CodeGeneration

Diony Rosa

October 20, 2014

Contents

1 Building & Running

Steps to build and run our DAMAJ Decaf Compiler.

These instructions assume you are running on Athena.

Clone the repository or otherwise obtain a copy of the code.

Run `add -f scala`. use `scala -version` to make sure that scala is using version 2.11.2. We use `fsc` to compile scala a little faster. If you happen to have any trouble because of weird fsc version conflicts, please try killing your fsc server (find it with `ps aux | grep fsc`). Don't set any env variables like `SCALA_HOME` as this could force you to use the wrong version of scala.

Once you have the right version of scala run `make` or `build.sh` to compile the project. If you have trouble with this or later steps, try 'make clean' to reset the build files.

Now you should be able to run the compiler using run.sh. For example:

```
./run.sh -t assembly --debug tests/codegen/input/01-callout.dcf
./run.sh -t assembly --debug tests/codegen/input/04-math2.dcf -o 04-math2.asm
```

Alternatively, you can use `compile.sh` to compile and run a program. `./compile.sh tests/codegen/input/01-callout.dcf` This will assemble the program into `tmp/out.S`, print the assembly, create an executable at `tmp/out`, and run the executable, printing its output as well.

2 IR Preprocessing

Relevant files: `src/scala/compile/ir_to_ir2/*` In part 2, we generated an IR for the decaf progam (see `docs/semantic_checker/Lab2.pdf`). In part 3, we start by applying a series of transformations on the IR. We want to do operations at the highest level we can, so that it's easier to debug and simpler to convert to a low-level IR. We also modified the IR slightly, so that any statement which has a condition (if, while, for) now also takes a list of statements that must be executed before the condition is evaluated.

2.1 Turn And/Ors into If/Else

To implement short-circuiting, we turn And/Or expressions into an If/Else block and a temp var that stores the result. Inside the If/Else block, the variable set according to the result of the sub-expressions.

2.2 Turn Ternaries into If/Else

Ternaries are expanded straightforwardly into if/else blocks.

2.3 Flatten Expressions

Expressions (`IR.Expr`) have become non-recursive. The arguments of expressions are always loads (either `LoadLiteral` or `LoadField`). So a statement such as `a = b + c + d` turns into two lines; `_t0 = b + c; a = _t0 + d`.

There is a temp generator in `src/scala/compile/TmpVarGen.scala` that handles the naming of new temporary variables.

2.4 Expressions are only in Assignments

In places like conditionals, or the argument to Return, they now only accept a Load instead of an Expression. That way, the code generation is more straightforward.

2.5 Simplify Whiles

Any while which had a max number of iterations now has that encoded in it's condition expression.

2.6 Turn Fors into Whiles

For loops were converted to while loops. So for example `for (i=0; 10) {...}` is now `i=0; while (i<10) {...; i = i + 1}`

2.7 Initialize variables to 0

All variables declared are immediately assigned value 0 or false.

3 New low-level IR

Relevant files: `src/scala/compile/IR2.scala` The low-level IR is named `IR2`. A program is comprised of:

- A list of fields
- A list of methods, with the `main` method saved separately.

The callouts were discarded, since we don't need to declare them in assembly.

3.1 Fields

A `Field` is simply a string ID and a size. In assembly, a portion of memory (or space on the stack, for locals) is allocated for the field and given the ID as a label, so we can reference fields by their ID.

3.2 Methods

A `Method` has an ID, a list of parameters (`Field~s), and a ~CFG`. Like fields, methods are labelled by their ID in assembly so that they can be called by name. They are also augmented with a `SymbolTable` and a return type, to give context to the statements inside.

3.3 CFG

The Control Flow Graph, or CFG, is the central data structure in `IR2`. It has:

- A start `Block`. This is the entry point to the CFG.
- An end `Block`. This is the unified exit point. It exists so that CFGs can be easily chained.
- A map from ~Block~s to ~Transition~s. Every block except the end block must have a transition from it.

3.3.1 CFG Blocks

A `Block` in the CFG is different than an `IR.Block`. ~Block~s in the CFG are a sequence of simple (non-branching) statements. It also contains a SymbolTable.

3.3.2 CFG Transitions

There are two types of transitions, `Edge` and `Fork`.

- `Edge` is when one CFG block will always go to a next CFG block. It is represented as `Edge(destination)`, where the source block is the key in the map. So an edge from block `a` to `b` would be an entry in the map like `a -> Edge(b)`.
- `Fork` is when the execution flow could jump to 2 different places, depending on a condition. A `Fork` has a `condition`, a `Block` to jump to if the condition is true, and a `Block` for if it's false.

4 Code Generation

Relevant files: `src/scala/compile/AsmGen.scala`

4.1 Calling Conventions

We will use the System V AMD64 ABI calling convention as described by http://en.wikipedia.org/wiki/X86_calling_conventions#x86-64_calling_conventions
Arguments:

- First 6 args passed in registers RDI, RSI, RDX, RCX, R8, R9.
- Additional args passed on the stack with the first arg at the latest of the stack.
- Callee saves RBP, RBX, R12-R15. All other regs must be assumed munged.
- Return value returned in RAX

The only exception is that we actually save all registers besides `rax` before each call. It's overly cautious, but allows for fewer mistakes.

4.2 Division by Zero

Damaj does not handle division by zero, but a valid way to prevent division by zero before the processor has to call a interupt, is to insert assembly code that checks the divisor at run time and jumps to exit handler code if it equals zero. If it equals zero, the exit handler will terminate the program and have an exit value of -3 (or some other value the programmer would know about)

4.3 DSL

We wrote a domain-specific language for assembly instructions. Each instruction is a scala function and there are helper functions as well. This way, we can generate assembly strings in a way that looks quite similar to writing assembly directly.