Course Unit Compilers

Masters in Informatics and Computing Engineering (MIEIC)

Department of Informatics Engineering

University of Porto/FEUP

2° Semester - 2020/2021

OO-based Low-Level Intermediate representation (OLLIR)

Intermediate representation for the compiler backend

v0.4, April 22, 2021

Table of Contents

Abstract: This document describes the OO-based Low-Level Intermediate representation (OLLIR) to be used in the compiler project. The OLLIR of a class shall be output by the frontend of the compiler, i.e., after parsing, semantic analysis and some possible optimizations at the frontend level. The OLLIR output is then input to the backend of the compiler responsible for optimizations and code generation.

1 Introduction

The backend of the compiler is responsible for the selection of the JVM instructions [1], the assignment of the local variables of methods to the local variables of the JVM, and the generation of JVM code in the *jasmin* format [2].

The backend receives a file representing a class in which each method is in a three-address code (TAC) based format[1]. This TAC format identifies the types of all operations and variables by including the type information after each operation and variable using the following format:

```
VAR.TYPE
OP.PRIMITIVE_TYPE
ASSIGN.TYPE
```

Where:

```
PRIMITIVE_TYPE → i32 | bool

TYPE → PRIMITIVE_TYPE | CLASSNAME | (.array)+.ARRAY_TYPE | String

ARRAY_TYPE → PRIMITIVE_TYPE |String | CLASSNAME
```

And ***i32*** represents the 32-bit signed integer (int), ***CLASSNAME*** represents a class, ***array*** represents a reference to an array, and ***String*** a reference to a String object.

We note that this version of OLLIR only accepts ***int*** and ***boolean*** as primitive types[2].

For example, when considering the input Java statement:

a=b*c;

where all the variables are of type ***int***, the equivalent OLLIR code is:

a.i32 :=.i32 b.i32 * c.i32;

For the statement:

a=b*c+d;

where all the variables are of type ***int***, the equivalent OLLIR code can be:

t1.i32 :=.i32 b.i32 * c.i32; // t1 = b * c

a.i32 :=.i32 t1.i32 + d.i32; // a = t1 + d;

where t1 is an auxiliary variable.

2 OLLIR Basics

OLLIR is inspired in the JVM specification [1] and uses many JVM properties.

Calls to methods can be of one the following:

[1] Actually, the OLLIR includes instructions with multiple operands. Such cases are related with method calls.
[2] The support of the other primitive types, i.e., byte, short, long, float, double, char, would require the use in TAC of, e.g., I16, I8, I64, F32, F64, C.

Type of method invocation	Usage	Parameters
invokedynamic	Invoke dynamic method. Not currently supported in OLLIR	First: object (it can be *this*) Others: arguments of the called method
invokevirtual	Invoke instance method and dispatch based on class	First: object (it can be *this*) Others: arguments of the called method
invokeinterface	Invoke interface method. Not currently used in OLLIR	First: object (it can be *this*) Others: arguments of the called method
invokestatic	Invoke static method	First: class Others: arguments of the called method
invokespecial	Invoke instance method. Special handling for superclass, private, and instance initialization method invocations	First: object (it can be *this*) Others: arguments of the called method

The syntax of the call includes the return type (TYPE), also including the possibility to be void (V), and the arguments of the methods being called.

The accesses to fields of classes use one of the following get and put operations, being ***getfield*** and ***putfield*** for non-static classes and ***getstatic*** and ***putstatic*** for static classes. The following table shows the type of field access and the parameters involved in each one.

Type of field access	Usage	Parameters
getfield	Get field from object	First: object (it can be *this*)
putfield	Set field in object	First: object (it can be *this*)
getstatic	Get static field from class	First: class
putstatic	Set static field in class	First: class

The following are the type of access modifiers for classes, fields, and methods.

Type of method/class/field
public
private
protected
static
final

The examples shown in Figure 1, Figure 2, Figure 3, and Figure 4 illustrate input classes and their representation in OLLIR.

The elements of the OLLIR that start with a '$' are parameters of the method and the number following the '$' identifies the position of the parameter in the method signature from 0 to N-1 (N is the number of parameters of the method) for static methods, and from 1 to N for the other methods (in this later case the parameter 0 is the **this**).

```
class myClass {

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

(a)

```
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;
  }
}
```

(b)

Figure 1. First example of a program with a loop and the sum of all array elements: (a) input language code; (b) OLLIR used;

```
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));
  }
}
```

(a)

```
Fac {
  .construct Fac().V {
    invokespecial(this, "<init>").V;
  }

  .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;
  }
}
```

(b)

Figure 2. Second example of a program with a recursive function: (a) input language code; (b) OLLIR used;

```
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;
  }
}
```

(a)

```
myClass {
  .construct myClass().V {
    invokespecial(this, "<init>").V;
  }

  .method public sum(A.array.i32, B.array.i32).array.i32 {

      t1.i32 :=.i32 arraylength($1.A.array.i32).i32;
      C.array.i32 :=.array.i32 new(array, t1.i32).array.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;
      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;
  }

}
```

(b)

Figure 3. Third example of a program considering the creation of an array: (a) input language code; (b) OLLIR used;

```
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("val = ", this.get());

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

    c1.put(2);
    io.println("val = ", c1.get());
  }

  public static void main(String[] args){
    myClass A = new myClass();
    A.m1();
  }
}
```

(a)

```
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;
  }
}
```

(b)

Figure 4. Fourth example of a program considering the access to class variables: (a) input language code; (b) three-address code used;

```
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;
  }
}
```

(a)

```
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.i32;
      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;
  }

}
```

(b)

Figure 5. Fifth example of a program considering the use of Boolean expressions, WHILE and IF constructs: (a) input language code; (b) OLLIR used;

3 OLLIR Implementation

We provide an implementation of the grammar of OLLIR in JavaCC. Please note that only some verifications are done, since the OLLIR code is expected to be output by a compiler, which we assume will be correct.

In addition, we provide a frontend tool for OLLIR that consists of the complete parser for OLLIR code, Java classes that contain all the information of the OLLIR code, and a control-flow graph for each of the methods in the class.

The tool provided also includes methods to show and get the local variables and parameters of each method.

4 Code Generation for JVM

We present in this section some considerations regarding the translation of OLLIR to JVM instructions. We note that the selection of JVM instructions is a phase to be applied to the OLLIR.

The OLLIR generated by the frontend may use the variable identifiers present in the input language being compiled, and additional identifiers used by the frontend to represent the auxiliary variables needed to store intermediate subexpression values. All the parameters and the local variables of methods need to be assigned to JVM local variables. This can be done by using a symbol table for each method that associates each identifier of a variable to a JVM local variable.

Table I. Translation between some OLLIR elements and the JVM. Examples are in blue.

OLLIR	**JVM in jasmin format**
“<init>”	java/lang/Object."<init>":()V
String	Ljava/lang/String;
boolean	Z
i32	I
myArray.array.i32	[I
myArray.array.array.i32	[[I
myArray.array.String	[Ljava/lang/String;
myArray.array.MyClass	[LMyClass;
Assuming: import org.Class1; myArray.array.Class1	 [Lorg/Class1;
invokevirtual (a function) invokevirtual(c1.myClass, "put", 2.i32).V;	invokevirtual (an instruction) aload_2 *; push into the stack the reference to c1 (assuming in this case that is stored in local variable 2)* iconst_2 *; push the constant 2 into the stack* invokevirtual myClass.put(I)V; *;object c1 and constant 2 must be in stack*
invokespecial (a function) invokespecial(this, "<init>").V;	invokespecial (an instruction) aload_0 *;push this reference to the stack* invokespecial java/lang/Object.<init>()V;
invokestatic (a function) invokestatic(ExMain, “mult”, a1.i32, a2.i32).i32	invokestatic (an instruction) invokestatic ExMain.mult(II)I *;before the two values of variables a1 and a2 must be pushed into the stack*

getfield (a function)
 	
t1.i32 :=.i32 getfield(this, a.i32).i32;
 	
t1.i32 :=.i32 getfield(o1, a.i32).i32;
	getfield (an instruction)
	
	aload_0
	getfield I a
	istore_2 *; assuming that variable t1 is stored in local variable 2*
	
	aload_2 ; assuming the object ref o1 in local variable 2
	getfield I a
	istore_2 *; assuming that variable t1 is stored in local variable 2*

OLLIR	JVM in jasmin format

putfield (a function)
 	
putfield(this, a.i32, *3.i32*).V;
 	
putfield(o1, a.i32, *3.i32*).V;
	putfield (an instruction)
	
	iconst_3
	putfield this/a I
	
	iconst_3
	putfield o1/a I

getstatic (a function) t1.i32 :=.i32 getstatic(MyClass, a.i32).i32;	getstatic (an instruction) getstatic MyClass/a I istore_2 *; assuming that variable t1 is stored in local variable 2*
putstatic (a function) putstatic(MyClass, a.i32, *b.i32*).V;	putstatic (an instruction) iload_2 ; assuming *the value of b* in local variable 2 putstatic MyClass/a I
public	public
private	private
protected	protected
static	static
final	final

new (operands are the class of the object to be created)
 	
// Java example: Fac aux1 = new Fac();
aux1.Fac :=.Fac new(Fac).Fac;
invokespecial(aux1.Fac,"<init>").V;
 	
// Java example: Fac aux1 = new Fac(2);
aux1.Fac :=.Fac new(Fac).Fac;
invokespecial(aux1.Fac,"<init>", 2).V;
	New
	
	new Fac ;create object of class Fac
	dup; duplicate the object reference in the top of the stack
	invokespecial <init>()V ; call constructor
	astore_2 ; pop and store the object reference in local variable 2
	
	new Fac ;create object of class Fac
	dup; duplicate the object reference in the top of the stack
	iconst_2 ; push constant 2 to the stack
	invokespecial <init>(I)V ; call constructor
	astore_2 ; pop and store the object reference in local variable 2

new (operands are the class of the elements, sizes of each dimension and the number of dimensions, returns the reference to the array)
 	
// Java example: int[][][] a3 = new int[5][4][3];
a3.array.array.array.i32 :=.array.array.array.i32 new(array, 5.i32, 4.i32, 3.i32).array.array.array.i32;
	multianewarray (an instruction to create a new multidimensional array)
	(operands are sizes of each dimension and the number of dimensions, returns the reference to the array)
	
	iconst_5 *; push 5 to the stack*
	iconst_4 *; push 4 to the stack*
	iconst_3 *; push 3 to the stack*
	multianewarray [[[I 3 *; create an array of 5x4x3 elements of type int*
	astore_2 *; store the array reference in local variable 2*

OLLIR	JVM in jasmin format
// Java example: String[][] a2 = new String[5][4]; a2.array.array.String :=.array.array.String new(array, 5.i32, 4.i32).array.array.String;	iconst_5 *; push 5 to the stack* iconst_4 *; push 4 to the stack* multianewarray [[Ljava/lang/String; 3 *; create an array of 5x4 elements of type String* astore_2 *; store the array reference in local variable 2*

new (operands are the type of the elements, the number of elements, returns the reference to the array)
 	
// Java example: int[] a1 = new int[4];
a1.array.i32 :=.array.i32 new(array, i32, 4.i32).array.i32;
	newarray (an instruction to create a 1D array of primitive types)
	(operand is the number of elements, returns the reference to the array)
	
	iconst_4 *; push 4 to the stack*
	newarray int *; create an array of 4 elements of type int*
	astore_2 *; store the array reference in local variable 2*

new (operands are the class of the elements, the number of elements, returns the reference to the array)
 	
// Java example: MyClass[] a1 = new MyClass [4];
a1.array.MyClass :=.array.MyClass new(array, 4.i32).array. MyClass;
	anewarray (an instruction to create a 1D array of reference)
	(operand is the number of elements, returns the reference to the array)
	
	iconst_4 *; push 4 to the stack*
	anewarray MyClass *; create an array of 4 elements of type MyClass*
	astore_2 *; store the array reference in local variable 2*

arraylength (function having as input the array reference and returns the size of the respective dimension of the array) t1.i32 :=.i32 arraylength($1.A.array.i32).i32;	arraylength (instruction that assumes that the array reference is on top of the stack and pushes to the stack the size of the respective dimension of the array) aload_2 *; push into the stack the reference of myArray 1 dimensional array (assuming in this case that it is stored in local variable 2)* arraylength istore_3 *; pop and store value in local variable 3*
Return an int value from method ret.i32 myVar.i32;	ireturn (assumes the top of stack with the int value to return, i.e., it requires that previously the value of myVar has been pushed to the stack) iload_2 *; push into the stack the value of myVar (assuming in this case that it is stored in local variable 2)* ireturn *; return the int in the top of the stack*
Return a Boolean value from method ret.bool myVar.bool;	ireturn (assumes the top of stack with the int value representing the Boolean value to return, i.e., it requires that previously the value of myVar has been pushed to the stack) iload_2 *; push into the stack the value of myVar (assuming in this case that it is stored in local variable 2)* ireturn
Return a reference from method ret.myClass myVar.myClass;	areturn (assumes the top of stack with the reference to return)

OLLIR	JVM in jasmin format
ret.array.i32 myArray.array.i32;	aload_2 *; push into the stack the reference of myVar object (assuming in this case that it is stored in local variable 2)* areturn aload_2 *; push into the stack the reference of myArray 1 dimensional array (assuming in this case that it is stored in local variable 2)* areturn
Return from void method ret.V;	return (return from method) return

if (…) goto Label;
 	
if ($1.num.i32 >=.i32 1.i32) goto else;
	If<cond> Label (Branch if `int` comparison with zero succeeds): ifeq, ifne, iflt, ifge, ifgt, ifle
	
	if_icmp<cond> Label (Branch if `int` comparison succeeds): if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple
	
	both types of conditional branches require the operands in the stack
	
	iload_1 *; push into the stack the value of num, the first parameter of the method it is stored in local variable 1)*
	iconst_1 *; push into the stack the value 1*
	if_icmpge else *; compare the two values in the top of the stack and if "ge" then jump to the instruction immediately preceded by label "else"*

goto Label;	goto Label
Label:	Label:

5 Optimizations

The register allocation phase can be implemented at the OLLIR level. In this case, it constrains the maximum number of JVM local variables to be used in each method. In case it is not possible to compile using a specific maximum number of local variables, the compiler must not generate JVM code and must report the impossibility to compile with that number of local variables.

The templates used for loops shall be applied before the generation of the OLLIR (in this case the OLLIR code outputted by the previous compiler stage (fronted) is already according to the target implementation of loops) instead of transforming the OLLIR.

References

[1] The Java Virtual Machine Specification, http://java.sun.com/docs/books/jvms/
[2] Jasmin Home Page, http://jasmin.sourceforge.net/