Programming Languages and Techniques (CIS120e)

Lecture 10
Sep 29, 2010
Abstract Stack Machine & First-Class Functions

Announcements

- Homework 3 is due tonight at 11:59:59pm.

- Homework 4 will be available soon.
  - Due next Weds. (Oct. 6th)

- Midterm 1 will be in class on Friday, October 15th.

Abstract Machines

- The job of a programming language is to provide some abstraction of the underlying hardware
- We can think of this abstraction as a thing in its own right — an abstract machine
- There are lots of ways of visualizing abstract machines. You saw last time: A stack machine
- This model...
  - is a good way of understanding how recursive functions work
  - gives an accurate picture of how OCaml data structures are shared internally (which helps predict how fast programs will run), and
  - will extend smoothly to include imperative features (assignment, pointer manipulation) and objects

Stack Machine

- Three “spaces”
  - workspace
    - contains the expression we are currently working on simplifying
  - stack
    - temporary storage for remembering bindings and partially simplified expressions
  - heap
    - storage area for large data structures

- Initial state:
  - workspace contains whole program
  - stack and heap empty
Values and the Heap

A value is either:
- a primitive value like an integer, or,
- a pointer into the heap

The heap contains two kinds of data:
- a cell, labeled by a datatype constructor, and containing
  arguments of the constructor
  - the arguments are themselves values.
- a function, written*
  \[
  \text{fun } (x_1:t_1)...(x_n:t_n) \rightarrow e
  \]

*Simplification

The abstract machine operates by repeatedly looking for the first
(leftmost) “ready subexpression” in the workspace and
simplifying it...
- A let-expression “let x = e in body” is ready if the
  expression e is a value
  - it is simplified by adding a binding of x to e at the end of the stack and
    leaving body in the workspace
- A variable is always ready
  - it is simplified by replacing it with its value from the stack, where
    binding lookup goes in order from most recent to least recent
- A primitive operator (like +) is ready if both of its arguments
  are values
  - it is simplified by replacing it with the result of the operation

Simplifying Datatypes

- A datatype constructor (like \texttt{Nil} or \texttt{Cons}) is ready if all its
  arguments are values
  - it is simplified by:
    - creating a new heap cell labeled with the constructor and containing
      the argument values*
    - replacing the constructor expression in the workspace by a pointer to this
      heap cell

*Simplifying Match

- A match expression
  \[
  \text{begin match } e \text{ with }
  \begin{align*}
  \text{ | } & \text{pat}_1 \rightarrow \text{branch}_1 \\
  \text{ | ...} \\
  \text{ | } & \text{pat}_n \rightarrow \text{branch}_n \\
  \text{end}
  \end{align*}
  \]
  is ready if e is a value
  - Note that e will always be a pointer to a constructor cell in the heap
  - This expression is simplified by finding the first pattern \texttt{pat}_1 that
    matches the cell and adding new bindings for the pattern variables (to
    the parts of e that line up) to the end of the stack
  - replacing the whole match expression in the workspace with the
    corresponding \texttt{branch}_1

*Note: in OCaml, using a datatype constructor causes some space to be automatically allocated on the heap.
Other languages have different mechanisms for accomplishing this: for example, the keyword ‘new’ in Java does
is similar (as we’ll see in a few weeks).
Simplifying Functions

• A function definition “let rec \(f\) \((x_1::t_1)\)\(\ldots\)\((x_n::t_n)\) = e in body” is always ready.
  – It is simplified by replacing it with “let \(f\) = fun \((x::t)\) = e in body”

• A function “fun \((x_1::t_1)\)\(\ldots\)\((x_n::t_n)\) = e” is always ready.
  – It is simplified by moving the function to the heap and replacing the function expression with a pointer to that heap data.

• A function call is ready if the function and its arguments are all values
  – it is simplified by
    • saving the current workspace contents on the stack
    • adding bindings for the function’s parameter variables (to the actual argument values) to the end of the stack
    • copying the function’s body to the workspace

Example

```ocaml
let rec append (l1: 'a list) (l2: 'a list) : 'a list = begin match l1 with
  | Nil -> l2
  | Cons(h, t) -> Cons(h, append t l2)
end in

let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in
append a b
```

Completion

When the workspace contains just a single value, we pop the stack by removing everything back to (and including) the last saved workspace contents.

The value currently in the workspace is substituted for the function application expression in the saved workspace contents, which are put back into the workspace.

If there aren’t any saved workspace contents in the stack, the whole computation is finished and the value in the workspace is its final result.

“Blackboard” Animation

see append.pdf
The first few steps of Simplification

Workspace | Stack | Heap
---|---|---
let rec append (l1: 'a list) (l2: 'a list) = begin match l1 with | Nil -> l2 | Cons(h, t) -> Cons(h, append t l2) end in let a = Cons(1, Nil) in let b = Cons(2, Cons(3, Nil)) in append a b

Function Definition

Workspace | Stack | Heap
---|---|---
let rec append (l1: 'a list) (l2: 'a list) = begin match l1 with | Nil -> l2 | Cons(h, t) -> Cons(h, append t l2) end in let a = Cons(1, Nil) in let b = Cons(2, Cons(3, Nil)) in append a b

Rewrite to a “fun”

Workspace | Stack | Heap
---|---|---
let append = fun (l1: 'a list) (l2: 'a list) => begin match l1 with | Nil => l2 | Cons(h, t) => Cons(h, append t l2) end in let a = Cons(1, Nil) in let b = Cons(2, Cons(3, Nil)) in append a b

Function Expression

Workspace | Stack | Heap
---|---|---
let append = fun (l1: 'a list) (l2: 'a list) => begin match l1 with | Nil => l2 | Cons(h, t) => Cons(h, append t l2) end in let a = Cons(1, Nil) in let b = Cons(2, Cons(3, Nil)) in append a b
Copy to the Heap, Replace w/Pointer

Workspace
let append = in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in
append a b

Stack

Heap
fun {l1: 'a list} {l2: 'a list} ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end

Let Expression

Workspace
let append = in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in
append a b

Stack

Heap
fun {l1: 'a list} {l2: 'a list} ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end

Create a Stack Binding

Workspace
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in
append a b

Stack

Heap
fun {l1: 'a list} ->
begin match l1 with
| Nil -> 12
| Cons(h, t) ->
  Cons(h, append t 12)
end

First-class Functions

see funs.ml

Note that the pointer to a function in the heap is a value.
First-Class Functions Revisited

```ocaml
let rec map (f:'a -> 'b) (l:'a list) : 'b list =
begin match l with
  | []   -> []
  | h::t -> (f h)::(map f t)
end
```

Anonymous Functions

- In OCaml, functions are \textit{values}.
  - We saw this in the abstract machine.
- The syntax an un-named (anonymous) function is the same as what we saw there:
  - Note that, unlike named functions, there is no "return type" annotation.

```ocaml
fun (x_1:t_1) (x_n:t_n) -> e
```

- For example, to map an "increment" function across a list of integers, we can write:

```ocaml
map (fun (x:int) -> x + 1) [1;2;3]
```

- This program will simplify to [2;3;4]