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]Pure 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:0`.

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
```

`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***

Figure 7.1: Grammar for the Reflang Language. Expressions in bold are new to the 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.

$$\texttt{value (VarExp var) env h = get(env, var) h,}$$

$$\texttt{where var} \in \texttt{Identifier, env} \in \texttt{Env, h} \in \texttt{Heap}$$

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.

$$\texttt{value (AddExp e}_0 \texttt{ ... e}_n\texttt{) env h = v}_0 \texttt{ + ... + v}_n\texttt{, h}_n$$

$$\texttt{if value e}_0 \texttt{ env h = v}_0 \texttt{ h}_0\texttt{, ..., value e}_n \texttt{ env h}_{n-1} \texttt{ = v}_n \texttt{ h}_n$$

$$\texttt{where e}_0\texttt{, ..., e}_n \in \texttt{Exp, env} \in \texttt{Env, h, h}_0\texttt{,...h}_n \in \texttt{Heap}$$

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.

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

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

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

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

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 $\mathtt{v_0}$ allocate a new location `l` in the heap and store the value $\mathtt{v_0}$ at that location. The value of the reference expression is the reference value `l`, and the modified heap $\mathtt{h_2}$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:

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

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

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

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

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

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:

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

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

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

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

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:

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

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

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

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

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. The listing in figure 7.3 shows the concrete implementation of `RefVal`. The realization intentionally makes objects of this class immutable.

```
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

Several enhancements to the implementation of `RefVal` are possible. For example, this class can maintain information about whether the encapsulated location has been accessed, and with what frequency. This class can also be extended to be able to refer to a larger heap. Currently the maximum heap size that can be accessed is determined by the maximum value of `int`. The class can also be extended to maintain information about the type of value found by dereferencing this reference value.

The realization of heap in figure 7.4 provides implementation of each of the four methods `ref`, `deref`, `setref`, and `free`.

The `ref` method is like `malloc` in languages like C and C++ in that it allocates a new memory location. It is, however, different because it requires a value that is to be stored in that memory location. Traditional `malloc` requires the size of memory that is to be allocated. The allocation procedure is relatively simple. The method takes the next available location in the internal representation `_rep` and adjusts `index`. If the heap is full, a `DynamicError` is returned signaling that we have run out of memory.

The `deref` method is similar to their counterpart in other languages. It returns the value stored at a given reference. This method raises another kind of `DynamicError` signaling that an attempt to access a memory location outside the legal heap has been made. In some language implementations, for the sake of efficiency, such checks are not performed. In the absence of such checks, malicious programs may read arbitrary memory locations which may be storing data from other programs. Fortunately, modern operating systems provide checks against malicious usage. However, such checks do not guard multiple threads of execution within a single process from each other.

The `setref` method changes the value at a given reference. This method also raises another kind of `DynamicError` signaling that an attempt to access a memory location outside the legal heap has been made.

The `free` method removes the value stored at a given reference. This method also raises another kind of `DynamicError` signaling that an attempt to access a memory location outside the legal heap has been made. The current implementation of this method doesn't consider freeing an already free or unallocated location an error. The reader may attempt to enhance the semantics of their interpreter to declare freeing an already free or unallocated location an error. Also, notice that the free method sets the memory location being freed to `null` thereby removing the older value from that memory location. In some language implementations, in favor of efficiency, memory location is not erased by the implementation of free expressions. In such languages malicious programs may allocate large chunks of memory simply to read data leftover by previous programs.

```java
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;
      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

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. Those 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.

```
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

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.

Evaluation of a reference expression `RefExp` proceeds by evaluating its subexpression, and then make use of the heap's helper function `ref` to store that value in a memory location. The value of this expression is the reference value. Notice that this expression does not attempt to stop error values from being stored in the heap. The reader may choose to enhance their interpreter to implement an alternative semantics where dynamic errors are not stored in the heap.

```
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

Evaluation of a dereference expression `DerefExp` also proceeds by evaluating the subexpression and directly builds on the helper method `deref` provided by the heap abstraction.

Evaluation of an assignment expression `AssignExp` proceeds by evaluating the right-hand-side (RHS) followed by the left-hand-side (LHS) expression. This order of evaluation ensures that the changes in the heap made by the RHS expression are visible to the LHS expression. The implemented semantics says that the value of an assignment expression is the value of the RHS. An alternative semantics is to take the value of an assignment expression to be a unit value (like the free expression). The reader is encouraged to think about the tradeoff between these two semantic choices especially from the point-of-view of chaining multiple assignments.

Evaluation of a free expression `FreeExp` evaluates its subexpression to a reference value and makes use of the helper method `free` to remove the stored value at that location in the heap.

Exercise

7.8.1. *[Memory-mapped I/O]* Modify the Reflang language so that the locations `0`, `1`, `2` are treated specially as standard input, standard output, and standard error. Take the following steps to implement this functionality.

1. Modify the heap abstraction such that it allocates memory starting with location `3`.
2. Modify the initial environment such that the names `stdin`, `stdout`, `stderr` are mapped to reference values `loc:0`, `loc:1`, and `loc:2` respectively.
3. Modify the semantics of free expression such that attempts to free locations `1`,`2` and `3` result in a dynamic error with messages, `"Illegal attempt to deallocate stdin"`, `"Illegal attempt to deallocate stdout"`, `"Illegal attempt to deallocated stderr"`.
4. Modify the semantics of dereference expression such that attempts to dereference locations `1` and `2` result in a dynamic error with messages `"Illegal read from stdout"` and `"Illegal read from stderr"` respectively. Further, modify the semantics

of dereference expression such that attempts to dereference location `0` results in a call to internal read from standard input in your defining language and returns the line read from the standard input as a string value. If reading from input causes an input exception the result should be a dynamic error with message `"Read from stdout failed"`.

5. Modify the semantics of assign expression such that attempts to assign to location `0` results in a dynamic error with message `"Illegal write to stdin"`. Further, modify the semantics of assign expression such that attempts to assign to locations `0` and `1` results in printing the value being assigned (RHS) on `System.out` and `System.err` respectively. The value of assign expression in those cases should be a unit value.

The following interaction log illustrates the properties of the resulting language.

```
$ (ref 342)
loc:3
$ stdin
loc:0
$ stdout
loc:1
$ stderr
loc:2
$ (free stdin)
 Illegal attempt to deallocate stdin (free stdin)
$ (free stdout)
 Illegal attempt to deallocate stdout (free stdout)
$ (free stderr)
 Illegal attempt to deallocate stderr (free stderr)
$ (deref stdout)
 Illegal read from stdout (deref stdout)
$ (deref stderr)
 Illegal read from stderr (deref stderr)
$ (deref stdin)
Memory mapped I/O is cool!
Memory mapped I/O is cool!
```

```
$ (set! stdout "Hello World!")
Hello World!
$ (set! stderr "Hello World!")
$ Hello World!
```

7.8.2. *[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.

7.8.3. *[Alias notification]* This problem is about references and aliasing. In Reflang an expression like the following creates two aliases (`class` and `course`) to the memory cell storing `342`.

```
(let ((class (ref 342))) (let ((course class)) (deref course)))
```

Modify the Reflang interpreter so that it prints a message when an alias is created. An example appears below.

```
$ (let ((class (ref 342))) (let ((course class)) (deref course)))
Alias created: name class ref value loc:0.
Alias created: name course ref value loc:0.
342
```

7.8.4. *[Typed heap]* This question is about a semantic variation of the heap abstraction.

Typed heap enforces the property that in a memory location, only values of compatible type can be stored. Two types are compatible if one is the subtype of the other. Extend the Reflang interpreter to support typed heap.

Modify, semantics of assign expression assignexp to check that upon setting the value of a location the type of the new value is compatible with the type of the old value already stored in that location. Otherwise, raise a dynamic error.

Hint: in Java you can use isAssignableFrom to check for compatibility of types.

The following log interaction illustrates the semantics of typed heap.

```
$(let ((x (ref 0))) (set! x 12))
12

$(let ((x (ref 0))) (set! x #t))
Assigning a value of an incompatible type to the location in (set
    ! x #t)

$(let ((x (ref (ref 0)))) (set! x (ref(ref(ref 5)))))
loc:4
```

7.8.5. *[Hot locations]* This question is about a semantic variation of the reference related expressions.

A hot heap location is one that is accessed more often by a program. Extend Reflang to support hot store locations.

- First step in keeping track of hot store locations is to augment the store locations to have an addition integer field `access-count`.
- Second step is to enhance the logic of accessing a store location to increment the `access-count` on both reads and writes of memory locations.

To achieve this semantics, modify the Reflang interpreter as follows:

1. Modify the heap so that each location is a pair of the value and a number (from now on we call this number `access_count`). You can also use a separate list to keep track of `access_count` for locations. Initially, the `access_count` for each location is `0`.
2. Modify the interpreter such that, for every read of a location and assignment to the location, the field `access_count` of the location is incremented by `1`.
3. Add a `frequency` expression to Forklang with the following syntax that takes a location and evaluates to the `access_count` of that location.

   ```
   freqexp: '(' 'frequency' exp ')' ;
   ```

7.8.6. *[Strict free]* This problem is about memory deallocation. In Reflang the `free` expression deallocates memory. Current semantics of `free` expression is permissive in that it allows a memory location to be deallocated even if it has been deallocated previously.

Change the semantics of the `free` expression such that the attempts to deallocate a value that is already allocated results in a dynamic error with a message `"Illegal deallocation of ref value loc:0"`. For example,

```
$ (let ((c (ref 342))) (let ((d (free c))) (free c)))
 Illegal  deallocation  of  ref  value  loc:0
```

7.8.7. *[Reference arithmetic]* This problem is about explicit references. In current realization of the Reflang language arithmetic operations are not permitted on a reference value.

1. Modify the semantics of dereference expression such that it can dereference locations specified as explicit natural numbers. See the interaction log below for an example.

   ```
   $ (let ((class (ref 342))) (deref 0))
   342
   ```

2. Modify the semantics of assignment expression such that it can assign locations specified as explicit natural numbers. See the interaction log below for an example.

   ```
   $ (let ((class (ref 342))) (set! 0 541))
   541
   ```

3. Modify the semantics of addition and subtraction expressions such that the addition and subtraction are permitted on reference values.

   In this resulting language, adding one or more numeric values to a reference value will result in a reference value. See the interaction log below for an example.

   ```
   $ (+ 1 (ref 342) )
   loc:1
   $ (+ 1 (ref 342) 1)
   loc:2
   ```

In this language, subtracting one or more numeric values from a reference value will result in a reference value. See the interaction log below for an example.

```
$ (− (+ 1 (ref 342)) 1)
loc:0
```

7.8.8. *[Uses reference arithmetic]* This problem is about explicit references. It builds on the previous problem.

1. In the previous problem, we have enhanced the Reflang language to allow reference arithmetic. Add a new predicate expression, `rarith?` to check if an expression uses reference arithmetic during its evaluation.

   The expression `rarith?` follows the grammar below.

   ```
   rarithexp : '(' rarith ? exp ')';
   ```

   The following interaction log illustrates the semantics of this expression

   ```
   $ ( rarith ? (+ 1 2))
   #f
   $ ( rarith ? (+ 1 (ref 342)))
   #t

   $ (define raddn (lambda (n r) (+ n r)))
   $ ( rarith ? (raddn 1 ( ref 342)))
   #t
   ```

2. Write an example Reflang program that has reference arithmetic as its subexpression, but that subexpression is not executed during its evaluation. Your program must use at least one lambda expression.

7.8.9. *[Reachability]* This problem is about reachability, which is an important concept for automatic memory management. Given a name, the set of reachable memory locations are those that can be accessed directly or indirectly using that name. For instance, for the expression below reachable memory locations for name `c` are: `loc:0`

```
$ (let ((c (ref 342))) (reachable c))
{ loc: 0 }
```

Similarly, for the expresion below reachable memory locations for name `c` are:

```
$ (let ((c (ref (list (ref 342) (ref 541))))) (reachable c))
{ loc: 0, loc: 1, loc: 2 }
```

Here, value `342` is stored at `loc:0`, value `541` is stored at `loc:1` and the list value is stored at `loc:2`.

Add an expression `reachable` that follows the grammar below.

```
reachableexp: '(' reachable Identifier ')';
```

The value of the reachable expression is a string value that starts with an open brace '`{`' and ends with a close brace '`}`'. The string value contains a comma-separated list of reachable locations. Some examples appeared previously.