Discussion 9

October 25

Logistics

- Midsemester survey due next Friday
- A6 due next Friday
- A7 out next Friday

Agenda

- Imperative ALFA
- Imperative OCaml
    - ref cells
    - encapsulated commands

Imperative ALFA

Side Effects

- A lot can be done with pure expressions

- But we side effects for:
  - Reading and writing from memory
  - Writing to the console
  - Displaying images
  - Networking

- Let’s expand our language with this capability

Imperative ALFA

	Structural Syntax	Concrete Syntax	
$\tau ::=$	$\ldots$		
	$\mid \mathrm{Ref}(\tau)$	τ ref	
$e ::=$	$\ldots$		
	$\mid \mathrm{Alloc}(e)$	alloc (e)	(intro. form)
	$\mid \mathrm{Deref}(e)$	$!e$	(elim. form)
	$\mid \mathrm{Assign}(e, e)$	$e := e$	(elim. form)
	$\mid \mathrm{Loc}[\ell]$	$\#\ell$	

New Imperative ALFA Rules

$$\frac{e \parallel \mu_0 \Downarrow v \parallel \mu_1 \qquad (\ell \text{ fresh})}{\mathsf{alloc}(e) \parallel \mu_0 \Downarrow \#\ell \parallel \mu_1, \ell \hookrightarrow v} \; (\text{Eval-Alloc}) \qquad \frac{}{\#\ell \; \textbf{val}} \; (\text{V-Loc})$$

$$\frac{e \parallel \mu \Downarrow \#\ell \parallel \mu' \qquad \mu'[\ell] = e'}{!e \parallel \mu \Downarrow e' \parallel \mu'} \; (\text{Eval-Deref})$$

$$\frac{e_1 \parallel \mu_1 \Downarrow \#\ell \parallel \mu_1' \qquad e_2 \parallel \mu_1' \Downarrow v_2 \parallel \mu_2, \ell \hookrightarrow \_}{e_1 := e_2 \parallel \mu_1 \Downarrow () \parallel \mu_2, \ell \hookrightarrow v_2} \; (\text{Eval-Assign})$$

Changed Rules

- We have to modify our rules to account for side effects. For instance:

$$\frac{e_1 \mathrel{\|} \mu_1 \ \Downarrow\ \underline{n_1} \mathrel{\|} \mu_1' \qquad e_2 \mathrel{\|} \mu_1' \ \Downarrow\ \underline{n_2} \mathrel{\|} \mu_2}{e_1 + e_2 \mathrel{\|} \mu_1 \ \Downarrow\ \underline{n_1 + n_2} \mathrel{\|} \mu_2}\ \text{(Eval-Plus)}$$

Order matters?

- In *non-imperative* ALFA, $e_1 + e_2 \equiv e_1 + e_2$ in general (how can you prove this?)
  - `5 + 6 ≡ 6 + 5`
  - `(if true then 0 else 1) + (5 + 6) ≡ (5 + 6) + (if true then 0 else 1)`

- What about now (in imperative ALFA)?

No, it is not always the case that $e_1 + e_2 \equiv e_1 + e_2$

Order matters!

- Consider (this very contrived example):

```
let x = ref true in
(x := false; 5) + (if !x then 5 else 6)
```

- Possible results?

Example

$$\frac{e \parallel \mu_0 \Downarrow v \parallel \mu_1 \qquad (\ell \text{ fresh})}{\mathsf{alloc}(e) \parallel \mu_0 \Downarrow \#\ell \parallel \mu_1, \ell \hookrightarrow v} \text{ (Eval-Alloc)}$$

$$\frac{e \parallel \mu \Downarrow \#\ell \parallel \mu' \qquad \mu'[\ell] = v}{!e \parallel \mu \Downarrow v \parallel \mu'} \text{ (Eval-Deref)}$$

$$\frac{v \textbf{ val}}{v \parallel \mu \Downarrow v \parallel \mu} \text{ (Eval-Val)}$$

$$\frac{e_1 \parallel \mu \Downarrow v_1 \parallel \mu_1 \qquad [v_1/x]e_2 \parallel \mu_1 \Downarrow v \parallel \mu'}{\mathsf{let}\ x : \tau?\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 \parallel \mu \Downarrow v \parallel \mu'} \text{ (Eval-Let)}$$

$\text{let } r = \text{alloc}(\underline{490}) \text{ in } !r \parallel \bullet \Downarrow \underline{490} \parallel \ell_r \hookrightarrow \underline{490}$

Example Solution

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\underline{490}\ \mathrm{val}}\ (\text{V-NumLit})
}{
\underline{490} \parallel \cdot \Downarrow \underline{490} \parallel \cdot
}\ (\text{Eval-Val})
}{
\mathrm{alloc}(\underline{490}) \parallel \cdot \Downarrow \#\ell_r \parallel \ell_r \hookrightarrow \underline{490}
}\ (\text{Eval-Alloc})
\qquad
\dfrac{
\dfrac{
\dfrac{}{\#\ell_r\ \mathrm{val}}\ (\text{V-loc})
}{
\#\ell_r \parallel \ell_r \hookrightarrow \underline{490} \Downarrow \#\ell_r \parallel \ell_r \hookrightarrow \underline{490}
}\ (\text{Eval-Val})
}{
!\#\ell_r \parallel \ell_r \hookrightarrow \underline{490} \Downarrow \underline{490} \parallel \ell_r \hookrightarrow \underline{490}
}\ (\text{Eval-Deref})
}{
\mathrm{let}\ r = \mathrm{alloc}(\underline{490})\ \mathrm{in}\ !r \parallel \cdot \Downarrow \underline{490} \parallel \ell_r \hookrightarrow \underline{490}
}\ (\text{Eval-Let})
$$

Refs in OCaml

Refs in OCaml

- We’ve used OCaml (as with Hazel) as a pure functional language
  - … no explicit memory allocation, looping, etc
- OCaml does have the capability for memory allocation, mutable variables, and loops!

Refs in OCaml

```
ref x (* allocate/create reference *)

!x (* dereference reference *)

x := e (* ref assignment *)

e1; e2 (* evaluate e1, may have side effects,
then evaluate e2 *)
```

Refs in OCaml

- What does this code evaluate to?

```
let x = ref 5 in
  x := 7;
  (!x) + 1
```

8

Ref-based Linked Lists

- OCaml's built-in Linked Lists are functional, but we can easily implement an imperative version

```
type ‘x mutlist = ‘x cell ref
and ‘x cell = MNil | MCons of ‘x * ‘x mutlist

let init () : ‘x mutlist = ref MNil
let push (x : ‘x) (xs : ‘x mutlist) : unit =
  xs := MCons (x, ref (!xs))

let xs0 = init ()
let xs1 = xs0
push 0 xs0; push 1 xs1
!xs0  (* {contents = MCons (1, {contents = MCons (0, {contents = MNil})})} *)
```

IO Encapsulation

with monads!

Why side effects

- Necessary to do “real” things
  - File, network I/O
  - Nondeterminism
  - Communicate with the real world
- Performance
  - Imperative algorithms

- “Real programming language”

Why not side effects?

- Loss of purity
  - Loss of equational reasoning
  - Loss of potential parallelism

- How can we maintain many of the benefits of pure functional programming?

Could we separate the effectful computations from the pure ones?

Separation the side effects

- **Goal:** separate pure expressions and side effects through *encapsulation*
  - Encode effectful computations as values of a “command” type
  - We can now treat effects like any other expression
  - These “recipes” can be passed around and executed separate from pure code

- This is called **phase distinction**

Example: reading input

- From the OCaml standard library we have
  - `print_string : string → unit` (prints a string to stdout)
  - `read_line    : unit → string` (reads a line from stdin)

- Consider:

```
let first = read_line () in
let last = read_line () in
print_string (first ^ " " ^ last ^ "!\n");
(first, last)
```

Example: reading input — io type

```
(* Description of effectful computations as pure expressions *)
type _ io¹ =
    | Print  : string → unit io
      (* command for print_String *)
    | Read   : unit → string io
      (* command for read_line *)
    | Bind   : `a io → (`a → `b io) → `b io
      (* operator to combine two commands *)
    | Return : `a → `a io
      (* operator to return a value as a command *)

(* Imperative program to actually execute the commands *)
exec : `a io → `a
```

[1]This is a *generalized algebraic datatype* (*GADT*)

Example: reading input – understanding `Bind`

```
Bind : `a io → (`a → `b io) → `b io
```

- We often want to use the result of a previous command
- `Bind (cmd, fun v → cmd')`
  - Take a command `cmd` and evaluate it into some result
  - Funnel the result as v into `cmd'` and return the resulting command

Example: reading input — exec

```
type _ io =
    | Print  : string → unit io
    | Read   : unit → string io
    | Bind   : `a io → (`a → `b io) → `b io
    | Return : `a → `a io
```

```
let rec exec : type¹ a. a io → a = fun cmd →
    match cmd with
    | Print s → print_string s
    | Read () → read_line ()
    | Bind (cmd, f) → exec (f (exec cmd))
    | Return v → v
```

[1]Polymorphic syntax for locally abstract types, necessary when writing recursive functions on GADTs

Example: reading input – encapsulate!

```
type _ io =
      | Print  : string → unit io
      | Read   : unit → string io
      | Bind   : `a io→ (`a → `b io) → `b io
      | Return : `a → `a io
let rec exec : type a. a io → a = …
```

```
let first = read_line () in
let last = read_line () in
print_string (first ^ “ ” ^ last ^ “!\n”);
(first, last)
```

```
let cmd : (string, string) io =
    Bind (Read (), fun (first : string) →
    Bind (Read (), fun (last  : string) →
    Bind (Print (first ^ “ ” ^ last ^ “!”, fun () →
         Return (first, last)))))
in (exec cmd : (string, string))
```

Cool! but syntax sucks

- Typing `Bind` over and over again gets old fast
  - Also pretty hard to read
- Can we apply some syntactic sugar?

- **Solution:** `let*`

let* in OCaml

```
(* OCaml lets us define our own “let operators” *)

let (let*) : `a → `b → `c = fun (a : `a) (b : `b) → … in


(* such that *)

let* x = e₁ in e₂


(* is treated as syntactic sugar for *)

(let*) e₁ (fun x → e₂)
```

Example: reading input — binding with let*

```
let (let*) = fun cmd f → Bind (cmd, f) in

(* recall: Bind (cmd, f) evaluates as exec (f (exec cmd)) *)
```

```
let cmd : (string, string) command =
      Bind (Read (), fun (first : string) →
      Bind (Read (), fun (last  : string) →
      Bind (Print (first ^ “ ” ^ last ^ “!”,
            fun () → Return (first, last)))))
in (exec cmd : (string, string))
```

```
let cmd : (string, string) command =
    let* first = Read () in
    let* last  = Read () in
    let* () = Print (first ^ “ ” ^last ^ “!”) in
    Return (first, last)
in (exec cmd : (string, string))
```

Monads

- **Monads**(!) capture the essence of this encapsulation
  - Higher-order (or parametric) type, and two operators: `bind` and `return`
  - `bind` "combines" instances of the type—what this means depends on what the type is, as long as they obey the *monad laws*
    - `return` is left- and right-identity for `bind`; and `bind` is essentially associative
  - Very very applicable
- Further reading (big research area)
  - "Notions of computation and monads" (Moggi 1991)
  - "Monads for functional programming" (Wadler 1993)
  - "Call-by-push value: a subsuming paradigm" (Levy 1999)