Chapter 7

Reflang: A Language with References

Most programming languages have features that can produce side-effects, i.e. the programming language feature can change the state of the program besides its output. Some examples of side-effects include:

- reading or writing memory locations,
- printing on console, reading user input,
- file read and file write,
- throwing exceptions,
- sending packets on network,
- acquiring mutual exclusion locks, etc...

Although understanding programs that use side-effects is often more difficult compared to those that are functional[1], side-effects can be indispensable for certain use cases.

Although each kind of side-effect mentioned above has some unique semantic properties, semantic issues and design tradeoffs pertaining to reading or writing memory locations is a representative kind of side-effect.

[1]Functional programs can be understood in terms of their input and output. Given the same input a functional program would produce the same output.

7.1 Heap and References

To support reading or writing memory locations, typically programming languages include two new concepts in their definitions.

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

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

- *manual memory management:* In this model, 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:* In this model, the language does not provide explicit feature for deallocation. Rather, the language implementation is responsible for reclaiming ununsed memory. Languages like Java, C# adopt this model.

Programming languages also differ in how they support references.

- *explicit references:* In some languages, references are program objects available to the programmer. Examples of such languages includes C, C++.
- *implicit references:* In other languages, references are only available to the language implementation. Some actions of programs implicitly create references.

Languages also differ in what operations are supported on references.

- *reference arithmetic:* In some languages, references are treated as positive integers and all arithmetic operations on references are available to the programmer. Examples of such languages includes C, C++.
- *deref and assignment only:* In other languages, references can only be used for two operations: dereference to get the value stored at that location in the heap, and assignment to change the value stored at that location in the heap.

Last but not least, languages also differ in how individual memory locations in heap are treated.

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

Exercise

7.1.1. *[Heap]* Design and implement Heap, a new abstraction representing area in the memory reserved for dynamic memory allocation. For testing, you can assume the capacity of the heap to be 8 KB.

- The heap abstraction internally maintains a contiguous space of memory and a *free list*, which is a collection of 2-tuple (location, size) representing available space. The collection of tuples representing available space starts out with a single entry (0, size), where size is the capacity of the heap.
- The heap abstraction provides four operations: `alloc`, `get`, `set`, and `free`.
- The `alloc` operation takes a single parameter, size of desired memory space, a numeric value in the language. It scans the free list, and finds the first available space sufficient for allocation, returns that location, and adjusts free list to reflect this memory allocation.

  If contiguous space is not available, the alloc operation throws an exception of type `InsufficientMemoryException`.
- The `free` operation takes two parameters location and size, both are numeric value in the language. If the location is already in free list, it does nothing. Otherwise, it puts the location in the free list.
- The `get` operation takes a single parameter: location, which is a numeric value in the language, and returns value stored at

that location. If the location is in free list, an exception of type `SegmentationFault` is raised. If the location is out of bounds of the heap, again an exception of type `SegmentationFault` is raised.

- The `set` operation takes two parameters: location, which is a numeric value in the language, and value, which is also a numeric value to be stored at that location. If the location is in free list, an exception of type `SegmentationFault` is raised. If the location is out of bounds of the heap, again an exception of type `SegmentationFault` is raised. Otherwise, the heap is modified so that value is stored at location.

7.1.2. *[Fragmented heap]* Create an example allocation and deallocation test case for heap, which allocates `n` chunks of memory of size `s`, where `n` is `size/s`. Here, size is the capacity of the heap. The test case then frees every alternate chunk of memory to effectively free approximately half of the heap, i.e. 0th chunk, second chunk, fourth chunk and so on. Finally, the test case should try to allocate a chunk of memory of size `2s`.

7.1.3. *[Array allocation and access]* Extend the heap abstraction with three operations: `allocArray`, `getAt` and `setAt`, which allows treating a chunk of memory as a two-dimensional array. Given the row size and column size, the `allocArray` allocates a chunk of memory sufficient to hold the array. If contiguous space is not available, the `allocArray` operation throws an exception of type `InsufficientMemoryException`.

Given the location of the memory chunk, the row number, column number, row size, and column size the `getAt` operation returns the value stored at that location in the chunk. If the accessed element is outside the legal bounds of the array, the operation throws an exception of type `IndexOutOfBoundsException`.

Given the location of the memory chunk, the row number, column number, row size, column size, and a new value the `setAt` operation changes the stored value at that location in the chunk so it is the new value. If the accessed element is outside the legal bounds of the array, the operation throws an exception of type `IndexOutOfBoundsException`.

7.1.4. *[Untyped access]* Allocate a chunk of memory of size 16 containing consecutive natural numbers 1-16 using the `alloc` and `set` operations provided by the heap abstraction. Then treat this chunk of memory as a 4 x 4 array, and use the `setAt` operation to set diagonal elements of this array to the value 0.

7.1.5. *[Array Val and Operations]* Extend the Funclang language to add *array of numeric values* as a new kind of value to the programming language. This would require adding three new kinds of expressions `arrayexp`, `indexexp`, and `assignexp`. You will also need to generalize the array allocation and access operations for heap from the previous question.

To create an array with three rows, one can use the arrayexp as follows.

```
$ (array 3)
[ 0
  0
  0 ]
```

In the output we have adjusted spacing for clarity, but you are simply required to produce output that is equal to these.

To create an array with three rows and four columns, one can use the arrayexp as follows.

```
$ (array 3 4)
[[0  0 0  0]
 [0  0 0  0]
 [0  0 0  0]]
```

To create a three-dimensional array with three rows, four columns, and height two, one can use the arrayexp as follows.

```
$ (array 3 4 2)
[[[0  0 0  0]
   [0  0 0  0]
   [0  0 0  0]]
  [[0  0 0  0]
   [0  0 0  0]
   [0  0 0  0]]]
```

To access the second element in an array with three rows, one can use the indexexp as follows.

```
$ (index (array 3) 1)
0
```

To access the element in second row and first column in an array with three rows and four columns, one can use the indexexp as follows.

```
$ (index (array 3 4) 1 0)
0
```

To assign the second element in an array with three rows, one can use the assignexp as follows.

```
$ (assign (array 3) 1 342)
[ 0
  342
  0 ]
```

To assign the element in second row and first column in an array with three rows and four columns, one can use the indexexp as follows.

```
$ (assign (array 3 4) 1 0 342)
[[0  0 0  0]
 [342  0 0  0]
 [0  0 0  0]]
```

7.2 Memory related Operations in Reflang

In the rest of this chapter, we will develop *Reflang* a language with references. Reflang contains expressions for allocating a memory location, dereferencing a location reference, assigning a new value to an existing memory location and freeing a previously allocated memory location. For example, we can allocate a new piece of memory using the *reference expression* as follows.

```
$ (ref 1)
loc:0
```

A reference expression is like the *malloc* statement in C, C++. It will result in a memory cell being allocated at the next available memory location. That location will contain value `1`. The value of the reference expression is the *location* at which memory was allocated, here `loc:1210190219`.

The reference expression is also different from malloc statement in C and C++. Unlike malloc that accepts the size of the memory that is to be allocated, the argument of the reference expression is a value that is to be stored at the newly allocated location. From this concrete value, both the type of the value and the size required to store it can be derived.

In Reflang language we can explicitly free a previously allocated memory location using the *free expression* as follows.

```
$ (free (ref 1))
```

A free expression is like its namesake in languages like C, C++. It will result in a memory at the location that is the value of the expression `(ref 1)` to be deallocated.

We can also dereference a previously allocated memory location using the dereference expression

```
$ (deref (ref 1))
1
$ (let ((loc (ref 1))) (deref loc))
1
```

Dereferencing a memory location is a way to find out what is stored at that location. So a dereference expression takes a single expression, expression that evaluates to a memory location, and the value of the dereference expression is the value stored at the memory location.

We can also mutate the value stored at a memory location using the assignment expression

```
$ (let ((loc (ref 1))) (set! loc 2))
2
$ (let ((loc (ref 3))) (set! loc (deref loc)))
3
$
```

An assignment expression `set!` has two subexpressions, left hand side (LHS) expression that evaluates to a memory location, and right hand side (RHS) expression that evaluates to a value, and the value of RHS expressions is stored at the memory location that is the value of the LHS expression.

Reflang has an untyped heap, i.e. the type of the value stored a memory location in heap may change over time. We can mutate a memory location to store different kinds of values.

```
$ (let ((loc (ref 1))) (set! loc "2"))
2
$ (let ((loc (ref 3)) (loc2 (ref 4))) (set! loc loc2))
loc:1
$
```

In two examples above, initially at the location `loc` a number is stored. In the first example, the assignment expressions stores a string “1” at that location. In the second example, the assignment expression stores a memory location `loc2` at that location.

7.3 Parsing Reference-related Expressions

Main changes in the grammar for Reflang are highlighted in figure 7.1. This grammar builds on the grammar for the Funclang language.

The notation `...` in figure 7.1 is a shorthand, and it means that the definition of `exp` includes all alternatives defined for the Funclang language.

New expressions also follow the prefix form that we have been using so far. To store these new expressions, we also need to introduce four new abstract syntax tree nodes: `RefExp`, `DerefExp`, `AssignExp` and `FreeExp`. As is usual, adding new abstract syntax tree nodes requires extensions to other parts of the interpreter that must process each kind of expression, e.g. the `Visitor` interface, expression formatter, etc...

7.4 RefVal, a New Kind of Value

For the Funclang language, the set of normal values is given by:

```
Value :  NumVal | BoolVal | StringVal | PairVal | FunVal
```

We also have unit and null values.

```
Value :  ...  | NullVal | UnitVal
```

```
grammar Funclang;
program : ( definedecl )∗ (exp)? ;
exp :    ...
          | refexp
          | derefexp
          | assignexp  ;

refexp  :  '('  Ref exp  ')'  ;
derefexp  :  '('  Deref  exp  ')'  ;
assignexp  :  '('  Assign  exp exp  ')'  ;

Ref  :  'ref'  ;
Deref  :  'deref'  ;
Assign  :  'set!'  ;
```

Figure 7.1: Grammar for the Reflang Language. Non-terminals that are not defined in this grammar are exactly the same as that in Funclang.

Finally, we have a value that represents abnormal state of programs.

```
Value :  ...  | DynamicError
```

To support memory-related operations, we add a new kind of value `RefVal` to the Reflang language.

```
Value :  ...  | RefVal
```

The choice to create a separate kind of value, as opposed to using `NumVal` to represent references, has several consequences.

- `NumVal` cannot be mistaken for references in Reflang programs, which would prevent Reflang programs from accessing arbitrary locations in memory. On the other hand, standard locations e.g. the memory address of memory mapped devices, cannot be easily encoded.
- Operations on `NumVal` such as addition, subtraction, multiplication, etc... cannot be applied to references, which would also prevent Reflang programs from accessing arbitrary locations in memory. This

eases understanding of Reflang programs. On the other hand, optimized (in terms of size) representation of certain data structures e.g. messages sent on a network cannot be easily created.

- Extra meta-data about references may be encoded in the representation of `RefVal`. This has the advantage of encapsulating related information in a single abstraction. A separate table of meta-data indexed by the reference's numeric value can also be created, but that will require additional maintenance efforts.

7.5 Heap Abstraction

As discussed previously, `Heap` is an abstraction to represent dynamically allocated memory locations. In essence, a heap maps each reference value to a value.

```
Heap :  RefVal -> Value
```

It may be the case that the heap maps some reference values to the error value `DynamicError`.

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

Figure 7.2: The heap abstraction in the Reflang language.

7.6 Semantics of Reflang Expressions

Now since we have defined two essential concepts: heap and references, we can give semantics to Reflang programs.

Let `Program` be the set of all programs in Reflang, and `Exp` be the set of all expressions in Reflang. Also, let `p` be a program, i.e it is in set `Program` and `e` be an expression, i.e. it is in set `Exp` such that `e` is the inner expression of `p`. With these assumptions, in the presence of environments, we stated the semantics of a program as "In an environment `env`, the value of a program is the value of its component expression in the same environment `env`."

```
value p env = value e env
```

Here, `e` is an expression, `env` is an environment. In the presence of declarations, we further extended this rule, but for simplicity let us ignore declarations at the moment.

In the presence of heaps, the value relation is extended further.

```
value p env h = value e env h
```

We can state this relation as "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`."

Semantics of expressions that do not affect Heap

There are three kinds of expressions in Reflang, those that do not affect heap either directly or indirectly, those that only affect heap via their subexpressions, and those that directly affect heap.

A constant expression is an example of expression in any Reflang program that does not affect heap directly, and since it doesn't have any component subexpressions it cannot indirectly affect heap either. The value of a constant expression is a `NumVal` value. Let `e` be a constant expression that encapsules the numeric value `n`. Then,

```
value e env h = (NumVal n) h
```

Here, `n` is a `Number`, `env` is an environment, and `h` is a heap.

The meaning of a variable expression in a given environment and heap is simple. As in previous languages, it is the value obtained by looking up that variable name in the current environment. We can write the relation above as follows.

```
value (VarExp var) env h = get(env, var) h,
 where var ∈ Identifier, env ∈ Env, h ∈ Heap
```

The variable expression is another expression that does not affect.

Semantics of expressions that indirectly affect Heap

Most compound expressions can affect heap, if their subexpressions can affect heap. For these expression, most important consideration is the order in which side-effect of one expression is visible to the next expression. To illustrate consider the case for addition expression.

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

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

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

Since an addition expression has no effects on the environment, all of its subexpressions are evaluated in the same environment. However, each subexpression of the addition may affect the heap. Therefore, a left-to-right order is used in the relation above for side-effect visibility.

In defining such semantic relations for memory-related operations, the order in which side effects from one subexpression are visibile to the next subexpression has significant implications on the semantics of the defined programming language. For instance, consider an alternate semantics of the addition expression in which each subexpression $\texttt{exp}_0$ to $\texttt{exp}_n$ is evaluated using the heap `h`. Such a model would offer different tradeoffs.

Semantics of Heap-related expressions

Three expressions in Reflang directly affect heap. These are reference expression, assignment, and free expression. The dereference expression reads memory locations in the heap but doesn't change them.

$$\texttt{value (RefExp e) env h} = \texttt{l},\ h_2$$

$$\texttt{if value e env h} = v_0\ h_1$$

$$h_2 = h_1 \cup \{\ \texttt{l} \mapsto v_0\ \} \qquad \texttt{l} \notin \texttt{dom}(h_1)$$

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

The relation above says that in order to evaluate the value of a reference expression `(RefExp e)`, we must first find value of the subexpression `e`, and if that value is v_0 allocate a new location `l` in the heap and store the value v_0 at that location. The value of the reference expression is the reference value l, and the modified heap $h_2$contains a mapping from this new location to value. Also, notice that this new heap reflects all side-effects, i.e. memory read/write performed during the evaluation of the subexpression exp.

The value relation for the assignment expression is defined as:

$$\texttt{value (AssignExp } e_0\ e_1\texttt{) env h = } v_0\texttt{, } h_3$$

$$\texttt{if value } e_1 \texttt{ env h = } v_0\ h_1 \qquad \texttt{value } e_0 \texttt{ env } h_1 \texttt{ = l } h_2$$

$$h_3 = \{\ l \mapsto v_0\ \} \cup (h_2 \setminus \{\ l \mapsto \_\ \}) \qquad l \in \texttt{dom}(h_2)$$

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

Like reference expression, first subexpressions are evaluated. The order of the evaluation is right hand side (RHS) e_1 and then left hand side (LHS) e_0, i.e. side effects produced by e_1 will be visible to e_0. The notation $h_2 \setminus \{\ l \mapsto \_\ \}$ means subtracting mappings for l from the set h_2.

Several variations of the assignment expression can be conceived, e.g. the value relation above defines an assignment expression whose value is that of RHS. This semantics allows us to write statements like `x = y = z` in some programming languages, but it also causes one of the most common logical error `if (x = y) { ... }` when the programmer actually meant a comparison instead of assignment. A programming language can prevent such errors by taking the value of an assignment expression to be unit (or void as it is known in some languages).

The value relation for the free expression is defined as:

$$\texttt{value (FreeExp e) env h = unit, } h_2$$

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

$$h_2 = h_1 \setminus \{\ l \mapsto \_\ \}$$

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

The relation above insists that the location to be freed l is actually present in the heap h_1. This may cause dynamic errors in the program. Although freeing a memory location twice does reflect logical problems in the program, in some domains it may not be considered a critical flaw. So a variation of this semantics may omit this check.

The value relation for the dereference expression is defined as:

$$\texttt{value (DerefExp e) env h = v, h}_1$$

$$\texttt{if value e env h = l h}_1 \quad \texttt{l} \in \texttt{dom(h}_1\texttt{)}$$

$$\{\ \texttt{l} \mapsto \texttt{v}\ \} \subseteq \texttt{h}_1$$

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

7.7 Realizing heap

Given the semantic relations that define Reflang, we can now begin to implement the programming language. To that end, figure 7.4 shows a realization of the heap abstraction. This implementation uses an array as a backend storage to implement the heap. So references (`RefVal`) encapsulate the index of the location in the backend storage.

```
class RefVal implements Value { //New in the reflang
    private int _loc = -1;
    RefVal(int loc) { _loc = loc; }
    String tostring() {
        return "loc:" + this._loc;
    }
    int loc() { return _loc; }
}
```

Figure 7.3: An implementation of the RefVal abstraction

The implementation of heap in figure 7.4 makes no attempt to compact the memory locations that are freed. The implementation also does not attempt to recycle memory locations that have recently been deallocated. These enhancements are subject of some of the exercises in this chapter.

7.8 Evaluator with references

The implementation of new expressions related to references in the evaluator closely models the semantic relations discussed previously.

As figure 7.5 shows, every program is evaluated in a fresh heap.

The methods in figure 7.6 implement semantic relations for reference, dereference, assignment, and free expressions. Notice that unlike mathematical relations that are defined using heaps that are immutable sets, in the actual implementation the global heap of the evaluator is modified in each of the reference, assignment and free expressions. This realization of these reference-related expressions heavily relies upon the semantics provided by the operations of the heap abstraction.

Exercise

7.8.1. *[Versioned heap]* Modify the semantics of Reflang to implement a heap abstraction that implements versions of a heap. In a versioned heap, writing to an existing memory location does not overwrite the old value, rather it creates a new version of that memory location that contains the new value.

```
class Heap16Bit implements Heap {
  static final int HEAP_SIZE = 65_536;

  Value[] _rep = new Value[HEAP_SIZE];
  int index = 0;

  Value ref (Value value) {
    if(index >= HEAP_SIZE)
      return new Value.DynamicError("Out of memory error");
    Value.RefVal new_loc = new Value.RefVal(index);
    _rep[index++] = value;
    return new_loc;
  }
  Value deref (RefVal loc) {
    try {
      return _rep[loc.loc()];
    } catch (ArrayIndexOutOfBoundsException e) {
      return new DynamicError("Segmentation fault at access " + loc);
    }
  }
  Value setref (RefVal loc, Value value) {
    try {
      return _rep[loc.loc()] = value;
    } catch (ArrayIndexOutOfBoundsException e) {
      return new DynamicError("Segmentation fault at access " + loc);
    }
  }
  Value free (RefVal loc) {
    try {
      _rep[loc.loc()] = null; //REMARK: Add this location to free list.
      return loc;
    } catch (ArrayIndexOutOfBoundsException e) {
      return new DynamicError("Segmentation fault at access " + loc);
    }
  }

  Heap16Bit(){}
}
```

Figure 7.4: An implementation of the heap abstraction in Reflang

```
class Evaluator implements Visitor<Value> {
    Heap heap = null; //New for reflang

    Value valueOf(Program p) {
      heap = new Heap16Bit();
    return (Value) p.accept(this, initEnv);
  }
     ...
}
```

Figure 7.5: Evaluator with reference expressions

```
class Evaluator implements Visitor<Value> {
    ...
    Value visit (RefExp e, Env env) {
        Exp value_exp = e.value_exp();
        Value value = (Value) value_exp.accept(this, env);
        return heap.ref(value);
    }

    Value visit (DerefExp e, Env env) {
        Exp loc_exp = e.loc_exp();
        Value.RefVal loc = (Value.RefVal) loc_exp.accept(this, env);
        return heap.deref(loc);
    }

    Value visit (AssignExp e, Env env) {
        Exp rhs = e.rhs_exp();
        Exp lhs = e.lhs_exp();
        //Note the order of evaluation below.
        Value rhs_val = (Value) rhs.accept(this, env);
        Value.RefVal loc = (Value.RefVal) lhs.accept(this, env);
        Value assign_val = heap.setref(loc, rhs_val);
        return assign_val;
    }

    Value visit (FreeExp e, Env env) {
        Exp value_exp = e.value_exp();
        Value.RefVal loc = (Value.RefVal) value_exp.accept(this, env);
        heap.free(loc);
        return new Value.UnitVal();
    }

}
```

Figure 7.6: Evaluator with reference expressions