RubyWrite: A Ruby-Embedded Domain-Specific Language for High-Level Transformations

Andrew Keep, Arun Chauhan, Chun-Yu Shei, and Pushkar Ratnalikar

Indiana University, Bloomington, IN 47405, USA
{akeep,achauhan,cshei,pratnali}@cs.indiana.edu

Abstract. We introduce a Domain Specific Language (DSL), called `RubyWrite`, 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 purely object-oriented design and meta-programming capabilities. The syntactic flexibility and advanced language features of Ruby drove our choice of it as the host language. Easy integration of external C interfaces seamlessly within Ruby lets us move performance critical operations to C, or link with external libraries. `RubyWrite` was developed to aid our compiler research where it provides the core of three compiler infrastructure development projects, and is used as a teaching aid in a graduate-level compilers course. We expect `RubyWrite` to be widely applicable and a proof of concept in leveraging an existing modern language to write a portable compiler infrastructure core.

1 Introduction

We describe a Domain Specific Language (DSL), embedded within Ruby, aimed at simplifying the task of source-level transformations through term-rewriting. The DSL, called `RubyWrite`, is motivated by the need for a compiler development environment that allows 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 development experience in C++ had been excruciatingly slow and error-prone. After a pilot study, Stratego/XT [1] was selected as an appropriate language, due to its powerful term-rewriting capabilities, clean and compact syntax, and built-in support for data flow analysis. It resulted in an immediate order of magnitude productivity gain in our development effort. It also proved to be an effective teaching aid in a graduate-level compilers course, letting students implement advanced compiler techniques within class assignments—something that would have been unthinkable in a general-purpose language like C++ without substantial preparatory efforts.

As the compiler development effort in our research group progressed, the analysis 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 we were able to rework the traditional static-single assignment algorithm to operate directly on the abstract syntax trees, other analyses have resisted such adaptation unless algorithmic efficiency is compromised. For instance, a vectorizer for Octave [2], 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 started escaping 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.

The development of `RubyWrite` was undertaken to address these issues. The Ruby language was chosen for its highly flexible syntax and powerful metaprogramming features that make it possible to embed a DSL within it. Ruby's object-oriented programming paradigm is much closer to the more familiar mainstream languages, substantially lowering the barrier to using the language. A large (and growing) collection of libraries for Ruby make a diverse set of components available to aid in the complex task of developing a compiler, for example, the Ruby Graph Library [3] and Ruby bindings for LLVM [4]. Finally, the growing popularity of the language has resulted in heightened interest in addressing its performance bottlenecks [5,6,7]. Ruby also has a well-integrated mechanism to write extensions in C. Combined with libraries, such as RubyInline [8], hotspots can be cleanly optimized by rewriting in C. Some of our own research is also aimed at improving Ruby performance.

`RubyWrite` has turned out to be effective also as a teaching aid. Even though most graduate students are unfamiliar with Ruby, they are well versed in object-oriented programming and conversant with the C-like syntax that many contemporary programming languages (including Ruby) use. 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 the 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 that a similar strategy will enable us and others to achieve high programming productivity in developing compilers using `RubyWrite`, while ensuring that it is not stymied by performance bottlenecks.

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

2 Ruby

```
# -- Reopen the Fixnum class and define the method factorial
class Fixnum
  def factorial
    (self == 0) ? 1 : (self * (self - 1).factorial)
  end
end
# -- Call factorial on the range 1..5 using the map iterator
(1..5).map { |n| n.factorial }  => [1, 2, 6, 24, 120]
# -- Define a module for factorial, so we can use it 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
```

Fig. 1. Example program in Ruby illustrating some of its capabilities

Ruby is a dynamically-typed, purely object-oriented language created by Yukihiro Matsumoto (Matz) [9,10]. Ruby draws heavily from Smalltalk, but incorporates C-like syntax for branching and looping. Ruby also provides facilities for functional programming, providing a syntax for expressing closures.

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 can also be used to define namespaces and both classes and modules can have other class and module definitions embedded within them. 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 new methods that will be used 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, which can be called on the instance objects of a given class. Beyond the traditional alphanumeric characters for method names, `?`, `!`, and `=` can be used as suffixes for method names and 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 the idea of an iterator, 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 parenthesis to be dropped when the meaning of an expression is unambiguous. Embedded DSL writers utilize this feature to blur the line between keyword and method calls. Integers, floating point numbers, strings, arrays, hashes, ranges, and regular expressions all have a literal syntax and 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, while using valid Ruby syntax. In the example in Fig. 1 we define the `factorial` function on `Fixnum` by reopening the class, then create a `Factorial` module and use it to define `factorial` for `Float` as well.

3 Design of `RubyWrite`

An appropriately designed tool can bring dramatic productivity gains to the task of compiler development. Section 9 describes several such tools. `RubyWrite` focuses on rewriting abstract syntax trees (AST) using pattern matching and transformation. However, it also provides other useful features and tools. `RubyWrite`'s nature as an embedded DSL means new features can be added by using Ruby natively, allowing us to test and later incorporate features into the core library. It also means that a growing list of existing libraries are easily leveraged within the language, such as those for parsing, graph manipulations, and code generation. Further, Ruby provides a mechanism to write seamless wrappers to external C or C++ libraries, opening the path to utilize numerous code analysis and optimization libraries. The mechanism can also be used to rewrite those parts of the compiler in C that might become a performance bottleneck although, we have not encountered any such bottlenecks so far in our compilers that could not be eliminated by recoding within Ruby.

Since the primary mechanism in `RubyWrite` is pattern matching and replacement it is more suited to transforming code at relatively high levels where a graph-based intermediate representation is worthwhile, rather than linear intermediate representation.

3.1 Features

`RubyWrite` has been designed to agree with its host language in syntactical conventions. 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` is able to provide 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. Modules are also the way in Ruby to provide "mixins." Making `RubyWrite` available as a module, instead of a class, frees users to incorporate its functionality within any user class and at any point within their class hierarchy.

ShadowBoxing, another embedded language available within the `RubyWrite` module, provides a convenient and succinct way to pretty-print code. Designed after the "Generic Pretty Printer" based on the BOX language [11], ShadowBoxing can be used to naturally specify indentation and bracketing for programming constructs.

Leveraging Ruby's ability to treat classes as objects, `RubyWrite` lets users specify sequences of rewriting classes to implement arbitrary phase order. A compiler using `RubyWrite` may also choose to construct the ordering dynamically based on the input.

`RubyWrite` comes with several convenience methods to traverse the AST in different ways.

3.2 Syntax

As with any other embedded DSL, `RubyWrite` must follow the syntax of the host language, in this case Ruby. Particular attention has been paid to keep the syntax of `RubyWrite` as much "Ruby-like" as possible, with the understanding that there is likely to be much other surrounding code. For example, user methods that transform ASTs are defined with special syntax, but can be used as any other Ruby methods. The methods themselves may contain any arbitrary Ruby code. Similarly, the user is free to use Ruby variables to store references to subtrees and use any helper methods.

4 Using `RubyWrite` for Tree Rewriting

A user class accesses features of `RubyWrite` by including the `RubyWrite` module. If a class needs only a subset of features then it may include specific sub-modules selectively. The sub-modules are `Basic`, `Traversals`, `PrettyPrint`, and `ShadowBoxing`.

4.1 Representing ASTs

`RubyWrite` defines a `Node` class that is used to represent ASTs internally. A node in the AST is an instance of Ruby `String` class, Ruby `Array` class, or `Node` class. An instance of `Node` consists of a label, which is a Ruby `Symbol`, and an arbitrary number of children. All internal AST nodes must be instances of either `Node` or `Array` classes. All leaf nodes must be instances of the `String` class.

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

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

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

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

4.2 Sub-module Basic

Defining `RubyWrite` methods A user class accesses `RubyWrite` by including it as a module. Any method that uses `RubyWrite` must be defined using one of the `define_rw_*` methods, such as the `define_rw_method` syntax. As a convenience, the `RubyWrite` module provides a `run` method that automatically instantiates the class and calls the `method` to perform its transformation. The code in Fig. 2 defines a user class `Example`, and uses the `run` method to process the AST `program`. A `rubywrite` method may be called anywhere a standard method of the user class can be called, using identical syntax.

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

Fig. 2. Minimal `RubyWrite` program

The `match?` and `build` primitives There are two fundamental primitives for tree rewriting in `RubyWrite`, `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 to match and the AST node to match against that pattern. The pattern is specified using a nested-list syntax similar to that used for writing AST. The only difference is that in a pattern leaf nodes may also be `Symbol` objects. `RubyWrite` uses recursive tree matching to perform the matching, using the following recursive 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, each of which matches the pattern.
3. An `Array` in pattern matches an `Array` in AST with the same number of elements, each of which matches the pattern.

4. An unbound `Symbol` in pattern matches any subtree in AST and the symbol is bound to the subtree within the AST that it matches. If `Symbol` has already been bound within the pattern then it matches exactly its current binding. If the symbol is `:_` no binding is created.

If the match succeeds `match?` returns `true` with symbols specified in pattern bound to the corresponding subtrees in AST. If a symbol occurs more than once in pattern then the corresponding subtrees in AST must be identical. If match fails no new bindings are created and `match?` returns `false`. The bindings may be looked up using a `lookup` method. A new tree can be created using the `[]` method on `Symbol`. Alternatively, a shortcut method `build` automatically replaces any occurrence of a `Symbol` at the leaf level by the tree to which it is bound. The following simple example to commute the operands of a binary operation illustrates the syntax.

```
if match? :BinOp[:op1, :binop, :op2], node
   commuted = :BinOp[lookup(:op2), lookup(:binop), lookup(:op1)]
end
```

OR

```
if match? :BinOp[:op1, :binop, :op2], node
   commuted = build :BinOp[:op2, :binop, :op1]
end
```

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]}
```

An important thing to note here is that `match?` and `build`, along with the optional blocks passed to them, modify the bindings of the surrounding method. In contrast, any other method gets its own set of bindings that is discarded at the end of the method. Standard Ruby scoping rules apply to the blocks, which are closures with any free variables looked up in the surrounding context. If a new variable is defined within a block then it is *not* visible outside the block. Thus, if the variable `commuted` is defined for the first time in the block passed to `match?` then it will not be visible outside the block.

Defining `RubyWrite` rewriters In addition to methods, a user class may also define *rewriters* that are special methods consisting of a set of *rewrite rules*. A rewriter method is always invoked on an AST node, which is matched against the patterns specified within the rewrite rules and the corresponding code block executed. The argument to the rewriting block, `n`, gets bound to the AST node that matched. The pattern is specified using the same syntax that `match?` expects and if a match is successful, `Symbol`s at leaf nodes are bound to subtrees similarly to `match?`. The blocks may call the method being defined, `xform_statements`, recursively. An example in Fig. 3 illustrates the syntax.

The patterns specified for rewriting must match non-overlapping instances of the AST. `RubyWrite` has no way to verify that this is indeed the case, however the behavior of the rewriter may be non-deterministic if more than one specified pattern may match a given AST. If a default is specified then RubyWrite invokes its block if the given AST node fails to match any pattern. If no default has been specified, then it raises an exception when none of the given patterns matches.

Other basic primitives The try primitive lets any code be executed conditionally. If the code returns nil or false then any changes it made to Symbol bindings are rolled back. A binding may be created explicitly using set! that takes a Symbol and an AST node. Finally, match (not ending in ?) behaves similarly to match?, except that it raises an exception when the match fails.

```
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
```

Fig. 3. An example of using a rewriter

Semantics The behavior of RubyWrite primitives is summarized in Fig. 4. $\mathcal{E}$ is the environment that binds names to AST nodes. λ is a code block that may be passed as a Ruby block, ϕ is user-defined RubyWrite method or rewriter. We use the notation $n.f$ to indicate that f is invoked on the AST node n. The rest of the

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

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

Fig. 4. Semantics for basic RubyWrite primitives

notations are standard. Notice that these semantics are only for the environment maintained by `RubyWrite` and do not account for Ruby's environment.

Exceptions raised in any code block passed to `match` or `build` are passed back to the caller. Similarly, any uncaught exceptions inside `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.

4.3 Sub-module Traversals

It is often convenient to encode a compiler pass as a transformer that operates on specific node types, or performs related but different actions on each node type. This can be achieved with rewriters described earlier in Sec. 4.2. However, a compiler pass may simply be an analysis phase or information gathering phase, in which case a rewriter is somewhat unwieldy. `RubyWrite` provides two other mechanisms to encode traversals through an AST where a different action is needed based on node types.

Special types of `RubyWrite` methods may be defined using `define_rw_postorder` or `define_rw_preorder`. The definition body contains a sequence of actions (Ruby blocks), one for each node type, along with an optional default action. In the case of preorder traversal if a block returns `nil` or `false` (i.e., `nil` or `false` is the last expression to be executed within the block) then traversal stops for that subtree. In other words, no further subtrees under that node are visited. Fig. 5 is a simple example to gather all variable names that appear on the left hand sides of assignments. Notice that the currently matched node is passed as an argument to the block. Notice also that one of the blocks uses the local variable `names`. This is possible since, unless overridden, a Ruby block is treated as a closure and, consequently, has access to its surrounding scope.

```
names = NameTable.new
define_rw_preorder :gather_variable_names do
  upon :Assignment 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
```

Fig. 5. Code to gather all variables

Sometimes it is useful to traverse the children of a node in reverse order. `RubyWrite` provides `define_rw_rpostorder` and `define_rw_rpreorder` for such cases.

`RubyWrite` also has several predefined tree traversal methods that have been motivated by Stratego [12]. In each case, the traversal method must be passed the AST to be traversed and a block of code. The code is executed at each node that is traversed. If the code returns `nil` or `false` it is assumed to *fail* on that node. Otherwise, it is assumed to *succeed*. There are two versions of each traversal method, one ending in `!` and another ending in `?`. The former modifies the AST as it is traversed, while the latter only returns a Boolean status indicating whether the traversal was successful without modifying the AST.

Method `all!` applies the given block to each child of the given AST node. If any application fails it raises an exception. `one!` applies the given block to each child, one at a time, and returns as soon as it succeeds on a child. It raises an exception if the block fails on all the children. `topdown!` applies the block to the given node *and* recursively to all its children. The `bottomup!` traversal does the same in a bottom-up fashion, i.e., applies the block to each child and then to the node. The `alltd!` traversal is similar to `topdown!` except that it recursively traverses the children of a node only if the block fails on the node.

The versions ending in `?` behave similarly, but never raise an exception and do not modify the AST. Instead, they return `false` on failure and `true` on success.

The code in Fig. 5 to gather left hand sides of all assignments can also be written using predefined traversal methods, as in Fig. 6. Since we do not need to modify the AST we use `alltd?`, even though we ignore the return value.

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

Fig. 6. Variable gathering rewritten with `alltd?` traversal

5 Pretty Printing and Other Support

`RubyWrite` supports pretty printing of an arbitrary AST and also provides support for building an "unparser", to translate an AST into the concrete syntax of the target language, through the module `ShadowBoxing`.

5.1 Sub-module PrettyPrint

ASTs may be dumped, or pretty-printed, using the `PrettyPrint` sub-module. The pretty-printer makes use of the `ShadowBoxing` sub-module internally, which is described in Sec. 5.2. Note that 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.

```
class UnparseMATLAB
  def unparse(node)
    boxer = ShadowBoxing.new do
      rule :Var do |var| var end
      rule :Const do |val| val end
      rule :Assignment 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
  end
end
```

Fig. 7. A snippet of a pretty-printer for MATLAB

5.2 Sub-module ShadowBoxing

The `ShadowBoxing` sub-module implements its own embedded language for writing pretty-printers for concrete syntax—in other words "unparsers"—which is particularly suited to pretty printing structured languages. The ShadowBoxing language is motivated from the BOX language that has also been used to build the Generic Pretty Printer (GPP) [11].

A concrete syntax pretty printer is created by instantiating the `ShadowBoxing` class and passing the `new` method a block that defines the printing rules. Rules are specified with `rule` statements, one for each type of node in abstract syntax. This is similar to per-node actions used within a traversal method defined with `define_rw_preorder`, described in Sec. 4.3. The printing format is specified with one of two types of "boxes", `h` and `v`. Everything within an `h` box is typeset horizontally, and everything within a `v` box is typeset vertically. Spacing options may be specified using optional arguments. Fig. 7 shows some code snippets of a pretty-printer for MATLAB that we use within our ParaM compiler.

When the block for a node is invoked it is passed all the children of the node as distinct arguments. For example, the rule for `:Function` nodes expects four arguments. Each printing rule must return a string. A printing rule could be very simple, such as the one for `:Var`, that simply returns the variable name. However, it could also make use of the formatting primitives. The `:Assignment` rule uses the `h` primitive and the formatting option `:hs => 1` specifies that there should be a one-character spacing between each of the arguments, formatted horizontally. The first argument to a formatting primitive is always a hash of formatting options. Each formatting primitive returns a string and the primitives can be freely nested. The rule for `:While` nests a `v` and an `h` primitive inside the `v` primitive. The `:is` option specifies the amount of indentation. A formatting primitive may be passed either a string or an AST node as an argument. In creating the output string it reproduces the string arguments literally and invokes the rules recursively to format any AST node. Finally, the rule for `:Function` demonstrates another formatting primitive `h_star`. It accepts an array and a *separator* between the array elements. The array may consist of any combination of strings and AST nodes.

5.3 Concrete Syntax

`RubyWrite` supports instantiation of code directly in concrete syntax. This is done by letting a special type of AST node indicate that it contains concrete syntax, thus allowing mixing of abstract and concrete syntax. Since Ruby provides a mechanism to escape into arbitrary Ruby code within a string, there is very little that `RubyWrite` has to do. For example, we can write the code to swap the operands of a binary operation as follows.

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

In this case using concrete syntax does not buy us much. Nevertheless, it illustrates the technique. Here, `pp` is a method that looks up the given `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 inside the `match?` primitive or a rewriter. In that case, the user class must supply a parser that `RubyWrite` can use to parse strings into ASTs. The compiler writer is also responsible for ensuring that the portions of code represented concretely within an AST are pretty-printed correctly by supplying appropriate rules. A simple, albeit somewhat inefficient, way to handle this is to first convert all concretely specified subtrees back to abstract syntax just before pretty printing the entire AST.

Unfortunately, the ability to specify patterns for matching using concrete syntax depends on the specific language being compiled. We have kept `RubyWrite` independent of the source language as well as the parsing tool that might be employed by the compiler. Our projects that employ `RubyWrite` work with multiple languages and use different parsing methods. Even though we have no immediate need for this within our own compiler projects, we plan to investigate the relative benefits of providing an optional feature to interface with an established Ruby parsing library, such as `Racc` or `TreeTop`.

5.4 Composing Compilation Phases

The simplest way to compose compiler phases is to chain calls to them. For example, the following calls three different phases.

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

This works fine as long as the exact sequence of phases to be applied is fixed and known. However, it is possible to make the syntax more readable and `RubyWrite` does that by providing a method called `xform`.

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

Transformations are listed by their class names. Alternatively, the arguments may be instances of those classes, or any combination of classes and instances. The argument may also be an arbitrary array of the phases to be applied to transform `input` to `output`. This flexibility enables phases to be selected dynamically, and ordered dynamically, even based on the program being compiled.

6 Data Flow Analysis in RubyWrite

In order to perform data flow analysis directly on an AST, the exact action to perform on each AST node type depends on the language being compiled. Additionally, the transfer functions and meet operations depend not only on the language but also on the specific data flow analysis problem that needs to be solved. However, using `RubyWrite` it is not very difficult to write a language-specific data flow analysis framework that parameterizes problem-specific aspects. Fig. 8 shows such a generic framework for a hypothetical language consisting of assignments and two types of compound statements—`if-else` statements and `while` loops. It uses the preorder method of AST traversal and specifies actions for each node type. A helper function to compute fixed points over sets is used to handle loops. The method `analyze` encapsulates problem-specific behavior by providing a hook into how the set of values being computed needs to be updated. Note that `analyze` can itself be implemented as a rewriter (for example) and can also be recursive. The meet operation (“`|`”) may be defined on the class of `set` as appropriate for a problem.

In general, any data flow support framework must require the user class to define the meet operations and equality operations to compute fixed points. Fortunately, the built-in data structures in Ruby support a rich set of operations, including comparison, merging, etc. As a result, building a basic framework for data flow analysis is straightforward.

In Fig. 8, we take the approach of working directly on the AST, instead of first constructing a control flow graph. Using graph libraries, such as the Ruby Graph Library (RGL), and Ruby’s support for set manipulation, data flow analysis can also be easily implemented using the conventional approach of setting up global data flow equations and solving them iteratively. The advantage of working with a control flow graph is that since control flow is captured as a graph, the framework is language independent. The framework can be instantiated with problem-specific aspects of set comparison, initial values, and meet operations

```
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
```

Fig. 8. A generic data flow analyzer that may serve as a base class

over the set of values being computed. RubyWrite provides such a framework as a sub-module under the assumption that the control flow graph uses RGL representation and each node contains only one statement or expression. In this way, users have a choice of working directly with the AST and using the template in Fig 8 to write a general 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 build using RGL.

In order to better illustrate the RubyWrite data flow module, we provide two examples of constant propagation on a simple imperative language with branching and looping constructs in Figs. 9 and 10. In both implementations an environment ("e") providing meet and copy operations encapsulates environment handling. In Ruby, we can implement this by adding these methods to the existing Hash class. Both implementations also rely on an is_constant? method to know when an expression is a constant.

The first implementation in Fig. 9 is written using RubyWrite rewriters and runs directly over the AST of the incoming program. This approach allows us to skip building the graph, but is still a relatively compact implementation. In addition to defining rewriters for updating the environment though, we must also handle the meet and fixed point operations for if and while explicitly. It makes use of the fixed_point method defined in Fig. 8 to handle while.

In Fig. 10 is an example of how the same process might be written using the data flow library. In this version the compiler writer must provide methods for building the graph representation in RGL format. This is needed because RubyWrite would otherwise not know where edges need to be added to the control flow graph. As mentioned earlier each node should contain a single statement

```
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
```

Fig. 9. Constant propagation using RubyWrite 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
```

Fig. 10. Constant propagation using `RubyWrite` data flow.

or expression. Once this work is done the job of specifying the constant propagation is much simpler, since the data flow module takes care of performing meet and fixed point operations on the graph. The compiler writer need only specify a transfer function to update the environment and replace variable references with constants when appropriate.

7 Implementation

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

7.1 `RubyWrite` as a Module

Implementing `RubyWrite` as a module lends flexibility to the compiler design. Since Ruby supports single inheritance, the module is a mixin. The flexibility of reopening classes lets the code be organized into sub-modules based on functionality, rather than class hierarchy. As a result, a user class is free, and even encouraged, to include only those sub-modules that it needs. Multiply included modules are automatically detected by Ruby and included exactly once.

7.2 Creating `RubyWrite` Method

`RubyWrite` requires special syntax to create methods that will use `RubyWrite` primitives because each method must be surrounded by some code to set up an execution environment and tear it down when the method finishes. Two Ruby features make this possible. Ruby allows arbitrary code inside class definitions. Thus, method definitions can be interleaved with method calls. The `define_rw_*` primitives are really calls to methods defined in `RubyWrite::Basic` module. Moreover, Ruby has extensive meta-programming support, allowing dynamic definition of new methods and classes. Left hand side of Fig. 11 shows the code for the `RubyWrite` primitive `define_rw_method`.

```
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
```

Fig. 11. Implementation of method definition within the `RubyWrite` module

Notice that the method definition starts out by creating a new binding of `Symbol` to AST nodes, and saving the current one. The original bindings are restored using the `ensure` block after the method finishes or raises an exception. There is one complication: since the `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 its instance methods are available, its class methods are not. This means that the `define_rw_method` cannot simply be a part of the `RubyWrite` module. Our solution is to 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 each time the module is mixed into a user class. The `included` method *extends* the user class, making the methods in `ClassMethods` class methods of the user class. Remember that in Ruby everything is an object, including the class definition. When we extend the class object it adds to the list of method definitions in its meta-class, creating them as class methods. Adding methods to the meta-class of a class is, in fact, how class methods are always created in Ruby, although some syntactic sugar masks this fact. The code appears on the right hand side of Fig. 11.

7.3 Creating Rewriter

In addition to the complication of creating a method, rewriters also pose the problem of matching patterns efficiently. A simple solution could be to use the Ruby `case` statement, which is equivalent to using a sequence of `if-then-else`. Hashing on labels of the root nodes of all the patterns is a better solution, but still sub-optimal since multiple nodes often have identical root labels. Instead, we use a trie that indexes on the complete pattern. This has the advantage that some overlapping patterns can be detected at the time of constructing the trie, which is created once at the class definition time and reused for each instance of that class.

As the example in Sec. 6 showed, rewriters can be passed any arbitrary arguments, which are in turn passed on to the blocks that get executed upon matching their specified patterns. Users must ensure that the number of arguments match and they are consistent across the rewriter and the per-pattern blocks.

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. However, we would like to make this call within the context of an internal `Rules` class that encapsulates the trie and other data. This is possible in Ruby using `instance_exec`. Unfortunately, this places the block passed to `rewrite` in context of `Rules` as well, whereas the code there really belongs to the class that is implementing the rewriter. Again we use a call to `instance_exec` to bring the code back into the original context.

7.4 Handling Traversals

The various tree traversal methods defined with `define_rw_preorder` and other similar primitives are implemented similarly to rewriter. Since the actions in these traversals are distinguished only by the node label a simple hash-based scheme to find the code to run is efficient and effective.

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

Fig. 12. Implementing `alltd!`

Other recursive traversals are relatively simple to implement. The `all!` and `one!` traversals are simple non-recursive methods. The `topdown!` and `bottomup!` traversals use simple recursion on the tree. Finally, `alltd!` uses the `try` primitive to make sure that the side-effects on bindings can be rolled back upon failure.

7.5 Handling Compilation Phases

Since classes may be treated as regular objects in Ruby, they can be passed as arguments and assigned to variables. This greatly simplifies the implementation of `xform` method, as the code in Fig. 13 shows.

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

Fig. 13. Implementing support for compilation phases

8 Compiler Projects Using RubyWrite

We have been using RubyWrite in three different compiler research projects. This section describes our experience with each of those and also with a graduate compilers course that uses RubyWrite.

8.1 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 [13] and the output converted to RubyWrite representation. Type inference and data flow analyses are implemented using RubyWrite with techniques described in this paper. There is a dependence analyzer currently under development that uses the Ruby Graph Library for building and manipulating control flow graphs and program dependence graphs [3]. ParaM uses ShadowBoxing to pretty-print the translated code to MATLAB or C, as appropriate.

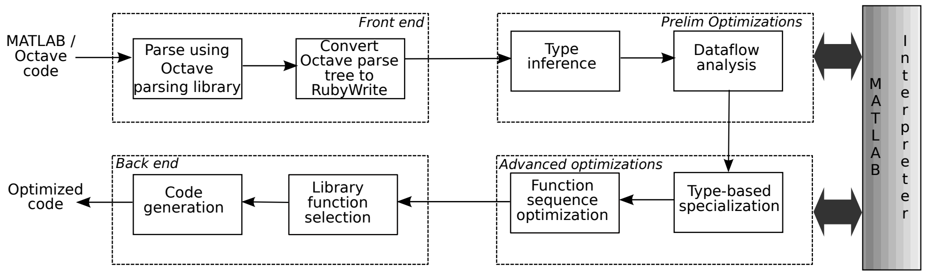

Fig. 14. Overall architecture of the ParaM compiler.

Figure 14 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 flattening must process 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 transformation clear and self documented. Figure 15 shows sections of code from the MATLAB expression flattener used in ParaM.

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.

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.

8.2 RubyC

RubyC is a project to optimize Ruby by partially evaluating it.

A parser based on Ruby's own YACC file is used to parse Ruby code and the distributed Ruby (dRb) library is used to interpret Ruby within a sandbox for partial evaluation.

Once the Ruby code has been parsed into an AST, `RubyWrite` rewriters are used with a `bottomup!` traversal to provide a simple way to remove syntactic sugar from the parsed source code. For example, forms like `unless` are replaced with their equivalent `if` expressions and binary operators are turned into the appropriate method invocations. This allows the partial evaluator to focus on the core forms of the language. RubyC then uses `RubyWrite` rewriters to implement a data flow analysis based partial evaluation, maintaining information about the Ruby environment as partial evaluation proceeds. During this stage, ShadowBoxing is used to unparse AST nodes into Ruby source that can be passed to the dRb session that runs in parallel with the partial evaluator.

A dead code analysis stage is also under development to clean up unnecessary assignments and unused expressions once the partial evaluation is finished. The final stage uses ShadowBoxing again to generate the final partially evaluated version of the program.

8.3 DELTA-P

DELTA-P is a project to explore automatic parallelization techniques, currently for MATLAB programs, on heterogeneous platforms such as those comprising multi-core CPUs and GPGPUs.

To analyze MATLAB programs, DELTA-P leverages the Octave compiler similar to ParaM. It uses the parser to automatically instrument the code for

```
require 'rubywrite'

class Flattener
  include RubyWrite::Basic
  include RubyWrite::Collectives
  include RubyWrite::PrettyPrint
  ...
  # Statement flattener always 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
```

Fig. 15. MATLAB expression flattener with `RubyWrite`

empirical measurements and for automatic analysis of profiler output. Finally, the Ruby Graph Library is used to construct a simple data dependence graph to guide program transformation. Using Ruby has enabled this highly experimental effort to build an integrated framework for instrumentation, running, data collection and analysis, and program transformation within a single tightly coupled application.

Another goal of the DELTA-P project is to develop compiler techniques for a declarative parallel language that our group is developing for retargetable performance optimization of parallel programs. Toward that goal, the project is exploring integration of `RubyWrite` with LLVM, which will be used to handle transformations of C/C++ and Fortran programs close to source level.

8.4 Teaching

`RubyWrite` has been used in a graduate compilers course. Students received it positively, even though most had not programmed in Ruby before. Three programming assignments were implemented by all the students in the class, including simple loop-transformations and a compiler for a subset of OpenMP on a subset of C. The final class project did not require students to use `RubyWrite`, but 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 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 (and is, in fact, the approach taken by DELTA-P project). Overall, several projects successfully used `RubyWrite`. The projects 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 already evinced interest in using `RubyWrite` in their compilers course.

9 Related Work

The original inspiration for `RubyWrite` was Stratego/XT [1], 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 tree 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 POET [14], ROSE [15], JastAdd [16], CodeBoost [17], and Rhodium [18] target similar functionality, though often with different emphases from `RubyWrite`.

POET combines a transformation language and an empirical testing system to allow transformation to be tuned [14]. 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 much of our work we need the full AST, since our source language and target language may not be the same. The flexibility of `RubyWrite` also allows us to plug-in existing parsers for the languages we process.

The ROSE [15] 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 JastAdd [16] 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 that encapsulates the transformations for that node type. This means instead of AST pattern matching, as provided by `RubyWrite`, method dispatch can be 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.

An example of a more domain specific tool is CodeBoost [17]. It was originally developed to support work on the Sophus numerical library, and while the main focus of the original development was to support the Sophus implementation, it provides a fairly simple way to do compiler transformations within C++ code. It was 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 as the more general purpose `RubyWrite` library.

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 [18]. 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`.

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 [19,20,21,22].

Finally, the LLVM [23] 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 [4] 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.

10 Conclusion

This paper has described a domain-specific language, called `RubyWrite`, embedded within the increasingly popular language, Ruby. Embedding within Ruby gives it the power to integrate source-level analysis and transformation within an advanced modern language. We have been using `RubyWrite` for our own compiler development efforts for MATLAB and Ruby. Our experience so far with the research projects and a graduate compilers course shows dramatically improved productivity over traditional languages and lower barrier to entry compared to special purpose external DSLs.

References

1. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A Language and Toolset for Program Transformation. Science of Computer Programming **72**(1–2) (June 2008) 52–70 Special issue on experimental software and toolkits.
2. van Beusekom, R.: 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 (February 2005)
3. : Ruby Graph Library. On the web `http://rgl.rubyforge.org/`.
4. : LLVM Ruby. On the web `http://llvmruby.org/wordpress-llvmruby/`.
5. : IronRuby: A Fast, Compliant Ruby Powered by .NET. On the web `http://www.ironruby.net/`.

6. : Rubinius: The Ruby Virtual Machine. On the web `http://rubini.us/`.
7. : Starfish - Ridiculously Easy Distributed Programming with Ruby. On the web `http://rufy.com/starfish/doc/`.
8. : Ruby Inline. On the web `http://rubyforge.org/projects/rubyinline/`.
9. Flanagan, D., Matsumoto, Y.: The Ruby Programming Language. O'Reilly, Sebastopol, CA, USA (2008)
10. Thomas, D., Fowler, C., Hunt, A.: Programming Ruby. Second edn. The Pragmatic Programmers, LLC. (2004)
11. de Jonge, M.: 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)
12. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: 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)
13. Eaton, J.W.: GNU Octave Manual. Network Theory Limited (2002)
14. Yi, Q., Seymour, K., You, H., Vuduc, R., Quinlan, D.: POET: Parameterized Optimizations for Empirical Tuning. Technical Report CS-TR-2006-006, University of Texas - San Antonio (2006)
15. Quinlan, D., Schordan, M., Yi, Q., de Supinski, B.R.: 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. Springer Berlin / Heidelberg (2004) 524–538
16. Ekman, T., Hedin, G.: The JastAdd system – modular extensible compiler construction. Science of Computer Programming **69**(1-3) (December 2007) 14–26
17. Bagge, O.S., Kalleberg, K.T., Haveraaen, M., Visser, E.: Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++ Programs. In Binkley, D., Tonella, P., eds.: Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003), Amsterdam, The Netherlands, IEEE Computer Society Press (September 2003) 65–75
18. Lerner, S., Millstein, T., Rice, E., Chambers, C.: 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, New York, NY, USA, ACM (2005) 364–377
19. Willcock, J.J.: A Language for Specifying Compiler Optimizations for Generic Software. Doctoral dissertation, Indiana University, Bloomington, Indiana, USA (December 2008)
20. Wilson, R., French, R., Wilson, C., Amarasinghe, S., Anderson, J., Tjiang, S., Liao, S., Tseng, C., Hall, M., Lam, M., Hennessy, J.: The SUIF Compiler System: a Parallelizing and Optimizing Research Compiler. Technical Report CSL-TR-94-620, Stanford University, Stanford, CA, USA (1994)
21. Sarkar, D., Waddell, O., Dybvig, R.K.: A nanopass infrastructure for compiler education. In: ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM (2004) 201–212
22. Reeuwijk, C.V.: Tm: A Code Generator for Recursive Data Structures. Software: Practice and Experience **22**(10) (October 1992) 899–908
23. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). (2004)