Code Generation

Part VI

Phases In a Compiler

Part VI

- A modern compiler is usually partitioned into three major parts:
  1. The front-end – Responsible for validating that the program is legal in the source language.
  2. The "middle-end" – Optimizations.
  3. The back-end – Generating a program in the target language, usually some form of machine code.

More on Phases

Part VI

- The first phase has to be tailored for the particular language to compile but is really independent of the machine to compile to.
- The last phase has to be tailored for the machine to compile to but is to a large extent independent of the source language.
- Some optimizations are dependent on the source language or the target language, but most optimizations are independent.

Intermediate Representation

Part VI

- An intermediate representation (IR) can make the front-end and the back-end truly independent.
- This will make it possible to reuse parts of the compiler in other projects.
  - By plugging in different back-ends we can get compilers for different machines (SPARC, x86, JVM, EVM)
  - By plugging in different front-ends we can get compilers for different languages (Expressive, Java, C, C++)

Intermediate Representation

Part VI

- The intermediate representation has to be carefully designed.
  - It should support constructs for several different programming languages (functions, objects, types).
  - It should be easy to map to several different target machines.
- There are several common ways to structure the IR.
  - 3-address instructions, Control Flow Graphs (CFG), Expression-Trees, Static Single Assignment (SSA)

Optimization

Part VI

- The IR is ideal for optimizing on:
  - Unnecessary syntactical constructs have been removed.
  - Strange machine specific peculiarities have not been introduced.
- By using the IR for our optimization phases we can reuse optimizations in other compilers we write. ;)

Project Note

- In the project we are not going to do any optimizations (or porting).
- Hence, for Expressive we will not use an intermediate representation.
- Instead we will generate (virtual) machine code directly.

The Target Language

- In order to generate code we need to know the target language.
- In our project the target language is the assembly language of a virtual machine called EVM (the Expressive Virtual Machine).
- Before describing code generation in detail we will take a detour and look at the EVM.

Code Generation

- The task of the code generator is to select machine instructions for each construct in the IR (or the AST in our case).
- In Java this can be done by applying a visitor on the tree nodes.
- In functional languages it can be done by pattern matching on tree nodes.

Designing The Code Generator

- Given:
  - A target language – The EVM in our case.
  - An intermediate representation in the form of a tree – the AST in our case
- Find:
  - For each node in the source tree find a combination of instructions in the target language.

The Generator

```
public class Generator implements
 Visitor, expressive.evm.Instructions {
  final Global global;
  final PrintWriter out;
  final Code code = new Code();

  public Generator(Global global,
                   PrintWriter out) {
    this.global = global;
    this.out = out;
  }

  public void generate(Tree tree) {
    tree.apply(this);
  }
…
```


Bottom up Design

- Lets try to design the code generator from the bottom up. That is, we will start with leaves in the AST and then work our way up.
- Since the target machine is a stack machine we will want to get the result of each expression on the top of the stack.
- We will use the Visitor pattern over the AST.
- We assume we have a function emit which produces code for the target.

Literals

- The literals in the AST can contain either a string or an integer.
- Since string literals can only occur in a `prints` node we will ignore them for now.
- For a value what we want to do is to push it onto the top of the stack.

Code for Literals

```
public void caseLiteral(Literal tree) {

  if (tree.str != null) // String or int lit?
   global.fatal(tree.pos,
    "Cannot generate code for a string literal");

  code.emit(PUSH, tree.value);
}
```


Identifiers

- Identifiers represents uses of variables.
- Variables will be stored in memory.
- Each identifier is associated with a symbol.
- The symbol can keep track of where in memory the variable is stored – let's assume it does.

Identifiers (cont)

- We need to get the contents of the variable from the memory to the top of the stack.
- The `LOAD` instruction does this, assuming we know the memory position.

Code for Identifiers

```
public void caseIdent(Tdent tree) {
  // id

  code.emit(LOAD,
            tree.symbol.getOffset());

}
```


Assignment

- The assignment node consists of an identifier and an expression:
  `id := expr`
- The statement should store the value of `expr` to the memory location of `id`.
- We assume we can call `generate` to generate code for an expression.
- We assume that the value of the evaluated expression will be on the top of the stack.
- All we need to do is emit code for storing the top of the stack to memory with `STORE mempos`.

Code for Assignment

```
public void caseAssign(Assign tree) {
  // id :- expr

  // Recursive call to generate code
  //  for the expression expr.
  generate(tree.rhs);

  // Code for storing the result in memory
  //  at the location of id.
  code.emit(STORE, tree.symbol);
}
```


Code

- We have assumed that we have an emit function and that each symbol is allocated to a memory position.
- This is handled by the Code class.
- We keep a vector of emitted instructions, and a count of used memory positions.

The Code Class

```
public class Code implements
   expressive.evm.Instructions {

 private final Vector/*<Integer>*/ code = new Vector();
 private int memSize = 0;
…
 public Code emit(int instr) {
  code.add(new Integer(instr));
  return this;
 }

 public Code emit(int instr, int arg) {
  emit(instr);
  return emit(arg);
 }
```


The Magic in Code

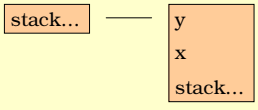
```
public Code emit(int instr, Symbol sym) {
 return emit(instr, declareVar(sym));
}

public int declareVar(Symbol sym) {
 // Offset -1 indicate unallocated symbol.
 if (sym.getOffset() < 0)
  sym.setOffset(memSize++);

 return sym.getOffset();
}
```


Operators

- The operators can be constant (`TRUE`, `FALSE`), unary (`NOT expr`) or binary (`expr ADD expr`, etc).
- For the binary operators we want the right hand side (y) on the top of the stack and the left hand side (x) below that on the stack.



Operators

- We can use `generate` to generate code for the sub trees, first left (x), then right (y).
- Then we just have to figure out what each operation should do.
- Most ops have a direct correspondence in the EVM.

Booleans

- In the EVM Booleans are represented by the integer 0 for false and any other integer for true.
- This means that we can implement the Boolean operators:
  - `! expr` (NOT x => x = 0)
  - `Expr | expr` (x OR y => x + y)
  - `Expr & expr` (x AND y => x * y)
- As:

NOT x:	x OR y:	x AND y:
PUSH 0	ADD	MUL
CMPE		

If

- To generate code for if we will need to use control flow instructions.
- For

  "`if Cond {Statements1} else {Statements2}`"

  we need to first generate code that evaluates `Cond`.

  Then we need to generate code that checks if `Cond` is false (=0) and if so jumps over all the statements in Statements1.

Labels

- In order to easily jump to some point in the code the Code class provides labels.
  - `getLabel()` returns a new label.
  - `markLabel(Label label)` marks the label to point to the current location in the code.
- For the implementation look in Code.java.

...If again

- Armed with labels we can make the following outline for the generated code for `if Cond {Statements1} else {Statements2}`

```
Code for Cond
JZ elseL // Jump if Cond=0 i.e. false
Code for Statements1
elseL:
Code for Statements2
```


...and again

- We do not want to execute the code for `Statements2` after `Statements1` if `Cond` is true:

```
Code for Cond
JZ elseL // Jump if Cond=0 i.e. false
Code for Statements1
GOTO exitL // Skip Statements2
elseL:
Code for Statements2
exitL:
```


Code for if

```
public void caseIf(If tree) {
  // if Cond {Statements1} [else {Statements2}]
  Code.Label elseL = code.getLabel();
  Code.Label exitL = code.getLabel();

  generate(tree.cond);       // Code for Cond
  code.emit(JZ, elseL);      // JZ elseL // Jump if Cond=0 i.e. false
  generate(tree.thenp);      // Code for Statements1
  if (tree.elsep.length > 0)
      code.emit(GOTO, exitL);// GOTO exitL // Skip Statements2
  code.markLabel(elseL);     // elseL:
  if (tree.elsep.length > 0) {
      generate(tree.elsep);  // Code for Statements2
      code.markLabel(exitL); // exitL:
  }
}
```

Part VI

That's about it

- You will get Code.java and a skeleton for Generator.java.
- You will have to figure out how to handle `caseProg`, `caseWhile`, and `caseIO` nodes by yourself.
- You will also have to finish `caseOP`.

Static Vs Dynamic (Compile-time)

- Scopes are static. (It is fixed during interpretation.)
- The abstract syntax tree contains static information.
- Most compiler passes works recursively on the abstract syntax tree and computes static information.

Static Vs Dynamic (Runtime)

- Environments are dynamic. (For each call to a function we create a new environment.)
- The interpreter computes dynamic information.
- The interpreter is not a simple recursion on the abstract syntax tree. In a function call it jumps from the call to its definition to execute the function body.

Static Vs Dynamic (Type Checking)

- For statically typed languages (Java, C, ML) type checking is done statically as part of the analysis.
- For dynamically typed languages (Erlang, Scheme), type checking is done dynamically, at runtime.
- (Some LISP interpreters also did name analysis at runtime – giving you dynamic scoping.)

Environments

- We have both static and dynamic information about identifiers.
- We want two different structures to keep track of the static and the dynamic information, a symbol table and an environment.
- The environment can be implemented as an array of values, the position in the array of an identifier can be stored in the symbol table.

Extending Symbol

```
class Symbol {
  …
  abstract Value load(Environment env);
  abstract void store(Environment env,
                      Value val);
}
```


Local Variables

```
class LocalSymbol extends Symbol {
   int offset;
   Value load(Environment env){
     return env.values[offset];
   }

   void store(Environment env,Value val){
     env.values[offset] = val;
   }
}
```

Global Variables

```
class GlobalSymbol extends Symbol {
   Value val;
   Value load(Environment env){
     return val;
   }

   void store(Environment env,Value val){
     this.val = val;
   }
}
```


Identifiers, version 2

Now we can get the value of an identifier in the interpreter:

```
…
void caseIdent(Tree.Ident tree) {
  // Try to find the symbol.
  Symbol sym = lookup(tree.name);
  if (sym == null) { // Not found.
      error();
  } else {
    result = tree.symbol.load(currentEnv);
  }
}
```


Function Calls

```
void caseFunCall {
  // call interpreter recursively on
  //   function arguments;
  Arguments args = interpret_args(tree.args);

  // Create a new Environment
  currentEnv = new Environment(currentEnv);

  // Store the arguments in the new environment.
  insert_args(args, currentEnv);

  // Call the interpreter recursively on the
  //   body of the called function, using the new
  //   environment.
  result = interpret(find_code(tree.funName));

  // Restore the environment.
  currentEnv = currentEnv.outer;
}
```


Summary Interpreter

- In general the interpreter can be implemented more or less as the Evaluator visitor, but control flow, and especially function calls complicates things a bit.
- We have to keep track of both static and dynamic information about identifiers.

Virtual Machines

- A virtual machine is a simple way to get a portable compiler.
- A virtual machine often resembles real physical machines but they are implemented in software instead of hardware.
- By implementing the VM in a widespread portable language like C (or Java) programs compiled to the machine can be executed on many different platforms.

Virtual Machines (cont)

- A virtual machine is basically just an interpreter for virtual machine code.
- The same techniques can be used to implement a VM as an interpreter.
- The virtual machine code is usually linear (an array of instructions) as opposed to a tree as we used in the interpreter.

Virtual Machines (cont)

- When designing a VM one has some design choices similar to those for the intermediate code.
  - Should the machine be used on several different physical architectures and operating systems? (JVM)
  - Should the machine be used for several different source languages? (CLI/CLR (.NET))
  - Should the machine be efficient and reliable? (BEAM – Erlang).

Virtual Machines (cont)

- I will not go into the details of how to implement a VM.
- If you are interested you can look at the EVM:
  http://lampwww.epfl.ch/teaching/compilationA/2003/html/project/files/EVM.java
- Or better yet, take my course "Compilation Avancée" next summer semester.

Other Topics in Compilation

- Code Generation
  - Register Allocation
  - Instruction selection
  - Instruction scheduling
- Optimization
  - Constant propagation and folding
  - Loop optimization
  - Inlining & Partial evaluation
- Advanced language features
  - Concurrent programming languages
  - Functional languages
  - Object Oriented languages
  - Logic programming languages
  - Advanced type systems
- Runtime Environments
  - Concurrency & Parallelism
  - Memory managements & Garbage Collection
  - Efficient Virtual Machines

Summary: Compilation

- Scanner
  - Converts a stream of characters into a stream of tokens.
  - Can be written by hand or with a scanner generator.
- Parser
  - Converts the stream of tokens to an abstract syntax tree.
  - Recognizes syntax errors.
  - Can be written by hand or with a parser generator.

Summary: Compilation

- Analysis
  - Associates definitions and uses of names by attaching symbols to the AST.
  - Uses scopes to store Symbols.
  - Recognizes undefined variables.
  - Attaches types to the AST.
  - Recognizes type errors.

Summary: Compilation

- Code generation
  - Turns the AST (or some other intermediate representation) into (virtual) machine code.
- Interpretation
  - Executes the user program