Store And Env!

CS 329-14-08-2002 **Varun Kacholia**

Need For Store :

The Env maps names to values which do not change (are immutable). We can shadow one mapping by another in an inner (static or dynamic) scope by suitably extending the Env, but at a given place in the code, a name can only bind to a fixed value.

The Store has a different purpose. It has no notion of name scope. It is accessed using "Locations" (pointers, memory addresses) taken from a "flat" space without any hierarchical scoping. Thus Store has the notions of **time** and **mutability** which the Env does not.

Store is a **flat** set of cells with each cell being assosiated with a unique Location which can be used to de-reference the contents of the cell aided with the capability of cell holding any value viz Int,Bool,Proc etc(rather than fixed-size as in von-Neumann architecture).

So far , we have thought of variables as being bound directly to value.To introduce variable assignment ,we instead think of variables as being bound to **locations**.Variable references now reder to the values contained in the locations.

Syntax for FLK!

E:: | (new E_v) : makes E to value;points it in a cell and returns that location.
| (read E_l) : reads the location
| (write E_l E_m) : returns the new value E_m

Semantic Domains

Store = Loc $\rightarrow$ Value + Error
New : Store $\rightarrow$ Loc x Store
Store $\rightarrow$ Val $\rightarrow$ Loc x Store (Pair)

Operational Semantics

1. (predicate involving $< E_{old}, S_{old} >$) / (predicate involving $< E_{new}, S_{new} >$)
2. $(E_1 \Rightarrow E_1^{'})/((E_1\ E_2) \Rightarrow (E_1^{'}\ E_2))$
3. $(E_2 \Rightarrow E_2^{'})/((E_1\ E_2) \Rightarrow (E_1\ E_2^{'}))$

4. * / $(((\lambda\ I_a\ E_b)\ E_a\ ) \Rightarrow [E_a/I_a]\ E_b)$

Making it deterministic

1. $(E_1 \Rightarrow E_1^{'})/((E_1\ E_2) \Rightarrow (E_1^{'}\ E_2))$
2. $(E_2 \Rightarrow E_2^{'})\ /\ ((\lambda\ I\ E)\ E_2) \Rightarrow (\lambda\ I\ E)\ E_2^{'})$
3. * / $(((\lambda\ I\ E)\ V_a\ ) \Rightarrow [V_a/I]\ E)$

where E :: general expresion & V:: Value

Sequencing in FLK!

Sequencing refers to the order in which *side-effects* are performed. In statement-oriented languages, sequencing is indicated by the order of the statements . In a expression-oriented language , sequencing is required only for expressions that result in side effect.

As long as we did not have a Store in FLK, we did not need any language construct for sequencing operations. (Depending on evaluation strategy, our program might or might not terminate, but if it did, the answer did not depend on the order of applying rules.) But now we need more explicit control on the order of operations on the Store.

So we introduce the new language construct "begin", used as
(begin E_1 E_2 ... E_n)

The idea is that E_i will be interpreted (executed) to normal form before E_{i+1} is touched, and the value of the whole expression is the value of E_n. All previous E_i's act as **Store-Transformers** .

- $(<E_1, S_1> \Rightarrow <E_1^{'}, S_1^{'}>)\ (\ <\ (\text{begin}\ E_1\ E_2)\ ,\ S_1> \Rightarrow <\ (\text{begin}\ E_1^{'}\ E_2)\ ,\ S_1^{'}>)$
- $(<E_2, S_2> \Rightarrow <E_2^{'}, S_2^{'}>)\ /\ (\ <\ (\text{begin}\ V_1\ E_2)\ ,\ S_2> \Rightarrow <\ (\text{begin}\ V_1\ E_2^{'})\ ,\ S_2^{'}>)$
- $<(\text{begin}\ V_1\ V_2)\ ,\ S\ > \Rightarrow <\ V_2\ ,\ S\ >$

FLK! Denotation

ϵ : Expr $\rightarrow$Env $\rightarrow$ Store $\rightarrow$ExpVal x Store (pair)

ϵ[(aproc I E)] = λu λs λe (ϵ[E] a$\frac{I->e}{u}$ s)

But this is wrong as is assosiates **Static scope!**

Proc :: ExpVal $\rightarrow$Store $\rightarrow$ ExpVal x Store

ϵ[(aproc I E)] = pair (λu λs λe (ϵ[E] $\frac{I->e}{u}$ s_{call}) , s_{decl})

- we can remove s_{call} $\rightarrow\eta$ rule !

- $<\ E_v, S\ > \Rightarrow\ <E_v^{'}, S^{'}>\ /\ <\ (\text{new}\ E_v), S\ > \Rightarrow\ <(\text{new}\ E_v^{'}\ ), S^{'}>$

- $(\underline{\text{new}}\ S\ V) \Rightarrow < L_{fresh}, S' > / <(\text{new}\ V)\ , S> \Rightarrow < L_{fresh}, S' >$
- $<E_l, S> \Rightarrow <E_l', S'> / <(\ \textit{read}\ E_l)\ , S > \Rightarrow <(\ \textit{read}\ E_l')\ , S' >$
- $<(\ \textit{read}\ L)\ , S > \Rightarrow <(\ \underline{\textit{read}}\ S\ L)\ , S >$
- $\epsilon[(\ \text{new}\ E)\ ] = \lambda u\ \lambda s_1$

  let $<v\ , s_2> = (\epsilon[E]\ u\ s_1)$
  let $<L, s_3> = (\underline{\text{new}}\ s_2\ v\ )$
  $< L\ , s_3>$

- $\epsilon\ [\ (\ \text{new}\ E)\ ] = \lambda u\ \lambda s_1$

  $\mathbf{(}\underline{\text{new}}\ (\text{Right}\ (\epsilon[E]\ u\ s_1)\ )\ (\text{Left}\ (\epsilon[E]\ u\ s_1)\ )\ \mathbf{)}$

- $\epsilon[(\ \text{read}\ E)\ ] = \lambda u\ \lambda s_1$

  let $<L\ , s_2> = (\epsilon[E]\ u\ s_1)$
  $< (\underline{\text{read}}\ s_2\ L\ )\ , s_2 >$

- $\epsilon\ [\ (\ \text{read}\ E\ )\ ] = \lambda u\ \lambda s_1$

  $< (\ \underline{\text{read}}\ \mathbf{(}\text{Right}\ (\epsilon[E]\ u\ s_1)\ \mathbf{)}\ \mathbf{(}\text{Left}\ (\epsilon[E]\ u\ s_1)\ \mathbf{)}\ , \mathbf{(}\text{Right}\ (\epsilon[E]\ u\ s_1)\ \mathbf{)} >$