Representing Non-numerical Data on a Computer

Max Hailperin

MC28, December 4–6, 1995

At the beginning of the semester we looked at the structure and capabilities of the SLIM computer, which was intended to be broadly similar to real computers. This should have helped convince you that computation doesn't require magic—it can be carried out by machinery. This handout is intended to strip away one remaining magical illusion and show how it too can be carried out by a machine such as SLIM. Namely, when we looked at SLIM before, we saw that it could only store and operate on numbers, yet the computations we usually work with (in Scheme, for example) involve not only numbers but also pairs, procedures, symbols, strings, vectors, the empty list, and booleans. In this handout we'll look at how such data types can be represented on a computer like SLIM. Rather than try to cover all the datatypes, we'll focus on pairs as an illustration of the general concepts that apply to all the types.

Since all SLIM can store or operate on is numbers, we'll have to encode all our data as numbers. There are lots of variants of how this can be done; we'll present one typical example. Each object, no matter of what type, will be represented by a number that is a memory address where information about the object is stored. So, we might represent a pair as the number 213, meaning that if you go to location 213 in memory (and some number of immediately following locations, to be specified below), you will find out everything you need to know about the object: that it is a pair, what its car is, and what its cdr is. We can even represent numbers this way; for example, we might represent the number 17 as the number 270, meaning that if you go to location 270 in memory (and some number of subsequent location), you will find out the information that this object is a number, and that its value is 17.

One of the things we want to be able to do with an object is determine its type; for example, we want to be able to use `pair?` to determine whether or not it is a pair, `number?` to determine whether or not it is a number, etc.

Therefore, we will establish a convention that the first memory word used to store the information about any object will contain an indication of what kind of object it is. We call this representational convention a *tagged-object representation* and call the type indicators stored in the first words *tags*. We can assign numbers to use as the tags in any way we like, so long as each type has its own tag. For example, we could tag all pairs with 0, numbers with 1, procedures with 2, symbols with 3, etc. Thus, in the above example, memory location 213 will contain a 0 and memory location 270 will contain a 1.

Each type of object will need some number of subsequent locations to store additional information. In the case of a pair, we can use two more memory locations, the first to store the representation of the pair's car and the second to store the representation of the pair's cdr. These two would be represented in the same way as everything else, as memory addresses. Continuing our above example, if the pair at location 213 has the number 17 as its car, then location 214 of memory would contain the number 270.

All the remains is to see how each operation on pairs would be performed. We will consider one by one the five operations relevant to pairs: `pair?`, `car`, `cdr`, `set-car!`, `set-cdr!`, and the constructor, `cons`.

Suppose that some SLIM register, perhaps the one named `obj` contains the representation of some object, and we would like to test whether it is a pair, placing a 1 into some other register if so and otherwise a 0. Let's say for example we wanted the result to go into a register called `dest`. Suppose further that we have a register called `zero` that contains the numerical value 0, which we are using as the tag for pairs. Given the above representation, all we need to do is

```
(ld dest obj)
(seq dest dest zero)
```

This loads in the tag word of the object into the `dest` register, then tests whether it is the right tag to be a pair.

The selectors `car` and `cdr` aren't any harder, though we have to add 1 or 2 to the memory address before we load. For simplicity, assume that the registers named `one` and `two` have already had 1 and 2 loaded into them, so that we don't need to stop to do an `li`. In this case we can do the `car` operation as

```
(add dest obj one)
(ld dest dest)
```

and the `cdr` operation as

```
(add dest obj two)
(ld dest dest)
```

In each of these cases, we first used the `dest` register temporarily to hold the address to load from, and then replaced that address with the loaded value, which is the representation of the car or cdr of the pair.

We can do the `set-car!` and `set-cdr!` operations similarly, using SLIM's `st` instruction to store into memory instead of loading from memory. Let's assume that the register named `new` contains the representation of the object we want to become `obj`'s new car or new cdr. We'll also need a spare register for temporary use; we'll call this `temp`. Given these assumptions, we can do `set-car!` as follow:

```
(add temp obj one)
(st new temp)
```

The `set-cdr!` operation would be identical except `two` would be used in place of `one` in the `add` operation.

At this point the only operation we have left is the constructor, `cons`. This will involve allocating a new block of three consecutive memory words that are not otherwise in use, storing the pair tag into the first of those words, the representation of the car into the second word, and the representation of the cdr into the third word. The address of the first word in the block will be the result of the `cons` operation, since it is the representation of the resulting pair.

Suppose for concreteness that we have a big chunk of the SLIM's memory that is available for us to allocate from. We will use two registers, `first-free` and `last-free`, to hold the addresses of the first and last words still available for us to use. We will allocate space from the beginning of this region towards the end, so we will increase `first-free` every time to indicate that the allocated space is no longer available. We can use `last-free` to check that we still have space remaining. Initially we'll assume that if we're out of space, we just give up on running the program, by executing the `halt` instruction. Later we'll look at a more realistic reaction to running out of space.

If we want to do a `cons` operation, we will have two registers that contain the representations of the car and the cdr; let's call these `the-car` and `the-cdr`. We'll also have a third register into which we want to store the representation of the resulting pair; let's call this `the-pair`. We'll also need a register called `allocate` to hold an instruction address we conditionally jump to if there is space available to allocate the new pair. Given these assumptions, and those from above, we can do the `cons` operation as follows:

```
(add temp first-free two)  ; this will be the last memory word used
(sgt temp temp last-free)  ; check if it runs off the end of our space
(li allocate label)
(jeqz temp allocate)       ; if space remains, jump to the label

(halt)                     ; otherwise give up (this will be improved)

label
(add the-pair first-free zero) ; copy this address as the result
(st zero first-free)       ; store the pair tag (0) into first word
(add first-free first-free one) ; and bump pointer
(st the-car first-free)    ; store the car into the next word
(add first-free first-free one) ; and bump pointer
(st the-cdr first-free)    ; store the cdr into the next word
(add first-free first-free one) ; and bump pointer
```

At this point, we've got a working implementation of the pair operations using our chosen representation. (At least, aside from the detail of a few `li` operations at the beginning of the program to store the appropriate constants into various registers.) However, it will only work so long as all the objects (such as pairs) allocated in the entire lifetime of the program can fit in the amount of memory available between the starting value of `first-free` and `last-free`. If too much allocation is attempted for the amount of memory available, the program will suddenly halt.

One possible improvement would be to print out some sort of explanatory message before halting. However, there is another, more fundamental, kind of improvement. Some pairs (or other objects) may get allocated, used for a while, and then fall out of use. The memory space they occupy should become available for re-use for other objects. That way if a program only uses a limited number of pairs at any one time, but keeps `cons`ing and then discarding pairs, it should be able to run forever without running out of memory. In other words, we should have a way of *deallocating* memory so that it can re re-allocated for another object. There are two ways this can be done. One is to have the programmer explicitly say "I am done with this pair now, you can re-use the memory." Many programming systems operate this way, with *manual deallocation*, but it is extremely problematic, since programmers have a difficult time accurately keeping track of which objects are still in use and which aren't. A better solution, used in Scheme, is to use *automatic deallocation*, also known as *garbage collection*. A garbage collector automatically discovers objects that cannot possibly still be in use and frees up their storage space for re-use.

To change our `cons` implementation to interface with a garbage collector, we'll just replace the `halt` instruction with the following two instructions:

```
(li temp the-garbage-collector-label)
(j temp)
```

This will jump to the start of the garbage collector; the `allocate` register serves as a continuation register for the garbage collector, in the sense that once the garbage collector has succeeded in freeing up some space, it should jump to instruction whose address is contained in that register. That way, the program will continue running with the allocation of the new pair. Since the garbage collector will have changed the `first-free` and `last-free` registers, the operation can now continue without running off the end of the available memory. (Presumably the garbage collector will have halted if it failed to find garbage to collect, which would indicate that we are really out of space.)

The job of the garbage collector is to determine which objects are still potentially in use and which aren't. It starts with the representations stored in registers, such as the `the-car` and `the-cdr` registers. Those objects definitely might get used. Moreover, depending on the objects' types, they might contain the representation of other objects. For example, a pair would contain representations of its car and cdr. Those objects might get used too. This continues, discovering layer by layer all the objects that can be reached starting from the registers. Since there are only a finite number of objects, eventually no new objects will be discovered. Any objects that have not been encountered definitely are no longer in use, since there is no way to get at them. These are the garbage objects. (As a detail, if any registers that contained object representations were saved to the stack, then we may have to use the stack locations as additional starting points for the discovery process.)

There are a variety of ways this process of discovery can be carried out, as well a variety of ways in which the memory used to store garbage objects can be made available for re-use. Here we'll consider one popular option, known as *stop-and-copy* garbage collection. In a stop-and-copy system, the total portion of the computer's memory that is available for storing object representations is divided into two equal sized *semispaces*. At any one time, only one semispace is in use. So, for example, we might initially store the starting address of one semispace into the register called `first-free` and the ending address of that same semispace into `last-free`. That way we will only allocate memory for objects from that one semispace.

When the semispace fills up, the garbage collector is invoked. At that point what it does is discover the reachable (non-garbage) objects and copy them into the other semispace as it is discovering them. The new copies are allocated starting from the beginning of the new semispace and working consecutively through it. Thus, all the new copies will be at the beginning of the semispace. If there was any garbage in the old semispace, it won't have been copied. Thus, the new semispace won't be entirely full, even though the old one was. At this point the `first-free` register can be updated to contain the address of the first unused location in the new semispace and the `last-free` register can be updated to contain the end of that semispace. As the discovery and copying was being done, any registers that contained object representations were updated to contain the addresses of the new copies rather than those of the old ones. Thus at this point the old semispace has fallen totally out of use and the computation can resume in the new semispace. When that new semispace fills, the process is repeated, this time copying back into the original semispace. By alternating between the two semispaces, any number of garbage collections can be done.

The process of discovering and copying the objects that are reachable is nearly identical to what you did in the `print-list-structure` lab. In that lab, you used two data structures: a table and a queue. The table was used to keep track of which pairs had already been assigned identifying numbers, and what numbers they were assigned. Analogously, the garbage collector will need to keep track of which objects have already been copied, and what address in the new semispace they were copied to. That way, if another reference to the same object is later discovered, it can simply be updated to point to the new location without any additional copying. The queue was used to keep track of pairs that had been discovered but had not yet been printed out. Analogously, the garbage collector will need to keep track of objects that have already been discovered and copied, but have not yet had their components examined and updated. (For example, pairs that still have cars and cdrs back in the old semispace.)

The only really clever idea in stop-and-copy collectors is this: rather than using more memory to hold the table and the queue, the semispaces can do double duty as these structures. The old semispace is used as the table, and the new semispace is used as the queue. We'll examine how this is done one at a time, starting is the use of the old semispace as the table.

Once an object has been copied from the old semispace to the new, we can clobber the memory it was occupying. So, we'll overwrite the first word, which had previously contained the type tag, with a new special type tag that indicates "this object has already been copied." That way, given an address of an object in the old semispace, we can directly determine whether it has been copied yet or not just by looking at the tag word. (This special tag is often called "broken heart," because it is what is left behind when an object moves.) The other information we need is what the "forwarding address" is, i.e., what the address is of the new copy in the other semispace. This can be stored into the second word of the old copy, overwriting for example the car, if this was a pair. (We'll assume that all objects have at least one word beyond their tag word.) Thus, if a broken heart tag is found, the next word can be consulted to find the object's new address. Thus, the role of the "table" has been filled without needing any additional memory.

Moving on to the use of the new semispace as the queue, all we have to do is use a couple registers to divide the semispace up into three consecutive sections: those objects that have been copied and updated, those that have been copied but not yet updated, and the remaining free space. The middle region is the queue. When a new object is copied, it is copied into the beginning of the free space, and the register containing the first free address is updated to reflect this, so that the object is now at the end of the "queue region." This corresponds to putting the object onto the queue. To remove an object from the queue, the register pointing at the first object that was copied but not yet updated is simply "pushed past" that object to point at the next. When this register catches up with the register marking the start of the free space, the queue has become empty and the garbage collection is complete.

When a pair is removed from the queue and updated, we examine its car and cdr components. These will be addresses in the old semispace. We look at the tags at those addresses. If they are broken hearts, then the objects that are the car and cdr have already been copied, and all we have to do is replace the old addresses with the new ones. Thus, the queue will have shrunk. On the other hand, if looking into the old semispace at the addresses of the car and cdr we find normal tags, then these two objects still need copying; we do so now. Since copying involves adding to the queue, removing one pair from the queue may have added two others, resulting in queue growth. Eventually, however, the queue will empty out, since each object is only copied the first time it is discovered.