Chapter 11 Compile-Time Data Structures

Block structured languages

Most modern computer languages are "block structured languages". A **block** often corresponds to a declaration/statement sequence. Generally, no two declarations in a block can have the same identifier. The declarations can usually be referred to by their identifier within the block and enclosed blocks, and cannot be referred to outside the block. A declaration may be **hidden** in an enclosed block, by another declaration with the same identifier.

Life is more complicated in modern object oriented languages, especially in Java. It is possible to **overload** an identifier in a block and have more than one declaration in the block with the same identifier. In Java, there is usually no conflict between class, variable, method, and label declarations. Moreover, methods may have the same identifier, so long as they do not have the same number and type of formal parameters. The scope rules are also more complex, because we can refer to members of a class outside the class.

Many object-oriented languages will have blocks corresponding to

- Global declarations.
- The declarations of the members of a class. (But note that even though a constructor is declared in a class, it is actually accessible outside the class.)
- The formal parameter and top level declarations of a method.
- The declarations in a compound statement.
- A loop or switch statement. (Even loop and switch statements without any declarations correspond to blocks, because they define the scope of a break or continue statement.)
- A labelled statement.

The B-- language does not have labelled statements or declarations inside control statements, so the last three cases do not occur.

Compile-Time Environments

For every block we create a data structure representing the compile-time environment, to indicate the accessible declarations for the block. Every construct needs a reference to the innermost enclosing compile-time environment.

A compile-time environment includes

- An indication of the kind of environment (global, class, or method, and in many languages, compound statement, loop, or labelled statement).
- The parent declaration or construct (for example, the class or method the block is a part of). This gives us a way of determining the type of "this" or the required type for the expression in a return statement.
- The list of declarations within the local block.
- A reference to the enclosing environment (for methods and classes).
- A reference to the extended environment (for classes).

- The current offset(s) from the base of appropriate run-time data structure(s) for this block. As declarations are processed, the current offset is allocated to the declaration, and incremented by the amount of space required for the run-time value.
- The environment for a class also needs to build a "method table", to implement overriding for object-oriented languages.

We use this information to map identifiers to declarations. A simple linear table of declarations for each block is sufficient. We can use data structures, such as hash tables to increase the efficiency of the search. However, programs in object oriented languages often have a large number of different blocks, with a small number of declarations in each block, so algorithms such as hashing might not work very well.

For example, in the B-- language we have

```
public class Env {
     public final static int ENV_NONE      = 0;
     public final static int ENV_CLASS     = 1;
     public final static int ENV_GLOBAL    = 2;
     public final static int ENV_METHOD    = 3;

     private int envKind;
     private Decl decl;
     private DeclList localDecls = new DeclList();
     private Env enclosingEnv;
     private Env extendedEnv;
     private Offset varOffset = new Offset();
     private Offset methodOffset = new Offset();
     private Offset paramStart = new Offset();
     private DeclList methodTable = new DeclList();
     ...
     }
```

The program Program/scope

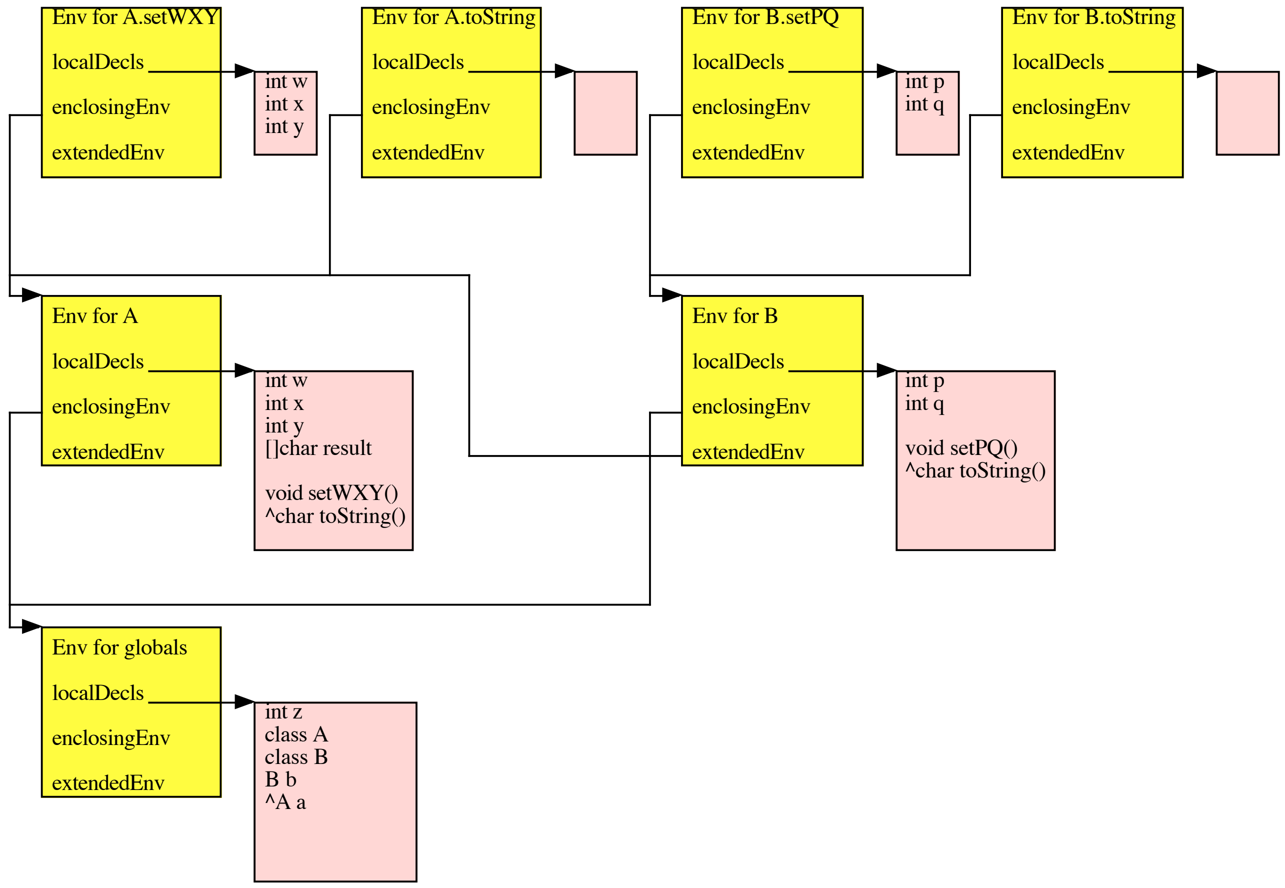
```
int z;

class A
     begin
          int w, x, y;
          [ 100 ]char result;
          void setWXY( int w, x, y; )
               begin
                    this.w = w;
                    this.x = x;
                    this.y = y;
               end
          ^char toString()
               begin
                    sprintf( result,
                         "class A: w = %d, x = %d, y = %d", w, x, y );
                    return result;
               end
     end

class B extends A
     begin
          int p, q;
          void setPQ( int p, q; )
               begin
                    this.p = p;
                    this.q = q;
               end
          ^char toString()
               begin
                    sprintf( result,
                         "class B: p = %d, q = %d, w = %d, x = %d, y = %d",
                         p, q, w, x, y );
                    return result;
               end
     end

B b;
b.setPQ( 111, 222 );
b.setWXY( 333, 444, 555 );
^A a = b;
printf( "a = %s\n", a.toString() );
```

generates compile-time environments:



Searching the compile-time environment

When processing a declaration, we need to check the identifier is not being redeclared in the same block (B-- does not support overloading). We search for the identifier in the local block, and generate an error message if we find it. If there is no redeclaration of the identifier, we insert it in the declaration list.

```
public class Env {

     ...
     public Decl searchLocal( String ident ) {
          return localDecls.searchDeclList( ident );
          }
     public Decl declare(
          String ident,
          int section,
          DeclNode declNode ) {
          if ( searchLocal( ident ) != null )
               throw new Error(
                    "Identifier " + ident + " redeclared in same environment" );
          return localDecls.declare( ident, section, declNode, this );
          }
     ...
     }
public abstract class DeclaratorNode extends OperandNode implements DeclNode {
     ...
     public void genEnv( int section, Env env ) {
          this.section = section;
          super.genEnv( env );
          decl = env().declare( ident, section, this );
          }
     ...
     }
```

When processing an expression involving member selection, we first determine the type of the object being selected, check it corresponds to a pointer to a class instance, and obtain the environment of the class from the information stored for the class type. We first search for the identifier in the local block of the class type, then, if not found locally, in the extended environment. A similar process occurs when determining whether a method overrides a method in the superclass.

```
public class Env {

     ...
     public Decl searchExtended( String ident ) {
          Decl localDecl;
          localDecl = searchLocal( ident );
          if ( localDecl != null )
               return localDecl;
          else if ( extendedEnv == null )
               return null;
          else
               return extendedEnv.searchExtended( ident );
          }
     ...
     }


public class MemberVariableNode extends VariableNode {

     ...
     public Type checkType() {
          Type oType = object.checkType();
          if ( ! ( oType instanceof PtrType ) )
               error( "Selection must be from a reference" );
          PtrType ptrType = ( PtrType ) oType;
          Type sType = ptrType.subType();
          if ( ! ( sType instanceof ClassInstanceType ) )
               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 )
               error( "Undeclared field " + ident );
          if ( decl.section() != Decl.DECL_VAR )
               error( ident + " must be field" );
          type = decl.type();
          return type;
          }
     ...
     }
```

When processing a variable corresponding to a simple identifier, we first search for the identifier in the local block, then, if not found locally, in the enclosing and extended environments. If there is a conflict, generate an error message.

```
public class Env {

     ...
     public Decl searchEnv( String ident ) {
          Decl localDecl;
          localDecl = searchLocal( ident );
          if ( localDecl != null )
               return localDecl;
          Decl extendedDecl;
          if ( extendedEnv == null )
               extendedDecl = null;
          else
               extendedDecl = extendedEnv.searchExtended( ident );
          Decl enclosingDecl;
          if ( enclosingEnv == null )
               enclosingDecl = null;
          else
               enclosingDecl = enclosingEnv.searchEnv( ident );
          if ( extendedDecl != null && enclosingDecl != null )
               throw new Error( "Ambiguous declaration of " + ident );
          if ( extendedDecl != null && enclosingDecl == null )
               return extendedDecl;
          else if ( enclosingDecl != null && extendedDecl == null )
               return enclosingDecl;
          else
               return null;
          }
     ...
     }

public class IdentVariableNode extends VariableNode {
     ...
     public Type checkType() {
          decl = env().searchEnv( ident );
          if ( decl == null )
               throw new Error( "Undeclared Identifier " + ident );
          switch ( decl.section() ) {
               case Decl.DECL_VAR:
               case Decl.DECL_PARAM:
                    type = decl.type();
                    return type;
               default:
                    throw new Error( ident + " must be variable or parameter" );
               }
          }
     ...
}
```

Declaration Lists

A DeclList is just the list of local declarations in a block.

```
public class DeclList {
     ...
     public Decl searchDeclList( String ident ) {
          for ( int i = 0; i < size(); i++ ) {
               Decl decl = elementAt( i );
               if ( decl.ident().equals( ident ) )
                    return decl;
               }
          return null;
          }
     ...
     }
```

Declarations

The data structure for a declaration includes

- The identifier.
- The kind of declaration (class type, method, variable, formal parameter).
- The type of the declaration.
- The node of the abstract syntax tree for the declaration.
- The environment of the declaration.
- The offset from the base of the run-time data structure for the block.
- Possibly the declaration it overrides, for a method declaration.

In other languages, we could have additional information. For example, flags to say whether the declaration is a static or instance declaration, final, abstract, access rights (public, private, protected or package), etc.

```
public class Decl {

     public final static int DECL_NONE     = 0;
     public final static int DECL_CLASS    = 1;
     public final static int DECL_METHOD   = 2;
     public final static int DECL_VAR      = 3;
     public final static int DECL_PARAM    = 4;

     private String ident;
     private int section;
     private Type type;
     private DeclNode declNode;
     private Env env;
     private Offset offset;
     private Decl overRiddenDecl;

     public void setType( Type type ) {
          this.type = type;
          }
     ...
     }
```

Types

We have an abstract Type class corresponding to types in general. Each actual type is represented by a concrete class that extends this abstract class. The Type class includes a static method balanceType, for determining the common type of two operands of the “==” and “!=” operators.

There are also methods for determining whether a type is equal to another, or castable to another. These methods are overridden in subclasses.

In B--, expressions involving variables declared of array or class instance type generate a value of pointer type. The formalType() method maps the declared type to the formal type. We also have to distinguish between the type of a class identifier, and the type of a variable corresponding to that class. The instanceType() method maps a class type to an instance type. These methods are overridden in subclasses.

```
public abstract class Type {

     public final static int CAST_EXPLICIT =    1;
     public final static int CAST_IMPLICIT =    2;
     public final static int CAST_PARAM    =    3;
     public final static int WORD          =    8;

     public static Type balanceType( Type type1, Type type2 ) {
          if ( type1.isCastable( CAST_IMPLICIT, type2 ) )
               return type1;
          if ( type2.isCastable( CAST_IMPLICIT, type1 ) )
               return type2;
          else
               return null;
          }

     public abstract boolean equals( Type type );
     public abstract String toString();

     public Type instanceType() {
          return this;
          }

     public Type formalType() {
          return this;
          }

     public Type primitiveType() {
          return this;
          }

     public boolean isCastable( int castKind, Type type ) {
          return equals( type );
          }

     public Offset size() {
          return new Offset( 1, WORD );
          }

     public abstract void initAddress( RunEnv runEnv, PtrValue address )
          throws ReturnException;

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

For example, the int type is represented by

```
public  class IntType extends BasicType {

     public final static IntType type = new IntType();
```

```
	public IntType() {
		}

	public boolean equals( Type type ) {
		return type instanceof IntType;
		}

	public String toString() {
		return "int";
		}

	public boolean isCastable( int castKind, Type type ) {
		return type instanceof IntType || type instanceof CharType;
		}

	public void initAddress( RunEnv runEnv, PtrValue address ) {
		address.setValue( IntValue.defaultValue );
		}
	}
```

A ponter type contains a reference to its subtype.

```
public class PtrType extends Type {

	private Type subType;
	public Type subType() { return subType; }

	public PtrType( Type subType ) {
		this.subType = subType;
		}

	public boolean equals( Type type ) {
		if ( ! ( type instanceof PtrType ) )
			return false;
		PtrType otherType = ( PtrType ) type;
		return subType().equals( otherType.subType() );
		}

	public boolean isCastable( int castKind, Type type ) {
		if ( type instanceof NullType )
			return true;
		else if ( equals( type ) )
			return true;
		else if ( type instanceof PtrType )  {
			PtrType ptrType = ( PtrType ) type;
			return subType.isCastable( castKind, ptrType.subType() );
			}
		else
			return false;
		}

	public String toString() {
		return "^" + subType;
		}

	public void initAddress( RunEnv runEnv, PtrValue address ) {
		address.setValue( NullValue.defaultValue );
		}
	}
```

An array type contains the array size and a reference to its subtype. Notice how formalType() is overridden to return the type ^subType.

```
public  class ArrayType extends Type {

     private int numElements;
     private Type subType;
     public int numElements() { return numElements; }
     public Type subType() { return subType; }

     public ArrayType( int numElements, Type subType ) {
          this.numElements = numElements;
          this.subType = subType;
          }

     public Type formalType() {
          return new PtrType( subType );
          }

     public boolean equals( Type type ) {
          if ( ! ( type instanceof ArrayType ) )
               return false;
          ArrayType otherType = ( ArrayType ) type;
          return numElements == otherType.numElements()
               && subType().equals( otherType.subType() );
          }

     public boolean isCastable( int castKind, Type type ) {
          return equals( type );
          }

     public String toString() {
          return "[ " + numElements + " ]" + subType;
          }

     public Offset size() {
          Offset elementSize = subType.size();
          return new Offset(
               numElements * elementSize.interp(),
               numElements * elementSize.compile() );
          }

     public void initAddress( RunEnv runEnv, PtrValue address )
          throws ReturnException {
          int elementSize = subType.size().interp();
          for ( int i = 0; i < numElements; i++ ) {
               subType.initAddress( runEnv,
                    address.elementAt( elementSize * i ) );
               }
          }
     ...
     }
```

A class type contains a reference to its environment, to provide access to its declarations. Notice how instanceType() is overridden to return the instance type corresponding to the class type.

```
public  class ClassType extends Type {

     private String ident;
     private Env env;

     public String ident() { return ident; }
     public Env env() { return env; }

     public ClassType( String ident, Env env ) {
          this.ident = ident;
          this.env = env;
          }

     public boolean equals( Type type ) {
          if ( ! ( type instanceof ClassType ) )
               return false;
          ClassType classType = ( ClassType ) type;
          return classType.ident().equals( ident );
          }

     public boolean isCastable( int castKind, Type type ) {
          return false;
          }

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

     public Type instanceType() {
          return new ClassInstanceType( ident, env );
          }

     public Offset size() {
          return new Offset( 0, 0 );
          }
     ...
     }
```

We have to distinguish between the type of a class identifier (ClassType) and a variable declared as an instance of a class (ClassInstanceType). Notice how formalType() is overridden to return the type ^classInstanceType.

```
public  class ClassInstanceType extends Type {

     private String ident;
     private Env env;

     public String ident() { return ident; }
     public Env env() { return env; }

     public ClassInstanceType( String ident, Env env ) {
          this.ident = ident;
          this.env = env;
          }

     public boolean equals( Type type ) {
          if ( ! ( type instanceof ClassInstanceType ) )
               return false;
          ClassInstanceType classInstanceType = ( ClassInstanceType ) type;
          return classInstanceType.ident().equals( ident );
          }

     public boolean isCastable( int castKind, Type type ) {
          if ( type instanceof ClassInstanceType ) {
               ClassInstanceType instanceType = ( ClassInstanceType ) type;
               return env.isExtensionOf( instanceType.env() );
               }
          return false;
          }

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

     public Type formalType() {
          return new PtrType( this );
          }

     public Offset size() {
          return env.varOffset();
          }

     public void initAddress( RunEnv runEnv, PtrValue address )
          throws ReturnException {
          address.setValue( new MethodTableValue( env.methodTable() ) );
          RunEnv localRunEnv
               = new RunEnv( runEnv.globals(), address, null );
          env.initInstance( localRunEnv );
          }
     ...
     }
```

The code in the Env class performs initialisation for the class and its superclasses.

```
     public void initInstance( RunEnv runEnv ) throws ReturnException {
          if ( extendedEnv != null )
               extendedEnv.initInstance( runEnv );
          ClassDeclNode classDeclNode = ( ClassDeclNode ) decl().declNode();
          DeclStmtListNode body = classDeclNode.body();
          body.eval( runEnv );
          }
```

A method type contains a reference to its return type and formal parameters.

```
public  class MethodType extends Type {

     private Type returnType;
     private DeclList formalDecls;
     public Type returnType() { return returnType; }
     public DeclList formalDecls() { return formalDecls; }

     public MethodType( Type returnType, DeclList formalDecls ) {
          this.returnType = returnType;
          this.formalDecls = formalDecls;
          }

     public boolean equals( Type type ) {
          if ( ! ( type instanceof MethodType ) )
               return false;
          MethodType methodType = ( MethodType ) type;
          return
               methodType.returnType().equals( returnType )
               && methodType.formalDecls().equals( formalDecls );
          }

     public String toString() {
          return returnType() + "(%+" + formalDecls() + " %-)";
          }

     public boolean isCastable( int castKind, Type type ) {
          return false;
          }
     ...
     }
```

Cyclic Dependencies

When processing class declarations, we need to ensure that the programmer did not create a cycle of dependencies, for example where class A extends B which extends C which extends A. This is easy. Just check that the chain of superclasses for the class being extended does not already contain the subclass.

In Java, array and class instance variables are actually reference (pointer) variables. The variable is initialised to a null pointer, and the programmer has to explicitly allocate space, if they want the variable to point to a non-null object.

However, in the C language (and hence in B-- and C++) array and class instance variables are allocated space "in-line", as a part of the overall block. This gives the programmer the potential to declare classes of infinite size. For example, if we write

```
class Node
     begin
          int value;
          Node next;
     end
```

then an instance of class Node requires the space needed for an int, together with the space for an instance of class Node! What should have been written was

```
class Node
     begin
          int value;
          ^Node next;
     end
```

C resolves this problem by requiring classes to be declared before they are used as fields, but permitting fields of pointer type for classes that are yet to be declared. A better way, that gives the programmer more flexibility in the way they order their code, would be to check for cycles.

The B-- compiler/interpreter fails to check for cyclic dependencies. An exercise for the future!