Compiler construction 2011

Lecture 2

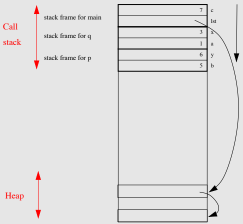
Code generation 1: Generating Jasmin code

- JVM and Java bytecode
- Jasmin
- Naive code generation

Memory at runtime (general, not JVM-specific)

```
int main () {
  int c = 7;
  List lst = ...
  q(3);
  ...
}
void q(int x) {
  int a = 1;
  p(2*x);
  ...
}
void p(int y) {
  int b = 5;
  ...
}
```

Before return from p



The Java Virtual Machine

Data types

- **Primitive types**, including integer and floating-point types of various sizes and the `boolean` type.
  The support for `boolean` is very limited; Java (and Javalette) boolean expressions are compiled to `int` values.
  Javalette needs only `int` and `double`.
- **Reference types**, used for references to objects; not needed by basic Javalette (but by the array extension).

Data areas

- Local variables and parameters are stored on the JVM **stack** (since Javalette is single-threaded there is only one stack).
- Objects (including Java arrays) are stored in the **heap**. The heap is not used by basic Javalette.

JVM stacks

Frames

- A JVM stack consists of **frames**. A new stack frame is allocated for each method invocation.
- Different JVM invocation instructions:
  - `invokestatic` for static methods; use this for Javalette functions.
  - `invokevirtual` for instance methods; not needed for Javalette.
  - `invokespecial` for special cases, e.g. initialization. Use in class file header.
- JVM handles bureaucracy of method invocation:
  - Allocating and deallocating frames.
  - making parameters available to invoked method.
  - making return value available to invoking method.

The structure of a frame

Local variables array

An array of words containing parameters and local variables.

- An `int` value requires one word, a `double` value two words.
- Parameters occupy the first part of the array, followed by locally declared variables.
- All variables are referred to using their index in this array.
- The size of the array is specified in the class file.

Operand stack

A stack of words for temporary storage. Many JVM operations manipulate this stack.

- Also here, of course, `double`'s require two words.
- Results from method invocations are left on top of stack.
- Maximal size of stack is specified in the class file.

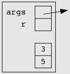
A simple example, 1

Example (`public` omitted)

```
class A {
 static void main(String[] args){
    int r = f(3,5);
    System.out.println(r);
 }
 static int f (int x, int y){
    int r = x+y+2;
    return r;
 }
}
```

JVM Stack

Snapshot: in `main`, before `invokestatic f`:



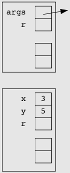
A simple example, 2

Example (`public` omitted)

```
class A {
 static void main(String[] args){
    int r = f(3,5);
    System.out.println(r);
 }
 static int f (int x, int y){
    int r = x+y+2;
    return r;
 }
}
```

JVM Stack

Snapshot: in `f`, before first instruction



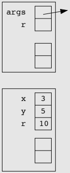
A simple example, 3

Example (`public` omitted)

```
class A {
 static void main(String[] args){
    int r = f(3,5);
    System.out.println(r);
 }
 static int f (int x, int y){
    int r = x+y+2;
    return r;
 }
}
```

JVM Stack

Snapshot: in `f`, before `return`:



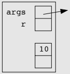
A simple example, 4

Example (`public` omitted)

```
class A {
 static void main(String[] args){
    int r = f(3,5);
    System.out.println(r);
 }
 static int f (int x, int y){
    int r = x+y+2;
    return r;
 }
}
```

JVM Stack

Snapshot: in `f`,
after `invokestatic f`:



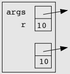
A simple example, 5

Example (`public` omitted)

```
class A {
 static void main(String[] args){
    int r = f(3,5);
    System.out.println(r);
 }
 static int f (int x, int y){
    int r = x+y+2;
    return r;
 }
}
```

JVM Stack

Snapshot: in `main`,
before invoking `println`:



Java bytecode

General properties

- Instructions to
  - push values on the operand stack,
  - store stack top in a variable,
  - do arithmetic on operands on the operand stack,
  - jump (conditionally) to other instructions,
  - invoke and return from methods,
  - ...
- One byte opcodes; ca 250 different instructions.
- Instructions may have arguments: (small) constants, indices to pool of (bigger) constants, local variable indices.
- Compact format, suitable for mobile code.
- Binary format, no official assembly form.

Jasmin assembler

Download it! (Link on course web site)

Unzipped directory contains `jasmin.jar` .
To assemble Jasmin file `myfile.j` containing Jasmin assembler code, you run

```
> java -jar jasmin.jar myfile.j
```

This produces `myfile.class`, which can be run by `java` interpreter.
Note that classpath must be set so `jasmin.jar` is found.
Option `-d` *`path`* writes the class file in directory *`path`*

Disassembling

```
> javap -c myfile
```

prints an assembler version of `myfile.class` on stdout in (almost) Jasmin syntax.

Jasmin instructions 1

Arithmetic

Push integer/string constant *c*: `ldc` *c*
Push double constant *c*: `ldc2_w` *c*
Perform binary operation on integers: `iadd isub imul idiv irem iand ior`
Perform binary operation on doubles: `dadd dsub dmul ddiv`
Booleans are treated as integers: false = 0, true = 1.

Typed operations

Different operations depending on type: e.g. `iadd` and `dadd`.

Also load/store operations are typed.

Consequence: You will need to know for all subexpressions which type they have.

You compute this during type-checking; we now see the benefit of saving this information.

Pushing values on the stack

Integer constants

- Small values: `iconst_1` pushes integer 1.
  Similarly for -1, 0, 2, 3, 4, 5.
- A little bigger: `bipush` *n* pushes *n*, for $-128 \leq n \leq 127$.
- Even bigger: `sipush` *n* pushes *n*, for $-32768 \leq n \leq 32767$.
- Arbitrary: `ldc` *n* pushes *n*.
  Value of *n* stored in constant pool, index in the instruction.
  `jasmin` handles constant pool; you can write constants.

To consider

You will need a datatype of instructions (in Haskell) or a class hierarchy (in Java/C++).
But will you need all four forms (10 opcodes) of push instructions?

Similar considerations for loading/storing local variables.

Jasmin instructions 2

Loading local variable to stack

Load (push) integer variable *n*: `iload` *n*.
If *n* = 0,1,2,3, there are one-byte variants `iload_0`, etc.
Load (push) double variable *n*: `dload` *n*.
If *n* = 0,1,2,3, there are one-byte variants `dload_0`, etc.

Storing stack top to local variable

Store (and pop) integer variable *n*: `istore` *n*.
If *n* = 0,1,2,3, there are one-byte variants `istore_0`, etc.
Store (and pop) double variable *n*: `dstore` *n*.
If *n* = 0,1,2,3, there are one-byte variants `dstore_0`, etc.

Increment of a variable can be done without loading and storing:
`iinc 1 17` increases varable nr 1 with 17.

Jasmin instructions 3

Labels

The label `L` itself is an instruction: `L:`
Make sure that all labels in a function are distinct!
In the JVM bytecode, code is stored in an array of bytes and the label is just the index.

Jumps

- Jump (unconditionally) to a label: `goto L`
- Jump if comparison holds between the topmost two integers on stack: `if_icmpeq L`, `if_icmplt L`, etc
- Jump if comparison holds between the topmost integer and zero: `ifeq L`, `iflt L`, etc.
- For doubles, the situation is different: `dcmpl`, `dcmpg` compare the two doubles and returns an integer -1, 0, or 1.

Generating code for expressions

The problem

Input: A type-annotated AST for an expression.

Output: A sequence of Jasmin instructions with **net effect** to push value of expr on the stack.

The solution: Syntax-directed translation

- Constants: Push on stack.
- Arithmetic expressions: Generate (recursively) and concatenate code for left and right operand; last instruction is arithmetic instruction.
- Function call: Generate and concatenate code for all arguments; last instruction is suitable `invoke` instruction.
- Variables: Load variable, using its number.
  But how do you know this number?

Additional input: The code generator's state.

The problem

As we saw on the previous slide, the AST is not enough;
we need to know the index (address) of each variable.
These will be computed by the compiler itself, when generating code for variable declarations.

More generally: Needed state information

- Variable number/index for each local variable (incl parameters).
- Type for each function.
- Index to use for next variable declaration.
- Number to use for next label.
- Current stack depth.
- Maximal stack depth.
- Code emitted so far.

A better way to present translation

Compilation schemes

Pseudocode notation, similar to Haskell, using monadic state operations.

Example

Defining `codeGenExp :: Exp -> Result ()`,
by pattern matching on the abstract syntax
(presented here in concrete syntax)

```
codeGenExp(exp1 + exp2 : int) =
    codeGenExp exp1
    codeGenExp exp2
    putCode [iadd]      -- add to code
    incStack (-1)       -- decrease current depth
-- other cases similar
```

Quiz

Where do you insert calls to `incStack`?

Define a suitable type for the state

In Java/C++

Define a class with methods for accessing and updating the various state components; don't use public instance variables!
Make use of suitable collection classes.

In Haskell

Use a state monad; also here, define suitable monadic functions for accessing and updating the state.

With suitable abstractions you will be able to modify your code easily if your early decisions need to be changed.

Generating code for statements

Again: syntax-directed translation

- Assignment: Generate code for RHS; store in variable (state gives index).
- Declaration: Get next variable index from state and update state with variable/index.
- Return: Generate code for expression; emit `return` instruction.
- Block: concatenate code from statements.
- ...

Some difficulties

- How does block structured scope affect translation?
- How to handle control structures that lead to jumps (`if` and `while`)?

What about block structure?

Result of preceding slide

Each local variable in a method gets its own index.

Example

Example code

```
int f (int x) {
  int a, b;
  ...
  {int a, c;
    ...
  }
  ...
  {int x, y;
    ...
  }
  ...
```

Questions

- How big **must** the local vars array be?
- How can we avoid making it bigger?

Boolean expressions

Booleans as integers

Booleans are treated as integers, translating `false` to 0, `true` to 1.

There are no JVM operations corresponding to the relational operators; we need to use jumps.

```
codeGenExp(exp1 > (exp2 : int)) =
  codeGenExp exp1
  codeGenExp exp2
  lab1, lab2 <- getLabel     -- get fresh labels
  putCode[if_cmpgt lab1,
             iconst_0, goto lab2,
          label lab1,
             iconst_1,
          label lab2]
```


Naive handling of while statemens

while scheme

```
codeGenStm (while (exp) stm) =
    lab1, lab2 <- getLabel
    putCode[label lab1]
    codeGenExp exp         -- push value of exp
    putCode[ifeq lab2]        -- if false, fall through
    incStack (-1)          -- the test popped exp
    codeGenStm stm
    putCode[goto lab1,        -- test again
            label lab2]       -- label for next stmt
```

Check stack effect!

An example: `while (i > 6) i-- ;`

Generated code (assume `i` is var #1)

```
lab1:
   iload_1
   bipush 6
   if_icmpgt lab3
   iconst_0
   goto lab4
 lab3:
   iconst_1
 lab4:
   ifeq lab2
   iinc 1 (-1)
   goto lab1
 lab2:
```

Better code

```
lab1:
  iload_1
  bipush 6
  if_icmple lab2
  iinc 1 (-1)
  goto lab1
lab2:
```

The problem

The good code is not **compositional**, i.e. not built by combining code from the immediate subtrees.

Example, continued

Recall source code

```
while (i > 6)
   i--;
```

Even better code

```
    goto lab2
  lab1:
    iinc 1 (-1)
  lab2:
    iload_1
    bipush 6
    if_icmpgt lab1
```

Comments

- Saves one JVM instruction per loop round
- You can get (almost) this code compositionally by
  - changing while scheme (for you to do!) and
  - changing treatment of Boolean expressions (next slide)

Note: Naive codegen is enough to pass, but better code not so difficult.

Boolean expressions revisited

Used in two different ways

- As test expressions in control structures (`while`, `if`).
- As right hand sides in assignments to boolean variables or actual (boolean) parameters in function calls.

Do code generation differently in these two cases!

Test expression

Define a scheme that takes as arguments

- the test expression and
- two labels to jump to when value is true and false, respectively.

Called from while and if schemes.

To compute Boolean value

Generate code as we indicated before, treating Booleans as integers.
Called from assignment and function call schemes.

May use scheme to the left when code with jumps needed (`&&` and `||`).

Translating function definitions

fundef scheme

```
codeGenDef (typ f (params) stms) =
   forAll params (ty x): addVar ty x
   forAll stms: codeGenStm

   mx   <- getMaxStack
   locs <- getLocals
   nm = jvmName f typ params
   code <- getCode
  -- now we can build and return the Jasmin abstract syntax
  -- for the function using these four values
```

Return checks and code generation

Return check

Recall (from project spec) that this is valid Javalette:

```
int f() {
   if (true)
     return 0;
   else
     {};
}
```

Your return checker must accept this code.

Code generation

You may **not** generate code for `f` that contains a jump to the empty else branch (even if that branch is never taken).

Such code would be rejected by the JVM code verifier.

Conclusion: also code generation must handle literals `true` and `false` as test expressions in `if` and `while` specially.

Unreachable code

A simple example

This is also valid Javalette:

```
int g(int x) {
   while (false)
      x++;
   return x;
}
```

It is, however, illegal as Java code; the statement `x++;` is obviously **unreachable**. Such control structure is not allowed.

`javac` will reject the function.

Code generation

Even if you would generate code with jumps (don't!), it would pose no problems; JVM can run the generated code.

Another Java example

This, surprisingly, is **valid** Java:

```
int g(int x) {
   if (false)
      x++;
   return x;
}
```

Reason: common pattern in conditional compilation ...

Predefined methods: output

Using Java library methods

All the print functions call `System.out.println`.
In Jasmin, `println` gets `out` as a first argument:

```
getstatic java/lang/System/out Ljava/io/PrintStream;
bipush 77
invokevirtual java/io/PrintStream/println(I)V
```

The first instruction pushes a reference to `out` on the stack.
Then we push the value we want to print.
Then `println` is invoked and gets these two arguments.

Predefined methods: input

Input in Javalette vs. Java

To read e.g. an integer in Javalette is simple:

```
  int main () {
    printInt (7 * readInt ()) ;
    }
```

In Java, a bit more is needed:

```
import java.util.* ;

class Read {
    public static void main (String [] args) {
        Scanner in = new Scanner(System.in);
        System.out.println(7 * in.nextInt());
    }
}
```

Avoiding trivial problems

The `Runtime` class

Instead of generating lots of instructions, you can define all predefined methods in a `Runtime` class. Just write it in Java and compile into a `.class` file!

Make sure you only create one `Scanner` object per program run (use a `static` object).

Calls to `printInt` must generate code as calls to `Runtime.printInt` .

Hints for Jasmin code generation

Use Java tools to see what `javac` does

- Write Javalette code as static methods in Java.
- Compile with `javac` and disassemble using `javap -c`; study the results.

Make it simple

- Start with simple code generation. We will accept also naive code.
- Look at code produced by `javac`; it is often straightforward!
- When your compiler runs, you may try to optimize if there is time.

Next time

- First extension: Arrays in JVM.
- JVM runtimes: JIT compilation, memory management.