CS 322: Languages and Compiler Design II

Mark P Jones, Portland State University

Spring 2015

Week 9: Using an intermediate representation

Source high, target low

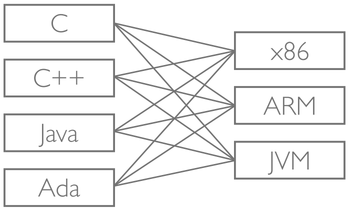
- How can we build an effective optimizer?
- Source code is often too high-level to reveal opportunities for optimization
  - Example: `a[i] = x + y` requires calculation of the address of the array element `a[i]`, not visible in source
- Target code is often too low-level to reveal opportunities for optimization
  - Examples: temporary values have already been assigned to registers; it is more difficult to identify repeated or redundant computations; …

Intermediate representations (IR)

- Intermediate representations provide a compromise between the extremes of source and target code
- A good IR aims to be:
  - sufficiently low-level to capture single steps in a program
  - sufficiently high-level to avoid machine dependencies and premature commitments in code generation
- IRs are usually some kind of idealized machine code (like the Target language of DemoComp!)
- As a useful side benefit, an IR can simplify the task of constructing compilers for multiple source languages on multiple platforms ...

Multiple languages and targets

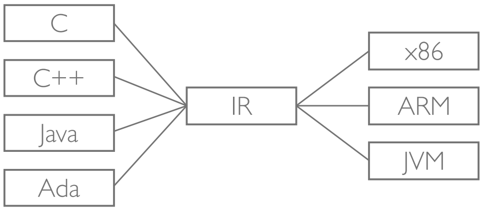
- Suppose that we want to write compilers for n different languages, with m different target platforms.



- That's n x m different compilers!

Using an intermediate representation

- Alternatively: design a general purpose, shared IR:



- Now we only have n front ends and m back ends to write!
- The biggest challenge is to find an IR that is general enough to accommodate a wide range of languages and machine types

IR design

- Surely there must be a standard IR that all compilers can use?
  - Then we'd just need one H ⟶ IR front end for each high-level language H
  - And one IR ⟶ P back end for each execution platform P
  - And just one, super-duper optimizer IR ⟶ IR
- What would this IR look like?
- What features would we expect it to support?

Complication: source language diversity

- It would need to support a wide range of source language features:
  - integers, floating point, characters, ...
  - arrays, pointers, objects, structures, ...
  - functions, call-by-reference, exceptions, threads, ...
  - statically typed vs. dynamically typed
  - multiple paradigms: imperative vs. functional vs. object-oriented vs. ...
  - ...

Complication: target language diversity

Example: accessing the value of an array element, a[i]

- Simple loads and saves:

  $t_1 \leftarrow 4*i$
  $t_2 \leftarrow a+t_1$
  $v \leftarrow [t_2]$

  a pure RISC design

- Indexed loads and saves:

  $t_1 \leftarrow 4*i$
  $v \leftarrow a[t_1]$

- Indexed and scaled loads and saves:

  $v \leftarrow a[4*i]$

  a CISC design

  $\texttt{movl n}(r_1,r_2,m)\texttt{, } r_3$

Conclusion: designing a general IR is hard

- It is difficult to get the right level of abstraction
- Too low-level ⟹ harder to generate good target code

  how can we recognize when a sequence of low-level instructions can be implemented effectively by one single higher-level instruction?

- Too high-level ⟹ hides opportunities for optimization

  if the optimizer cannot "see" critical details of how a construct is implemented, then it cannot optimize it.

- The search for a fully general IR goes on!
- Maybe we could we use C? JVM? CLR? RTL? or …

Introducing LLVM

A very brief overview of LLVM

- "The LLVM Project is a collection of modular and reusable compiler and toolchain technologies" (from llvm.org)
  - "a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages"
- Originally developed by Chris Lattner and Vikram Adve as a research project at the University of Illinois
- Now used commercially by:
  - Apple (Mac OS and iOS SDKs)
  - Open CL (GPGPU programming; Intel, NVIDIA, Apple)
  - Sony PS4
  - … and many more

Time for some lab exercises …