A New Backend for Standard ML of New Jersey

Kavon Farvardin
Computer Science
University of Chicago
Chicago, IL, USA
kavon@farvard.in

John Reppy
Computer Science
University of Chicago
Chicago, IL, USA
jhr@cs.uchicago.edu

ABSTRACT

This paper describes the design and implementation of a new backend for the Standard ML of New Jersey (SML/NJ) system that is based on the LLVM compiler infrastructure. We first describe the history and design of the current backend, which is based on the MLRisc framework. While MLRisc has many similarities to LLVM, it provides a lower-level, policy-agnostic, approach to code generation that enables customization of the code generator for non-standard runtime models (*i.e.*, register pinning, calling conventions, *etc.*). In particular, SML/NJ uses a stackless runtime model based on continuation-passing style with heap-allocated continuation closures. This feature, and others, pose challenges to building a backend using LLVM. We describe these challenges and how we address them in our backend.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Functional languages**.

KEYWORDS

Code Generation, Compilers, LLVM, Standard ML, Continuation-Passing Style

ACM Reference Format:
Kavon Farvardin and John Reppy. 2020. A New Backend for Standard ML of New Jersey. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20), September 2–4, 2020, Canterbury, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3462172.3462191

1 INTRODUCTION

Standard ML of New Jersey is one of the oldest actively-maintained functional language implementations in existence [1, 7]. Much like the proverbial "Ship of Theseus," every part of the compiler, runtime system, and libraries has been reimplemented at least once, with some parts having been reimplemented half a dozen times or more.

The backend of the compiler is one such example. The original code generator translated a direct-style λ-calculus intermediate representation (IR) to Motorola 68000 and DEC VAX machine code [7]. Inspired by Kranz *et al.*'s work on the ORBIT compiler for Scheme [24, 25], Appel and Jim converted the backend of the compiler to use what they called a "Continuation-Passing, Closure-Passing Style" [3, 6].[1]

At the same time, additional machine-code generators were written for the MIPS and SPARC architectures. With the proliferation of *Reduced-Instruction-Set Computers* (RISC) in the early 1990's, there was a need for even more backends. These code generators also suffered from the problem that they did not share code, each was a standalone effort, and that they did not support many machine-code-level optimizations. These problems lead to the development of MLRisc [21] as a portable machine-code generator for SML/NJ. MLRisc defines an abstract load-store virtual-machine architecture that sits between the language-specific parts of the code generator and the target-machine-specific parts, such as instruction selection, register allocation, and instruction scheduling. Over the past 25 years, MLRisc has been used to support roughly ten different target architectures in the SML/NJ system. It has also been used by several other compilers [15–17] and as a platform for research into advanced register allocation techniques [5, 20] and SSA-based optimization [29].

Unfortunately, MLRisc is no longer under active development,[2] so we need to consider alternatives. An obvious choice is the LLVM project, which provides a portable framework for generating and optimizing machine code [26, 27]. LLVM takes a language-centric approach to code generation by defining a low-level SSA-based [12] language, called LLVM IR, for describing code. LLVM IR has a textual representation, which we refer to as LLVM assembly code, as well as a binary representation, called *bitcode*, and a procedural representation in the form of a C++ API for generating LLVM IR in memory. The LLVM framework includes many analysis and optimization passes on both the target-independent LLVM IR and on machine-specific code. Most importantly, it supports the operating systems and architectures that SML/NJ supports, as well as some that we want to support in the future. While LLVM was originally developed to support C and C++ compilers, it has been used with varying degrees of success by a number of functional-language implementations [13, 14, 28, 34, 39, 40].

This paper describes our work building a new backend for SML/NJ using the LLVM infrastructure. We describe the challenges faced by this migration and how we addressed these challenges. While there are many similarities between this effort and previous applications of LLVM to functional-language compilers, there are also a number of novel aspects driven by the SML/NJ runtime model and compiler architecture. We conclude with an evaluation of our

[1] This CPS IR, with modifications to support multiple precisions of numeric types [18] and direct calls to C functions [9], continues to be used in the backend of the SML/NJ compiler.

[2] The last significant work was the addition of support for the x86-64 (a.k.a., amd64) architecture.

backend as well as some reflections on the pros and cons of using LLVM.

2 STANDARD ML OF NEW JERSEY

The Standard ML of New Jersey (SML/NJ) system provides both interactive compilation in the form of a *Read-Eval-Print Loop* (REPL) and batch compilation. In both cases, SML source code is compiled to binary machine code that is either loaded into a heap-allocated code object for execution or written to a file. Linking is handled in the elaborator, which wraps each compilation unit with a λ-abstraction that closes over its free variables; this code is then applied to the dynamic representation of the environment to link it. Dynamically, a compilation unit is represented as a function that takes a tuple of bindings for its free variables and returns a tuple representing the bindings that it has introduced. Thus, the SML/NJ system does not need to understand system-specific object-file formats or dynamic linking.

In the remainder of this section, we first describe SML/NJ's runtime conventions at an abstract level, then discuss the existing backend implementation, and the MLRisc-based machine-code generator.

2.1 Runtime Conventions

As described by Appel [2, 3], SML/NJ has a runtime model that can be described as a simple abstract machine (called the CMachine). The CMachine defines a small set of special *registers* to represent its state; these are:

- **alloc** is the allocation pointer, which points to the next word to allocate in the nursery.
- **limit** is the allocation-limit pointer, which points to the upper limit of the nursery minus a buffer of 1024 words. This buffer, which is called the *allocation slop*, allows most heap-limit tests to be implemented as a simple comparison.
- **store** is the store-list pointer, which points to a list of locations that have been modified since the last garbage collection (*i.e.*, it implements a write barrier).
- **exnptr** is the current-exception-handler pointer, which points to a closure representing the current exception handler.
- **varptr** is the var pointer, which is a global mutable location that can be used to implement features such as thread-local storage [30].
- **base** is the base-pointer register, which points to the beginning of the code object that holds the currently executing function. It is used to compute code addresses in a position-independent way.[3]

The **alloc** register is always mapped to a hardware register, the other special registers are either mapped to dedicated hardware registers or else represented by stack locations. For example, on the x86-64 target, which has 16 general-purpose registers, the **alloc**, **limit**, and **store** registers are mapped to hardware registers, but the **exnptr** and **varptr** are represented by stack locations. The first five of these registers (**alloc**, **limit**, **store**, **exnptr**, and **varptr**) are live throughout the execution of SML code and, thus, are implicitly passed as parameters across calls.

Table 1: CMachine general purpose registers

std-link	holds address of function for standard calls
std-clos	holds pointer to closure object for standard calls
std-cont	holds address of continuation
std-arg	first general-purpose argument register
misc$_i$	miscellaneous argument registers (including callee-save registers)

In addition, the compiler assumes that intermediate results, arguments to primitive operations, and arguments to function calls are always held in registers. The CMachine registers are assigned specific roles in the calling conventions as described in Table 1. Function calls come in three forms:

Standard function calls are calls to "escaping" functions that use a standard calling convention.[4] The first three arguments of a standard function call are the function's address (**std-link**), its closure (**std-clos**), and return continuation address (**std-cont**). Following these arguments are k callee-save registers [8] (typically $k = 3$), which are assigned to the first k miscellaneous registers ($\mathbf{misc}_0, \ldots, \mathbf{misc}_{k-1}$). The remaining arguments correspond to the user arguments to the function and are mapped to registers by type; *i.e.*, pointers and integers are assigned to **std-arg**, $\mathbf{misc}_k$, $\mathbf{misc}_{k+1}$, *etc.*, and floating-point arguments are assigned to floating-point registers.

Standard continuation calls are calls to "escaping" continuations. The first argument is the continuation's address and is assigned to the **std-cont** register; it is followed by the k callee-save registers, some of which are used to hold the continuation's free variables. The remaining arguments to the continuation are mapped to registers in the same way as for standard functions.

Known function calls are "gotos with arguments" [37] that represent the internal control flow (loops and join points) in a standard function or continuation. Because the code generator knows both the entry and call sites for known functions, it is able to arrange for arguments to be passed in registers without unnecessary copying [20].

To illustrate how these conventions are realized in the CPS IR, consider the following trivial SML function:

```
fun f x = if (x < 1) then x else f (x-1);
```

The first-order CPS is a single cluster consisting of two CPS functions as shown below.

```
fun f (link, clos, k, cs1, cs2, cs3, arg) =
   lp (arg, k, cs1, cs2, cs3)

and lp (arg, k, cs1, cs2, cs3) =
   if i63.>=(arg, 1) then
      let val tmp = isub63(arg, 1)
      in lp (tmp, k, cs1, cs2, cs3) end
   else
```

[3] The two architectures that we are currently supporting with our new LLVM backend (Arm64 and x86-64) both support PC-relative addressing, so we do not need the base pointer. The MLRisc backend, however, does not take advantage of such addressing modes.

[4] In SML/NJ, which does not do any kind of sophisticated control-flow analysis, escaping functions are those where at least some call sites or targets are statically unknown.

```
        k (k, cs1, cs2, cs3, arg)
```

Here we have taken the liberty of using meaningful variable names and an SML-like syntax for readability. The function `f` is a standard function, so its first three parameters are held in the **std-link**, **std-clos**, and **std-cont** CMachine registers. The next three parameters are the three callee-save registers followed by the function's actual argument (`arg`) in the **std-arg** register. The `lp` function is internal to the cluster, so the compiler is free to arrange its parameters in any order. The loop terminates by invoking the return continuation (`k`) using a standard continuation call. Here the first argument to the call (`k`) will be held in the **std-cont** register, then come the callee-saves, followed by the function's result in the **std-arg** register.

The code generator must support one other calling convention, which is the convention used to invoke the garbage collector (GC) [19]. This convention is a modified version of the standard function convention that uses a fixed set of registers (**link**, **clos**, **cont**, the callee-saves, and **arg**) as garbage-collection roots. Any additional live data, including all non-pointer register values (*e.g.*, untagged integer and floating-point registers), are packaged up in heap objects that are referred to by the **arg** register.

When a heap-limit check fails, control jumps to a block of code to invoke the GC. This code sets up the fixed set of root registers (as described above), fetches the address of an assembly-language function from the stack and then does a standard call to the assembly code, which, in turn, transfers control to the runtime system. After the GC finishes, control is returned back to the GC-invocation code, which restores the live variables and resumes execution of the SML code. Note that the return from the GC involves the exact same set of fixed registers that are passed as arguments, which is how the updated roots are communicated back to the program.

2.2 The Backend

The SML/NJ backend takes a higher-order continuation-passing-style (CPS) IR and, via a sequence of optimizing and lowering steps, produces a first-order CPS IR.[5] Unlike most other compilers, including other CPS-based compilers, SML/NJ foregoes use of a stack to manage calls and returns. Instead, all return continuations are represented by heap-allocated closures. The first-order CPS IR makes these closures explicit in the form of records and record selection operations. Because the runtime model uses heap-allocated continuation closures to represent function returns, the stack is not used in the traditional way. Instead, the runtime system allocates a single large frame that is used for register spilling and holding additional runtime values.

Along with this first-order IR, the compiler computes additional metadata about where heap-limit checks are needed and about which calling conventions should be used. This metadata is stored in auxiliary hash tables.

A program in the CPS IR is a collection of functions that represent both user functions and continuations. The body of a function is a CPS expression (`cexp`), where the leaves of an expression are applications. Thus, a `cexp` in the first-order CPS IR, where functions are not nested, can be viewed as an extended basic block [31].

The phases of this backend are illustrated in Figure 1. We describe those passes that are directly affected by the design and implementation of the new backend.

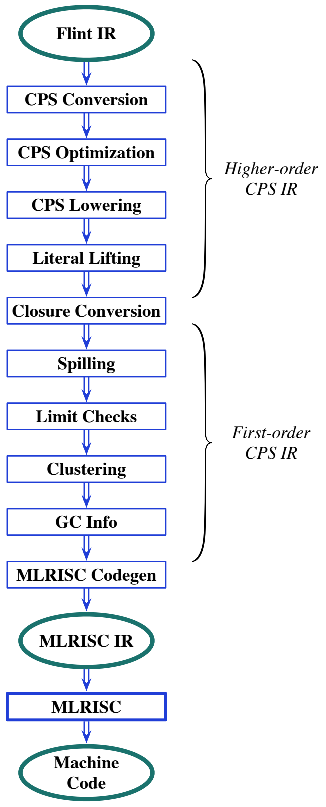

Figure 1: The existing backend

- The *CPS Lowering* phase is responsible for expanding certain primitive operations (primops) into lower-level code.
- The *Literal Lifting* phase lifts floating-point and string literals (as well as data structures formed from these literals) out of the code and replaces them with references to a per-compilation-unit tuple of literal values.
- The *Spilling* phase ensures that the number of live variables never exceeds the fixed-size spill area (1024 words).[6]
- The *Limit Checks* and *GC Info* phases are responsible for determining where heap-limit checks should be added and determining the live variables at those points. Allocation checks are placed at the entry to functions (both escaping and known) and continuations. As discussed above, most functions allocate fewer than 1024 words, so the allocation slop allows us to simply compare the allocation and limit pointer for these checks.
- The *Clustering* phase groups CPS functions into clusters, which are connected graphs of CPS functions where the edges correspond to *known* function calls. The entry nodes

[5] Note that while the invariants for the IR change with lowering, the actual representation as SML datatypes does not.

[6] Appel's original code generator used the spilling phase to ensure that the number of live variables did not exceed the available machine registers [3], but the switch to MLRisc, which has a proper register allocator, relaxed this constraint.

for a cluster are escaping functions and continuations; note that a cluster may have more than one entry.

2.3 MLRisc

The final step of the backend is to generate machine code using the MLRisc framework. MLRisc was designed to address many of the same problems as LLVM; it provides a low-level virtual machine based on a load-store (*i.e.*, RISC-like) model. More so than LLVM, MLRisc is a "mechanism, not policy," design leaving ABI issues such as calling conventions, stack layout, register usage, *etc.*, up to the compiler writer.[7] It makes heavy use of SML's functors to support specialization for both the target architecture and the source language. For example, the register allocator is defined by a functor that is parameterized over the spilling mechanism, which gives the compiler writer control over stack layout.

MLRisc's policy agnostic approach was heavily influenced by the needs of SML/NJ's runtime model. SML/NJ's stackless execution model meant that calling conventions could not be baked into the design. Likewise, use of dedicated registers for the allocation pointer, *etc.* in the standard calling conventions meant that MLRisc had to support some form of register pinning. The MLRisc register allocator is also able to handle the multi-entry functions that can arise from the clustering phase. Lastly, the need to generate binary machine code meant that MLRisc required an integrated assembler to resolve local branch offsets, but that it did not require a direct mechanism for generating object files.

3 CHALLENGES TO USING LLVM

LLVM was originally designed to support C and C++ compilers and, as such, maintains a significant architectural bias toward conventional runtime models. In this section, we enumerate some of the mechanisms that our MLRisc backend uses that do not have direct analogues in LLVM. We also discuss the challenges of incorporating a code generator written in C++ into a compiler written in SML. In this section, we are focused on the vanilla LLVM IR; as we describe in Sections 4 and 5, LLVM does provide ways to work around these limitations.

3.1 Comparing MLRisc and LLVM

MLRisc and LLVM are both designed to provide support for portable compilers. They are both based on a load-store model with an infinite supply of pseudo registers and a fairly standard set of basic instructions. A major difference, however, is that MLRisc abstracts over the instruction-set architecture, but not over the system ABI or runtime conventions. LLVM, on the other hand, has built in support for calling conventions, object-file formats, exception-handling mechanisms, garbage-collection metadata, and debugging information. Another major difference is in how they are used. While both systems define a virtual machine that a code generator can target, MLRisc only supports a procedural interface for code generation, whereas LLVM provides LLVM assembly, LLVM bitcode, as well as a procedural interface for code generation. The combination of builtin runtime conventions plus a textual representation of LLVM IR means that the only way to support different runtime models is to make changes to the LLVM implementation itself.

[7] It does provide some higher-level utilities, such as implementations of various C-language calling conventions;

3.2 Limitations of the LLVM Model

Many of the issues that we face are a consequence of the fact that LLVM abstracts away from the runtime model to a much greater degree than MLRisc.

No direct access to hardware registers. Ths SML/NJ runtime model relies on being able to map key CMachine registers, such as the allocation pointer, to dedicated hardware registers for efficient execution. Unlike MLRisc, LLVM does not provide a direct mechanism for mapping variables to specific hardware registers.

No direct access to the stack. SML/NJ uses specific slots in the stack frame to communicate information from the runtime system to the SML execution (*e.g.*, the address of the `callGC` function). Some CMachine registers on some targets are also represented by stack locations. In LLVM, however, the layout of a function's stack frame is largely opaque at the LLVM IR level and there is no way to specify access to specific locations in the stack.

Builtin calling conventions. As described in Section 2.1, SML/NJ defines its own register-based calling conventions that do not involve the stack in any way, as well as a stack-based convention for invoking the garbage collector. The **`call`** instruction in LLVM is a heavyweight operation that embodies the policy defined by its calling convention. While LLVM has a number of predefined calling conventions, including several language-specific ones, there is not a good match for the SML/NJ runtime. Defining a different convention requires modifying the LLVM source and recompiling the LLVM libraries.

Multi-entry-point functions. The clustering phase of the SML/NJ backend produces clusters that can have multiple entry points. For example, compiling the following function that walks over a binary tree

```
fun walk Lf = ()
  | walk (Nd(l, r)) = (walk l; walk r)
```

will produce a cluster for `f` with two entries: a standard function for calling `f` on the root or left subtree and a second continuation entry for calling `f` on the right subtree. While it is natural to think of mapping clusters to LLVM functions; LLVM functions are restricted to a single entry point.

Tail-call overhead. Efficient tail calls are critical to performance, since all calls in CPS are tail calls. While LLVM provides a tail-call optimization (TCO), its primary purpose is to avoid stack growth. Even when TCO is applied to a function call, the resulting code incurs the overhead of deallocating the caller's frame and then allocating a fresh frame for the callee.

Trapping arithmetic. The semantics of integer arithmetic in SML require that the `Overflow` exception be raised when the result exceeds the representable range of the type [18]. MLRisc supports this requirement by defining *trapping* versions of the arithmetic operations, with the semantics that an appropriate trap is signaled on overflow. The runtime system handles this trap by mapping it to a control transfer to the exception handler continuation. While LLVM provides intrinsic functions for integer arithmetic with overflow, it does not provide a mechanism for generating the appropriate trap.

Support for position-independent code. The machine code that SML/NJ uses must be position independent. We achieve this property by using the base pointer to compute absolute addresses from relative offsets, both for evaluating labels and for jump tables. While LLVM also supports position independent code, it does so by relying on a dynamic linker to patch code when it is loaded.

3.3 Integrating LLVM into the Compiler

There are two approaches to using LLVM as a backend for a compiler. The first, which is most common, is to generate LLVM assembly code into a text file and then use the command-line toolchain to convert that to a target-machine object file.[8] This approach has the advantage that it does not require a foreign-function mechanism to communicate between the compiler and LLVM. The downside, however, is that it adds significant overhead in the form of formatting textual output, parsing said output, and running subprocesses. For an interactive compiler, such as SML/NJ's REPL, this approach also requires using system-specific dynamic linking to load and execute the code that was just generated.

The other approach is to use LLVM's C++ APIs to construct a representation of the program directly, which can then be optimized and translated to machine code. This approach, which is used by industrial compilers, such as the **clang** C/C++ compiler, is similar to what we currently do with MLRisc, but it poses its own challenges. First of all, the C++ API for LLVM relies heavily on inline functions, which cannot be called from foreign languages. As an alternative, there is a C language wrapper for the C++ API that can be used, but it is less efficient than the C++ API and has a reputation of lagging behind changes in the C++ API. Another problem is the sheer volume of foreign calls that would be required for code generation. Given that foreign function calls in many functional-language implementations, including SML/NJ, are relatively expensive, this volume can add measurable overhead to code generation. Thus, the problem of efficient communication between the compiler and the code generator is a challenge for using LLVM as a library.

Another challenge with using LLVM for SML/NJ is that it produces object files (the specific object-file format depends on the system). For implementations that use traditional linking tools, this property is not an issue, but for a system like SML/NJ that works with raw code objects, it is necessary to both patch relocation information and to extract the code from the object file.

4 DESIGN OF THE NEW BACKEND

In order to use LLVM in the SML/NJ system, we need solutions to the two broad challenges described above: how to support the SML/NJ runtime model in LLVM (Section 3.2) and how to integrate a LLVM-based backend into a compiler written in SML (Section 3.3).

[8] Typically, this toolchain involves using **llc** to generate native assembly code and then running an assembler to produce object code.

4.1 Runtime conventions

Function entries and call sites are the key places where we need to guarantee that our register conventions are being followed, elsewhere in the function we can let the register allocator dictate where information is held. Thus, by modifying LLVM to add a new calling convention, we can dictate the register usage at those places. In previous work for the Manticore system [13], we described a new calling convention for LLVM, called *Jump With Arguments* (JWA), that can be used to support the stackless, heap-allocated-closure runtime model used by both Manticore and SML/NJ. The JWA calling convention has the property that it uses almost all of the available hardware registers for general-purpose parameter passing.[9] The convention also has the properties that no registers are preserved across calls and that the return convention uses exactly the same register convention as calls.

We furthermore mark every function with the `naked` attribute, which tells LLVM to omit generating the function prologue and epilogue.[10] Thus the function runs in whatever stack frame exists when it is called, which fits the SML/NJ model of a single frame shared by all SML code.

There is one minor complication, which is that we actually have several different conventions to support (*i.e.*, escaping and known functions, continuations, and GC invocation). While we could define multiple LLVM conventions, we can make them all fit within the JWA convention by careful ordering of parameters and by using LLVM's undefined values for registers that are not part of a particular convention (*e.g.*, the **link** and **clos** registers when throwing to a `STD_CONT` fragment).

4.2 Integrating LLVM into SML/NJ

Replacing MLRisc with LLVM raises the question of how to connect the SML/NJ compiler, written in SML, with an LLVM code generator, written in C++. Previous functional-language implementations have generated LLVM assembly code and used a command-line toolchain to translate that into object code, but we decided that this approach was not a good fit for SML/NJ. Specifically, we were concerned about compilation latency, since the interactive REPL is a central part of the SML/NJ system, and about the extra dependencies on executables that we would have to manage. Therefore, we decided to integrate the LLVM libraries into the runtime system.

Having decided to directly generate LLVM code in memory, there was the question of how to do that efficiently. Fortunately, the problem of how to connect compiler components that are implemented in different languages was addressed many years ago as part of the *Zepher Compiler Infrastructure Project* [41], which defined the *Abstract Syntax Description Language* (ASDL) for specifying compiler intermediate representations and a tool (**asdlgen**) for generating picklers and unpicklers in multiple languages. The original **asdlgen**

[9] For SML/NJ, we use the same register convention that is used in the existing MLRisc backend. On the x86-64, we omit the stack pointer and one scratch register from the convention, which leaves 14 registers available for parameter passing.

[10] The function prologue and epilogue is where the function's stack frame is allocated and deallocated.

tool does not support modern C++, so we built a new implementation of the tool that generates code that is compatible with LLVM.[11]

Thus, our original plan was to pickle the CPS IR on the SML side and thenb pass it to the runtime as input to a LLVM-based code generator that would essentially be a C++ rewrite of the existing MLRisc code generator. The resulting machine code would then be returned to the SML code as an array of bytes. As we began work on this approach, however, we discovered that the CPS IR was not the right IR for connecting to LLVM. First, the MLRisc code generator depended heavily on metadata that was external to the CPS IR. Second, the CPS primops were designed to model the corresponding SML operations (*e.g.*, addition on tagged integers), which added a lot redundancy and extra work to the code generation process. Thus, we decided to introduce a new, lower-level IR, that would be the vehicle for communicating with the LLVM-based code generator. This new IR, which we call the CFG IR, is described in detail in the next section, but its key features are that it is self-contained and that its semantics are much closer to both the semantics of LLVM and MLRisc.

4.3 The New Backend Pipeline

We conclude this section with a description of the new backend pipeline, which is illustrated in Figure 2. We have greyed out the labels of those passes from Figure 1 that are unchanged, but, for some passes, changes were required.

- The CPS Lowering phase is expanded to lower more CPS primops than before. These changes avoid some primops that are difficult to translate directly to LLVM.
- The Clustering phase is modified to avoid multi-entry-point clusters by introducing new CPS functions.
- The tracking of information about GC invocations is modified to work with the CFG code generator (discussed below in Section 6.6).
- The *CFG Codegen* phase replaces the old MLRisc Codegen phase.

The new code generation path first pickles the CFG IR and then passes the linearized representation to the runtime system where it is unpickled into a C++ representation of the CFG IR. We then generate LLVM IR code using a version of LLVM extended with the JWA calling convention. The next two sections describe the CFG and LLVM code generator in detail.

5 THE CFG REPRESENTATION

A major part of the new backend is the CFG IR that sits between the existing first-order CPS IR and the MLRisc and LLVM code generators. The CFG IR encodes many of the invariants of the CPS IR into its representation and makes the metadata required for code generation explicit. The main datatypes used to represent the CFG IR are shown in Figure 3; we omit the primitive operators and have simplified the types slightly for space and presentation reasons.

Each unit of compilation (*i.e.*, a definition typed into the REPL or a source file) is mapped to a CFG compilation unit, which consists of a list of clusters. The first cluster in the list is the entry cluster,

[11] The original implementation is still available at http://asdl.sourceforge.net; the new implementation, which currently only supports SML and C++ is included in the SML/NJ distribution.

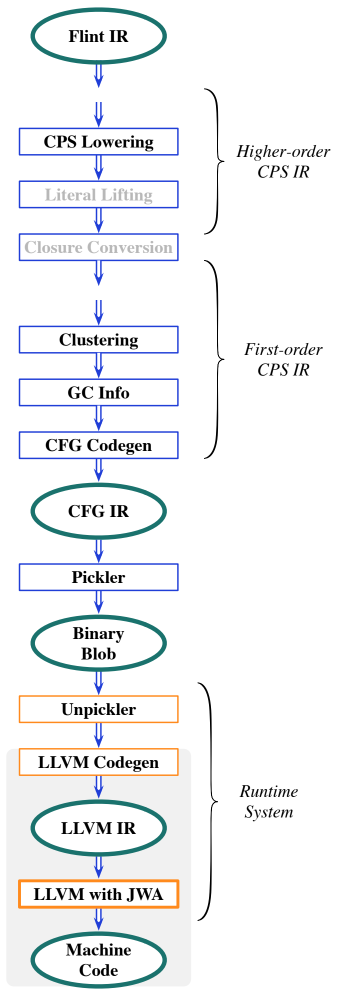

Figure 2: The new backend. Components represented by orange boxes are implemented in C++.

which handles linking the new code with the existing dynamic environment.

5.1 Clusters

CFG clusters roughly correspond to the clusters used in the MLRisc backend; each cluster consists of a list of *fragments*, which are extended basic blocks. Clusters also have attributes, which capture some basic information about the code in the cluster, such as does it require the base pointer register or does it contain trapping arithmetic operations?

Clusters are created by identifying collections of CPS functions that are connected by control transfers. It is frequently the case that these collections will have multiple entry points (an example was given in Section 3.2). For MLRisc, this property is not a problem, since it can handle multi-entry-point functions, but LLVM restricts functions to single entry.

Addressing this problem is one of the more complicated parts of the translation to the CFG IR. Once we have identified a connected collection of CPS functions, we have to normalize it into one or more single-entry-point clusters.

One complication for this normalization phase is that the new clusters may require access to the base pointer in order to compute label values. The original calls to these new clusters are unlikely to have the cluster's address as a parameter, since they are not standard calls. Thus, we have to change the calling convention slightly in these cases by adding the base pointer as an additional parameter.[12]

Normalization also constrains the flexibility of the register allocator, since internal control-flow edges (*i.e.*, *gotos*) are replaced with tail calls that use the JWA convention. A better approach may be to use the node-splitting (really cloning) techniques used to handle irreducible control-flow graphs [22] as a way to preserve some of the internal control flow in the case of loops.

5.2 Expressions and Statements

CFG expressions (`exp`) and statements (`stm`) are used to define the computations that make up the body of fragments. While the constructors of these datatypes are in close correspondence to the CPS IR, there are some important differences.

First, pure expressions are represented as trees (the `exp` type), instead of having each primitive operation be bound to a `lvar`. Sharing of common expressions is made explicit by the `LET` constructor. Using expression trees has a couple of advantages: it reduces the size of CFG terms, which speeds pickling, and expression trees match the procedural code-generation interfaces of both LLVM and MLRisc.

Operations in the CFG IR are closer to machine level than those of the CPS IR. For example, the default integer type in SML is represented by a tagged value that has its lowest bit set (*i.e.*, the integer n is represented as $2n + 1$). Arithmetic on tagged integers requires various additional operations to remove and add tags. In the old backend, these were added when generating MLRisc code; we now generate these operations as part of the translation to CFG. The CFG IR also replaces many specialized CPS operations for memory allocation and access with a smaller number of lower-level operators.

Figure 3 also shows the representation of types in the CFG IR. The types `LABt` (code addresses), `PTRt` (pointers or tagged values), and `TAGt` (tagged values) describe values that the garbage collector can parse and thus can be in a GC root. The other two types represent raw numeric data (integer and floating-point) of the specified size in bits. We map the `LABt` and `PTRt` types to the LLVM **`i64*`** type (**`i32*`** on 32-bit machines). The `TAGt` type is mapped to **`i64`**, while the `INTt` and `FLTt` types are mapped to the LLVM integer and float types of the specified size. We do not try to use LLVM's aggregate types to model heap allocated objects, since we usually only have that level of type information at the point of allocation.

5.3 Metadata

The other major difference between the CPS and CFG IRs is that the metadata for calling conventions and GC support has been incorporated into the CFG IR, instead of being held in external tables. This change makes transferring the information to the LLVM code generator much simpler, since we do not have to define a pickle format for the hash tables used to track the data.

[12] The presence of this extra parameter is recorded as part of the cluster's attributes instead of being explicitly represented in the IR.

```
datatype ty
  = LABt | PTRt | TAGt
  | NUMt of {sz : int} | FLTt of {sz : int}

type param = lvar * ty

datatype exp
  = VAR of {name : lvar}
  | LABEL of {name : lvar}
  | NUM of {iv : IntInf.int, sz : int}
  | LOOKER of {oper : looker, args : exp list}
  | PURE of {oper : pure, args : exp list}
  | SELECT of {idx : int, arg : exp}
  | OFFSET of {idx : int, arg : exp}

datatype stm
  = LET of exp * param * stm
  | ALLOC of alloc * exp list * lvar * stm
  | APPLY of exp * exp list * ty list
  | THROW of exp * exp list * ty list
  | GOTO of lvar * exp list
  | SWITCH of exp * stm list
  | BRANCH of branch * exp list * stm * stm
  | ARITH of arith * exp list * param * stm
  | SETTER of setter * exp list * stm
  | CALLGC of exp list * lvar list * stm

datatype frag_kind
  = STD_FUN | STD_CONT | KNOWN | INTERNAL

datatype frag = Frag of {
    kind : frag_kind,
    lab : lvar,
    params : param list,
    body : stm
  }

type attrs = { ... }

datatype cluster = Cluster of {
    attrs : attrs, frags : frag list
  }

type comp_unit = cluster list
```

Figure 3: The main CFG types

The calling-convention metadata is represented by three aspects of the IR:

(1) Fragments are annotated with a `frag_kind`; `STD_FUN` for escaping functions, `STD_CONT`, for continuations, and `INTERNAL` for internal known function calls. The `KNOWN` kind is used

for the functions that are introduced to avoid multiple entry-points during clustering.

(2) We use three different application forms: `APPLY` for functions, `THROW` for continuations, and `GOTO` for internal jumps. Calls to `KNOWN` functions are represented by an `APPLY` where the function is specified by a `LABEL` value.
(3) The `APPLY` and `THROW` constructs include the type signature of their arguments, which is necessary because LLVM requires a type signature for the function in a **`call`** instruction.

Garbage collection tests and invocations are explicit in the CFG IR. We describe this mechanism in more detail below in Section 6.6.

5.4 C++ Representation

The CFG IR is defined using the ASDL specification language [32], which provides language-independent mechanisms for specifying inductive types similar to those found in most functional programming languages. From this specification, we generate both the SML and C++ representations of the IR, as well as the pickling/unpickling code needed to communicate CFG values from SML to our LLVM code generator. As would be expected, the mapping from ASDL to SML types is straightforward. For C++, most types are represented as classes, but enumerations (*e.g.*, `frag_kind` in Figure 3) are mapped to C++ **`enum`** types. Sum types are represented with an abstract base class for the type and subclasses for the constructors.

6 IMPLEMENTATION DETAILS

In this section, we describe the LLVM code generator (*i.e.*, the orange boxes in Figure 2) in more detail. Our original implementation was targeted toward the x86-64 architecture, but we have recently ported the code generator to also support the Arm64 architecture. Supporting a second target required very few changes to the code generator; the bulk of the work was in adding JWA convention support for Arm64 to LLVM.

6.1 Modifying LLVM

Our code generator requires a modified version of LLVM. The changes are fairly straightforward. We add a description of our register-only calling convention to the specification of calling conventions for the target architecture. In addition, there are a number of places in the architecture-specific parts of LLVM where the calling convention is queried that require small additions or edits. The complete details of the patches are described in a developer note [33].

6.2 LLVM Code Generation

As described above, the `exp` and `stm` types in the CFG IR are represented as abstract classes in C++, with each constructor its own subclass. Code generation is implemented as a two-pass walk over the CFG IR. The first pass collects information, such as a mapping from labels to clusters and fragments, and allocates placeholder objects, such as LLVM functions for clusters, LLVM ϕ-nodes for `INTERNAL` fragments, and LLVM basic blocks for the arms of `BRANCH` and `SWITCH` statements. The second pass walks the representation generating LLVM code.

ASDL provides a mechanism for adding methods to the generated classes. For the `cluster`, `frag`, and `stm` classes, we define a virtual `init` method for the initialization pass. We also define a virtual `codegen` method for these classes and for the `exp` and various primitive operator classes. Dispatching on the constructor of a sum type is implemented using the standard object-oriented pattern of virtual-method dispatch.

The code generation process requires keeping track of a significant amount of state, such as the current LLVM module, function, and basic block, and maps from `lvar`s to their LLVM representations. We define a `code_buffer` object to encapsulate this information. This object also contains the implementation of various utility methods to support the calling conventions and GC invocation. Code generation for most of the CFG IR is straightforward, but we explain how we address the challenges of Section 3 and some other details in the sequel.

6.3 ϕ Nodes

LLVM's language is a Static-Single-Assignment (SSA) IR [12]. As the name suggests, variables (a.k.a. pseudo registers) in SSA are assigned to only once. When control flows into a block from multiple predecessors, it is necessary to introduce ϕ nodes, which make explicit the merging of values from multiple sources. Generating the SSA form from the CFG IR is quite straightforward.[13] During the initialization pass, we preallocate ϕ nodes for each `INTERNAL` fragment in a cluster. We define one ϕ node per fragment parameter plus additional nodes for those special registers that are mapped to hardware registers (*e.g.*, **alloc**, **limit**, *etc.*). When compiling a `GOTO` statement, we record the current values of the special registers and the values generated for the `GOTO`'s arguments in the ϕ nodes of the target fragment.

6.4 Stack References

As discussed in Section 3.2, we need to be able to generate references to specific locations in the stack frame. We have experimented with several possible mechanisms for accessing stack locations. Our first attempt was the **`@llvm.frameaddress`** intrinsic, but it requires using a frame pointer, which burns an additional register. We then took the approach of defining native inline assembly code for reading and writing the stack. This approach produced the desired code, but also introduced target-dependencies in the code generator. We finally settled on using the **`@llvm.read_register`** intrinsic to read the stack pointer.

One change that we had to make to our runtime model is the layout of the frame used to host SML execution. In the existing MLRisc code generator, the spill area is in the lower part of the frame and the locations used to represent special registers, *etc.* are in the upper part of the frame (close to to the top of the stack).[14] LLVM, however, follows the opposite convention and puts the spill area at the top of the frame. Therefore, we have switched the layout of the frame as of version 110.99 of SML/NJ. Fortunately, MLRisc's flexibility made this change easy to implement.

[13] As has been observed by others [4, 11, 23], there are strong similarities between λ-calculus IRs (especially CPS) and SSA form.

[14] We are using the terms "upper" and "lower" with respect to the direction of stack growth. Since the stack grows *down*, this means that the address of the lower part is *greater* than the upper part.

6.5 Position-independent Code

While any CFG compilation unit is closed with respect to the code labels it references, we need to be able to convert these labels to actual code addresses at runtime so that we can build function closures. As described in Section 2.1, the MLRisc code generator does this by explicitly maintaining a pointer to the beginning of the current module. For example, if the first function in a module has label l_0 and we have a standard function f with label f_l, then we can compute the base pointer by **base** = **link** − $(l_f - l_0)$, where $(l_f - l_0)$ is a compile-time constant. While MLRisc resolves these offsets prior to machine-code generation, LLVM does not. Instead, it generates relocation information that must be used to patch the machine code (see Section 6.8).

A related issue is supporting the `SWITCH` statement. Our compiler guarantees that a `SWITCH` is exhaustive; *i.e.*, that if the `SWITCH` has n cases, then the argument will be in the range $0..n-1$. We currently translate this construct to LLVM's **`switch`** instruction, but that is sub-optimal because LLVM's instruction does not assume exhaustiveness and requires a default case. The resulting machine code does a conditional test of the argument to see if it is in range before indexing the jump table. As an alternative, we experimenting with constructing an explicit jump table as a constant array value using LLVM's block addresses and the **`indirectbr`** instruction. This approach is more complicated to implement, but should produce slightly faster code.

6.6 Invoking GC

As described in Section 2.1, invoking the GC requires a fair amount of bookkeeping to preserve live data across the invocation. What makes it complicated is the combination of different cases that have to be managed. For example, a `STD_CONT` fragment does not use the **std-link** or **std-clos** registers, so these are either used to hold excess parameters or else must be nullified before the collection. Our original implementation handled the generation of this bookkeeping code in the C++ code generator, but the resulting implementation was both lengthy and complicated. While the MLRisc code generator also dealt with this complexity, it is a problem that is much easier to solve in SML than C++. We subsequently realized that a better strategy is to encode the GC invocation code in the CFG IR. To this purpose, we added a heap-limit check as a `branch` primop and the `CALLGC` statement form. The translation from CPS to CFG handles the generation of code to invoke the GC, as well as inserting the limit checks into the IR. In addition to moving complexity out of the C++ code generator, this approach also allows us to share the implementation of the GC invocation protocol between the LLVM and legacy MLRisc machine-code generators.

We also implement a feature of the MLRisc code generator that shares implementations of the GC invocation code between multiple `STD_FUN` and `STD_CONT` fragments. Because the parameters of these fragments are in known locations and the code address of these fragments are in known registers (*i.e.*, **std-link** or **std-cont**), we can move the invocation code into a function that can then be shared. Measurements done when the GC API was originally designed show that over 95% of `STD_FUN` GC invocations can be covered by just five different invocation functions, while almost 95% of `STD_CONT` GC invocations can be covered by just one invocation function [19].

The actual invocation of the GC uses a non-tail JWA call to a code address that is stored at a known stack location. We use the JWA calling convention so that the GC roots are in predictable registers and we mark the call as a non-tail call so that the runtime can return to the GC invocation code. The return type of the call is a struct with fields for each of the GC roots (recall that the JWA call uses the same register assignment for calls and returns). These are then bound to the variables specified by the `CALLGC` statement.

6.7 Trapping arithmetic

Our support for SML's integer arithmetic, which raises the `Overflow` exception on overflow, has gone through several design iterations. LLVM provides "arithmetic with overflow" intrinsic functions that return a pair of a result and an overflow bit. In the generated LLVM code, we test the overflow bit and jump to code that causes the `Overflow` exception to be raised. The need for this conditional control flow is one of the reasons why trapping arithmetic is represented as a `stm` in the CFG IR.

The mechanism for actually raising the `Overflow` exception has been the challenge. Ideally, we could just generate code to raise the exception and be done with it. Unfortunately, by the time that we get to code generation, we no longer have access to the environment information that would allow us to get the `Overflow` exception.

The MLRisc code generator uses a hardware trap instruction (*e.g.*, '**`int`** `4`" on the x86-64) to transfer control to a runtime-system signal handler, which then dispatches the current exception-handler function. We initially used LLVM's inline assembly-code mechanism to emulate the approach of the MLRisc code generator. This approach, however, adds dependencies on both the target architecture and the operating system to the implementation. We have since realized that a more portable approach is to use the same scheme that we use for invoking the garbage collector; *i.e.*, a call into the runtime system via an address stored in the stack frame. The runtime system can then raise the exception as before, but it does not require an OS-specific signal handler and the LLVM code generator does not require architecture-specific assembly code.

The other issue that we encountered is that the special CMachine registers must be both live and bound to their designated hardware locations at the time of the call to the runtime system (or trap in the earlier implementation). We can use the JWA calling convention for this purpose. Our final implementation uses a per-cluster basic block that does a non-tail JWA call to the runtime system.[15]

6.8 Just-in-Time Compilation

LLVM provides rich support for just-in-time (JIT) compilation, but its JIT infrastructure is primarily focused on the problems of multi-threaded compilation, compilation on demand, and dynamic linking. While multi-threaded compilation is a feature that we might want to explore in the future, we already address the problems of compilation on demand and linking in SML/NJ. Therefore, we use the batch compilation infrastructure, but specify an in-memory

[15] We do a non-tail call so that we have access to a code address in the code object containing the overflowing instruction. This address is used to include the source-file name in the location information for the exception.

output stream for the target of the machine-code generator to produce an in-memory object file.

As noted above, even though the compilation unit does not refer to any external labels, LLVM's assembly process does not resolve the offsets used in PC-relative addressing. Resolving the relocation entries introduces some architecture and object-file-format dependencies to the final step of code generation. A couple of examples are

- Floating-point negation and absolute-value instructions are implemented using bitmasks on the x86-64; these bit masks are in a data segment that immediately follows the text segment, but which has a different name in the MACH-O and ELF formats. Additionally, MACH-O uses offsets from the beginning of the code whereas ELF uses offsets relative to the beginning of the data segment to specify the relocation offset.
- When constructing code addresses (*e.g.*, to define closures), the 32-bit PC-relative offset is in its own four bytes of the instruction on the x86-64, but is split into two pieces that are embedded in two different instructions on the Arm64.

We are, however, able to use LLVM's generic object-file API to implement the code patching; we just include some conditional logic based on the target platform. After object-code generation, we identify the text segment and any data segments that need to be included and copy them to a heap-allocated code object. We then patch the PC-relative offsets and return the code object to the SML side of the compiler.

7 STATUS AND EVALUATION

We have integrated the LLVM code generator for the x86-64 architecture into a version of the SML/NJ runtime system. A control flag allows one two switch between the MLRisc and LLVM backends from the REPL. The system can compile substantial amounts of SML code, including the SML/NJ compiler itself, but there are a few remaining issues to sort out before it is self supporting. In this section, we report on the performance of the LLVM backend as compared to the MLRisc backend. The measurements are presented in Table 2; these were gathered on an Apple iMac with a 3.6GHz Intel i9-10910 processor running macOS 10.15.7 Catalina. We used a release build of LLVM 10.0.1 (with support for the JWA calling convention). The LOC column specifies the results from the **cloc** program (https://github.com/AlDanial/cloc). The experiments consist of six benchmarks of varying size, plus compiling the SML/NJ compiler itself.

7.1 Compile time

The first set of numbers in Table 2 show the time (in seconds) to compile the example programs using the MLRISC backend vs. using the LLVM backend. These numbers are the average of five compiles of each benchmark. The measured times include all stages of the compilation. As expected, using LLVM to generate code is significantly slower than the MLRisc backend. While there is a lot of variability in the slowdown, it is fast enough to provide interactive responsiveness for even hundreds of lines of code. Furthermore, SML/NJ has an effective compilation manager [10] that supports incremental recompilation, so developing larger projects should still be reasonable fast.

We chose to use LLVM's procedural interface as a way to reduce code generation latency when compared to the more common approach of LLVM assembly code. To get a lower bound on the cost of the alternative, we dumped the LLVM assembly for a couple of example programs and then measured the time required to generate an object file from the LLVM assembly (*i.e.*, by running the **llc** program to generate assembly and then the assembler to generate object code). The time take for this process was typically about 50% more than the time required in our code generator to go from pickled CFG to object code. Furthermore, we believe that the time needed to generate the LLVM assembly code would likely significantly exceed the pickler time, since the LLVM assembly code is 15 times larger than the pickle file. Thus, we think that using the procedural interface was a good choice.

7.2 Execution Time

The second set of performance numbers in Table 2 show the execution times for our examples. These numbers are the average of ten runs of each benchmark. Based on our previous experience with the LLVM and MLRisc backends for Manticore [13], we expected to see noticeable improvements for floating-point code (*e.g.*, the **mandelbrot** and **mc-ray** benchmarks), which we do see. Surprisingly, we also saw some significant speed up for other benchmarks, but since the running times on these programs are so small, measurement error may be more of an issue.

We should also note that the LLVM generated code was using an implementation of the SML Basis Library that was compiled using MLRisc. Once the entire code is compiled using LLVM, we may see further improvements. All in all, however, we are happy to see performance of the code generated by LLVM to be faster than MLRisc. Anecdotally, other functional-language implementations have seen slow downs when switching to LLVM.

7.3 Compiler Size

LLVM is a large system and that is reflected in the size of our compiler. The SML/NJ Version 110.99 runtime system is about 12K bytes of code and the interactive system's heap file is a bit over 14M bytes. In contrast, the runtime system including the statically linked LLVM code generator is over 20M bytes of code and 50K bytes of data.

On the other hand, the C++ source code for our backend is fairly small: about 6,000 lines of code (not counting comments or blank lines), of which about 2,500 is generated by ASDL.

8 RELATED WORK

The Pure programming language[16] appears to have been the first functional language to use LLVM in its implementation (starting in 2008). The implementation of the Pure interpreter is in C++ and LLVM is described in the documentation as being used as a JIT compiler, but there is no published description of the implementation.

Terei and Chakravarty's LLVM-based code generator for the Glasgow Haskell Compiler (GHC) [38, 39] is probably the earliest attempt to use LLVM for a language with a non-standard runtime

[16] See https://agraef.github.io/pure-lang.

Table 2: Performance data

		Compile Time (sec)			**Run Time** (sec)		
Program	**LOC**	**MLRISC**	**LLVM**	**Slowdown**	**MLRISC**	**LLVM**	**Speedup**
mandelbrot	48	0.006	0.016	2.79	3.758	3.075	1.22
life	111	0.035	0.113	3.26	0.005	0.005	1.00
mc-ray	455	0.096	0.342	3.55	5.727	4.994	1.15
lexgen	1,032	0.256	0.828	3.24	0.020	0.019	1.07
vliw	3,033	1.050	2.710	2.58	0.044	0.031	1.40
hamlet	16,930	5.548	10.491	1.89	0.104	0.087	1.20
smlnj	157,094	47.836	143.440	3.00	n.a.	n.a.	n.a.

model. As such, they were the first to confront and solve a number of the technical issues we describe here. In particular, they faced the problem of how to map logical registers in their runtime model to specific machine registers. It appears that Chris Lattner, the creator of LLVM, suggested defining a new calling convention to implement this mechanism.[17] The GHC calling convention is now a supported convention in LLVM.

The ErLLVM pipeline is an LLVM-based backend for the HiPE Erlang compiler [34]. As with GHC, and our system, the problem of targeting specific machine registers is solved with a new calling convention; the HiPE convention is also part of the official LLVM distribution. Unlike GHC and SML/NJ, ErLLVM uses, with some adaptation, LLVM's builtin mechanisms for garbage collection support and exception handling. The ErLLVM pipeline generates LLVM assembly and then uses the LLVM and system tools to produce an object file. They then parse the object file to extract a representation that is compatible with the HiPE loader, which is similar to what we do in SML/NJ.

We know of two other ML implementations that have LLVM backends. The SML# system generates fairly vanilla LLVM assembly code and uses LLVM's existing `fastcc` calling convention [40]. To ensure that tail recursion is efficient, they added loop detection to their compiler and generate branches in these cases, instead of relying on LLVM's tail-call optimization.[18]

The MLton SML compiler also has a LLVM backend [28]. Their LLVM compiler is modeled on their backend that generates C code, so they do not have the problems of mapping specialized runtime conventions onto LLVM. As with GHC and ErLLVM, they generate LLVM assembly code; one difference, however, is that they stack allocate all variables and then rely on LLVM's `mem2reg` pass to convert to SSA.

Our work reported here has as its roots the development of the JWA calling convention for use in Manticore's Parallel ML (PML) compiler [13]. As with the other examples above, the PML compiler generates LLVM assembly and uses the **llc** tool to generate native assembly code. Because PML programs are linked using standard tools, the compiler does not require special handling of position-independent code or global addresses, such as the code to invoke the GC. It also does not require access to specific locations in the stack. While PML is a dialect of SML, it has a different semantics for arithmetic (*i.e.*, no `Overflow` exceptions), so it was not necessary to use LLVM's arithmetic with overflow intrinsics.

Recently, we have used the PML compiler to explore performance and implementation tradeoffs between different runtime strategies for representing continuations and the call stack [14]. The implementation of heap-allocated continuations in that study was the version from our previous work [13], which lacks the more sophisticated closure optimizations implemented by the SML/NJ compiler [8, 35, 36]. It will be interesting to revisit the experiments using our new LLVM backend for SML/NJ.

9 CONCLUSION AND FUTURE WORK

We have described a new LLVM-based backend for the SML/NJ system. The main benefit of using LLVM is that it allows us to leverage the significant efforts of the LLVM developers in supporting new target architectures. The modifications to LLVM to support our JWA calling convention are fairly small and the complexity of the LLVM backend is comparable to that of the MLRISC backend. We gain some improvement in the generated code (especially for floating-point code), but suffer a significant, but acceptable, hit in compile time. Another problem with using LLVM is that it is a rapidly changing system, so we hope to get the JWA calling convention integrated into LLVM.

We also think that the approach that we took toward generating LLVM code has worked out well. The syntax of LLVM assembly code is quite verbose and finicky, which can make generating it difficult.

We are working to fully integrate the LLVM backend into the SML/NJ system and we expect to start including it in the SML/NJ distribution by the end of Summer 2021. LLVM will enable future ports to new architectures, such as RISC-V, as well as exposing hardware features, such as vector registers, to SML programmers.

REFERENCES

[1] Andrew Appel and David B. MacQueen. 1991. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming (PLILP '91) (Lecture Notes in Computer Science, Vol. 528)*, J. Maluszynski and M. Wirsing (Eds.). Springer-Verlag, New York, NY, USA, 1–13. https://doi.org/10.1007/3-540-54444-5_83

[2] Andrew W. Appel. 1990. A Runtime System. *Lisp and Symbolic Computation* 3, 4 (Nov. 1990), 343–380. https://doi.org/10.1007/BF01807697

[3] Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press, Cambridge, England, UK.

[17] See http://nondot.org/sabre/LLVMNotes/GlobalRegisterVariables.txt.

[18] Recall from Section 3 that LLVM's tail-call optimization does not avoid the overhead of allocating/deallocating stack frames.

[4] Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Notices* 33, 4 (April 1998), 17–20. https://doi.org/10.1145/278283.278285

[5] Andrew W. Appel and Lal George. 2001. Optimal Spilling for CISC Machines with Few Registers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)* (Snowbird, UT, USA). Association for Computing Machinery, New York, NY, USA, 243–253. https://doi.org/10.1145/378795.378854

[6] A. W. Appel and T. Jim. 1989. Continuation-passing, Closure-passing Style. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89)* (Austin, TX, USA). Association for Computing Machinery, New York, NY, USA, 293–302. https://doi.org/10.1145/75277.75303

[7] Andrew W. Appel and David B. MacQueen. 1987. A Standard ML Compiler. In *Functional Programming Languages and Computer Architecture (FPCA '87)* (Portland, OR, USA) *(Lecture Notes in Computer Science, Vol. 274)*. Springer-Verlag, New York, NY, USA, 301–324. https://doi.org/10.1007/3-540-18317-5_17

[8] Andrew W. Appel and Zhong Shao. 1992. Callee-save Registers in Continuation-Passing Style. *Lisp and Symbolic Computation* 5 (Sept. 1992), 191–221. https://doi.org/10.1007/BF01807505

[9] Matthias Blume. 2001. No-Longer-Foreign: Teaching an ML compiler to speak C "natively.". In *First workshop on multi-language infrastructure and interoperability (BABEL '01)* (Firenze, Italy) *(Electronic Notes in Theoretical Computer Science, Vol. 59)*. Elsevier Science Publishers, New York, NY, USA, 16 pages. Issue 1. https://doi.org/10.1016/S1571-0661(05)80452-9

[10] Matthias Blume and Andrew W. Appel. 1999. Hierarchical Modularity. *ACM Transactions on Programming Languages and Systems* 21, 4 (July 1999), 813–847. https://doi.org/10.1145/325478.325518

[11] Manuel M.T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. 2004. A Functional Perspective on SSA Optimisation Algorithms. *Electronic Notes in Theoretical Computer Science* 82, 2 (2004), 347 – 361. https://doi.org/10.1016/S1571-0661(05)82596-4 Proceedings of Compiler Optimization Meets Compiler Verification (COCV '03).

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

[13] Kavon Farvardin and John Reppy. 2018. Compiling with Continuations and LLVM. In Proceedings *2016 ML Family Workshop / OCaml Users and Developers workshops* (Nara, Japan) *(Electronic Proceedings in Theoretical Computer Science, Vol. 285)*, Kenichi Asai and Mark Shinwell (Eds.). Open Publishing Association, Waterloo, NSW, Australia, 131–142. https://doi.org/10.4204/EPTCS.285.5

[14] Kavon Farvardin and John Reppy. 2020. From Folklore to Fact: Comparing Implementations of Stacks and Continuations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)* (London, England, UK). Association for Computing Machinery, New York, NY, USA, 75–90. https://doi.org/10.1145/3385412.3385994

[15] Kathleen Fisher, Riccardo Pucella, and John Reppy. 2001. A framework for interoperability. In *Proceedings of the First International Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01) (Electronic Notes in Theoretical Computer Science, Vol. 59)*, Nick Benton and Andrew Kennedy (Eds.). Elsevier Science Publishers, New York, NY, 17 pages. Issue 1. https://doi.org/10.1016/S1571-0661(05)80450-5

[16] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. 2007. Manticore: A Heterogeneous Parallel Language. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming (DAMP '07)* (Nice, France). Association for Computing Machinery, New York, NY, USA, 37–44. https://doi.org/10.1145/1248648.1248656

[17] Fermín Javier Reig Galilea. 2002. *Compiler Architecture using a Portable Intermediate Language*. Ph.D. Dissertation. University of Glasgow, Glasgow, Scotland, UK.

[18] Emden R. Gansner and John H. Reppy (Eds.). 2004. *The Standard ML Basis Library*. Cambridge University Press, Cambridge, England, UK.

[19] Lal George. 1999. SML/NJ: Garbage Collection API. (May 1999). https://smlnj.org/compiler-notes/gc-api.ps

[20] Lal George and Andrew W. Appel. 1996. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems* 18, 3 (May 1996), 300–324. https://doi.org/10.1145/229542.229546

[21] Lal George, Florent Guillame, and John H. Reppy. 1994. A Portable and Optimizing Back End for the SML/NJ Compiler. In *Proceedings of the 5th International Conference on Compiler Construction (CC '94)*. Springer-Verlag, New York, NY, USA, 83–97. https://doi.org/10.1007/3-540-57877-3_6

[22] Johan Janssen and Henk Corporaal. 1997. Making Graphs Reducible with Controlled Node Splitting. *ACM Transactions on Programming Languages and Systems* 19, 6 (Nov. 1997), 1031–1052. https://doi.org/10.1145/267959.269971

[23] Richard A. Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations (IR '95)* (San Francisco, California, USA). Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/202529.202532

[24] David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. 1986. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the 1986 Symposium on Compiler Construction (SIGPLAN '86)*. Association for Computing Machinery, New York, NY, USA, 219–233. https://doi.org/10.1145/12276.13333

[25] David A. Kranz. 1988. *ORBIT: An Optimizing Compiler for Scheme*. Ph.D. Dissertation. Computer Science Department, Yale University, New Haven, Connecticut. Research Report 632.

[26] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)* (Palo Alto, California). IEEE Computer Society, Washington, D.C., USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[27] Chris Arthur Lattner. 2002. *LLVM: An infrastructure for multi-stage optimization*. Master's thesis. University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA.

[28] Brian Andrew Leibig. 2013. *An LLVM Back-end for MLton*. Master's thesis. Rochester Institute of Technology, Rochester, NY, USA. https://www.cs.rit.edu/~mtf/student-resources/20124_leibig_msproject.pdf

[29] Allen Leung and Lal George. 1999. Static Single Assignment Form for Machine Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)* (Atlanta, GA, USA). Association for Computing Machinery, New York, NY, USA, 204–214. https://doi.org/10.1145/301618.301667

[30] J. Gregory Morrisett and Andrew Tolmach. 1993. Procs and Locks: A Portable Multiprocessing Platform for Standard ML of New Jersey. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '93)* (San Diego, California, USA). Association for Computing Machinery, New York, NY, USA, 198–207. https://doi.org/10.1145/155332.155353

[31] Steven S. Muchnick. 1998. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[32] John Reppy. 2020. *ASDL 3.0 Reference Manual*. Included in the Standard ML of New Jersey distribution.

[33] John Reppy. 2021. *Notes on using LLVM for code generation in SML/NJ*. https://smlnj-gforge.cs.uchicago.edu/scm/viewvc.php/dev-notes/?root=smlnj

[34] Konstantinos Sagonas, Chris Stavrakakis, and Yiannis Tsiouris. 2012. ErLLVM: An LLVM Backend for Erlang. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang (ERLANG '12)* (Copenhagen, Denmark). Association for Computing Machinery, New York, NY, USA, 21–32. https://doi.org/10.1145/2364489.2364494

[35] Zhong Shao and Andrew W. Appel. 1994. Space-efficient Closure Representations. *SIGPLAN Lisp Pointers* VII, 3 (July 1994), 150–161. https://doi.org/10.1145/182590.156783

[36] Zhong Shao and Andrew W. Appel. 2000. Efficient and safe-for-space closure conversion. *ACM Transactions on Programming Languages and Systems* 22, 1 (2000), 129–161.

[37] Guy L. Steele Jr. 1977. *LAMBDA: The Ultimate GOTO*. Technical Report AI Memo 443. Massachusetts Institute of Technology, Cambridge, MA, USA.

[38] David A. Terei. 2009. Low Level Virtual Machine for Glasgow Haskell Compiler. , 73 pages. https://llvm.org/pubs/2009-10-TereiThesis.pdf Undergraduate Thesis.

[39] David A. Terei and Manuel M.T. Chakravarty. 2010. An LLVM Backend for GHC. In *Proceedings of the 2010 ACM SIGPLAN Symposium on Haskell (HASKELL '10)* (Baltimore, MD). Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/1863523.1863538

[40] Katsuhiro Ueno and Atsushi Ohori. 2014. Compiling SML# with LLVM: a Challenge of Implementing ML on a Common Compiler Infrastructure. In *Workshop on ML*. 1–2. https://sites.google.com/site/mlworkshoppe/smlsharp_llvm.pdf

[41] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. 1997. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL '97)* (Santa Barbara, California). USENIX Association, Berkeley, CA, USA, 15 pages. https://www.usenix.org/legacy/publications/library/proceedings/dsl97/wang.html