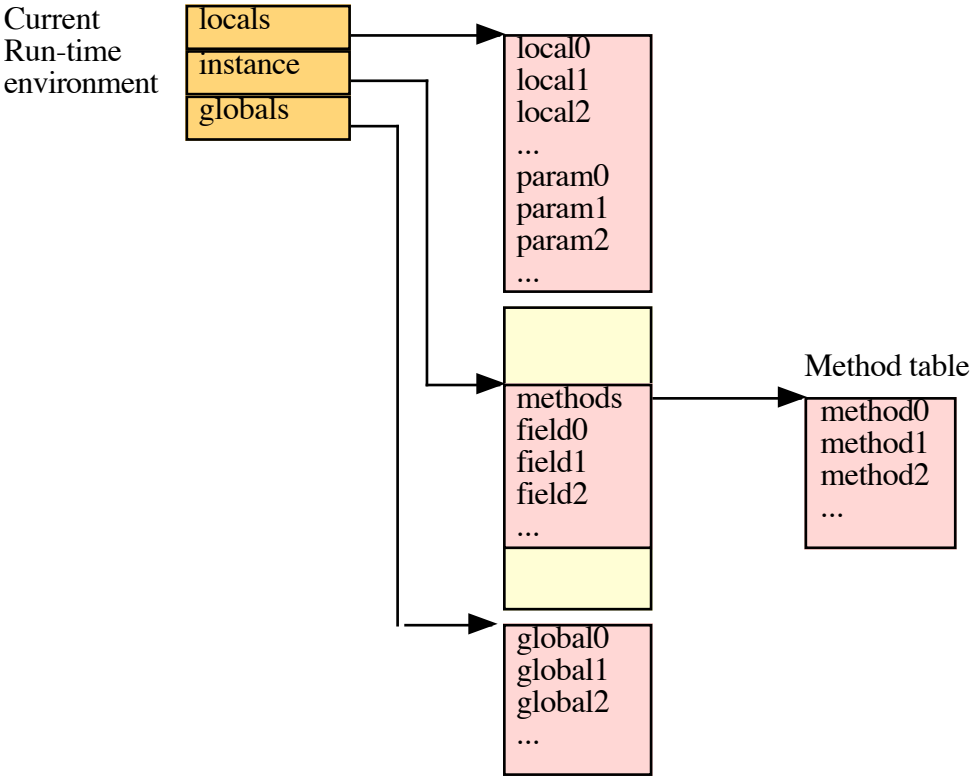
Chapter 15 Method Declarations and Invocations

Method Invocations

The layout for an activation record for a method is

- (In the compiler's model, but not the interpreter's model of an activation record, there is space for the saving the values of registers).
- The space for local variables, in order.
- The space for the parameters, in order.



To type check an invocation:

- Type check the method name and the actual parameters.
- Cast the actual parameters to the types of the formal parameters.
- The type of the invocation is the return type of the method.

To evaluate an invocation (in the interpreter):

- Evaluate the method name to obtain a new instance and the address of the method to invoke.
- Allocate space for the new activation record, evaluate the actual parameters, and store them in this activation record.
- Create a new run-time environment, with the new activation record, new instance, and existing globals.
- Evaluate the body of the method in the new run-time environment.
- To return the result, the evaluation of the body throws an exception containing the return value, that the evaluation code for the invocation catches. The invocation returns the value.

```
public class InvocationNode extends PrimaryNode {

     private MethodNameNode methodName;
     private ExprListNode actualParamList;
     private MethodType methodType;

     public InvocationNode(
          MethodNameNode methodName,
          ExprListNode actualParamList ) {
          this.methodName = methodName;
          this.actualParamList = actualParamList;
          }

     public String toString() {
          if ( actualParamList.size() == 0 )
               return methodName + "()";
          else
               return methodName + "( " + actualParamList + " )";
          }

     public void genEnv( Env env ) {
          super.genEnv( env );
          methodName.genEnv( env );
          actualParamList.genEnv( env );
          }

     public Type checkType() {
          methodType = ( MethodType ) methodName.checkType();
          actualParamList.checkType();
          actualParamList.castParams( methodType );
          type = methodType.returnType();
          return type;
          }

     public RunValue eval( RunEnv runEnv ) {
          ObjectMethodPair objectMethodPair = methodName.eval( runEnv );
          PtrValue instance = objectMethodPair.instanceValue();
          Decl methodDecl = objectMethodPair.methodDecl();
          MethodDeclNode methodDeclNode =
               ( MethodDeclNode ) methodDecl.declNode();
          Env localEnv = methodDeclNode.localEnv();
          int frameSize =
               localEnv.paramStart().interp() + actualParamList.size();
          Multiple locals = new Multiple( "" + methodName, frameSize );
          actualParamList.eval( runEnv,
               new PtrValue( locals, frameSize - actualParamList.size() ) );
          try {
               RunEnv invocationRunEnv
                    = new RunEnv( runEnv.globals(),
                         instance, new PtrValue( locals ) );
               DeclStmtListNode body = methodDeclNode.body();
               if ( body == null )
                    BuiltIn.eval( methodDeclNode.ident(), invocationRunEnv );
               else
                    methodDeclNode.body().eval( invocationRunEnv );
               }
          catch ( ReturnException exception ) {
               return exception.returnValue();
               }
          return VoidValue.value;
          }
```

```
    public void evalCode( OperandNode sibling ) {
         setSibling( sibling );
         saveAllReg( usage() );
         Code.loadA( SpecialReg.stackPtr,
              new Displacement( "-invoc.act" + actualParamList.size(),
                   SpecialReg.stackPtr ) );
         actualParamList.evalCode( sibling );
         methodName.evalCode( sibling );
         Code.jsr();
         Code.loadA( SpecialReg.stackPtr,
              new Displacement( "+invoc.act" + actualParamList.size(),
                   SpecialReg.stackPtr ) );
         setAddress( obtainReg( type, usage() ) );
         }
    }
```

Evaluation of the Method Name

There are two kinds of method name - a simple identifier, or member selection.

Evaluation of a method name returns a new instance, together with the method to be invoked.

```
public abstract class MethodNameNode extends OperandNode {
     protected Decl decl;
     public Decl decl() { return decl; }

     public abstract Type checkType();
     public abstract ObjectMethodPair eval( RunEnv runEnv );
     public abstract void evalCode( OperandNode sibling );
     }
```

For a simple identifier:

- To type check the method name, look up the identifier in the current environment, to obtain a method declaration, and return the type of the method.
- If the method is a global method, the new instance is null, and the method is the method obtained from the current environment.
- If the method is an instance method, the new instance is the current instance, and the method is obtained by using the offset of the declared method to look up the method table of the current instance. Because the current instance might be a subclass of its declared type, the method might be a method declared in the subclass.

```
public class IdentMethodNameNode extends MethodNameNode {

     private String ident;

     public IdentMethodNameNode( String ident ) {
          this.ident = ident;
          }

     public String toString() {
          return ident;
          }

     public Type checkType() {
          decl = env().searchEnv( ident );
          if ( decl == null )
               throw new Error( "Undeclared method " + ident );
          switch ( decl.section() ) {
               case Decl.DECL_METHOD:
```

```
                type = ( MethodType ) decl.type();
                return type;
            default:
                throw new Error( ident + " must be a method" );
            }
        }
```

```
    public ObjectMethodPair eval( RunEnv runEnv ) {
        // Return an ObjectMethodPair representing
        // the instance and code for the method
        switch ( decl.env().envKind() ) {
            case Env.ENV_CLASS:
                {
                PtrValue instanceValue = runEnv.instance();
                DeclList methodTable =
                    instanceValue.getValue().methodTable();
                return new ObjectMethodPair( instanceValue,
                    methodTable.elementAt( decl.offset().interp() ) );
                }
            case Env.ENV_GLOBAL:
                {
                return new ObjectMethodPair( null, decl );
                }
            default:
                throw new Error(
                    "Invalid section of "
                    + Decl.sectionText( decl.section() ) );
            }
        }
```

```
    public Register saveReg( int dataType, Usage usage ) {
        return null;
        }
```

```
    public void evalCode( Node sibling ) {
        setSibling( sibling );
        // Loads $nip and $pv with appropriate values.
        switch ( decl.env().envKind() ) {
            case Env.ENV_CLASS:
                Code.move( SpecialReg.instPtr, SpecialReg.newInstPtr );
                Code.load( IntType.type, SpecialReg.assemTemp,
                    new Displacement(
                        decl.env().absoluteName() + ".field.methodTablePtr",
                        SpecialReg.newInstPtr ) );
                Code.load( IntType.type, SpecialReg.procValue,
                    new Displacement(
                        decl.absoluteName(), SpecialReg.assemTemp ) );
                break;
            case Env.ENV_GLOBAL:
                Code.move( SpecialReg.zero, SpecialReg.newInstPtr );
                Code.loadImm( IntType.type, SpecialReg.procValue,
                    decl.absoluteName() + ".enter" );
                break;
            default:
                throw new Error(
                    "Invalid section of " +
                        Decl.sectionText( decl.section() ) );
            }
```

For a member selection:

- To type check the method name, type check the object, and make sure it has type ^classType, look up the identifier in the environment of the classType, to obtain a method declaration, and return the type of the method.
- The new instance is the object, and the method is obtained by using the offset of the declared method to look up the method table of the new instance. Because the new instance might be a subclass of its declared type, the method might be a method declared in the subclass.

```
public class MemberMethodNameNode extends MethodNameNode {

     private ExprNode object;
     private Type objectType;
     private String ident;

     public MemberMethodNameNode( ExprNode object, String ident ) {
          this.object = object;
          this.ident = ident;
          }

     public String toString() {
          return object + "." + ident;
          }

     public void genEnv( Env env ) {
          super.genEnv( env );
          object.genEnv( env );
          }

     public Type checkType() {
          Type oType = object.checkType();
          if ( ! ( oType instanceof PtrType ) )
               throw new Error( "Selection must be from a reference" );
          PtrType ptrType = ( PtrType ) oType;
          Type sType = ptrType.subType();
          if ( ! ( sType instanceof ClassInstanceType ) )
               throw new Error(
          "Selection must be from a reference to an instance of class type" );
          ClassInstanceType instanceType = ( ClassInstanceType ) sType;
          decl = instanceType.env().searchExtended( ident );
          if ( decl == null )
               throw new Error( "Undeclared method " + ident );
          if ( decl.section() != Decl.DECL_METHOD )
               throw new Error( ident + " must be method" );
          type = ( MethodType ) decl.type();
          return type;
          }

     public ObjectMethodPair eval( RunEnv runEnv ) {
          switch ( decl.env().envKind() ) {
               case Env.ENV_CLASS:
                    {
                    PtrValue instanceValue = object.eval( runEnv ).addressValue();
                    DeclList methodTable =
                         instanceValue.getValue().methodTable();
                    return new ObjectMethodPair( instanceValue,
                         methodTable.elementAt( decl.offset().interp() ) );
                    }
               default:
                    throw new Error(
                         "Invalid section of "
```

```
                        + Decl.sectionText( decl.section() ) );
                }
        }

    public Register saveReg( int dataType, Usage usage ) {
        return null;
        }

    public void evalCode( Node sibling ) {
        setSibling( sibling );
        object.evalCode( sibling );
        Register register = object.loadReg( usage() );
        Code.move( register, SpecialReg.newInstPtr );
        Code.load( IntType.type, SpecialReg.assemTemp,
            new Displacement(
                decl.env().absoluteName() + ".field.methodTablePtr",
                SpecialReg.newInstPtr ) );
        Code.load( IntType.type, SpecialReg.procValue,
            new Displacement( decl.absoluteName(), SpecialReg.assemTemp ) );
        }
    }
```

Evaluation of the Parameters

We have to check the number of actual parameters agrees with the number of formal parameters (unless the method is declared as being allowed to have additional parameters).

We have to check each actual parameter can be cast to the type of the corresponding formal parameter.

The eval() method takes a pointer as a parameter, to indicate where the parameters should be stored.

```
public class ExprListNode extends OperandNode {

    private Vector list = new Vector();

    public ExprListNode() {
        }

    public int size() { return list.size(); }

    public ExprNode elementAt( int i ) {
        return ( ExprNode ) ( list.elementAt( i ) );
        }

    public void setElementAt( int i, ExprNode expr ) {
        list.setElementAt( expr, i );
        }

    public void addElement( ExprNode node ) {
        list.addElement( node );
        }

    public String toString() {
        String result = "";
        for ( int i = 0; i < size(); i++ ) {
            if ( i > 0 )
                result += ", ";
            ExprNode expr = elementAt( i );
            result += expr;
            }
        return result;
```

```
        }

    public void genEnv( Env env ) {
        for ( int i = 0; i < size(); i++ ) {
            ExprNode expr = elementAt( i );
            expr.genEnv( env );
            }
        }

    public Type checkType() {
        for ( int i = 0; i < size(); i++ ) {
            ExprNode expr = elementAt( i );
            expr.checkType();
            }
        return null;
        }

    public void castParams( MethodType methodType ) {
        Offset paramOffset = new Offset();
        int formalSize = methodType.formalDecls().size();
        int actualSize = size();

        if ( actualSize > 0 && formalSize == 0 )
            throw new Error(
                "Method has actual parameters, but no formal parameters" );
        else if ( formalSize > 0 ) {
            Decl lastDecl =
                methodType.formalDecls().elementAt( formalSize - 1 );
            if ( lastDecl.ident().equals( "..." ) ) {
                --formalSize;
                if ( actualSize < formalSize )
                    throw new Error( "Too few actual parameters" );
                }
            else if ( actualSize < formalSize )
                throw new Error( "Too few actual parameters" );
            else if ( actualSize > formalSize )
                throw new Error( "Too many actual parameters" );
            }
        for ( int i = 0; i < formalSize; i++ ) {
            ExprNode expr = elementAt( i );
            Decl formalDecl = methodType.formalDecls().elementAt( i );
            Type formalType =
                methodType.formalDecls().elementAt( i ).type();
            setElementAt( i, expr.castTo( Type.CAST_PARAM, formalType ) );
            }
        }

    public void eval( RunEnv runEnv, PtrValue address ) {
        for ( int i = 0; i < size(); i++ ) {
            ExprNode expr = elementAt( i );
            RunValue exprValue = expr.eval( runEnv );
            address.elementAt( i ).setValue( exprValue );
            Print.error().debugln( "param " + i + " = " + exprValue );
            }
        }

    public Register saveReg( int regType, Usage usage ) {
        return null;
        }

    public void evalCode( OperandNode sibling ) {
```

```
            setSibling( sibling );
            for ( int i = 0; i < size(); i++ ) {
                ExprNode expr = elementAt( i );
                expr.evalCode( sibling );
                Register register = expr.loadReg( usage() );
                Code.storeQ( register,
                    new Displacement( "invoc.act" + i, SpecialReg.stackPtr ) );
                }
            }
        }
```

Return Statements

The way we implement the returning of a value from a method is interesting. We throw an exception, that contains the return value. The method invocation catches the exception, and returns the value as the result of the invocation. The reason we do it this way is because the return statement can be deeply nested within the subtree corresponding to the body, and hence its evaluation is deeply nested inside many invocations of eval().

```
public class ReturnExprStmtNode extends StmtNode {

    private ExprNode expr;
    private Decl methodDecl;
    private MethodDeclNode methodDeclNode;
    private MethodType methodType;
    private Type returnType;

    public ReturnExprStmtNode( ExprNode expr ) {
        this.expr = expr;
        }

    public String toString() {
        return "return " + expr +";";
        }

    public void genEnv( int section, Env env ) {
        super.genEnv( section, env );
        expr.genEnv( env );
        }

    public void checkType() {
        Type exprType = expr.checkType();
        methodDecl = env().methodDecl();
        if ( methodDecl == null )
            throw new Error( "Return not inside method" );
        methodDeclNode = ( MethodDeclNode ) methodDecl.declNode();
        methodType = ( MethodType ) methodDecl.type();
        returnType = methodType.returnType();
        expr = expr.castTo( Type.CAST_IMPLICIT, returnType );
        }

    public void eval( RunEnv runEnv ) throws ReturnException {
        RunValue result = expr.eval( runEnv );
        throw new ReturnException( result );
        }

    public void genCode() {
        Code.enter();
        expr.evalCode( null );
        expr.loadToReg( ( IntTempReg )
            ( new Usage( env() ).getFree( returnType ) ) );
```

```
          Code.br( new Relative( "return" ) );
          Code.exit();
          }
     }
```

Method Declarations

Method declarations have a local compile-time environment for the formal parameters and local variables. The local compile-time environment is passed down to the body of the method.

```
public class MethodDeclNode extends DeclStmtNode implements DeclNode {

     private TypeNode returnTypeNode;
     private String ident;
     private DeclStmtListNode formalParams;
     private DeclStmtListNode body;

     public String ident() { return ident; }
     public DeclStmtListNode formalParams() { return formalParams; }
     public DeclStmtListNode body() { return body; }

     private Env localEnv;
     private Type returnType;
     private Decl decl;

     public Env localEnv() { return localEnv; }
     public Type returnType() { return returnType; }
     public Decl decl() { return decl; }

     public MethodDeclNode(
          TypeNode returnTypeNode,
          String ident,
          DeclStmtListNode formalParams,
          DeclStmtListNode body ) {
          this.returnTypeNode = returnTypeNode;
          this.ident = ident;
          this.formalParams = formalParams;
          this.body = body;
          }

     public String toString() {
          String formalText, bodyText;
          if ( formalParams.size() == 0 )
               formalText = "()";
          else
               formalText = "(%+%n" + formalParams() + " )%-";
          if ( body == null )
               bodyText = ";";
          else if ( body.size() == 0 )
               bodyText = "%+%nbegin%nend%-";
          else
               bodyText = "%+%nbegin%+%n" + body + "%-%nend%-";
          return returnTypeNode + " " + ident
               + formalText + bodyText;
          }

     public void genEnv( int section, Env env ) {
          super.genEnv( section, env );
          localEnv = new Env( Env.ENV_METHOD, env );
          decl = env().declare( ident, Decl.DECL_METHOD, this );
          localEnv().setDecl( decl );
          returnTypeNode.genEnv( env );
```

```
        formalParams.genEnv( Decl.DECL_PARAM, localEnv );
        if ( body != null )
            body.genEnv( Decl.DECL_VAR, localEnv );
        }

    public void setType() {
        returnType = returnTypeNode.getType();
        formalParams().setType();
        if ( body != null )
            body().setType();
        DeclList paramDecls = localEnv().paramDecls();
        MethodType methodType = new MethodType( returnType, paramDecls );
        decl.setType( methodType );
        }

    public void checkType() {
        if ( body != null )
            body.checkType();
        }

    public void genOffset() {
        decl.genOffset();
        localEnv.varOffset().set( 0, 0 );
        if ( body != null )
            body.genOffset();
        localEnv.paramStart().set(
            localEnv.varOffset().interp(),
            localEnv.varOffset().compile() );
        formalParams.genOffset();
        }

    public void eval( RunEnv runEnv ) {
        }

    public void genDeclCode() {
        if ( body != null ) {
            Code.enterBlock( decl.declName(), "proc.sav0", "proc" );
            Code.enter( "code" );
            Code.align();
            Code.publicLabelDefn( "enter" );
            Code.loadA( SpecialReg.stackPtr,
                new Displacement( "-params", 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 );
            body.genCode();
            Code.clear( new IntTempReg( 0 ) );
            Code.labelDefn( "return" );
            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( "+params", SpecialReg.stackPtr ) );
```

```
                Code.ret();
                Code.exit( "code" );
                localEnv.genLocalDecls();
                Code.exitBlock( decl.declName() );
                }

    public void genCode() {
         }
    }
```