ECS652: Compilers (Spring 2018)

Tutorial 10

Cool Runtime System

Greta Yorsh

PA3

- Apologies for the delay
- Finally available in distro
- Deadline: midnight Sunday, 8 April
- Small tasks
- Marking: best 2 out of 3 for pa1-pa3
- Use distro github for bug or feature requests

COOL CODE GENERATION

Recap: Operational semantics

- Shows how to execute a program
- Value of expression derived from values of its subexpressions
- Operational rules are more precise than typing rules
- Why even bother with type checking ?
- Type checking can find potential errors **at compile time**
- Essentially, your task in PA3 is to convert operational rules into cgen functions, much the same way you converted typing rules into code in PA2

Code generation design

- There are many possible ways to write the code generator
  - how are objects represented in memory?
  - how is dynamic dispatch implemented?
  - how is code emitted for expressions?
  - where are locals and temporaries stored?
- The stack frame must be designed together with the code generator

Code generation design

- Putting together object layout and code generation for expressions and methods
- How to generate code for free variables in an expression

- Example: x.a + y.f() + t

Major tasks

- Cool object layout
  - assign tags to all classes in depth-first order
  - determine the layout of attributes, temporaries, and dispatch tables for each class
- Generate code
  - for global data: constants, prototype objects, dispatch tables
  - initialization for each method
  - code for each method using cgen environment

Passes

- Prepare offsets
  - decides the object layout for each class
  - compute offsets of attributes and methods
- Traverse each feature and generates machine code for each expression

Preparing offsets

- Traverse the inheritance graph
- Assign tags to all classes
- Allocate offset for class attributes
  - don’t forget to take superclass into account
- Allocate offsets for methods
  - ordering inside the dispatch-vector
  - again, don’t forget superclass

Code generation environment

- Layout information for free variable and method names
  - used in code generation for expressions
- Class name and file name
  - used by code that produces runtime error messages
- Global counter for generating unique labels

Distinct kinds of names

- Locals: stored in the temporary area of the stack frame
- Formal parameters: in the actuals area of the frame
- Self: in the designated register $s0
- Attributes: at an offset from the address given by self
- Method address: at an offset from the dispatch table

- Distinct conventions for generating code for allocation, reference, and update

Code generation for expressions

- code selection
- register allocation
- calling sequence
    - callee/caller save registers
    - explicit parameter passing
    - prolog, epilog
    - known offsets of formals and temporaries

Approaches

- Code generation for expressions
    - stack machine with accumulator
    - simple code selection with register allocation
    - weighted register allocation
- Code generation for basic blocks
    - graph coloring

COOL RUNTIME SYSTEM

Cool Runtime System

- What does it provide ?
- What does it assume about our code ?

Runtime system: what does it **provide**?

- Garbage collector
- Start-up code to invoke main
- Code for methods of basic classes
- Utilities
  - equality
  - runtime error reporting

Start-up in trap.handler

- Create copy of Main prototype object
- Call Main_init initialize Main’s parent classes and attributes of Main
- Call Main.main
  - $a0 contains Main
  - $ra contains return
- If Main.main returns, execution halts with “COOL program successfully executed”

Methods of Object

Object.copy	input: $a0 address of prototype object return: $a0 contains address of newly allocated object
Object.abort	prints the name of object in $a0 terminates execution
Object.type_name	returns in $a0 address of the string object that contains the name of the class of object passed in $a0 uses class tag and the table class_nameTab

Other methods

equality_test	args: $t1 $t2 if args have same primitive type and same value return $a0 o/w $a1
_dispatch_abort	dispatch on void print $t1 line number and $a0 filename
_case_abort	case with no match prints name of object at $a0
_case_abort2	case on void print $t1 line number and $a0 filename

System calls

Service	Code	Arguments	Result
print integer	1	$a0=integer	Console print
print string	4	$a0=string address	Console print
read integer	5		$a0=result
read string	8	$a0=string address $a1=length limit	Console read
exit	10		end of program

Runtime system: what does it **require**?

- Object layout: header, GC tag
- Global data: class name table, class object table, prototype objects, constants
- Naming conventions for labels
- Register usage
- Calling conventions for methods of runtime system
    - use the same for user-defined methods

Global data

- **Class name table** is a mapping from class tags to class names
  - used for error reporting
- **Class object table** is a mapping from class tag to address of prototype object for the class
  - used for allocation
- Prototype objects
- Constants
  - String: class names, filenames
  - Int and Bool constants

Naming conventions for labels

• Dispatch table	<classname>_dispTab
• Method entry point	<classname>.<method>
• Class init code	<classname>_init
• Prototype object	<classname>_protObj
• Integer constant	int_const<Symbol>
• String constant	str_const<Symbol>

Cool object prototypes and dispatch tables

```
        .word   -1
A_protObj:
        .word   5
        .word   5
        .word   A_dispTab
        .word   int_const0
        .word   int_const0
        .word   -1
B_protObj:
        .word   6
        .word   6
        .word   B_dispTab
        .word   int_const0
        .word   int_const0
        .word   int_const0
        .word   -1
C_protObj:
        .word   7
        .word   6
        .word   C_dispTab
        .word   int_const0
        .word   int_const0
        .word   int_const0
```

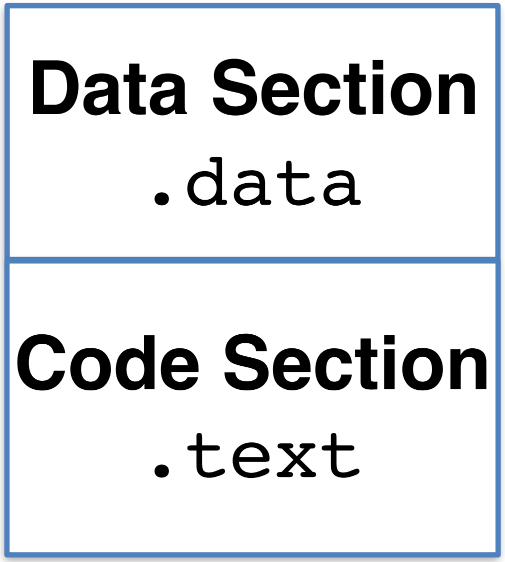
```
A_dispTab:
        .word   Object.abort
        .word   Object.type_name
        .word   Object.copy
        .word   A.f
B_dispTab:
        .word   Object.abort
        .word   Object.type_name
        .word   Object.copy
        .word   B.f
        .word   B.g
C_dispTab:
        .word   Object.abort
        .word   Object.type_name
        .word   Object.copy
        .word   A.f
        .word   C.h
```

Class A { a: Int ←0; d: Int ← 1; **f**(): Int { a ← a + d}; };
Class B inherits A { b: Int ←2; **f**(): Int {a}; **g**(): Int {a ← a - c}; };
Class C inherits A { c: Int ←3; **h**(): Int {a ← a * c}; };

Cool register conventions

- Zero $0
- ACC $a0
- SELF $s0 (callee saves)
- Stack pointer $sp
- Frame pointer $fp
- Return address $ra
- Arguments: self passed in $a0, others on the stack

Assembly file roadmap



Data Section

`.data`

Code Section

`.text`

Statically allocated data

- constants
- prototype objects
- dispatch tables

Code

- code for methods
- start-up code
- error handlers
- garbage collector

trap.handler