Chapter 16 Code Generation

Registers and Run-time Data Structures

Rather than performing a treewalk to interpret the program, we can generate assembly language. The assembly language can be assembled (once) then executed multiple times, with different inputs. The advantage of compiled code is that it executes faster (on the real architecture, not on a simulated architecture). The disadvantage is that it is very machine dependent.

The B-- compiler needs a model for how memory is managed and accessed.

Almost the same as for the B-- interpreter:

- A static block of memory is allocated for global variables. This memory exists for the duration of the program's execution.
- A dynamic block of memory is allocated for a method when the method is invoked, and deallocated when the method is returned from. This memory is allocated and deallocated, by adjusting the stack pointer register.
- Memory for array and class instance variables is allocated "in-line" as a part of the block of memory for the construct in which the variable is declared (either global memory or the activation record for a method).

  This makes memory management very simple and efficient. However, it lacks the power and ease of use of modern systems with garbage collection.
- String constants and a method table for each class type are also stored as separate blocks of memory. These never change, and exist for the lifetime of the program.

Registers are used to point to the blocks of memory. The compiler's model is more complex than that of the interpreter, because the compiled code has to deal withe the saving and restoring of state when a method is invoked and returned from. These actions are implicit in the interpreter, when it invokes the eval method with the new state as a parameter.

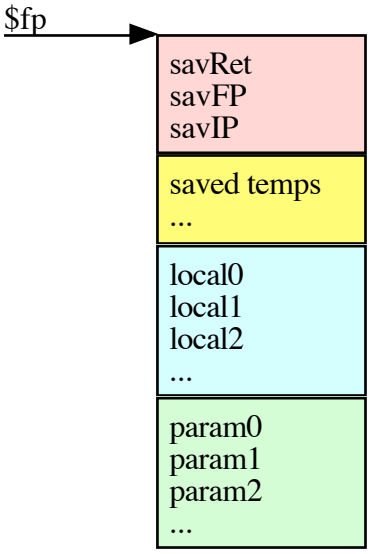
The registers used are:

- The program counter, that points to the next instruction to execute.
- The return address register $ra, that contains the saved value of the program counter, when in the process of invoking or returning from a method.
- The procedure value register $pv, that points to the address of the method to be invoked, when in the process of invoking the method.
- The stack pointer register $sp, that points to the top of stack. The stack grows towards low memory. To allocate or deallocate space on the stack, the amount of space needed is subtracted from or added to $sp.
- The frame pointer register $fp, that points to the activation record of the current method.
- The instance pointer register, $ip, that points to the current instance ("this").
- The new instance pointer register $nip, that points to the new instance, when in the process of invoking a method.
- The global pointer register $gp, that points to the global table of constants.

- The assembler temporary register $at, that is used for very short-term storage, often to implement “pseudoinstructions”.
- Temporary registers $t0, $t1, $t2, ..., to hold the values of operands when evaluating expressions. The return value of a method invocation is stored in $t0, rather than the conventional $v0.

An activation record is composed of:

- Space to save the value of $ra, $fp, and $ip.
- Space to save temporary values, when evaluating complex expressions that require more temporary registers than exist. In reality, this only occurs when evaluating expressions involving method invocations. (In fact, if we evaluate operands in decreasing order of complexity, only an expression with multiple method invocations would have to save temporary values in the activation record.)
- Space for the values of local variables, in order.
- space for the values of parameters, in order.



Method invocations

The algorithm for performing a method invocation is as follows.

In the Invoker

InvocationNode:

- Save the contents of any temporary registers in use in the activation record of the invoker.
- Allocate space on the stack for the parameters of the invoked method, by subtracting the amount of space needed from $sp. This space will become part of the activation record for the invoked method.

ExprListNode:

- Evaluate the actual parameters and store them on the stack, with the ith parameter in the ith quadword on the top of stack.

MethodNameNode:

- Set the $pv and $nip registers to the address of the method to be invoked, and the address of the new instance.

InvocationNode:

- Execute a jsr (jump to subroutine) instruction, to jump to the new method, and save the old value of the program counter in the $ra register.

——————————>

In the Invoked Method

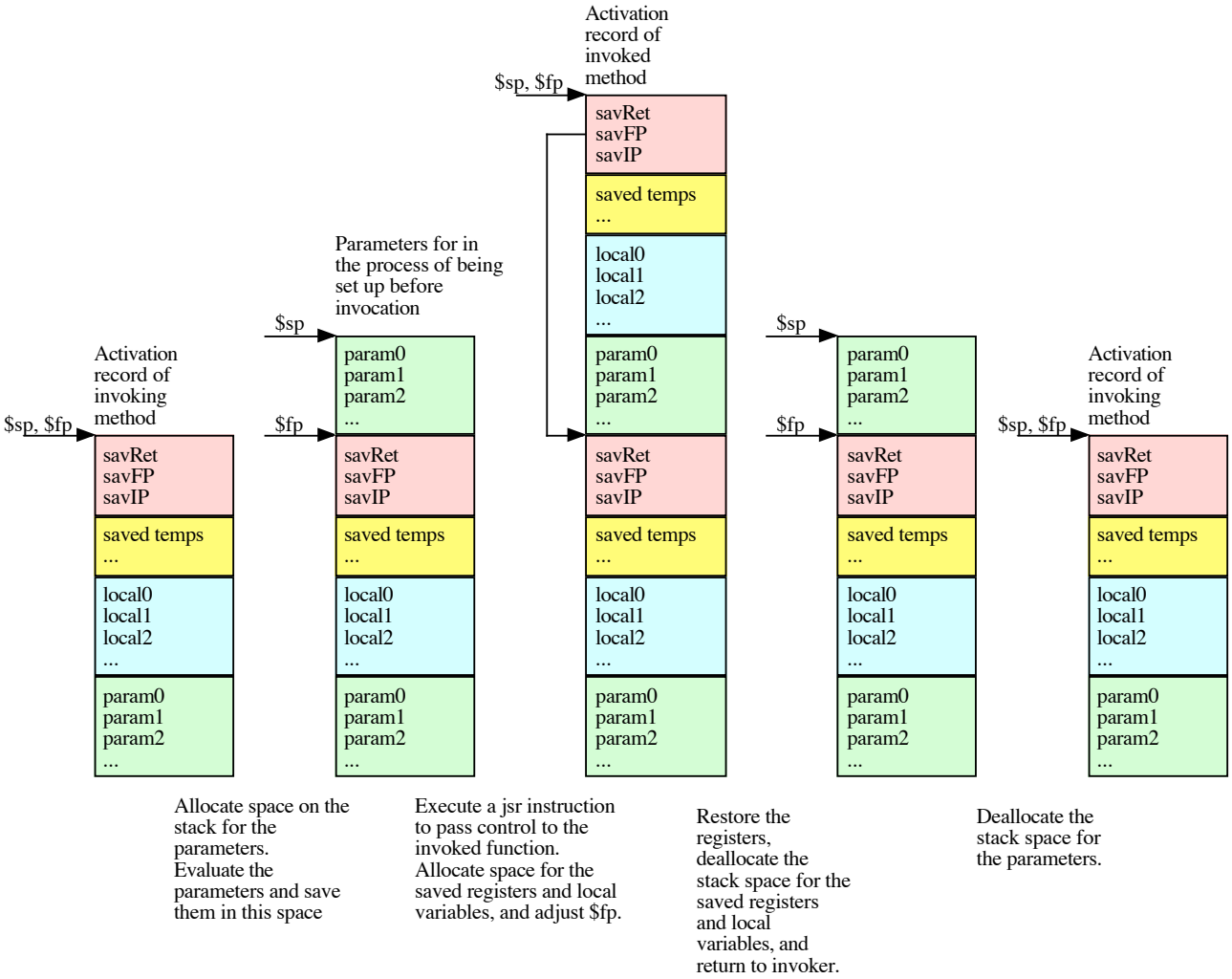
MethodDeclNode:

- Allocate space for the rest of the activation record, by subtracting the amount of space needed from $sp.
- Save the values of the $ra, $fp, and $ip registers on the stack.
- Set $fp = $sp; $ip = $nip;
- Execute the body of the method, leaving the return value in $t0.
- Restore the values of $ip, $fp, $ra from the stack.
- Deallocate space for the activation record (excluding the parameters) by adding the amount of space used to $sp.
- Execute a ret instruction to restore the program counter.

<——————————

InvocationNode:

- Deallocate space on the stack for the parameters of the invoked method, by adding the amount of space used to $sp.



For example, consider the program

```
int f( int formal1, formal2; )
     begin
          int local1, local2;
          local1 = formal1;
          local2 = formal2;
          return 1234;
     end
int actual1, actual2;
actual1 = 1;
actual2 = 2;
int result;
result = f( actual1, actual2 );
```

The invocation

```
result = f( actual1, actual2 );
```

translates into

```
            {
                lda $sp, -invoc.act2($sp);
                ldiq $t0, main.actual1_10;
                ldq $t0, ($t0);
                stq $t0, invoc.act0($sp);
                ldiq $t0, main.actual2_18;
                ldq $t0, ($t0);
                stq $t0, invoc.act1($sp);
                mov $zero, $nip;
                ldiq $pv, main.methodCode.f_0.enter;
                jsr;
                lda $sp, +invoc.act2($sp);
                ldiq $t1, main.result_20;
                stq $t0, ($t1);
            }
```

The declaration of f translates into

```
        public block f_0 extends proc.sav0 uses proc {
            code {
                align;
            public enter:
                lda $sp, -params($sp);
                stq $ra, savRet($sp);
                stq $fp, savFP($sp);
                stq $ip, savIP($sp);
                mov $nip, $ip;
                mov $sp, $fp;
                {
                    stq $zero, local1_0($fp);
                    stq $zero, local2_8($fp);
                }
                {
                    ldq $t0, formal1_10($fp);
                    stq $t0, local1_0($fp);
                }
                {
                    ldq $t0, formal2_18($fp);
                    stq $t0, local2_8($fp);
                }
                {
                    ldiq $t0, 0x4d2;
                    br return;
                }
                clr $t0;
            return:
                ldq $ip, savIP($sp);
                ldq $fp, savFP($sp);
                ldq $ra, savRet($sp);
                lda $sp, +params($sp);
                ret;
            } code
            local {
            public locals:
            public local1_0:
                quad;
                align;
            public local2_8:
                quad;
                align;
            public params:
```

```
        public formal1_10:
            quad;
            align;
        public formal2_18:
            quad;
            align;
        public max:
        } local
    } block f_0
```

Evaluation of Expressions

An expression can be evaluated using a stack, by the following recursive algorithm.

- Evaluate the operands, left to right, leaving their values on the stack.
- Pop the operands for the operator off the stack, perform the operation, and push the result back onto the stack.

An equivalent way of understanding the algorithm is to write the expression in reverse Polish, and process the components left to right.

- For a constant or variable, push the value of the constant or variable onto the stack.
- For an operator, pop the operands off the stack, compute the result, and push it onto the stack.

We can generate code to perform this algorithm. To make our code efficient, we can use the temporary registers to represent the stack.

For example, suppose we have the statement

```
result = a * b + c * d;
```

We could generate code

```
{
    ldq $t0, a_8($fp);
    ldq $t1, b_10($fp);
    mulq $t0, $t1, $t0;
    ldq $t1, c_18($fp);
    ldq $t2, d_20($fp);
    mulq $t1, $t2, $t1;
    addq $t0, $t1, $t0;
    stq $t0, result_0($fp);
}
```

Register Stack

Operation	$t0	$t1	$t2
a	a		
b	a	b	
*	a*b		
c	a*b	c	
d	a*b	c	d
*	a*b	c*d	
+	a*b+c*d		

However, there are problems with this algorithm.

- We could run out of registers. This used to be very important in the days when computers had only one or two registers ("accumulators") for evaluating expressions. However, this is unlikely to happen on a modern machine with lots of temporary registers.
- The expression might involve a method invocation. The trouble is that the method declaration does not know which registers the invoker is using, so cannot avoid conflicts. In fact the for recursive methods, the invoker and the invoked methods are the same method, so there will always be a conflict.

We could solve these problems by storing everything in memory, and using the stack pointer to indicate the top of stack. However, memory accesses are slower than register accesses, and modern computer architectures do not have special instructions to push and pop the stack.

For efficiency reasons, we store as much as possible of the top portion of the stack in registers. If we run out of registers, we save the bottom portion of the stack in memory. (The top portion of the stack is the part that is changing most frequently, so it is more important to keep it in registers.)

If we invoke a method, we know nothing about how it will use the temporary registers, so we assume it needs to use all temporary registers. Thus if invoking a method, the existing expression stack needs to be saved in memory. Our conventions mean the return value always comes back in $t0.

There are two approaches to register allocation.

- "Pre-planned" register allocation:

  The traditional way of generating code is to perform two treewalking passes.

  The first "tree-weighting" pass works out the number of registers needed to compute each node of the tree (the "weight" of the node).

  The second "code generation" pass actually produces assembly language. It takes into account the complexity of the operands, and usually evaluates them in decreasing order of complexity (to decrease the total register usage). If there are two operands, and both require all registers, the first one is evaluated, and the result stored in memory, then the second one is evaluated, then the first operand is then reloaded into a register. Finally the operation is performed, storing the result in a register.

  Evaluation of expressions can have side effects, and languages such as Java require at least the appearance of left to right evaluation, so we might have to maintain the order of evaluation. Both the tree-weighting and code generation algorithms have to change slightly under these circumstances.

- "On-the-fly" register allocation:

  An alternative is to evaluate the expression left to right, using registers. If we run out of temporary registers, we save the least recently used temporary register into memory, and use that register to grow our stack.

In practice, the code generated by the two algorithms is not so different, especially since the only constructs likely to cause registers to be saved in memory are method invocations. The ordering of the instructions, and the register numbers for constructs can differ a bit. Users tend to write the more complex operand first, so left to right evaluation tends to be the norm, even when not required.

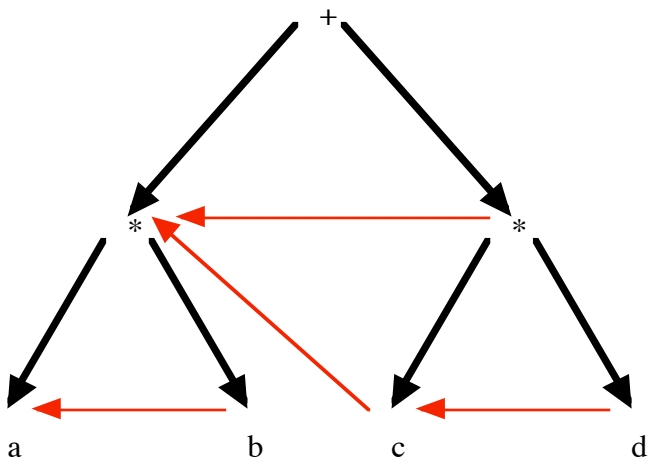
Having to specify both tree-weighting and code generation algorithms can cause problems, if the two are not consistent with each other, so from the software maintenance point of view, on-the-fly

register allocation is better. The support code for managing register allocation is more complex for on-the-fly register allocation, but it permits more flexibility. There are a few situations, such as the ++/-- operators, where the result may be returned in a different register from the one expected, with a purely stack-like use of registers.

The B-- compiler uses on-the-fly register allocation.

We need a way of specifying the conceptual stack of operands. To achieve this, we create a link from a node to its (usually left) sibling ("elder brother/sister"), or its ancestor's left sibling ("uncle/aunt") if the node itself has no left sibling. The nodes visited by chaining through the sibling links represent the conceptual stack of operands.

For example, consider the evaluation of "a * b + c * d". At the time we evaluate "d", our conceptual stack will contain the values of "a * b", "c", and "d". After we evaluate "c * d", our stack will contain "a * b", "c * d".



In each node, we have an indication of the addressing mode needed to access the value for that node. On the Alpha, this will correspond to a literal value, a temporary register, a displacement from a register, etc. We can scan through the sibling links, and look at the addressing mode information to determine which registers are currently in use.

Every expression has an evalCode() method that sets up the sibling link, and generates code to make its result addressable.

Consider a typical binary operator, such as multiplication.

First evalCode() generates code to make its operands addressable. Then it generates code to ensure its operands are in registers, obtains a free register for its result, and generates an instruction to perform the operation.

```
public abstract class Node {

     private Env env;
     protected Type type;

     private Node sibling;
     public Node sibling() { return sibling; }
     public void setSibling( Node sibling ) { this.sibling = sibling; }
...
     private Address address;
     public Address address() { return address; }
     public void setAddress( Address address ) { this.address = address; }

     private Usage usage( Usage usage ) {
          usage.plus( address );
```

```
          if ( sibling == null )
               return usage;
          else
               return sibling.usage( usage );
          }

     public Usage usage() {
          return usage( new Usage( env() ) );
          }
     ...
     }

public abstract class ExprNode extends Node {
     ...
     }

public abstract class BinaryNode extends ExprNode {
     ...
     public void evalCode( Node sibling ) {
          setSibling( sibling );
          left.evalCode( sibling );
          right.evalCode( left );
          left.loadReg( usage().plus( right.address() ) );
          right.loadReg( usage().plus( left.address() ) );
          setAddress( obtainReg( type, usage() ) );
          }
     }

public abstract class ArithNode extends BinaryNode {
     ...
     }

public class TimesNode extends ArithNode {
     ...
     public void evalCode( Node sibling ) {
          super.evalCode( sibling );
          Code.instrn( opCode, left.address(), right.address(), address() );
          }
     }
```

Now we need to look a bit more closely at what is going on, especially if we run out of registers.

The method usage() chains down the sibling links, looking at the addressing modes of operands to determine the current usage of registers.

The method obtainReg() tries to obtain a free register. If none is free, it generates code to save the least recently used register in memory, and changes the addressing mode for that node.

If the left operand is already addressable, then

```
          left.evalCode( sibling );
```

does not generate any code.

However, if the left operand is complex, it will generate code to load its values into a register.

Then

```
          right.evalCode( left );
```

will be executed.

Suppose the right operand is sufficiently complex so that it requires the use of all the registers, then an invocation of obtainReg() (somewhere inside right.evalCode()) will generate code to save the value of the left operand in memory, and alter the addressing mode for the left operand.

Thus by the time right.evalCode() returns, the left operand might no longer be in a register.

The line

```
left.loadReg( usage().plus( right.address() ) );
```

will ensure that the left operand is if necessary (re)loaded into a register. The parameter indicates the registers currently in use, namely those of stacked values, and any register used by the right operand.

The line

```
right.loadReg( usage().plus( left.address() ) );
```

will ensure that the right operand is if necessary (re)loaded into a register. The registers currently in use are those of stacked values, and any register used by the left operand.

The line

```
setAddress( obtainReg( type, usage() ) );
```

obtain a free register for the result (which may overlap the ones used for the operands), and records it in the node.

Finally,

```
Code.instrn( opCode, left.address(), right.address(), address() );
```

generates the instruction to perform the multiply.

Addition is a bit more complex, when dealing with pointer types. However, this is hidden in the code for performing a scaled add. If necessary, a s8addq instruction, or an extra multiplication is generated.

```
public void evalCode( Node sibling ) {
     super.evalCode( sibling );
     Code.scaledAdd( type, left.address(), right.address(), address() );
     }
```

The line

```
return comb( n - 1, r - 1 ) + comb( n - 1, r );
```

in

```
int comb( int n, r; )
     begin
          if r < 0 || r > n then
               return 0;
          elif r == 0 || r == n then
               return 1;
          else
               return comb( n - 1, r - 1 ) + comb( n - 1, r );
          end
     end
```

causes an operand to be saved on the stack.

It translates into

```
                              {
                                   lda $sp, -invoc.act2($sp);
                                   ldq $t0, n_0($fp);
                                   ldiq $t1, 0x1;
                                   subq $t0, $t1, $t0;
                                   stq $t0, invoc.act0($sp);
                                   ldq $t0, r_8($fp);
                                   ldiq $t1, 0x1;
                                   subq $t0, $t1, $t0;
                                   stq $t0, invoc.act1($sp);
                                   mov $zero, $nip;
```

```
                                ldiq $pv, main.methodCode.comb_0.enter;
                                jsr;
                                lda $sp, +invoc.act2($sp);
                                stq $t0, sav0($fp);
                                lda $sp, -invoc.act2($sp);
                                ldq $t0, n_0($fp);
                                ldiq $t1, 0x1;
                                subq $t0, $t1, $t0;
                                stq $t0, invoc.act0($sp);
                                ldq $t0, r_8($fp);
                                stq $t0, invoc.act1($sp);
                                mov $zero, $nip;
                                ldiq $pv, main.methodCode.comb_0.enter;
                                jsr;
                                lda $sp, +invoc.act2($sp);
                                ldq $t1, sav0($fp);
                                addq $t1, $t0, $t0;
                                br return;
                        }
```

Assignment is similar. But in this case, we get slightly better code by evaluating the right operand before the left (the right operand tends to be the more complex than the left, and there are complications if we attempt to save an address in memory, then reload it later).

```
     public void evalCode( Node sibling ) {
          setSibling( sibling );
          expr.evalCode( sibling );
          variable.evalAddrCode( expr );
          expr.loadReg( usage().plus( variable.address() ) );
          variable.loadAddress( usage().plus( expr.address() ) );
          setAddress( null );
          Code.store( varType, ( Register ) expr.address(), variable.address() );
          }
```

Preincrement and predecrement are a little more tricky. They make the operand addressable, load the value into a register, increment/decrement it, and store it back. The adjusted value represents the result.

```
public abstract class PrefixVarNode extends ExprNode {
     ...
     public void evalCode( Node sibling ) {
          setSibling( sibling );
          right.evalAddrCode( sibling );
          }
     }
public class PreIncNode extends PrefixVarNode {
     ...
     public void evalCode( Node sibling ) {
          super.evalCode( sibling );
          Register valueReg = obtainReg( type, usage().plus( address() ) );
          Address address = right.loadAddress( usage().plus( valueReg ) );
          Code.load( type, valueReg, address );
          Code.inc( type, valueReg, valueReg );
          Code.store( type, valueReg, address );
          setAddress( valueReg );
          }

     }
```

Postincrement and postdecrement are even more tricky. They make the operand addressable, load the value into a register, increment/decrement it, storing the result in a different register, and store the result back. The initial value represents the result.

```
public abstract class PostfixVarNode extends ExprNode {
     ...
     public void evalCode( Node sibling ) {
          setSibling( sibling );
          left.evalAddrCode( sibling );
          }
     }
public class PostIncNode extends PostfixVarNode {
     ...
     public void evalCode( Node sibling ) {
          super.evalCode( sibling );
          Address address = left.loadAddress( usage() );
          Register preValueReg = obtainReg( type, usage().plus( address ) );
          Register postValueReg = obtainReg( type,
               usage().plus( address ).plus( preValueReg ) );
          Code.load( type, preValueReg, address );
          Code.inc( type, preValueReg, postValueReg );
          Code.store( type, postValueReg, address );
          setAddress( preValueReg );
          }
     }
```

Accessing Global Variables and Methods

Global variables and methods are accessed via symbolic names, not as an offset from a register.

For example, consider the program

```
void increment( ^int dest; int change; )
     begin
          dest^ = dest^ + change;
     end

int result = 4;
increment( &result, 3 );
```

The declaration

```
int result = 4;
```

translates into

```
                {
                    ldiq $t0, 0x4;
                    ldiq $at, main.result_10;
                    stq $t0, ($at);
                }
```

The invocation

```
increment( &result, 3 );
```

translates into

```
                {
                    lda $sp, -invoc.act2($sp);
                    ldiq $t0, main.result_10;
                    stq $t0, invoc.act0($sp);
                    ldiq $t0, 0x3;
                    stq $t0, invoc.act1($sp);
                    mov $zero, $nip;
                    ldiq $pv, main.methodCode.increment_0.enter;
                    jsr;
                    lda $sp, +invoc.act2($sp);
                }
```

Note the fact that the address of result is passed as a parameter, not its value.

Accessing Local Variables

Local variables and formal parameters are accessed as an offset from $fp. Var parameters require an additional level of indirection to access the value. Because this is a C-like language, the indirection is explicit in the high-level language, but in a language such as Pascal, it would be implicit.

The declaration of increment above translates into

```
public block increment_0 extends proc.sav0 uses proc {
    code {
        align;
    public enter:
        lda $sp, -params($sp);
        stq $ra, savRet($sp);
        stq $fp, savFP($sp);
        stq $ip, savIP($sp);
        mov $nip, $ip;
        mov $sp, $fp;
        {
            ldq $t0, dest_0($fp);
            ldq $t0, ($t0);
            ldq $t1, change_8($fp);
            addq $t0, $t1, $t0;
            ldq $t1, dest_0($fp);
            stq $t0, ($t1);
        }
        clr $t0;
    return:
        ldq $ip, savIP($sp);
        ldq $fp, savFP($sp);
        ldq $ra, savRet($sp);
        lda $sp, +params($sp);
        ret;
    } code
    local {
    public locals:
    public params:
    public dest_0:
        quad;
        align;
    public change_8:
        quad;
        align;
    public max:
    } local
} block increment_0
```

Note the additional indirection needed to access the value via the var parameter.

Accessing Fields

Fields of the current instance are accessed via an offset from $ip.

Consider the program

```
class C
     begin
          int value;
          void increment( int change; )
               begin
                    value = value + change;
               end
     end

C c;
c.value = 4;
c.increment( 3 );
```

The class declaration translates into

```
    public block C_0 {
        // Field table layout for main.C_0
        public block field {
            local {
            public methodTablePtr:
                quad;
            public value_8:
                quad;
                align;
            public max:
            } local
        } block field
        // Method table layout for main.C_0
        public block method {
            local {
            public increment_0:
                quad;
                align;
            public max:
            } local
        } block method
        // Actual method table for main.C_0
        public block methodTable {
            const {
            public enter:
                quad main.C_0.methodCode.increment_0.enter;
            } const
        } block methodTable
        // Code for methods for main.C_0
        public block methodCode {
            public block increment_0 extends proc.sav0 uses proc {
                code {
                    align;
                public enter:
                    lda $sp, -params($sp);
                    stq $ra, savRet($sp);
                    stq $fp, savFP($sp);
                    stq $ip, savIP($sp);
                    mov $nip, $ip;
                    mov $sp, $fp;
                    {
```

```
                        ldq $t0, main.C_0.field.value_8($ip);
                        ldq $t1, change_0($fp);
                        addq $t0, $t1, $t0;
                        stq $t0, main.C_0.field.value_8($ip);
                    }
                    clr $t0;
                return:
                    ldq $ip, savIP($sp);
                    ldq $fp, savFP($sp);
                    ldq $ra, savRet($sp);
                    lda $sp, +params($sp);
                    ret;
                } code
                local {
                public locals:
                public params:
                public change_0:
                    quad;
                    align;
                public max:
                } local
            } block increment_0
        } block methodCode
} block C_0
```

For fields accessed via member selection, the address of the object has to be loaded into a register, and the field is accessed as an offset from this register.

The assignment statement

```
c.value = 4;
```

translates into

```
                {
                    ldiq $t0, main.c_10;
                    ldiq $t1, 0x4;
                    stq $t1, main.C_0.field.value_8($t0);
                }
```

For invocations of instance methods, $nip must be set up to point to the new instance, and the method obtained from the method table.

The invocation

```
c.increment( 3 );
```

translates into

```
               {
                    lda $sp, -invoc.act1($sp);
                    ldiq $t0, 0x3;
                    stq $t0, invoc.act0($sp);
                    ldiq $t0, main.c_10;
                    mov $t0, $nip;
                    ldq $at, main.C_0.field.methodTablePtr($nip);
                    ldq $pv, main.C_0.method.increment_0($at);
                    jsr;
                    lda $sp, +invoc.act1($sp);
                }
```

Generating and Accessing the Method Table

Consider the program

```
class A
     begin
          int field1, field2;
          void aInstance1()
               begin
                    printf( "aInstance1\n" );
               end
          void override1()
               begin
                    printf( "A override1\n" );
               end
          void override2()
               begin
                    printf( "A override2\n" );
               end
          void aInstance2()
               begin
                    printf( "aInstance2\n" );
               end
     end

class B extends A
     begin
          int field3, field4;
          void bInstance1()
               begin
                    printf( "bInstance1\n" );
               end
          void override1()
               begin
                    printf( "B override1\n" );
               end
          void bInstance2()
               begin
                    printf( "bInstance2\n" );
               end
          void override2()
               begin
                    printf( "B override2\n" );
               end
     end

B b;
^A a = b;
a.aInstance1();
a.override1();
a.override2();
a.aInstance2();
```

When executed, we generate the output

```
aInstance1
B override1
B override2
aInstance2
```

The compiler generates code

```
    public block A_0 {
        // Field table layout for main.A_0
        public block field {
            local {
            public methodTablePtr:
                quad;
            public field1_8:
                quad;
                align;
            public field2_10:
                quad;
                align;
            public max:
            } local
        } block field
        // Method table layout for main.A_0
        public block method {
            local {
            public aInstance1_0:
                quad;
                align;
            public override1_8:
                quad;
                align;
            public override2_10:
                quad;
                align;
            public aInstance2_18:
                quad;
                align;
            public max:
            } local
        } block method
        // Actual method table for main.A_0
        public block methodTable {
            const {
            public enter:
                quad main.A_0.methodCode.aInstance1_0.enter;
                quad main.A_0.methodCode.override1_8.enter;
                quad main.A_0.methodCode.override2_10.enter;
                quad main.A_0.methodCode.aInstance2_18.enter;
            } const
        } block methodTable
        // Code for methods for main.A_0
        public block methodCode {
            ...
        } block methodCode
        // Code to initialise an instance of class A_0
        public block initInstance extends proc.sav0 uses proc {
            ...
        } block initInstance
    } block A_0
    public block B_0 {
        // Field table layout for main.B_0
        public block field extends main.A_0.field uses main.A_0.field {
            local {
            public field3_18:
                quad;
                align;
```

```
            public field4_20:
                quad;
                align;
            public max:
            } local
        } block field
        // Method table layout for main.B_0
        public block method extends main.A_0.method uses main.A_0.method {
            local {
            public bInstance1_20:
                quad;
                align;
            public bInstance2_28:
                quad;
                align;
            public max:
            } local
        } block method
// Actual method table for main.B_0
        public block methodTable {
            const {
            public enter:
                quad main.A_0.methodCode.aInstance1_0.enter;
                quad main.B_0.methodCode.override1_8.enter;
                quad main.B_0.methodCode.override2_10.enter;
                quad main.A_0.methodCode.aInstance2_18.enter;
                quad main.B_0.methodCode.bInstance1_20.enter;
                quad main.B_0.methodCode.bInstance2_28.enter;
            } const
        } block methodTable
        // Code for methods for main.B_0
        public block methodCode {
            ...
        } block methodCode
        // Code to initialise an instance of class B_0
        public block initInstance extends proc.sav0 uses proc {
            code {
                align;
            public enter:
                lda $sp, -max($sp);
                stq $ra, savRet($sp);
                stq $fp, savFP($sp);
                stq $ip, savIP($sp);
                mov $nip, $ip;
                mov $sp, $fp;
                bsr main.A_0.initInstance.enter;
                {
                    lda $nip, main.B_0.field.field3_18($ip);
                    stq $zero, ($nip);
                    lda $nip, main.B_0.field.field4_20($ip);
                    stq $zero, ($nip);
                }
            return:
                mov $ip, $nip;
                ldq $ip, savIP($sp);
                ldq $fp, savFP($sp);
                ldq $ra, savRet($sp);
                lda $sp, +max($sp);
                ret;
            } code
            local {
```

```
            public max:
            } local
        } block initInstance

} block B_0
```

Note how the method table for B includes all the entries for A, in the same order, but the overridden entries have been replaced by methods declared in B.

The code to allocate space for the global variables is

```
    // Global variable declarations for main
    data {
    public true_0:
        quad;
        align;
    public false_8:
        quad;
        align;
    public b_10:
        space 0x28;
        align;
    public a_38:
        quad;
        align;
    } data
```

Note that the space for “b” is the space needed for the fields, while the space for “a” is just the space for a pointer.

Variables need initialisation code. For class instances we need to set up the pointer to the method table, then execute the initialisation code.

```
                {
                    ldiq $t0, main.b_10;
                    lda $nip, ($t0);
                    ldiq $at, main.B_0.methodTable.enter;
                    stq $at, ($nip);
                    bsr main.B_0.initInstance.enter;
                }
                {
                    ldiq $t0, main.b_10;
                    ldiq $at, main.a_38;
                    stq $t0, ($at);
                }
```

When we invoke an instance method, we first determine the address of the instance, and store it in $nip, then determine the address of the method by looking up the method table.

For example, for the invocation

```
a.override1();
```

we generate code

```
                {
                    lda $sp, -invoc.act0($sp);
                    ldiq $t0, main.a_38;
                    ldq $t0, ($t0);
                    mov $t0, $nip;
                    ldq $at, main.A_0.field.methodTablePtr($nip);
                    ldq $pv, main.A_0.method.override1_8($at);
                    jsr;
                    lda $sp, +invoc.act0($sp);
                }
```

So it uses the offset from the declared type (A) to access the method table. This offset is of course same as that from the actual type (B).

Initialisation of variables

Variables need to be initialised. This is especially true of class instance variables. It is a matter of courtesy to the programmer, to initialise ordinary variables to their default values (essentially always 0, although in the interpreter, might be 0, '\0', or null). But for class instance variables, the method table has to be set up, and the code for the body of the method executed. For arrays, the elements need to be initialised. Again, this is important for arrays of class instances.

Initialisation for variables starts in UninitDeclaratorNode.

```
public class UninitDeclaratorNode extends DeclaratorNode {
     ...
     public void genCode() {
          switch ( decl.env().envKind() ) {
               case Env.ENV_GLOBAL:
               case Env.ENV_METHOD:
               case Env.ENV_CLASS:
                    setAddress( decl.address() );
                    Displacement varAddress = loadAddress( new Usage( env() ) );
                    Code.loadA( SpecialReg.newInstPtr, varAddress );
                    decl.type().initAddressCode();
                    break;
               default:
                    throw new Error(
                         "Invalid environment of "
                         + Env.envKindText( decl.env().envKind() )
                         + " for identifier " + decl.ident() );
               }
          }
     }
```

For primitive types, and pointer types, this generates a store instruction.

```
public abstract class Type {
     ...
     public void initAddressCode() {
          Code.store( this, SpecialReg.zero,
               new Indirect( SpecialReg.newInstPtr ) );
          }
     }
```

For class instance types, we have to set up the method table, then invoke code to perform the body of the class.

```
public  class ClassInstanceType extends Type {
     ...
     public void initAddressCode() {
          Code.loadImm( IntType.type, SpecialReg.assemTemp,
               env.absoluteName() + ".methodTable.enter" );
          Code.store( IntType.type, SpecialReg.assemTemp,
               new Indirect( SpecialReg.newInstPtr ) );
          Code.bsr( new Relative( env.absoluteName() + ".initInstance.enter" ) );
          }
     }
public class ClassDeclNode extends DeclStmtNode implements DeclNode {
     ...
     public void genInstanceCode() {
          Code.comment(
               "Code to initialise an instance of class " + decl.useName() );
          Code.enterBlock( "initInstance", "proc.sav0", "proc" );
```

```
        Code.enter( "code" );
        Code.align();
        Code.publicLabelDefn( "enter" );
        Code.loadA( SpecialReg.stackPtr,
             new Displacement( "-max", SpecialReg.stackPtr ) );
        Code.storeQ( SpecialReg.retAddr,
             new Displacement( "savRet", SpecialReg.stackPtr ) );
        Code.storeQ( SpecialReg.framePtr,
             new Displacement( "savFP", SpecialReg.stackPtr ) );
        Code.storeQ( SpecialReg.instPtr,
             new Displacement( "savIP", SpecialReg.stackPtr ) );
        Code.move( SpecialReg.newInstPtr, SpecialReg.instPtr );
        Code.move( SpecialReg.stackPtr, SpecialReg.framePtr );
        if ( extendedDecl != null )
             Code.bsr( new Relative(
                  extendedDecl.absoluteName() + ".initInstance.enter" ) );
        body.genCode();
        Code.labelDefn( "return" );
        Code.move( SpecialReg.instPtr, SpecialReg.newInstPtr );
        Code.loadQ( SpecialReg.instPtr,
             new Displacement( "savIP", SpecialReg.stackPtr ) );
        Code.loadQ( SpecialReg.framePtr,
             new Displacement( "savFP", SpecialReg.stackPtr ) );
        Code.loadQ( SpecialReg.retAddr,
             new Displacement( "savRet", SpecialReg.stackPtr ) );
        Code.loadA( SpecialReg.stackPtr,
             new Displacement( "+max", SpecialReg.stackPtr ) );
        Code.ret();
        Code.exit( "code" );
        localEnv.genTempSpace();
        Code.exitBlock( "initInstance" );
        }

    public void genDeclCode() {
        Code.enterBlock( decl.declName(), null, null );
        localEnv.genFieldDeclOffsetCode();
        localEnv.genMethodDeclOffsetCode();
        localEnv.genMethodTable();
        localEnv.genMethodDeclCode();
        genInstanceCode();
        Code.exitBlock( decl.declName() );
        }

    public void genCode() {
        }
    }
```

For arrays, we have to generate a loop, to initialise the elements.

```
public  class ArrayType extends Type {
    ...
    public Type primitiveType() {
        return subType.primitiveType();
        }

    public int numPrimitiveElements() {
        return numElements * subType.numPrimitiveElements();
        }

    public void initAddressCode() {
        Code.enter();
        Code.loadA( SpecialReg.stackPtr,
             new Displacement( "-8", SpecialReg.stackPtr ) );
```

```
        Code.storeQ( SpecialReg.s0, new Indirect( SpecialReg.stackPtr ) );
        Code.labelDefn( "for" );
        Code.loadImm( IntType.type, SpecialReg.s0, numPrimitiveElements() );
        Code.labelDefn( "while" );
        Code.instrn( "ble", SpecialReg.s0, new Relative( "end" ) );
        primitiveType().initAddressCode();
        Code.instrn( "addq", SpecialReg.newInstPtr,
             new Literal( primitiveType().size().compile() ) );
        Code.instrn( "subq", SpecialReg.s0, new Literal( "1" ) );
        Code.instrn( "br", new Relative( "while" ) );
        Code.labelDefn( "end" );
        Code.loadQ( SpecialReg.s0, new Indirect( SpecialReg.stackPtr ) );
        Code.loadA( SpecialReg.stackPtr,
             new Displacement( "+8", SpecialReg.stackPtr ) );
        Code.exit();
        }
    }
```

Control Statements and Boolean Expressions

Generation of code for control statements is actually quite easy. It is little more than code for the sub-constructs, together with a few labels and branch statements.

For example, the following generates code for a for statement

```
    public void genCode() {
        Code.enter();
        Code.labelDefn( "for" );
        initial.evalCode( null );
        Code.labelDefn( "while" );
        cond.boolCode( null, new Relative( "do" ), new Relative( "end" ),
             true );
        Code.labelDefn( "do" );
        stmtList.genCode();
        Code.labelDefn( "continue" );
        increment.evalCode( null );
        Code.br( new Relative( "while" ) );
        Code.labelDefn( "break" );
        Code.labelDefn( "end" );
        Code.exit();
        }
```

For example, if we have the program

```
int i, sum = 0;
for i = 0; i < 10; i++
do
    sum = sum + i;
end
```

We generate the code

```
                {
                 for:
                     ldiq $t0, 0x0;
                     ldiq $t1, main.i_10;
                     stq $t0, ($t1);
                 while:
                     ldiq $t0, main.i_10;
                     ldq $t0, ($t0);
                     ldiq $t1, 0xa;
                     cmplt $t0, $t1, $t0;
                     blbc $t0, end;
                 do:
                     {
```

```
                ldiq $t0, main.sum_18;
                ldiq $t1, main.i_10;
                ldq $t0, ($t0);
                ldq $t1, ($t1);
                addq $t0, $t1, $t0;
                ldiq $t1, main.sum_18;
                stq $t0, ($t1);
            }
        continue:
            ldiq $t0, main.i_10;
            ldq $t1, ($t0);
            addq $t1, 0x1, $t2;
            stq $t2, ($t0);
            br while;
        break:
        end:
        }
```

The invocations of enter() and exit() just enclose the code in a local block {...} in the assembly language.

We generate symbolic labels “for”, “while”, “do”, “continue”, “break” and “end”, and recursively perform code generation for the sub-constructs. The code for the increment needs to go at the end of the loop, and is followed by a branch back to the condition.

The method genCode() for an if statement is similar.

```
public void genCode() {
    Code.enter();
    Code.labelDefn( "if" );
    cond.boolCode( null, new Relative( "then" ), new Relative( "else" ),
        true );
    Code.labelDefn( "then" );
    thenPart.genCode();
    Code.br( new Relative( "end" ) );
    Code.labelDefn( "else" );
    elsePart.genCode();
    Code.labelDefn( "end" );
    Code.exit();
    }
```

Control statements all involve generating code for a Boolean expression.

This is done by a method

```
public void boolCode(
    Node sibling,
    Relative trueLabel,
    Relative falseLabel,
    boolean flowIntoTrue );
```

Rather than evaluating the condition, it generates code that causes control to pass to one of two labels (trueLabel or falseLabel). One of these labels always follows the construct, and the flag flowIntoTrue indicates which one follows.

For Boolean expressions involving “||” or “&&”, it might not be necessary to evaluate both operands. For example, if A is true, then A || B must be true.

to generate code for “||”

```
public void boolCode(
    Node sibling,
    Relative trueLabel,
    Relative falseLabel,
    boolean flowIntoTrue ) {
```

```
        setSibling( sibling );
        Relative secondPart = Label.newLabel();
        Code.enter();
        left.boolCode( sibling, trueLabel, secondPart, false );
        Code.labelDefn( secondPart );
        right.boolCode( sibling, trueLabel, falseLabel, flowIntoTrue );
        Code.exit();
        }
```

What does it mean? If the left operand is true, we do not need to evaluate the right operand. We branch straight to the trueLabel. If the left operand is false, we need to evaluate the right operand. Moreover, the code for evaluating the left operand flows into the code for the right operand, and the code for the right operand is executed when the left operand is false.

The real work is done within relational operators.

For example, for the < operator

```
public abstract class RelationNode extends BinaryNode {
    ...
    public void evalCode( Node sibling ) {
        super.evalCode( sibling );
        Code.instrn( opCode, left.address(), right.address(), address() );
        }
    }
public class LessThanNode extends RelationNode {
    ...
    public void boolCode(
        Node sibling,
        Relative trueLabel,
        Relative falseLabel,
        boolean flowIntoTrue ) {
        super.evalCode( sibling );
        if ( flowIntoTrue )
            Code.instrn( "blbc", address(), falseLabel );
        else
            Code.instrn( "blbs", address(), trueLabel );
        }
    }
```