DEGREE PROJECT 2004:PM08

Division of Computer Science at the Department of Informatics and Mathematics

RubyComp

A Ruby-to-LLVM Compiler Prototype

Anders Alexandersson

HÖGSKOLAN
TROLLHÄTTAN • UDDEVALLA

RubyComp - A Ruby-to-LLVM Compiler Prototype

Anders Alexandersson
Dept. of Computer Science
University of Trollhättan/Uddevalla
Sweden
anders.alexandersson@student.htu.se

Abstract

Dynamic programming languages are not generally pre-compiled, but are interpreted at run-time. This approach has some serious drawbacks, e.g. complex deployment, human readable source code not preserving the intellectual properties of the developers and no ability to do optimizations at compile-time or run-time.

In this paper we study the possibility to pre-compile the Ruby language, a dynamic object-oriented language, into Low Level Virtual Machine (LLVM) code for execution by the LLVM run-time, a compiler framework for lifelong optimization of an application. The result of the project is a Ruby compiler prototype, describing the infrastructure and overall design principles to map the highly dynamic properties of the Ruby language into low-level static constructs of the LLVM language.

The LLVM framework supports different hardware platforms, and by using LLVM as the target of compilation the benefits of that portability are gained.

1. Introduction

Software engineers have historically used models such as the waterfall model [18] to develop software, which were simple flows from specification through analysis, design, implementation and testing. These early models were very formal and it was easy to divide the work on different experts. The problem, however, was when a change was needed, since the models did not embrace alterations of the initial specification. Over time more flexible processes developed as e.g. the Unified Process (UP) [13], that contained processes specifically dealing with change, to better cope with customer updates and new requirements. In UP an initial specification is made, after which a prototype is developed and presented for the customer, who can give feedback, should something need to be changed. Other examples [20] of models embracing change are exploratory development, throw-away prototyping, incremental development, spiral development and recent arrivals like extreme programming.

This trend of development by prototyping created a need for programming languages that were easy and quick to use, in order to easily develop the prototype. Languages such as C/C++ and Java have a complex and sensitive syntax, where a small error generates compile-time failures. These languages are hard to work with when quick experiments are wanted, and are therefore not suitable for fast prototyping.

One answer to this problem are the dynamic languages, which do not require detailed specifications on variable types, return types etc. at the time of programming but are able to solve that automatically at run-time. The programmer simply creates the logic, and the language itself takes care of the syntactic details. However, this flexibility comes with a price - performance. In [4] it is shown that the dynamic object-oriented language Smalltalk had only 5-20% of the performance of C programs. The reason for this loss of performance is the fact that the program is interpreted at run-time, requiring extensive processor resources.

Therefore it would be desirable to be able to compile the dynamic programs to native code, or at least closer to it, to minimize the processor load of transferring the program from a high-level language into a low-level language at run-time. Then, the processor would primarily deal with executing the program itself and not with the overhead of interpretation.

One example of a relatively new dynamic programming language is Ruby [21]. It was developed by Yukihiro Matsumoto and publicly released in 1995, and has since then grown at an astounding rate in Japan. Its popularity can be explained by its integration of the power of object-orientation, the convenience of scripting languages, the simple and transparent syntax and the open source license [21].

In this paper we focus on emitting Low Level Virtual Machine (LLVM) code [16]. LLVM is a compiler framework for lifelong optimization of an application from initial compilation and optimization to analyses of execution on

the end-users machine with following dynamic reoptimizations and recompilations.

By using this framework the benefits of the continuous optimizations and the LLVM built-in support for different processor architectures are gained, currently x86 and SPARC V9 [17]. In this way the Ruby program is only needed to be compiled once into LLVM code, and the LLVM compiler framework takes care of the rest, down to the native code of a specific processor architecture, analogous to the Java Virtual Machine and its byte-code.

The goal of this paper is thus to study the feasibility of and present a design of a Ruby compiler prototype, that transforms Ruby source code into LLVM code executable by the LLVM run-time, at the same time preserving the dynamic properties of the Ruby language, including the possibility of updating the source code at run-time.

In section 4 the limitation of this project is defined, and in sections 5 and 6 the compiler- and run-time architectures is presented. In section 7 follows a discussion on supporting Ruby dynamics, and finally in sections 8-11 results, conclusions, suggestions for future work and overall discussion can be found.

2. Background

The term dynamic programming language can be defined as a programming language in which programs can change their structure as they run [23]. New functions or classes can be introduced, overriding or replacing old ones at run-time.

Much work has been made on the diverse issues of the compilation of dynamic languages. In [3] the Self (a language related to Smalltalk) compiler is described, and how to map the dynamic language Self to optimized native code using a collection of optimizing techniques such as type analysis, customization, splitting etc. The focus of this work is to analyze the program and replace complex structures, as e.g. messages and polymorphic methods, with simpler and thus faster constructs. The result is a performance of half that of a C program.

In [1] it is argued that the flexibility of the dynamic languages is good in exploratory programming, as in prototypes, but that the same dynamic properties limit the type-safety checking and optimizations at delivery time. This work deals with solving dynamic inheritance, primarily for Self, but the technique applies to other languages as well.

Further in [22], the issue of the bad performance of dynamic languages is discussed when making a method call to a class, which has to be recursively resolved at run-time by following the tree of inheritance. According to the authors, this is a common phenomena in many languages, also in static languages including C++, as not even then the target of a call is always known at compile-time. If the target of a particular call could be determined at compile-time, the cost would be no more than any regular function call. The authors in [22] present a fast and intuitive technique for generating compact selector-indexed dispatch tables, which boosts performance considerably.

In [2] a study of how the dynamic language Kawa, a dialect of Scheme [10], is compiled into Java byte code is described. The authors utilize e.g. the Java *ClassLoader* to dynamically load source code entered at run-time. In a related work [19] extensions to the Java Virtual Machine is suggested in order to add support for dynamic languages, and it is argued that by using Java as the target of compilation, practically any hardware in the world can be reached.

In this paper LLVM is the target of the Ruby compilation due to the hardware portability offered, and also partly due to the lifelong optimization techniques. The LLVM run-time uses a jitter to load code dynamically, and as the main difficulty of compiling Ruby is the fact that the program can be updated during run-time, this feature is a prerequisite.

As all classes in Ruby are open, i.e. extendable at any time, a user can add new methods to classes, replace methods or add new ones. Because of this, any part of the program has to be recompilable during run-time, which puts considerable demands on the compiler- and run-time architecture compared to a static language compiler. In this paper suggestions for such architectures, in collaboration with the LLVM framework architecture, are proposed in sections 5 and 6.

3. Methodology

In this project a step by step approach is used to be able to develop independent parts of the prototype separately. The details of these steps can be found in Appendix A. The overall process is to implement the first steps of Appendix A in LLVM code manually, which represent the core of the run-time architecture as described in later sections, after which the same code is created automatically.

Also, the architects of the LLVM framework have offered support on the LLVM developers mailing list [14], regarding the low-level details of the LLVM run-time, and the LLVM syntax as a whole.

4. Limitations

This project serves only as an overall proof of concept. Therefore, there is no support for the complete Ruby class library or non-trivial concepts as e.g. threads, files, garbage collection etc., but focus is on the core infrastructure of representing classes, function calls and inheritance, as described in Appendix A, with special focus on preserving the dynamic Ruby properties.

5. Design of Compiler Architecture

The overall focus regarding the compiler architecture is to describe a core design, supporting the most basic concepts as classes, method calls etc., at the same time being flexible enough to allow smooth future extensions.

Below follows a presentation of the actual implementation of the prototype [9], along with theoretical suggestions on how to realize topics not yet implemented. A more extensive theoretical discussion of future work can be found in later sections.

5.1. Overview

The compiler itself is completely written in Ruby. Initially, the compiler takes the Ruby source code and parses it into an abstract syntactic tree (AST). The resulting AST is then processed and a model of the LLVM program is created. When the AST has been completely processed, and the model is complete, the model is transformed into LLVM code by a simple mapping. An overview of the compiler architecture is seen in figure 1.

However, to be able to handle dynamic updates during run-time, the compiler has to be linked in together with the program being executed, and thus reside as LLVM representation during run-time as well as an initial stand-alone application.

The implementation of the integration of the compiler, as LLVM representation, is outside the scope of this project. Theoretically however, the compiler could be applied on itself generating the compiler LLVM representation, and linked into the compiled application. Also, the LLVM jitter could be integrated in the same binary, to further simplify deployment and eliminate the need for any LLVM installation on the user system. These integration topics are further discussed in later sections.

This paper presents an LLVM Modelling Layer (LML), which serves as a framework of classes that can be used when processing the AST, hiding the complexity of the LLVM syntax, and generates the LLVM representation automatically. An LLVM Abstraction Layer [8] is used in the modelling phase.

5.2. Parser

For generating ASTs for experimental purposes the Rockit parser [6] has been used.

However, the details of the parser do not affect the mechanisms in the translation process between the AST and the LLVM representation, and therefore the processing of the AST is not elaborated on here.

Thus, any parser can be used, although it is preferable that it is written in Ruby, in order to be able to apply the compiler on itself, generating the integrated compiler, as described earlier in section 5.1.

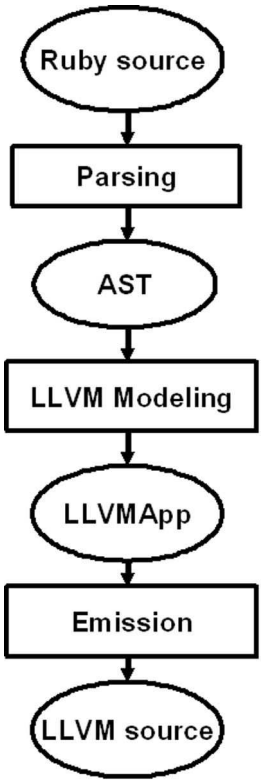

Figure 1. Compiler architecture.

If another language is used for the parser, an LLVM compiler for that language is needed. The architects behind the LLVM framework provide a complete C/C++ to LLVM compiler [17], which could be used.

5.3. LLVM Modelling Layer

The LLVM application model in the LML is built of classes representing the LLVM concepts, which can be mapped into a sequence of LLVM instructions.

Below is a summary of the currently implemented classes in the LML. A Class diagram of the most important classes can be found in Appendix B, and an emitted source code example in Appendix C.

1. LLVMApp

   The *LLVMApp* class represents the complete LLVM application. As an LLVM application has very much the same structure as a standard C program, the *LLVMApp* defines different sections for constants, function declarations, function definitions and main.

   These sections are defined by the use of *Sequence* classes [8] which are merely arrays of LLVM instruc-

tions. Instructions can be added to an *LLVMApp* section simply by calling methods on the desired *Sequence* class, passing the instruction as parameter.

This design offers complete control as to where in the LLVM application instructions are created, and more sections can be added, if needed in the future.

2. LLVMApp::addClass

   The *LLVMApp* class provides the method *addClass* that represents the addition of a class definition to an application.

```
llapp = LLVMApp.new
llapp.addClass(”myClass”)
```

   The naming mechanism in this prototype utilizes the fact that any string surrounded by double quotes can be used as a variable in LLVM [17]. These strings will be transformed into LLVM identifiers in the emission phase, and thus plain strings are used as LLVM variable names in the LML.

   An alternate way of defining an LLVM variable in the LML is by using the Ruby class *Symbol*, i.e. the syntax *:mySymbol* [21]. In the emission phase e.g. *:myVariable* will be transformed into the identifier *%myVariable*, which is the second way of representing a variable in LLVM. (This alternate way of representing variables is necessary, as external functions in the LLVM framework utilize that form. The *printf* C function [12] is e.g. accessed by the identifier *%printf*.)

3. LLVMApp::getClass

   To retrieve a certain class from the LLVM application in the LML, the method *getClass* is used, passing the desired class as parameter. The *LLVMClass* is returned.

4. LLVMClass

   The *LLVMClass* represents the concept of a class in LLVM. The class methods are represented as an array of *LLVMMethod*s. Also, further details are held in this class to support dynamic compilation, which is discussed in later sections.

5. LLVMClass::addMethod

   To add a method to a class in the LML the method *addMethod* is used, passing the *LLVMMethod* and its signature as parameters, which adds the *LLVMMethod* to the hash table of class methods in the *LLVMClass*.

   A hash table is used in order to be able to extract a specific method based on its signature, which is needed during dynamic redefinitions etc. of methods at runtime. See further discussion below regarding dynamic issues.

6. LLVMMethod

   The *LLVMMethod* class represents the concept of an LLVM function. It contains the return value, identifier, parameters and body of the function, and provides methods for processing those attributes. These methods are used later in order to instantiate the class of which the method is a member, as described below.

   Assuming that *Puts* is an inheritance of an *LLVMMethod*, with the specific properties of the *put string* function *puts* assigned to its instance variables, the following example illustrates how a method can be added to a class definition:

```
puts = Puts.new

llapp = LLVMApp.new
llapp.addClass(”MyClass”)
llapp.getClass(”MyClass”).addMethod(puts.IDsignature, puts)
```

   Note the use of the method *puts.IDsignature* to extract the LLVM representation signature of that method, which always will be a unique key as not two methods with identical signatures can co-exist in a class.

   Having a minimal class definition, an instance of that class can now be created by the *LLVMApp*.

7. LLVMApp::createClass

   The *createClass* method of the *LLVMApp* class is responsible for building the LLVM representation of a class and instantiate it with the class- and instance names passed as arguments:

```
llapp.createClass(”MyClass”, ”myInstance”)
```

   Still, the symbols ”MyClass” and ”myInstance” represent the LLVM identifiers of the class and instance respectively.

   Having an instance, a method call can now be made.

8. LLVMApp::call

   To call a method of a class in the LML, the *call* method of the *LLVMApp* is used, passing the return variable (optional), instance identifier, the method to be called and an array of parameters to that method:

```
llapp.call(”returnVariable”, ”myInstance”, puts, [”Hello World”] )
```

In this example one parameter only is passed, but the general form is [arg1, arg2,..., argN].

If the method returns *void* or the return value can be ignored, *nil* can be passed:

```
llapp.call(nil, ”myInstance”, puts, [”Hello World”] )
```

By passing *nil*, the compiler architecture instructs the LLVM run-time to create an unnamed variable automatically, that receives the value. This value can simply be ignored.

The return variable identifier can be used in later calls to other methods.

9. LLVMClass::addSuperClass

   The prototype supports inheritance by the use of the *addSuperClass* method. By calling this method and passing the superclass as parameter, the superclass will be added to the class.

```
llapp.getClass(”MyClass”).addSuperClass(”MySuperClass”)
```

   If a method is overridden by the class currently being created (i.e. the class has an *addMethod* call with a method of the same signature) the superclass version of that method will not be used, but the overridden version will be used, as specified by the Ruby inheritance semantics.

10. LLVMApp::shutdown

    The *shutdown* method of the *LLVMApp* is simply responsible for cleaning up any allocated memory and finalizing the main method, completing the application.

11. LLVMApp#to_llvm

    After the call to the *shutdown* method, the job of the LML is finished, and a final call to the method *to_llvm* of the *LLVMApp* is issued, which assembles the complete LLVM code of the LLVM application, as defined in the different *Sequences* described above, and can be written to file.

The complete model for creating a minimal LLVM application is, then, created as follows:

```
puts = Puts.new

llapp = LLVMApp.new
llapp.addClass(”MySuperClass”)
llapp.getClass(”MySuperClass”).addMethod(puts.IDsignature, puts)

llapp.addClass(”MySubClass”)
llapp.getClass(”MySubClass”).addSuperClass(”MySuperClass”)
llapp.createClass(”MySubClass”, ”mySubInstance”)
llapp.call(nil, ”mySubInstance”, puts, [”Hello World”] )

llapp.shutdown

f = File.new(”hello_world.ll”, ”w+”)
    f.puts llapp.to_llvm
f.close
```

The current prototype supports the addition of any number of classes, containing any number of methods, having any number of parameters. Presently only the *put string* method *puts* is implemented, which can be called with any string, any number of times.

5.4. Extending the LLVM Modelling Layer

To extend the prototype with support for more Ruby methods, all that is needed is to implement derivatives of the *LLVMMethod* class, which contains all mechanisms needed in order to integrate the method into the system architecture.

The task of implementing the LLVM body of that method is further eased by the fact that the C front-end [17] created by the LLVM architects can be used to obtain the correct LLVM code for built-in core functions. The corresponding C function can be implemented in a standard C program, after which the front-end can compile that C program into an LLVM program, which in turn can be disassembled using the llvm-dis tool [17] and analyzed.

Using this technique e.g. the *printf* C function is shown to be an external function accessible in the LLVM run-time through the LLVM pointer *%printf(sbyte*, ...)* and simply used in a standard call:

```
%result = int %printf(sbyte* %string)
```

In the same way e.g. the *getchar()* C function is shown to be *%result = int %getchar(...)*.

By systematically using this technique, any function supported by the C/C++ language and which has logical equivalence in Ruby can be implemented smoothly.

To further add support for other Ruby concepts, as e.g. class instance variables, modules, etc. new methods can be implemented in the *LLVMApp* or its member classes. The support for adding an instance variable can e.g. be a member of the *LLVMClass* called *addInstanceVariable*.

6. Run-Time Architecture

6.1. Overview

At run-time the original classes, functions etc. of the Ruby program are represented by the building blocks of LLVM [17], as described below.

First the concept of class representation is described, then the *map* concept, kernel concept, inheritance and finally run-time meta-data.

6.2. Classes

Classes are represented as LLVM *structs* containing a pointer to a *map*, which holds the class methods. The *struct* also supports the possibility to implement variables representing the instance variables of any type, although this is outside the scope of this project, as specified by Appendix A.

This design provides the possibility to update the methods without affecting the instance variables.

6.3. Maps

A *map* is a *struct* containing function pointers to functions representing the class methods. The *kernel* (see next section) functions have fixed positions in the map, to simplify dynamic updates of redefined functions. The number of functions a specific map holds is determined at the initial compilation, as each *LLVMClass* knows how many methods it contains (see section 5.3, point 4). In this initial version of the prototype, all instances have their own map. This can in the future be improved to a design where all instances of a class share the same map.

By separating the methods and the class with a *map*, the possibility opens up to add new methods to a class at run-time without loosing any instance variables. A new *map* is simply allocated and initialized with the current number of methods (*struct* holds one additional pointer), the old one discarded and the pointer from the class *struct* is updated to point to the new *map*. The instance variables are held in the class *struct*, which is unchanged.

The complete class representation in LLVM can be seen in figure 2.

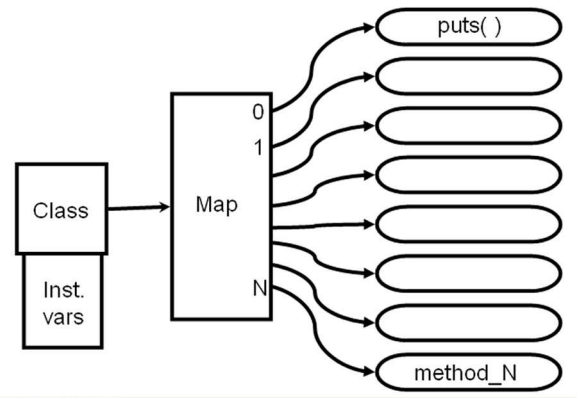

Figure 2. Run-time architecture

6.4. Run-Time kernel

The run-time architecture contains the concept of the *kernel*, which is a special class that holds the built-in basic functions of Ruby, as e.g. basic I/O operations etc. It is created before any user classes.

In this project the *kernel* is limited to the *put string* method *puts* only, but can be extended in the future, as said in 5.4.

6.5. Inheritance

With [22] in mind, inheritance is hard-coded in this prototype, i.e. there are no pointers to a superclass in the run-time architecture. The methods and other properties of the superclass are identified by the compiler and direct pointers to those functions are created in the *map*. Any instance variables can also be copied into the *struct*. In this way no pointer traversing is needed, which is a slow mechanism used in e.g. C/C++ [22]. Any dynamic or polymorphic behavior is handled by the meta-data and the dynamic compilation as described in the next section.

6.6. Run-Time Metadata

As inheritance, function pointers etc. are statically coded in the run-time architecture, metadata describing these relationships is needed in order to have control over the system state. If e.g. a method in a superclass is redefined, in the source code or at run-time, all existing (and future) classes and sub-classes of that class are expected to use the new method instantly.

Such information is specified automatically in the LML, as the classes therein contain this state information.

In the case of a redefinition in the source code the LML can be processed directly residing as Ruby classes, and the appropriate LLVM code be generated to implement the new method and update all affected pointers in all affected *maps*.

In the case of a redefinition at run-time by user input, the LLVM representation of the LML, generated by the future complete version of the compiler, applied on itself, constitutes the run-time metadata. New source code read from the user, can be parsed, compiled and loaded as discussed in the next section, and the LLVM version of the LML be processed as above to update the system state.

7. Supporting Ruby Dynamics

The below discussion is an outline of possible solutions that need to be elaborated on and further refined in future work.

7.1. Class dynamics

The source code dynamics, i.e. source code updates specified in the original source code file, can be handled by incorporating meta-data in the LML classes. If a method of a class is redefined, all *maps* of all instances of that class have to be updated to point to the new method.

To implement this mechanism, an LML method:

```
LLVMClass::redefine(oldMethod, newMethod)
```

is suggested, passing the *LLVMMethod* instances as parameters.

```
llapp.getClass("myClass").refdefine(puts, new_puts)
```

This method can be designed analogous to the *LLVMApp::createClass* method (see generated code example in Appendix C), which takes a class instance and assigns pointers to the corresponding *map*. The *redefine* method can also take a class instance, but *reassign* the pointer of a specific *map* position, specified in the *LLVMMethod*s passed (all *LLVMMethods* know their own position in a *map*).

After the method pointer in the *map* is updated, all currently created instances of that class have to be updated as well. As the *redefine* method will generate an LLVM function analogous to *createClass* which updates one specific position in a map, that function can be reused for all instances of the updated class.

By creating a property in the *LLVMClass* holding all identifiers of all created instances of that specific *LLVMClass* at the time of creation in *createClass*, a class will always know all of its instances. By processing this list and generating successive calls to the LLVM function above, all current instances of an updated class can also be updated.

7.2. Subclass dynamics

A method can also be inherited, and if a superclass is updated, all subclasses have to be updated too. As methods are statically linked in this prototype, there are no pointers to the superclass. Therefore meta-data is again needed, to keep track of which methods are inherited, and need updating.

A suggestion for solving this is to add a property to the *LLVMMethod*, which holds the identifier of the *LLVMClass* which implements that method. When the method is inherited, which occurs in *createClass*, that property can be extracted and used to retrieve the implementing class. This is as earlier described made by the *LLVMApp::getClass* method. By adding another property, analogous to the list of instances described above, to the *LLVMClass* which describes all subclasses that uses specific methods of that LLVMClass, these subclasses can be updated too.

The data structure for describing these subclass relationships is suggested to be a hash table, having the signature of the method being updated (contained in the *LLVMMethod* itself) as the key and an array of subclasses using that method as value.

By using the method signature, which is unique in a specific class, all possible subclass-method combinations can be specified unambiguously.

As information is specified regarding all instances of a specific class (direct instances and inherited instances) it is possible to update the entire system state to reflect the dynamic update.

7.3. Run-Time input dynamics

Regarding the user input dynamics, it is assumed that the compiler can be completed, and is capable of compiling the compiler source code itself. It is also assumed that the resulting LLVM representation of the compiler can be integrated into the application binary together with the LLVM jitter, to create a stand-alone compiler-jitter-application system, which is a prerequisite in order to be able to parse and compile user input.

If a user updates the source code during execution, the complete program representation needs to be recompiled to reflect those changes. If e.g. a class method is redefined, that method has to be parsed and compiled by the built-in compiler and loaded, after which all pointers in all *maps* to that method need to be updated.

The LLVM compiler framework provides a function *ParseAssemblyFile* which takes a file of human readable LLVM code, and returns a pointer to an newly created LLVM Module [17]. Provided that this function can be compiled into LLVM, which future work will disclose, it can be disassembled using the llvm-dis tool in the LLVM framework into human readable form, and included as a source code function in the LLVM representation of the compiler.

The LLVM compiler framework also provides a function:

```
ExecutionEngine::getPointerToGlobal(Function*)
```

which takes a pointer to a byte code LLVM representation of an LLVM function, loads that function and returns an LLVM pointer to that loaded function, which can be used to call the function.

The first problem is, however, that the *getPointerToGlobal* function resides in an executing component in the LLVM run-time jitter, i.e. the *ExecutionEngine*, and to be able to call the *getPointerToGlobal* function, a pointer to the *ExecutionEngine* is needed. This is according to the LLVM

architects not currently supported by the LLVM framework, and therefore modifications of the framework are needed.

The second problem is that the *getPointerToGlobal* only handles complete LLVM modules [17] and not separate functions. Running the *getPointerToGlobal* on a separate function would, according to the LLVM architects, create linking problems when resolving external function calls and types, as the *getPointerToGlobal* works with mapping of memory addresses, which is not possible over different modules.

The first problem can according to the LLVM architects partly be solved by simply adding a method to the *ExecutionEngine* that returns the instance pointer, i.e. the *this* pointer, which can be used to call the *getPointerToGlobal* function. The part not solved, however, is that the *ExecutionEngine* resides as a compiled C++ program, which means that the calls from the LLVM representation of the compiler need to be able to call C++ functions. According to the LLVM architects, there is currently no support for doing this either, and further modifications are thus needed.

The second problem could possibly be solved by integrating the function into an LLVM module first. However, the LLVM architects suggest a different approach altogether, which involves writing the Ruby compiler in C++ instead, making it possible to interact seamlessly with the LLVM run-time. A deeper discussion of this issue can be found in section 9.

Assuming though, that these interaction details can be solved, a user-defined function entered at run-time can be compiled, loaded and its properties analyzed after which pointer updates or recompilation of *map*s as described in 7.1 and 7.2 can be made, depending on whether the function is a redefinition or a new one.

This approach of dynamically recompiling all classes when a redefinition occurs at run-time, puts some demand on processor resources. This kind of changes are however likely to be rare, and thus the overall performance over time is likely to remain high.

8. Result

The current prototype implements:

1. Specification of classes
2. Addition of methods to those classes of any signature
3. Instantiation of classes
4. Inheritance
5. Calls to methods, with any number of arguments of any type
6. Memory management, freeing any allocated memory

Currently only the *puts* method is implemented through the *Puts* class which inherits from *LLVMMethod*. This inheritance is the mechanism for adding any new methods, and once created in this way they are automatically integrated in the system architecture, and can be added by the *addMethod* call.

Apart from the above implementation, the prototype is prepared for implementation of dynamic behavior as follows:

1. The run-time architecture supports dynamic updates in original source code by the use of the metadata in the LML. In this way direct pointers can be used.

2. The run-time architecture supports dynamic updates during run-time, provided that the interaction with the LLVM run-time jitter can be solved, either by modifying the open source LLVM framework, or by using C++ instead of Ruby for the compiler.

Currently, the compiler prototype does not contain any parser, and thus does not process any AST. The prototype however does provide the LML classes needed in order to manually implement the example programs 1,2 and 3 in Appendix A.

Regarding example program 4, the compiler- and run-time architectures are prepared for it and can be extended to support it in the future, by simply implementing the *redefine* method as outlined in 7.1.

9. Discussion

The main problem identified by this paper is the interaction with the LLVM run-time, in order to load new source code dynamically.

Two solutions are proposed:

1. Modify the LLVM framework, to still be able to use Ruby as the language of the compiler.

2. Write the compiler in C++ instead to integrate seamlessly with the LLVM run-time.

First of all, it is desirable to keep the complexity as low as possible in the compiler. In the case of using Ruby, only two languages are used: Ruby for the compiler and source code, and emitted LLVM code. If C++ is used, three languages are used: Ruby source code, C++ compiler code and emitted LLVM code. If three languages are to be mastered, it is likely that fewer people can work on the compiler, as an open source project, compared to the two-language case.

Secondly Ruby is more flexible than C++, which makes it easier to experiment with the prototype and improve it.

The *PyPy* project [11] is an example of this, where the dynamic language *Python* is implemented in itself, creating code that is easier to understand due to the high level of abstraction, compactness and modularity, compared to C.

Further, in the *RubyVM* project [7] the construction of a Ruby virtual machine written in sRuby, a static dialect of Ruby, is outlined where it is argued that the resulting code would be easier to debug, analyze, and change.

The advantages of using C++ for the compiler are on the one hand the easier integration with the LLVM run-time and on the other hand better performance. The integration issue can however be solved as the LLVM framework is an open source project, and the code can be modified to fit the needs of the Ruby compiler.

Also, regarding the performance aspect, it is not a problem that the compiler is slow, since dynamic updates are likely to be rare in the context of the execution lifetime. In the initial compilation, performance is even less of a problem, as it is done one time. Furthermore, the LLVM representation of the compiler undergoes continuous optimizations [16], and by the use of LLVM as target, performance is boosted automatically throughout the compiler lifetime.

Thus, as the current trend in software development aims at rapid prototyping [13, 5] the choice of Ruby as the language for the compiler is still suggested in this paper, with its support for flexibility, modularity and extensive ease of use.

10. Conclusion

The core of the prototype design is to handle dynamic behavior by recompilation from one static state to another, by the use of meta-data. Between recompilations static pointers are used, which enables good performance.

The prototype contains an LLVM Modelling Layer, hiding the LLVM syntax which is used to process the AST.

Some important obstacles have been identified in the LLVM framework, preventing support for full dynamic behavior, and to solve this, the framework needs modifications.

As a whole and provided that the LLVM framework can be modified to fit the needs of the Ruby dynamic requirements, compiling Ruby to LLVM is indeed feasible.

11. Future work

The first thing suggested to be done in the future, is to solve the interaction problems between the compiler and the LLVM run-time. The following source code [15] needs to be analyzed ($R local repository folder):

1. $R/llvm/tools/llvm-as/llvm-as.cpp

   How can the method *ParseAssemblyFile(InputFilename)* be modified to take a pointer to a string in memory representing an LLVM function instead of a file on disk representing a complete LLVM module [17]? How can it then be modified to return a *Function** for use in *ExecutionEngine::getPointerToGlobal(Function*)* instead of a *Module**?

2. $R/llvm/lib/ExecutionEngine/ExecutionEngine.cpp

   How can the *ExecutionEngine* be modified to return a pointer to itself? Suggested solution by the LLVM architects is to return the *this* pointer of the instantiated class.

Furthermore, how can calls from LLVM code be made to C++ objects running in memory?

Once this interaction is solved guaranteeing support for dynamic updates at run-time, more Ruby concepts are suggested to be implemented, as e.g. modules, threads, garbage collection etc.

Finally the current prototype emits human-readable LLVM code which has to be assembled into binary LLVM code by the *llvm-as* tool. Future work thus also involves how to emit LLVM binary byte code directly. A suggestion from the LLVM architects is to study the *ParseAssemblyFile(InputFilename)*, and the llvm-as.cpp in general.

12. Acknowledgements

I would like to thank my examiner Robert Feldt and my supervisor Richard Torkar for excellent guidance and help, navigating a for me new field.

I also deeply thank Chris Lattner and Misha Brukman for the help and support regarding the LLVM syntax and the LLVM framework architecture.

References

[1] O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. *Lecture Notes in Computer Science*, 707:247–??, 1993.

[2] P. Bothner. Kawa—compiling dynamic languages to the Java VM. In *Proceedings of the USENIX 1998 Technical Conference, FREENIX Track*, New Orleans, LA, 1998. USENIX Association.

[3] C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph.d. diss, Stanford University, 1992.

[4] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 150–164, 1990.

[5] L. DeRose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. An environment for the rapid prototyping and development of numerical programs and libraries for scientific computation, 1994.

[6] R. Feldt. Rockit parser. `http://rockit.sf.net`, 2004.

[7] R. Feldt. Rubyvm - a library of virtual machine components for executing ruby programs written in ruby. `http://www.ce.chalmers.se/~feldt/ruby/ideas/rubyvm/rubyvm.pdf`, 2004.

[8] R. Feldt and A. Alexandersson. Rubycomp backend svn repository. `http://www.pronovomundo.com/open_svn/llvm`, 2004.

[9] R. Feldt and A. Alexandersson. Rubycomp svn repository. `http://www.pronovomundo.com/open_svn/rubycomp`, 2004.

[10] J. D. Guttman, J. D. Ramsdell, and M. Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.

[11] J. Hallen. Pypy - implementing python in python. `http://www.python.org/pycon/dc2004/papers/27/`, 2004.

[12] E. Huss. The c library reference guide. `http://www.acm.uiuc.edu/webmonkeys/book/c_guide/`, 2004.

[13] C. Larman. *Applying UML and Patterns*. Prentice Hall, 2nd edition, 2001.

[14] C. Lattner. Llvm developers mailing list. `http://mail.cs.uiuc.edu/mailman/listinfo/llvmdev`, 2004.

[15] C. Lattner. Llvm framework cvs repository. `http://llvm.cs.uiuc.edu/releases/register.html`, 2004.

[16] C. Lattner and V. Adve. Code generation and optimization, 2004. cgo 2004. international symposium on, vol., iss., 20-24 march 2004. pages 75–86, 2004.

[17] C. Lattner and V. Adve. Llvm language reference manual. `http://llvm.cs.uiuc.edu/docs/LangRef.html`, 2004.

[18] W. W. Royce. Managing the development of large software systems. In *Proceedings WESCON*, August 1970.

[19] O. Shivers. Supporting dynamic languages on the java virtual machine. Technical Report AIM-1576, 1996.

[20] I. Sommerville. *Software Engineering*. Pearson Education Limited, 2nd edition, 2001.

[21] D. Thomas and A. Hunt. Programming ruby - the pragmatic programmer's guide. `http://rubycentral.com/book/index.html`, 2004.

[22] J. Vitek and R.N. Horspool. Taming message passing: Efficient method look-up for dynamically typed languages. *Lecture Notes in Computer Science*, 821:432–??, 1994.

[23] Wikipedia. Dynamic programming language. `http://en.wikipedia.org/wiki/Dynamic_language`, 2004.

Appendix

A. Detailed Steps in Project

A.1. Step 1

Method calls using strings, to standard output. Ex. puts ”Hello world!”

A.2. Class definitions

Class definitions, using method calls from step 1.

```
class C
  def m1
    puts ”Hello world!”
  end
end

c = C.new
c.m1
```

Output:
Hello world!

A.3. Inheritance

```
class Super
  def s1
    puts ”s1!”
  end
end

class C < Super
  def m1
    puts ”m1!”
  end
end

c = C.New
c.s1
c.m1
```

Output:
s1!
m1!

A.4. Redefining methods

Redefining a method with a new one, or overloading with a new version.

```
class C
  def m1
```

```
    puts ”Hello world!”
  end
end

c = C.new
c.m1

class C
  def m1
    puts ”Redefined m1!”
  end
end

c.m1
```

Output:

```
Hello world!
Redefined m1!
```

B. LLVM Modelling Layer Class diagram

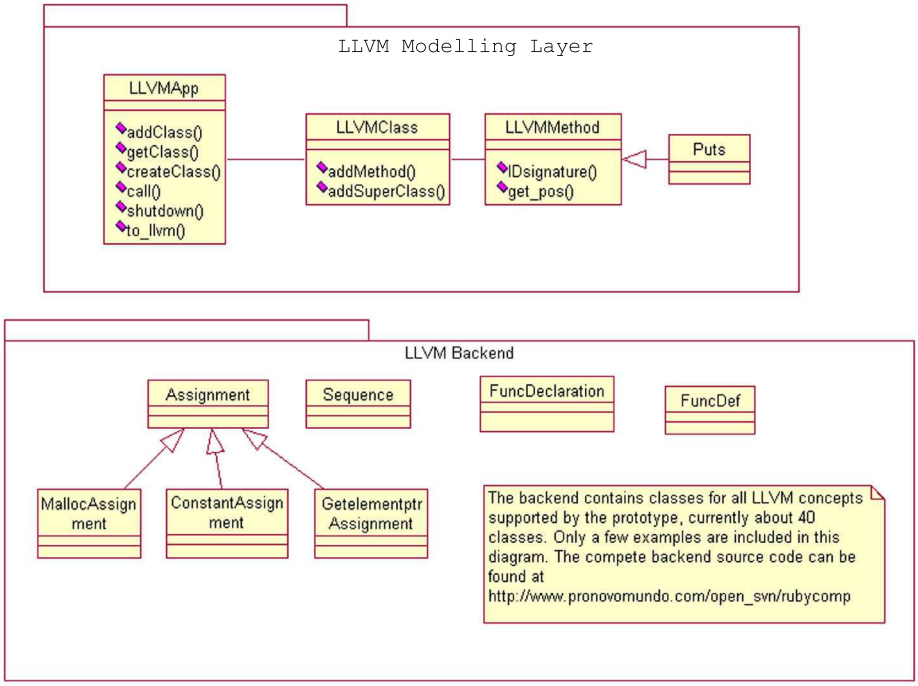

Figure 3. Compiler Class Diagram

C. Emitted Source Code Example

C.1. LML instructions

```
puts_kernel = Puts.new

llapp = LLVMApp.new
llapp.addClass(”Kernel”)
llapp.getClass(”Kernel”).addMethod(puts_kernel.IDsignature, puts_kernel)
llapp.createClass(”Kernel”, ”myKernel”)
llapp.call(nil, ”myKernel”, puts_kernel, [”Kernel: Hello Kernel”] )

llapp.addClass(”Subclass”)
llapp.getClass(”Subclass”).addSuperClass(”Kernel”)
llapp.createClass(”Subclass”, ”mySubclass”)
llapp.call(nil, ”mySubclass”, puts_kernel, [”Subclass: Hello World”] )

llapp.shutdown

f = File.new(”hello.ll”, ”w+”)
    f.puts llapp.to_llvm
f.close
```

C.2. LLVM instructions

```
    Contents of hello.ll, with added comments

; Hard coded strings, represented as constants
”Kernel: Hello KernelConst” = internal constant [22 x sbyte ] c”Kernel: Hello Kernel\0A\00”
”Subclass: Hello WorldConst” = internal constant [23 x sbyte ] c”Subclass: Hello World\0A\00”

; Class definition of Kernel
”myKernelMap” = type {int (sbyte*)*}

    ”Kernel” = type {”myKernelMap”*}

; Declaration of external function in the LLVM framework
declare int %printf(sbyte*, ...)

; Class definition of Subclass
”mySubclassMap” = type {int (sbyte*)*}
”Subclass” = type {”mySubclassMap”*}

; Function definition
int %puts_kernel(sbyte* %string)
{
    %tmp0 = call int (sbyte*, ...)* %printf(sbyte* %string)
    ret int 0
}

; Create the class in memory
”Kernel”* ”createKernel”()
{
    ; Allocate the map of the class
    ”myKernelMapInstance” = malloc ”myKernelMap”
```

```
    ; Allocate the class
    ”myKernel” = malloc ”Kernel”

    ; Get a pointer to the allocated map
    ”myKernelMapPTR” = getelementptr ”Kernel”* ”myKernel”, long 0, ubyte 0

    ; Store the pointer value of the map pointer in the class, connecting them
    store ”myKernelMap”* ”myKernelMapInstance”, ”myKernelMap”** ”myKernelMapPTR”

    ; Get a pointer to the function at position 0, the puts function
    ”puts_kernelPTRPTRMAP” = getelementptr ”Kernel”* ”myKernel”, long 0, ubyte 0
    ”puts_kernelPTRMAP” = load ”myKernelMap”** ”puts_kernelPTRPTRMAP”
    ”puts_kernelPTR” = getelementptr ”myKernelMap”* ”puts_kernelPTRMAP”, long 0, ubyte 0

    ; Store the poiner value of the %puts_kernel function
    store int (sbyte*)* %puts_kernel, int (sbyte*)** ”puts_kernelPTR”

    ; Class is now created, return the pointer to it
    ret ”Kernel”* ”myKernel”
}

; Function for calling the puts_kernel function of myKernel
int ”callmyKernelputs_kernel”(sbyte* ”Kernel: Hello Kernel”,”Kernel”* ”myKernel”)
{
    ; Get a pointer to the %puts_kernel function via the myKernel pointer
    ”PTRMAP” = getelementptr ”Kernel”* ”myKernel”, long 0, ubyte 0
    ”MAP” = load ”myKernelMap”** ”PTRMAP”
    ”puts_kernelPTR” = getelementptr ”myKernelMap”* ”MAP”, long 0, ubyte 0

    ; Load the pointer value of the %puts_kernel function
    %tmp = load int (sbyte*)** ”puts_kernelPTR”

    ; Call %puts_kernel
    ”tmp_result” = call int (sbyte*)* %tmp(sbyte* ”Kernel: Hello Kernel”)

    ; Return result
    ret int ”tmp_result”
}

; Create SubClass. Static pointers are assigned, inheritance controled by meta data
”Subclass”* ”createSubclass”()
{
    ”mySubclassMapInstance” = malloc ”mySubclassMap”
    ”mySubclass” = malloc ”Subclass”
    ”mySubclassMapPTR” = getelementptr ”Subclass”* ”mySubclass”, long 0, ubyte 0
    store ”mySubclassMap”* ”mySubclassMapInstance”, ”mySubclassMap”** ”mySubclassMapPTR”
    ”puts_kernelPTRPTRMAP” = getelementptr ”Subclass”* ”mySubclass”, long 0, ubyte 0
    ”puts_kernelPTRMAP” = load ”mySubclassMap”** ”puts_kernelPTRPTRMAP”
    ”puts_kernelPTR” = getelementptr ”mySubclassMap”* ”puts_kernelPTRMAP”, long 0, ubyte 0
    store int (sbyte*)* %puts_kernel, int (sbyte*)** ”puts_kernelPTR”
    ret ”Subclass”* ”mySubclass”
}

; Function for calling the puts_kernel function of mySubclass
int ”callmySubclassputs_kernel”(sbyte* ”Subclass: Hello World”,”Subclass”* ”mySubclass”)
{
    ”PTRMAP” = getelementptr ”Subclass”* ”mySubclass”, long 0, ubyte 0
```

```
    ”MAP” = load ”mySubclassMap”** ”PTRMAP”

    ”puts_kernelPTR” = getelementptr ”mySubclassMap”* ”MAP”, long 0, ubyte 0
    %tmp = load int (sbyte*)** ”puts_kernelPTR”
    ”tmp_result” = call int (sbyte*)* %tmp(sbyte* ”Subclass: Hello World”)

    ret int ”tmp_result”
}

; — Main. Start of program. —
int %main()
{
    ; Create myKernel
    ”myKernel” = call ”Kernel”* ”createKernel”()

    ; Get a pointer to the string constant
    ”Kernel: Hello Kernel” = getelementptr [22 x sbyte]* ”Kernel: Hello KernelConst”, long 0, long 0

    ; Call %puts_kernel of myKernel
    call int ”callmyKernelputs_kernel”(sbyte* ”Kernel: Hello Kernel”,”Kernel”* ”myKernel”)

    ; Create mySubclass
    ”mySubclass” = call ”Subclass”* ”createSubclass”()

    ; Get a pointer to the string constant
    ”Subclass: Hello World” = getelementptr [23 x sbyte]* ”Subclass: Hello WorldConst”, long 0, long 0

    ; Call %puts_kernel of mySubclass
    call int ”callmySubclassputs_kernel”(sbyte* ”Subclass: Hello World”,”Subclass”* ”mySubclass”)

    ; — Free allocated memory —

    ; Destroy map of myKernel first
    ”myKernel_MapPTRPTR” = getelementptr ”Kernel”* ”myKernel”, long 0, ubyte 0
    ”myKernel_MapPTR” = load ”myKernelMap”** ”myKernel_MapPTRPTR”
    free ”myKernelMap”* ”myKernel_MapPTR”

    ; Destroy myKernel
    free ”Kernel”* ”myKernel”

    ; Destroy map of mySubclass first
    ”mySubclass_MapPTRPTR” = getelementptr ”Subclass”* ”mySubclass”, long 0, ubyte 0
    ”mySubclass_MapPTR” = load ”mySubclassMap”** ”mySubclass_MapPTRPTR”
    free ”mySubclassMap”* ”mySubclass_MapPTR”

    ; Destroy mySubclass
    free ”Subclass”* ”mySubclass”

    ; Exit program
    ret int 0
}
```