OLLIR (Object-Oriented Low-Level Intermediate Representation) and the OLLIRTool

Compilers course

Masters in Informatics and Computing Engineering (MIEIC), 3rd Year

João M. P. Cardoso
Email: jmpc@fe.up.pt

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Outline

- About three-address code representations
- About OLLIR
- OLLIR Examples
- About the OLLIRTool
- From OLLIR code to JVM code

About three-address code representations

- Intermediate representations mainly consisting of instructions with three operands
  - typical instructions include an assignment and a binary operation
  - closer to the register file based machines, such as RISC machines
  - includes conditional and unconditional branches
  - includes memory accesses
- Example of a three-address code representations
  - GCC RTL code

https://en.wikipedia.org/wiki/Three-address_code

About OLLIR

- Inspired by three-address code representations
- Includes calls to functions following a multiple arguments, single result, format
- An OLLIR file represents a class
- Includes put and get primitives to write/read fields
- Supports classes and objects
- Elements of the OLLIR statements include attributes explicitly identifying the type of data

- It can be seen an intermediate low-level representation to represent Java classes

OLLIR Examples (1)

ex1.lir

```
myClass {
  .construct myClass().V {
     invokespecial(this, "<init>").V;
  }
  .method public sum(A.array.i32).i32 {
        sum.i32 :=.i32 0.i32;
        i.i32 :=.i32 0.i32;
     Loop:
        t1.i32 :=.i32 arraylength($1.A.array.i32).i32;
        if (i.i32 >=.i32 t1.i32) goto End;
        t2.i32 :=.i32 $1.A[i.i32].i32;
        sum.i32 :=.i32 sum.i32 +.i32 t2.i32;
        i.i32 :=.i32 i.i32 +.i32 1.i32;
        goto Loop;
     End:
        ret.i32 sum.i32;
  }
}
```

```
class myClass {
  public int sum(int[] A){
     int sum = 0;
     for(int i=0; i<A.length; i++) {
        sum += A[i];
     }
     return sum;
  }
}
```

OLLIR Examples (2)

ex3.lir

```
myClass {
   .method public sum(A.array.i32, B.array.i32).array {
         t1.i32 :=.i32 arraylength($1.A.array.i32).i32;
         C.array :=.array new(array, t1.i32).array;
         i.i32 :=.i32 0.i32;
      Loop:
         t1.i32 :=.i32 arraylength($1.A.array.i32).i32;
         if (i.i32 >=.i32 t1.i32) goto End;
         t2.i32 :=.i32 $1.A[i.i32].i32;
         t3.i32 :=.i32 $2.B[i.i32].i32;
         t4.i32 :=.i32 t2.i32 +.i32 t3.i32;
         C[i.i32].i32 :=.i32 t4.i32;
         i.i32 :=.i32 i.i32 +.i32 1.i32;
         goto Loop;
      End:
         ret.array.i32 C.array.i32;
   }
}
```

```
class myClass {
  public int[] sum(int[] A, int[] B){
    int[] C = new int[A.length];
    for(int i=0; i<A.length; i++) {
      C[i] = A[i] + B[i];
    }
    return C;
  }
}
```

ex4.lir

```
myClass {
    .field private a.i32;
    .construct myClass(n.i32).V {
        invokespecial(this, "<init>").V;
        putfield(this, a.i32, $1.n.i32).V;
    }
    .construct myClass().V {
        invokespecial(this, "<init>").V;
    }
    .method public get().i32 {
        t1.i32 :=.i32 getfield(this, a.i32).i32;
        ret.i32 t1.i32;
    }
    .method public put(n.i32).V {
        putfield(this, a.i32, $1.n.i32).V;
    }
    .method public m1().V {
        putfield(this, a.i32, 2.i32).V;  // this.a = 2;
        t2.String :=.String ldc("val = ").String;
        t1.i32 :=.i32 invokevirtual(this,"get").i32;
        invokestatic(io, "println", t2.String, t1.i32).V;  //io.println("val = ", this.get());
        c1.myClass :=.myClass new(myClass,3.i32).myClass;
        invokespecial(c1.myClass,"<init>").V;  // myClass c1 = new myClass(3);
        t3.i32 :=.i32 invokevirtual(c1.myClass, "get").i32;
        invokestatic(io, "println", t2.String, t3.i32).V; // io.println("val = ", c1.get());
        invokevirtual(c1.myClass, "put", 2.i32).V;  // c1.put(2);
        t4.i32 :=.i32 invokevirtual(c1.myClass, "get").i32;
        invokestatic(io, "println", t2.String, t4.i32).V; // io.println("val = ", c1.get());
    }
    .method public static main(args.array.String).V {
        A.myClass :=.myClass new(myClass).myClass;
        invokespecial(A.myClass,"<init>").V;
        invokevirtual(A.myClass,"m1").V;
    }
}
```

Note: since the Java-- version used this year does not accept Strings, some of the instructions in this exemple are not needed (see next slide with a similar exemple, but without strings) ex4b.lir

```
myClass {
   .field private a.i32;
   .construct myClass(n.i32).V {
      invokespecial(this, "<init>").V;
      putfield(this, a.i32, $1.n.i32).V;
   }
   .construct myClass().V {
      invokespecial(this, "<init>").V;
   }
   .method public get().i32 {
      t1.i32 :=.i32 getfield(this, a.i32).i32;
      ret.i32 t1.i32;
   }
   .method public put(n.i32).V {
      putfield(this, a.i32, $1.n.i32).V;
   }
   .method public m1().V {
      putfield(this, a.i32, 2.i32).V;  // this.a = 2;
      t1.i32 :=.i32 invokevirtual(this,"get").i32;
      invokestatic(io, "println", t1.i32).V;  //io.println(this.get());
      c1.myClass :=.myClass new(myClass,3.i32).myClass;
      invokespecial(c1.myClass,"<init>").V; // myClass c1 = new myClass(3);
      t3.i32 :=.i32 invokevirtual(c1.myClass, "get").i32;
      invokestatic(io, "println", t3.i32).V; // io.println(c1.get());
      invokevirtual(c1.myClass, "put", 2.i32).V;  // c1.put(2);
      t4.i32 :=.i32 invokevirtual(c1.myClass, "get").i32;
      invokestatic(io, "println", t4.i32).V; // io.println(c1.get());
   }
   .method public static main(args.array.String).V {
      A.myClass :=.myClass new(myClass).myClass;
      invokespecial(A.myClass,"<init>").V;
      invokevirtual(A.myClass,"m1").V;
   }
}
```

```
class myClass {
  int a;

  myClass(int n){
    this.a = n;
  }
  myClass(int n){
    this.a = n;
  }
  public int get(){
    return this.a;
  }
  public void put(int n){
    this.a = n;
  }
  public void m1(){
    this.a = 2;
    io.println(this.get());

    myClass c1 = new myClass(3);
    io.println(c1.get());

    c1.put(2);
    io.println(c1.get());
  }
  public static void main(String[] args){
    myClass A = new myClass();
    A.m1();
  }
}
```

OLLIR Examples (4)

ex5.lir

```
myClass {
   .construct myClass().V {
      invokespecial(this, "<init>").V;
   }
   .method public check(A.array.i32, N.i32, T.i32).bool {
         i.i32 :=.i32 0.i32;
         all.bool :=.bool 0.bool;
      Loop:
         t1.bool :=.bool i.i32 <.i32 $2.N.i32;
         t2.i32 :=.i32 $1.A[i.i32].i32;
         t3.bool :=.bool t2.i32 <.i32 $3.T;
         if (t1.bool & .bool t3.bool) goto Body;
         goto EndLoop;
      Body:
         i.i32 :=.i32 i.i32 +.i32 1.i32;
         goto Loop;
      EndLoop:
         if (i.i32 ==.i32 $2.N.i32) goto Then;
         goto End;
      Then:
         all.bool :=.bool 1.bool;
      End:
         ret.bool all.bool;
   }
}
```

```
class myClass {
  public boolean check(int[] A, int N, int T){
    int i = 0;
    boolean all = false;
    while((i < N) && (A[i] < T)) {
      i++;
    }
    if(i == N) all = true;
    return all;
  }
}
```

OLLIR Examples (5)

```java
class Fac {
   public int compFac(int num){
      int num_aux;
      if (num < 1)
         num_aux = 1;
      else
         num_aux = num * (this.compFac(num-1));
         return num_aux;
   }
   public static void main(String[] args){
      io.println(new Fac().compFac(10));
   }
}
```

Fac.lir

```
Fac {
    .method public compFac(num.i32).i32 {
            if ($1.num.i32 >=.i32 1.i32) goto else;
            num_aux.i32 :=.i32 1.i32;
            goto endif;
        else:
            aux1.i32 :=.i32 $1.num.i32 -.i32 1.i32;
            aux2.i32 :=.i32 invokevirtual(this, "compFac", aux1.i32).i32;
            num_aux.i32 :=.i32 $1.num.i32 *.i32 aux2.i32;
        endif:
            ret.i32 num_aux.i32;
    }
    .method public static main(args.array.String).V {
        aux1.Fac :=.Fac new(Fac).Fac;
        invokespecial(aux1.Fac,"<init>").V;
        aux2.i32 :=.i32 invokevirtual(aux1.Fac,"compFac",10.i32).i32;
        invokestatic(io, "println", aux2.i3).V;
    }
}
```

About the OLLIRTool

- Parses OLLIR code and represents the code as Java classes
  - Each method includes a CFG with the OLLIR instructions
  - Each method includes a VarTable (a symbol table of parameters and local variables)
- Current parser does not include error recovery and grammar is permissive
- The parser considers that the OLLIR code has been generated from a compiler and is correct
  - so, it is possible that some OLLIR incorrect code is not identified as incorrect!

From OLLIR code to JVM code (1)

- The JVM is a stack-based machine
  - The instructions of stack-based machines are also known as zero-address code
- The translation of OLLIR code to JVM code is almost direct, with exception of the instructions of each method
  - A translation between three-address code to zero-address code is needed!

From OLLIR code to JVM code (2)

- Non-optimized JVM code – each three-address OLLIR instruction is translated to a sequence of JVM instructions always loading/storing values from/to JVM local variables
  - i.e., without considering that the stack can be very useful to communicate intermediate results

source code:

```
int a, b, c;
…
a = b+c;
```

OLLIR code:

```
a.i32 :=.i32  b.i32 +.i32 c.i32;
```

Symbol Table:

OLLIR var	JVM local var
…	…
b	1
c	2
a	3
…	…

JVM code:

```
iload_1
iload_2
iadd
istore_3
```

From OLLIR code to JVM code (3)

- Non-optimized JVM code – each three-address OLLIR instruction is translated to a sequence of JVM instructions always loading/storing values from/to JVM local variables
  - i.e., without considering that the stack can be very useful to communicate intermediate results

source code:

```
int a, b, c;
…
a = b+c+1;
```

OLLIR code:

```
t1.i32 :=.i32  b.i32 +.i32 b.i32;
a.i32 := t1.i32 +.i32 1.i32;
```

Symbol Table:

OLLIR var	JVM local var
…	…
b	1
c	2
t1	3
a	4
…	…

JVM code:

```
iload_1
iload_2
iadd
istore_3
iload_3
iconst_1
iadd
istore_4
```

JVM code:

```
iload_1
iload_2
iadd
iconst_1
iadd
istore_4
```