IB Semantics - Supervision 2

Nandor Licker <nl364@cl.cam.ac.uk>

Due noon before two days before supervision

We are going to extend L1 with objects and garbage collection, changing the syntax and semantics of the language. We will not adjust typing rules since the required rules exceed the scope of this course. Our heap will be organised similarly to the OCaml heap: it will contain blocks of a fixed length, where each word in a block is either an address pointing to another block or a primitive integer. Note that OCaml has 63 bit integers, since the last bit of all words is used to distinguish primitives from addresses. As an example, consider the following statement:

```
type 't list = Nil | Cons of 't * 't list
let lst = Cons(1, Cons(2, Nil))
```

The statement would allocate two blocks on the heap, as the `Nil` constructor has no arguments and can be represented using the primitive 0. The objects would be connected as shown in Figure 1.

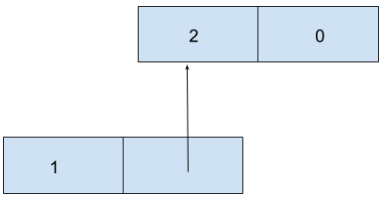

Figure 1: `Cons(1, Cons(2, Nil))`

Garbage-collected languages usually have a set of root pointers - whenever the heap is full and collection occurs, only the objects referenced directly or indirectly from these root pointers are kept on the heap, all the others are deleted, freeing memory. We will be using the original store as the set of root pointers, adding an additional parameter H to the state and all reduction rules to keep track of the state of the heap, which is simply a mapping from addresses to blocks on the heap.

1. The new values can be either primitives or addresses - adjust the set of L1 values and operations to reflect this, define it formally. Use the notation $\bar{i}$ for integers and $\tilde{l}$ for addresses.
2. Introduce syntax to allocate a new block on the heap of a fixed size, returning its address. Define its reduction rule $\langle \texttt{new}\ n, s, H\rangle \to ....$ Carefully choose a representation for heap-allocated blocks, remembering that sets do not support indexing. Assume that the size of the heap is unlimited.
3. Introduce syntax to read the field of a block. Define its reduction rule $\langle \tilde{l}[\bar{i}], s, H\rangle \to ...$
4. Introduce syntax to write to a field of a block. Define its reduction rule $\langle \tilde{l}[\bar{i}] := e, s, H\rangle \to ...$
5. Extend the proof of determinism to include the new rules.
6. The size of the heap is now bounded: $|H| < s$ is a constraint that must always hold, for some $s \in \mathbb{N}$. Adjust the rule for allocation to collect all unused objects in order to free some memory before allocating a new object. Carefully consider the case when there is no more memory available.

7. Compact the heap: whenever garbage is collected, live objects are relocated such that the addresses are contiguous. Adjust reduction rules and redo the determinism proof for the allocation statement.

8. Provide a type system and typing judgements, assuming the language only supports lists. How would you extend the type system to support more data types? Can you define a type system that accepts all programs which run to completion?

9. Suppose our language relies on reference counting. Update the definition of the heap to include reference counts and modify the relevant rules to adjust the counts, collecting objects whenever possible. Write a program in the language that creates an object which can never be collected, but has no references to it.

10. Collection pauses are a major concern for garbage-collected languages. Sometimes it is beneficial to perform collection consistently throughout the execution of a program, not only when an allocation encounters a full heap[1]. To model this, we define the following rule:

$$\langle e, s, H\rangle \to \langle e, s', H'\rangle, \text{ where } (s', H') = gc(e, s, H) \text{ and } |H'| + 1 = |H|$$

Describe the meaning of the reduction rule. Why does the gc function include e as a parameter? Define an allocation rule which does not trigger a collection. Why is this version of the language not deterministic anymore? Define a notion of equivalence between states and redefine determinism to make use of it. Prove determinism for *(op +)* and field writes.

Hint: It might be useful to define an equivalence relation between pairs of heaps and live objects, constructing isomorphisms.

[1] If you are interested, you can read up on IBM's Metronome