Administrative Notes

Reading: today's lecture based on Chapter 10.
Read papers on abstract data types for Monday and Tuesday:

- Liskov et al, Abstraction Mechanisms in CLU
- Guttag, Abstract Data Types and the Development of Data Structures

Class on Monday at 11am in 373 Soda.

Administrative Notes

Programs written in *imperative* languages consist of a sequence of *commands* that modify a global program state.

PostFix is an example of an imperative language.

In principle any sequence of states could be described as an expression, with each state being the value of an intermediate subexpression.

```
c = n; z = 0;
while (c > 0) {
    z = f(z);
    c -= 1;
}
```

sets `z` to the same value returned by the expression `G(n, 0)`, where (assuming all quantities to be integers, and `f` references no global variables),

```
int G(int c, int z) {
    return (c > 0) ? G(c-1, f(z)) : z;
}
```

Semantics of Imperative Languages

First, add a sequencing construct, `begin`, which wasn't useful in FL. The operational semantics for begin are:

$$(\texttt{begin}\ V\ E) \Rightarrow E \qquad \text{[begin-return]}$$

$$\frac{E_1 \Rightarrow E_1'}{(\texttt{begin}\ E_1\ E_2) \Rightarrow \texttt{begin}\ E_1'\ E_2)} \quad \text{[begin-progress]}$$

The denotational semantics are:

$$\mathcal{E}[\![(\texttt{begin}\ E_1\ E_2)]\!] = \lambda\ e\ .\ (\textit{with-value}(\mathcal{E}[\![E_1]\!]e)(\lambda\ v\ .\ (\mathcal{E}[\![E_2]\!]e)))$$

The full language also allows for a sequence of expressions in a `begin` which are desugared in the obvious way. This is not particularly interesting without state changes.

Mutable Data: FL!

Rather than making every value in `FL` assignable, `FL!` defines an explicit set of cells that are assignable.

$$
\begin{array}{lcll}
E & ::= & \ldots & [FLKExpressions] \\
  & | & (\texttt{cell}\ E) & [Cell] \\
O & ::= & \ldots & [FLKPrimop] \\
  & | & \texttt{cell-ref} & [Dereference] \\
  & | & \texttt{cell-set!} & [Assignment]
\end{array}
$$

- (`cell` E) returns a new mutable cell initialized to E.
- (`cell-set!` E_1 E_2) assuming E_1 is a cell, its contents are set to the value of E_2.
- (`cell-ref` E) assuming E is a cell, get its contents.

Operational Semantics of FLK!

In the SOS for FL, our configurations were simply expressions, while in Postfix, they were expression/stack pairs. In FLK!, we need the program store along with the expression.

$$
\begin{array}{lcllcl}
L & \in & \text{Location} & = & \text{Nat} \\
S & \in & \text{Store} & = & \text{Assignment*} \\
Z & \in & \text{Assignment} & = & \text{Location} \times \text{Value}
\end{array}
$$

We will use a function *get* that finds the first value associated with location in the store.

get: Location $\rightarrow$ Store $\rightarrow$ Value

$$
\begin{array}{l}
(get\ L\ \langle L,\ V\rangle\ .\ S) = V \\
(get\ L_1\ \langle L_2,\ V\rangle\ .\ S) = (get\ L_1\ S),\ \text{where } L_1 \neq L_2
\end{array}
$$

Note that we could instead represent the store by a function.

SOS for FLK!

$$\frac{\langle E,\ S\rangle \Rightarrow \langle E',\ S'\rangle}{\langle(\texttt{cell}\ E),\ S\rangle \Rightarrow \langle(\texttt{cell}\ E'),\ S'\rangle} \qquad [\mathit{cell\text{-}progress}]$$

$$\langle(\texttt{cell}\ V), S\rangle \Rightarrow \langle(*cell*L_{fresh}), \langle L_{fresh},\ V\rangle\ .\ S\rangle \qquad [\mathit{cell\text{-}alloc}]$$

$$\langle(\texttt{cell-ref}\ (*\texttt{cell}*\ \ L)),\ S\rangle \Rightarrow \langle V,\ S\rangle$$
$$\text{where } (get\ L\ S) = V \qquad [\mathit{cell\text{-}ref}]$$

$$\langle(\texttt{cell-set!}(*\texttt{cell}*\ \ L)\ V),\ S\rangle \Rightarrow \langle\texttt{\#u},\ \langle L,\ V\rangle\ .\ S\rangle \quad [\mathit{cell\text{-}set}]$$

Extending FL Rules with a Store

In addition, the rules for `begin`, `if`, and all of the other `FL` constructors are extended to handle the extra state in each configuration. For example, the `begin` rules change as follows:

$$\langle (\texttt{begin}\ V\ \ E),\ S\rangle \Rightarrow \langle E,\ S\rangle \qquad \text{[begin-return]}$$

$$\frac{\langle E_1,\ S\rangle \Rightarrow \langle E_1',\ S'\rangle}{\langle (\texttt{begin}\ E_1\ E_2),\ S\rangle \Rightarrow \langle (\texttt{begin}\ E_1'\ E_2),\ S'\rangle} \ \text{[begin-progress]}$$

Recursion and the Store

Why do we have to be careful about the rules for recursion? Consider the following example:

```
(let ((counter (cell 0)))
   (begin ((rec fact
               (begin (cell-set! counter
                                 (+ (cell-ref counter) 1))
                      (lambda (n)
                         (if (= n 0)
                             1
                             (* n (fact (- n 1)))))))
            5)
           (cell-ref counter)))
```

Using a CBN style rule for `rec`, the counter will be incremented 6 times, once for each decrement of n in the fact body. We therefore use a CBV style rule.

Rules for Rec in FLK!

The SOS rules for `rec` delay eveluation of the recursive expression until we have a value in the `rec` expression, e.g., a lambda expression.

$$\frac{\langle E,\ S\rangle \Rightarrow \langle E', S'\rangle}{\langle(\texttt{rec}\ I\ E),\ S\rangle \Rightarrow \langle(\texttt{rec}\ I\ E'),\ S'\rangle} \quad [rec\text{-}body]$$

$$\langle(\texttt{rec}\ I\ V),\ S\rangle \Rightarrow \langle[(\texttt{rec}\ I\ V)/I]V,\ S\rangle \quad [rec]$$

Semantic Domains for FLK!

The denotational semantics has to be extended in a similar manner to the operataional, i.e., by adding a store. Here, by convention, we represent stores as functions.

$$s \in Store = Location \to Assignment$$

$$l \in Location = Nat$$

$$\alpha \in Assignment = (Storable + Unassigned)$$

$$\sigma \in Storable = Value \qquad \text{language dependent}$$

$$Unassigned = \{unassigned\}$$

Why are their two mappings, both environments and stores?

Denotationl Semantics in General

We now have three interesting domains for each language:

- expressible

- denotable

- storeable

Denotational Semantics of FLK!

$$
\begin{aligned}
c &\in \mathit{Computation} = \mathit{Store}(\mathit{Expressible} \times \mathit{Store}) \\
\delta &\in \mathit{Denotable} = \mathit{Value} \\
\sigma &\in \mathit{Storable} = \mathit{Value} \\
e &\in \mathit{Expressible} = (\mathit{Value} + \mathit{Error})_\perp \\
v &\in \mathit{Value} = \ldots + \mathit{Location}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E} &: \mathrm{Exp} \to \mathit{Environment} \to \mathit{Computation} \\
\mathcal{P} &: \mathrm{Primop} \to \mathit{Value}* \to \mathit{Computation}
\end{aligned}
$$

Where $\mathcal{P}$ is the primitive operator evaluation function, for FLK!, which behaves as in FLK for non-cell operators, none of which affect the state.

Denotational Semantics of FLK!

$$\mathcal{E}[\![(\texttt{cell}\ E)]\!] = \lambda e\ .\ (\mathit{with\text{-}value}(\mathcal{E}[\![E]\!]e)$$
$$\lambda v\ .\ (\mathit{allocating}\ v\ (\lambda\ l\ .$$
$$(\mathit{val\text{-}to\text{-}comp}\ (\mathit{Location} \mapsto \mathit{Value}\ l)))))$$

$$\mathcal{E}[\![(\texttt{begin}\ E_1\ E_2)]\!] = \lambda e\ .\ (\mathit{sequence}(\mathcal{E}[\![E_1]\!]e)(\mathcal{E}[\![E_2]\!]e))$$

where the helping functions are defined as:

$$\mathit{sequence} : \mathit{Computation} \to \mathit{Computation} \to \mathit{Computation}$$
$$= \lambda c_1\ c_2\ .\ (\mathit{with\text{-}value} c_1(\lambda v\ .\ c_2))$$

$$\mathit{allocating} : \mathit{Storable} \to (\mathit{Location} \to \mathit{Computation}) \to \mathit{Computation}$$
$$= \lambda \sigma\ f\ .\ \lambda\ s\ .\ (f(\mathit{fresh\text{-}loc}\ s)(\mathit{assign}\ (\mathit{fresh\text{-}loc}\ s)\ \sigma\ s))$$

Valuation Functions for Cell Primitives

$$\mathcal{P}[\![\texttt{cell-ref}]\!] = \lambda\ [v]\ .\ (\textit{check-location}\ v$$
$$(\lambda\ l\ .\ (\textit{fetching}\ l\ (\lambda\ v.(\textit{val-to-comp}\ v)))))$$

$$\mathcal{P}[\![\texttt{cell-set!}]\!] = \lambda[v_1, v_2]\ .$$
$$(\textit{check-location}\ v_1\ (\lambda\ l\ .\ (\textit{update}\ l\ v_2)))$$

$$\mathcal{P}[\![O]\!] = \lambda v * .(\textit{expr-to-comp}(\mathcal{P}_{FLK}[\![O]\!]))$$

Making Variable Mutable: FLAVAR!

$\delta \in Denotable = Location$
$\sigma \in Storable =$ depends on parameter passing mechanism

$\mathcal{E}[\![I]\!] =$ depends on parameter passing mechanism
$\mathcal{E}[\![(call\ E_1\ E_2)]\!] =$ depends on parameter passing mechanism

$$\begin{aligned}\mathcal{E}[\![(\texttt{set!}\ I\ E)]\!] = \lambda e\ .\ (&with\text{-}value(\mathcal{E}[\![E]\!]\ e)\\ &(\lambda\ v\ .\ (with - denotable\ (lookup\ e\ I)\\ &\quad(\lambda\ l\ .\ (update\ l(val\text{-}to\text{-}storable\ v))))))\end{aligned}$$