Imperative Languages

Section 5 Continued ...

A Dynamically Typed Language

- Variable may take on values from different data types.
- Run-time type checking required.
- Input and output included.
- Introduction of ``type tags''
  - Storable-value = Tr + Nat
  - Store = Id → Storable-value
  - Errvalue = Unit
  - Expressible-value =
  - Storable-value + Errvalue

Valuation Functions

- **State = Store x Input x Output**
  - store, input and output buffers
- **Post-State = OK + Err**
  - OK = State, Err = State
  - successful evaluation or type error
- **P: Program → Store → Input → Poststate⊥**
  - Program takes input store and input buffer and returns a new state

- **C: Command → State → Post-state⊥**
  - Command takes state and returns a new state.
- **E: Expression → Store → Expressible-value**
  - Expression takes store and returns a value.

Composition of States

- **C[[C1 ; C2]] = C[[C1]] check-cmd C[[C2]]**

```
Command = State → Post-state ⊥
check-cmd: Command x Command → Command
h 1 check-cmd h 2 =
        λa. let z = h1(a) in
        cases z of
             isOK(s, i, o) → h 2 (s; i; o)
             isErr(s, i, o) → z
        end
```

1. Give current state a to C[[C 1]] producing a post-state z = C[[C 2]]a'.
2. If z is a proper state a' and if the state component is OK, produce C[[C 2]]a'
3. If z is errnoneous, C[[C 2]] is ignored and z is the result.

Similar for check of expression results.

Error Handling

- **Algebra operations abort normal evaluation when type error occurs.**
  - Representation of low-level (e.g. hardware-level) fault detection and branching mechanisms.
- **Machine action:** on fault, branch out of the program.
- **Semantics:** on fault, branch out of function expression
  - Propagation of type errors yields same result as machine action.

Altering the Properties of Stores

1. Store critical to evaluation of a phrase.
2. Only one copy of store exists during execution.
3. Store serves as means of communication between phrases.

Typical features of a store in sequential programming languages.
What happens if we relax each of these restrictions?

Delayed Evaluation

How to rewrite a function application f(e)?

- Call-by-value simplification
  - e evaluated before f is executed
  - Safe method if f is strict.
- Call-by-name simplification
  - f executed with non-evaluated e;
  - Safe method also if f is non-strict.
    - f: Nat ⊥ → Nat ⊥
    - f = λ x: zero
    - f(⊥) = zero
    - E[[e]] = ⊥
    - f(E[[e]])) ⇒ ?
- Simplification of argument may require in-finite number of steps!

Non-strict Store Updates

May store operate on improper values?

- **Store = Id → Nat⊥**
  - Improper values may be stored.
- **update: Id → Nat⊥ → Store → Store**
  - -- update = λ i: λ n: λ s:[i → n]s
  - -- update[[I]](E[[E]]s) s) is defined even in the``loop forever situation'' E[[E]]s = ⊥.
  - -- Unevaluated expressions may be stored in s.
- **E[[E]]s needs not be evaluated until used.**
  - -- Delayed (lazy) evaluation.
  - -- Value must be determined with respect to the store that was active when [[E]] was saved.

```
EXAMPLE:
begin
   X := 0
   Y := X+1
   X := 4
return Y
```

K: Block → Store ⊥ → Nat ⊥

K[[begin C return E]] =
λ s: E[[E]] (C[[C]]s)

K[[begin X:=0; Y:= X+1; X:=4 return Y]]s0

= **E[[Y]]** (C[[X:=0; Y:= X+1; X:=4]]s0)

= E[[Y]] (C[[Y:= X+1; X:=4]] **(C[[X:=0]]s0))**

= E[[Y]] (C[[Y:= X+1; X:=4]]
(update [[X]] (E[[0]]s0) s0))

= E[[Y]] (C[[Y:= X+1; X:=4]] **s1**)

s 1 = (E[[0]]s0) needs not be simplified!

EXAMPLE:

```
s2 = update [[Y]] (E[[X+1]]s1 ) s1
s3 = update [[X]] (E[[4]]s 2 ) s2

E[[Y]] (C[[Y:= X+1; X:=4]] s1 )
= E[[Y]]s3
= access [[Y]] s3
= E[[X+1]]s1
= E[[X]]s1 plus one
= (access [[X]] s1 ) plus one
= E[[0]]s0 plus one
= zero plus one
= one
```

Non-Strict Command Execution

- Carry delayed evaluation up to level of commands.
- Make C, E, and K non-strict in their store arguments.
- Only those commands need to be evaluated that have an effect on the output of a program!

EXAMPLE:

begin X:=0; diverge; X:=2 **return** X+1

```
K[[begin X:=0; diverge; X:=2 return X+1]]s0
= E[[X+1]] (C[[X:=0; diverge X:=2]]s0 )
= E[[X+1]] (C[[X:=2]](C[[diverge; X:=2]]s0 ))
= E[[X+1]](C[[X:=2]]s1 )
= E[[X+1]](update [[X]] (E[[2]]s1 ) s1 )
= E[[X+1]]([ [[X]] → (E[[2]]s1 ) ]s1 )
= E[[X]]([ [[X]] → (E[[2]]s1 ) ]s1 ) plus one
= E[[2]]s 1 plus one
= two plus one
= three
s 1 = C[[X:=0; diverge]]s 0 needs not be
simplified!
```

Retaining Multiple Stores

E[[E 1 + E 2]] = λ s.E[[E 1]]s plus E[[E 2]]s

- E uses store s in ``read only'' mode.
- No second store required for implementation.

E[[begin C return E]] =
λ s: let s’ = C[[C]]s in E[[E]]s’

- Side effects in expression evaluation possible.
- Multiple stores required for implementation.

```
E[[E 1 + E 2 ]]
= λ s: let (v’; s’ ) = E[[E 1 ]]s
(v’’ ; s’’ ) = E[[E 2 ]]s’
in (v’ plus v’’ ; s’’ )

E[[begin C return E]] =
λ s: let s’ = C[[C]]s in (E[[E]]s’, s’)
```

- Local updates remain in global store.
- Expression evaluation returns new store.

Non-Communicating Commands

- Store facilitates the building up of side effects that lead to some final value.
- Command advances computation by reading and modifying the values left in the store by previous commands.
- Otherwise, communication breaks down and language loses sequential flavor.

```
combine: D x D → D
Domain s2 Store = Id → D

C: Command → Store ⊥ → Store ⊥
C[[C 1;C 2]] = λs. join (C[[C1]]s) (C[[C2 ]]s)

join: Store ⊥ → Store ⊥ → Store ⊥
join = λs1 :λs2:(λi:s1 (i) combine s2 (i))

Parallel but non-interfering parallelism.
```