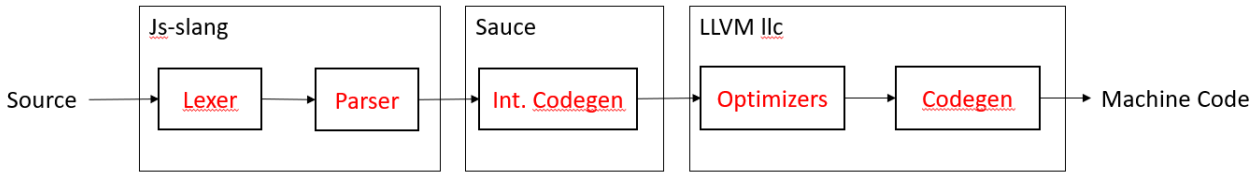
LLVM Sauce Compiler Specification

Jiachen, Xiaodong

National University of Singapore
School of Computing

April 23, 2021

1 Overview



LLVM Sauce is a Source 1 compiler that respects the Proper Tail Call (PTC) semantics of JavaScript built on top of the LLVM compiler framework. More formally, LLVM Sauce is a Source 1 front end that takes in a source program and outputs LLVM Intermediate Representation (IR). This works by relying on the JS-Slang parser to produce a valid Abstract Syntax Tree (AST). It then performs intermediate code generation on this AST to produce LLVM Intermediate Representation (IR). This LLVM IR can then be compiled further or interpreted with tools such as `llc` and `lli`.

The complexities involved in LLVM Sauce mostly reside in intermediate code generation. By leveraging the JS-Slang parser, we can ignore whole classes of errors including syntactical and distant name errors which will be caught by the parser. In addition, by leveraging the LLVM framework, we get to use all the existing optimizer passes that run on our IR as well as all the back ends available!

Through comprehensive testing with custom test cases as well as SICP chapter 1 code examples we are certain that LLVM Sauce is fully Source 1 language compliant. Deviations from Source 1 will be specified in more detail in later sections. Major deviations include number literals types, lack of library functions and lack of garbage collection.

In summary, the major contribution of this project is a working Source 1 compiler that allows users (for the first time) to compile and run Source nativity on a whole host of machine architectures supported by LLVM including x86, ARM, MIPS and even WASM.

2 Runtime Environment

In this section, we detail the run time environment that our generated code will interact with at run time including the memory model of our program. LLVM Sauce supports any 64bit Linux distribution since we rely on libc++ system calls for string and memory operations.

2.1 Library Functions

We only include one library function. This library function is generated as LLVM IR and appended to the generated IR.

- `display(x)`: prints the value of x to standard output.

The other library functions for things like lists and math are left as a future extension.

2.2 Garbage Collection

LLVM Sauce currently has no garbage collection and this is left as an important future extension as well.

2.3 Memory Model

To conform with the dynamic semantics of Source, storing variables on the stack will not suffice. As such, everything we create will be allocated and stored on the heap via `malloc`. The stack will be left for maintaining the call stack similar to the runtime stack in the SVML VM. More details on the layout of these structures will be provided in the next section.

3 Intermediate Code Generation

3.1 Source A — Expression language

Source A is an expression based language forming the backbone of the Source language. It is defined using the Backus-Naur form as follows.

program	:=	*statement*	program
statement	:=	**`const`** *name* **`=`** *expression* **`;`**	constant declaration
	|	*block*	block statement
	|	*expression* **`;`**	expression statement
block	:=	**`{`***expression...***`}`**	block statement
expression	:=	*number*	primitive number expression
	|	**`true`** | **`false`**	primitive boolean expression
	|	*string*	primitive string expression
	|	*string*	primitive string expression
	|	*name*	name expression
	|	*expression binop expression*	binary operator combination
	|	*expression unop*	unary operator combination
binop	:=	**`+`** | **`-`** | **`*`** | **`/`** | **`%`** | **`===`** | **`!==`**	
	|	**`>`** | **`<`** | **`>=`** | **`<=`** | **`&&`** | **`||`**	binary operator
unop	:=	**`!`** | **`-`**	unary operator

3.1.1 Compiling programs

A Source program is made up of a single *program* expression (see syntax above). In LLVM, there is a C-style requirement of requiring a main function as the entry point. Hence every *program* has at least one function, which has *program* as its body.

3.1.2 Compiling literals

Source A introduces 3 types of literals: numbers, booleans and strings. In line with the dynamic semantics of Source, type signatures need to be stored to allow runtime type checking. Type signatures are in turn just a unique double value. We subsequently detail the exact structure of each literal object during runtime.

Number All numbers are stored and represented as doubles (64-bit floating-point value). For example, the number literal `1` will be allocated as:

Type	1.0
Value	1.0

Boolean Boolean literals are also represented as doubles which take on the value 1.0 for 'true' or the value 0.0 for 'false'. For example, the boolean literal **`false`** will be allocated as:

Type	2.0
Value	0.0

String String literals are stored and represented as an immutable global null terminated character byte array. String manipulation is done using `libc++` functions such as `strlen`, `strcat` and `strcpy`. For example, the string literal `"abc"` can be allocated as:

Type	3.0
Value	`*char: 0x1234`

3.1.3 Compiling blocks

When compiling a block, we allocate and create a new environment frame. Similar to Source, we scan out the block to find all declarations we have in the block in order to allocate enough memory. The compiler also keeps a mapping of variable names to offsets within the environment frame in order to facilitate efficient look up of names and identifiers. There is also a pointer that points to the parent frame in which the block is declared in.

For example the block `{const x = 1; const y = 2; const z = 3; display(x + y + z);}` would result in the following environment frame.

parent frame pointer
literal pointer
literal pointer
literal pointer

3.1.4 Compiling names

In Source A, names and identifiers are invisible to the runtime. The LLVM intermediate representation refers to variables only as pointers and memory addresses, and there does not exist a concept of names at all. Through lexical scoping rules we are able to determine which frame we need to go to and which offset this variable resides in within the frame. This is represented as a pair `(jumps, offset)` which specify the number of enclosing environments (jumps) and the specific memory chunk (offset) in the variable's resident environment. For example, consider the following Source program:

```
{
    const x = 2; // offset 1
    const y = 1; // offset 2
    {
        {
            y;
        }
    }
}
```

The lookup of name **`y`** in the innermost frame will result in 2 parent jumps and an offset of 2. In section 5, we will examine some optimizations to this lookup process.

3.1.5 Compiling operators

There are LLVM instructions for every operator we need in Source. Therefore we can compile Source operators in the usual way. We first recursively compile the operand(s) and then supply them as arguments to the corresponding LLVM instructions. We also need to check the operand types. This is done during runtime by inspection of the "type" section of each literal. With the knowledge of operand types we can also define "overloaded" operators like the addition operator **+**, which performs ordinary real value addition on doubles and switches to `strcat` on strings.

3.2 Source B — Runtime type checking

During type checking for operators we see a need for error handling. Source B expands on Source A by adding errors. However we do not have actual errors or exceptions that users can use. What we do implement is a way for programs to quit on the spot and return an error code. This occurs for instance if we try to add a number to a string. To achieve this we have predefined functions like `errorWithValue` that the runtime is able to call.

On a runtime type error, we print an error to stdout and then invoke the 'exit' syscall with return value of 1 indicating a failure halting the entire program.

3.3 Source C — Control Flow

Source C adds control flow constructs to the language. Prior to this booleans were not very useful. The following lists additional rules added upon those in Source A. The slight difference between Source C and Source §1 is that there is no mandatory **`else`** after the **`if`**. This might have been a small oversight.

if-statement	:=	**`if`** **`(`***expression***`)`** *block*	
	|	**`if`** **`(`***expression***`)`** *block*	
		`else` (*block* | *if-statement*)	conditional statement

The evaluation rule for ternary statements is summed up as follows:

$$\frac{\begin{array}{l}\texttt{E1}\hookrightarrow\texttt{s1, v1}\\ \texttt{E2}\hookrightarrow\texttt{s2, v2}\\ \texttt{E3}\hookrightarrow\texttt{s3, v3}\end{array}}{\texttt{E1 ? E2 : E3} \hookrightarrow \begin{array}{l}\texttt{s1.condbr v1, Con, Alt.}\\ \texttt{Con:. s2. br End.}\\ \texttt{Alt:. s3. br End.}\\ \texttt{end: phi v2, v3}\end{array}}\ [\texttt{? :}]$$

It is easy to generalise this to other conditionals.

3.4 Source D — Functions

Source D introduces functions into our compiler. This is the arguably the most essential language feature in Source. This is due to some properties of function in Source listed below that make it hard to translate to a procedural styled language like LLVM intermediate representation.

- Frames need to live beyond the lifetime of the accompanying function.
- Function callees cannot be statically resolved at compile time.
- Functions should be first class objects (they should just be another literal type).

Source D introduces the following grammar rules to the language:

expression	:=	...
	|	(*name* | (parameters)) **=>** *expression*
	|	(*name* | (*parameters*)) **=>** *block*
	|	*expression* (*expressions*)
statement	:=	...
	|	**`function`** *name* (*parameters*) *block*
parameters	:=	ϵ | *name* (, *name*) ...
expressions	:=	ϵ | *expression* (, *expression*) ...

3.4.1 Function literals

At the center of everything lies the function literal. If we can compile a function down to a literal object correctly, function declaration and lambda expressions become trivial. In Source D we introduce an additional literal, the function literal.

The layout of the function literal looks like:

Type	4.0
Env	environment pointer
Function	function pointer

Note this is similar to a closure in SVML compiler. However, formals are not needed at runtime, leaving us with the enclosing environment and the start address (which is also the function pointer).

Parent environment As per Source semantics, this enclosing frame points to the environment in which the function is declared in.

Function pointer Because functions cannot be resolved to functions names statically, we have to rely on function pointers to execute functions instead of their function names.

3.4.2 Function Signature

As LLVM IR is particular and strict about types, all our functions have to be of the same function type (list of arguments and return type form the function signature / type) so we can have a consistent type for our function literal. To get around this restriction we standardize our LLVM intermediate representation functions to take in two arguments and return a literal pointer: firstly the enclosing environment frame, and secondly an array of literal arguments. A function would always return a literal, and in the case where nothing is returned `undefined` is returned instead. An example function compiled to LLVM would look something like

```
define %literal* @__f(%literal** %env, %literal** %args) {
    ...
}
```

3.4.3 Compiling a Function Declaration

Compiling a function declaration consists of compiling the function expression into a function literal and then optionally assigning it to its name if it has one. Every function declaration in Source will create a new function in LLVM IR with its name followed by arbitrary numbers to prevent name space conflicts since LLVM names have to be globally unique. Since we never refer to or call functions by their names, we do not have to maintain a mapping of them.

Compiling a function expression takes place in three major steps.

Prologue In the function prologue, we create a new environment frame with enough space for the function arguments. We then copy the function arguments over from the argument array (which is also the second argument). We point the parent of this frame to the enclosing environment as the first argument.

Body Due to Source semantics the body is guaranteed to be a *block* or an *expression*. We can therefore recursively compile them using the rules we have already defined previously.

Epilogue In the epilogue, we check if the basic block have been terminated. This would imply all paths in the control flow graph have returned some value. Otherwise, we return a value of `undefined`. This marks the end of the compilation of the function expression.

3.5 Comping a function call

To compile a function call, we follow the following compilation rule. Note that a function always takes in two arguments are mentioned earlier.

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{callee} \hookrightarrow s_0, f \\ \Gamma \vdash \texttt{par} \hookrightarrow s_1, ..., s_n, v_1, ..., v_n\end{array}}{\texttt{callee(par)} \hookrightarrow s_0, s_1, ..., s_n\texttt{.call f (env, [}v_1, ..., v_n\texttt{])}}$$

4 Proper Tail Calls (PTC)

One key goal of our project was to build a compiler that respects the Proper Tail Call semantics of the ECMAScript 2015 Specification. To do this, we need to first correctly detect and identify tail recursive calls and secondly replace them with custom instructions. By doing this, we are able to avoid allocating a new stack and environment frame and just execute the new function in the current context with new arguments.

4.1 Tailcall Detection

When compiling a new function declaration, one of the first things we do is scan the function looking for tail recursive calls.

Definition 4.1 (Tail Recursive Calls)**.** A call expression is considered to be *tail call recursive* if and only if it is in tail position and it is a call to the enclosing function. □

Definition 4.2 (Tail Recursive functions)**.** A function is considered to be *tail recursive* if and only if it contains one or more tail recursive calls within the function body. □

Example 4.1.

```
// f is considered tail recursive
const f = () => true ? true : f();

// f is tail call recursive
function f(x) {
    return x ? f(!x) : false;
}

// fact is not tail call recursive
function fact(n) {
    return n === 1 ? n : n * fact(n - 1);
}
```

```
// fact is tail call recursive
function fact(n, acc) {
    return n === 1 ? acc : fact(n - 1, n * acc);
}
```

◊

We detect these tail recursive calls by exploring a function looking for tail calls in the expression within return statements. Such an expression contains a tail recursive call if the expression is

1. a call expression to the enclosing function by its name, or
2. a ternary operator where at least one of the consequent or alternative expressions contain a tail recursive call

4.1.1 Marking

After detecting these tail recursive calls, we prepend a delimiter of '#' to signify that this function call is a tail recursive call. The symbol '#' is not allowed in Source function names so this is safe.

4.2 Compiling a tail call

When compiling a function call, we check if the function name starts with '#'. If it does, instead of generating normal call instructions, we first evaluate the arguments, then we calculate the position in the function frame where they live, and replace them in-place. We then jump to the basic block at the start of the function in the basic block 'f.entry'. This allows us to execute the new function in the current frame without allocating more space.

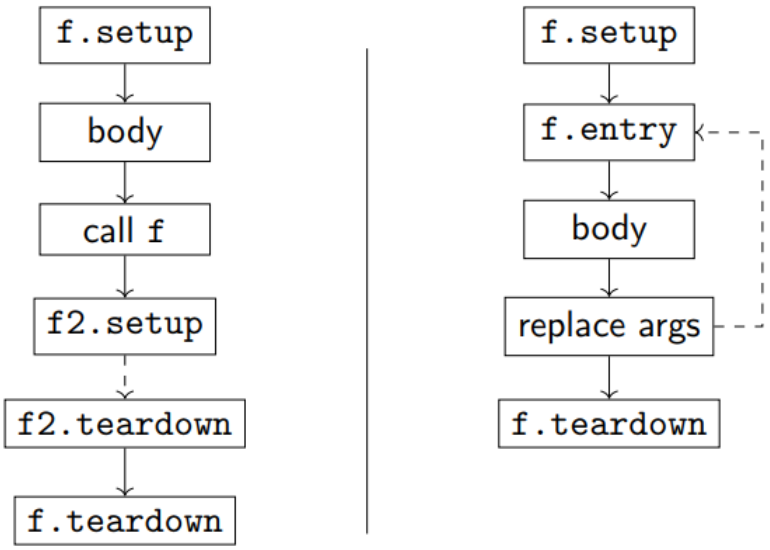


Figure 1: Tail Call Basic Blocks

5 Load Store Optimizations

Before any optimizations, lookup for identifiers would start from the current innermost environment and follow pointers to the target frame. Although we know exactly where these frames are, we still have to follow a series of pointers to them. This can be a huge bottleneck and load store optimizations are a solution to this problem. It both reduces generated code size and also the efficiency of our code. Note that this optimization will only work with Source 1 since variables are immutable (no **`let`** statement).

5.1 Optimization 1: Efficient Lookup

When looking up an identifier or name, we can consider two cases. Firstly, where the variable is within the current function, and secondly, where the variable occurs free outside of the function.

Case 1: If a variable occurs within the scope of the function (not necessarily the same environment frame), we already have access to some temporary variable that stores the pointer to this literal. In this case, we can guarantee that the basic block will precede the current basic block.

Case 2: If the variable falls outside of the function we have no choice but to follow the parent pointers in the environment frames. However, we do not have to start with the innermost frame but can start with the function frame and work our way backwards.

5.2 Optimization 2: Caching with Virtuals

The second optimization we made was to cache these variables presented in case 2 of the previous optimization where we resolved variable names outside of the function. After resolving for the first time, we can add this mapping as a virtual to our compile time environment as a virtual.

In the future, when the variable is looked up again, as we recursively lookup the name in the environment we also look out for virtual. If we come across a virtual before we find the actual variable we used the virtual instead which would contain some temporary register in our LLVM function that will point to the literal at run time,

For example,

```
const x = 1;

function f() {
    x; // load variable x from the enclosing frame
    return x; // we now have a temp register containing the literal, use that instead!
}

f();
```

Will the produce the following 3AC pseudo code:

```
function f(env, args) {
    // lookup the first x;
    t1 = env.parent;
    t2 = t1 + 1 * sizeof(pointer);
    t3 = load t2; // load literal x from address t2

    // HIT! lets use t3 instead
    ret t3;

}
```

Through these two optimizations we noticed a 20% reduction in the size of the generated LLVM IR code.