Heap Allocation of Data with Dynamic Extent

sk and dbtucker

November 15, 2000

We are showing how to build a compiler by presenting the output of each phase by hand. So far we used CPS to make the stack explicit, then got rid of the stack argument. We also showed how to put the program into register form.

In the previous lecture, we removed those instances of *cons* that were used for the stack representation. We still have *cons*es that build the lists consumed and produced by the *filter* function. Why don't we just put these *cons*es on the stack as well?

The problem is that different data may have different *lifetimes* (or *extents*). The lifetime of the *cons* used to represent the stack is known statically; it lives only as long as the current stack frame. In constrast, the lifetime of the *cons* used to represent the lists can only be determined dynamically; it can live indefinitely. Also, the *cons* for the stack can only be referred to by a single variable, whereas the *cons* for the list may have multiple references.

The heap

We will use a heap to store values that have indefinite lifetimes. As we did with the stack, we represent the heap as a vector:

```
(define Heap-Page-Size 100)
(define Heap (make-vector Heap-Page-Size))
(define Heap-Ptr 0)
```

The function *alloc-one-heap-cell* allocates a cell on the heap, stores a value in it, and returns the address of the cell:

```
(define (alloc-one-heap-cell v)
  (ensure-heap-space 1)
  (let ([addr Heap-Ptr])
    (vector-set! Heap Heap-Ptr v)
    (set! Heap-Ptr (+ Heap-Ptr 1))
    addr))
```

Now look at the code for *filter* on page 1 of today's handout. Notice that we have made two changes to the code:

1. We renamed several functions to end with */prim* (e.g., *cons/prim*, *even?/prim*). These functions operate on the heap.

2. We renamed several variables to end with *-val*. The values of these variables are heap addresses.

The place to start is *cons/prim*, which allocates two cells on the heap:

```
(define (cons/prim fst-val rst-val)
  (let* ([fst (alloc-one-heap-cell fst-val)]
         [rst (alloc-one-heap-cell rst-val)])
    fst))
```

The selectors *first/prim* and *rest/prim* take a heap address and return the appropriate component:

```
(define (first/prim list-val)
  (vector-ref Heap list-val))

(define (rest/prim list-val)
  (vector-ref Heap (+ list-val 1)))
```

The function *filter-test* allocates constants on the heap, creates a list using *cons/prim*, and calls *filter/split*. The important point to note is that the constant *one* is the address of a cell containing the number 1. So, each *cons/prim* pair comprises the address of a constant's cell and the address of the next pair. Now we can interpret the final answer:

- The final answer location is 25.
- Cell 25 contains 4, which is the address of the constant 2. Cell 26 contains 23, which is the address of the next *cons/prim* pair.
- Cell 23 contains 6, which is the address of the constant 4. Cell 24 contains 21, which is the address of the next *cons/prim* pair.
- Cell 21 contains 8, which is the address of the constant 6. Cell 22 contains 2, which is the address of the constant 'Empty-list.

The result list is thus '(2 4 6), as expected.

The extra level of indirection in the above example leads to wasted heap space. We would like to represent an immediate value, such as the constant 4, as an immediate datum on the heap rather than a heap reference. On page 2 of the handout, we make a distinction between two types of cells in the heap: Immediates and References. An Immediate cell contains a value, whereas a Reference cell contains the address of another cell. Look at the resulting heap: the final answer location 16 contains '(Immediate 2), which represents the value 2. Location 17 contains '(Reference 14), which represents address 14, the location of the next pair.

On page 3, we simply change the representation of immediate and reference cells. Heap values less than 1000 refer to constants, which are stored in a

separate table (in this case, we print out the table). Heap values greater than or equal to 1000 are references; the number minus 1000 is the address. The final answer location in the example is 16; cell 16 contains 4, which represents the constant 2, and cell 17 contains 1014, which represents the heap address 14.

There is still one issue we haven't considered—the tagging of values. In order to ensure type safety, primitive operations must check whether their arguments have the right type. So far, we haven't tagged values in the heap, so we could very well execute (*rest empty*). (The result would be a vector out-of-bounds error, since (*Reference-location* 2) is -998.) The representation of *cons/prim* values on page 4 solves this problem by allocating three cells for each *cons/prim* pair: one cell for the tag 'Cons and two cells for the data. The selectors *first/prim* and *rest/prim* then check to see whether the 'Cons tag is present; otherwise, they report an error (at run-time).