Compilers

INF-400

Burak Arslan
ext-inf400@burakarslan.com

Galatasaray Üniversitesi

Lecture IX
2023-12-21

Course website

burakarslan.com/inf400

Our compiler so far ...

We have covered the front-end phases:

- Lexical analysis
- Parsing
- Semantic analysis

Next are the back-end phases:

- Code generation
- Optimization [1]

[1] Out of scope of this course

Code Generation

. . . ok but;

What code we will generate? For what platform?

Code Generation

WebAssembly - Overview

Our target language is **WebAssembly**:

- A virtual ISA, descendant of asm.js
- In continuous development
- Many runtime environments (Web, WASI, . . .)
- Many implementations (V8, SpiderMonkey, etc.)

Code Generation

WebAssembly - Overview

WebAssembly 2.0:

- Still a working draft (ie. not yet fully standardized)
- Partially implemented in popular platforms
- We need it because we want garbage collection!
  - Enabled by default in **Chrome** $\geq$ 119 and **Firefox** $\geq$ 120

Code Generation

WebAssembly - Overview

WebAssembly 2.0 Text format (extension: `.wat`):

- Has 1-to-1 correspondence with the binary format [2]
- Based on **S-expressions**
- Peruse its grammar from: https://webassembly.github.io/spec/core/bikeshed#text-format

[2] There is apparently sort of a minor impedence mismatch but it won't affect us

Code Generation

WebAssembly - Overview

WebAssembly 2.0 Text format (extension: `.wat`):

- **This is going to be the actual output of our compiler**
- We will use `wat2wasm` in our compilation pipeline in order to create the actual wasm binary

Code Generation

WebAssembly - Overview

So we got the answers to our questions at the beginning:

- We will generate WebAssembly 2.0 Text Format
- We will target Firefox 120+ and Chrome 119+

Code Generation

WebAssembly - Overview

Analogous answers if the target language was x64:

- We will generate code for Intel Broadwell architecture
- We will target GNU/Linux 4.14

Code Generation

WebAssembly - Concepts

WebAssembly implements a **stack machine**:

- Sequentially executed instructions.
- Instructions manipulate values on an implicit operand stack

Code Generation

WebAssembly - Concepts

WASM has two types of instructions:
(this means it's not a pure stack machine)

- **Simple instructions**: Pop arguments from the operand stack and push results back
- **Control instructions**: Alter program flow:
  - Control flow is **structured** – it's expressed with well-nested constructs such as blocks, loops, and conditionals.
  - This means eg. no jumps that can land on arbitrary addresses

Code Generation

WebAssembly - Concepts

WebAssembly types are:

- **Four basic number types**: `i32`, `i64`, `f32`, `f64`.
  `i32` type also serves as Boolean and as memory addresses.
- **A single 128 bit wide vector type** representing either 4 32-bit, or 2 64-bit IEEE 754 numbers, or either 2 64-bit integers, 4 32-bit integers, 8 16-bit integers or 16 8-bit integers.
- **An Opaque reference** type that represent pointers towards different sorts of entities.
- **An array of function handles**.
  - In WASM terms, they are called **tables**

Code Generation

WebAssembly - Concepts

Emphasis on:

> *i32 type also serves as [. . .] memory addresses.*

This means any WASM program is limited to 4GB of memory!

Code Generation

WebAssembly - Concepts

WebAssembly code has native functions:

- Functions can take and return zero or more sequential values.
- Functions can have local mutable variables
- There is an unobservable implicit call stack – recursive calls are possible.

Code Generation

WebAssembly - Concepts

WebAssembly code can produce **traps**:

- They can't be handled by WASM code,
- Execution halts – it's the platform's job to clean up the mess.

Code Generation

WebAssembly - Concepts

WebAssembly code works on a single[3] contiguous memory block:

- It's a mutable block of raw types
- Out-of-bounds access results in a trap
- Memory segments can grow but not shrink

[3] Multiple memory blocks proposal is not yet accepted.
https://github.com/WebAssembly/multi-memory/issues/50

Code Generation

WebAssembly - Concepts

A WebAssembly binary takes the form of a **module**:
It contains definitions for:

- Functions
- Tables
- Linear memory segments
- Global variables
- Initialization data for memory segments or tables
- A start function that is automatically executed.

Code Generation

WebAssembly - Concepts

Definitions inside modules can be;

- Imported specifying a module/name pair and a suitable type
- Exported under one or more names.

More on stack machines

Code Generation

Stack Machines

Stack machines offer:

- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results
- Sequentially executed Instructions;

Code Generation

Stack Machines

Execution means to:

- Pop operands from the top of the stack (as many as needed)
- Perform the required operation on them
- Push the result back to the top of the stack

Code Generation

Stack Machines

Quite simple to implement as:

- Each operation takes operands from the same place and puts results in the same place
- This means a uniform compilation scheme

Code Generation

Stack Machines

Results in more compact programs because:

- Location of the operands is implicit
  - Always on the top of the stack
- No need to specify operands explicitly
- No need to specify the location of the result
- Instruction "add" as opposed to "add r1, r2"

Code Generation

Stack Machines

One example as to why it's also fast:

- The add instruction does 3 memory operations:
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed
- Idea: keep the top of the stack in a register (called accumulator)
  - Register accesses are faster
- The add instruction is now: `acc += top_of_stack`
  - Only one memory operation!

Code Generation

Stack Machines

An example:

```
(module
    (func $add (param i32) (param i32) (result i32)
        local.get 0
        local.get 1

        i32.add
    )
    (export "add" (func $add))
)
```

Runtime environment

The implementation of the ensemble of abstractions embodied in the language definition is called
a runtime environment

Runtime environment

The compiler runtime deals with details like;

- The layout and allocation of storage locations for the objects named in the source program
- The mechanisms used by the target program to access variables
- The linkages between procedures
- The mechanisms for passing parameters
- The interfaces to the operating system, eg. input/output devices and other programs

Runtime environment

The kiraz compiler runtime answers questions like:

- The size of a byte (in binary data)
- The size of a character (in a string)
- The size of an integer
- The layout of the members of a class
- etc.

Runtime environment

Alignment

- On most hardware platforms, data on memory needs to align with (ie to start from) certain memory addresses.
- If a word is 4 bytes, the starting address of word-aligned data needs to be a multiple of 4.
- Unaligned access is either;
  - Disallowed
  - Slow

Runtime environment

Alignment

- WebAssembly doesn't require aligned access
- But real machines generally do!
- Finding the fastest access pattern requires:
  - Doing lot of profiling
  - Doing it on every new platform release (new hardware, new virtual machines etc.)

Runtime environment

WASM Loader

First part of the compiler runtime is the **loader**:

- A compiled binary is just a bunch of bytes
- Loader is the program that parses the executable format
- Sets the stage for the target code to run

Runtime environment

WASM Loader

The host platform needs:

- The entry point of the binary (In our case, the `main()` function)
- Resources that the binary needs (eg. memory, storage, graphics canvas)
- Platform facilities that the binary needs (eg. functions used for storage access, network access, graphics manipulation, hardware acceleration etc.)

Runtime environment

WASM Loader

Kiraz runtime is pretty static:

- No graphics access
- No input from outside world
- Only text output to the console

. . . which simplifies the loader quite a lot

Runtime environment

WASM Loader

Analogous answers if the target platform was Android:

- Various app permissions (access to contacts, network, storage, position)
- Subject to battery optimizations?
- Program may change behavior based on screen size, amount of ram, device orientation (portrait/landscape)
- Storage of secrets like login tokens, private keys etc.

Runtime environment

WASM Loader

The host platform needs to know:

- The entry point of the binary (In our case, the `main()` function)
- Resources that the binary needs (eg. amount of memory, access to storage (which kind?), acess to graphics (canvas? webgl?))
- Platform facilities that the binary needs (eg. functions used for storage access, network access, graphics manipulation, hardware acceleration etc.)

Runtime environment

WASM Loader

The wasm binary needs access to:

- Handles to the functions that give access to various platform facilities
- Memory

Runtime environment

WASM Loader

Since we are targeting the Web Platform, our loader is . . .

Runtime environment

WASM Loader

Since we are targeting the Web Platform, our loader is . . .

a HTML document!..