Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2000 IR Framework Tuesday, September 28

1 Introduction

This handout describes an IR framework that you may use in your compiler. An IR serves two purposes. First, it should be able to represent a program in an abstract manner, as independently as possible of both the source language and the target machine. Second, it must have easy to use interfaces and primitives to allow various optimization implementations. Traditionally, a parser builds an abstract syntax tree (AST) that may contain language dependent constructs. A compiler converts the AST into an IR by replacing those constructs with language-independent constructs.

Our IR framework is flexible enough that you do not need to build a separate AST. You may, in your parser implementation, generate a high-level IR with high-level control flow constructs. Furthermore, the IR framework allows (and in some cases requires) compiler extensions. Thus your compiler can define more language-specific constructs, such as types and immediates. After the parser phase of your compiler, you may perform transformations on the high-level IR to convert it into a language-independent IR.

2 IR Constructs

Constructs in the IR framework can be divided into three sections: high-level control flow constructs, instructions, and helper constructs such as Descriptors.

2.1 High Level Control Flow Constructs

There are two kinds of high-level control flow constructs in our IR framework. The first is the TreeNode classes:

- TreeWhile (extends TreeNode) - Represents a while loop.
- TreeFor (extends TreeNode) - Represents a for loop.
- TreeIf (extends TreeNode) - Represents an if-else construct.
- TreeBlock (extends TreeNode) - Represents scopes, such as if bodies or procedure bodies.

A program is represented by a tree of TreeNode objects. At the high level, a program may be a TreeBlock of methods, each of which is a TreeBlock of TreeNodes such as Instructions, TreeWhile objects, TreeIf objects, etc.

TreeBlock can be used to store a list of TreeNode objects. When a TreeNode object is inserted into a TreeBlock for the first time, this TreeBlock serves as the parent of the TreeNode. Since each

TreeBlock also has a TreeNode as the parent, this creates a chain of TreeNode/TreeBlock that can be used to do scope lookups. For example, a TreeBlock is usually embedded within a TreeBlock object. Therefore, any TreeNode on the TreeBlock is also embedded within the TreeBlock object. Thus, we can find the symbol table for a TreeNode by looking up through its TreeBlock at the TreeBlock that contains the TreeNode.

After the initial insertion to a TreeBlock, subsequent insertions of a TreeNode into other lists do not generate further bindings, unless the initial TreeBlock is removed as the parent of the TreeNode via either a call to setParentList(null), or removal of the TreeNode from the parent list. Thus, if you wish to remove a TreeNode from a list and insert it into another list, and have the second list be the parent of the TreeNode, then the correct order is: remove, then insert.

Compilers can further extend the TreeNode class or any of its subclasses to create even more high-level constructs. For example, a compiler may want to capture more information in a procedure representation than what's there in a TreeBlock, therefore it may create its own TreeProc class, perhaps containing a TreeBlock object as the procedure's body, a Type object as the returning type, etc.

2.2 Instructions

The basic building block of TreeNode objects is an Instruction object. All TreeNode objects can be decomposed into multiple Instruction objects. An Instruction object represents an operation on one or two operands. Subclasses extending the Instruction abstract class define their own semantics.

Three default Instruction subclasses are given:

- BranchInstr (extends Instruction) - Represents a procedure call instruction. Contains a destination (LabelInstr) and an optional condition (Rhs).
- LabelInstr (extends Instruction) - Represents a branch target, as a textual label. Contains a label name (String).
- ReturnInstr (extends Instruction) - Represents a return instruction. Contains a return value (Rhs, possibly empty).
- SimpleInstr (extends Instruction) - Represents a simple assignment operation, or an operation where the value is thrown away. Contains a destination (Lhs, possibly empty), and a source (Rhs). Examples of this include

  ```
  a = b+c;
  a = foo();
  a;
  foo();
  ```

Instruction objects that exist in low-level IR should either have a one-to-one mapping with a machine instruction, or should have a simple mapping to a set of machine instructions, whereas Instruction objects that exist in a high-level IR may, for convenience reasons, be more abstract. For example, the source of a SimpleInstr can be any Rhs, including a complicated Expression tree (see Section 2.3), but in the low-level IR you'll want to flatten this by defining intermediate variables and converting to a sequence of SimpleInstr nodes that compute the value of the Expression tree.

2.3 Expressions

The Expression class is used to create expression trees, similar to expression trees one might define in a simple mathematical grammar. Expression is an abstract class in our IR, but several classes extend it:

- AllocExpr - Represents an allocation
- ArrayExpr (implements Lhs) - Represents an array access.
- CallExpr - Represents a function call.
- OpExpr - Represents simple ¡expr¿ OP ¡expr¿ combinations.
- ImmedExpr - Represents immediate value in an expression.
- VarExpr - Represents variable in an expression.

Using these classes, you can build up a complicated expression tree involving function calls, mathematical operations, and array and variable accesses. Expressions implement the `Rhs` interface, which means they can be passed as arguments to Instructions. Nodes such as TreeIf also use expressions for the conditional.

Not all of the expressions are completely implemented. ImmedExpr is an abstract class which needs to be extended to do anything useful. VarExpr is complete, but uses the VarDescriptor class which is abstract.

The opcode argument to an OpExpr in the IR framework is some arbitrary integer. Compilers are expected to define their own opcodes for their languages and platforms. The following is an example of an possible Opcode class:

```
public class calculatorOpcode
{
    public static final int ADD = 0; // addition
    public static final int SUB = 1; // subtraction
    public static final int MUL = 2; // multiplication
    public static final int DIV = 3; // division
    public static final int POW = 4; // power
}
```

Here, the first four opcode types are basic operations. The fifth opcode, POW, is used to compute power. When generating low IR, the compiler may translate a SimpleInstr object with POW into multiple instructions with more basic opcodes.

2.4 Lhs and Rhs

Lhs and Rhs are interfaces in the IR. They don't specify any methods, but they're used to designate which types of IR nodes can be used as arguments to other nodes. For example, SimpleInstr takes an Lhs destination and an Rhs source. All Expression-derived nodes implement Rhs, so they can all be used as the source for SimpleInstr nodes, or as either branch of an OpExpr, or as the condition in a TreeWhile, and so on. ArrayExpr and VarDescriptor (hence VarExpr) implement Lhs, so those can be used as the destination of a SimpleInstr.

You may find it useful to extend the Rhs/Lhs definitions to include methods, and implement those methods in extensions of the various IR classes.

2.5 Descriptor

The Descriptor classes in the framework hold information about named objects. They all implement a constructor that takes a String and a method toString() that returns that string. LabelDescriptors can be used to generate unique labels and keep track of reference counts. The ProcDescriptor and VarDescriptor classes are abstract and need to be extended to be useful. VarDescriptor implements Rhs and Lhs, so it can be used as nodes in Expression trees, or as conditionals in TreeIfs.

Compilers should extend these classes to represent and capture information for these constructs.For example, you may also want to create a procDescriptor class extending ProcDescriptor to represent procedures, and this class may keep records such as modifiers, return types, location of the code, its index in a method table, etc.

2.6 SymbolTable and IRType

Our IR does not hold any concrete notion of symbols and symbol tables. Your compiler must define different types of symbols and the symbol table that can be used to resolve them. The SymbolTable class is merely a place holder.

The IR framework also does not have a notion of Type, since types are strictly language-specific and thus should be defined by the compiler. In a compiler implementation, a symbol table often encapsulates the bindings between symbols and types. We define IRType as an empty abstract class.

Typically you will include type and symbol table information using annotations (see Section 4).

3 Example Program

Example of a program and its TreeNode representation:

```
void foo()
{
  boolean b;
  if (b) {
    ...
    b = false;
  }
  ...
}
```

can be represented as:

```
TreeBlock (program):
   |
   |--> TreeBlock (foo's body):
            |
            |---> TreeIf
            |       |----> Expression (condition)
            |       |          |
            |       |          |-----> VarDescriptor: b
            |       |----> TreeBlock (body)
            |                   |----> ...
            |                   |----> ...
            |                   |----> SimpleInstr (b = false)
            |                              |
            |                              |----> VarDescriptor: b
            |                              |----> ImmedExpr: false
            |---> ...
```

4 Annotations

Annotations can be used to pass information between different phases of the compiler. All objects of the IR framework extend from the superclass IRObject, which implements the Annotable interface. Annotable objects implement getAnnotation() and annotate(). Annotations are tagged with integers, and can contain any Object as the data. Typically you'll cast the return value of getAnnotation() to a known type before using it.

There are many uses of annotations. For example, when implementing optimizations, different optimization algorithms may wish to share information. These information may be stored within Annotation objects. Type propagation can also take advantage of annotations. You can store the type of an operand as an Annotation hooked to the Operand object. Finally, the getHierAnnotation, implemented under the TreeNode object hierarchy, may be useful for storing symbol table information.

5 Debugging Tools

The IR framework provides two simple debugging tools: an IR visualizer, and an ASCII tree printer. Both of these tools operate on objects implementing the Walkable interface. Methods defined by this interface provide information such as name of the node (to be displayed by the visualizer) and number of children (when printing or displaying children).

You may use the visualizer to view any IR built by your parser. To create a visualizer, pass the the root node of the IR to the constructor of the java6035.tools.IRVis.IRVisWin class. To pop a window open, call the popup() method.

There are two viewing modes, Tree Mode or List Mode. In Tree Mode, your IR tree is drawing top down as an inverted tree. The root is on top, at the next level are children of the root, then children of those nodes, and so on. In List Mode, all children of a node are listed under the node, vertically. This is similar to GUI directory viewing tools.

Within the debugger window, you may click on any node to view its structure and annotations. This information appears on the bottom of the window. To expand a node and view the subtree below it, double click on the node. In Tree Mode, you may also drag a node around, then select the "Refresh" option under the Display menu to redraw its subtree.

An ASCII representation of the tree can be obtained by calling the PPrint() method on the root of the tree. See the API documentation for a detailed description of the arguments to this method.

If your compiler defines extensions to the IR framework, those extensions should also implement the Walkable interface. This way your extended classes will also show up on the visualizer and in the printed ASCII tree.

The visualizer is not an efficient tree drawing program. When the tree gets large, it can be slow. If you want to use the visualizer to debug your program, you should narrow down to a smaller tree where the problem may occur and then pass it to the visualizer.

6 Generation of high-level IR

You should generate your high-level IR in the CUP parser you built. This can be done by inserting IR generation code into the action field of your grammar file. For example, suppose you are generating IR for the following rule:

```
EXPRESSION:e := EXPRESSION:e1 op EXPRESSION:e2
```

You can define your EXPRESSION non-terminal to be of type Expression in the beginning of the grammar. Then you can add this rule to the action for that production:

```
{: e = new OpExpr(op, e1, e2); :}
```

Now the expression e carries a new Expression which is an OpExpr object made from the two subexpressions.

You should also build your symbol table objects while parsing the input file. You may do so by using a stack of symbol tables to capture scopes in your language. That is, at the beginning of every scope, push a new symbol table onto the stack, and at the end of the scope, pop it off. This method, together with some clever CUP specifications, will allow you to resolve all identifiers for simple languages (where identifiers must be defined before used) at parsing time.