Intermediate Representations
Concepts of Programming Languages (CoPL)

Malte Skambath
malte@skambath.de

November 16, 2015
Overview

We need Compilers!

Classical Compiler Process

Machine Models
  Stack Machines
  Register Machines

Implementations
  LLVM
  CIL

Conclusion
Developing Software
We need compilers!

Intermediate Representations
Malte Skambath

We need Compilers!
Classical Compiler Process
Machine Models
Stack Machines
Register Machines
Three-Address Code
Static-Single-Assignment
Implementations
LLVM
CIL
Conclusion
Developing Software

We need compilers!

Intermediate Representations
Malte Skambath

We need Compilers!

Classical Compiler Process

Machine Models
Stack Machines
Register Machines
Three-Address Code
Static-Single-Assignment

Implementations
LLVM
CIL

Conclusion

Diagram:
- An input file (.cpp) is shown.
- It leads to three different processors:
  - x86
  - AMD64
  - ARM
Developing Software

We need compilers!

Intermediate Representations

Malte Skambath

We need Compilers!

Classical Compiler Process

Machine Models
- Stack Machines
- Register Machines
- Three-Address Code
- Static-Single-Assignment

Implementations
- LLVM
- CIL

Conclusion
Intermediate Representation

The solution!

We need Compilers!

Classical Compiler Process

Machine Models
- Stack Machines
- Register Machines
- Three-Address Code
- Static-Single-Assignment

Implementations
- LLVM
- CIL

Conclusion
Intermediate Representation

Definition
An *intermediate representation* (IR) is data structure as representation of a program between a high-level programming language and machine code.

An *intermediate language* (IL) is a low-level assembly language as IR for a virtual machine.
Classical Compiler Process

Intermediate Representations

Malte Skambath

We need Compilers!

Classical Compiler Process

Machine Models
- Stack Machines
- Register Machines
- Three-Address Code
- Static-Single-Assigment

Implementations
- LLVM
- CIL

Conclusion
Abstract Syntax Tree

An *abstract syntax tree (AST)* ... 

... describes the syntactical structure of a program 
... depends on the programming language 
... is generated during by the parser

```
program
     /
    /
block
     /
    /
while
     /
    /
condition
     /
     /
variable sum
     /
assign
     /
     /
variable sum
     /
bin op: *
     /
variable i
```
Control-Flow-Graph

```c
int s = 1;
for(int i=1; i<=10; i++)
    s += i;
return (s);
```
Stack Machines

Definition
A general *Stack Machine* has

- a stack as storage
- a set of instructions / operations $\text{op} = F(a_1, a_2, \ldots, a_n)$ including (push and pop)

Executing an operation takes the arguments from top of the stack, computes the result in the accumulator, and pushes the result back to the stack.

Example

```
push 1
push 2
push 3
add
pop
```

![Diagram of stack machine operations]

1 2 3 5 1
We can generate the control by traversing the syntax tree. Assume we have to compute the expression $\sqrt{x^2 + y^2}$. 

We can generate the control by traversing the syntax tree. Assume we have to compute the expression $\sqrt{x^2 + y^2}$.
Stack Machines

Summary

- Programs for stack machines are short
  Only the opcodes (or constants) in the byte code.
- In practical use stack machines can be extended
  1. An external memory to store and load values
     (computations are still limited to the stack)
  2. Top-Level registers
  3. Metainformations (see CIL later)
- Problem: Most processor-architectures use registers.
  ⇒ Hybrid Models, Special informations in the intermediate representation.
Register Machines

Definition

A register machine . . .

- consists of an infinite number of memory cells named registers
- each register is accessible
- has a limited set of instruction / operations:
  1. Arithmetical Operations: Computes a function $F$ using selected registers $⟨o_1, . . . , o_n⟩$ as operands and stores the result in a target register $⟨r⟩$
  2. Jumps/Branches
Three-Address Code (3AC/TAC)

- Each TAC is a sequence of instructions $l_1, l_2, \ldots, l_n$ for a register machine.
- Instructions can be
  1. Assignments $r_1 := r_0$
  2. Unconditional Jumps (Instructions can be labeled)
     
     \[ L0: \text{goto } L1 \]
     
     \[ \ldots \]
     
     \[ L1: r_0 := 1 \]
  3. Conditional Branches
     
     \[ \text{if } a<b \text{ then goto } L1 \]
  4. Arithmetical operations $r_3 := \text{add}(r_1, r_2)$
- Each instruction contains at most 3 registers
Three-Address Code (3AC/TAC)

- Each TAC is a sequence of instructions $I_1, I_2, \ldots, I_n$ for a register machine.
- Instructions can be
  1. Assignments $r_1 := r_0$
  2. Unconditional Jumps (Instructions can be labeled)
    
    \[
    \text{L0: } \text{goto } \text{L1} \\
    \text{\ldots} \\
    \text{L1: } r_0 := 1
    \]
  3. Conditional Branches
    
    \[
    \text{if } a < b \text{ then goto L1}
    \]
  4. Arithmetical operations $r_3 := \text{add}(r_1, r_2)$
- Each instruction contains at most 3 registers

Example ($\sqrt{x^2 + y^2}$)

\[
\begin{align*}
t_1 & := x \times x \\
t_2 & := y \times y \\
t_3 & := t_1 + t_2 \\
\text{result} & := \text{sqrt}(t_3)
\end{align*}
\]
Three-Address Code (3AC/TAC)
How to design the Byte-Code

For practical use we should store TAC in byte code format.

- Each operation has an **opcode** for the virtual machine
- Each instruction can be represented by tuples

<table>
<thead>
<tr>
<th>Quadruples</th>
<th>Triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1</td>
<td>op2</td>
</tr>
<tr>
<td>t1</td>
<td>MUL</td>
</tr>
<tr>
<td>t2</td>
<td>MUL</td>
</tr>
<tr>
<td>t1</td>
<td>ADD</td>
</tr>
<tr>
<td>res</td>
<td>SQRT</td>
</tr>
</tbody>
</table>

**Note**

Registers can be assigned implicitly (Triples). But then each register has to be assigned only once.
Static-Single-Assigment

Definition (Static-Single Assignment)
A Three-Address Code is in Static-Single Assignment from if each register gets assigned once in the code.

Example ($\sqrt{x^2 + y^2}$)

Not in SSA

L1: $x := x \times x$
L2: $y := y \times y$
L3: $x := x + y$
L4: $z := \sqrt{x}$

SSA

L1: $x_0 := x \times x$
L2: $y_0 := y \times y$
L3: $x_1 := x_0 + y_0$
L4: $z := \sqrt{x_1}$
Static-Single-Assigment

How to get SSA-form?

A simple Algorithm

- For each used register: \(<R>\)
  1. Check if \(<R>\) gets assigned more than once
  2. For each assignment/definition of \(<R>\):
     - Rename on the left side to \(<R.i>\) if this assignment is the \(<R>\):
       - \(i\)-th assignment to \(<R>\)
  3. For each use of \(<R>\):
     - Replace \(<R>\) with \(<R.j>\) where \(<R.j>\) was the previous replacement for \(<R>\).

Is this algorithm correct?
Static-Single-Assigment
How to get SSA-form?

A simple Algorithm

- For each used register: \( \langle R \rangle \)
  1. Check if \( \langle R \rangle \) gets assigned more than once
  2. For each assignment/definition of \( \langle R \rangle \):
     - Rename on the left side to \( \langle R.i \rangle \) if this assignment is the \( i \)-th assignment to \( \langle R \rangle \)
  3. For each use of \( \langle R \rangle \):
     - Replace \( \langle R \rangle \) with \( \langle R.j \rangle \) where \( \langle R.j \rangle \) was the previous replacement for \( \langle R \rangle \).

Is this algorithm correct?
No!
Static-Single-Assignment

What if we have branches?

```plaintext
if a > b then goto L_A
max := b;
goto L_END
L_A:
max := a;
goto L_END
L_END:
```

![Diagram showing the flow of control for the comparison of 'a' and 'b'.]
Static-Single-Assignment

The \( \Phi \)-function

The \( \Phi \)-function computes the value depending on the incoming branch.

\[
\begin{align*}
a & \leftarrow 1 \\
b & \leftarrow 2 \\
x & \leftarrow \Phi(a, b)
\end{align*}
\]

Note
There is no real operation like \( \Phi \) in real machines. After optimization \( \Phi \)-statements have to be removed.
Static-Single-Assignment

The $\Phi$-function

The $\Phi$-function computes the value depending on the incoming branch.

\[ a \leftarrow 1 \]
\[ b \leftarrow 2 \]
\[ x \leftarrow \Phi(a, b) \]

$x$ has value 1

Note
There is no real operation like $\Phi$ in real machines. After optimization $\Phi$-statements have to be removed.
Static-Single-Assignment

The $\Phi$-function computes the value depending on the incoming branch.

\[ a \leftarrow 1 \quad b \leftarrow 2 \]
\[ x \leftarrow \Phi(a, b) \]

$x$ has value 2

Note

There is no real operation like $\Phi$ in real machines. After optimization $\Phi$-statements have to be removed.
Getting Code in SSA-form.

L1:
if r_a < r_b then goto L3:

L2:
t_1 := r_a
goto L4

L3:
t_2 := r_b
goto L4

L4: max := phi t_1 [from L2], t_2 [from L3]
Converting to SSA-Form

1. Place $\Phi$-function terms
2. Rename registers to achieve SSA-form

Using the $\Phi$-function after each branch for previous registers is an unpractical solution.
Dominance Frontiers

Definition
We say $x$ dominates $y$ ($x \text{ dom } y$) if on all paths to $Y$ in the CFG the program has to run over $X$.

Definition
$y$ is in the dominance frontier of $x$ ($\text{DF}(x)$) iff not $x \text{ dom } y$ and $y$ has a direct predecessor on all paths to $y$

\[ \text{DOM}(5) = \{5, 6, 7, 8\} \]
\[ \text{DF}(5) = \{4, 9\} \]
Dominance Frontiers

Assume that node 3 defines variable $x$, $DF(3) = \{5\}$

Is 5 the only node we need to insert a $\Phi$-function for $x$?
Dominance Frontiers

Assume that node 3 defines variable $x$, $\text{DF}(3) = \{5\}$

Is 5 the only node we need to insert a $\Phi$-function for $x$?

No, at node 6. Why?
LLVM uses a special intermediate representation (LLVM-IR) for a virtual register machine.
lldrers typically are limited to low-level optimizations such as profile-driven code layout, interprocedural register allocation, constant propagation, and dead code elimination.

Profile information is traditionally collected by creating a special "profile enabled" build of the application, running it with representative input, and feeding the data generated back into another build cycle. The primary problems with this approach are that it requires extra work for the developers to use (which reduces the likelihood that profile directed compilation will be used), and that getting a representative input sample can be very difficult for some classes of application.

Traditionally, once the application has been linked, the executable remains frozen throughout its lifetime. While speculative, we believe there may be use for runtime and offline optimization, which are not available in existing systems. Program transformation extremely late in the lifetime of the application is necessary to adapt to changing usage patterns of the end-user and to adapt to hardware changes that may occur (allowing retuning for a new CPU, for example).

In contrast, LLVM enables novel interprocedural transformations at link time, runtime, and even in the field. It fits the standard "compile-link-execute" build model, working as a drop-in replacement for an existing compiler in build scripts, easing deployment of new optimizers. LLVM can gather profiling information transparently to the programmer, without requiring a special profile build and "representative" training executions.

This paper describes the high level approach to compilation used by LLVM (Section 2), contains detailed information about the virtual instruction set used throughout the compilation process (Section 3), describes some of the transformations implemented in the LLVM infrastructure (Section 4), and describes key components of our infrastructure (Section 5). An online reference manual describes the individual LLVM instructions and tools in more detail.

2 The LLVM Approach

The centerpiece of the system is the LLVM virtual instruction set. It combines a low-level representation with high-level information about operands, by using Static Single Assignment (SSA) form and high-level, language independent type information (more detail is provided in Section 3). The RISC like, low-level instructions of LLVM allow for many traditional optimizations to be directly applied to LLVM code, while the high-level type information allows for novel high-level transformations as well. This common representation enables the powerful multi-phase compilation strategy shown in Figure 1.

As Figure 1 illustrates, the LLVM compilation strategy exactly matches the standard compile-link-execute model of program development, with the addition of a runtime and offline optimizer. Unlike a traditional compiler, however, the .o files generated by an LLVM static compiler do not contain any machine code at all – they contain LLVM code in a compressed format.

The LLVM optimizing linker combines LLVM object files, applies interprocedural optimizations, generates native code, and links in libraries provided in native code form. An interesting feature of the executables produced by the

C. Lattner, The LLVM Instruction Set and Compilation Strategy, 2002
Intermediate Representations
Malte Skambath

We need Compilers!

Classical Compiler Process

Machine Models
- Stack Machines
- Register Machines
- Three-Address Code
- Static-Single-Assignment

Implementations
- LLVM-IR
- LLVM
- CIL

Conclusion

LLVM-IR

@.str = private unnamed_addr constant [11 x i8] c"
  %d\n<=%d\n\00", align 1

; Function Attrs: nounwind uwtable
define void @minmax(i32 %a, i32 %b) #0 {
  %1 = icmp sgt i32 %a, %b
  br i1 %1, label %2, label %3

  ; <label>:2
  br label %4

  ; <label>:3
  br label %4

  ; <label>:4
  %max.0 = phi i32 [%a, %2 ], [%b, %3 ]
  %min.0 = phi i32 [%b, %2 ], [%a, %3 ]
  %5 = call i32 (i8*, ...) @printf(i8*
    getelementptr inbounds ([11 x i8], [11 x i8]*
      @.str, i32 0, i32 0), i32 %min.0, i32 %max.0)
  ret void
}
LLVM-IR

- LLVM is register-based. Registers are written as %<registernname> (e.g. %R1 = ...) @ is used for global variables (e.g. function names)
- LLVM use types i1, i8, i32 for boolean, Byte and 32-Bit Integer values
- Reduced instruction-set
  - Memory Access %ptr = alloca i32
  - Comparing %res = icmp <opt> <type> %a, %b
- Conditional Branches
  - br i1 %cond, label %IfLabel, label %ElseLabel
- Function calls %res = call
- phi-Instruction for assignments depending on the control flow
- Functions:
  define <type> @FctName(<type> %arg1,...){...}
- Metadata
Intermediate Representations

We need Compilers!

Classical Compiler Process

Machine Models
Stack Machines
Register Machines
Three-Address Code
Static-Single-Assignment

Implementations
LLVM
CIL

Conclusion

LLVM-IR

```llvm
@.str = private unnamed_addr constant [11 x i8]
c"_%d_<=_%d_\00", align 1

; Function Attrs: nounwind uwtable
define void @minmax(i32 %a, i32 %b) #0 {
%1 = icmp sgt i32 %a, %b
br i1 %1, label %2, label %3

; <label>:2 ; preds = %0
br label %4

; <label>:3 ; preds = %0
br label %4

; <label>:4 ; preds = %3, %2
%max.0 = phi i32 [ %a, %2 ], [ %b, %3 ]
%min.0 = phi i32 [ %b, %2 ], [ %a, %3 ]
%5 = call i32 (i8*, ...) @printf(i8*
    getelementptr inbounds ([11 x i8], [11 x i8]*
        @.str, i32 0, i32 0), i32 %min.0, i32 %max.0)
ret void
}
```
Another example

```llvm
define i32 @main() #0 {
  %c = alloca [10 x i32], align 16
  br label %1

1:
  %sum.0 = phi i32 [ 0, %0 ], [ %4, %7 ]
  %i.0 = phi i32 [ 1, %0 ], [ %8, %7 ]
  %2 = icmp sle i32 %i.0, 10
  br i1 %2, label %3, label %9

3: ; preds = %1
  %4 = add nsw i32 %sum.0, %i.0
  %5 = sext i32 %i.0 to i64
  %6 = getelementptr inbounds [10 x i32], [10 x i32]* %c, i32 0, i64 %5
  store i32 %4, i32* %6, align 4
  br label %7

7: ; preds = %3
  %8 = add nsw i32 %i.0, 1
  br label %1

9: ; preds = %1
  ret i32 0
}
```
The CLR is the CLI-Implementation of Microsoft and part of the .net-Framework. The CLI also specifies a Type System (CTS) and a basic set of class libraries.
CIL

- Stack based virtual machine.
- Each method has a header
- Typed instruction-set (e.g. ldci4.0 load constant 0 as 4-Byte int)
- Access to local variables ldloc.<index>, stloc.<index>
- Object oriented
  - Load field values
    ldfld string Program/Person::prename
  - Create new objects newobj instance void class <CLASS>’.ctor’(...)
CIL

An Example

```
.method public static hidebysig
default int32 sum (int32 a, int32 b) cil managed {
  .maxstack 2
  .locals init (int32 V_0, int32 V_1)
  IL_0000:  ldc.i4.0  //
  ...
}
```
CIL

An Example

IL_0000: ldc.i4.0  //
IL_0001: stloc.0  // sum = 0
IL_0002: ldarg.0  // load a on the stack
IL_0003: stloc.1  // store a in first var
(i=a)
IL_0004: br IL_0011  // --+
IL_0009: ldloc.0  //  |  <---+
IL_000a: ldloc.1  //  |   |
IL_000b: add  //  |
IL_000c: stloc.0  //  |
IL_000d: ldloc.1  //  |
IL_000e: ldc.i4.1  //  |
IL_000f: add  //  |    .
IL_0010: stloc.1  //  |
IL_0011: ldloc.1  //  <+-  .
IL_0012: ldarg.1  // load b   |
IL_0013: ble IL_0009  // i<=b   +-  
IL_0018: ldloc.0  
IL_0019: ret
Intermediate Representations . . .

- allow a clean and general compiler-architecture/infrastructure
- allow mixing different programming languages
- programmer may lose control on the real control-flow
- program-flow can be optimized
- adaptation to different hardware configurations (including GPU-support).
- improve the development of new programming languages
- can realize translations between different languages
Any Questions?