Intercept Compiler Developer Reference

Rylan Lens Kellogg

February 13, 2023

Contents

Overview

Currently, Intercept has three stages (or phases, or levels, anything to that regard—we'll call them stages).

The parser converts Intercept source code into a data structure that represents the program in it's entirety. The parser relies on the lexer, and the two work in tandem.

The type-checker then validates this data structure, ensuring that the semantics of types are being followed properly.

The code generation stage creates a new data structure (an in-memory intermediate representation) that follows static single assignment form. This intermediate representation gets converted by each backend into architecture specific implementations (machine code, asm, etc).

Lexing

Lexing The process of converting a sequence of characters into a sequence of lexical tokens, or *lexemes*.

The compiler uses lexing to separate the source code into meaningful pieces; after all, it's all just bytes in a file. The lexer is the first stage in syntax analysis, and works in tandem with the parser.

The lexer returns a string view: a pointer to the beginning and end of a sequence of bytes. This returned string is then further analyzed by the parser, until it has been handled, and the parser can once again get a new string view from the lexer.

The lexer works by following these steps:

1. Set beginning of lexeme to SOURCE, the string given, after skipping all initial whitespace.
2. Set end of lexeme to beginning of lexeme.
3. Skip beginning and end of lexeme past any comments.
4. Skip end of lexeme to the first occurence of whitespace after the beginning.
5. If beginning and end of lexeme are equal, return a 1-length lexeme.

NOTE: Lots of error checking steps were left out, for the sake of brevity.

Whitespace `\t\r\n`
Delimiters `\t\r\n,{}()[]<>:&@.`

For example, here is how the following source code would be split into lexemes.

```
numA : integer = 42
numB : integer = 69
```

```
"numA"
":"
"integer"
"="
"42"
"numB"
":"
"integer"
"="
"69"
```

Parsing

The job of the parser is to create an understanding of the program, from the compiler's point of view. To do this, the program must be unambigous, and contain only sequences of lexemes that are understood.

One thing you may be wondering: how does a computer program (the compiler) *know* something? How can you make it *understand*? The idea is that by defining a data structure in our compiler that can store the meaning of programs, as well as any necessary data along with it, the compiler can construct one of these structures and end up with an understood program, because the data structure is already understood.

There are two usual choices for data structures that can fully define the semantics of a program, while also storing data along with it.

One, the most common, is called an *abstract syntax tree*, or AST. The AST is a fancy name for a tree that houses only the bare minimum data of what is needed for the program moving forward. For example, parentheses aren't needed in an AST, because they don't actually do anything. They are only for the parser to understand what order to parse things in; since we don't need them after that (the AST is already constructed in the right order), they don't get included in the AST itself. This is why it is an *abstract* syntax tree; the tree doesn't map one-to-one with the source code syntax. Some compilers do build what's called a parse tree that does map one-to-one with the source code.

Another, still fairly common, is called a *directed acyclic graph*, or DAG. A DAG is much like an AST, except that it is not a tree structure, but a graph. This means any node can be connected to any node, and lots of complicated group and category theory ensues. Because a single node can be referenced by any other node, it does mean that it takes less memory to store the same program, and is likely more efficient overall. The problem lies within it's complexity: DAGs are hard to manage, and when it goes wrong, it goes very, very wrong. However, it is clear that a DAG is better in almost every way than an AST, if you are willing to deal with the headaches.

For the sake of simplicity and understanding, Intercept uses an AST, or abstract syntax tree, and generates it from recognized sequences of lexemes.

A tree, in computer programming, is a data structure that can branch into multiple data structures. An AST happens to be a recursive tree, in that each of the branches of the tree can have branches, and those branches can have branches, and so on. A lot of compilers use a binary tree, or btree, which is a tree with a recursive left hand side and right hand side. The problem you see in using binary trees is that it is rather difficult to represent anything with more than two children (as one might expect). This

is quite a common case in programming language ASTs; for example, when calling a function, there may be any amount of arguments. Most compilers go the LISP route, and have these sort of lists implemented as recursive pairs, where as long as the lhs is another pair, the list continues. I feel like this is a messy implementation, and results in code that is quite hard to read, due to chains of dereferences, i.e. `lhs->lhs->rhs->value`.

The compiler includes a text-debug format to print the abstract syntax tree that is produced, and can be printed out for any program by adding the verbose command line flag: `-v`.

Let's take a look at an example.

```
1  ;; Intercept Parsing Example
2  fact : integer (n : integer) = integer (n : integer) {
3    if n < 2 {
4      1
5    } else {
6      n * fact(n - 1)
7    }
8  }
9
10 fact(5)
```

The parser would produce the following abstract syntax tree.

```
PROGRAM
  VARIABLE DECLARATION
    SYM:"fact"          <- colon separates type from value
  VARIABLE REASSIGNMENT
    SYM:"fact"
    FUNCTION
      SYM:"integer" (0) <- level of pointer indirection
      NONE              <- stores parameter types
        SYM:"integer"
      NONE              <- stores body expressions
        IF
          BINARY OPERATOR:"<"
            VARIABLE ACCESS:"n"
            INT:2
          NONE          <- stores 'then' body
            INT:1
          NONE          <- stores 'otherwise' body
            BINARY OPERATOR:"*"
              VARIABLE ACCESS:"n"
              FUNCTION CALL
                SYM:"fact"
                BINARY OPERATOR:"-"
                  VARIABLE ACCESS:"n"
                  INT:1
  FUNCTION CALL
    SYM:"fact"
    INT:5
```

As you can see, this is a complete representation of the program. It is able to understand high level concepts that the programmer construed in the source code, like conditional control flow, variable accesses, function calls with arguments, and more.

Type Checking

Type-checking refers to the process of ensuring the types of expressions are what they are expected to be.

This is only relevant due to Intercept being statically typed. This means the type of a variable is known at compile-time, due to the programmer declaring it. This declared type is associated with the variable, and any expressions assigned to it must return a compatible type. It also comes into play with binary expressions: they operate on two objects, each of some type. It wouldn't make any sense to try to add a function to an array, would it? Function calls' arguments must match a function's declared parameters' types. A dereference may only operate on a pointer. All of these semantics of the language are enforced by the type-checker.

Code Generation

Code generation is—by far—the most complicated part of the compiler. It is what inspired me to create this document, so as to ease further developers' use of similar architectures, or even help them understand this compiler so that they can make a pull request. Keep in mind that the compiler is a moving target, so this documentation is "as up to date as can be". Things change all the time, and technical writing is hard.

5.1 Terminology

IR Stands for *Intermediate Representation*. An intermediate representation is like a high level assembly language that isn't platform specific.

IR Function A list of basic blocks that may branch to each other.

IR Basic Block A list of instructions in which control flow begins at the start and exits at the end. That is, a basic block must have only one branch instruction, and it must be at the very end.

IR Instruction A tagged union, storing the type and value of the instruction, as well as other metadata like its allocated register and a unique ID for debugging.

Backend The code that translates the intermediate representation into actual, usable code. For example, the x86_64 code generation backend converts IR into x86_64 assembly code.

5.2 Source Code → IR

```
1  4 + 4 + 4 + 4
```

The IR for this program is as follows.

```
f1
  bb1
     %1 | 4
     %2 | 4
     %3 | add %1, %2
     %4 | 4
     %5 | add %3, %4
     %6 | 4
     %7 | add %5, %6
     %8 | return
return value: %7
```

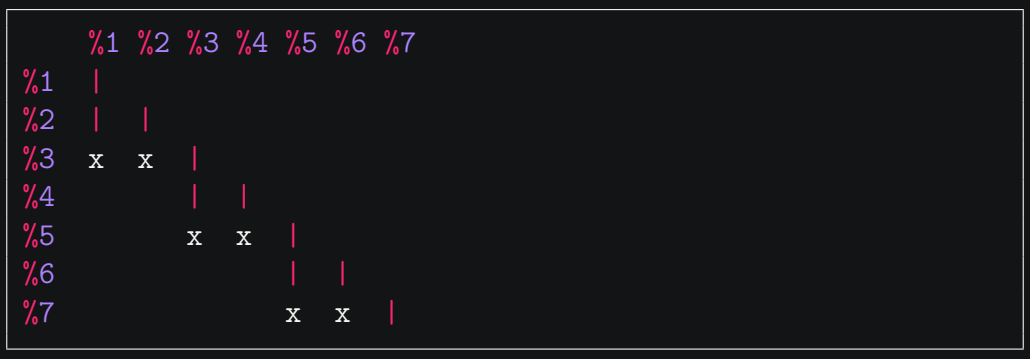
As you can see, there is a single IR function *f1*, a single basic block *bb1*, eight IR instructions within that basic block, with a branch instruction at the end: *%8*.

Another thing you may notice is that the addition no longer happens between numbers, but between IR instructions. For example, *%3* produces the sum of instructions *%1* and *%2*. It is these instructions that refer to an immediate value. This extra layer of abstraction helps to separate things that require computation into their own instruction. That is, an immediate value needs to be loaded before addition can take place.

The major advantage of this representation is something called *Single Static Assignment*, or SSA. SSA refers to a property of a program that means no value gets assigned to twice. This makes it *much* easier to do things like register allocation, and program optimization.

For example, we are able to (*somewhat*) easily compute whether or not a variable is live at any given point in the program, by checking it's latest use and it's definition. Liveness analysis is done by keeping track of every use of some calculated value in a linked list of `Use` structures (pronounce like the noun, the 'S' is voiceless). This is done during intermediate representation generation. Effectively, we build our IR with live ranges built-in.

Here are the live ranges from the above IR example.



What happens next is rather complicated, and there is probably 100 pages we could write to describe it: *Register Allocation.*

Register allocation refers to assigning a hardware register to each instruction in the IR that produces a value. This allows the code generation backend to use the allocated register, and not have to keep track of which to use itself. It also allows for less code duplication, and quite a few optimizations to take place.

The problem of register allocation may be framed in a more formal sense. That is, rather than asking the question, "How do I assign registers to each of the instructions that needs them?", we should be asking, "How do I color a graph with less than k colors?". Yes, register allocation is just graph theory. Each instruction refers to a vertex, or a node, in the graph. The edges of the graph are made by interferences between instructions. An interference occurs when two instructions' results are in use at the same point in the program. It's reasonable that if both are being used at once, then you can't store them in the same place without getting in the way of each other.

So, with this example, we would end up with seven instructions that need a register. After all, a return statement doesn't actually need to be stored anywhere. During register allocation, we collect a flat list of instructions that need registers, and disregard all of the other instructions. This allows the graph to reduce in size, making the problem easier to solve.

The register allocator also simplifies the IR in other ways, attempting to lower PHI instructions, remove COPY instructions, etc. It also pre-allocates any registers that are known ahead of time at this point, like function call arguments according to call convention, and function return values.

Next, the register allocator creates an *adjacency matrix*. An adjacency matrix stores a boolean value for every unique pair of coordinates possible in a two dimensional array that is the size of the flat list of instructions just collected. It is effectively the data version of the live range visualization above.

```
0 |
1 |  1
2 |
3 |        1
4 |
5 |              1
6 |
  |  0  1  2  3  4  5  6
```

Take note that we now use index into the flat list of instructions to refer to each instruction, and that these are different from the IDs we were using before. This is why there is no percentage next to the label numbers.

While it may be confusing at first, it actually makes a lot of sense. As *%0* and *%1* are both live at the same time, they must be stored in separate temporary registers, and therefore interfere. However, as they both are dropped to ground by *%3*, it isn't necessary that all three are stored in separate registers; that is, *%3* could be stored in the same hardware register as *%0* or *%1*. This is why they are said to not interfere, and the adjacency matrix contains a zero (shown as blank space).

From here, the register allocator creates adjacency lists from the adjacency matrix. Adjacency lists are another data structure to represent the same data that the matrix represents, just in a more convenient form for what we are about to do.

An adjacency list stores a list of adjacencies for each node, or vertex, in the register allocation graph. So, any time you see a 1 in the adjacency matrix, the node at both the x and y will have each other added to each other's adjacency list.

Here is what the adjacency lists look like from the above example. Keep in mind that the number with a percentage refers to ID, and the number without refers to index.

```
%1::0: 1
%2::1: 0
%3::2: 3
%4::3: 2
%5::4: 5
%6::5: 4
%7::6:
```

The register allocator now creates what is called a *coloring stack*. This is a stack, or list, of indices that *should* be able to be colored with less than k colors. However, it isn't guaranteed, as this problem is NP-complete, and therefore the solution we find may not be 100% ideal. Currently, we use the Chaitin–Briggs algorithm of register allocation (more or less).

Here is the coloring stack generated by the above program. This is where things get really complicated, so I'm pretty sure we don't have this 100% correct, hence the duplicate zeros.

```
0, 5, 4, 3, 2, 1, 0
```

You'll notice that some indices are missing from this list; this is due to being pre-allocated (i.e. function call arguments according to calling convention).

The register allocator may now, finally, color the graph (and therefore allocate the registers). To do this, the register allocator colors the graph in order of the coloring stack, making sure to avoid coloring a node with the same color as any of the nodes within the adjacency list of the node being colored. This means that no two values that interfere will be allocated the same register.

Our final IR ends up looking like so, when registers are allocated for the x86_64 architecture.

```
f1
  bb1
     %1 | RAX | 4
     %2 | RBX | 4
     %3 | RBX | add %1, %2
     %4 | RAX | 4
     %5 | RBX | add %3, %4
     %6 | RAX | 4
     %7 | RAX | add %5, %6
     %8 |     | return
return value: %7
```

5.3 It's About Φ

Φ, or Phi, is a Greek letter that is used to describe a certain type of instruction within the compiler's intermediate representation. This instruction represents a merging of control flows. For example, the return value of an IF expression will always be a PHI node; this is because IF may branch, or not branch, causing different control flows to be taken that end up in the same destination. The purpose of this instruction is to be able to keep track of the different values that may be returned from the IF expression, and make sure that they end up in the expected places. On most backends, this means allocating registers, but it could be solved in any way, really.

Let's take a look at a real example.

```
1 cond : integer = 1
2 a    : integer = 0
3 if cond {
4   a := a + 5
5   a
6 }
7 a
```

The above source code would be compiled into the following intermediate representation.

```
f1
  bb1
     %1 | RAX | 1
     %2 |     | g.store %1, cond
     %3 | RAX | 0
     %4 |     | g.store %3, a
     %5 | RBX | g.load cond
     %6 |     | branch.conditional %5, bb2, bb3
  bb2
     %7 | RDX | g.load a
     %8 | RCX | 5
     %9 | RCX | add %7, %8
    %10 |     | g.store %9, a
    %11 | RCX | g.load a
    %12 | RAX | copy %11
    %13 |     | branch bb4
  bb3
    %14 | RBX | 0
    %15 | RAX | copy %14
    %16 |     | branch bb4
  bb4
    %17 | RAX | phi [bb3 : %15], [bb2 : %12]
    %18 | RAX | g.load a
    %19 |     | return
return value: %18
```