

## Compilation Fourth Step: Intermediate Representation (IR)

Flattening the AST to a sequence of instructions

December 17, 2018

# IR properties

- **Independent** of the source language

clang(C/CPP)	→	
flang(Fortran)	→	
ghc(Haskell)	→	LLVM IR
llgo(Go)	→	
...	→	

- **Independent** of the target language

	→	x86
	→	ARM
LLVM IR	→	WebAssembly
	→	Mips
	→	...

- Contains the **entire** information needed for final translation

# IR of Industrial Compilers

## ► LLVM IR

```
oren@oren: ~/GIT/COMPILATION_TAU_FOR_STUDENTS/FOLDER...  
File Edit View Search Terminal Help  
$ cat example_01.c  
#include <stdio.h>  
int foo(int x,int y,int z)  
{  
    return x+y+z;  
}  
int main(int argc, char **argv)  
{  
    printf("%d\n",foo(1,2,3));  
    return 0;  
}  
$ clang -c -emit-llvm example_01.c  
$ llvm-dis example_01.bc  
$ sed -n '7,22p;23q' example_01.ll  
; Function Attrs: nounwind uwtable  
define i32 @foo(i32 %x, i32 %y, i32 %z) #0 {  
entry:  
    %x.addr = alloca i32, align 4  
    %y.addr = alloca i32, align 4  
    %z.addr = alloca i32, align 4  
    store i32 %x, i32* %x.addr, align 4  
    store i32 %y, i32* %y.addr, align 4  
    store i32 %z, i32* %z.addr, align 4  
    %0 = load i32* %x.addr, align 4  
    %1 = load i32* %y.addr, align 4  
    %add = add nsw i32 %0, %1  
    %2 = load i32* %z.addr, align 4  
    %add1 = add nsw i32 %add, %2  
    ret i32 %add1  
}
```

# IR of Industrial Compilers

## ► GCC's IR (GIMPLE)

```
oren@oren: ~/GIT/COMPILATION_TAU_FOR_STUDENTS/FOLDER...  
File Edit View Search Terminal Help  
$ cat example_02.c  
#include <stdio.h>  
int foo(int n)  
{  
    int sum=17;  
    for (int i=0;i<n;i++)  
    {  
        sum=sum+i+46;  
    }  
    return sum;  
}  
int main(int argc, char **argv) {  
    return printf("%d\n",foo(argc));  
}  
$ rm *.gimple && gcc -O0 -c -fdump-tree-all example_02.c  
$ cp *.gimple example_02.gimple && rm example_02.c.*  
$ sed -n '1,22p;23q' example_02.gimple  
foo (int n)  
{  
    int D.2261;  
    int sum;  
  
    sum = 17;  
    {  
        int i;  
  
        i = 0;  
        goto <D.2254>;  
    <D.2253>:  
        _1 = sum + i;  
        sum = _1 + 46;  
        i = i + 1;  
    <D.2254>:  
        if (i < n) goto <D.2253>; else goto <D.2255>;  
    <D.2255>:  
    }  
    D.2261 = sum;  
    return D.2261;  
}
```

# IR of Industrial Compilers

## ► (MONO) C# CIL

```
oren@oren: ~/GIT/COMPILATION_TAU_FOR_STUDENTS/FOLDER_1_TIRGULIM...
File Edit View Search Terminal Help
$ cat example_03.cs
using System;
namespace ARITH
{
    class MUL
    {
        static int foo(int x, int y, int z)
        {
            int t=77;
            return x*y+z+t;
        }
        static void Main(string[] args)
        {
            foo(1,2,foo(3,4,5));
        }
    }
}

$ mcs -debug example_03.cs
$ monodis example_03.exe > example_03.cil
$ sed -n '67,83p;84q' example_03.cil
        default void Main (string[] args) cil managed
        {
            // Method begins at RVA 0x2077
            .entrypoint
            // Code size 18 (0x12)
            .maxstack 8
            IL_0000: nop
            IL_0001: ldc.i4.1
            IL_0002: ldc.i4.2
            IL_0003: ldc.i4.3
            IL_0004: ldc.i4.4
            IL_0005: ldc.i4.5
            IL_0006: call int32 class ARITH.MUL::foo(int32, int32, int32)
            IL_000b: call int32 class ARITH.MUL::foo(int32, int32, int32)
            IL_0010: pop
            IL_0011: ret
        } // end of method MUL::Main
```

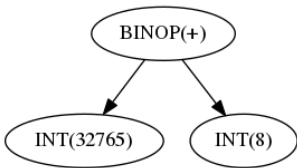
# IR of Industrial Compilers

## ► Java Bytecode

```
oren@oren: ~/GIT/COMPILATION_TAU_FOR_STUDENTS/FOLDER_1/TIRGULIM/SLIDES_04_IR/EX...  
File Edit View Search Terminal Help  
$ cat example_04.java  
public class example_04  
{  
    public static void foo(String s)  
    {  
        if (s.charAt(14+s.length()) == 'a')  
        {  
            s = "MMM";  
        }  
    }  
    public static void main(String[] args)  
    {  
        foo(args[0]);  
    }  
}  
$ javac example_04.java  
$ javap -l -c example_04.class > example_04.bytecode  
$ sed -n '11,23p;24q' example_04.bytecode  
public static void foo(java.lang.String);  
    code:  
      0: aload_0  
      1: bpush      14  
      3: aload_0  
      4: invokevirtual #2          // Method java/lang/String.length:()I  
      7: ladd  
      8: invokevirtual #3          // Method java/lang/String.charAt:(I)C  
     11: bpush      97  
     13: if_icmpne  19  
     16: ldc        #4              // String MMM  
     18: astore_0  
     19: return
```

## IR Introductory Example: 32765+8

- ▶ IR is produced by scanning the AST recursively as follows:
  - ▶ First, the left subtree (a leaf actually) is scanned, producing the IR command: **li Temp\_29, 32765**.
  - ▶ Then, the right subtree (a leaf too) is scanned, producing the IR command: **li Temp\_30, 8**.
  - ▶ Finally, the binop father node uses the temporaries returned from its operand sons to produce the IR command: **add Temp\_31, Temp\_29, Temp\_30**.



- ▶ Note that the IR recursive scan of the AST resembles the scan performed by the semantic analyzer. However here expression subtrees return their temporary, not their type.

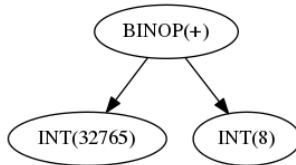
## IR Example: **if (2<6) { PrintInt(3); }**

- ▶ IR is produced by scanning the AST recursively as follows:
  - ▶ First, the condition subtree is scanned, producing the IR commands:
    - ▶ **li Temp\_74, 2**
    - ▶ **li Temp\_75, 6**
    - ▶ **li Temp\_76, 1**
    - ▶ **blt Temp\_74, Temp\_75, label\_cond\_end**
    - ▶ **li Temp\_76, 0**
    - ▶ **label\_cond\_end**
  - ▶ Then, the if-father-node uses the temporary returned from its condition-son and wraps the IR commands produced by its body-son as follows:
    - ▶ **beq Temp\_76, 0, label\_if\_end**
    - ▶ **li Temp\_77, 3**
    - ▶ **call PrintInt( Temp\_77 )**
    - ▶ **label label\_if\_end**

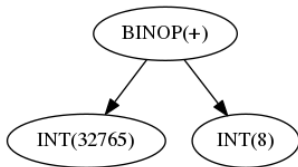


# IR in our project

- ▶ Designing a good IR is more art than science.
- ▶ Specially true in our project where there's only one source language (Poseidon) and one target language (MIPS).
- ▶ In fact, do we even *need* an IR in our project? **Why not translate directly AST  $\rightarrow$  MIPS?**
- ▶ For example, how should we *really* translate  $32765+8$ ? (remember that addition is done with 16 bits overflow).
  - ▶ Should we handle overflow in AST  $\rightarrow$  IR phase?
  - ▶ Or should we handle it in the IR  $\rightarrow$  MIPS phase?



## IR in our project :: Overflow handled in IR $\rightarrow$ MIPS

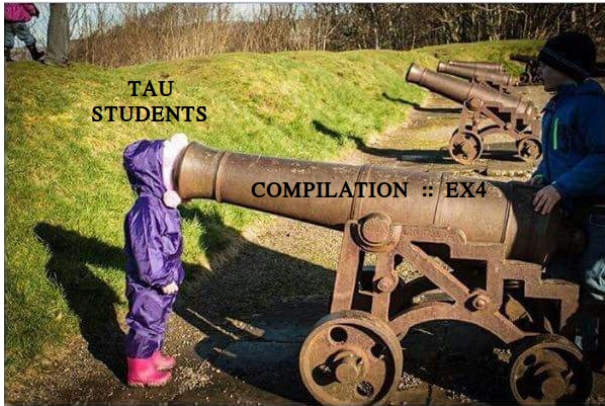


- ▶ Handling arithmetic overflow in the IR  $\rightarrow$  MIPS phase will yield the following (simple) IR code for the addition above:
  - ▶ **li Temp\_29, 32765**
  - ▶ **li Temp\_30, 8**
  - ▶ **add Temp\_31, Temp\_29, Temp\_30**
- ▶ What are the benefits of a simpler IR? How will the add instruction be translated to MIPS eventually?

## IR in our project :: Overflow handled in AST $\rightarrow$ IR

- ▶ Handling arithmetic overflow in the AST  $\rightarrow$  IR phase will yield the following IR code for the addition above:
  - ▶ **li Temp\_29, 32765**
  - ▶ **li Temp\_30, 8**
  - ▶ **add Temp\_31, Temp\_29, Temp\_30**
  - ▶ **li Temp\_32, 32767**
  - ▶ **li Temp\_33, -32768**
  - ▶ **bgt Temp\_31, Temp\_32, label\_overflow**
  - ▶ **blt Temp\_31, Temp\_33, label\_underflow**
  - ▶ **# What should we write here?**
  - ▶ **label\_overflow:**
  - ▶ **# and here?**
  - ▶ **label\_underflow:**
  - ▶ **# and here too?**
  - ▶ **label\_end:**

## IR in our project :: Next Steps



- ▶ How to handle local variables? function input parameters? class data members?
- ▶ How to handle calls to global functions? calls to class methods? calls to library functions (like PrintInt)?