Compiler Techniques

Lecture 9: Code Generation: Soot

Tianwei Zhang

Outline

- **Overview of Soot**
  - **Jimble IR**
- **Code Generation using SooT**
  - **From AST to Jimble**
  - **From Jimble to Bytecode**
- **Appendix: Soot and Jimple APIs**

Outline

- **Overview of Soot**
  - **Jimble IR**
- **Code Generation using SooT**
  - **From AST to Jimble**
  - **From Jimble to Bytecode**
- **Appendix: Soot and Jimple APIs**

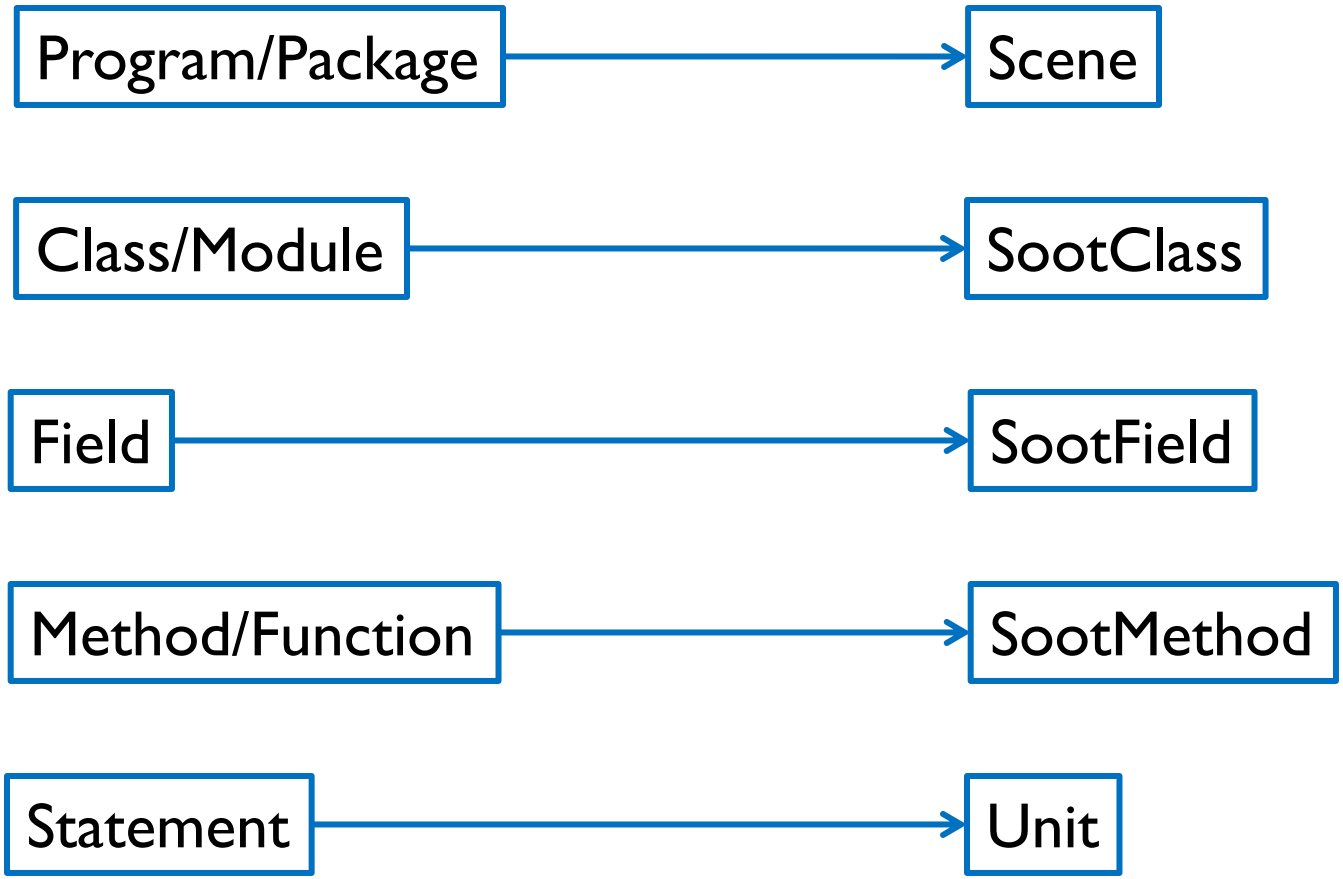
What is Soot

- A framework for generating and optimizing JVM bytecode

- Defines several higher-level intermediate representations for bytecode that are easier to work with, transform and optimize than JVM bytecode
  - **Baf**: similar to bytecode
  - **Jimple**: most important IR; only 15 instructions
  - **Shimple**: variant of Jimple; better for some optimisations
  - **Grimp**: high-level representation with nested expressions

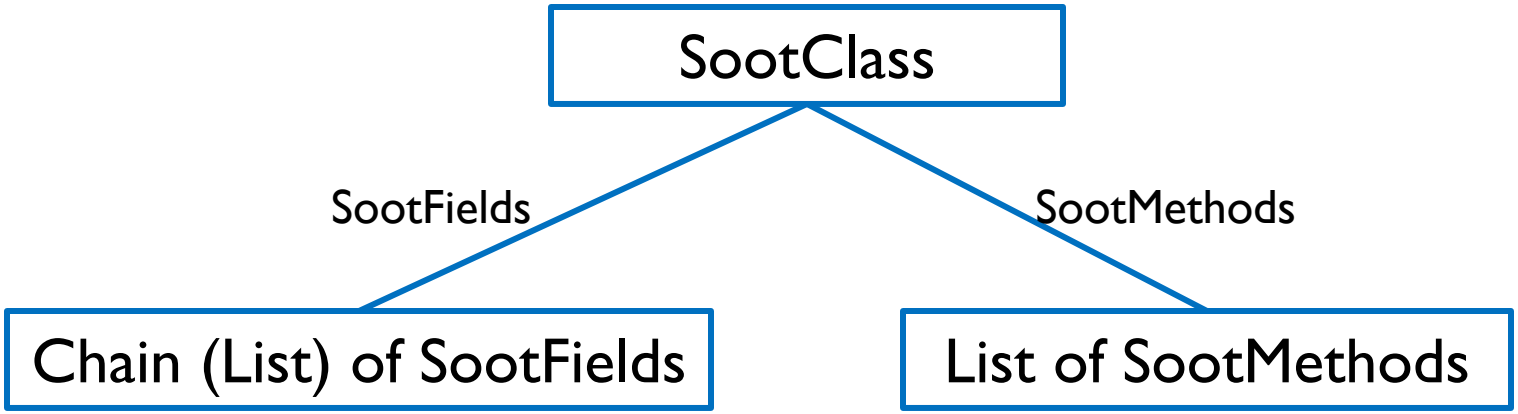
- Provides translators between these four IRs and JVM bytecode

High-level Data Abstractions in Soot



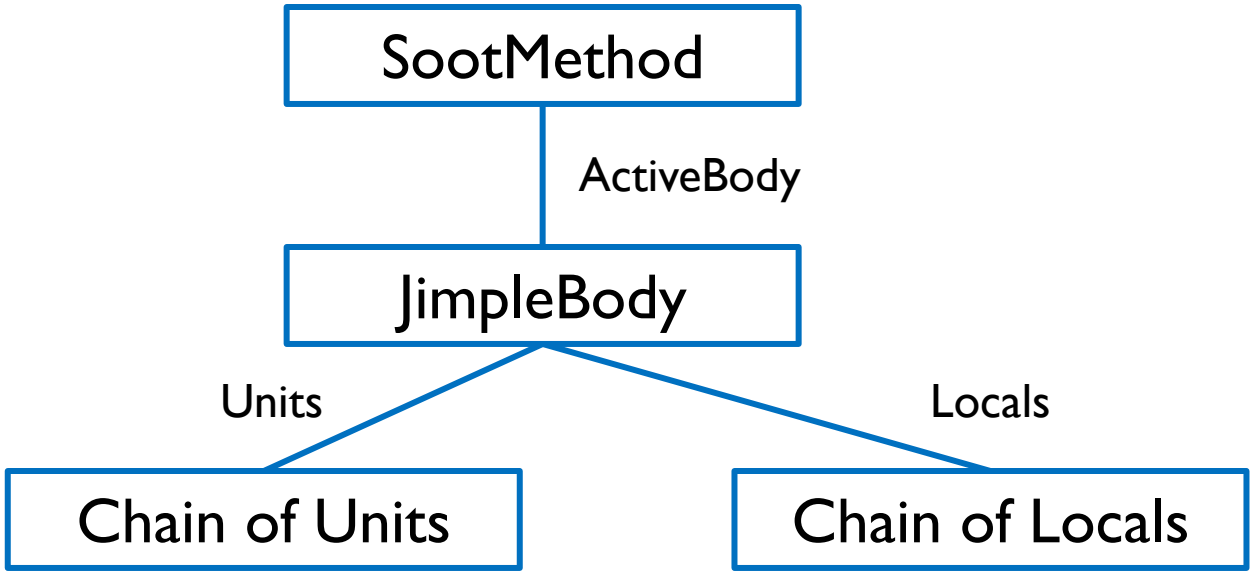
SootClass

- Fields are added to a class with **`SootClass.addField`**
- Methods are added to a class with **`SootClass.addMethod`**



SootMethod

- The Body is set using **`SootMethod.setActiveBody`**
- The Units and Locals are added to the respective Chains
  - e.g. **`body.getUnits()`**`.`**`add`**`(stmt)`
  - e.g. **`body.getLocals()`**`.`**`add`**`(var)`



Jimple

- *3-address code* intermediate representation
- Most instructions work on three operands:
  - two operands are inputs
  - one operand is the output where the result is stored
- Format: x = y *op* z
  - y and z are inputs, x is output
  - y and z themselves cannot be complex expressions;
  - Temporary variables are used to compute nested expressions
- Some instructions have fewer than three operands
- All operands are explicit

Examples of Jimple Expressions and Statements

- To create expressions or statements, we use factory methods in class `Jimple` (use method `Jimple.v()` to obtain the singleton instance of class `Jimple` first)

- Addition expression in Jimple:
  - `Jimple.newAddExpr(Value op1, Value op2)`

- If statement:
  - `Jimple.newIfStmt(CmpExpr cond, Unit target)`

- The "do nothing", i.e., nop statement:
  - `Jimple.newNopStmt()`

Jimple-like 3-address Instructions

- Jimple-like instructions: in 3-address format.
  - Operands should not be complex expressions

- Some typical Jimple-like instructions:
  - `x = y op z:` where x, y and z are variables or literals, and op is an arithmetic operator.
  - `goto L:` where `L` is a label.
  - `if x # y goto L:` where # is a relational operator.
  - `x = y`
  - `return x`

Examples of Jimple-like 3-address Instructions

- Valid 3-address forms:
  - `a = b + c`
  - `x = y`
  - `r = fib(n)`
  - `r = foo(a, b, c)`
  - `if a > b goto label0`

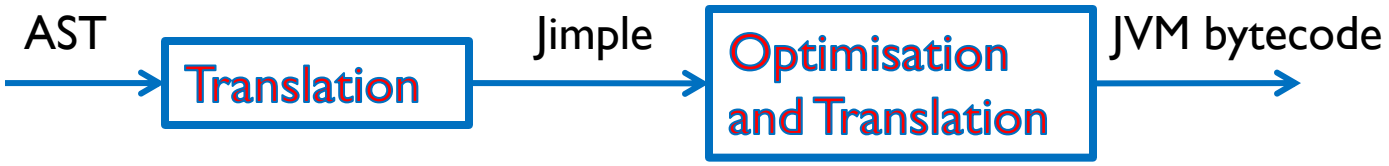
- Invalid 3-address forms:
  - `r = fib(n - 1)`
  - `a = b * c + d`
  - `if a + 1 > b goto label0`

Outline

- Overview of Soot
  - Jimble IR
- **Code Generation using SooT**
  - From AST to Jimble
  - From Jimble to Bytecode
- Appendix: Soot and Jimple APIs

Compiling to Bytecode using Soot

- Our goal: translate AST to Jimple (3-address form) code
- Then Soot translates Jimple to bytecode
- Three major advantages:
  1. Jimple is (slightly) more high-level than raw bytecode, and hence is easier to generate
  2. It is easier to perform optimisations at the Jimple level
  3. We can reuse optimisations already available in Soot



Outline

- Overview of Soot
  - Jimble IR
- **Code Generation using SooT**
  - **From AST to Jimble**
  - From Jimble to Bytecode
- Appendix: Soot and Jimple APIs

Compiling to Jimple

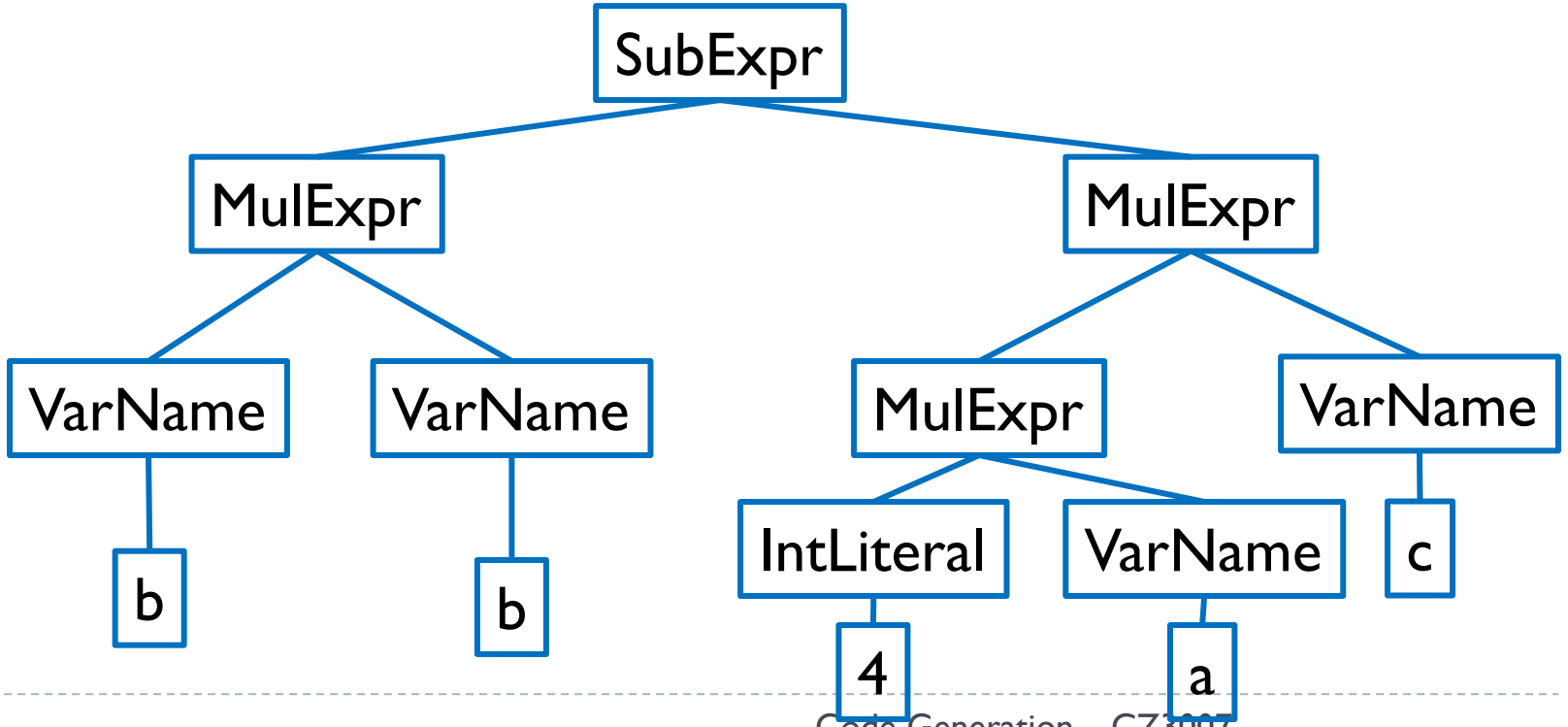
- Two problems need to be solved:
  1. How to map the *data types* and *data abstractions* of the source language to the target language?
  2. How to map the *code abstractions* of the source language to the target language?
- Soot provides same high-level *data abstractions* (classes, objects and arrays) as JVM, so the mapping is straightforward; this is more complicated for native code
- But we still need to map *code abstractions*:
  1. Compile nested expressions to Jimple's 3-address representation
  2. Compile structured control flow to Jimple's conditional jumps

Compiling Expressions

- High-level programming languages allows nested expressions:

```
b * b – 4 * a * c
```

- In Jimple, every instruction only performs a single operation
- Introduce ***temporary variables*** to hold intermediate results



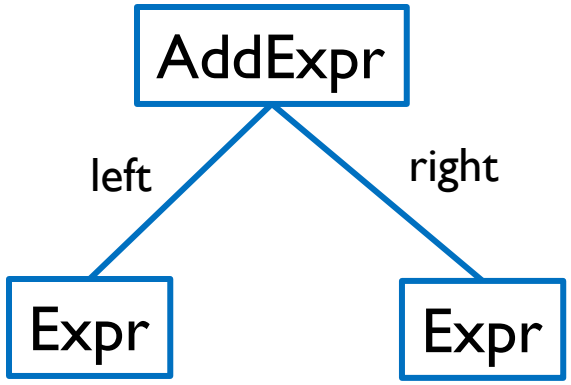
1. Translating Literals, Variables and Fields

- Nothing much to do as there are direct mappings in Soot

2. Translating Binary Operators

- Recursively generate code for the left and right operands
  1. Recursively generate code for operands
  2. If an operand is a complex expression, introduce a fresh temporary variable and generate an assignment to store the operand into the temporary
  3. Construct appropriate final expression

```
tleft = left expr
tright = right expr
res = tleft + tright
```



2. Translating Binary Operators

- Example: `r = b * b – 4 * a * c`

- Compile the above expression into Jimple-like 3-address code, using temporary variables (denoted with a prefix $) to store intermediate results:

```
$i0 = b * b
$i1 = 4 * a
$i2 = $i1 * c
r = $i0 - $i2
```

2. Translating Binary Operators

- Code generation for logical operators (both unary and binary) works the same as arithmetic operators

- Short-circuiting operators are an exception: they are just a shorthand for an **if** statement. So they must be compiled using conditional jumps (see later)
  - if (i < n && a[i] != 0)
  - if (i < n || a[i] != 0)

3. Translating Assignments

- Both left and right hand side are recursively compiled

- The left hand side should be either
  - A local variable: the right hand side can be an expression, so no need to be stored in a temporary
  - A reference to a field or array element: a temporary variable needs to be introduced if the right hand side is an expression:

```
tright = right expr

a[i] = tright
```

4. Translating Method Calls

- Introduce a fresh temporary variable into which to store the returned result if any.

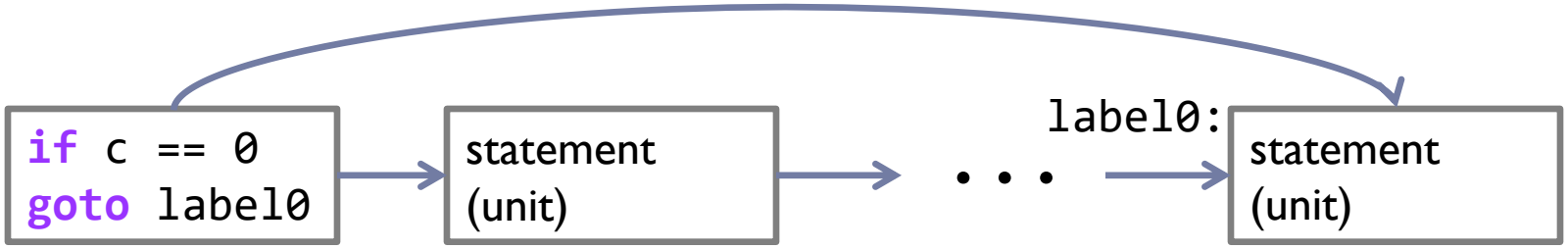
Code Generation for Statements

- Recursively generate code for statements. We first generate code for child expressions (or statements)
  - e.g. for an if statement, we generate code for the condition, then the then part of the statement, then the else part of the statement
- Code generation for statements does not return any value. It simply adds the generated code to the list of statements of the enclosing **`JimpleBody`**
  - A statement is represented by a **`Unit`** in Jimple. Suppose we are generating a new statement and units is the list of statements generated so far:

```
Unit stmt = statement; // generate code for current statement
units.add(stmt);       // add this statement to list
```

Code Generation for Jump Statements

- Use a reference to the Java object representing the statement which is the target of the jump
- When Soot pretty-prints Jimple code, it introduces labels to refer to statements, but these labels do not really exist: they are simply a way to print the statement references in a human-readable way



5. Translating If Statements

- For an **if** statement, we generate the code for the conditional expression and assign the result to an object c of type Value
- If the condition is false, we should skip the **then** part, so we need to generate the target statement for the branching

Basic idea:

```
if(cond)
        stmt1
else
        stmt2
```

```
        // compute value of cond
        c = …
        if c == 0 goto label0
        // code for stmt1
        goto label1
label0:
        // code for stmt2
label1:
```

5. Translating If Statements

- However, when generating the code for the jump, we do not know which statement should be our target statement (*i.e.*, which unit is the statement after the **then** part)

Problem

When we generate this statement, we do not yet know what unit is at label0!

Same problem for `label1`

```
        // compute value of cond
        c = …
        if c == 0 goto label0
        // code for stmt1
        goto label1
label0:
        // code for stmt2
label1:
```

5. Translating If Statements

Possible Solutions:

1. Backpatching: put in **null** references for the jump targets; overwrite once we know the right values

Limitation:

If we have many statements in the **then** part, we need to overwrite the null reference in a jump that was generated a long time ago

```
    // compute value of cond
    c = …
    if c == 0 goto label0
    // code for stmt1
    goto label1
label0:
    // code for stmt2
label1:
```

5. Translating If Statements

Possible Solutions:

2. NOP Padding: Generate two NOP instructions as jump targets, and insert them in the right positions (Soot eliminates NOP when generating bytecode)

```
NopStmt label0 =
        j.newNopStmt();

Unit stmt =
        j.newIfStmt(..., label0);

units.add(stmt);
```

```
// compute value of cond
c = …
if c == 0 goto label0
// code for stmt1
goto label1
label0:
    // code for stmt2
label1:
```

NOP Padding for If Statements

INPUT: if statement

label0 = new **NOP** statement
label1 = new **NOP** statement
c = generate code for if condition
emit statement `if c == 0 goto label0`
generate code for `stmt1`
emit statement `goto label1`
emit statement `label0`
generate code for `stmt2`
emit statement `label1`

```
    // compute value of cond
    c = …
    if c == 0 goto label0
    // code for stmt1
    goto label1
label0:
    // code for stmt2
label1:
```

6. Translating While Statements

Basic idea:

```
while(cond)
        body
```

```
label0:
   // compute value of cond
   c = …
   if c == 0 goto label1
   // code for body
   goto label0
label1:
```

- Same problem as before with label1; can be solved with NOP padding

```
NopStmt label1 = j.newNopStmt();
units.add(label1);
```

6. Translating While Statements

- The condition may be a complex expression
  - Translated recursively
  - Involve assignments to temporary variables at different levels of recursion
  - Not easy to determine the unit that is the start of the condition
- Generate a NOP instruction representing `label0`

```
NopStmt label0 = j.newNopStmt();
units.add(label0);
```

```
label0:
   // compute value of cond
   c = …
   if c == 0 goto label1
   // code for body
   goto label0
label1:
```

Pseudo Code of NOP Padding

INPUT: while statement

`label0` = new **NOP** statement
`label1` = new **NOP** statement
emit statement `label0`
`c` = generate code for condition
emit statement **if** `c == 0` **goto** `label1`
generate code for body
emit statement **goto** `label0`
emit statement `label1`

```
label0:
   // compute value of cond
   c = …
   if c == 0 goto label1
   // code for body
   goto label0
label1:
```

6. Translating While Statements

- Another reason for `label0` and `label1`:
  - A break statement is translated into a jump to `label1`
  - A continue statement is translated into a jump to `label0`
  - For nested while loops, we keep a mapping (e.g., a HashMap) from while loops to their corresponding break targets and continue targets

```
label0:
   // compute value of cond
   c = …
   if c == 0 goto label1
   // code for body
   goto label0
label1:
```

6. Translating While Statements

INPUT: while statement

`label0` = new **NOP** statement
`label1` = new **NOP** statement

// Assume nd is the AST node of the while loop

`put(nd,label0)` in the HashMap for continue targets
`put(nd,label1)` in another HashMap for break targets

```
// etc
```

6. Translating While Statements

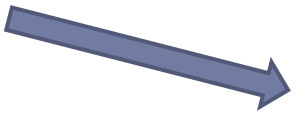
- Call `getEnclosingLoop()` in the AST to return the AST node of the closest enclosing `while` loop
  - Defined during semantic analysis using an inherited attribute

- Break statement: look up the `while` loop node in the HashMap for break targets to get the corresponding target (`label1`) and generate a jump to that target

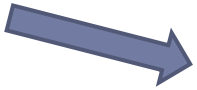
- Continue statement: look up the `while` loop node in the HashMap for continue targets to get the corresponding target (`label0`) and generate a jump to that target

NOP Padding for Nested While Statements

```
while(cond1)
    ...
    while(cond2)
        ...
        continue;
        ...
    continue;
    ...
```

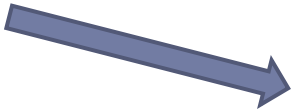


```
label0A = new NOP
emit label0A
(ndA, label0A) → continue HashMap
```



```
label0B = new NOP
emit label0B
(ndB, label0B) → continue HashMap
```

```
ndB= getEnclosingLoop()
label0B =HashMap(ndB)
emit goto label0B
```



```
ndA= getEnclosingLoop()
label0A =HashMap(ndA)
emit goto label0A
```

7. Translating Other Statements

- Expression: simply generate code for the expression
- Block: generate code for each statement in the block in sequence
- Loops and do-while loops: translated to while statements
- Switch: translated to nested if statement
  - Jimple provides special instructions, not discussed here
- Return:
  - for `return expr`: `Jimple.newReturnStmt`
  - for returning without a value: `Jimple.newReturnVoidStmt`
- Exception throwing and handling: not discussed here

Outline

- Overview of Soot
  - Jimble IR
- **Code Generation using SooT**
  - From AST to Jimble
  - **From Jimble to Bytecode**
- Appendix: Soot and Jimple APIs

From Jimple to JVM Bytecode

- Four main steps necessary for this conversion:

1. Perform a tree traversal of the Jimple code, directly translating it to naïve Baf stack machine code
2. Optimise the naïve Baf by eliminating redundant store/load instructions
3. Pack local variables to minimize the number of slots used
4. Translate the Baf code directly to JVM bytecode, calculating the maximum height of the operand stack for each method

1. Direct Translation to Naïve Baf

- Direct translation

```
x = y op z;
```

3-address code

=>

```
load y
load z
op
store x
```

- Example

```
x = (y * z)/2;
```

Java

=>

```
$i0 = y * z;
x = $i0 / 2;
```

3-address code

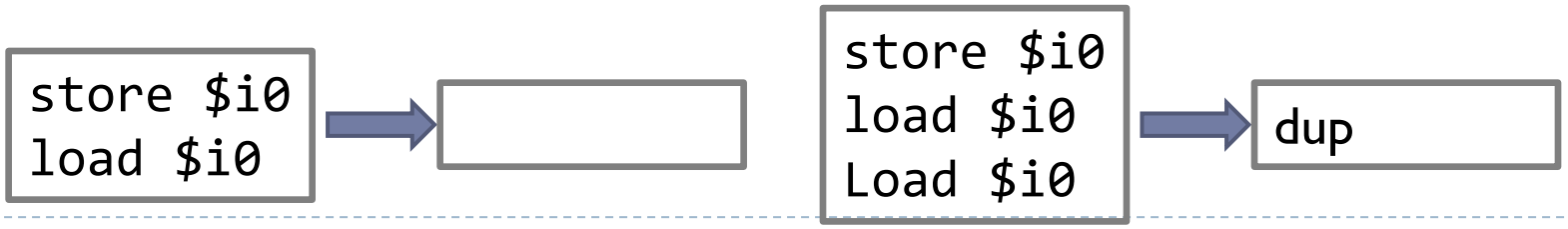
=>

```
load y
load z
mul
store $i0
```

```
load $i0
load 2
div
store x
```

2. Optimising Naïve Stack Code

- Eliminate redundant store/load instructions
  - store/load: a store instruction followed by a load instruction referring to the same local variable *with no other uses*
    - both the store and load instructions can be eliminated, and the value will simply remain on the stack
  - store/load/load: a store instruction followed by two load instructions, all referring to the same local variable *with no other uses*
    - the three instructions can be replaced by a dup instruction to duplicate the value left on the stack



2. Optimising Naïve Stack Code

- Easy to eliminate redundant patterns when the relevant instructions directly follow each other
- Difficult when there are intermediate instructions between the relevant instructions

```
store $i0
load $i0
```

```
store $i0
mul
load $i0
```

```
store $i0
push a
push b
mul
store c
load $i0
```

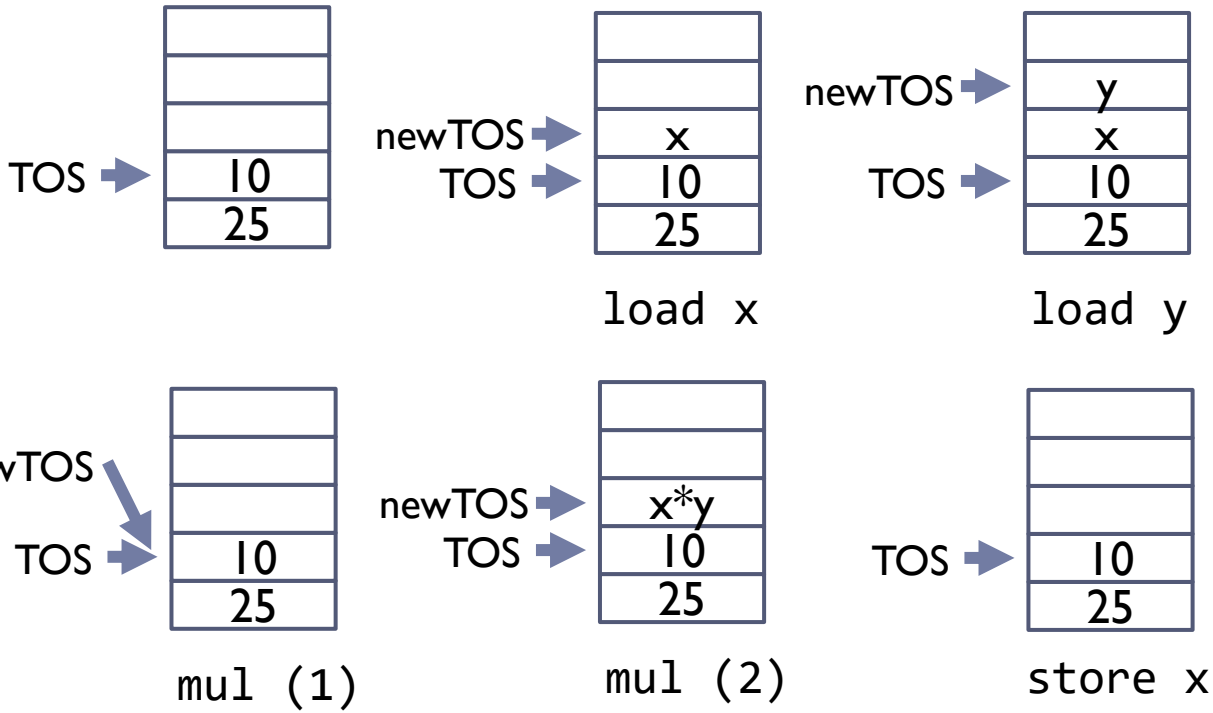
2. Optimising Naïve Stack Code

- Judgement
  - net stack height variation (nshv): the net change of top-of-stack pointer from its original position
  - minimum stack height variation (mshv): the lowest position of the top-of-stack pointer with respect to the original position
  - We can eliminate the pair (or triple) only if nshv and mshv are both equal to zero
    - If nshv or mshv is not zero, it may be possible to reorder the stack machine instructions to make them zero

Examples

- Intermediate instructions:

```
load x
load y
mul
store x
```

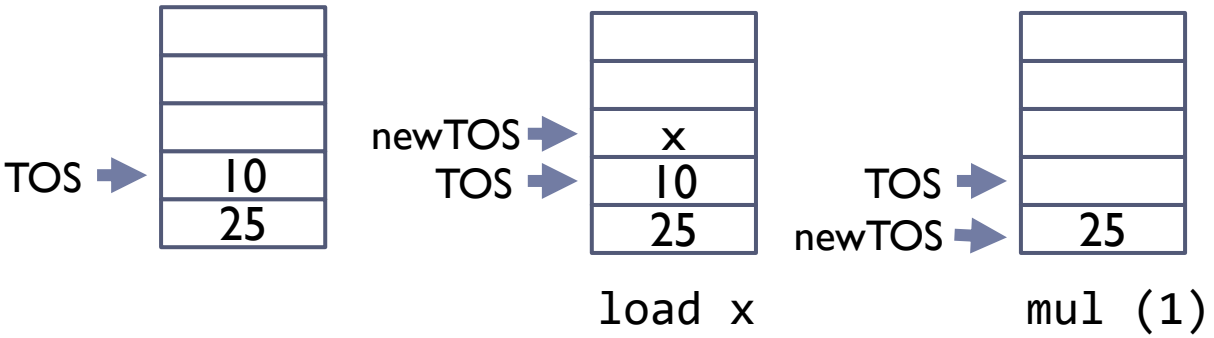


- nshv = 0
- mshv = 0

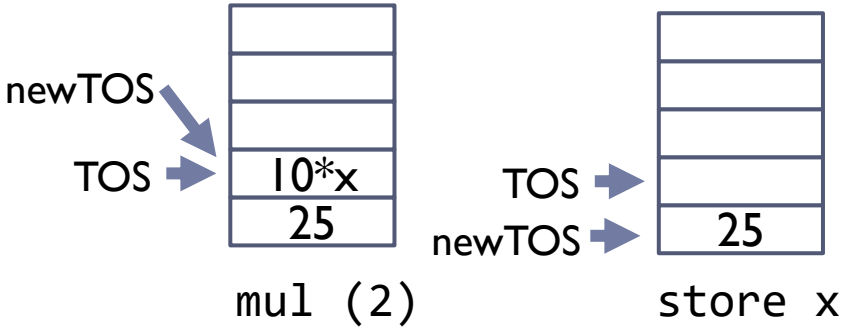
Examples

- Intermediate instructions:

```
load x
mul
store x
```



- nshv = -1
- mshv = -1



2. Optimising Naïve Stack Code

- Patterns might be interleaved with each other, so we need to iterate until no more can be eliminated

- Example

```
r = b*b - 4*a*c;
```

Java

=>

```
$i0 = b * b;
$i1 = 4 * a;
$i2 = $i1 * c;
r = $i0 - $i2;
```

3-address code

=>

Instr	nshv	mshv
load b load b mul store $i0		
load 4 load a; mul store $i1	+1 +2 +1 0	0 0 0 0
load $i1 load c mul store $i2	+1 +2 +1 0	0 0 0 0
load $i0 load $i2 sub store r		

2. Optimising Naïve Stack Code

- The instructions between store $i0 and load $i0 have nshv = 0 and mshv = 0 so this store/load pair can be eliminated

Instr	nshv	mshv
load b load b mul ~~store $i0~~		
load 4 load a; mul store $i1	+1 +2 +1 0	0 0 0 0
load $i1 load c mul store $i2	+1 +2 +1 0	0 0 0 0
~~load $i0~~ load $i2 sub store r		

2. Optimising Naïve Stack Code

- The instructions between store `$i0` and load `$i0` have nshv = 0 and mshv = 0 so this store/load pair can be eliminated
- load `$i1` directly follows store `$i1`, so this pair can be eliminated

Instr	nshv	mshv

`load b`
`load b`
`mul`		

`load 4`
`load a;`
`mul`
~~`store $i1`~~
	+1
	+2
	+1
	0
		0
		0
		0
		0

~~`load $i1`~~
`load c`
`mul`
`store $i2`
	+1
	+2
	+1
	0
		0
		0
		0
		0

`load $i2`
`sub`
`store r`		

2. Optimising Naïve Stack Code

- The instructions between store `$i0` and load `$i0` have nshv = 0 and mshv = 0 so this store/load pair can be eliminated
- load `$i1` directly follows store `$i1`, so this pair can be eliminated
- After eliminating store `$i0` and load `$i0`, load `$i2` directly follows store `$i2`, so this pair can be eliminated

Instr	nshv	mshv

`load b`
`load b`
`mul`		

`load 4`
`load a;`
`mul`
	+1
	+2
	+1
	0
		0
		0
		0
		0

`load c`
`mul`
~~`store $i2`~~
	+1
	+2
	+1
	0
		0
		0
		0
		0

~~`load $i2`~~
`sub`
`store r`		

2. Optimising Naïve Stack Code

- The instructions between store `$i0` and load `$i0` have nshv = 0 and mshv = 0 so this store/load pair can be eliminated
- load `$i1` directly follows store `$i1`, so this pair can be eliminated
- After eliminating store `$i0` and load `$i0`, load `$i2` directly follows store `$i2`, so this pair can be eliminated
- This gives the final optimised stack machine code as shown

Instr	nshv	mshv

`load b`
`load b`
`mul`		

`load 4`
`load a;`
`mul`
	+1
	+2
	+1
	0
		0
		0
		0
		0

`load c` `mul`	+1 +2 +1 0	0 0 0 0
`sub` `store r`		

3. Packing Local Variables

- Minimize the number of local slots used
  - Local variables whose lifespans do not overlap may share the same slot
  - This is similar to register allocation and uses an interference graph and graph colouring algorithm (will be discussed in Lecture 12)

```
z = x + y
y = z + y
```

y
x/z

3. Packing Local Variables

- Minimize the number of local slots used
  - Local variables whose lifespans do not overlap may share the same slot
  - This is similar to register allocation and uses an interference graph and graph colouring algorithm
  - Two packings are performed, one for 32-bit variable slots and another for 64-bit variable slots

4. Translate Baf to JVM

- JVM requires that the maximum operand stack height be specified for each method
- This can be computed by performing a simple traversal of the stack machine code and recording the effect that each instruction has on the stack height

Native Code Generation

- Compiling JVM code ("HotSpots") to native code:
  - Register allocation: native instructions often require that operands be in registers
  - Optimisation: to make native code run even faster
  - Instruction selection: to choose and generate the appropriate native code instructions
  - Runtime support: e.g., creating new objects on the heap

Outline

- Overview of Soot
  - Jimble IR
- Code Generation using SooT
  - From AST to Jimble
  - From Jimble to Bytecode
- **Appendix: Soot and Jimple APIs**

Appendix: Soot and Jimple APIs

- The slides in this part are useful for the last lab assignment.
- They are not examinable.

The Soot API

- Soot provides a Java API for reading in classes from external files, or constructing them from scratch using any of the IRs
- The most fundamental data structure is class `soot.Scene`; it has a single instance, available through method `Scene.v()`
- The scene keeps track of all the classes that have been loaded from bytecode or created by Soot (automatically)
- It also stores analysis information about the entire program (as opposed to individual classes or methods)
- Method `Scene.loadClassAndSupport` is used to load a Java class from the standard library
- Usually, we want to load `java.lang.Object` first thing:

```
Scene.v().loadClassAndSupport("java.lang.Object");
```

Representing classes

- Each class or interface is represented as a `SootClass` object
- Can be constructed from scratch:

```
SootClass klass=new SootClass("tst.Test",Modifier.PUBLIC)
```

  This creates a new public class with qualified name tst.Test

- A previously loaded/constructed class can be obtained from the scene by its fully qualified name:

```
SootClass object =
        Scene.v().getSootClass("java.lang.Object")
```

- After creating a new SootClass, we need to set its super class:

```
klass.setSuperclass(object)
```

- We can also use `SootClass.addInterface` to add interfaces that the class implements
- Soot (like JVM bytecode) considers interfaces as classes with a special modifier **interface**

Representing types, fields and methods

- Types are represented by objects of type `soot.Type`:
  - Singleton classes (class with single instance) for primitive types, e.g. `IntType` represents `int`, an instance can be generated using `IntType.v()`
  - Class `RefType` represents class/interface types, can be obtained through `SootClass.getType()`
  - Class `ArrayType` represents array types
- Fields are represented by class `soot.SootField`:

```
SootField(String name, Type type, int modifiers)
```

- Methods are represented by class `soot.SootMethod`:

```
SootMethod(String name, List<Type> paramTypes,
           Type returnType, int modifiers,
           List<SootClass> thrownExceptions)
```

Method Bodies

- Every `SootMethod` has a body; there are different types of bodies, one for each intermediate representation
- We will be working with class `JimpleBody`, which represents method bodies implemented in Jimple
- Each body keeps track of three things:
  1. A list of local variables (class `soot.Local`)
  2. A list of statements, called "units" in Soot (class `soot.Unit`)
  3. A list of exception handlers, called "traps" in Soot (class `soot.Trap`)
- These are represented using Soot's own implementation of Lists (class `soot.Chain`), which mostly has the same methods as `java.util.List`

Local variables

- Local variables are represented by class `soot.Local`
- New locals are created by factory method

  ```
  Jimple.newLocal(String name, Type type)
  ```

- Add them to method body like this:

  ```
  body.getLocals().add(var)
  ```

- Parameters are represented in the same way, but they need to be initialised in a special way
- Local variables in Jimple can be of any type including **byte**, **short**, and **char**; Soot will map them to the appropriate stack types when translating to bytecode

The Jimple API: Statements

- There are 15 statement types in Jimple:
  1. **Assignments**
  2. **Identity statements**
  3. **If statements**
  4. **Goto statements**
  5. **Invocation statements**
  6. **Return statement**
  7. **Return void statement**
  8. **Nop statement**
  9. Tableswitch statement
  10. Lookupswitch statement
  11. Throw statement
  12. Monitor enter statement
  13. Monitor exit statement
  14. Breakpoint statement
  15. Ret statement

We will not use types 9-15

Jimple Operands

- Operands can be:
  - Local variables (interface `soot.Local`)
  - Field/array element reference (interfaces `soot.jimple.FieldRef` and `soot.jimple.ArrayRef`)
  - Constants (class `soot.jimple.Constant` and subtypes)
  - Expressions (interface `soot.jimple.Expr` and subtypes)
- There are many different kinds of expressions
- Expressions cannot be nested: the operands of an expression can be locals or constants, but cannot be other expressions
- All statements and expressions must be constructed using factory methods in class `Jimple`

Jimple Expressions

- The usual arithmetic and logical expressions are represented by subclasses of `BinopExpr` and `UnopExpr`; as with bytecode, there is a special `LengthExpr` for determining array length
- `CastExpr` and `InstanceOfExpr` represent cast expressions and `instanceof` checks
- For each expression type, there is a factory method in class `Jimple` (use method `Jimple.v()` to obtain the singleton instance of class Jimple)
  - e.g. `Jimple.newAddExpr` to create an add expression

Assignments and Identity Statements

- An assignment (`soot.jimple.AssignStmt`) has a left operand and a right operand (accessor methods `getLeftOp` and `getRightOp`)
- Assignments are constructed by factory method

  `Jimple.newAssignStmt(Value left, Value right)`

  Here, `left` should be a local or a reference to a field
- Identity statements (`soot.jimple.IdentityStmt`) are special assignments that copy a parameter into a local; there should be one identity statement for every parameter

  `Jimple.newIdentityStmt(Local l, ParameterRef p)`
- `ParameterRef` objects are constructed by

  `Jimple.newParameterRef(Type type, int index)`

  Here, `index` is the number of the parameter, `type` is its type

If statements and goto statements

- An if statement (`soot.jimple.IfStmt`) contains a condition and a target (accessor methods `getCondition` and `getTarget`)
- If statements are constructed by factory method

```
Jimple.newIfStmt(CmpExpr cond, Unit target)
```

- The target is the statement where execution continues if the condition evaluates to true (otherwise it continues at next statement)
- A goto statement (`soot.jimple.GotoStmt`) is similar to an if statement, except that it has no condition: execution always continues at the target

Method Calls

- Methods are identified by method references (`soot.SootMethodRef`), which can be constructed using method `Scene.makeMethodRef`
- Instance method calls are represented by class `soot.jimple.InstanceInvokeExpr`; they consist of a method reference, a base value, and a list of arguments
- Static method calls (`StaticInvokeExpr`) are similar, but have no base value
- Method calls to non-void methods usually appear on the right hand side of assignment statements
- Calls to void methods must be wrapped into an invocation statement (`soot.jimple.InvokeStmt`)

Return statements

- There are two types of return statements: `soot.jimple.ReturnStmt` and `soot.jimple.ReturnVoidStmt`
- The former statement returns a value; its factory method is

```
Jimple.newReturnStmt(Value value)
```

- The latter statement does not return a value; its factory method is

```
Jimple.newReturnVoidStmt()
```

- In Jimple (as in JVM bytecode), every control flow path must end either with a throw statement or with a return statement; "falling off" the end is not allowed, even in a **void** method

Other statements

- The nop statement is the same as the JVM nop statement: it does not do anything
- Its factory method is

```
Jimple.newNopStmt()
```

- The other statement types will not be discussed, they are not important for the practical assignment

Factory Design Pattern

- Soot and Jimple make heavy use of the **factory design pattern**
- In Soot, the standard name of v() is used for the (value) factory method, where the value of the instance is important (v stands for value)
  - An example of the use of a value factory method is to implement a constant, e.g. `IntConstant.v(23)` generates the instance representing the integer constant 23
- The **singleton pattern** is a special case of the value factory pattern that takes no arguments – the same object is guaranteed to be returned each time
  - An example is a primitive type, e.g. `IntType.v()` generates the object representing an integer type

Reference material on Soot

- List of tutorials: https://github.com/Sable/soot/wiki/Tutorials
- In particular:
  - Soot survivor's guide: http://www.brics.dk/SootGuide/
  - "Creating a Class File from Scratch"
  - "On the Soot menagerie -- Fundamental Soot Objects"
  - Blog posts on Soot: http://www.bodden.de/tag/soot-tutorial/
- Note, though, that many of these materials are more about implementing compiler optimisation using Soot