in AST when it is first encountered. If the `Symbol` is encountered again, the subtree at each point the `Symbol` is used must match. The `Symbol :_` creates no binding.

A successful `match?` returns `true` and the matching environment is updated with symbols specified in pattern bound to the corresponding subtrees in AST. On failure no new bindings are created and `match?` returns `false`. Bindings can be looked up using the `lookup` method. New trees can be created using the `[]` method on `Symbol`, as indicated above. Alternatively, the `build` method automatically replaces occurrences of each `Symbol` at the leaf level by the corresponding tree.

Both `match?` and `build` also take an optional block. The block is executed *after* `match?` succeeds, or *before* `build` begins. This leads to the following idiom for the above example.

```
match?(:BinOp[:a,:op,:b],node) {
  commuted = build :BinOp[:b,:op,:a]
}
```

It is important to note that `match?` and `build`, along with their optional blocks, modify the match environment in the surrounding method. In contrast, other methods get a new set of bindings that is discarded when the method returns. Standard Ruby scoping rules apply to the blocks, which are closures. If a new variable is defined in a block, it is *not* visible outside the block. Thus, if the variable `commuted` is defined for the first time in the block passed to `match?` it is invisible outside the block.

Defining `RubyWrite` methods

In order to allow `RubyWrite` to set up the pattern matching environment correctly, a special syntax is used to define methods that will use tree rewriting and traversal primitives. The methods are defined with `define_rw_method`. These methods thread an environment for storing the results of pattern matching. As a convenience, the `RubyWrite` module provides a `run` method that automatically instantiates the class and calls the `main` method to perform the transformation. The code in Figure 4.2 defines a class `Example`, and uses the `run` method to process the AST `program`.

Methods defined this way do not need any special calling syntax and are called similar to standard Ruby methods. Moreover, these may be called from standard Ruby methods defined

```
class Example
  include RubyWrite
  define_rw_method :main do |n|
    ...
  end
end
Example.run program
```

Figure 4.2: Minimal `RubyWrite` program

using `def` and may also call other standard methods—for instance, helper methods that do not themselves use any tree rewriting or traversal and hence may be defined as standard Ruby methods.

Defining `RubyWrite` rewriters

While `match?` and `build` primitives are sufficient, they are not the most convenient when trying to match one of a large number of node types. In those cases, one might be forced to write a long sequence of `if-elsif` expression, which would not only be awkward but also highly inefficient. `RubyWrite` provides a highly efficient alternative in the form of *rewriters*.

In addition to methods, a class can also define rewriters. Rewriters are special methods consisting of a set of *rewrite rules*. The block associated with a rewriter is invoked on an AST node when the pattern specified in the left hand side matches. Additional arguments can also be passed after the node argument. An example in Figure. 4.3 illustrates the syntax. The pattern uses the same syntax as `match?` and when successfully matched, `Symbols` are bound to subtrees. Rewriters can call the method being defined recursively.

Patterns specified for rewriting are assumed to be non-overlapping, although that is not enforced. If overlapping patterns are used, the behavior of the rewriter is unspecified. If a default is specified `RubyWrite` invokes its block if the given AST node fails to match any patterns. When no default is specified, an exception is raised if no pattern matches.

Each rewriter must return an AST node, which replaces the node passed to it. Even though these are called rewriters they do not necessarily have to rewrite the tree. For example, each

```
class Example
  include RubyWrite
  define_rw_rewriter :xform_statements do
    rewrite :IfElse[:cond,:then,:else] do |n|
       # handle if-else statements
    end
    rewrite :While[:cond, :body] do |n|
       # handle while statements
    end
    ...
    default do |n|
      # optionally, specify a default action
    end
  end
end
```

Figure 4.3: An example of using a rewriter

rewriter could process the node it is passed (`n` in the example of Figure 4.3) and simply return that node. In this way the rewriters can also serve as an efficient idiom to traverse the AST when different actions are needed for different types of subtree patterns.

The traversal through the matched subtrees is explicit, unlike the unparser that does the recursive traversal automatically. Thus, if the body of the while-loop needs to be processed a method or rewriter must be called explicitly on the body. The syntax for calling a rewriter is identical to calling a standard Ruby method.

Other basic primitives

The `try` primitive executes code conditionally. If the code returns `nil` or `false`, any changes it made to the match environment are rolled back. For example, in the following code,

```
try { match? :if[:a, :binop, :b], node }
```

the symbols `:a`, `:binop`, and `:b` are unbound if `match?` fails.

Bindings can be created explicitly by passing `set!` a `Symbol` and an AST node. So, we could ensure that the symbols are bound in the above example by adding the following before the above code.

```
set! :a, :Empty[]
set! :b, :Empty[]
set! :binop, :Empty[]
```

Finally, `match` (not ending in `?`) behaves similar to `match?`, except it raises an exception if the match fails.

4.3.2 Sub-module Traversals

It is often convenient to encode a compiler pass as a transformer operating on specific node types, or performing related actions on each node type. This can be achieved with rewriters as we described earlier. However, a compiler pass might be a simple analysis or information gathering phase, where a rewriter is somewhat unwieldy. `RubyWrite` provides two other mechanisms to traverse an AST.

Simple traversal methods are defined using `define_rw_postorder` or `define_rw_preorder`. The definition pairs node types and Ruby blocks, specifying the action for each, along with an optional default action. If a block returns `nil` or `false`, traversal stops for that subtree.

```
names = NameTable.new
define_rw_preorder :gather_variable_names do
  upon :Assign do |n|
    names.add n[0] # the 0th child is the LHS
    nil            # stop further descent
  end
  upon_default do
    true           # continue the descent
  end
end
gather_variable_names ast
```

Figure 4.4: Code to gather all variables

```
names = NameTable.new
alltd? ast do |n|
  if match? :Assign[:x, :v], n { names.add lookup(:x) }
    true
  else
    false
  end
end
```

Figure 4.5: Variable gathering rewritten with `alltd?` traversal

Figure 4.4 is a simple example to gather all assigned variable names. The currently matched node is passed the first argument to the block. Notice that we take advantage of the fact that blocks are closures to use the local variable `names`. `RubyWrite` also provides `define_rw_rpostorder` and `define_rw_rpreorder` to perform reverse order traversals.

`RubyWrite` has several predefined tree traversal methods motivated by Stratego [3]. In each case, the traversal method is passed the AST to be traversed and a block of code. The code is executed on each node as it is traversed. If the code returns `nil` or `false` it is assumed to *fail* on that node. Otherwise, it is assumed to *succeed*. Each traversal method comes in two flavors one ending in `!` and one ending in `?`. The `!` method modifies the AST, while the `?` method only returns a Boolean status indicating success or failure, without modifying the AST.

- Method `all!` applies the block, one level deep, to each child of the AST node. If any application fails it raises an exception.
- Method `one!` applies the block to each child until it finds one that succeeds. On success it returns the child, otherwise it raises an exception.
- The `topdown!` method applies the block to the node *and* recursively to its children.
- The `bottomup!` traversal applies the block to each child and then to the node.
- The `alltd!` traversal is similar to `topdown!` except that it only recursively traverses the children of a node if the block fails on the node.

The versions ending in `?` behave similarly, but do not modify the matching environment, and return only a Boolean value.

The code in Figure 4.4 to gather all assigned variables can also be written using traversal methods, as in Figure 4.5. We use `alltd?`, since there is no need to modify the matching environment, although its return value is ignored.

4.3.3 Sub-module PrettyPrint

ASTs can be dumped, or pretty-printed, using the `PrettyPrint` sub-module. The pretty-printer uses `ShadowBoxing` internally. An AST is pretty-printed as a tree, without conversion to concrete syntax—`RubyWrite` cannot, in general, convert an AST to concrete syntax since it has no knowledge of the source language.

4.4 Concrete Syntax

ASTs can be instantiated directly in concrete syntax. A special type of AST node indicates it contains concrete syntax, allowing a mix of abstract and concrete syntax. Using Ruby's interpolated strings, `RubyWrite` gets this feature mostly for free. The following example swaps the arguments of a binary operator.

```
if match? :BinOp[:op1, :binop, :op2], node
   commuted = Node.concrete(
     "#{pp(:op2)} #{pp(:binop)} #{pp(:op1)}")
end
```

This can be used to compactly specify large chunks of code in concrete syntax, where specifying the AST might be unwieldy. Here, the `pp` method looks up `Symbol` and returns the AST bound to it pretty-printed into a `String`. A subtree containing concrete syntax is not converted back to abstract syntax unless necessary, e.g., when trying to match it against a pattern. To support this the class supplies a parser `RubyWrite` can use to parse `String`s into ASTs. The compiler writer is responsible for ensuring portions of code represented concretely are pretty-printed correctly by supplying an appropriate rule to `ShadowBoxing`. A simple, albeit somewhat inefficient, way to handle this is to first parse concretely specified subtrees into abstract syntax before pretty printing the AST.

Currently, patterns cannot be specified using concrete syntax. `RubyWrite` is independent of the source language and parser employed by the compiler. Our projects use `RubyWrite` with multiple languages and different parsing methods. Although, we have no immediate need for this, we plan to investigate the benefits of providing an optional feature to interface with a Ruby parsing library, such as `Racc` or `TreeTop`.

4.5 Composing Compilation Phases

Compiler phases are composed by chaining calls to them. For example, the following applies an expression flattener pass (`Flatten`), followed by a constant propagation pass (`ConstProp`), and a dead code elimination pass (`DCE`).

```
output = DCE.run(ConstProp.run(Flatten.run(input)))
```

This works when the sequence of phases to be applied is fixed. `RubyWrite` also provides a more readable syntax through the `xform` method.

```
output = input.xform Flatten, ConstProp, DCE
```

Transformations are listed as classes or objects supporting a `run` method. Arguments can also be an arbitrary array of the phases to be applied to transform `input` to `output`. This enables phases to be selected and ordered dynamically, based on the program being compiled.

Chapter 5

Data Flow Analysis with `RubyWrite`

In order to perform data flow analysis directly on an AST, the action to perform on each AST node type depends on the language being compiled. Transfer functions and meet operations also depend on the specific data flow analysis problem being solved. `RubyWrite` can be used to write a language-specific data flow analysis framework that parameterizes problem-specific aspects. Figure 5.1 shows such a generic framework for a hypothetical language consisting of assignments

```
def fixed_point val
  if block_given?
    new_val = yield val
    while (new_val != val) new_val = yield val; end
  end
  val
end

class GenericDFA
  define_rw_preorder :forward_data_flow do
    upon :If do |n, set|
      set = analyze n[0], set
      analyze(n[1], set) | analyze(n[2], set)
    end
    upon :Assign do |n, set|
      analyze n[1], set
      update_set n[0], n[1]
    end
    upon :While do |n, set|
      set = fixed_point(set) { |set| analyze n[1], set }
      analyze n[0], set
    end
    default { |n, set| analyze n, set }
  end

  define_rw_rpostorder :reverse_data_flow do
    ... # similar to code above
  end
end
```

Figure 5.1: A generic data flow analyzer that may serve as a base class

and two types of compound statements—`if-else` statements and `while` loops. It uses the preorder traversal and specifies actions for each node type. A helper function computes fixed points for the loops. The `analyze` method encapsulates problem-specific behavior, allowing the programmer to specify the action taken on update. The `analyze` method can be implemented as a rewriter (for example) and can be recursive. The meet operation ("`|`") can be defined on the class of `set` as appropriate for the problem.

In general, a data flow framework requires meet and equality operations to compute fixed points. Fortunately, built-in Ruby data structures support a rich set of operations, including comparison, merging, etc. As a result, building a basic data flow analysis framework is straightforward. In Figure 5.1, the AST is used directly, instead of constructing a control flow graph.

Data flow analysis can also be implemented by setting up global data flow equations and solving them iteratively, using a control flow graph. Graphs can be built using graph libraries,

```
class ConstantProp
  include RubyWrite
  def main n; ast, e = cp(n, {}); ast; end
  define_rw_rewriter :cp do
    rewrite :Assign[:Var[:lhs],:rhs] do |node, e|
      rhs, e = cp lookup(:rhs), e
      if is_constant? rhs then e[lookup(:lhs)] = rhs
      else e.delete lookup(:lhs) end
      [build(:Assign[:Var[:lhs],rhs]), e]
    end
    rewrite :Var[:v] do |node, e|
      [((e[lookup(:v)]) ?
         e[lookup(:v)] : build :Var[:v]), e]
    end
    rewrite :If[:t,:c,:a] do |node, e|
      t, e = cp lookup(:t), e
      (c, ce), (a, ae) = [cp(lookup(:c), e.copy),
                          cp(lookup(:a), e.copy)]
      [:If[t,c,a], ce.meet(ae)]
    end
    rewrite :While[:t,:b] do |node, e|
      t, b = lookup(:t), lookup(:b)
      tn, e = cp t, e
      e = fixed_point(e) do
        bn, be = cp b, e.copy
        tn, e = cp t, e.meet(be)
        e
      end
      t, en = cp t, e
      b, en = cp b, e
      [:While[t,b], e]
    end
    ...  # code for handling default cases
  end
end
```

Figure 5.2: Constant propagation using rewriters.

```
class ConstantProp
  include RubyWrite
  def main n
    graph = cp(build_graph(n), {})
    unbuild_graph(n)
  end

  define_rw_dataflow :cp do
    transfer_function do |node, e|
      if match? :Assign[:Var[:lhs],:rhs], node
        if is_constant? lookup(:rhs) then
          e[lookup(:lhs)] = rhs
        else
          e.delete lookup(:lhs)
        end
        [:Assign[:Var[:lhs],:rhs], e]
      elsif match? :Var[:v], node
        [((e[lookup(:v)]) ?
           e[lookup(:v)] : build :Var[:v]), e]
      else
        [node, e]
      end
    end
    define_rw_method :build_graph do
      ... # code to build the RGL based graph
    end
    define_rw_method :unbuild_graph do
      ... # code to rebuild node for next pass
    end
  end
end
```

Figure 5.3: Constant propagation using data flow.

such as the Ruby Graph Library (RGL). One advantage of a control-flow graph approach is that the framework can be language independent, since control flow is captured in the graph. The framework can be instantiated with problem-specific aspects of set comparison, initial values, and meet operations over the set of values being computed. `RubyWrite` provides such a framework as a sub-module assuming the control flow graph uses an RGL representation and each node contains a single statement or expression. In this way, users can decide to work directly with the AST using the template in Fig 5.1 to write a language-specific, but problem-independent, data flow analysis framework. Alternatively, they can use the `DFA` sub-module of `RubyWrite` that works with a control flow graph built using RGL.

Figures 5.2 and 5.3 illustrate the data flow module with two examples of constant propagation. Both implementations use an environment (“e”) providing `meet` and `copy` operations. We add these methods to the existing `Hash` class. Both implementations also rely on an `is_constant?` method to determine when an expression is a constant.

The implementation in Figure 5.2 uses rewriters and runs directly over the AST of the program. This approach skips building the graph, but is still a relatively compact implementation. Meet and fixed point operations for `if` and `while` are handled explicitly. We use the `fixed_point` method defined in Figure 5.1 for `while`.

Figure 5.3 is the same process written using the data flow library. In this version, the compiler writer provides methods for building the graph representation in RGL format. `RubyWrite` otherwise cannot determine where edges need to be added to the control flow graph. As mentioned earlier each node contains a single statement or expression. Once this is done the job of specifying the constant propagation is simpler, since the data flow module performs meet and fixed point operations on the graph. The compiler writer provides transfer functions to update the environment and replace variable references with constants when appropriate.

Chapter 6

Implementation

RubyWrite is written in Ruby. If performance becomes a bottleneck and hotspots are identified, portions of the implementation can be moved to C, as needed.

6.1 Implementing as a Module

Implementing RubyWrite as a module allows it to be used as a mixin in any class. Reopening classes lets us organize code into sub-modules based on functionality, rather than class hierarchy. User classes are then free, and even encouraged, to include only those sub-modules it needs. Multiply included modules are detected by Ruby to ensure they are only included once.

6.2 Creating Methods

RubyWrite requires special syntax for creating methods that will use RubyWrite primitives because these methods are surrounded by code to set up and tear down an environment used in matching. We use the executable class body along with Ruby's meta-programming support to dynamically add regular methods to the class. The `define_rw_*` primitives are really calls to methods defined in the `RubyWrite::Basic` module. The `define_rw_*` primitives then create normal Ruby methods that thread a matching environment through the body of the method. This is important because it allows RubyWrite methods to be used as normal methods, for instance in test cases, while still providing the extra support needed by the `match?`, `build`, and related primitives. Figure 6.1 shows the code for the RubyWrite primitive `define_rw_method`.

The method definition begins by creating a new binding environment, and saving the current one. The original bindings are restored in the `ensure` block after the method finishes or raises an exception.

There is one complication: since `define_rw_*` methods are called during class definition they need to be "class" methods instead of instance methods. Unfortunately, when a module is included in a class only instance methods are included, class methods are not. This means the `define_rw_method` cannot simply be a part of the RubyWrite module. We define these methods as instance methods of a sub-module called `ClassMethods`, and then override the `included` method of the module. Ruby invokes the `included` method on a module when a it is mixed into a class. The `included` method *extends* the user class, making the methods in the `ClassMethods` class available in the user class. When the class object is extended methods are added to the list of method definitions in its meta-class, creating class methods. This is how

```
def define_rw_method name, &blk
  define_method name do |*args|
    begin
      saved_env = @bindings
      @bindings = Environment.new
      instance_exec *args, &blk
    ensure
      @bindings = saved_env
    end
  end
end
```

```
module Base
  module ClassMethods
    def define_rw_method name, &blk
      ...
    end
  end
  # constructor for Base module
  def self.included user
    user.extend ClassMethods
  end
end
```

Figure 6.1: Implementation of method definition within the `RubyWrite` module

class methods are normally created in Ruby, although some syntactic sugar masks this fact. The code appears in Figure 6.1.

6.3 Creating Rewriters

Similar to the other `RubyWrite` methods, rewriters are normal Ruby methods allowing them to leverage the existing exception handling framework and also providing support for writing unit tests. In addition to the complication of creating a method, rewriters also pose the problem of matching patterns efficiently. One solution would be to use a sequence of `if-then-else` statements. Hashing on the labels of the root nodes of all the patterns is a better solution, but is still suboptimal since multiple patterns might have identical root labels. Instead, we use a trie, to index on the complete pattern. This has the advantage that some overlapping patterns can be detected at the time of constructing the trie. The trie is created at class definition time and reused for each instance of that class. Trie lookup is also asymptotically faster then a sequential pattern search, with a lookup time of O(m), where m is the length of the key. helping to provide good performance when matching.

As the example in Chapter 5 showed, rewriters can be passed arbitrary arguments. These are in turn passed to the blocks when the associated pattern matches the current AST node. The programmer is responsible for ensuring that arguments to each block are consistent across rewriter rules.

Another complication arises in executing the blocks. Inside the block passed to `define_rw_rewriter` the call to `rewrite` is in the context of the surrounding code. We would like to make this call in the context of the `Rules` class that encapsulates the trie and other data. We use Ruby's

```
def alltd! (node, &b)
  if (t = try(node, &b))
    t
  else
    all!(node) |n| alltd!(n,&b)
  end
end
```

Figure 6.2: Implementing `alltd!`

```
    # traversal as a method call, node is an argument
    all! node do |n|
      ... # action to take on the node
    end

    # equivalent call, using the object-oriented method
    node.all! do |n|
      ... # action to take on the node
    end
```

Figure 6.3: Different styles of traversal calls

`instance_exec` to perform this task. It allows an object to call a block in the object's context, so the block from the definition point is treated as though it was defined in the context of the `Rules` object. Unfortunately, this means the block passed to `rewrite` is executed in the context of `Rules` as well. We use `instance_exec` again to execute the code back in the original context.

6.4 Handling Traversals

Tree traversal methods defined with `define_rw_preorder` and other traversal primitives are implemented similar to rewriters. Since traversal actions are distinguished only by the node label, a simple `Hash`-based scheme to find the code to run is efficient and effective. Traversal methods lookup the associated block in the `Hash` table. If no rule matches, the optional default block is used.

Other recursive traversals are relatively simple to implement. The `all!` and `one!` traversals are simple non-recursive methods. The `topdown!` and `bottomup!` traversals use recursion on the tree. Finally, `alltd!` uses `try` to ensure side-effects on bindings can be rolled back upon failure. These traversal methods are also defined on the `Node`, `Array`, and `String` classes in order to apply them using a more object-oriented syntax for the traversals. For example, the code examples in Figure 6.3 are equivalent.

6.5 Handling Compilation Phases

Since classes are objects in Ruby, they can be passed as arguments and assigned to variables. This simplifies the implementation of `xform` method, as seen in Figure 6.4.

```
class Node
  def xform *args
    n = self
    args.each { |c|
      n = (c.instance_of?(Class)) ? c.run(n) : c.main(n)
    }
    n
  end
end
```

Figure 6.4: Implementing support for compilation phases

6.6 Pretty-printing support

The `ShadowBoxing` library, used to support pretty-printing, is similar to the traversal libraries. Rather then sending the node along as an argument though, the children of the node are sent. This uses Ruby's `*` calling convention that allows an array to be sent as an argument list. Pretty-printing is a two step process. In the first step, the AST is transformed into a set of `h` and `v` boxes, with `String` leaves. These are processed recursively into the final printed version. The current indentation level is tracked through the process ensuring spacing remains consistent without special attention from the programmer.

Chapter 7

RubyWrite in Practice

`RubyWrite` has been used in three different compiler research projects and as a teaching aid in two different graduate-level compilers courses. This chapter discusses the role of `RubyWrite` in one of the compiler projects, called ParaM, and in teaching.

7.1 ParaM

ParaM is a project to compile MATLAB to C. It involves type inference, code specialization, and translation to C in addition to several other standard data flow analyses such as constant propagation, dead code elimination and copy propagation. Matlab code is parsed using the Octave parser [5] and the output converted to `RubyWrite` representation. Type inference and data flow analyses are implemented using `RubyWrite` with techniques described in this paper. A dependence analyzer using the Ruby Graph Library for building and manipulating control flow graphs and program dependence graphs [14] is also available, along with a translation into and out of single-static assignment form. ParaM uses `ShadowBoxing` to pretty-print the translated code to MATLAB or C, as appropriate.

Figure 7.1 shows the overall architecture of the compiler. `RubyWrite` rewriters are used extensively in ParaM. For instance, one of the early phases in the compiler flattens all the expressions to their simplest form. The flattener inserts temporary variables in the place of nested expression to simplify the task of later steps in the compiler. Flattening processes each statement and expression type differently. Rewriters let this be expressed cleanly, focusing only on the actual transformation with a small amount of surrounding code. This makes the

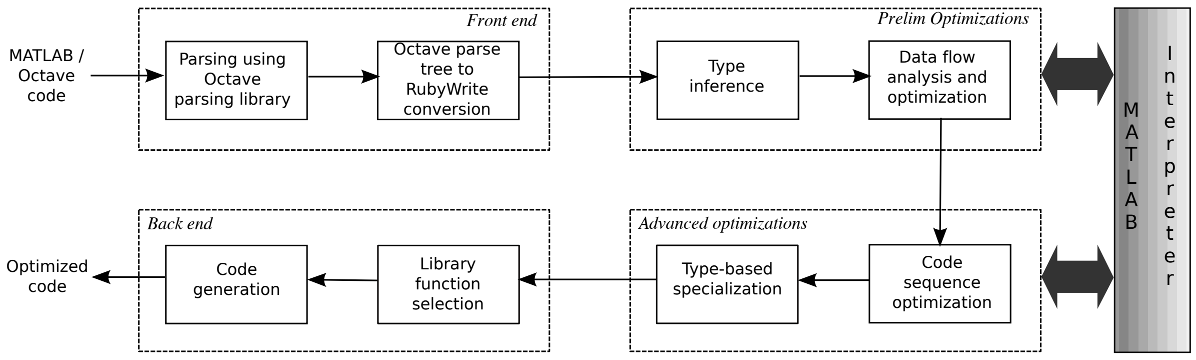
MATLAB / Octave code
Front end
Parsing using Octave parsing library
Octave parse tree to RubyWrite conversion
Prelim Optimizations
Type inference
Data flow analysis and optimization
MATLAB
Interpreter
Back end
Optimized code
Code generation
Library function selection
Advanced optimizations
Type-based specialization
Code sequence optimization

Figure 7.1: Overall architecture of the ParaM compiler.

transformation clear and self documented. Figure 7.2 shows sections of code from the MATLAB expression flattener used in ParaM. Here we see simple recursion with a helper function `flatten_expr_completely` used for `:For` and `:While` loops, while `:Ifs` are handled entirely in a separate helper function `flatten_ifs`.

Another application of rewriters is in framing data flow analysis problems. The transfer functions across different expressions and statements are easily specified using rewriters, which can be directly used in the code passed to the data flow analysis framework in `RubyWrite`. Indeed, rewriters are a common idiom to encode a compiler pass that must do things that are slightly different depending on the statement or expression it is operating on.

ParaM also uses the Ruby Graph Library (RGL) to construct a control-flow graph (CFG) for handling both dependence analysis and SSA form translations. In both cases code is matched using rewriters and the CFG is constructed as the tree is traversed. Once in graph form, algorithms that use the CFG for dependence analysis or the translation into and out of CFG for can be used. Further analysis and transformation steps, taking advantage of the SSA form, can now be implemented more easily and use the graph-based data flow algorithms as they are often described in the literature.

For certain other actions that are either generic for all nodes, or handle only very few nodes specially, generic traversals such as `alltd!` or `bottomup!` are handy. These can usually be more compactly specified than rewriters. In ParaM such traversals are used for actions such as gathering all lvals or rvals, and for quickly getting to all assignment statements within a piece of code. Note that rewriters and other traversals can be mixed, and they often are in ParaM.

The type inference pass in ParaM uses the MATLAB engine library along with the Octave-based parser to help determine the actual types of MATLAB code. Simple C and C++ wrappers provide the interfaces for these tools and allow them to be called directly from Ruby code.

ParaM has backends to compile to both MATLAB and C++. These backends are also coded as simple `RubyWrite` classes using rewriters and full methods to generate the resulting code. Tools written for ParaM are also being reused in other projects, such as a project on analyzing JavaScript for security and privacy issues and a project on compiling the language R

In developing ParaM we extended `RubyWrite` to support annotations on the nodes of the AST, called attributes. This simple change proved useful, not only in being able to track information from the initial MATLAB source code, but is also useful for statement numbering used in some of our transformations. This useful functionality is now a part of the core `RubyWrite` library, as discussed in Section 4.2, and can be used to record meta-data about nodes in the AST.

7.2 Teaching

`RubyWrite` has been used in two graduate compilers courses for several years. Students have received it positively, even though most possessed no prior experience with Ruby. Over the years, students have successfully used `RubyWrite` to implement a variety of compiler passes, including simple loop-transformations, a compiler for a subset of OpenMP on a subset of C, and LLVM code generation. In one year, when the final class project did not require students to use `RubyWrite`, all teams, except one, chose to use `RubyWrite`. The one team that did not use `RubyWrite` needed to work with complete C and Fortran programs and opted to use LLVM, instead. Another team that started out by using LLVM switched to `RubyWrite` mid-way by choosing to focus only on a subset of C for which they could easily write a parser—there was

```
require 'rubywrite'

class Flattener
  include RubyWrite::Basic
  include RubyWrite::Collectives
  include RubyWrite::PrettyPrint
  ...
  # Returns an array of flattened statements.
  define_rw_rewriter :flatten_stmt do
    rewrite :For[:var,:range,:StmtList[:s]] do
      f_range_stmts,range_var = flatten_expr_completely(lookup(:range))
      f_range_stmts << :For[lookup(:var),range_var,:StmtList[flatten_stmts(lookup(:s))]]
    end

    rewrite :While[:cond,:StmtList[:s]] do
      f_cond_stmts,cond_var = flatten_expr_completely(lookup(:cond))
      f_cond_stmts << :While[cond_var, :StmtList[flatten_stmts(lookup(:s))]]
    end

    rewrite :Ifs[:if_list] do
      flatten_ifs lookup(:if_list)
    end

    rewrite :Break[] do |n|
      [n]
    end
    ...
    # Everything else is an expression,
    # being treated as a statement
    default do |n|
      f_stmts,f_expr = flatten_expr(n)
      f_stmts << f_expr
    end
  end

  def main (node)
    if match? :Function[:rvals, :name, :args, :StmtList[:s]], node
      # we got a full function
      build :Function[:rvals,:name,:args, :StmtList[flatten_stmts(lookup(:s))]]
    elsif match? :StmtList[:s], node
      # we got a script
      :StmtList[flatten_stmts(lookup(:s))]
    else
      raise Fail.new("Expected a :Function, " + "got node #node.to_string")
    end
  end
end
```

Figure 7.2: MATLAB expression flattener with RubyWrite

not enough time left in the semester to undertake conversion of the AST produced by LLVM into the form that could be manipulated by `RubyWrite`, even though that is feasible. Overall, several projects successfully used `RubyWrite`. The projects have included a translator from a subset of OpenMP to OpenCL, syntactic support in C for a task library, and translation of declarative specification of parallelism to MPI.

At least one other department outside Indiana University has also used `RubyWrite` in their graduate-level compilers course.

Chapter 8

Related Work

Stratego/XT [2] is the inspiration for `RubyWrite`, and at the core of `RubyWrite` are methods similar to Stratego/XT's combinators. The methods `define_rw_method` and `define_rw_rewriter` in particular, roughly correspond to Stratego strategies and rules, with `match?` and `build` providing pattern matching and term construction. `RubyWrite` includes tree traversal methods, similar to those found in Stratego/XT for applying transformations or analyses across an AST.

While Stratego provides a flexible framework for compiler development and is more mature then `RubyWrite`, the functional nature of Stratego makes maintaining state, such as control flow graphs difficult. Stratego/XT extensions must be written in C, which is also less desirable for complicated analyses that make use of graphs. `RubyWrite`, as an embedded DSL in Ruby, allows us to escape into Ruby for these purposes. Ruby also allows easy access to existing libraries through C extensions.

Other source-to-source transformation tools, including JastAdd [6], POET [24], ROSE [12], Rhodium [10], and CodeBoost [1] target similar functionality, though often with different emphases from `RubyWrite`.

The JastAdd [6] system targets a similar area of compiler construction to `RubyWrite`, using Java's object-oriented class hierarchy along with an external DSL to specify the abstract syntax tree and analysis and transformations on these trees. In JastAdd, unlike `RubyWrite` each type of node in the AST has an associated class encapsulating the transformations for that node type. Instead of AST pattern matching, as provided by `RubyWrite`, method dispatch is used for implementing a given compiler class. We believe there is an advantage in providing pattern matching, in that it simplifies expressing transformations that need to look at the children of a given node, and provides a compact syntax for what would otherwise require a local tree traversal.

POET combines a transformation language and an empirical testing system to allow transformation to be tuned [24]. Although, POET does allow for some generic manipulation of an AST, similar to `RubyWrite` it is largely focused on targeting specific regions of source code to be tuned. To this end it provides a language for specifying parsing of parts of source code and then acts on these AST fragments, preserving unparsed code across the transformation. In our work we often use the full AST, since we might transform from one language into another. The flexibility of `RubyWrite` allows us to use existing parsers for the languages we process.

The ROSE [12] compiler infrastructure provides a C++ library for source-to-source transformation, providing frontends and backends for both C/C++ and Fortran code. Internally ROSE represents source code using the ROSE Object-Oriented IR with transformations written in C++. Although there is nothing specific that ties the ROSE transformation framework to Fortran or C/C++, there are no tools to easily add new frontends and backends, limiting its usefulness for other languages.

The Rhodium framework provides a very different emphasis from `RubyWrite` and the other tools discussed here. Instead of building from term rewriting, Rhodium bases its transformations on data flow equations and provides a framework for proving the soundness of transformations [10]. The emphasis on soundness is very interesting, and something that would be more difficult to implement in a framework like `RubyWrite`, but ultimately transformations such as removing syntactic sugar from a source language seem more natural to implement in a term rewriting system like the one provided by `RubyWrite`.

CodeBoost [1] is an example of a more targeted tool. While the main focus of the original development is to support the Sophus numerical library, it provides a fairly simple way to do compiler transformations within C++ code. It is implemented using Stratego, but provides an array of tools to make writing C++ code transforms easier. While the simple rules support is very similar to some of the work we were originally doing with Stratego, the focus on C++ as a source and destination language is not as good a fit for our purposes as the more general purpose `RubyWrite` library.

Beyond these more general frameworks, a number of tools for specific purposes are also similar to `RubyWrite`. Pavilion, a DSL for writing analysis and optimizations focuses on transforming programs based on a model of their runtime behavior; the Stanford University Intermediate Format provides tools for common C/C++ and Fortran optimizations along with tools for writing compiler passes; the nanopass compiler framework provides a tool for writing compilers through a number of small passes with formal intermediate languages; and the template meta-compiler provides a tool for back-end generation [22, 23, 19, 13].

Finally, the LLVM [9] project provides a set of tools for writing low-level compiler passes. We look at this as largely complementary to our efforts, since we have focused primarily on creating a tool for source-to-source translation while the LLVM project has been very successful in providing tools for developing the compiler back-end and optimizations on its typed-intermediate representation. In fact, the LLVM-Ruby project [11] hints at the possibility of tying `RubyWrite` as a front end compiler framework to the LLVM back-end, allowing both tools to be used together.

Chapter 9

Future Work

While `RubyWrite` is already proving itself useful in our current development projects, we envision additional features that would be useful in the future. Performance in compiling large programs is one potential downside to using Ruby, and while fixes to the performance issues of Ruby are being explored, one possible solution is to parallelize the compilation process. The distributed Ruby library makes it fairly easy to spawn a new process to take on some of the work. We envision two uses for this. First, it could be used in a map-reduce style, where, for instance a global analysis could be run on multiple functions simultaneously and the final results gathered in the central process after the work is completed. Another potential use is a work queue, which is a common formulation of analysis and transformation passes that use data-flow analysis.

Another interesting direction to explore, which is alluded to earlier, is support for concrete syntax in matches as well as builds. One approach would be to integrate a parsing library like `Racc` or `TreeTop` into `RubyWrite` and provide a syntax for specifying how pattern variables can be inserted into a section of concrete source. We see a need for a flexible mechanism for this, since we do not want our selection of a given syntax to make it more difficult for `RubyWrite` to support a given language.

This page intentionally left blank.

Bibliography

[1] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, Sept. 2003. IEEE Computer Society Press.

[2] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52–70, June 2008. Special issue on experimental software and toolkits. DOI: 10.1016/j.scico.2007.11.003.

[3] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. Technical report UU-CS-2005-005, Utrecht University, Department of Information and Computing Sciences, Utrecht, The Netherlands, May 2005.

[4] M. de Jonge. A pretty-printer for every occasion. Technical report SEN-R0115, Centre for Mathematics and Computer Science (CWI), P. O. Box 94079, 1090 GB Amsterdam, The Netherlands, May 2001.

[5] J. W. Eaton. *GNU Octave Manual*. Network Theory Limited, 2002.

[6] T. Ekman and G. Hedin. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, Dec. 2007. DOI: 10.1016/j.scico.2007.02.003.

[7] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly, Sebastopol, CA, USA, 2008.

[8] Ironruby: A fast, compliant Ruby powered by .NET. On the web. `http://www.ironruby.net/`.

[9] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[10] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 364–377, New York, NY, USA, 2005. ACM. DOI: 10.1145/1040305.1040335.

[11] LLVM Ruby. On the web. `http://llvmruby.org/wordpress-llvmruby/`.

[12] D. Quinlan, M. Schordan, Q. Yi, and B. R. de Supinski. Semantic-driven parallelization of loops operating on user-defined containers. In *Lecture Notes in Computer Science*, volume 2958/2004 of *Lecture Notes in Computer Science*, pages 524–538. Springer Berlin / Heidelberg, 2004.

[13] C. V. Reeuwijk. Tm: A code generator for recursive data structures. *Software: Practice and Experience*, 22(10):899–908, Oct. 1992. DOI: 10.1002/spe.4380221008.

[14] Ruby Graph Library. On the web. `http://rgl.rubyforge.org/`. DOI: .

[15] Rubinius: The Ruby virtual machine. On the web. `http://rubini.us/`.

[16] About ruby. On the web. `http://www.ruby-lang.org/en/about/`.

[17] Starfish - ridiculously easy distributed programming with Ruby. On the web. `http://rufy.com/starfish/doc/`.

[18] Ruby inline. On the web. `http://rubyforge.org/projects/rubyinline/`.

[19] D. Sarkar, O. Waddell, and R. K. Dybvig. A nanopass infrastructure for compiler education. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 201–212, New York, NY, USA, 2004. ACM. DOI: 10.1145/1016850.1016878.

[20] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby.* The Pragmatic Programmers, LLC., second edition, 2004.

[21] R. van Beusekom. A vectorizer for Octave. Masters thesis, technical report number INF/SRC_04_53, Utrecht University, Center for Software Technology, Institute of Information and Computing Sciences, Utrecht, The Netherlands, Feb. 2005.

[22] J. J. Willcock. *A Language for Specifying Compiler Optimizations for Generic Software.* Doctoral dissertation, Indiana University, Bloomington, Indiana, USA, Dec. 2008.

[23] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. The suif compiler system: a parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Stanford University, Stanford, CA, USA, 1994.

[24] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. Technical Report CS-TR-2006-006, University of Texas - San Antonio, 2006.