Compiling Scala to LLVM

Geoffrey Reedy
The University of New Mexico
greedy@cs.unm.edu

Abstract

This paper describes ongoing work to implement a new backend for the Scala compiler that targets the *Low Level Virtual Machine (LLVM)*. LLVM aims to provide a universal intermediate representation for compilers to target and a framework for program transformations and analyses. LLVM also provides facilities for ahead-of-time and just-in-time native code generation. Targeting LLVM allows us to take advantage of this framework to compile Scala source code to optimized native executables. We discuss the design and implementation of our backend. We also outline the additional work needed to produce a robust backend.

Keywords LLVM, Language Implementation, Virtual Machines, Code Generation, Compilers

1. Introduction

In this paper we present a new backend for the Scala compiler with targets the *Low Level Virtual Machine (LLVM)*. At its core LLVM provides a type-safe universal assembly language designed to be targeted by high-level compilers. LLVM also supplies a framework for life-long program analysis, optimizations and just-in-time compilation of programs written in for LLVM. LLVM is an attractive target because it can handle low level optimizations and native code generation. Language specific optimizations can also be written using the LLVM API.

The Scala compiler currently targets the *Java Virtual Machine (JVM)* and the *Common Language Infrastructure (CLI)*. Both the JVM and the CLI are managed runtimes for class based object oriented programming languages. They provide runtime features such as garbage collection, method dispatch, thread-based concurrency, and rich libraries. LLVM, true to its name, does not provide these features. Instead it supplies what a language implementer needs to build exactly those facilities that are needed by the language.

Because LLVM can generate standalone native code it does not require a virtual machine on deployment targets. This could lead to Scala being viable for writing programs for platforms where a JVM is not available including embedded systems and Apple's iOS platform. Scala programs compiled to LLVM would also not suffer from the long startup time required for the JVM making Scala a more attractive choice for system programming and scripting.

Scala on LLVM can also provide a platform for experimentation in implementation techniques for a hybrid functional/object oriented language.

2. The Scala Compiler

Scala's compiler is built on a powerful and flexible framework that allows the compiler to be assembled from separate components each implementing different aspects of compilation. The target platform for code generation is one of these components. This section gives an overview of the architecture of Scala's compiler as it relates to creating a new backend, a full treatment of this architecture can be found in Odersky's 2005 paper *Scalable Component Abstractions*[10].

2.1 Compiler Phases

The compilation process is implemented as a collection of phases that are called for each compilation unit. Each phase declares which other phases it depends on and the compiler runs the phases in an order satisfying these ordering constraints.

The first phase parses each source file into a program tree. The following phases enter symbols to the compiler's symbol table and annotate program trees with types. Subsequent phases lower, desugar and perform other type-directed operations on the typed trees.

Code for the target platform is not generated directly from these trees. Instead one of the later compiler phases translate the typed trees into an intermediate representation called ICode. There are a few phases that perform optimizations by rewriting ICode. The final phases generate code for the target platform from the ICode.

2.2 ICode

ICode is a stack based intermediate representation for Scala programs. The top level entities in ICode are classes. Classes in ICode have fields and methods as members. Both fields and methods refer to a symbol in the compiler's symbol table which holds the name and type information for the member. Besides the symbol, fields contain no extra information; methods however contain a list of local variables (including parameters) and a list of basic blocks containing the code for the method.

Basic blocks in ICode have a single entry and normal exit point. However, execution may also leave a basic block prematurely when an exception is raised. Each method contains a (possibly empty) list of the exception handlers for the code in the method. An exception handler may cover one or more basic blocks. The handlers are stored in order from innermost to outermost so that iteration over the handlers for a given block will yield the handlers in the order that they should be applied.

The execution model for ICode is that of a stack based machine with a single operand stack. Each instruction consumes its arguments from and produces its results on this stack. The semantics of ICode instructions are quite similar to that of Java bytecode. As an example the Scala source and resulting ICode for a factorial function is shown in Figure 1.

ICode instructions are annotated with enough type information that the types of values consumed from and pushed onto the operand stack can be precisely determined. This allows verification that optimizations and other ICode transformations preserve type-safety. It is important to note, however, that in ICode type parameters have been erased. This design choice seems to have been driven by the fact that the first targets for code generation did not support parameterized types.

Scala source for `fact`

```
def fact(n: Int): Int = {
  if (n == 0) 1 else n * fact(n-1)
}
```

ICode for `fact`

```
def fact(n: Int (INT)): Int {
  locals: value n
  startBlock: 1
  blocks: [1,2,3,4]

  1:
    LOAD_LOCAL(value n)
    CONSTANT(0)
    CJUMP (INT)EQ ? 2 : 3

  2:
    CONSTANT(1)
    JUMP 4

  3:
    LOAD_LOCAL(value n)
    THIS(fact)
    LOAD_LOCAL(value n)
    CONSTANT(1)
    CALL_PRIMITIVE(Arithmetic(SUB,INT))
    CALL_METHOD fact.fact (dynamic)
    CALL_PRIMITIVE(Arithmetic(MUL,INT))
    JUMP 4

  4:
    RETURN(INT)

}
```

Figure 1. ICode Example: Factorial

2.3 Platforms

One of the components plugged into the compiler object describes the target platform. This component defines how information on external types is located and loaded into the symbol table, the platform specific phases that should be incorporated and the implementation of a few other platform abstractions. For example, the JVM platform specifies that referenced types should be loaded from Java class files found on the compilation classpath and that the `genJVM` phase should be run to generate Java class files from the ICode classes. The main addition to the Scala compiler for this project is the additional platform component for LLVM and the code generation phase it instructs the compiler to use.

3. Low Level Virtual Machine

The *Low Level Virtual Machine (LLVM)* is an open source compiler framework for optimization[6], code generation and lifelong program analysis[7]. It began as a research project at the University of Illinois and is now being used by a number of open source and commercial projects[1]. It is used as the target for a number of functional and object oriented languages including Haskell[11] and Python[2].

3.1 Intermediate Representation

There are three equivalent representations for LLVM assembly, called LLVM Intermediate Representation (LLVM IR), a textual syntax for human inspection and authoring, a space efficient binary format and an in memory representation used by the LLVM tools an libraries. LLVM includes tools for processing code in the binary format (LLVM Bitcode) and converting between the textual and binary forms, these tools are described in more detail below.

The semantics of LLVM IR are defined in the *LLVM Language Reference Manual*[8]. LLVM IR is essentially a register transfer language (RTL)[4] given in static single assignment (SSA) form [3]. LLVM IR is distinguished from most RTL languages by the presence and use of high-level type information. This type information and the dataflow information implicit in the SSA form provide opportunities

```
define i32 @factorial(i32 %n) {
entry:
  %iszero = icmp eq i32 %n, 0
  br i1 %iszero, label %return1, label %recurse
return1:
  ret i32 1
recurse:
  %nminus1 = add i32 %n, -1
  %factnminusone =
    call i32 @factorial(i32 %nminus1)
  %factn = mul i32 %n, %factnminusone
  ret i32 %factn
}
```

Figure 2. LLVM Example: Factorial

for much richer optimizations and analyses than are typically possible for a low level representation. As an example, type safe pointer manipulation enables much more precise alias analysis.

A complete function written in LLVM IR is shown in Figure 2. This demonstrates the explicit nature of control flow and typing in LLVM IR. Every instruction contains type information for its arguments, and when necessary the result type. In this case only two types are used: `i32` and `i1`; 32 bit and 1 bit integers respectively. In reality a third type is used in this fragment, namely the type of `@factorial`: `i32(i32)*`. This type can be inferred by the return type and argument types used in the call instruction and does not need to be named explicitly. Each basic block is introduced by a label and is explicitly terminated by a control flow instruction. Even if execution should fall through to the next basic block, a branch instruction is still necessary.

LLVM and its intermediate representation supports other important features such as efficient tail calls, arbitrary instruction metadata, precise garbage collection[9], zero-cost exception handling[5] and atomic memory operations. These features make LLVM IR an attractive target for compilers. Compilers that target LLVM get all of these features, analyses and optimizations for free, the quality and quantity of which are consistently increasing.

3.2 Clang: A C Compiler Frontend

The LLVM Project is also developing a compiler for the C family of programming languages. It aims to be a drop-in replacement for GCC that natively targets LLVM IR. While the project is young and under active development, it is already considered to be production ready. This project is currently using Clang to compile the base runtime library for Scala on LLVM.

4. The LLVM Backend

The LLVM backend for the Scala compiler defines a new platform component that is selected when the compiler is invoked with the `-target:llvm` option. The main contribution of the platform component is to register the LLVM IR generation phase in the compiler pipeline. This section describes the design and implementation of this phase.

```
; Primitive Types
%Boolean  = type i1
%Byte     = type i8
%Short    = type i16
%Int      = type i32
%Long     = type i64
%Float    = type float
%Double   = type double
%Char     = type i32
%Unit     = type void

%.class = type {
  %.utf8string,     ; class name
  i32,              ; instance size
  %.class*,         ; parent class
  %.vtable*,        ; class vtable
  %.class*,         ; array class
  %.class*,         ; element type
  i32,              ; interface count
  [0 x %.ifaceinfo] ; interface vtables
}
%.ifaceinfo = type {
  %.class*,         ; pointer to interface class
  %.vtable*         ; vtable for interface methods
}

%.object = type { %.class* }

%.ifaceref = type { %.object*, %.vtable* }

; The instance type for
; class object_with_int { var x: Int }
%object_with_int = type {
  %.object,
  i32
}
```

Figure 3. Scala types in LLVM

4.1 Types

This subsection describes how Scala types are mapped to LLVM types. The definitions for some Scala types in LLVM IR syntax is shown in Figure 3.

Primitives

Primitive types in Scala map directly to LLVM integer and floating point types. A notable difference between the primitive sizes we have chosen and the other targets for Scala is that we define characters to be 32 bits. We have made this choice so that each Unicode codepoint can be represented by a single character value. This may also ease interoperability with C libraries on systems where `wchar_t` is 32 bits.

Objects and Object References

Objects are represented as structures whose first element is the instance structure for the parent class and the remaining elements are the fields of the object. An object reference is simply a pointer to one of these structures. This arrangement reduces casting between object types to a reinterpretation of the pointer type.

The base object structure contains a pointer to the class of the object. Any object reference can be casted to a pointer to the base object structure to retrieve the class pointer.

Interface References

An interface reference is a pair containing an object reference and a pointer to the appropriate interface vtable for the object.

Classes

Classes are global structures containing the class name, the size of an instance, a pointer to the super class, the class vtable and a list of interface vtables. The structure also contains a few other fields for cached derived values and some fields that are used only for classes representing arrays.

Arrays

Arrays are essentially a special type of object that are instances of special array classes. Array classes for each primitive type are predefined, array classes for other class types are created on demand and cached in the class record for the element type. Array classes also contain a pointer back to the element class. Array instances directly contain the array data in the same way as a flexible array member in C.

Methods

Methods are translated to functions taking the receiver, if any, as the first argument and the formal parameters of the method as the remaining arguments. The function names encode the argument and return types for proper overload resolution. An example of the `fact` method from Figure 1 compiled to LLVM and optimized by the LLVM offline optimizer can be found in Figure 4.

4.2 Mapping Stacks to SSA

LLVM is a register machine requiring all code to be in SSA form while ICode is a stack machine. Therefore to generate LLVM IR from ICode a method of mapping from a stack based machine to an SSA register machine is needed. Fortunately a straightforward mapping exists.

Within a basic block it is sufficient to maintain a stack of SSA values while translating the ICode instructions. When an instruction is translated, the values it consumes are popped from this stack and the values that are produced are pushed. The ICode type of each value is also maintained on this stack.

Conveying values between basic blocks is somewhat more complicated. Each basic block is analyzed to deter-

```
define i32
@method__Ofact_Mfact_Ascala_DInt_Rscala_DInt
  (%_Ofact* %.this, i32 %n) {
.entry.:
  %"1" = icmp eq i32 %n, 0
  br i1 %"1", label %bb.2.-1, label %bb.3.0

bb.2.-1:
  ret i32 1

bb.3.0:
  %"15" = getelementptr inbounds
            %_Ofact* %.this, i32 0, i32 0
  invoke void @rt_assertNotNull(%.object* %"15")
          to label %bb.3.1
          unwind label %bb.3.exh.-2

bb.3.1:
  %"14" = add i32 %n, -1
  %"17" = invoke i32
    @method__Ofact_Mfact_Ascala_DInt_Rscala_DInt
      (%_Ofact* %.this, i32 %"14")
    to label %bb.3.2
    unwind label %bb.3.exh.-2

bb.3.2:
  %"18" = mul i32 %"17", %n
  ret i32 %"18"

bb.3.exh.-2:
  %"19" = tail call i8* @llvm.eh.exception()
  %"20" = tail call i32 (i8*, i8*, ...)*
    @llvm.eh.selector(
      i8* %"19",
      i8* bitcast (i32 (i32, i32, i64, i8*, i8*)*
                   @scalaPersonality to i8*),
      %.class* @class_java_Dlang_DThrowable,
      i32 0)
  %"21" = tail call %.object*
            @getExceptionObject(i8* %"19")
  %"22" = tail call i32
            @_Unwind_RaiseException(i8* %"19")
  unreachable
}
```

Figure 4. Generated LLVM Example: Factorial

mine how many values are required to be on the stack upon entering the block and how many values the block leaves on the stack upon exit. The outgoing stack slots of each block are given unique names based on the block identifier and the position of the slot on the stack. Before translating the contents of the basic block, phi instructions are inserted that map the outgoing stack slots of the predecessor blocks to the incoming slots of the block. Before completing translation of a basic block the values remaining on the stack are assigned to SSA registers with the appropriate distinguished names. Note that each block is responsible for selecting the stack slots it uses directly as well as those it merely passes along to its successors.

4.3 Translating ICode

The LLVM code generation phase iterates over all of the ICode classes created by the compiler converting each one to its own LLVM IR file. The type information of the ICode class is used to build the definition of the global class structure and the type for instances. If the class is represents a singleton object a global variable is defined for the instance and a initialization function is emitted. A global structure containing any static fields of the class is also emitted.

Each method within the class is translated by creating an initialization basic block that allocates stack storage for local variables and copy any arguments into the corresponding local. Each ICode basic block in the method is translated into a sequence of LLVM basic blocks by translating each instruction in sequence, splitting the basic block as necessary due to exception handling points.

Each ICode instruction is translated to a sequence of LLVM instructions that implement the semantics if the ICode instruction. Simple ICode operations can be performed in one or two LLVM instructions while more complicated operations may require many more. For example, ICode assumes implicit coercions between object and interface references as well as various primitive types so extra instructions to perform these coercions must be inserted. On the other hand, stack manipulation operations like `DROP` and `DUP` do not actually require any LLVM instructions, just manipulation of the value stack that is used when translating the next instruction. Detailed description of the translation for each instruction is outside the scope of this paper. Interested readers can inspect the source code.

The LLVM code emitted by the backend is simplistic and often contains redundant instructions. This makes it easier to observe that the proper semantics are implemented but direct execution would be slower than necessary. The LLVM optimizer can eliminate much of the redundancy producing faster code. However, some redundancies are not currently eliminated by the optimizer. For example, repeated loading of the class or vtable for a reference is not eliminated because the optimizer does not realize that the class of a reference cannot be changed. This example and some others can be addressed by using functions that lie about their memory effects. In this case a function that loads a vtable could be declared with an attribute stating it does not read any memory even though the definition does. The optimizer would then correctly assume that the vtable is at the same address each time it is loaded. Scala specific optimization passes could be written to address remaining inefficiencies.

4.4 Method Dispatch

Method dispatch is one of the more complicated parts of compiling Scala to LLVM. The JVM and CIL have built in method resolution but here we must build it from scratch. We choose to use a simple vtable method.

Each class contains a single class vtable and any number of interface vtables. The class vtable is built by iterating over the ancestor classes. An interface vtable is created for each interface implemented by the class for the methods defined in that interface. The entries in the vtable are pointers to the functions implementing the method corresponding to each slot in the vtable. Methods are ordered in the vtables so that the vtable for a subclass contains an appropriate vtable for the superclass methods as a prefix.

When calling a class method the receiver's vtable is loaded through the object's class pointer. The proper index in the vtable is loaded and the opaque pointer is casted to the method's function type and the method is invoked. Invoking interface methods proceeds in the same way except the vtable is extracted directly from the interface reference.

4.5 Native Methods

Methods annotated with `@native` do not have their method bodies translated to LLVM. Instead the function symbol for the method is declared as externally provided but is still registered in any of the class' vtables. This is used for methods which are provided as part of the runtime library.

4.6 Exception Handling

The LLVM backend deals with exceptions using LLVM's support for zero-cost exception handling as documented in *Exception Handling in LLVM*[5]. Functions that can raise exceptions are called using the `invoke` instruction. The `invoke` instruction specifies the successor block for a normal return from the function and the successor block for abnormal exit of the function.

Each ICode method contains a list of exception handlers that specify which basic blocks are covered by the handler, the type of exception accepted by the handler and the initial basic block of the handler's code. Basic blocks that belong to exception handlers are no different from the other basic blocks in the method except that they are not present in the normal successor set of the method's entry point.

The unit of coverage for exception handlers in ICode is the basic block so it suffices to have a single exception landing pad for each ICode basic block. This block calls the LLVM exception handling intrinsics that denote the block as an exception landing pad and extracts the Scala exception

object from the unwind context. The type of the exception is tested against the types accepted by the handlers that cover the block in order from inner-most to outer-most. Control is transfered to the first handler that matches the exception type. If no exception handlers match the exception is rethrown.

Since `invoke` is a control flow instruction it terminates LLVM basic blocks. However, ICode does not consider exception handling control flow for defining basic blocks so the code generator must split the ICode basic blocks. To do this the code is first generated as a single block ignoring the control flow effects of the `invoke` instruction. Note that at this point the block is not valid LLVM IR. Then the instructions are traversed and the block is split each time an `invoke` instruction is seen, the normal successor is set to the successor block and the unwind successor is set to the exception handling block.

5. Runtime

The runtime library for Scala on LLVM implements a subset of the Java API and a small number of primitive functions used by the generated LLVM IR, implemented with a mixture of Scala and C/C++. The class structure and methods for java.lang.Object, java.lang.Class and java.lang.String are implemented entirely in C. The remaining classes are implemented in Scala with native methods for operations which cannot be implemented directly in Scala. The primitive functions implement operations for null checking, value boxing/unboxing, string concatenation, runtime type checks, interface casts, instance allocation and initialization. Going forward we intend to continue writing as much of the runtime in Scala as possible.

The final piece necessary to run Scala programs with LLVM is the loader. The loader is written in C++ and uses the LLVM API. It performs the following steps:

1. Initializes LLVM
2. Loads the program's bitcode
3. Creates a JIT execution engine
4. Initializes the object whose main will be run
5. Dynamically creates a function that will
   (a) Install a toplevel exception handler
   (b) Convert the `argv` C array to a Scala `Array[String]`
   (c) Invoke main
6. Calls the created function

The loader currently only supports JIT compilation but it could be extended to support ahead of time compilation by emitting the generated function as LLVM bitcode to be linked with the original program.

6. Current Status

The backend and runtime together are capable of compiling and running a sample program that uses classes, traits, overloading, overriding, inheritance, exceptions and arrays. Though this program is simple it exercises a large number of ICode opcodes and we believe it is representative of what would be seen in real programs. The backend can also compile all of Scala's runtime library to well-formed LLVM IR. However at this time, programs using the standard library cannot be run, mainly because of dependencies on Java classes that remain unimplemented in the runtime library.

The LLVM backend is still quite far from production ready, however. The largest gaps at compile time and runtime, respectively, are separate compilation and garbage collection. The rest of this section lays out these and other shortcomings and plans for addressing them.

6.1 Separate Compilation

Separate compilation is not currently possible because the code generated for method can field resolution depends on the physical layout of the vtables and object structures. This information is not available unless the classes involved are being compiled from source. One strategy for addressing these types of dependencies is to emit code using a more declarative style and writing a link-time pass that would assemble the structures and instructions to be used at runtime.

Another aspect of separate compilation that is not currently implemented is a loader for external type information. We are currently using the classpath and symbol loaders from the compiler's Java backend. These components will need to be replaced with implementations that read type information from LLVM IR and LLVM library archives. LLVM IR supports arbitrary metadata that could be used to embed an analog of the ScalaSignature attribute added to class files. Alternatively, type information could be read from additional files included in the LLVM library archives.

6.2 Garbage Collection

The runtime for Scala on LLVM does not implement any kind of automatic memory management so storage allocated for instances and arrays is never reclaimed. Unlike the JVM and CIL, LLVM does not supply a garbage collector. It does provide the capability programs in LLVM IR to denote GC roots and establish read and write barriers. This information can be used by an LLVM pass to create stack maps and other structures and code needed to support precise precise garbage collection. We hope to integrate an existing collector or find a way to implement the collector in Scala itself as is done with MMTk in the Jikes Research Virtual Machine.

Supporting garbage collection may require changes to the way object and interface references are handled within functions because LLVM requires that GC roots be in stack allocated variables while we use LLVM registers to hold object references. We may implement this by maintaining

a pool of stack variables for references. When a reference is pushed to the stack we will store that reference in one of these variables and clear the variable when the reference is popped.

6.3 Reflection

Fully supporting Scala's structural types will require reflection facilities. We have ideas to elide reflection when the static type of a reference meets the structural type refinement but reflection will still be required when this information must be computed at runtime. We are hopeful that declarative method tables used to enable separate compilation can be used to implement reflection as well.

6.4 Runtime Library

Most existing Scala libraries and programs, including the Scala compiler and standard libraries, are written assuming the availability of the Java APIs. The CIL backend for Scala is exploring a source-to-source transformation to deal with this problem. However, we do not have any pre-existing runtime to fall back on and must implement these base libraries from scratch. We plan to implement them as much as possible in Scala but recognize that we may need to use native code to implement certain parts.

6.5 Threads

Our implementation is currently strictly single threaded and contains non-reentrant code. We will eventually tackle this problem implementing thread primitives using either native platform threading libraries or a portable threading abstraction layer. This particular limitation is currently viewed as less severe as many useful programs can be written without support for threads.

7. Future Goals

In this section we propose some future work that goes beyond creating a basically functional backend.

7.1 Lightweight Functions

When compiling Scala to Java bytecode each anonymous function is implemented as a unique class because the JVM does not have first-class functions or closures. However with LLVM we could treat functions as a primitive type, much the same as is done with the JVM primitive types. The Function class would be represented in LLVM as a pair containing a pointer to the function and a pointer to the closure context. This would require changes to other parts of the compiler because functions are converted to classes early in the pipeline and ICode has no representation for functions.

7.2 Platform Abstraction of Scala Libraries

Scala's standard library is currently dependent on the Java API. It would be good for both the LLVM backend and the CIL backend to separate the parts of the library that are tied to the JVM from those that are not. For example, the JavaConversions class in the collection package would not have a direct analog on other platforms.

There are also parts of the library that use Java classes in their implementation for proper interaction with the underlying platform. Example of this include I/O, exception classes, comparators and mathematics. A potential strategy is include the platform neutral portions directly in the class and mix in a trait from a `scala.platform` package. Each backend would then have a different implementation of the traits in `scala.platform`.

7.3 Foreign Function Interface

A foreign function interface (FFI) would make interaction with native libraries much easier. We intend to explore an annotation driven FFI at some point in the future. The annotations would provide a declarative means to specify parameter and result marshalling. A successful implementation should nearly eliminate the need for the native methods currently implemented in C. These methods would instead be implemented with the FFI.

7.4 Scala Specific LLVM Optimizations

LLVM provides facilities for writing language specific optimization passes. These optimizations can exploit high-level information about the program communicated by the compiler by metadata attached to the LLVM instructions. There may be specific Scala idioms that would benefit from a targeted LLVM optimization.This could include whole program optimizations performed at link time. As an example, if a base class method is only invoked on instances of a certain subclass loads of the receiver vtable could be replaced by a direct reference to the subclass vtable.

References

[1] The LLVM Compiler Infrastructure: LLVM Users. http://llvm.org/Users.html, 2011.

[2] unladen-swallow: A faster implementation of Python. http://code.google.com/p/unladen-swallow/, 2011.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991. doi: http://doi.acm.org/10.1145/115372.115320. URL http://doi.acm.org/10.1145/115372.115320.

[4] J. W. Davidson and C. W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Trans. Program. Lang. Syst.*, 2:191–202, April 1980. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357094.357098. URL http://doi.acm.org/10.1145/357094.357098.

[5] J. Laskey. *Exception Handling in LLVM (Version 2.8)*. The LLVM Project, Oct. 2010.

[6] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., Univer-

sity of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

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

[8] C. Lattner and V. Adve. *LLVM Language Reference Manual (Version 2.8)*. The LLVM Project, Oct. 2010.

[9] C. Lattner and G. Henrikson. *Accurate Garbage Collection with LLVM (Version 2.8)*. The LLVM Project, Oct. 2010.

[10] M. Odersky and M. Zenger. Scalable component abstractions. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 41–57, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094815. URL http://doi.acm.org/10.1145/1094811.1094815.

[11] D. A. Terei and M. M. Chakravarty. An llvm backend for ghc. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 109–120, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi: http://doi.acm.org/10.1145/1863523.1863538. URL http://doi.acm.org/10.1145/1863523.1863538.