Compilation: PA4
Fall 2014

Ice Coffee To LIR: Documentation

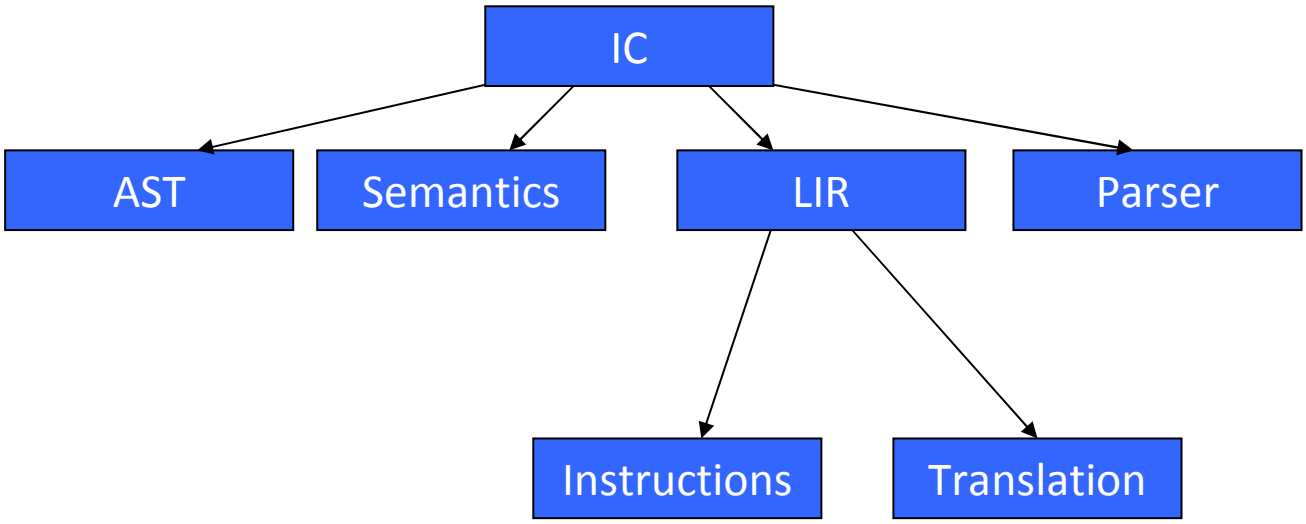
Ori Lentzitzky	200446094
Idan Shaby	200789691
Guy Engel	300455672

Table Of Contents

Package Structure

In PA4, we've built upon the previous assignment (PA3), and placed our new code with in the *LIR* sub-package.

The following diagram describes the hierarchy of the project:



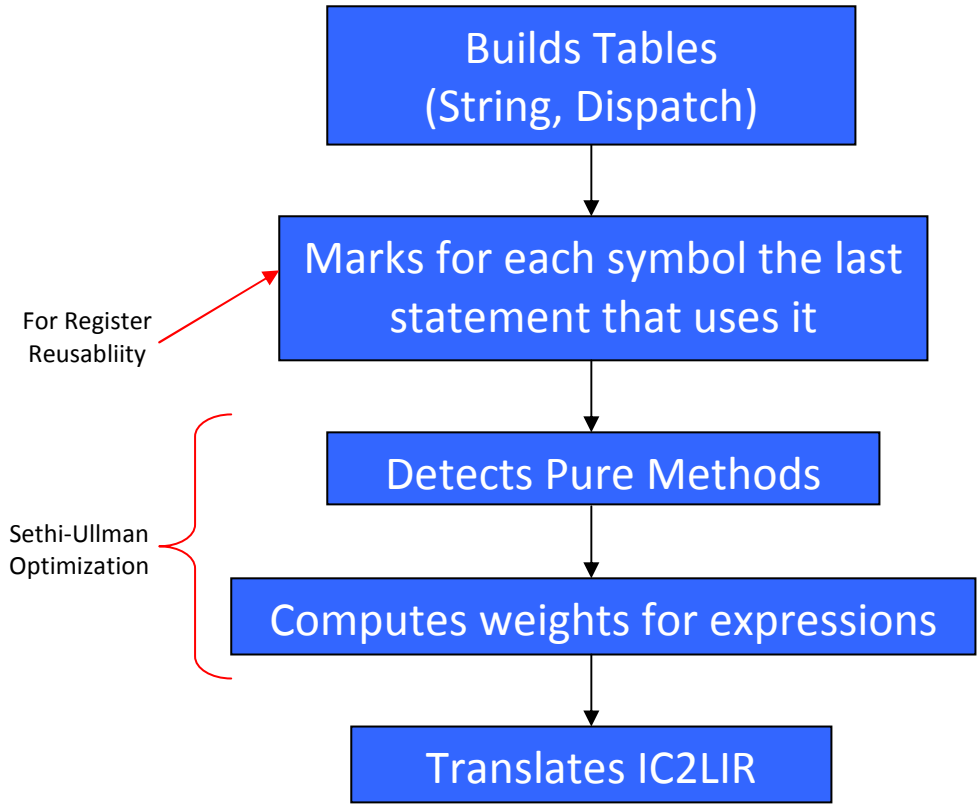
LIR.Instructions contains a low-level description of LIR objects. In a way, we are buliding a "LIR AST" from the IC AST. Each instruction is represented with an appropriate object, linking to the AST node which created it.

LIR.Translation does all the heavy lifting of the translation itself (including preparations, printing and data structures which support the translation).

LIR is the main package. It mostly handles the LIR translation process itself (described next).

Translation Process

The following diagram shows the process that *LIRTranslationProcess* class executes when calling its' *run* method:



String Literal Sets

For each string literal in the AST, we store an element in a set for it. We use this set to print the list of strings that will be used in the LIR code.

We keep a different element for each string, since the IC specificiation clearly states that comparing two strings via equality operator (==) does not check for the string contents. **However, the LIR simulator seems to always equate strings by their character representation, not their addresses**.

An exception to the above are runtime error messages (only one created for each kind, no matter how many LIR nodes may print it).

The LIR Simulator does not support escape characters. For that matter, we use the AST's *toFormattedString* method to print out the string tables. That way, the escape characters are present (i.e. \n instead of an actual new line) and the simulator can read the file successfully (since it considers a new line within a string an error).

Dispatch Tables

For each object (a.k.a class type), we build a dispatch table.

Dispatch tables consist of:

- Name of table (_DV_Name)
- Offsets of the object's fields (where 0 is the DV_Name address)
- Offsets of the ***virtual*** methods' fields
- A reference to the super class dispatch table, if extends any.

Method Overrides

For class B, that extends class A, A's fields and methods are given the first offset ordinals in the dispatch table of B.

That is true even if class B overrides any methods.

Example:

```
class A {
      void foo1() { ... }
      void foo2() { ... }
      void foo3() { ... }
      void foo4() { ... }
}

Class B extends A {
      void nofoo() { ... }
      void foo2() { ... }
      void bar() { ... }
}
```

B's dispatch table will be in the following order (first line is offset 0):

- foo1 Address of A's *foo1* method
- foo2 Address of B's *foo2* method
- foo3 Address of A's *foo3* method
- foo4 Address of A's *foo4* mehtod
- nofoo Address of B's *nofoo* method
- bar Address of B's *bar* method

Symbol's Last Used in Statement

In order to faciliate with register reusability, we mark for each symbol the last statement which uses its' value. After that statement, we can consider its' register dead and dispose of it (i.e, use it again shortly).

Example:

```
1     int foo(int a, int b, int c, int d) [
2           int x = a;
3           x = a + b;
4           c = 3 – x;
5           return x + a*2;
6     }
```

a's last statement is return statement in line 5.
b's last statement is return statement in line 3.
c's last statement is return statement in line 4.
d's last statement is null (never used).
x's last statement is return statement in line 5.

We also maintain a "used in last expression" information for very specific occurances (for example, for if condition; a symbol which is last used in the condition portion of the if statement, can be reused in the operation portion of the statement).

It should be noted that just because we mark a symbol as being used, does not garantuee we'd later consider it a register. If we only need to read from it, we might just use it as a *memory* reference (meaning, use it by its' variable name). However, once we assigned it to a register, we'd look to the "last statement used" to see when to dispose of it.

It should also be noted that during translation, LIR may use many registers which cannot be foreseen in advance (for instance, to store an intermediate calculation value). Those registers are disposed at the translation process discretion.

Fields

An object's field could be "used" in more than one of the object's method. Therefore, storing a "last symbol used" for fields would be irrelevent and distrubtive. For that reason, for fields we can a "last used in statement" for every method in which the field is used / referenced.

Classes / User Types

Class objects (user type objects) are treated the same way. Since we may need to use a DV's offset at certain points of the code, we attempt to store it for each method for the last possible need (however are willing to relax this constraint, since we can retrieve the DV's addresses relatively easy during execution, especially the *this* "reference").

Instances in which we'd need the access the DV_PTR of the object (access an offset):

- Calls to virtual methods
- Assignments to field

The first point is fairly trivial – each call to a virtual method is done solely via the offset. The second point is a bit trickier. Look at the following code, for example:

```
class A {
      int x;

      void foo(int a) {
            int y = x / 2;
            x = a + 1;
            ...
            Library.printi(x);
      }
}
```

On the surface, it would appear x is used lastly by the printi call, and therefore we need x alive until then, but the DV_PTR is no longer needed once we'd read x's value.

However, since we're *changing* x's value and want any future reference to x in this or any other method to have the correcr value, we also must store the DV_PTR until such time as its' last field is assigned to, not just when it is used to call virtual methods or read field values (that were not read so far).

Moreover, each method maintains that it must write back any field it has written to in register form, back to the field, so other methods could read the updated and correct value of the field!

While Loops

Consider the following code:

```
while (x + y < 10)
      x = x + 1;
```

It may appear as if x is last used during the while statement operation and y is last used during the while statement condition. That would be wrong. Since we must assume there could be several iterations of the loop, we must consider any symbol used prior to the loop to be used **by the entire loop statement – condition and body together!**

If we were to assume y is last used in the condition, then the next iteration will attempt to check the condition only to find no record of y. Furthermore, if we dispose of x after running the statement within the operation (consider *that* statements as x's last used in statement), we would not be successful in running subsequent iterations or condition checks.

This because much more taxing when dealing with nested loops:

```
while (x + y < 10)
      while (z > 5)
            x = x + z;
```

we cannot dispose of z after the inner loop finishes, because there could still be another iteration of the outer loop which might want to use it.

➔ While loops cause serious damage to our ability to reuse registers!

Pure Methods Detection

We consider a method to be pure if it has no side-effects. Meaning if it:

- Never assigns values to any fields.
- Assigns values to formals **only** if they are primitive and are not arrays (int, string, boolean).
- Performs calls to other methods that are pure as well.
- Does not perform any print or read of any kind.

A method which reads a field's value, for instance, but does not change its' value throughout its' run is still considered pure.

As a rule, we do not garantuee we'd detect all pure methods as such. We do garanturee **no impure method would be detected as pure**.

The reason we do not garantuee that is because of the complexity in detecting that subsequent calls are in fact pure. Not all methods which a specific method calls may have been checked and marked as pure yet, for instance.

To avoid overhead of detecting circularity and repeated checks of methods with undetermined call statements, we simply perform a single run and attempt to find breaches in the pureness definition. Any doubt is no doubt – we only mark a method as pure when we're 100% sure.

Examples for **pure** methods:

```
int x;

int foo1(int a) {
      return a;
}

int foo2(int a, int b) {
      int c = a*a + b*b;
      return x + c;
}

int foo3(int a) {
      return foo1(a);
}
```

Examples for **impure** methods:

```
int x;

void foo1(int a) {
      x = a;
}

void foo2(int a, int b) {
      int c = someFoo(a, b); //don't know if someFoo is pure
      return x + c;
}

void foo3(int[] a) {
      a[0] = x;
}

void foo4(A a) {
      a.x = x;
}
```

Library Methods

As a whole, we consider all library methods to be impure.

External Methods

External Methods (i.e. a.foo()) are considered to be impure.

Sethi-Ullman Weight Assignment

We assign each expression a weight as best we can, where a negative weight alerts that we cannot use Sethi-Ullman optimization on that expressions since it has side-effects (involves a call to an impure method).

We attempt to give weights to both math expressions and logical expression (that are **not** logical-and or logical-or expressions).

Translation to LIR

RegisterPool

We maintain a pool of registers to assist us in writing the LIR code.

The pool is comprised of a Sorted Set of integers. At first, the set is empty. Every time we require a register, the pool checks the set. If empty, it generates a new register (i.e. a new integer). When we're done with a register, we put it back in the pool and its' integer is added to set. We always remove the smallest integer from the pool when we give an existing register.

```
private static int counter = 0;
private static SortedSet<Integer> pool;

 public static Register get() {

	if (!pool.isEmpty()) {
		Register reg = new Register(pool.first());
		pool.remove(pool.first());
		return reg;
	}

	return new Register(node, ++counter);
}

public static void putback(Register reg) {
	pool.add(reg.getNum());
}
```

The actual pool is a bit more complex, but the aboved code is close enough to the implementation without adding too many complex explanations.

The pool is "flushed" at the end of each method translation (meaning, all registers are returned to the set and are considered unused).

LabelMaker

Each label is prefixed with the object (class) and method in which it appears. i.e, every label in A's doSomething method is prefixed with *_A_doSomething*.

In order to avoid creating the same label twice in the code, we maintain a mapping of all labels used and the number of times they were used. Whenever we try to create the same label twice (for example, because of an if / while / need of a semantic check / etc), we check if we have already used it. If so, we simply add a number from the counter at its' end. i.e if we've already used *_A_doSomething_while_label*, then we'd simply use the label *_A_doSomething_while_label*1 instead.

Method Labels

Labels announcing the start of a method have a unique name.

For virtual methods, that name corresponds to the name given to the method in the dispatch table of the object.

For static methods, we add a suffix "_static" to the label to avoid a clash when the object has two methods with the same name – one virtual, one static. Since IC does not allow method overloading, no label with the same name is created twice, especially for methods.

Unused Variable Declaration

We do not translate "useless" code from IC to LIR. Meaning, variable declarations that are never used. For example:

```
int k = 5;
```

No code is generated for that statement, if k is never used again. Code will be generated if k is used, even if without a real purpose:

```
int k = 5;
k = k + 1;
```

Meaning, we're only looking to see if a variable is declared but never used (read/written). Once declared and used in any way, we do generate code for its' declaration.

An excpetion to the above, is if the initialization includes a call to an **impure** method.

```
int k = 2 + pure();
```

won't be translated to LIR, but:

```
int k = impure() + 2*pure();
```

for example, will be translated. That is because the programmer expects the impure method to be called, and thus may expect / count on any side effect.

Conditions' Negation

There's nothing to special about the way we translated conditions, however we did play a little trick with the negation of conditions.

First let us look at the following condition:

```
if ((a < b) || (c == d))
```

The translation of the code if fairly simple (check first condition. If true, jump to instructions. If false, check second condition. If true, continue to instructions. If false, jump to end of instructions).

Suppose now the condition was:

```
if (!((a < b) || (c == d)))
```

We could calculate the condition and then flip it. However, that's easier said than done, since the condition does not have a "value", but has a series of jump decision.

We implement this using De-Morgan's Law:

Not(a OR b) = Not(a) AND Not(b)
Not(a AND b) = Not(a) OR Not(b)

Also, we simply take the complementary case of the simpler conditions. For example:

Not(a < b) ➔ a >= b
Not(c == d) ➔ c != d

And so on.

So actually, for the condition above we are translating it as if it were:

```
if (!(a < b) && !(c == d))
if ((a >= b) && (c != d))
```

What to do when the condition is:

```
if ((a < b) || !(c == d))
```

then we simply translate it as if it were

```
if ((a >= b) && (c == d))
```

Meaning, we ignore double negations.

Moreover, for:

```
if (!((a < b) && ((c == d) || !(b > c))))
```

we look it as if it were:

```
if ((a >= b) || ((c != d) && (b > c)))
```

And so on…

Array Allocation

1-dimensional array allocations are simply done using the library __*allocateArray* method.

However, n-dimensional arrays are much more complex (n > 1).

For example: new `arr[size1][size2][size3]` will require us to:

- Allocate 1 array of size size1.
- Allocate size1 arrays of size size2 and place their addresses within the respected elements of the first array.
- Allocate size2 arrays of size size3 and place their addresses in the 2nd array's cells.

This means a simple new array statement might be generated into a large LIR code.

Note, for multidimensional arrays we perform the runtime check of **__checkSize** before allocating any of the arrays (meaning, we check all sized are legal before we allocate memory).

Testing

We wrote several test suites to test our translation (and its' run on the LIR Simulator).

We will give a few of them here (usually only in IC form; LIR code is big and heavy).

Arrays & Objects Test

The following tests that two separate objects are indeed created in an array of objects:

```
class Bar {
      Bar[] bars;
      int x;

      void foo() {
            bars = new Bar[2];
            bars[0] = new Bar();
            bars[0].x = 1;
            bars[1] = new Bar();
            bars[1].x = 4;
            bars[0].foo2();
            bars[1].foo2();
      }

      void foo2() {
            Library.printi(x);
      }
}

class Main {

      static void dosomething(Bar bar) {
            bar.foo();
      }

      static void main(string[] args) {
            Bar b = new Bar();
            dosomething(b);
      }

}
```

LIR statistics:

```
========================================
#instructions:          33
#labels:                4
#registers (total):     43
#registers (different): 3
#variables:             5
#strings:               0
#dispatch tables:       1
```

Override & Inheritance Test

Checks we are actually executing overriden method:

```
class A {
	int foo(int a, int b) {
		return (a + b);
	}
	int foo2(int a) {
		return 2*a;
	}
}

class B extends A {

	void myfoo() { }

	int foo2(int a) {
		return a*a;
	}
}

class Main {

	static void main(string[] args) {
		A a = new B();
		int y = a.foo2(8);
		Library.printi(y);
		a = new A();
		y = a.foo2(8);
		Library.printi(y);
	}
}
```

LIR Statistics:

```
================================================
#instructions:          23
#labels:                4
#registers (total):     25
#registers (different): 4
#variables:             8
#strings:               0
#dispatch tables:       2
```

Sethi-Ullman Test

This tests that the execution is done by the less-weighted subtree first (and uses a pure method, as well; changing said method to be impure causes the entire expression to be calculated regularly with no regards to weights):

```
class A {

      int x;
      int y,z;

      int foo(int a, int b, int c, int d, int e) {
            return (a+b*c)-(((d*x)-foo2())*z + d*a);
      }

      int foo2() {
            return y;
      }
}

class Main {

      static void main(string[] args) {
            A a = new A();
            a.x = 1;
            a.y = 2;
            a.z = 5;
            Library.printi(a.foo(1, 2, 3, 4, 5));
      }
}
```

LIR code produced (we can see Sethi-Ullman worked!):

```
Move this,R1
MoveField R1.1,R2
Move R2,R3
Mul d,R3
Compare 0,R1
VirtualCall R1.1(),R2
Sub R2,R3
MoveField R1.3,R4
Mul R4,R3
Move a,R4
Mul d,R4
Add R4,R3
Move c,R5
Mul b,R5
Move R5,R6
Add a,R6
Sub R3,R6
```

LIR Statistics:

```
========================================
#instructions:          55
#labels:                8
#registers (total):     54
#registers (different): 7
#variables:             20
#strings:               1
#dispatch tables:       1
```

Matrix: Multiplications Table

```
class Matrix {

      int[][] mat;
      int n;

      void createMatrix() {
            mat = new int[n][n];
            int i = 0;
            while (i < n) {
                  int j = 0;
                  while (j < n) {
                        mat[i][j] = (i+1)*(j+1);
                        j = j + 1;
                  }
                  i = i + 1;
            }
      }

      void printMatrix() {
            int i = 0;
            while (i < n) {
                  int j = 0;
                  while (j < n) {
                        Library.printi(mat[i][j]);
                        Library.print(" ");
                        j = j + 1;
                  }
                  Library.println("");
                  i = i + 1;
            }
      }

      static void main(string[] args) {
            Matrix m = new Matrix();
            m.n = 10;
            m.createMatrix();
            m.printMatrix();
      }

}
```

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

LIR statistics:

```
=======================================
#instructions:          132
#labels:                22
#registers (total):     114
#registers (different): 8
#variables:             41
#strings:               4
#dispatch tables:       1
```

Quicksort

```
class Quicksort {
    int[] a;

    int partition(int low, int high) {

    int pivot = a[low];
    int i = low;
    int j = high;
    int tmp;

    while (true) {
        while (a[i] < pivot) i = i+1;
        while (a[j] > pivot) j = j-1;

        if (i >= j) break;

        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
        i = i+1;
        j = j-1;
    }
    return j;
    }

    void quicksort(int low, int high) {

    if (low < high) {
        int mid = partition(low, high);
        quicksort(low, mid);
        quicksort(mid+1, high);
    }
    }

    void initArray() {
      a[0] = 9;
      a[1] = 8;
      a[2] = 2;
      a[3] = 5;
      a[4] = 0;
      a[5] = 6;
      a[6] = 7;
      a[7] = 1;
      a[8] = 3;
      a[9] = 4;
    }

    void printArray() {
    int i = 0;
    Library.print("Array elements: ");
```

```
    while(i<a.length) {
        Library.printi(a[i]);
        Library.print (" ");
        i = i+1;
    }
    Library.print("\n");
    }
}

class Main {

    static void main(string[] args) {
      int n = 10;
      Quicksort s = new Quicksort();
      s.a = new int[n];

      s.initArray();
      s.printArray();
      s.quicksort(0, n-1);
      s.printArray();
    }
}
```

```
Array elements: 9 8 2 5 0 6 7 1 3 4 \n
Array elements: 0 1 2 3 4 5 6 7 8 9 \n
```

LIR Statistics:

```
==============================================
#instructions:          95
#labels:                15
#registers (total):     103
#registers (different): 7
#variables:             34
#strings:               3
#dispatch tables:       1
```