Compilation Fourth Step: Intermediate Representation (IR)

Flattening the AST to a sequence of instructions

December 21, 2018

IR properties

- **Independent** of the source language

clang(C/CPP)	$\rightarrow$	
flang(Fortran)	$\rightarrow$	
ghc(Haskell)	$\rightarrow$	LLVM IR
llgo(Go)	$\rightarrow$	
...	$\rightarrow$	

- **Independent** of the target language

	$\rightarrow$	x86
	$\rightarrow$	ARM
LLVM IR	$\rightarrow$	WebAssembly
	$\rightarrow$	Mips
	$\rightarrow$	...

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

IR of Industrial Compilers :: LLVM Bitcode
Global variables handled *similarly* in IR and ASM

```
$ cat example_01.c
int x;
int y;
int z;
int w;

int main()
{
        x = 5;
        y = 6;
        z = 7;
        w = 8;

        return x+y+z+w;
}
$ clang -c -emit-llvm example_01.c
$ opt -instnamer -o example_01.bc example_01.bc
$ llvm-dis example_01.bc
$ sed -n '5,27p;28q' example_01.ll
@x = common global i32 0, align 4
@y = common global i32 0, align 4
@z = common global i32 0, align 4
@w = common global i32 0, align 4

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 5, i32* @x, align 4
  store i32 6, i32* @y, align 4
  store i32 7, i32* @z, align 4
  store i32 8, i32* @w, align 4
  %tmp = load i32, i32* @x, align 4
  %tmp1 = load i32, i32* @y, align 4
  %add = add nsw i32 %tmp, %tmp1
  %tmp2 = load i32, i32* @z, align 4
  %add1 = add nsw i32 %add, %tmp2
  %tmp3 = load i32, i32* @w, align 4
  %add2 = add nsw i32 %add1, %tmp3
  ret i32 %add2
}
```

- declarations (red)
  - default value 0
- stores (blue)
  - name based access
- loads (how many?)
  - name based access
- temps (how many?)
  - tmp,tmp1,tmp2,...
  - add,add1,add2,...
  - the more the marrier?

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
```

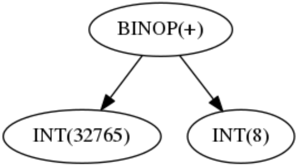
IR of Industrial Compilers

- 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: bipush        14
       3: aload_0
       4: invokevirtual #2                  // Method java/lang/String.length:()I
       7: iadd
       8: invokevirtual #3                  // Method java/lang/String.charAt:(I)C
      11: bipush        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**.



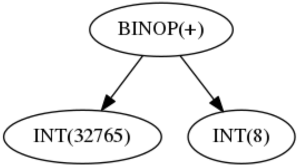
- Note that the IR recursive scan of the AST resembles the scan performed by the semantic analyzer. However here expression subtrees return their temporary, not their type.

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

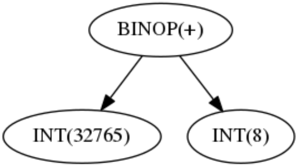
- IR is produced by scanning the AST recursively as follows:
  - First, the condition subtree is scanned, producing the IR commands:
    - **li Temp_74, 2**
    - **li Temp_75, 6**
    - **li Temp_76, 1**
    - **blt Temp_74, Temp_75, label_cond_end**
    - **li Temp_76, 0**
    - **label_cond_end**
  - 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 → IR

- Handling arithmetic overflow in the AST → 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)?