

sRuby - A Ruby dialect for low-level programming

Robert Feldt

April 4, 2001

Version: *Revision* : 1.3 *Date* : 2001/02/13 12 : 51 : 13

Abstract

This is an evolving document describing sRuby, a subset of Ruby that can be easily compiled to fast low-level code. The purpose of developing sRuby is to use it to implement a Ruby virtual machine. However, we anticipate that it can be used to write Ruby extensions that needs to bridge the gap between Ruby and a low-level language (C) in an easy and portable way.

1 Introduction

See documents on RubyVM to get a background on why sRuby is needed.

2 Notation

In this text we use the following notational conventions:

-

We also use the following terms:

- VM = Virtual machine
- mother VM = The Ruby virtual machine running the sRuby compiler. This is MRI if not explicitly stated otherwise.
- child VM = The Ruby virtual machine being created by the sRuby compiler.

3 Related work

This section introduces two related projects that have influenced the design of sRuby. It also briefly summarizes other related work that might be interesting to take a look at in the future development of sRuby.

3.1 Slang in the Smalltalk environment Squeak

Squeak is a Smalltalk environment with a VM written in a subset of Smalltalk called Slang [1]. Slang programs can be developed and debugged inside a Smalltalk environment and then translated to C code. The C code can be compiled with an (external) C compiler to get an executable. Details on Slang was taken from [Extending Squeak2000].

Slang is basically C code with a Smalltalk syntax. No type inference is carried out; types are assumed to be 32-bit ints unless explicitly declared something else. No code blocks (closures) can be used, only the “basic” control structures while and if. Smalltalk (ST) methods are translated into C language functions (returning int). Other characteristics of Slang:

- ST literals (integers, chars and strings) are translated to C constants. Symbol literals (?) translated into C strings.
- Array literals not allowed.
- Expression assignment translated directly to C.
- Message sends converted to function calls where receiver of message is first argument and arguments are rest of arguments.
- Some message sends are mapped directly to C code: and, or, arithmetic ops, shifting, inverting, etc.
- Array access mapped to C array access.

```
foo at: exp -> foo[exp]
```

The main controlling factor behind the development of Slang seems to have been ease of translation. There is no need for any advanced compilation techniques. Slang source code need only be parsed with a Smalltalk parser. The translation can occur in one pass without the need for any type analysis or intermediate code generation.

The only optimization of Slang code is the inlining of small methods. Larger methods can also be inlined by explicitly marking them. Inlining was reported as a very important factor in getting good performance [1]. Efficiency of the VM has later been improved by care in translation leading to better use of register variables, strategies for garbage collection and interrupt handling [BTF once more, Ingalls 2000].

Squeak supports the compilation of “pluggable primitives” [BTF once more, Ingalls 2000]. They are Slang programs compiled and linked to dynamically linked libraries that can be dynamically plugged into the Squeak VM. They have been used to add graphics, sound and copression. It is unclear why the VM needs to be extended to support these functions (compared to the Ruby way of writing an extension).

Slang also contains a number of support functions for converting between C and ST values. They are similar to the C API in Ruby. Details can be found in [Extending Squeak2000].

3.1.1 Discussion of Slang

The decision to convert almost everything to int's may not be a bad choice. It's actually very similar to the Ruby way of using VALUE for oop's. VALUE's are simply unsigned longs, ie. very similar to making everything int's. Simple macros can be used to determine if the object is Fixnum or a direct-memory pointer. Memory can be accessed by `SRubyLowLevel::Memory[address]`. This would make things very simple since there is no need for type inference.

3.2 SPiCE Smalltalk-to-C translation

3.3 Pre-Scheme in Scheme48

Scheme48 is a Scheme virtual machine written in a statically typed dialect of Scheme called Pre-Scheme [Kelsey97]. Pre-Scheme preserves the semantics of Scheme with the caveat that Pre-Scheme programs may run out of space since it lacks garbage collection and full, proper tail recursion. The main differences between Pre-Scheme and Scheme programs:

- Statically typed by the compiler using a type reconstruction algorithm.
- No garbage collection.
- Top-level forms are evaluated at compile time and may make use of full Scheme.
- Not all tail-recursive calls are done with proper tail recursion.

The above restrictions were not enough to get desired performance. A powerful compiler doing partial evaluation was also needed. The compiler is a transformational compiler using a number of optimization techniques: beta reduction, block compilation (aka whole-program compilation/transformation), transforming tail recursion to iteration, hoisting closures, constant folding and C translation tricks exploiting the capabilities of the C compiler.

Excluding comments the Scheme48 VM is 2314 lines of Pre-Scheme code and is compiled into 8467 lines of C code. The performance of the VM was comparable (and even superior on a number of benchmarks) to comparable programs written directly in C. Inlining was very important in realising this performance.

3.3.1 Type inference/reconstruction

“The goal of Pre-Schemes static type checking is to model Scheme’s dynamic typing as accurately as possible while still allowing the compiler to decide upon a machine representation for every variable, to insert any necessary coercions, and to produce intelligible type error messages” [Kelsey 97]. A slightly modified Hindley/Milner polymorphic type reconstruction algorithm is used. Only cheap coercions are used, for example between different numeric types. The output

from the type reconstruction algorithm is a program augmented with coercions and a set of type relations encoding constraints on the coercions.

Four different kinds of polymorphism is allowed:

- No polymorphism
- single size polymorphism. Different types of values are allowed, but they must share representation size.
- multi-size polymorphism. Different copies of the procedure are produced for different sizes of the values.
- full polymorphism. A separate copy of the procedure is produced (and inlined) for each use.

3.4 Other related projects

- IBM's JavaInJava system Jalapeno

4 sRuby

This section presents the goals with sRuby, how it differs from Ruby and describes what Ruby constructs are available in it.

4.1 Goals

Goals with sRuby in order of importance:

1. sRuby programs are valid Ruby programs, ie. sRuby is a subset of Ruby (syntax and/or semantics).
2. sRuby can be compiled to efficient binaries.
3. sRuby should give access to low-level, machine-level data structures (addresses etc).
4. sRuby should need only a minimum of special support classes but when needed they should be cleanly separated from other classes.

Goal 1 is important since you should be able to use ordinary Ruby tools to work with your sRuby programs. You shouldn't need to learn a new language.

Goal 2 is important since we intend to write applications in sRuby where performance is crucial. Ruby VM's are the most obvious example. If they are not efficient people won't use them even though they are neat or conceptually pleasing.

Goal 3 is important since we anticipate that people will have to "fiddle with the machine" in typical systems or low-level applications that sRuby is intended for.

Goal 4 states that we should strive for not using special support classes for sRuby that violates standard Ruby semantics. However when there are good reasons to use a support class it should be cleanly separated from the rest of the classes, preferably in a separate namespace SRuby. However, support classes should generally be avoided since they will be an additional thing that people have to learn to start using sRuby.

4.2 Differences between sRuby and Ruby

sRuby should be as close to Ruby as possible; we want to write code in almost the same way we would have done it in full Ruby. However we will need to make some deviations from Ruby semantics if we want good performance. Otherwise we are faced with writing a full Ruby-*i*C translator (and not like rb2c that (correct me if I'm wrong) needs matz Ruby implementation to work), ie. essentially a Ruby compiler. I think this is doable in the long run but it'll not be easy and will require tight integration with a VM/run-time.

Things that we anticipate will be different between sRuby and Ruby:

- No garbage collection. (the idea is that we should write Ruby's GC in this subset so we cannot presuppose GC in the subset. Besides we'd lose performance. However, a q for matz: Does the interpreter itself make use of GC heap mem for internal data structures?)
- Statically typed. To get good performance we need to resolve Ruby's dynamic typing at compile-time. This can be done either by using (complex) type inference/reconstruction algorithms or by using the existing Ruby interpreter (I'm thinking something like: install trace function that intercepts each "call" and "c-call" and saves the types of the arguments to the call (we can get them from the binding, right?), then run the program and collect type info, then convert each method invocation with a unique type signature into a unique C function)
- No dynamic redefinition of classes, ie. you cannot redefine methods dynamically. Actually, you're allowed to do it but the semantics isn't preserved; later additions will override previous ones prior to any code execution. Reason is we'll probably have to compile methods into C functions we need some static mapping. We might work around this with some name mangling technique but I'm not sure its worth the effort.
- No eval. When writing the interpreter its not there so... (Same goes for a lot of the stuff currently in Ruby like safe levels etc)
- No full Ruby Std lib. The basic stuff like Fixnums and Float can be converted directly but we will probably have to deal with the other basic ones like String, Array and Hash. Anyone knows what data structures are used in the current interpreter and will be crucial when writing VM components? We can take two approaches here: (1) Write CArray and

CHash that give functionality similar to Array and Hash but are written directly in the subset language, or (2) implement ObjectMemory with GC and then "real" Array and Hash using the ObjectMemory. Latter will be easier and less error-prone but probably slower. How does matz do today? I guess he uses the low-level hash implementation (st?)?

I think we should also disallow singleton classes. It might be possible to add them with some kind of name mangling techniques but I don't think they'll be necessary for the VM components.

4.3 So whats left in sRuby?

The above section listed the main differences between sRuby and Ruby. This section lists the things that are still in there, ie. the Ruby constructs that can be used in sRuby.

- Classes
- Modules
- Iterators (converted to iteration, ie. for loops)
- Blocks/closures (converted to functions and function pointers)
- Basic Ruby syntax
- globals, class vars, instance vars and local/temp vars

* Classes and Modules (converted to structs for instance vars and function for methods) * Iterators (converted to iteration, ie. for loops) * Blocks/closures (converted to functions and function pointers) * Basic Ruby syntax * globals, class vars, instance vars and local/temp vars

5 Translation of sRuby to C

This section describes how the different constructs of sRuby can be translated to C. One section is devoted to the crucial step in the translation process: type inference, ie. reconstructing the actual types of a Ruby program from the "untyped" Ruby source code.

The translation process has two modes: partial evaluation mode (PEM) and translation mode (TRM). The former is applied on all methods and declarations that do not depend on external (ie. not in the compiled program) data. The latter is the default mode and is applied whenever the compiler cannot be sure the statements are independent on external data. In PEM the mother VM evaluates the statements. When all statements have been evaluated the resulting objects are translated to C code.

There is a special feature that can be used to classes to execute code after PEM but before TRM. By using the directive 'call_after_pem_then_exclude' the

corresponding method will be called (once for each instance) and then excluded (undef:ed) before the instances are translated to C code. This can be used to generate methods and code based on setup information assembled during PEM.

An example making use of this feature is the `ObjectFormat` class. Instances of this class should have information about the format of all objects in the VM, ie. their representation, size etc. Since we do not want to write static `ObjectFormat` classes for all possible objects we require all classes to register their format to the `ObjectFormat` during PEM. The method `ObjectFormat#generate_trm_methods` will then use the information registered about the object formats to generate the code needed when running a VM. This way the final code can be specifically tailored to the actual objects we need make available in the VM.

A number of shortcuts are taken in this translation if there is only one instance of a class:

- Instance variables are mapped to globals.
- Methods do not have a reference to a self object but instead use the (global) instance vars directly.

These shortcuts/optimizations will give increased performance of the child VM since its objects need not pass around self references. The PEM mode also allows full Ruby syntax and semantics when setting up data structures needed during execution.

When PEM is finished and all `'call_after_pem_then_exclude'` methods have been called the compiler enters TRM. TRM is the default mode and the one described below. In the next section, the translation of different Ruby constructs are described. In the descriptions TRM is the default mode described. The differences in PEM are explicitly noted.

NOTE! We need a way to specify which of the PEM-translated classes are accessible at run-time. It should be possible to rename classes and methods and alias methods in this specification. Also possible to exclude methods from TRM translation, ie. they are only needed to setup objects during PEM.

5.1 Special sRuby constructs in SRubyLowLevel

Some support classes are used to ensure an efficient implementation of crucial data structures and resources. All these things are kept in a unique namespace `SRubyLowLevel`, that are handled in a special way by `srbc`: variables, classes and modules in `SRubyLowLevel` are not compiled but ignored by `srbc`. The `SRuby` constructs are instead mapped to pre-determined C constructs that efficiently gives the same behavior. This design makes it possible to have the Ruby code giving the exact same semantics as the pre-determined C constructs by giving its code in `SRuby`.

This section describes the contents of `SRuby` and how it is translated into C.

5.1.1 SRubyLowLevel::Memory - Access to memory

SRubyLowLevel::Memory gives access to the (hardware) memory of the computer. It has the following methods:

- [](address) - Read 32-bit value at 'address'
- []=(address, value) - Write 32-bit 'value' to 'address'
- byte(address) - Read 8-bit value at 'address'
- byte=(address, value) - Write 8-bit 'value' at 'address'

The translation is a direct translation to the respective macro (defined in sruby.h):

```
#define WORD(address) (*((int*)(address)))
#define SET_WORD(address,value) (*((int*)(address)) = value)
#define BYTE(address) (*((unsigned char *)(address)))
#define SET_BYTE(address,value) (*((unsigned char *)(address)) = value)
```

5.1.2 SRubyLowLevel::CStruct - Creating C struct's

To represent objects in a compact way we need to give sRuby programmers the ability of creating C structs. This is done in a way similar to the standard Ruby class Struct but in addition to specifying field names you can also specify the type of the field. The following types can be used:

- “unsigned long” - unsigned long
- “char” - char
- “unsigned char” - unsigned char
- sRuby class
- Other SRubyLowLevel::CStruct

Instead of simply giving a symbol when calling CStruct you give an Array with the symbol followed by one of the above types. To simplify access to bits in a CStruct you can also specify bit accessors. They are Arrays with two or three elements. If three elements its a bit field (with second and third element specifying the bit positions) and if two elements its a single bit. Single bit accessors with a question mark in the end of the name return true or false. If no question mark (and if a bit field) the accessor can take an optional arguments specifying that the value should be shifted or not. Default is to shift. An Example of these different cases can be found in the example below.

The interface to CStruct is basically the same as that for Struct. It has all the methods that Struct and the generated structs has but adds:

- Adds a new argument to the new method, of the created CStruct, which is the memory address pointed to. (created CStruct)
- address - return the address pointed to. (created CStruct)

Here's an example of creating a struct for the basic object format used in MRI and then using it:

```
ObjRepBasic = SRubyLowLevel::CStruct.new('ObjRepBasic', [:flags, 'unsigned
long', [:otype, 0, 5], [:mark?, 6], [:finalize, 7]], [:class, 'unsigned long'])
o = ObjRepBasic.new(12345)
puts o.address                               # -> 12345
o.flags = 0xc3
puts '#{o.otype}, #{o.mark?}'               # -> 3, true

o.otype = 0xff                               # Only 6 bits used
puts '#{o.otype}, #{o.mark?}'               # -> 127, true
o.mark = false
puts '#{o.otype}, #{o.mark?}'               # -> 127, false
puts '#{o.finalize}, #{o.finalize(false)}'  # -> 1, 128
```

which is translated to C code as (well not directly but after optimizations):

```
struct ObjRepBasic {
    unsigned long flags;
    unsigned long class;
};
#define OBJREP_BASIC_TYPE = 0x3f
#define OBJREP_BASIC_TYPE_STARTBIT = 0
#define OBJREP_BASIC_MARK = 0x40
#define OBJREP_BASIC_MARK_STARTBIT = 6
#define OBJREP_BASIC_FINALIZE = 0x80
#define OBJREP_BASIC_FINALIZE_STARTBIT = 7

ObjRepBasic* o = (ObjRepBasic*)12345;
printf('%d\n', 12345);
o->flags = 0xc3;
printf('%d, %s\n',
    ((o->flags&OBJREP_BASIC_TYPE)>>OBJREP_BASIC_TYPE_STARTBIT),
    INSPECT_BOOLEAN(((o->flags&OBJREP_BASIC_MARK)>>OBJREP_BASIC_MARK_STARTBIT)));
SETBITS(o->flags, 0xff, OBJREP_BASIC_TYPE)
```

which used some macros from sruby.h:

```
#define INSPECT_BOOLEAN(bool) ((bool)?'true':'false')
#define SETBITS(d, val, mask) (d = d&~mask
```

5.2 Translation of Ruby constructs

5.2.1 Classes

A class describes the properties of a set of objects. Objects have instance variables (ivars) and methods. The ivars are unique to each object but the methods are the same for all objects of the same class. We map the ivars to a struct in C and the methods directly to C functions.

5.3 Type inference

To generate efficient code we need to know the types of variables and arguments. We do not want the programmer to explicitly have to declare the types; it violates the principle that sRuby programs should be valid Ruby programs and it is also inelegant. The standard solution to this problem would be type inference. Type inference analyses the program structure and infers the types of variables and arguments. A number of authors have dealt with this problem for object-oriented programs, eg. [2].

We anticipate that a sRuby compiler will have to implement some of these type inference algorithms in the future. However, they are typically complex to implement (and thus error-prone) and take a long time to execute (someone mentioned about 20 minutes for 1000 line program but I guess it can't be that?). As a quick solution we can use the dynamical nature of Ruby to get the type information in a simpler way. We simply write a trace function that save a type signature for each call to a method. Then we execute the program for some typical data and get the type information. Some additional magic along these lines can be used to collect type info for data polymorphism (variables) in addition to the function polymorphism (args of methods). Here's some code:

```
def method_name(klass, id)
  name = klass.to_s
  if name.nil? then name = '' end
  if klass.kind_of? Class
    name += "#"
  else
    name += "."
  end
  name + id.id2name
end

trace_func = proc { |event, file, line, id, binding, klass|
  when "call", "c-call"
    # I'm unsure here but I'll guess the args are in the binding?
    types = args.each {|a| a.type}
    type_info[method_name(klass,id) + file + line.to_s] = types
  when return
    # Get arg from binding?
```

```

    type = arg.type
    return_type_info[method_name(klass,id) + file + line.to_s] = type
}

```

An alternative solution would be to use AspectR to attach the type saving code as pre and post advices.

Idea for getting args from binding: Get the line from the source file and grab the args between the paranthesis, split by comma and the evaluate them (arg1.type) in the context of the binding, gives you the type.

6 Optimizations when compiling sRuby

None so far. Ones we might introduce:

- Inlining of small (or often called) functions
- Paying attention to the generated C code so that it is optimization-friendly for the C compiler. I'm not sure how to do this but both the Pre-Scheme paper and Squeak papers mention they've done this [1] [?].

7 Benchmarking sRuby / Performance of compiled sRuby programs

Compare to MRI on some small benchmark programs. Maybe use the ones in the "Language Shoot-out"?

8 Future additions to sRuby/Ruby compiler

- Partial Evaluation of OO programs [3]
- Also check out the paper on specialization of OO simulations by wise 92 in cyberlib. They use symbolic objects during execution to specialize oo programs. Quite nice and probably matches well with Ruby's features. "Accelerating OO Simulation via Automatic Program Specialization" Daniel Weise and SCott Seligman 1992.

References

- [1] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 000–000, November 1997. Published as *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, volume 21, number 11.

- [2] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages, 1994.
- [3] Ulrik Schultz. Partial evaluation for class-based object-oriented languages.