UNIVERSITY OF GOTHENBURG

Adding an LLVM Backend to Bigloo Scheme

A prototype of a portable & simple way of generating native code

Master of Science Thesis in Computer Science

MIKAEL BROCKMAN

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, March 2013

Adding an LLVM Backend to Bigloo Scheme
A prototype of a portable & simple way of generating native code

Examiner: BJÖRN VON SYDOW

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden March 2013

Contents

1 Introduction

The Scheme language has a history of being a vehicle for much experimentation in design and implementation techniques, owing to its simple semantic underpinning based on the untyped λ-calculus [see Sussman and Steele Jr., 1998]. Implementations abound: at the time of writing, the Scheme FAQ [Scheme Wiki, 2012] listed a full 78 different interpreters and compilers.

At present, the Bigloo Scheme compiler is mainly used with a backend that generates C code, though it also includes backends for the Java and .NET virtual machines. My project extends Bigloo to portably produce native code using LLVM [Lattner and Adve, 2004], a compilation framework that has been successfully used (among other uses) in a backend for the Glasgow Haskell compiler [Terei, 2009]. The question was whether this would be viable as an alternative or primary backend, and what benefits could be reaped.

1.1 What is Bigloo Scheme?

Scheme is one of the two major dialects of Lisp, the other being Common Lisp. Standard Scheme has a minimalistic semantics and a syntax based on S-expressions[1]. It keeps classic Lisp features like λ-expressions, strict call-by-value evaluation, linked lists, garbage collection, and dynamic typing, while adding a few new features like a hygienic macro system, mandatory tail call optimization, and first-class continuations. Scheme can be called a "functional language," but it is also an imperative language; in modern dialects, even an object-oriented language.

The Bigloo system is a mostly-compliant implementation of Scheme as specified in the R^{5}RS report [Kelsey and Clinger, 1998]. Though it contains a simple interpreter, its most important part is a static compiler. The explicit objective of the Bigloo project is to allow Scheme to be used where otherwise C or C++ would be required [Inria Sophia-Antipolis, 2011]. This implies some requirements: the generated code must be fast enough; the compiler's output must be native binary objects; and the compiler must have good support for using libraries written in C. In order to achieve these, Bigloo attempts to translate Scheme into rather straightforward C code. Its optimizations and transformations are geared towards producing C code that is structurally similar to how a C programmer would have written the same program; that avoids unnecessary boxing; that does iteration instead of tail recursion; and so on.

[1] Symbolic expressions, a fully-parenthesized prefix notation characteristic of the Lisp family; example: `(+ (* 2 4) 1)`.

1.2 Native Compilation: C vs LLVM

The strategy of compiling to C has many advantages over producing machine code directly: most importantly, there is no need to write special code for each CPU architecture, since this hard work is already done by portable C compilers such as GCC. But as might be expected, since C was not designed to be used as an intermediate language, there are also disadvantages. The authors of Bigloo acknowledge this [Serrano et al., 1995], mentioning specifically the difficulty of implementing `call/cc` (see section 5.1 about continuations) and the necessity of using conservative garbage collection.

The motivation for this project comes from the view that by using the intermediate language provided by the LLVM project where C is now used, the Bigloo compiler could have a portable native backend combining most of the advantages of C with new opportunities stemming from LLVM's novel design. The LLVM project provides a "collection of modular and reusable compiler and toolchain technologies" [Lattner, 2011]. The subproject relevant to my project is called LLVM Core, and essentially consists of an optimizing compiler for a generic assembly language called LLVM IR. This intermediate language has some interesting properties described in section 4. Potential advantages to using LLVM IR instead of C include

- guaranteed support for full tail-call optimization, which is required by the Scheme specification but cannot be guaranteed with C function calls;
- eliminating the dependence on C compilers, which are large and complicated, and are different enough that Bigloo's C code must use conditional compilation for some constructs;
- providing the future possibility to use LLVM's support for Just-In-Time optimization and dynamic compilation, which could greatly enhance the performance and flexibility of Bigloo, since Scheme is a dynamic language; and
- being able to implement nonconservative garbage collection, feasible with LLVM since it provides flexible hooks for GC implementation, while with C one is basically restricted to using Boehm's conservative GC library.

2 The Scheme Language

2.1 Historical Roots

The origins of Lisp lie deep within the ancient history of programming, as recorded by its primary inventor John McCarthy in a historical review [McCarthy, 1979]. Along with FORTRAN, it was one of the very earliest high-level programming languages. The initial impulse came in 1956 as a desire for an algebraic list processing language for the IBM 704 computer. The desire for such a language was related to the emerging field of research into artificial intelligence and computer processing of symbolic logic. At this time, FORTRAN was in the developing stages, and it was unsure whether it would turn out to be suitable for the kind of list processing in which McCarthy was interested.

The designing of Lisp had two sometimes conflicting goals: to have a programming language that was mathematically useful—i.e., one that could be reasoned about algebraically and that could naturally describe mathematical problems, especially symbolic such—and to have a practical and efficient way of programming the specific type of computers available. McCarthy describes for example how he chose to formulate the basic list operations as functions just like the arithmetical operators rather than as special statements—because this simplified the semantics and made programs easier to reason about. This kind of mathematical aesthetics was also the reason why garbage collection was chosen over explicit memory management. However, for practical reasons, "impure functions" (like the `RPLACA` and `RPLACD` functions, mutating the head and tail components of a pair, respectively) were also allowed, though research on proving properties of programs in the "pure" subset of Lisp was done in the 1970s.

The early formulations of Lisp [e.g. McCarthy, 1962] define a language that operates exclusively on symbolic data called *S-expressions*, which are binary tree structures whose nodes are symbolic atoms, e.g., `(FOO . BAR)`[2]. The programming language itself consists of what were called *M-expressions* (or *meta-expressions*), which describe recursive functions of S-expressions, with basic operations like *cons*, *car*, and *cdr* to construct pairs and extract their elements, *eq* to check equality of S-expressions, conditional expressions (at the time a novel feature), along with functional abstraction and application.

When Lisp was being developed, writing a compiler was considered a very large

[2] Lists were encoded as right-leaning trees ending with a `NIL` atom; with a syntactical shortcut, the structure `(FOO . (BAR . (BAZ . NIL)))` can be written simply `(FOO BAR BAZ)`. This syntax remains.

Listing 1: The *maplist* function written in the LISP 1.5 dialect

```
(MAPLIST (LAMBDA (L FN) (COND
   ((NULL L) NIL)
   (T (CONS (FN L) (MAPLIST (CDR L) FN))) )))
```

project, and much planning and research was done before any implementation began. McCarthy and his colleagues sometimes hand-compiled M-expressions to IBM 704 code for testing. As part of his research, McCarthy decided that in order to demonstrate that Lisp was a good theoretical tool, he would formulate the equivalent of the universal Turing machine—the Turing machine that simulates arbitrary Turing machines—i.e., a Lisp function for evaluating arbitrary Lisp functions. This required a way of representing Lisp functions as Lisp data—i.e., a translation of M-expressions into S-expressions—and such a representation was developed for the theoretical purpose of allowing the description of the *evalquote* function, which indeed turned out to be simple and elegant. A graduate student named S.R. Russell realized that this function could be hand-compiled just as any other function—and did so, resulting in what McCarthy calls "the unexpected appearance of an interpreter." Programming with S-expressions became the standard procedure, though in the *LISP 1.5 Programmer's Manual* [McCarthy, 1962] this is still seen as only an internal notation. The plan to develop a way to permit the input of M-expressions was gradually abandoned, and the notion of "code as data" took off and became one of Lisp's most distinguishing features. As McCarthy writes,

> One can even conjecture that Lisp owes its survival specifically to the fact that its programs are lists, which everyone, including me, has regarded as a disadvantage. [McCarthy, 1979]

2.1.1 The λ-calculus

The λ-calculus was developed by Alonzo Church in the late 1920s and first published in 1932 [see Church, 1932]. It was a formal system intended as a foundation for logic, in the vein of Russell's theory of types and Zermelo's set theory, but more natural, being based on functions instead of sets [Cardone and Hindley, 2009]. The calculus consists only of *abstractions*, of the form $\lambda x.M$ (where x is a variable and M is an expression), and *applications* of the form FX (where F is an expression denoting some abstraction and X is an expression denoting a value to which the abstraction is to be applied). Precise rules governing the evaluation of this calculus

were given by Church, though the first published papers were found to be contradictory; the substitution rules for application (β-substitution) are nontrivial. The λ-calculus is considered to be the first in-depth exploration of the formal properties of functional abstraction [see Cardone and Hindley, 2009].

When McCarthy was developing Lisp, he did not set out to create an computer language implementation of the λ-calculus. He was somewhat familiar with Church's notation, but did not understand it fully; describing the insights gathered from working on symbolic differentiation as a sample problem, McCarthy writes:

> To use functions as arguments, one needs a notation for functions, and it seemed natural to use the λ-notation of Church [1941]. I didn't understand the rest of his book, so I wasn't tempted to try to implement his more general mechanism for defining functions. [McCarthy, 1979]

2.2 Core Lisp Features

The novel ideas of Lisp, as enumerated by Paul Graham in an article making the case for Lisp as a repository of powerful language techniques with which the rest of the programming world has not yet quite caught up[3] [Graham, 2002], are the following:

1. *conditional expressions* instead of simple conditional jumps;
2. *first-class functions*, i.e., functions that can be returned and passed as arguments;
3. *recursion*, which was prohibited in FORTRAN I;
4. *dynamic typing*, along with the idea of all variables being pointers to some general kind of object;
5. *garbage collection*;
6. a nestable *expression syntax* not just for arithmetic;
7. *symbol values*, being essentially immutable token strings with quick equality tests;
8. *code as data*, i.e., as S-expressions; and
9. the *integration of read-time, compile-time, and run-time*.

[3] Graham lists them "in the order of their adoption by the mainstream."

2.3 Towards Scheme

Throughout the 1960s and 1970s, Lisp began to be used at many different universities (e.g., the AI labs at MIT, Stanford, and CMU) and research departments (Xerox, Symbolics) across the U.S. [see Pitman, 1996], and dialects and implementations proliferated, adding new features and improving performance.

Other research in programming language semantics also made progress. For the narrative leading towards the development of Scheme, of particular relevance is the work on languages and theories that treat communicating parallel processes as the basic abstraction. Simula [Nygaard and Dahl, 1981], the first object-oriented language, was developed by Norwegian researchers Nygaard and Dahl at the Norwegian Computer Centre between 1962 and 1967 as a language for describing and simulating "discrete event networks," and became very influential [see Holmevik, 1994], in large part through inspiring Alan Kay to create the Smalltalk system [Kay, 1993][4]. As these systems were developed, there were also attempts to formalize their semantics mathematically. One of these attempts was the *actor model of computation* first published in 1973 [Hewitt et al., 1973], for which Hewitt developed a language called Planner-73 (later PLASMA).

And this is how Scheme began: as an attempt by Sussman and Steele to understand the "unusual terminology" of Hewitt's actor model in terms of ordinary programming notions [Sussman and Steele Jr., 1998]. They began to write a "toy implementation" of the actor model as a Lisp dialect. From Algol they took the idea of lexical scoping, which they thought would provide a simple way to implement the "acquaintances" of the actor model, while letting actors and functions be implemented in similar ways. They had the keyword `alpha`, which worked almost the same as `lambda`, but creating an actor instead of a closure. Message passing was done by applying an actor just like one applies a function. It eventually turned out that except that actors worked by invoking *continuations* (see section 5.1) instead of returning values, actors and functions in this language were semantically identical. They were pleased to have found a very simple core language, built on the λ-calculus, that supported the advanced programming style of the actor model. Sussman's and Steele's ideas about language design and implementation were published in a series of papers called "the lambda papers" between 1975 and and 1980. Steele wrote the first optimizing Scheme compiler RABBIT in 1978. After pub-

[4] Itself inspired by Lisp; as Kay writes, "The biggest hit for me while at SAIL in late '69 was to *really understand LISP*. Of course, every student knew about *car*, *cdr*, and *cons*, but Utah was impoverished in that no one there used LISP and hence, no one had penetrated the mysteries of *eval* and *apply*. I could hardly believe how beautiful and wonderful the idea of LISP was."

lishing a revised report on the language [Steele Jr. and Sussman, 1978], it spread as a tool for research and teaching. It was officially standardized in 1991 [IEEE Std 1178-1990, 1991]. At present, the most widely implemented standard is that presented in the *Fifth Revised Report* [a.k.a R^{5}RS; Kelsey and Clinger, 1998].

Syntax. The syntax of Scheme and other languages in the Lisp family is unusually simple. There are essentially only three kinds of entities in the Scheme syntax: literals, symbols, and combinations. A literal is a number, string, or some such value. Symbols are names like `foo`. Combinations are arbitrary-length lists of entities, like `(x y z)`. Evaluating a combination usually means invoking the function denoted by the first element with the rest of the elements as arguments, but some symbols (`if`, `lambda`, etc) make the combination evaluate as a "special form."

There is a close connection between the syntax and semantics. The connection, perhaps the most distinguishing feature of the Lisp family, called *homoiconicity*, is that program code can be read as elements of the fundamental data types of the language itself: symbols and literals are first-class values, and combinations are simply lists. So a Scheme program can be seen as simply a particular kind of Scheme value serialized in Unicode form. Homoiconicity is part of what allows for the Lisp style of metaprogramming by working directly with source trees through "macro" functions integrated seamlessly with the programming environment. In Common Lisp, macros are like functions that receive their argument expressions unevaluated and return source trees to be inserted by the compiler (during "macro-expansion"). Standard Scheme instead provides a more sophisticated system of template-like macro definition that is "hygienic," i.e., offers certain guarantees regarding variable name capture. Bigloo, like most Scheme implementations, offers both styles of macro programming.

3 Introduction to Bigloo

Bigloo is mostly written in Scheme; it is a "bootstrapping" compiler, meaning that to compile it from scratch requires a pre-compiled Bigloo executable. That the entire compiler is written in Scheme makes working on it relatively easy compared to if it had been written in C. Parts of the runtime library that compiled programs call upon are written in C.

The Bigloo design is structured using two non-standard features of Bigloo's dialect of Scheme: a module system and a framework for object orientation. So the

Listing 2: Annotated Bigloo factorial function

```
(define (factorial::int n::int)
  (if (<= n 1)
    1
    (* n (factorial (- n 1)))))
```

large-scale structure of the system is organized using modules: different compilation passes reside in different modules, for instance, but mostly operate on the same object-oriented data structures. This design makes it convenient to extend the system with a new backend: one only needs to extend the `backend` class, register a construction function with the compiler driver module, and add a new command-line option to choose it.

3.1 Language

Since the standard Scheme of the reports is so minimal, concrete Scheme systems usually provide extensions, both to the language itself and to the standard library. Bigloo also chooses to deviate from the specification in some instances where following the exact Scheme semantics would make the resulting C code slower or less straight-forward with little benefit.

3.1.1 Type System

Standard Scheme is latently typed, meaning that types do not appear explicitly in the source code, and that in the semantics, only values have types, not variables. Standard functions are defined to cause errors when incorrectly applied; for instance, `first` causes an error if applied to a non-list. There is no explicit way to define derived types like records and product types; with standard Scheme, the programmer can either use metaprogramming macros to define facilities like `define-record`, or simply represent composite values as lists or vectors, perhaps providing accessor functions to create a pseudo-abstract data type.

In Bigloo's language, a syntax extension provides optional manifest typing of variables, which besides being a kind of documentation and a way to detect bugs at compile time can also help the compiler generate better code. For example, listing 2 shows a recursive factorial function defined with type annotations. The `int` type it uses is one of Bigloo's primitive integral types. With the C backend, this function is compiled to code that uses C's primitive `int` type; the LLVM backend uses the

Listing 3: A simple module header clause

```
(module math_stuff
  (import math_helpers) ;; Import another module.
  (export
    (factorial::int ::int) ;; Export a function
    *pi*) ;; and a variable.
  (main math_test)) ;; Declare a main function.
```

i32 type. Without annotations, the function would also be able to handle floating-point numbers, bignums, and so on, with corresponding cost in performance. By default, Bigloo will also generate runtime type checking code, though this can be disabled.

Manifest typing of function parameter is also an important part of the object system, described in section 3.1.3. Also, Bigloo supports the semi-standard SRFI-9 framework for defining record types.

When using the C backend, the Bigloo type system is integrated with the C type system. This makes it easy to import and export functions between C and Bigloo.

3.1.2 Module System

To help give structure to larger programs, Bigloo code is organized in modules with statically defined interfaces. The module is the basic unit of compilation; one module corresponds to one object file. Tools are provided to calculate dependencies between modules to make compilation of large systems easy. A module is defined by a module header clause at the beginning of a source file, see e.g. listing 3.

3.1.3 Objects

Object oriented programming is built into Bigloo on a fundamental level: class declarations belong to module interfaces, object operations are primitive within the compiler, and so on, though it is not the case that "everything is an object." The object system is similar to CLOS [Common Lisp Object System; see Pitman, 1996, chapter 7] but was primarily inspired by another Scheme object system called Meroon [see Queinnec, 1993]. Compared to CLOS, Bigloo's object system (like Meroon) is more similar to other widely-used object systems in that its runtime dispatching only considers one argument, the "self" or "this" parameter, whereas

CLOS uses "multiple dispatch." Also, Bigloo's object system, again like Meroon, supports only single inheritance.

As with CLOS, classes and methods are defined separately. A method is a function that is specialized on the runtime type of its first argument. Many methods can be collected by belonging to the same *generic function*. A generic function consists of a name and a type signature, and belongs to a certain class. Different concrete methods for every subclass can then be associated with this generic function. To invoke a generic function, the appropriate method for the class of the "this" parameter must be found. The LLVM backend uses the dispatching code of the C runtime library to do this.

3.2 Limitations

Bigloo's most significant deviation from R5RS is that tail calls in the general case are not optimized in the way the specification requires. Tail calls to functions locally defined (using `define` or the functional `let` construct) are optimized into jumps, which takes care of the most common cases, but this is not enough to support some kinds of programming that Scheme was designed to allow: coroutines, actor-style control flow, and so on. At least the GNU C compiler attempts to do some tail call optimization, but makes no guarantees. In practice, Bigloo developers are recommended to avoid general tail recursion, lest the program run out of stack space.

3.3 Optimizations

Aside from standard optimizations like tail-call optimization, loop unrolling, and inlining, Bigloo uses some novel techniques that are well-suited for Scheme's dynamic nature. A compilation pass called *storage use analysis* [see Serrano and Feeley, 1996] is used to discover information pertaining to the optimal allocation of value storage: replacing heap allocations with stack allocation, handling closures efficiently, allowing unboxed representations, and eliminating redundant type checks. Bigloo also incorporates research on when automatic inline expansion of functions should be done [see Serrano, 1997].

4 Introduction to LLVM

The name LLVM was originally an acronym for "Low-Level Virtual Machine." Unlike most other virtual machines, such as the Java and .NET virtual machines,

LLVM has no security features (like bytecode verification or obligatory checking of dynamic types or array bounds), no built-in garbage collection, and no concern for cross-platform portability of compiled modules. In many respects it has more in common with a C compiler than a typical virtual machine, except instead of C it works with a language explicitly designed to be an intermediate language for compilers; and instead of following the classical model of a compiler executable transforming an input source file into an output object file, it also provides an object-oriented library for compiling and executing functions.

4.1 Virtual Machine Characteristics

4.1.1 Values and Types

The type system of LLVM IR is one of its most important features. Functions, instructions, and values all have well-defined and statically checked types. In typical machine assembly languages, there is a set of untyped but word-sized registers, and a byte-addressed store of untyped memory; in LLVM IR, instead there are typed names representing values of a given size and type, which in the compiled machine code may be held in either registers or memory.

The essential primitive value types of LLVM are integral and floating-point numbers of various sizes; there is also the void type, the type of labels, and a type for what LLVM calls "metadata." From these types, we can build derived types: pointers, structures, arrays, and vectors (like arrays of primitive values intended to be optimized with SIMD techniques).

LLVM strictly upholds the difference between integers and pointers. There is a well-typed instruction (`getelementptr`) for calculating the address of a particular element of a derived type; if one wishes to do explicit pointer arithmetic, one must use explicit conversion instructions.

4.1.2 Functions, Blocks, and Instructions

All IR instructions occur within some function definition. They are also organized into "basic blocks," such that every label must be preceded by a "terminator instruction," e.g., a branch or return.

4.1.3 Static Single Assignment

Value names in LLVM IR are sometimes called "registers," but they differ from typical machine registers in that they can only be assigned to once. This require-

ment comes from the fact that LLVM enforces so-called *static single assignment form* (SSA), which is a way of writing code that makes many optimizations easier. However, code generators do not need to worry about this too much, because all variables can be coded as loads from and stores to stack or heap memory; one of LLVM's optimization passes (`mem2reg`) then effectively transforms this usage into conventional SSA form.

4.2 API vs Generating Assembly

There are three different manifestations of LLVM IR: the internal representation of programs within the LLVM runtime, storable "bitcode," and human-readable assembly text. The LLVM C++ library provides an official API for directly creating programs in the internal format, which has some advantages over going via bitcode or assembly. One slight benefit is related to performance: the work of writing and reading assembly code is eliminated. But the major advantage is that using the API lets one bypass the whole procedure of generating an object file, opening up for other models of compilation than the classical static model: for example, allowing programs to access the compiler at runtime to specialize closures, or doing different kinds of dynamic optimization. LLVM itself has support for JIT compilation, and there is an ever-growing field of literature about how to optimize dynamic programs based on statistical information about runtime execution paths [see e.g. Arnold et al., 2005]. Scheme specifies the `eval` function for evaluating arbitrary forms at runtime, which Bigloo currently implements by interpretation; it would be nice to also offer runtime compilation.

5 Compilation Topics

5.1 Continuations

The ability to directly manipulate the *current continuation* is one of the distinguishing features of Scheme. The notion of a continuation (or "context") is often used in formal specifications of programming language semantics, and denotes, for some particular expression which is to be evaluated, the entire "remaining" computation to be performed with the expression's value as input. In Scheme, this remaining computation can be "reified" as a function-like value using the special form `call-with-current-continuation` (or `call/cc`). The meaning of invoking such a reified continuation is to replace the current continuation with the continuation that was reified, with the arguments to the functions becoming the inputs to

that continuation. The `call/cc` form takes as argument a unary function, which is applied to the reified continuation of the `call/cc` form; the function's return value becomes the value of the `call/cc` form. But since the continuation of the form has been reified, and possibly saved in some variable in an outer scope, it is possible for the `call/cc` form to "return any number of times." This operation allows the implementation of many different more or less exotic control structures: exceptions can be quite naturally formulated in terms of `call/cc`, as well as "backtracking" nondeterministic operators like `amb`, cooperative threads, and so on.

Compiling first-class continuations efficiently is notably difficult. One strategy involves using the *CPS transform* to automatically rewrite the input program into "continuation-passing style," [see e.g. Appel, 1992] in which all functions are made to accept the current continuation as a functional argument, and in which instead of returning its computed value, a function body passes control to the continuation with a tail call. Thus the `call/cc` operator is trivial to implement as simply making visible the otherwise implicit continuation parameter.

In this style, functions never return, and if compiled into C code, where tail call optimization cannot be guaranteed, the call stack will grow indefinitely. There are ingenuous methods for solving the stack growth problem, even with reasonably portable C code, but these incur some function call overhead. Also, the "inside-out" call structure makes it trickier to support callback functions from C into Scheme: for example, consider using a comparison function in continuation-passing style with the standard C `qsort` function.

Bigloo does not use the CPS transform. Instead, it compiles function calls and returns in the obvious way. The `call/cc` operator is implemented by simply copying the relevant execution context—i.e., all registers and the entire stack containing arguments, local variables, and return addresses—into heap-allocated memory. The register set is saved and restored using the ISO C functions `setjmp` and `longjmp`. There is no legal way to save and restore the contents of the stack, but the way Bigloo's C backend implements this works in practice: it simply uses `memcpy` to copy memory relative to the address of the "first" variable of the `main` function.

For the LLVM backend, I chose to simply disable the `call/cc` operator, as the JVM backend also does. The stack-copying approach using `setjmp` and `longjmp` could be implemented without much difficulty, but LLVM currently does not encourage this, and using such techniques may cause hard-to-diagnose problems.

5.2 Exceptions

Theoretically, exceptions can be viewed as restricted continuations: ones that may not escape their dynamic scope. In terms of the execution stack, throwing an exception simply means discarding ("unwinding") the top entries of the stack until a matching `catch` statement is found, whereas restoring a continuation in the general case may require loading a totally different stack.

In Bigloo, exceptions are, as continuations are in the C backend, implemented using `setjmp` and `longjmp`. Scheme code using exceptions is transformed into code that only uses simple primitives for pushing and popping the execution contexts off a global stack of contexts. This was also implemented in the LLVM backend.

5.3 Runtime Library

The code that Bigloo's C backend generates depends on a library of runtime functionality to implement fundamental data structures and built-in functions. This common functionality is separately compiled and linked together with the generated code as a dynamic or static library. Its data types and functions are exposed to the generated code through the single header file `bigloo.h`. This library contains a large amount of support code for file and socket I/O, threads, process control, arithmetic and string operations, Unicode support, etc. The LLVM backend reuses as much of this as possible.

6 Results

6.1 An LLVM Backend for Bigloo

The result of my project is an extension of the Bigloo compiler system that compiles Scheme modules into native code through LLVM. It handles a large subset of the Bigloo language with only a few exceptions. It is compatible with modules compiled through the C backend and uses the standard Bigloo runtime library. The runtime performance is currently worse than that of the C backend, but I expect that this is mostly because of the workaround described in section 6.1.2, which could be fixed somewhat straightforwardly, though it would take more time than allowed by the scope of this project.

Listing 4: Assigning a ϕ-node

```
(instantiate::ir-assignment
  (name "%foo")
  (node (instantiate::ir-instr-phi
         (table `((,(make-ir-lit-int i32 0)
                   ,(make-ir-label "a"))
                  (,(make-ir-lit-int i32 5)
                   ,(make-ir-label "b")))))))
```

6.1.1 IR library

As there was no existing library for interfacing Bigloo and LLVM, I wrote a module to do this. It would have been possible to create a binding to the C++ LLVM API, but I decided to generate IR text files instead: this was easier to implement, and fits well with Bigloo's existing model of compilation. However, the module is designed to be adaptable; if an API binding is made in the future, the object representation of IR code could be translated into API calls.

IR nodes are represented as instances of Bigloo classes. For example, the node corresponding to the IR code

```
%foo = phi i32 [0, %a], [1, %b]
```

is shown in listing 4.

6.1.2 Macro Functions

Many standard functions in the Bigloo library are defined as "macro functions." This is a third type of function that is defined neither as a body of Scheme code nor as an FFI-style declaration of an external C function, but as a format string yielding a string containing C code, applications of which the code generator handles by simply splicing in the given string. For example, the `c-eq?` function is defined like this:

```
(extern (infix macro c-eq?::bool (::obj ::obj) "=="))
```

This introduces a difficulty in creating a new backend. There is a facility for adding backend-dependent variants; the Java version of this function, for example, is defined like this:

```
(java (class foreign
        (method static c-eq?::bool (::obj ::obj) "EQ")))
```

For the LLVM backend, it would be good to be able to define this kind of macro as a function returning IR code. I chose to postpone this in favor of a simple temporary solution that works without needing to manually write new definitions of the many macro functions spread throughout the Bigloo system. The temporary solution I implemented creates a C file for each compiled module, containing function variants of every used macro function; for the `c-eq?` function, C code like the following might be generated:

```
obj_t bgl_macro_c_eq (obj_t a, obj_t b) { return a == b; }
```

Uses of the `c-eq?` function are then compiled as function calls to this external function, which is compiled with a C compiler and linked into the final program.

This solution is basically a hack, but considering the large quantity of macro functions defined in the standard library—as well as in other libraries and applications—I chose to do the simplest thing that would work and let me continue with the rest of the backend. Of course, this compromises one of the project's goals, namely to get rid of the need for a C compiler, and also has a negative effect on runtime performance. If this backend is to be included with the Bigloo system, a system for defining inline LLVM macros will have to be designed.

6.1.3 Using Clang to Translate Prelude Code

The header file `bigloo.h` included by all compiled Bigloo programs contains a lot of definitions of data structures and macros, as well as declarations of library functions. Much of this is relevant to the LLVM backend, but cannot be used directly, since LLVM does not support C header files.

Translating C code into LLVM IR is a problem adequately solved by Clang, an LLVM subproject that implements a fully functional C compiler using LLVM as a backend. To translate the needed parts into LLVM code semi-automatically, I used the Clang compiler with the `-emit-llvm` option on a dummy C program that used only the needed parts of Bigloo's header file.

However, this approach only works for data structure definitions and macros that evaluate to expressions which can be represented as LLVM functions. Some macros evaluate to global definitions of values (e.g., `DEFINE_STRING`); since LLVM has no macro facilities, I implemented this kind of functionality by hand in the backend.

6.1.4 LLVM Optimization

LLVM IR code generated by the new backend is suboptimal but highly amenable to the kinds of automatic optimization performed by LLVM's optimizer. Most obvious is the elimination of allocations, loads, and stores done by the `mem2reg` optimization pass. Before LLVM's optimization, a simple factorial function compiles to 85 instructions and labels; after optimization, this number is reduced to 19. The unoptimized code does 20 stack allocations, 22 loads, and 22 stores; the optimizer eliminates *all* of these through *phi* conversion and strength reduction.

7 Future Work

Some work remains for the LLVM backend to be a usable replacement for the C backend.

7.1 Macros

For performance, the macro functions addressed in section 6.1.2 should be implemented as LLVM functions or inlined IR code. The current design for specifying macro functions puts the definitions in the module header, but it does not seem like a good choice to put large amounts of IR generating code there. One possibility would be to allow just giving the name of an IR generating function. I also think that given the large amount of macro functions for which IR generating code would need to be hand-written, the IR library should offer a (perhaps macro-based) wrapper to write IR code more concisely.

7.2 64-bit

The LLVM backend currently assumes that Bigloo is being used in the 32-bit mode, mostly because the development computers I had access to had 32-bit CPUs. Supporting 64-bit compilation would require a new compiler switch and some conditional compilation—Bigloo's data structures and object tags are different on 64-bit architectures.

7.3 Tail Calls

The LLVM backend is not concerned with tail call optimization. As mentioned in section 3.2, the Bigloo optimizer converts some tail-calling functions into loops, but

in general, Bigloo intentionally deviates from the Scheme requirement for full tail call optimization. This is a compromise to enable more straightforward compilation and compliance with C calling conventions.

LLVM itself has support for tail call optimization, but only when both caller and callee functions are marked to use the non-compliant `fastcc` calling convention. Functions using this calling conventions can only be called from within LLVM. However, the LLVM backend could mark module-local functions with this option, since these functions will not be called from outside LLVM.

7.4 API & JIT

In the future, it would be nice to create a binding to LLVM's API for dynamic code generation, and use this instead of source file generation. This would potentially allow for dynamic recompilation and adaptive optimization, though that would require some changes to the Bigloo design.

8 Conclusion

The project has demonstrated that using LLVM as a backend for the Bigloo Scheme compiler is a viable option. Future work to make the LLVM backend fully integrated may encourage a more general way of defining external C functions within Bigloo itself and the Bigloo community. Having a preliminary LLVM backend points towards a future where Bigloo users may enjoy the benefits of runtime compilation, which is a strong part of the Lisp heritage [Graham, 2002].

References

Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN 0-521-41695-7.

Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005. ISSN 0018-9219. doi: 10.1109/JPROC.2004.840305.

Felice Cardone and J. Roger Hindley. *Logic from Russell to Church*, volume 5 of *Handbook of the History of Logic*, chapter Lambda-calculus and Combinators in the 20th Century. Elsevier, 2009.

Alonzo Church. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics*, 2(33):346–366, 1932.

Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941. ISBN 0691083940.

Paul Graham. Revenge of the Nerds, May 2002. URL `http://www.paulgraham.com/icad.html`.

Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL `http://dl.acm.org/citation.cfm?id=1624775.1624804`.

Jan Rune Holmevik. Compiling SIMULA: a historical study of technological genesis. *IEEE Annals of the History of Computing*, 16(4):25–37, 1994.

IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

Inria Sophia-Antipolis. Bigloo homepage. URL: `http://www-sop.inria.fr/indes/fp/Bigloo/`, Sep 2011.

Alan Kay. The Early History of Smalltalk. *SIGPLAN Not.*, 28(3):69–95, 1993. ISSN 0362-1340. doi: 10.1145/155360.155364.

Richard Kelsey and William Clinger. Fifth Revised Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.4808.

Chris Lattner. The LLVM Compiler Infrastructure. URL: http://llvm.org/, 2011.

Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Code Generation and Optimization, IEEE/ACM International Symposium on*, 0:75–86, 2004. doi: 10.1109/CGO.2004.1281665.

John McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962. ISBN 0262130114.

John McCarthy. History of Lisp, February 1979. URL http://www-formal.stanford.edu/jmc/history/lisp.html.

Kristen Nygaard and Ole-Johan Dahl. The Development of the SIMULA Languages, 1981.

Kent Pitman. The Common Lisp HyperSpec, 1996. URL http://www.lispworks.com/documentation/common-lisp.html.

Christian Queinnec. Designing Meroon V3. http://pagesperso-systeme.lip6.fr/Christian.Queinnec/Papers/oopil.ps.gz, 1993.

Scheme Wiki. Scheme Frequently Asked Questions, January 2012. URL http://community.schemewiki.org/?scheme-faq-standards.

Manuel Serrano. Inline Expansion: When and How? In *International Symposium on Programming Languages Implementations, Logics, and Programs*, pages 143–147. Springer, 1997.

Manuel Serrano and Marc Feeley. Storage Use Analysis and its Applications. In *Proceedings, 1996 ACM SIGPLAN International Conference On Functional Programming*, pages 50–61, 1996. doi: 10.1.1.50.8423.

Manuel Serrano, Manuel Serrano, and Pierre Weis. Bigloo: A Portable and Optimizing Compiler for Strict Functional Languages, 1995.

Guy Lewis Steele Jr. and Gerald Jay Sussman. The Revised Report on Scheme, a Dialect of Lisp. MIT Artificial Intelligence Memo 452, January 1978.

Gerald Jay Sussman and Guy Lewis Steele Jr. The First Report on Scheme Revisited. *Higher Order Symbol. Comput.*, 11:399–404, December 1998. ISSN 1388-3690. doi: 10.1023/A:1010079421970.

David Anthony Terei. Low Level Virtual Machine for Glasgow Haskell Compiler. Bachelor's thesis, University of New South Wales, October 2009.