Computer Languages

Intermediate Representation - LLVM

Verónica Gaspes
www2.hh.se/staff/vero

Högskolan Halmstad

Center for Research on Embedded Systems
School of Information Science, Computer and Electrical Engineering

The compiler

After scanning, parsing and type-checking the compiler has

1. understood that the program text is meaningful (it makes sense as a program),
2. built a representation of the program as a data structure (the abstract syntax tree).

An executable has to be generated that has the same meaning as the source

How do we get there?


```
7F 45 4C 46  01 01 01 00  00 00 00 00  00 00 00 00
02 00 03 00  01 00 00 00  B0 D5 04 08  34 00 00 00
8C 10 0E 00  00 00 00 00  34 00 20 00  08 00 28 00
1F 00 1C 00  06 00 00 00  34 00 00 00  34 80 04 08
34 80 04 08  00 01 00 00  00 01 00 00  05 00 00 00
04 00 00 00  03 00 00 00  34 01 00 00  34 81 04 08
34 81 04 08  13 00 00 00  13 00 00 00  04 00 00 00
01 00 00 00  01 00 00 00  00 00 00 00  00 80 04 08
00 80 04 08  60 7A 0D 00  60 7A 0D 00  05 00 00 00
00 10 00 00  01 00 00 00  00 80 0D 00  00 00 12 08
00 00 12 08  A8 6E 00 00  58 FF 00 00  06 00 00 00
00 10 00 00  02 00 00 00  48 86 0D 00  48 06 12 08
48 06 12 08  10 01 00 00  10 01 00 00  06 00 00 00
04 00 00 00  04 00 00 00  48 01 00 00  48 81 04 08
48 81 04 08  20 00 00 00  20 00 00 00  04 00 00 00
04 00 00 00  50 E5 74 64  84 37 0C 00  84 B7 10 08
84 B7 10 08  6C 25 00 00  6C 25 00 00  04 00 00 00
04 00 00 00  51 E5 74 64  00 00 00 00  00 00 00 00
00 00 00 00  00 00 00 00  00 00 00 00  06 00 00 00
04 00 00 00  2F 6C 69 62  2F 6C 64 2D  6C 69 6E 75
78 2E 73 6F  2E 32 00 00  04 00 00 00  10 00 00 00
```


Intermediate representations

An assembler-like language to translate to as an intermediate stage (Virtual assembler or abstract assembler)

1. It should be language independent,
2. it should not have high level constructions,
3. it should have enough information to allow for optimizations

There are lots of great compiler technology ideas that can be implemented as libraries of compiler components that can be put together to build good compilers.

Optimizations - some examples easy to understand

Constant propagation

Looks for instructions involving only constants and replaces them with a constant value

`add i32 1, 2` becomes `i32 3`

Unswitch loops

Transforms loops that contain branches on loop-invariant conditions to several loops

```
for(...){
  A;
  if(lic) B;
  C
}
```

becomes

```
if(lic)
  for(...){A;B;C}
else
  for(...){A;C}
```

Registers vs memory

One particular optimization has to do with the use of registers instead of memory cells.

Registers

Registers are fast but scarce!

1. register files are part of the CPU
2. instructions operate on data in registers
3. assembly languages use registers that correspond to CPU registers

Memory

Memory is cheaper but slower!

1. data has to be transfered to and from registers (load & store)
2. this adds to the number of instructions that have to be executed

More advanced optimizations - register allocation

The compiler will try to use registers as much as possible for the variables in the source program.

The optimization

1. First assume an unbounded ammount of registers (call them temporaries): you can choose a new one for each variable in the source!
2. Calculate at what instructions temporaries are alive
3. Build an interference graph:
   1. Nodes: Temporaries
   2. Edges: Between temporaries that are alive at the same instructions.
4. Colour the nodes with registers! (like when you colour a map).

Register allocation - ctd.

What if there are not enough registers?

Some of the temporaries are assigned to memory cells and instructions for load and store are generated.

LLVM - low level virtual machine

LLVM - a compiler construction project

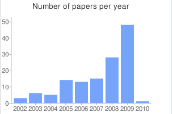
1. Low level virtual instruction set - works as the intermediate representation (IR)
2. CLANG, a front-end for C (to replace gcc)
3. Several back-ends
4. Many optimizations as modules that can be integrated to build a compiler
5. Many tools built from all these (assemblers, linker, compiler driver)

llvm.org

History

Origins
Started as an academic project: the master thesis of Chris Lattner at University of Illinois at Urbana-Champaign, from December 2002.

Currently
Develoment mostly at Apple. The last developer meeting sponsored by Apple, Google, Adobe and Qualcomm Incorporated.

Publications


`llvm.org/pubs/` (Feb. 2010)

Plan for this part of the course

1. Introduce the low level virtual instruction set (IR)
2. Study some examples of IR generated by CLANG
3. Look at what optimizations can do
4. Introduce java classes for the abstract syntax of a fragment of IR needed to compile minijava
5. Show how to generate IR for a minijava program

The low level virtual instruction set

Three-address instructions

```
%var = add i32 4, %x ; a comment!

%var = mul i32 %x,  4 ; Notice the types!

%r = icmp slt i32 %tmp, %tmp1 ; also eq, neq, ...
```

and several more. . .

The low level virtual instruction set

Memory instructions

```
%tmp = load i32* %i ; %i is an address holding an i32

store i32 %add4, i32* %result  ; a value to an address

%retval = alloca i32
```

and we discuss `getelementptr` later on . . .

Example

```
%ptr = alloca i32          ; yields i32*:ptr
store i32 3, i32* %ptr     ; yields void
%val = load i32* %ptr      ; yields i32:val = i32 3
```

An llvm program

```
@h = private constant [8 x i8] c"hello!\0A\00"

declare i32 @puts(i8*)

define i32 @main(){
entry:
  %res = call i32 @puts(i8* getelementptr ([8 x i8]* @h,
                                          i32 0,
                                          i32 0))

  ret i32 0
}
```


An llvm program

The parts of a program

1. String is named `@h`, a global constant (global names start with `@`). Note escape sequences!
2. The library function `puts` is declared: it is given a type signature.
3. `main` (and other functions) are defined.

The file can be assembled to bitcode using `llvm-as` and then executed with `lli`:

```
prompt> llvm-as -f hello.s -o hello.bc
prompt> lli hello.bc
hello!
prompt>
```


SSA - static single assignment

A constraint that facilitates optimizations

In the llvm language a register cannot be assigned more than once (in the text, during execution it might be assigned more than once!)

In other words

A new register for each result!

An incorrect program

```
declare void @printInt(i32 %n)
define i32 @main() {
entry: %t1 = call i32 @sum(i32 100)
       call void @printInt(i32 %t1)
       ret i32 0
}
define i32 @sum (i32 %n) {
entry: %sum = i32 0
       %i = i32 0
       br label %lab1
lab1: %sum = add i32 %sum, %i
      %i = add i32 %i, 1
      %t = icmp eq i32 %i, %n
      br i1 %t, label %end, label %lab1
end: ret i32 %sum
}
```

Reasons

1. Important reason
   Not SSA form: Two assignments to %i and %sum.
2. Trivial reason
   There is no `reg = val` instruction

A correct version of sum

```
define i32 @sum (i32 %n) {
entry: %sum = alloca i32
       store i32 0, i32* %sum
       %i = alloca i32
       store i32 0, i32* %i
       br label %lab1
lab1:  %t1 = load i32* %i
       %t2 = load i32* %sum
       %t3 = add i32 %t1, %t2
       store i32 %t3, i32* %sum
       %t4 = add i32 %t1, 1
       store i32 %t4, i32* %i
       %t = icmp eq i32 %t1, %n
       br i1 %t, label %end, label %lab1
end:   ret i32 %t3
}
```

Comments

%i and %sum are now pointers to memory locations. Only one assignment to any register.

Problem

This program has a lot more memory traffic!

Optimization mem2reg

The opt tool

The opt command is the modular LLVM optimizer and analyzer. It takes LLVM source files as input, runs the specified optimizations or analyses on it, and then outputs the optimized file or the analysis results.

Example

```
prompt > opt -mem2reg correct.bc > correctReg.bc
prompt > llvm-dis correctReg.bc
prompt > lli correctReg.bc
5050
prompt >
```


The optimized sum

```
define i32 @sum(i32 %n) {
entry:
  br label %lab1

lab1:
  %i.0 = phi i32 [ 0, %entry ], [ %t4, %lab1 ]
  %sum.0 = phi i32 [ 0, %entry ], [ %t3, %lab1 ]
  %t3 = add i32 %i.0, %sum.0
  %t4 = add i32 %i.0, 1
  %t = icmp eq i32 %i.0, %n
  br i1 %t, label %end, label %lab1

end:
  ret i32 %t3
}
```


Analysis of the optimized code

1. There are no more `alloca` instructions, no more memory instructions! All addresses have been made into registers (temporaries, remember that llvm has an unbounded number of virtual registers)
2. A new type of instruction was generated: Φ-functions:
   `result = phi type [value1,label1] [value2,label2]`

Φ

At runtime, the phi instruction logically takes on the value specified by the pair corresponding to the predecessor basic block that executed just prior to the current block.

Basic blocks

A bunch of assembler instructions that starts with a label and ends with a branch instruction. There are no other branch instructions or labels in the basic block.

Basic blocks of a function form the Control Flow Graph for the function (many optimizations work on this graph!)

Basic blocks can be arranged in any order!

Why organize code in basic blocks?

1. Frequently executed blocks can be put together at the start of the function and hopefully the number of fall-through conditional branches can be increased.
2. Basic blocks with no predecessors can be removed.

Using CLANG to generate llvm

In the lab you will have to generate llvm assembler for minijava programs.

We will now see what it looks like when CLANG generates code for C programs.

Our C example

```
#include <stdio.h>
int sum(int n){
  int result = 0;
  for(int i = 0; i<n; i++)
    result = result + i + 1;
  return result;
}
int main(int argc, char *argv[]){
 printf("%d\n",sum(atoi(argv[1])));
}
```


The basic blocks of sum

```
define i32 @sum(i32 %n) {
entry:
  %retval = alloca i32
  %n.addr = alloca i32
  %result = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %n, i32* %n.addr
  store i32 0, i32* %result
  store i32 0, i32* %i
  br label %for.cond
```

The first basic block in a function is special in two ways: it is immediately executed on entrance to the function, and it is not allowed to have predecessor basic blocks.

The basic blocks of sum

```
for.cond:
  %tmp = load i32* %i
  %tmp1 = load i32* %n.addr
  %cmp = icmp slt i32 %tmp, %tmp1
  br i1 %cmp, label %for.body, label %for.end

for.body:
  %tmp2 = load i32* %result
  %tmp3 = load i32* %i
  %add = add nsw i32 %tmp2, %tmp3
  %add4 = add nsw i32 %add, 1
  store i32 %add4, i32* %result
  br label %for.inc
```

The basic blocks of sum

```
for.inc:
  %tmp5 = load i32* %i
  %inc = add nsw i32 %tmp5, 1
  store i32 %inc, i32* %i
  br label %for.cond

for.end:
  %tmp6 = load i32* %result
  store i32 %tmp6, i32* %retval
  %0 = load i32* %retval
  ret i32 %0
}
```


Optimized sum

The opt tool again

```
opt -std-compile-opts sum.bc > sumOpt.bc
```

```
define i32 @sum(i32 %n) {
entry:
  %cmp3 = icmp sgt i32 %n, 0
  br i1 %cmp3, label %bb.nph, label %for.end

for.end:
  ret i32 0
}
```


Optimized sum

```
bb.nph:
  %tmp = shl i32 %n, 1
  %tmp5 = add i32 %n, -1
  %tmp6 = zext i32 %tmp5 to i33
  %tmp7 = add i32 %n, -2
  %tmp8 = zext i32 %tmp7 to i33
  %tmp9 = mul i33 %tmp6, %tmp8
  %tmp10 = lshr i33 %tmp9, 1
  %tmp11 = trunc i33 %tmp10 to i32
  %tmp12 = add i32 %tmp, %tmp11
  %tmp13 = add i32 %tmp12, -1
  ret i32 %tmp13
```

What happened to the for-loop?