CS352s2014 Project 5: Register Allocation

March 28, 2014

1 Objective

The objective of this project is to write a register allocator for MiniJava SSA, as represented in the classes `edu.purdue.cs352.minijava.ssa.*`.

2 Code

The project's code environment is available on the course web page. We are providing Java class files for a complete compiler, including the register allocator, but are not providing accompanying Java source. Through the course of these projects, it will be your job to rewrite the Java source for each component, creating a complete compiler of your own design. You are advised to keep a copy of the Java class files so that you can use them to generate correct output and test against your own.

You are free to use the Java class files provided in lieu of your own implementations of the lexer, parser, SSA compiler and type checker, or to use your own if you prefer.

2.1 Testing

The compiler provides a number of tools for testing each stage. They are available in the `bin` directory. Two are of value for testing the register allocator.

- `bin/mjcompile-ssa`: Compiles MiniJava code into SSA form, optionally type checks and allocates registers, and writes it to standard output. To enable register allocation, use the `-r` flag. The `-t` flag (type checking) is not required.
- `bin/mjcompile-mips`: Compiles MiniJava code into MIPS assembler, using type-checked, register-allocated SSA as an intermediary.

The behavior of your register allocator should follow precisely the algorithm described in class, but may not be precisely the same as the version provided, as the final selection of a register is nondeterministic (all registers are considered equally).

3 Register Allocation

Register allocation is performed over SSA statements. All MiniJava values are the same size (equal to the size of one register), so all registers may be considered equal.

The MIPS linkage convention assures that all methods may have their register allocation performed independently. Register allocation therefore is performed for each SSAMethod completely independently of all others.

The steps in register allocation are liveness analysis and register allocation. Liveness analysis involves unifying variables and determining their liveness over the control-flow graph. Your code must construct the control-flow graph, unify the variables, and determine the `in[n]` and `out[n]` sets for every control-flow node. With that information, it must use a graph-coloring approach to assign registers.

You are expected to implement an efficient register allocator, and are recommended to implement precisely the algorithm presented in class. You will need to implement any necessary datatypes to perform this task.

The register allocator you will implement is generic, and does not need to know particular register names for a target platform. Registers are simply numbered from 0 to (register_count - 1).

Because spills are possible, a register allocator may in fact generate new SSA statements; namely, `Store` and `Load`. Some `SSAStatement`s may be pinned (have their registers assigned a priori), and you are expected to respect this pinning when assigning registers. To test with register pinning, you will need to use `bin/mjcompile-mips` (which pins the `Arg` operation to argument registers) rather than `bin/mjcompile-ssa`, or write your own testing code.

4 Goals

For this assignment, you must implement one Java class: `edu.purdue.cs352.minijava.backend.RegisterAllocator`. The interface to `RegisterAllocator` is the `alloc` method, which takes an `SSAProgram` and number of registers as arguments. `alloc` performs register allocation, assigning registers to `SSAStatement`s and generating `Store`s and `Load`s as necessary. `alloc` cannot fail.

A template will be provided in `edu/purdue/cs352/minijava/backend/RegisterAllocator.java-template`.

An allocator which is correct but inefficient (generating unnecessary `Store`s and `Load`s) will receive a grading deduction. You are recommended to implement precisely the algorithm described for this reason.

A number of examples are provided in the `examples` directory. These examples will help in testing, but are *not* the examples that will be used to grade your submission. In particular, none of these examples are incorrect MiniJava. The frontend used to test both SSA compilation and register allocation is `bin/mjcompile-ssa`, which in turn uses the `edu.purdue.cs352.minijava.SSACompilerFrontend` class. You are not expected to reimplement `SSACompilerFrontend`, only `RegisterAllocator`.

4.1 Example

Given the following file in `examples/test.java`:

```
class Simple {
    public static void main(String[] a) {
        System.out.println(42);
    }
}
```

`SSACompilerFrontend` should output something similar to:

```
program:
main:
method main:
0(0):  Int *42:  int
1(0):  Print 0:  void
```

Please note that your output may not be identical to the canonical one. It is not mandatory that you assign precisely the *same* registers as the canonical one does, only that you do not make inefficient use of registers.

5 Submission

You will submit your project code via the `turnin` command. Please include the entire project code directory, including all the class files provided by us, the `bin` and `examples` directories, etc.

To turn in a project directory: `turnin -c cs352 -p proj5 <project directory>`

This project is due by Friday, April 11, 2014 at 12:30PM Purdue time. i.e., before class, *not* before midnight. Late submissions will not be accepted.

6 Grading

We will test your code on machines similar to those in the Linux labs used by this course's PSOs. Your grade will be based on correctness in the following criteria:

- Correct unification of variables from `SSAStatement`s.
- Generation of CFG nodes from `SSAStatement`s.
- Generation of a complete CFG from a method body.
- Correct `in[n]` and `out[n]` computation (liveness analysis).
- Generation of the variable interference graph.
- Correct interference graph simplification.
- Register selection from a simplified graph.
- Rewriting of statement bodies to include spills when necessary.
- Correct generation of non-interfering spills.