COS320: Compiling Techniques

Zak Kincaid

April 30, 2019
• Reminder: HW5 is due today
• HW6 released Tuesday
  • Dataflow analysis
  • Dead code elimination
  • Alias analysis
  • Constant propagation
  • Register allocation
• Come to class Thursday prepared with questions
Compiling object-oriented languages
Objects

- An object consists of
  - Data (attributes) –
  - Behavior (methods) –

```java
class AstNode {
    location loc;
    public AstNode(location nodeloc) { loc = nodeloc; }
    public location getLocation() { return loc; }
}

abstract class Expr extends AstNode {
    public abstract int eval(Env);
    public Expr(location loc) { super(loc); }
}

public class AddExpr extends Expr {
    public AddExpr(int loc, Expr x, Expr y) {
        super(loc); left = x; right = y;
    }
    public int eval(Env env) {
        return left.eval(env) + right.eval(env);
    }
}
```
Objected oriented languages

- Compiling OO languages with single inheritance:
  - Each class is associated with a dispatch vector (aka virtual table, vtable), which is a record of function pointers – one for each method
  - Each object is associated with a record, with one field for the dispatch vector of its class, and one field for each attribute
Implementing methods

Each method extended with an additional parameter for the current object

- Gives the method access to the attributes of the object
- Dispatch vector enables dynamic dispatch

```cpp
location AstNode_getLocation(self) {
    return self.loc;
}
int AddNode_eval(self, env) {
    return self.dispatch.eval(self, self.left) + self.dispatch.eval(self, self.left);
}
int IntNode_eval(self, env) {
    return self.value;
}
```
Subtyping

- Recall the *Liskov substitution principle*: if \( s \) is a subtype of \( t \), then terms of type \( t \) can be replaced with terms of type \( s \) without breaking type safety.
- If \( B \) extends \( A \), then \( B \) is a subtype of \( A \)
- This works for the same reason that record width subtyping works:
  - If \( A \) has a method \( \text{foo} \), it appears in the same position in \( A \) and \( B \)'s dispatch vector
  - If \( A \) has an attribute \( x \), then \( A \) objects and \( B \) objects place \( x \) in the same position in object records

\[
\text{RECORDWIDTH} \\
\vdash \left\{ \text{lab}_1 : s_1 ; \ldots ; \text{lab}_m : s_m \right\} <: \left\{ \text{lab}_1 : s_1 ; \ldots ; \text{lab}_n : s_n \right\} \quad n < m
\]
Testing class membership

• Some OO languages support testing whether an object belongs to a given class, and performing (checked) downcasts
• To implement, we need a run-time representation class of the class hierarchy
• Possible solution:
  • The dispatch table serves as a type tag
    (i.e., \( \text{typeof(o)} == \text{AddExpr} \iff o.\text{dispatch} = \text{DispatchVector(AddExpr)} \))
  • The first member of each dispatch table is a pointer to parent type
  • To check \( o \text{ instanceof C} \), walk up the class hierarchy
    • \( o.\text{dispatch} = \text{DispatchVector(C)}, \text{or} \)
    • \( o.\text{dispatch} != \text{DispatchVector(Object)} \text{ and } o.\text{dispatch}.\text{parent} = \text{DispatchVector(C)}, \text{or} \)
    • \( o.\text{dispatch} != \text{DispatchVector(Object)} \text{ and } o.\text{dispatch}.\text{parent} \neq \)
      \( \text{DispatchVector(Object)} \text{ and } o.\text{dispatch}.\text{parent}.\text{parent} = \text{DispatchVector(C)}, \text{or} \)
    • ...
  • Checked downcasting: if \( o \text{ instanceof } c \) then bitcast, otherwise throw run-time exception.
Multiple inheritance

- Some languages (such as C++) support a class extending more than one base class.
- Previous strategy does not work: bases classes have conflicting ideas about where methods are stored in vtable.
- Solution: Use hash tables instead of records.
- Cost can be reduced with optimizing compiler:
  - Perform a conservative analysis to determine the class of (some) objects. If known statically, can replace dynamic dispatch with static dispatch.
  - JIT compilation:
    - At compile time, we have more precise information about object classes.
    - Replace dynamic dispatch with static dispatch, optimize & compile the result.
Compiling functional languages
First class functions: functions are values just like any other
  - can be passed as parameters (e.g., map)
  - can be returned (e.g. \((+) 1\))

Functions that take functions as parameters are called higher-order

A higher-order functional language is one with nested functions with lexical scope

In higher-order functional languages, a function value is a closure, which consists of a function pointer and an environment
  - Environment is needed to interpret variables from enclosing scope
let compose =
    fun (f : int -> int) ->
        (fun (g : int -> int) ->
            (fun (x : int) ->
                f (g x)))
let add10 = fun (x : int) -> x + 10
let mul2 = fun (x : int) -> 2 * x
let result = compose add10 mul2 100
let compose = 
  fun (f : int -> int) -> 
    (fun (g : int -> int) -> 
      (fun (x : int) -> 
        f (g x)))
let add10 = fun (x : int) -> x + 10
let mul2 = fun (x : int) -> 2 * x
let result = compose add10 mul2 100
let compose = fun (f : int -> int) ->
    (fun (g : int -> int) ->
     (fun (x : int) ->
      f (g x)))

let add10 = fun (x : int) -> x + 10
let mul2 = fun (x : int) -> 2 * x
let result = compose add10 mul2 100
let compose = fun (f : int -> int) ->
  (fun (g : int -> int) ->
   (fun (x : int) ->
    f (g x)))

let add10 = fun (x : int) -> x + 10
let mul2 = fun (x : int) -> 2 * x
let result = compose add10 mul2 100
let compose =
  fun (f : int -> int) ->
    (fun (g : int -> int) ->
      (fun (x : int) ->
        f (g x)))
let add10 = fun (x : int) -> x + 10
let mul2 = fun (x : int) -> 2 * x
let result = compose add10 mul2 100
Compiling closures

- fun expressions evaluate to a pair \((body, env)\) consisting of
  - \(body\): A pointer to a function that implements the body of the closure
    - Takes an extra parameter, \(env\) (similarly to \(self/this\) in OO)
  - \(env\): A pointer to the activation record of the enclosing function

- Functions are first-class values, so they may be returned from functions
  - I.e., a closure may outlive the activation record of the enclosing function
  - activation records must be heap-allocated!
Closure conversion

- **Closure conversion** transforms a program so that no function accesses free variables.

```ocaml
let f(a,b,c) = let g = fun x -> x + a in (fun y -> g(g(y)), fun y -> y * c)
```

- We say that \(a\), \(c\), and \(g\) escape: they appear free in the body of a nested function.
  - Each escaping var must be stored in an environment. Non-escaping vars can be discarded.
  - First field in the environment is a pointer to enclosing environment.

```ocaml
let f(p,a,b,c) =
  let r1 = (p,a,c) in
  let g = (fun (p, x) -> x + (#1 p), r1) in
  let r2 = (r1,g) in
  let res1 = fun (p, y) ->
    let g = #1 p in
    ((#0 g) (#1 g, y))
  in
  let res2 = fun (p, y) -> (y * (#2 (#0 p))) in
  ((res1, r2), (res2, r2))
```
let root = ()
let compose =
  (fun (p, f) ->
   let r1 = (p, f) in
   (fun (p, g) ->
     let r2 = (p, g) in
     (fun (p, x) ->
       let g = #1 p in
       let f = #1 (#0 p) in
       ((#0 f) ((#1 f), (#0 g) (#1 g, x)))
       r2),
       r1),
   root)
let add10 = (fun (p, x) -> x + 10, root)
let mul2 = (fun (p, x) -> 2 * x, root)
let result =
  let compose_add10 = (#0 compose) (#1 compose, add10) in
  let compose_add10_mul2 = (#0 compose_add10) (#1 compose_add10, mul2) in
  ((#0 compose_add10_mul2) (#1 compose_add10_mul2, 100))
Functional optimizations

- **Tail call elimination**: functional languages favor recursion over loops, but loops are more efficient (need to allocate stack frame, push return address, save registers, ...)
  - Tail call elimination searches for the pattern
    
    \%x = call foo ...; ret \%x
  
    and compiles the call as a jump instead of a callq
Functional optimizations

- **Tail call elimination**: functional languages favor recursion over loops, but loops are more efficient (need to allocate stack frame, push return address, save registers, ...)
  - Tail call elimination searches for the pattern
    \[
    \%x = \text{call} \ foo \ldots; \ \text{ret} \ \%x
    \]
    and compiles the call as a jump instead of a \text{callq}
- **Function inlining**: functional programs tend to have lots of small functions, which incurs the cost of more function calls than there may be in an imperative language
  - Inlining replaces function calls with their definitions to alleviate some of this burden
Functional optimizations

- **Tail call elimination**: functional languages favor recursion over loops, but loops are more efficient (need to allocate stack frame, push return address, save registers, ...)
  - Tail call elimination searches for the pattern
    \[
    \%x = \text{call foo} \ldots; \text{ret} \%x
    \]
    and compiles the call as a jump instead of a callq

- **Function inlining**: functional programs tend to have lots of small functions, which incurs the cost of more function calls than there may be in an imperative language
  - *Inlining* replaces function calls with their definitions to alleviate some of this burden

- **Uncurrying**: in some functional languages (e.g., OCaml), functions always take a single argument at a time
  - E.g., in `let f x y = \ldots, f` takes one argument `x`, and returns a closure which takes a second argument `y` and produces the result
  - A single OCaml-level function call may result in *several* function calls and closure allocations
  - *Uncurrying* is an optimization that determines when a function is always called with more that one parameter (`f 3 4`), and compiles it as a multi-parameter function.
Garbage collection
Garbage collection

• Many modern languages feature garbage collectors, which automatically reclaim memory that was allocated by a program but no longer used
• Garbage collection is usually the job of a language runtime
  • Usually, the most complicated part
• A memory location is *garbage* if it will not be used in the remainder of the program.
• Determining whether it will not be used is undecidable.
  • *But*, we are happy with a conservative approximation: free memory if it *cannot possibly be used* in the remainder of the program.
• Usually not a *static analysis*, but rather a *dynamic analysis*.
  • *Static analyses* collect information about a program without running it.
  • *Dynamic analyses* collect information about a program while running it.
Reference counting

- Each memory location gets an extra int field to hold the number of active references to that memory
- Collect when count is zero
Reference counting

- Each memory location gets an extra int field to hold the number of active references to that memory
- Collect when count is zero
- *Cyclic* data structures never get collected
Reference counting

- Each memory location gets an extra int field to hold the number of active references to that memory
- Collect when count is zero
- *Cyclic* data structures never get collected
Reference counting

- Each memory location gets an extra int field to hold the number of active references to that memory
- Collect when count is zero
- *Cyclic* data structures never get collected
Reference counting

- Each memory location gets an extra int field to hold the number of active references to that memory
- Collect when count is zero
- *Cyclic* data structures never get collected
Tracing-based GC

- **Tracing garbage collection**: a memory location is garbage if it is unreachable from the program's roots
  - *roots* = register, stack, global static data
Mark-and-sweep

• Each memory location gets an extra bit to hold a “mark”
• When there is no remaining free memory, run a DFS search from the roots, marking all memory locations
• Traverse the entire heap; unmarked nodes are collected
• Generational GC
  • Most memory becomes garbage quickly after allocation
  • Memory that does not quickly become garbage is likely to not be garbage for a very long time
  • So: maintain several heaps (“generations”) $G_0$, $G_1$, ...
    • Allocate in $G_0$, and scan frequently
    • Scan $G_1$ less frequently, $G_2$ less frequently than that, ...
    • After collecting garbage in $G_i$, non-garbage is promoted to $G_{i+1}$