RefLang: a language about references/pointers

March 11, 2021

Side Effect

- Pure functional programs have no side effects: given the same input a functional program would produce the same output.
- Side effect: change the state of the program besides its output, i.e.,it can potentially effect other functions and programs.
- Examples:
  - Reading or writing memory locations
  - Printing on console
  - File read and file write
  - Throwing exceptions
  - Sending packets on network,
  - Acquiring mutual exclusion locks

Memory management

Types of memory where a user program stores their data during execution of the program:

- data section/static: allocated when the execution starts
- stack: local variables, function invocation
- heap: allocation and deallocation by user programs
  - memory allocation
  - memory deallocation
  - memory access: dereference (get values from memory location via pointers), reference (associate pointer with memory location)
  - memory operation

Heap and references

- **Heap**: an abstraction representing area in the memory reserved for dynamic memory allocation
- **References**: locations in the heap

Decisions for Language Designers – Heap

Heap size is finite, programming languages adopt strategies to remove unused portions of memory so that new memory can be allocated.

- manual memory management: the language provides a feature (e.g. free in C/C++) to deallocate memory and the programmer is responsible for inserting memory deallocation at appropriate locations in their programs.
- automatic memory management: the language does not provide explicit feature for deallocation. Rather, the language implementation is responsible for reclaiming ununsed memory (Java, C#) – garbage collection

Decisions for Language Designers – Heap

How individual memory locations in the heap are treated:

- untyped heap: the type of value stored at a memory location is not fixed, can be changed during program execution
- typed heap: each memory location has an associated type and it can only contain values of that type, the type of value stored at a memory location doesn't change during the program's execution

Decisions for Language Designers – Reference (pointers)

1. Explicit references: references are program objects available to the programmer
2. Implicit references: references only available to implementation of the language
3. Reference arithmetic: references are integers and thus we can apply arithmetic operations
4. Deref and assignment only: get the value stored at that location in the heap, assignment can change the value stored at that location in the heap

Decisions for Language Designers – Examples

- C: manual memory management, explicit reference, untyped heap, reference arithmetic
- Java: automatic memory management, deref and assignment only, untyped heap, implicit reference
- Reflang: manual memory management, deref and assignment, untyped heap, explicit references

RefLang

ref, free, deref, set!

- Allocate a memory location
- Free previously allocated memory location
- Dereference a location reference
- Assign a new value to an existing memory location

Examples:
$(ref 1)
loc: 0

- Return: the location at which memory was allocated (next available memory location)
- Side effect: assign value 1 to the allocated memory location

Reflang Expressions

ref: This expression evaluates its subexpression to a value, allocates a new memory location to hold this value, and returns a reference value that encapsulates information about the newly allocated memory location.

$ (define loc1 (ref 12))
// stores value 12 at some location in memory, creates a reference value to encapsulate (and remember) that location, and stores that reference value in variable loc1

$ (define loc2 (ref 45))

$ loc1 // check the reference value stored in variable loc1
loc:0

$ loc2
loc:1

Reflang Expressions

deref: This expression evaluates its subexpression to a value. If that evaluation evaluates to a reference value, and that reference value encapsulates a location l, then it retrieves the value stored in Heap at location l.

$ (deref loc1) // gives the value stored at loc1
12

$ (deref loc2) // gives the value stored at loc2
45

$ (+ (deref loc1) (deref loc2)) //access both values and adds them
57

$ (deref 12) // throws Dynamic error

Reflang Expressions

assign: This expression is used to change the value stored on some location in Heap. It will return the newly assigned value.

```
$ (set! loc1 23) //previous value 12 is overwritten by 23
23

$ (set! loc2 24) //previous value 45 is overwritten by 24
24

$ loc1 // loc1 still has address 0 but value has changed now
loc:0

$ loc2 // loc2 still has address 1 but value has changed now
loc:1

$ (+ (deref loc1) (deref loc2)) // different value different summation
value
47
```

Reflang Expressions

free: This expression is used to deallocate the reference stored in Heap.

$ (free loc1) // deallocates the memory address 0

$ loc1 // variable loc1 still points to same location loc:0

$ (deref loc1) // dereference loc1
Error:null // invalid because memory location has been freed

$ (free loc2) // deallocates the memory address stored in loc2

$ (deref loc2) // dereference loc2
Error:null // invalid because memory location has been freed

RefLang: More Examples

```
$ (free (ref 1)) // delocate the memory location where 1 is stored
$ (deref (ref 1)) // deref a memory location defined by ref 1

$ (let ((loc (ref 1))) (deref loc))
$ (let ((loc (ref 1))) (set! loc 2))

$(let ((loc (ref 10))) (let ((loc2 loc)) (+ (set! loc 1) (deref loc2))))
2
$ (let ((loc (ref 10))) (let ((loc2 loc)) (+ (deref loc2) (set! loc 1))))
11
```

Reflang: Grammar

Program	::=	DefineDecl* Exp?	*Program*
DefineDecl	::=	(define Identifier Exp)	*Define*
Exp	::=		*Expressions*
		Number	*NumExp*
	|	(+ Exp Exp+)	*AddExp*
	|	(- Exp Exp+)	*SubExp*
	|	(* Exp Exp+)	*MultExp*
	|	(/ Exp Exp+)	*DivExp*
	|	Identifier	*VarExp*
	|	(let ((Identifier Exp)+) Exp)	*LetExp*
	|	(Exp Exp+)	*CallExp*
	|	(lambda (Identifier+) Exp)	*LambdaExp*
	|	(ref Exp)	***RefExp***
	|	(deref Exp)	***DerefExp***
	|	(set! Exp Exp)	***AssignExp***
	|	(free Exp)	***FreeExp***

RefLang programming exercises

Write some RefLang programs

RefLang and FuncLang Programming

(11 pt) In this question you will implement a linked list. In a linked list, one element of the node is reference to another node. Each node will have two fields. First field of the node is a number while second element will be reference to other node, defined as:

$(define pairNode (lambda (fst snd) (lambda (op) (if op fst snd))))

(remember in lambda encoding, we use functions to represent data and operations, here is the similar idea).

i. (2 pt) define the head of the linked list with node 1

ii. (5 pt) write a lambda method 'add', which
   - takes two parameters
   - first parameter 'head' is head of linked list
   - second parameter 'ele' is a node
   - the function adds ele at the end of linked list, if successful, the value of the lambda method is ele.

iii. (4 pt) write a 'print' function
   - takes node as parameter (representing head of linked list)
   - returns a list of numbers present in linked list.

RefLang and FuncLang Programming

(a)

```
(define pairNode (lambda (fst snd) (lambda (op) (if op fst snd))))
(define node (lambda (x) (pairNode x (ref (list)))))
(define head (node 1))
```

(b)

```
(define getFst (lambda (p) (p #t)))
(define getSnd (lambda (p) (p #f)))

(define add
  (lambda (head ele)
    (if (null? (deref (getSnd head)))
        (set! (getSnd head) ele)
      (add (deref (getSnd head)) ele))))
```

Example scripts:

$ (getFst head)
1
$ (getSnd head)
loc: 0

RefLang and FuncLang Programming

(c)

```
(define print
  (lambda (head)
    (if (null? (deref (getSnd head)))
      (cons (getFst head) (list))
      (cons (getFst head)
          (print (deref (getSnd head)))))))
```

RefLang and FuncLang Programming

```
$ (add head (node 2))
(lambda ( op ) (if op fst snd))
$ (add head (node 3))
(lambda ( op ) (if op fst snd))
$ (print head)
(1 2 3)
$ (add head (node 0))
(lambda ( op ) (if op fst snd))
$ (add head (node 6))
(lambda ( op ) (if op fst snd))
$ (print head)
(1 2 3 0 6)
```

Reflang Interpreter

Semantics:

- values
- abstractions added? env, heap ...
- operational semantic rules

Reflang: Extending Values

- RefVal $\neq$ NumVal
  - prevent from accessing arbitrary memory location
  - no arithmetics
  - extra metadata

RefLang: Heap Abstraction

Heap : RefVal $\rightarrow$ Value

```
public interface Heap {
  Value ref (Value value) ;
  Value deref (RefVal loc) ;
  Value setref (RefVal loc, Value value) ;
  Value free (RefVal value) ;
}
```

- In the RefLang interpreter code, heap implementation helps you update the heap
- And, evaluator implementation (operational semantics) is about how to evaluate the expressions making use of the heap

Reflang Operational Semantics

- value p env h = value e env h
  In an environment env and a heap h, the value of a program is the value of its component expression e in the same environment env and the same heap h.
- Expressions that do not affect heap directly or indirectly:
  - Constant expression: value e env h = (NumVal n) h, where n is a Number, env is an environment, h is a heap
  - Variable expression – look up names for values:
    value (VarExp var) env h = get(env, var) h
- Expressions that indirectly affect heap through their subexpressions
- Expressions that directly affect heap

Reflang: Expressions that indirectly affect heap

- the order in which side effects from one sub-expression are visible to the next sub-expression
- Add/subtraction/multiplication/division expression:

$$\texttt{value (AddExp } e_0 \ \ldots \ e_n\texttt{) env h = } v_0 + \ldots + v_n\texttt{, } h_n$$

$$\texttt{if value } e_0 \texttt{ env h = } v_0 \ h_0\texttt{, } \ldots\texttt{, value } e_n \texttt{ env } h_{n-1} = v_n \ h_n$$

$$\texttt{where } e_0, \ \ldots, \ e_n \in \texttt{Exp, env} \in \texttt{Env, h, } h_0, \ldots h_n \in \texttt{Heap}$$

a left-to-right order is used in the relation above for side-effect visibility

Reflang: Expressions that directly affect heap

- ref, set!, free
- deref: read from memory only

Reflang: RefExp

$$\texttt{value (RefExp e) env h = l, h}_2$$

$$\texttt{if value e env h = v}_0\ \texttt{h}_1$$

$$\texttt{h}_2 = \texttt{h}_1 \cup \{\ \texttt{l} \mapsto \texttt{v}_0\ \} \qquad \texttt{l} \notin \texttt{dom(h}_1\texttt{)}$$

$$\texttt{where e} \in \texttt{Exp} \quad \texttt{env} \in \texttt{Env} \quad \texttt{h,h}_1\texttt{,h}_2 \in \texttt{Heap} \quad \texttt{l} \in \texttt{RefVal}$$

- The rule says, to compute the value of RefRxp e under env and current heap location and update the heap to h_2
  - If the value of e under same env and same h returns value v_0 in h_1
  - and heap is computed using the updated heap h_1 union RefVal l with the mapping to value v_0
    N.B. heap is mapping between the reference value to actual value that is stored in that location space.

Reflang: AssignExp

$$\frac{\texttt{value}\ e_1\ \texttt{env}\ h = v_0\ h_1 \qquad \texttt{value}\ e_0\ \texttt{env}\ h_1 = l\ h_2 \qquad h_3 = \{\ l \mapsto v_0\ \} \cup (h_2 \setminus \{\ l \mapsto \_\ \}) \qquad l \in \mathrm{dom}(h_2)}{\texttt{value (AssignExp}\ e_0\ e_1\texttt{)}\ \texttt{env}\ h = v_0,\ h_3}$$

$$\text{where } e \in \texttt{Exp} \quad \texttt{env} \in \texttt{Env} \quad h, h_1, h_2, h_3 \in \texttt{Heap} \quad l \in \texttt{RefVal}$$

- The rule says, to compute the value of AssignExp e_0 e_i under env under current heap location, (it directly affects heap) the result is the value v_0 and updated heap is h_3

  Below order of subexpression evaluation is important:
  - If value of e_1 is evaluated under env and h and you get a value v_0 and updated heap h_1
  - and then value of e_0 is evaluated under env and h_1 and it evaluates to a RefVal l and modify the heap to h_2
  - To compute h_3: add the pair (l $\rightarrow$ v_0) i.e., store v_0 in l and delete previously stored value (the underscore) from l in h_2.

Reflang: FreeExp

$$\frac{\texttt{value e env h = l } h_1 \qquad \texttt{l} \in \texttt{dom}(h_1) \qquad h_2 = h_1 \setminus \{\texttt{ l} \mapsto \_ \}}{\texttt{value (FreeExp e) env h = unit, } h_2}$$

where $e \in \texttt{Exp}$ $\texttt{env} \in \texttt{Env}$ $h, h_1, h_2 \in \texttt{Heap}$ $l \in \texttt{RefVal}$ $\texttt{unit} \in \texttt{Unit}$

- The rule says, to compute the value of FreeExp e under env and current heap location h, the result is unit value and updated heap is h_2
  - If value of e under env and h is evaluated and result is a RefVal l with updated h1
  - Note, if l is not under the domain of h_1, you can throw dynamic error
  - To compute h_2: h2 becomes h1 and mapping from l to some value (represented by underscore) is deleted

Reflang: DerefExp

$$\texttt{value (DerefExp e) env h = v, } h_1$$

$$\texttt{if value e env h = l } h_1 \qquad \texttt{l} \in \texttt{dom}(h_1)$$

$$\{ \texttt{l} \mapsto \texttt{v} \} \subseteq h_1$$

$$\texttt{where e} \in \texttt{Exp} \quad \texttt{env} \in \texttt{Env} \quad \texttt{h}, h_1 \in \texttt{Heap} \quad \texttt{l} \in \texttt{RefVal} \quad \texttt{v} \in \texttt{Value}$$

- The rule says, to compute DerefExp e under env and heap h (it directly affects heap), the result will return v in updated h_1
  - As evaluation of e under env and heap h may modify the value of heap, hence it is updated to h_1 and l is a RefVal
  - Here, mapping of l to v belongs to the heap h_1

RefLang Implementation: Heap and Evaluator

See RefLang interpreter Code