pT q6
Am

Compiler Construction 2002

Compiling Techniques y2002p5.tex

(a) Assuming a Java type is given to each variable, state a method by which an overloaded operator (such as +,- etc.) in a Java program can be determined to be an `int`, `real` or other operator. [3 marks]

(b) Explain, using pseudo-code as appropriate, how to convert a syntax-tree into stack code such as that used in the JVM. For the purposes of this question, you only need consider trees representing bodies of `void`-returning functions, and these bodies only as consisting of a list of statements of the form `int` x `=` e`;` or $x = e$; where x ranges over variables and e over expressions; expressions contain variables, integer constants, the binary operator + and static method invocations. [10 marks]

(c) Show how a sequence of simple stack code instructions, such as those used in your answer to part (b) above, can be translated into a sequence of instructions for a register-oriented architecture of your choice, for example ARM or Pentium. [7 marks]

Solution Notes y2002p5.tex

(a) In a language like Java, every variable and function name is given an explicit type when it is declared. This can be added to the symbol table along with other (location and name) attributes. The language specification then gives a way of determining the type of each sub-expression of the program. For example, the language would typically specify that $e + e'$ would have type `float` if e had type `int` and e' had type `float`.

This is implemented as follows. Internally, we have a data type representing language types (e.g. Java types), with elements like `T_float` and `T_int` (and more structured values representing things like function and class types which we will not discuss further here). A function `typeof` gives the type of an expression. It would be coded :

```
fun typeof(Num(k))        = T_int
  | typeof(Float(f))      = F_float
  | typeof(Id(s))         = lookuptype(s)  // looks in symbol table
  | typeof(Add(x,y))      = arith(typeof(x), typeof(y));
  | typeof(Sub(x,y))      = arith(typeof(x), typeof(y));
   ...
fun arith(T_int, T_int  )   = T_int
  | arith(T_int, T_float)   = T_float
  | arith(T_float, T_int)   = T_float
  | arith(T_float, T_float) = T_float
  | arith(t, t') = raise type_error("invalid types for arithmetic");
```

So, when presented with an expression like $e+e'$, the compiler first determines (using `typeof`) the type t of e and t' of e'. The function `arith` tells us the type of $e+e'$. Knowing this latter type enables us to output either a `iadd` or a `fadd` JVM operation.

(b)

```
fun trexp(Num(k))        = gen2(OP_iconst, k);
  | trexp(Id(s))         = trname(OP_iload,s);
  | trexp(Add(x,y))      = (trexp(x); trexp(y); gen1(OP_iadd))
  | trexp(Apply(f, el))    =
                   ( trexplist(el);           // translate args
                     gen2(OP_Invokestatic, f)) // Compile call to f

fun trexplist[] = ()
  | trexplist(e::es) = (trexp(e); trexplist(es));

fun trcom(Assign(s,e))   = (trexp(e); trname(OP_store, s))
  | trcom(InitDecl(s,e)) = (trexp(e); trname(OP_store, s))
```

Function `trname(s)` translates the string form of a name unto an fp-offset using a table `env`. We also need a function `addname(s,k)` which adds a given name to `env` with a given offset.

Then `trbody` first wanders the list of commands in the body calling

```
  addname(s,count++);
```

for every `Initdecl(s,e)` encountered. Then it rescans the body calling

```
  trcom(c);
```

for every command encountered.

(c) Each JVM operation can be translated into a small number of ARM or Pentium operations. Assume the registers SP points to stack fringe and FP to the stack frame, then each intermediate instruction listed above can be mapped into a small number of ARM or Pentium instructions, essentially treating JVM instructions as a macro for a sequence of Pentium instructions. Doing this naïvely will produce very unpleasant code, for example recalling the

```
    y := x<=3 ? -x : x
```

example and its intermediate code with

```
     iload_4        load x (4th load variable)
     iconst_3       load 3
     if_icmpgt L36  if greater (i.e. condition false) then jump to L36
     iload_4        load x
     ineg           negate it
     goto L37       jump to L37
     label L36
     iload_4        load x
     label L37
     istore_7       store y (7th local variable)
```

could expand to (assuming a descending stack and 10 stack locations used for parameters and local variables):

```
movl   %eax,40-16(%fp) ; iload_4
pushl  %eax            ; iload_4
movl   %eax,#3         ; iconst_3
pushl  %eax            ; iconst_3
popl   %ebx            ; if_icmpgt
popl   %eax            ; if_icmpgt
cmpl   %eax,%ebx       ; if_icmpgt
bgt    L36             ; if_icmpgt
movl   %eax,40-16(%fp) ; iload_4
...
```

However, delaying output of PUSHes to stack by caching values in registers and having the compiler hold a table representing the state of the cache can improve the code significantly:

```
      movl   %eax,40-16(%fp) ; iload_4        stackcache=[%eax]
      movl   %ebx,#3         ; iconst_3       stackcache=[%eax,%ebx]
      cmpl   %eax,%ebx       ; if_icmpgt      stackcache=[]
      bgt    L36             ; if_icmpgt      stackcache=[]
      movl   %eax,40-16(%fp) ; iload_4        stackcache=[%eax]
      negl   %eax            ; ineg           stackcache=[%eax]
      pushl  %eax            ; (flush/goto)   stackcache=[]
      b      L37             ; goto           stackcache=[]
L36:  movl   %eax,40-16(%fp) ; iload_4        stackcache=[%eax]
      pushl  %eax            ; (flush/label)  stackcache=[]
L37:  popl   %eax            ; istore_7       stackcache=[]
      movl   40-28(%fp),%eax ; istore_7       stackcache=[]
```