Computer Languages

Intermediate Representation - LLVM

Verónica Gaspes
www2.hh.se/staff/vero

Center for Research on Embedded Systems
School of Information Science, Computer and Electrical Engineering

Types in LLVM abstract assembler

LLVM is strongly typed. All instructions are typed, all values have a type!

Primitive types

- Integer types `i1`, `i2`, `i3`, `...i8`, `...i16`, `...i32`, `...`
- Other primitive types like `label` and `void`
- There are also floats and doubles (we will not need them for compiling minijava!)

Types in LLVM abstract assembler

Derived types

- Array types `[<# elements> x <elementtype>]`
  e.g. `[40 x i32]`
- Pointers `<type> *`
  e.g. `[4 x i32]*`
- Structures `{ <type list> }`
  e.g. `{ i32, (i32)*, i1 }`
- Function types `<returntype list> (<parameter list>)`
  e.g. `i32 (i32)`
  e.g. `{i32, i32} (i32)`

The `getelementptr` instruction

Syntax

```
<result> = getelementptr <pty>* <ptrval>
                         {, <ty> <idx>}*
```

Purpose

The `getelementptr` instruction is used to get the address of a subelement of an aggregate data structure. It performs address calculation only and does not access memory.

The first argument is always a pointer, and forms the basis of the calculation.

The remaining arguments are indices that indicate which of the elements of the aggregate object are indexed. The interpretation of each index is dependent on the type being indexed into.

Understanding getelementptr

An example in C

```
struct RT {
  char A;
  int B[10][20];
  char C;
};
struct ST {
  int X;
  double Y;
  struct RT Z;
};

int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

An llvm version

```
%RT = type
 {i8,[10 x [20 x i32]],i8 }
%ST = type
 {i32,double,%RT }
define i32* @foo(%ST* %s) {
entry:
  %reg = getelementptr
           %ST* %s,
           i32 1,
           i32 2,
           i32 1,
           i32 5,
           i32 13
  ret i32* %reg
}
```


Infrastructure for the minijava compiler

Goal

We want to generate LLVM abstract assembler for minijava programs. We can generate very naive code, then we can use the llvm tools to optimize and generate executables in a number of architectures!

How?

By traversing the abstract syntax tree of a minijava program generate LLVM code.
In practice this is done programming a visitor for the abstract syntax tree.

What do we generate?

1. a string (like the `PrettyPrintVisitor`), or
2. a value in some datastructure that in turn can be printed

Abstract syntax for LLVM abstract assembler

A collection of classes that implement the abstract syntax trees for LLVM types, values and instructions.
It is what you will get for your project work.

You will have to

1. design a symbol table,
2. program a visitor that does declaration elaboration filling the symbol table and
3. program a visitor that generates llvm abstract assembler.

The abstract classes

LlInstruction.java

```
package astLlvm;
public abstract class LlInstruction{
}
```

We will make all declarations, labels and instructions inherit from this class.

The abstract classes

LlType.java

```
package astLlvm;
public abstract class LlType{
}
```

We will make all types inherit from this class.

LlValue

```
package astLlvm;
public abstract class LlValue{
    public LlType type;
}
```

All values, named values, function values, constant values will inherit from this class.

The Values

Integer Literals

```
package astLlvm;
public class LlIntegerLiteral extends LlValue{
    public int value;
    public LlIntegerLiteral(int value){
       type = LlPrimitiveType.I32;
       this.value = value;
    }

    public String toString(){
       return ""+ value;
    }
}
```


The Values

Named Values

```
package astLlvm;
public class LlNamedValue extends LlValue{
    public String name;
    public LlNamedValue(String name, LlType type){
       this.type = type;
       this.name = name;
    }
    public String toString(){
       return name;
    }
}
```


The Instructions

Add

```
package astLlvm;
public  class LlAdd extends LlInstruction{
    public LlNamedValue lhs;
    public LlType type;
    public LlValue op1, op2;
    public LlAdd(LlNamedValue lhs, LlType type,
                 LlValue op1, LlValue op2){
       this.lhs = lhs;this.type = type;
       this.op1 = op1;this.op2 = op2;
    }
    public String toString(){
       return "  " +lhs + " = add "
             + type + " " + op1 + ", " + op2;
    }
}
```

Compiling classes

Example

```
class Test{
  public static void main(){
    System.out.println(new A().f(3));
  }
}
class A {
   int x;
   int[] a;
   ...
   int f(int y){
     ...
     return x + a[y];
   }
}
```

Alert!

I will use concrete syntax but you should thing of the abstract syntax tree!

Two passes

Elaborating declarations

One visitor that generates an environment for the identifiers of the program.

class	information about fields and methods
field	an offset among the fields in the class and an llvm type
method	a unique llvm identifier and a type
argument	an llvm identifier and a type
local variables	an llvm identifier and a type

Two passes

Code generation

- One visitor that collects a List of LLVM instructions
- Then you can go through this list and print all instructions to a file (there is a method `toString` on all parts of the LLVM's assembler abstract syntax)
- You will have to keep other things, like a number for naming temporaries and a number for naming labels.
- When visiting an expression you should return the value you generate, because you will need it as argument to instructions!

Two passes

One possible way of doing things

```
class CodeGenerator
       implements Visitor<LlValue, CodeSymbolTable>{

  private List<LlInstruction> assembler;
  private int tmpNr;
  private int ifLabelNr;

  public CodeGenerator(){
    assembler = new LinkedList<LlInstruction>();
    tmpNr = 0;
    ifLabelNr = 0;
  }
```

Two passes

Code generation - easy cases: e1+e2

```
public LlValue visit(Plus n, CodeSymbolTable e){
   LlValue v1 = n.e1.accept(this,e);
   LlValue v2 = n.e2.accept(this,e);
   LlNamedValue lhs =
      new LlNamedValue("%tmp"+(tmpNr++),
          LlPrimitiveType.I32);
   assembler.add(
      new LlAdd(lhs,LlPrimitiveType.I32,v1,v2));
   return lhs;
}
```

1+2+3

```
%tmp0 = add i32 1, 2
%tmp1 = add i32 %tmp0, 3
```


Two passes

Code generation - easy cases: IF

```
public LlValue visit(If n, CodeSymbolTable e){
   LlValue cond = n.e.accept(this,e);
   LlLabelValue ifThen =
        new LlLabelValue("if.then"+(ifLabelNr));
   LlLabelValue ifElse =
        new LlLabelValue("if.else"+(ifLabelNr));
   LlLabelValue ifEnd =
        new LlLabelValue("if.end"+(ifLabelNr++));
   ...
}
```


Two passes

Code generation - easy cases: IF

```
public LlValue visit(If n, CodeSymbolTable e){
   ...
   assembler.add(new LlConditionalBranch(cond,
                                          ifThen,
                                          ifElse));
   assembler.add(new LlLabel(ifThen));
   n.s1.accept(this,e);
   assembler.add(new LlBranch(ifEnd));

   assembler.add(new LlLabel(ifElse));
   n.s2.accept(this,e);
   assembler.add(new LlBranch(ifEnd));

   assembler.add(new LlLabel(ifEnd));
   return null;}
```


Two passes

Code generation

1. Declarations to implement `System.out.println(intValue)`
2. Definitions for each of the functions. Remember to organize the code in basic blocks (related to `if` and `for`)
3. Generate a new register name for each instruction that returns a value.
4. Use `alloca` for the variables in a function.
5. For `new A()` use `malloc` to get heap space. You will need to use a structure type, then the components of the structure can be retreived using `getelementptr` and the offset .
6. For `new int[length]` use `malloc` to get heap space. You will need to use an array type, then the elements can be accessed using `getelementptr` and an index.

Why place objects and arrays in the heap?

alloca is used to allocate memory in the frame of a function that is placed on the stack when the function is called and removed from the stack when the function returns.

malloc is used to allocate memory in the heap: available during all of program life.

Example in minijava

```
class F{
    int x;
    int [] a;
    void build(int length){
        x = length;
        a = new int[length];
        for(int i = 0; i<length; i++)
            a[i]=length-i;
    }
    int sum(){
        int s = 0;
        for(int i = 0; i<x; i++)
            s = s + a[i];
        return s;
    }
```


Two passes

Code generation - method call

`obj.m(args)`

Generate a call instruction with

1. The function name assigned to m that you find in the environment. It should be qualified with the class(es) name!
2. The function type assigned to m that you find in the environment. It should include one extra argument type for the type of obj
3. The arguments should include obj

Two passes

Code generation - this

What does this mean? What do the fields in class refer to?

It is only inside method calls that we can find references to this or to the fields of class:

```
class A{
  int y;
  T m(int x){
    this.f();
    y = 3;
    return new T();
  }
```

It is the object on which the metod is called

`obj.m(3)`

In the translation m will have one extra argument:

`T m(A thisArg, int x)`

And it is thisArg that is this and it is thisArg fields that are refered to!