Backbone Compiler

Sabrina Jiang, Will Sherwood, and Kasra Sadeghi*

E-mail: sabrinaj@utexas.edu, kasra@cs.utexas.edu, wisherwood@utexas.edu

Abstract

The goal of this project is two-fold: firstly, design a programming language that has enough features to run system calls and do basic programming tasks. Secondly, the programming language should be designed with the intent of it being able to compile itself. We completed the first task by creating Backbone, a lisp-like language that has recursive functions, let bindings, arithmetic and more basic features. To complete the second task, we wrote a compiler for Backbone in C to translate it directly into LLVM bytecode. This was created in a way to make it easy to translate into Backbone later, once the language had been developed more. This would result in a self-bootstrapping compiler in Backbone.

Introduction

Our language, Backbone, was designed to be able to eventually compile itself. We worked towards this goal by first writing a C compiler in a way that allowed us to easily translate into Backbone later. This meant that at times, we couldn't use convenient C shortcuts to write the compiler, since it was likely that Backbone wouldn't be able to support such complex features.

Despite failing to complete the Backbone compiler written in Backbone for this project, we were able to flesh out a parser, LLVM generator, and more in the amount of time we had.

Additionally, having these written in C was an advantage because we wanted Backbone to be inspired in part by C.

We chose to compile Backbone into LLVM, a widely used low-level language that was created to be flexible and reliable. LLVM started as a research project and has now become a library of modular technologies intended for use in a compiler, perfect for our needs in this project. Also, we use Clang and CMake to compile our source code, both of which are respectable C compilers. Clang in particular was convenient because it is an LLVM - native compiler.

The rest of this paper will spend its time detailing the design decisions made for Backbone. It will cover its features, syntax, and compiler semantics. There are code examples sprinkled throughout to help show how Backbone looks and feels.

Language

The motivation behind the design of the Backbone language was to be simple enough to write a simple compiler for it, but also have enough features to be able to write a backbone compiler in the backbone language. Through many design choices, we came to the language syntax detailed below.

After some trial and error, we concluded that the best way to make syntax decisions was to first write the target feature in LLVM, examine the syntax, and then reverse-engineer the corresponding Backbone syntax. For example, while creating function definitions, the original syntax was as follows:

```
(def main (paramlist (param argc i32) (param argv i8**))
```

However, after examining the LLVM and discussing which keywords were necessary, we decided on the following format instead:

```
(def main (params (argc i32) (argv i8**)) i32
```

This new format was much better. Firstly, it provides everything needed to generate the LLVM. We realized the return type was required knowledge, and added that to the definition. Secondly, it eliminates some of the unnecessary keywords.

The rest of our syntax was decided in a similar manner. The biggest consideration was balancing between inferring information, such as types of identifiers and arguments, and requiring the programmer to explicitly state the types. In the end, we generally decided that asking the programmer for more information was a better choice, as it made our jobs a bit easier. This helped eliminate the need for multiple passes through the program.

Flattening Compiler Pass

Overview

Flatten is unique in that it is the only transformation from a symbolic expression to another symbolic expression. It is unlike the other phases, Parser and Gen, in that it is an operator. This makes it a pure transformation of trees, and it is how we implement compiler passes.

The structure of our language, compared to the LLVM Intermediate Representation language (LLVM IR), gives more flexibility to the expression-value relationship. In LLVM IR, there is one expression evaluation per instruction, as instruction evaluation is thought of as an opcode. In our language, we generalize the strictness of the LLVM IR in order to allow for a more natural expression of ideas and a way to elide the needless use of extra variable names and register management.

In our language, we separate Expressions that are immediate Values or Expressions that can have Expressions inside of them, which we call Tall Expressions (also Unflat Expressions).

```
Expression = Value | Tall
```

In LLVM IR, there are first-class types. These correspond to our language's Value Expressions. It often requires that the arguments to an opcode be first-class types. For example, every argument in the argument list in a function call instruction must be a first-class type. Our language defines call's arguments as expressions, and then it flattens each argument (see Flattening a Tall Expression) if it is not an immediate value.

Flattening the Program

Flattening a program basically boils down to flattening each of it's blocks, which correspond to LLVM control blocks. Each of Backbone's blocks must end in a return, while in LLVM there are other kinds of terminal instructions.

Flattening a Tall Expression

Flattening a Tall Expression extracts the expression and places it in a let. To create a value for the let, we have a global counter for the whole flatten call. Local variable names only have to be unique in each function definition, so we reset the counter each time we flatten a function definition. Once we extract the let, we insert the let before the statement we extracted it from with the name received from the local counter (e.g. $5).

To keep invariants consistent, the size of the encompassing block must be increased by one, and with the introduction of if-statements we have recursive search through blocks, so we have to keep track of the current block and cache it every time we recurse inwards (fBlock).

Once we've inserted the let statement, we also need to recurse into it to make sure that the expression we just flattened does not need further flattening (see Flattening Let Statements).

Flattening Let Statements

Flattening let statements is where we do the bulk of the initial investigation into flattening expressions. We check to see if the child of each let is one of the expressions that can have expression children, i.e. they are Tall. This is also where we do Call-as-a-Statement.

Call-as-a-Statement

Flatten also has a Call-as-a-Statement call as it traverses through statements as a convenience. Sometimes we call a function and we want to ignore the result. So we just place the call into a let, replace it, and use the local counter that flatten has available to it to ease the work.

Parser

Overview

The parser of most languages is fairly tedious to write and a great deal of research has gone into writing efficient parsers. There's a great number of parsing algorithms to choose from, but most big projects roll their own parser or use a parser-lexer framework like flex/bison (or lex/yacc). We don't have a back-end for any of the frameworks because our language doesn't exist yet, so it's a better idea to write a very simple parser for a language with very simple structure.

It would be a tremendous time investment to write a parser for our language in C and then write it again in our language if we were to choose a complex structure for the parser. There are two language structures that are fairly simple to parse. Concatenative languages like Forth or Factor have a very simple structure. Each token is separated by a space, and the evaluation of the language is token by token, in-order (most of the time). The other choice is a LISP-like language. We chose to use a LISP-like language structure because it is more natural to express ideas in that are already structured like a tree, and our bootstrapping language is C which has a tree-based structure. It also doesn't allow for ambiguous parses, because there is no order of operations in a LISP (there are more than enough parentheses to clarify).

Overlying Decisions

If you look in our examples/ directory, you can see the code samples we thought about to make decisions about the structure of the language. We grounded on the following rules:

1. our generator is not going to be very interesting
2. we're not going to do any optimization in the first phase.
3. the syntax for the language must align closely with the information available in LLVM IR
4. there should usually be a one to one correspondence, or a very simple lookup in the tree.
5. names are more important than types, so types should follow the names.

Divergence from Lisp

Our language looks like a LISP, but only for the easy parsing. We don't have any Lisp features like homoiconicity, hygienic macros, garbage collection, etc. We also don't obey the traditional LISP definition of a list. In LISP:

```
S-expression = List | Atom
List = ( Sexp* )
```

However, in Backbone, we have:

```
S-expression = List | Atom
List = ( Word Sexp* )
```

```
Atom = Word | String | Char
```

Future Plans

Verify the characters allowed in different expressions and allow for better error reporting for incorrect syntax. Validate the resulting tree and make sure it is a some minimal amount of a valid Backbone program instead of crashing or generating invalid LLVM. These were not prioritized because making the language Turing complete and capable of bootstrapping itself were prioritized during the first phase.

Backbone

Language terminals

A few language terminals will be described here and used in the rest of the language to construct most of the other features.

$$id \rightarrow [a - zA - Z\_][a - zA - Z0 - 9\_] \quad (identifiers) \tag{1}$$

$$type \rightarrow (i8|i16|i32|i64|u8|u16|u32|u64) *^{*} \quad (types) \tag{2}$$

$$\tag{3}$$

Expressions and Statements

There are two kinds of code execution in Backbone: Statements and expressions. Statements are a standalone piece of execution that executes only for its side effects. Specifically, input, output, and let bindings are the most useful applications of statements in Backbone. Expressions, on the other hand, are evaluated for their use in statements. Expressions simply reduce to values at runtime, and live to give meaning to the side effects produced by statements.

Specifically, an "Expression", which will be used in the grammar definitions of many structures throughout backbone, can be any of the following: A `call`, `call-vargs`, an arithmetic operator such as $+$ or $-$, a comparison, a `load`, or itself just a value.

Statements, on the other hand, have a variety of meanings: Statements include let bindings, return statements, conditional evaluations, calls and call-vargs, automatically deleted declarations, and storing a value into a variable. Notice that both variadic and regular calls are statements and expressions. The reason for this is that sometimes the value is not needed for a function, and it was a nice language feature to be able to simply call a function without having to use a let binding with an ignored variable name.

Auto: the way it was meant to be

The C Programming Language has a feature that goes fairly unused, and in fact most C programmers have probably never heard of it. The `auto` keyword in C denotes automatic allocation of a variable on the stackframe of the current function call and automatically freeing it on return.

Backbone does not have a way to express modification of a name that has been `let` to be some result because that's misleading. You said `let a be 5`. `a` should always be `5`. Because of this, you cannot get the address of a value that has been let and you cannot store a value to be some primitive directly. `let` statements must be the result of some computation in order for them to actually be used. In mathematics, there is also a `let` statement for aliasing a value, but that has not (yet) been implemented as we prioritized `auto`.

`Auto` represents stack allocation of a variable of a certain size, and it sets the name assigned to be the pointer to the current stack allocation. This allows passing by reference more naturally, gives semantics to getting the address of a stack variable, and distinguishes between values you can reassign and values you can't.

The only way to mutate a value is to call some heap allocator (`malloc` family) and then store different values into it or get a pointer using `auto`.

Structs

Users can define their own struct data types as a statement. The general grammar is defined as follows:

$$Struct \rightarrow (\texttt{struct}\, id\, Fields) \tag{4}$$

$$Fields \rightarrow Field\, Fields \mid Field \tag{5}$$

$$Field \rightarrow (id\, type) \tag{6}$$

An example of this syntax would be the following struct definition:

```
(struct SymbolicExpression
 (value i8*)
 (list SymbolicExpression**)
 (len u64)
 (cap u64))
```

Structs can be indexed using the `(index`*idtype*`Expression)` expression. Here, the id is the identifier for the struct, type is the definition of the struct, and Expression is an expression that evaluates to a `i32` value. This expression returns the value that is stored inside the struct given the specific index. In the future, we would like to support named variables in structs which would allow for more code readability, but would not add to functionality.

Since a pointer to an element in a struct can be retrieved using *index*, values can easily be modified by using the `store` keyword. Likewise, an element inside of a struct can be retrieved using the `load` keyword after indexing. The following is an example of the use of structs, followed by the corresponding LLVM genereated by the compiler:

```
(struct Basic
 (a i32))


(str-table
 (0 "Basic{a = %d}0A00"))


(decl calloc (types i64 i64) i8*)


(def makeBasic (params (a i32)) Basic*
 (let r (cast i8* Basic* (call calloc (types i64 i64) i8* (args 1 8))))
 (store a i32 (index r Basic 0))
 (return r Basic*))


(decl printf (types i8* ...)  i32)


(def main (params (argc i32) (argv i8**)) i32
 (let t (call makeBasic (types i32) Basic* (args 7)))
 (call-vargs printf (types i8* i32) i32 (args
 (str-get 0)
 (load i32 (index t Basic 0))))
 (return 0 i32))
```

And the corresponding LLVM bytecode:

```
%struct.Basic = type { i32 }
@str.0 = private unnamed_addr constant [15 x i8] c"Basic{a = %d}0A00", align 1


declare i8* @calloc(i64, i64)
declare i32 @printf(i8*, ...)
define %struct.Basic* @makeBasic(i32 %a) {
entry:
 %$0 = call i8* (i64, i64) @calloc(i64 1, i64 8)
 %r = bitcast i8* %$0 to %struct.Basic*
 %$1 = getelementptr inbounds %struct.Basic, %struct.Basic* %r, i32 0, i32 0
 store i32 %a, i32* %$1
 ret %struct.Basic* %r
}

define i32 @main(i32 %argc, i8** %argv) {
entry:
 %t = call %struct.Basic* (i32) @makeBasic(i32 7)
 %$2 = getelementptr inbounds %struct.Basic, %struct.Basic* %t, i32 0, i32 0
 %$1 = load i32, i32* %$2
 %$0 = call i32 (i8*, ...)  @printf(i8* getelementptr inbounds ([15 x i8], [15 x i8]* @s
 ret i32 0
}
```

Which has the expected output of `Basic{a = 7}`.

Calls

Backbone supports statically typed functions that can be recursive and nested. Splitting code into subroutines or functions is incredibly important in software engineering, and are vital to effectively implementing system calls.

The general grammar is as follows:

$$\texttt{call}\ \mathit{id}\ (\texttt{types}\,\mathit{type}^{*})\ (\texttt{args}\ \mathit{Expression}^{*})\ \mathit{type}$$

Call is one of the more interesting features since it is both a statement and an expression. The implications of this is that we can have nested calls. Since LLVM requires one expression evaluation per instruction, calls with nested expressions need to be evaluated one at a time. We do this by making a pass over the syntax tree and flattening expressions. This is further discussed in our section on flattening.

An example of this syntax would be the following call statement:

```
(call printSexp void ((program Sexp*) (0 i32)))
```

Qualified Types

Converting between backbone and LLVM types is fairly simple. If the type is a user defined struct, the LLVM type replaced is simply %*struct*. concatenated with the user defined struct type. If it is a primitive type, or a pointer to a primitive type, they have the same representation. Here are two examples of qualified types:

$$\texttt{i32***} \quad \rightarrow \quad \texttt{i32***} \tag{7}$$

$$\texttt{SymbolicExpression**} \quad \rightarrow \quad \texttt{\%struct.SymbolicExpression**} \tag{8}$$

Let

Let statements are used to assign an identifier to a value, and are very convenient for programmers because they allow them to give names to all the values they use.

The value that you are binding to a let must be the result of some computation. You cannot let to an immediate register value (another name that has been let or a literal.

The general structure of a let binding uses a `let` keyword, the identifier, then the value to set it to:

`let` *id* `expression`

This example sets the id `c` to the value computed by `(+ i32 2 3)`, 5.

```
(let c (+ i32 2 3))
```

Conditionals

One of the most basic functionalities of programming languages is the ability of the program to branch, typically completed with some sort of condition-checking. Backbone, like many other languages, uses the `if` keyword to indicate this. After the `if` keyword, the programmer should have a predicate, and then an expression that should be evaluated if the predicate evaluates to a non-zero number.

The general grammar is as follows:

```
if (predicate)
 (statement⁺)
```

The following syntax example has a predicates that compare two numbers, the syntax of which will described shortly:

```
(if (< i32 2 1)
 (call puts (types i8*) i32 (args (str-get 0))))
(return 0 i32)
```

Return

Of course, it is important to be able to return from a function call with useful information, so that the result of the function can be used elsewhere. Our return simply requires the programmer to use the `return` keyword, followed by the value to return and then its return type. This return type should match the return type of the function that was declared. The general grammar is as follows:

`return expression` *type*

An example of this syntax would be the following:

```
(return i i32)
```

Declarations

We support declaration statements similar to C with the following syntax:

```
(decl id (types type*) type)
```

An example of a declaration would be the "puts" system call:

```
(decl puts (types i8*) i32)
```

Function Definitions

As mentioned, Backbone supports function definitions, useful for separating code into reusable chunks and making programming similar tasks much simpler. Despite being a more challenging feature to support, we all agreed that functions were completely necessary for this project.

Firstly, to define the name, return type, and arguments of the function, the programmer should use the `def` keyword, followed by the name of the function. Next, the parameters should be listed after a `params` keyword, each with the id and type of the argument. The return type is then added on after, and finally, the body of the function. This should end in a return statement, such that the return statement's type matches the return type of the function definitions. The general syntax is as follows:

`(def` *id* `(params (` *id* *type* `))` *type* `(statement`$^{+}$`))`

The following syntax creates a function called `func` with two arguments, `a` of type 32-bit int, and `b` also of type 32-bit int. `func` simply returns the value of its argument a:

```
(def func (params (a i32) (b i32)) i32
 (return a i32))
```

Binary Operations

Backbone supports a number of binary operations, including arithmetic and comparisons.

Comparators

As mentioned above, the ability to compare two different values is essential for conditionals and other basic calculations. Our comparator syntax has four parts.

The first part is the desired operation $<$, $\leq$, $>$, $\geq$. Additionally, the type must be specified, such as a 32-bit integer. Finally, the two expressions whose values are to be compared.

The general grammar is as follows:

comparator *type* `expression expression`

Some simple examples are below:

```
(< i32 3 2)

(< i32 x y)
```

Arithmetic

The grammar for performing arithmetic is similar:

operation *type* `expression expression`

A simple example follows:

```
(+ i32 2 3)
```

Samples

This section contains some simple examples that we used for unit testing, and are great for showcasing the full process.

Here is an example Backbone program. It simply calls a function to print ”Hello World”, but we are more interested in the LLVM it generates.

```
(str-table
 (0 "Hello World\00"))
(decl puts (types i8*) i32)
(def voidcall (params) voi
 (call puts (types i8*) i32 (args (str-get 0)))
 (return void))
(def main (params) i32
 (call voidcall (types) void (args))
 (return 0 i32))
```

The corresponding LLVM that would be generated is below. Notice how named functions and constants are directly passed down into the LLVM:

```
@str.0 = private unnamed_addr constant [12 x i8] c"Hello World\00",
align 1
declare i32 @puts(i8*)
define void @voidcall() {
entry:
 %$0 = call i32 (i8*) @puts(i8* getelementptr inbounds ([12 x i8],
 [12 x i8]* str.0, i64 0, i64 0))
 ret void
}
define i32 @main() {
entry:
 call void () @voidcall()
 ret i32 0
}
```

Conclusion

In conclusion, overall the language design was a success. Sadly we were not able to fully construct another version of the compiler written itself in backbone, but the C compiler is fully functional and should be a relatively simple transpose as a future work. Some of the harder problems that were overcome were tree flattening which wrote nested calls and returns as single evaluations, automatic variable deletion from stack, as well as defining useful data types such as structs that are interoperable with C and Linux system calls. In the future, it would be good to have some formal verification of the language (writing and checking semantics in a proof checker), and type inference. All in all, however, the language is usable and works to create full interoperability between calling C and Backbone functions as well as have it compile to a low level intermediate representation language (LLVM).