Co-degeneration

High level languages

Java

stack code

allocate variables to registers

try to avoid excessive copying

represent stack as temporary variables

analyze stack to determine size

Converting into real code

allocate variables to registers

try to avoid excessive copying

represent stack as temporary variables

analyze stack to determine size

Issue

verified when loaded

machine-independent stream of bytes

structure for de-spatching program

Class files

Compiling Java

Code generation

 peepehole

 free parsing

 free pattern matching

Code Generator Generators

– function calls

– higher order functions

– functional

– method invocation

– object-oriented

– allocate registers to top of stack

– stack code

Java

High level languages
Compiling stack code

General algorithm

- Determine local storage
- Max locals + max stack + max temps
- Translate each instruction
- Map to top of stack, first locals to registers

Register allocation approach

- More locations between memory and registers
- Translate each instruction
- Map each local/stack location to a frame

Naive approach

- Translate instructions
- Find stack height for instruction
- Form basic blocks

Register approach

- Fast block approach maps registers for basic blocks
- Fast approach maps registers for entire method
- Map top of stack, first locals to registers

Implementation

- Allows shared implementation of objects
- Organizes space of objects
- Collection of objects with same attributes

Classes

- Functions (methods) for operating on data
- A collection of data

Objects

Object-oriented (OO) languages

Objects

- Collection of data

- Store for local data
- Pointer to class type

Implementation

- Object record
- Storage for class data

( feminists (method table)
- Storage for local data
- Pointer to method table

Basic block approach maps registers for basic blocks

Locating — offset in object record/method table

Classes

- Functions (methods) for operating on data
- A collection of data

Objects

Object-oriented (OO) languages

Objects
Inheritance

Inheritance

Class Hierarchy

Inheritance

Data Layout Optimization

Single Inheritance

Multiple Inheritance

Impact

Can we eliminate overhead of data/method lookups?

- need to test tags at runtime
- non-constant data/method pointer offset
- class of object not completely known at compile-time

Multiple Inheritance → multiple inheritance

Subclasses may override methods from superclass

- expected

Subclasses should work whenever superclass is expected

- descendant class inherits attributes (superclass)
- descendant class does not inherit attributes (superclass)

Class may inherit data/methods from another

Inheritance
Functional Programming Languages

Method lookup optimization

- Eliminates call overhead
- Type propagation to prove class type

Additional optimizations
- Overwrite slots as needed
- Assign slots via graph coloring

Multiple inheritance
- Inherit multiple functions through function pointers
- Get function pointer at offset in method table
- Get method to class record from object
- Methods are executed through
- Ensure consistent offset for methods
- Overwrite methods by overwriting slot
- Arrange method tables entries via prefixing

Features
- Emphasis on function calls, recursion
- Higher order functions
- Emphasis on function calls, recursion
- Calculate solutions to equations (e.g., calculus)
- Emulates equation-based reasoning
- Tries to avoid side effects (e.g., assignment)

Examples
- Define FACT
  { lambda (n) [ cond [ (eq n 1) (lambda (n) define FACT) ] (mult n (FACT (sub n 1)) ) ] }
- Define ADD
  { lambda (n) [ lambda (x) addn x n ] ) }
Higher order functions

- May also inline functions
  (current state represented as closure)
- Recursive by adding argument for continuation
  can transform all function calls into tail
- Convert tail recursion from function call to goto
  of the parent procedure
- Tail recursion ← result of call is the return value

Function calls

- may be referred to by inner-nested functions
- analysis to determine when variables escape
- must allocate activation records on heap
- simple closure ← function & static link
- method to access nonlocal variables
- record containing pointer to function and
  represent function pointers as closures

Two major schools

Instruction matching
Tree pattern matching
Generator

Resulting code should run quickly
Resulting code should produce great code
Generator (C8)
Instruction description + IR description give code
Would like a description-based tool

Automate the process

Code Generator Generators

Auto generating code

Lecture 1/5, Page 9

Twomajorschools

Tree pattern matching
Instruction matching
This scheme should look familiar.

Tree pattern matching

1. Machine description is
   - Mapping of subtree into single node
   - Machine description is 

   (BIRs)

2. Associated code (to be emitted)

Assume that the program is represented as a set of

Tree pattern matching

\[ \{ \text{load } r_1, a; \text{load } r_2, b; \text{add } r_1, r_1, r_2 \} \]

- \[ x_1 \rightarrow a + b \]

Example pattern:

\[ \text{example pattern:} \]

\[ \text{treepatternmatching} \]

\[ \text{treewriting schemes} \]

\[ \text{BURS} \]

\[ \text{example pattern:} \]

\[ \{ \text{load } r_1, a; \text{load } r_2, b; \text{add } r_1, r_1, r_2 \} \]

- \[ x_1 \rightarrow a + b \]

Example pattern:

\[ \text{example pattern:} \]

\[ \{ \text{load } r_1, a; \text{load } r_2, b; \text{add } r_1, r_1, r_2 \} \]

- \[ x_1 \rightarrow a + b \]

Example pattern:
Tree rewriting schemes

Several basic techniques

Treeparsing schemes

Use LR parsers

Several treeparsing schemes

Linear time scheme

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift
- reduce
- shift/reduce
- reduce/reduce
- pick longer reduction

- reduce
- reduce/reduce
- pick longer reduction

- reduce
- reduce/reduce
- pick longer reduction

- reduce/reduce
- pick longer reduction

- reduce/reduce
- pick longer reduction

- reduce/reduce
- pick longer reduction

- shift
- shift/reduce
- reduce/reduce
- pick longer reduction

LR parsers

- shift
- reduce
- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction

- shift/reduce
- reduce/reduce
- pick longer reduction
Typical machines are represented in some low/intermediate representation. Assume the program is represented in some low-level machine-dependent optimizations.

Peephole optimization is used for implementing many complex instructions. Use pattern matching to synthesize more complex instructions. Use a very small context (2-10 instructions) and locally adjacent instructions that can be combined. Combined instructions

Candidate training sets offer a one-to-one translation for IR adding patterns to improve code (more complex paradigms). Use a linear time pattern matcher run from the tables produced by the trainer. Use a linear time pattern matcher run from the tables produced by the trainer. Read a set of representative programs to the trainer and let it build a table by exhaustive search.

Training Generator

Instructions and addressing modes

Peephole code generation

Intermediate representation (IR)