Compound Data and Memory Management

Handout #50

CS251 Lecture 39
May 9, 2007

Franklyn Turbak

Wellesley College

Compound Data

Simple data are atomic — they have no parts. E.g. integers, floats booleans, characters. (But some languages, esp. C, can view even simple data as sequences of bits.) Compound data have parts:

- Products: datum has multiple value components. E.g. pairs, tuples, arrays, vectors, strings, records, structs, etc.
- Sums: datum is a tagged choice of several values. E.g., oneofs, variants, tagged sums, tagged unions, discriminated unions.
- Sum of Products: datum is one of several possible tagged products: E.g. linked lists, binary trees, abstract syntax trees, s-expressions. Implemented via OCaml data types, Java objects, Pascal variant records, XML trees.

Product Dimensions

- How are product values created and later decomposed into parts? Is there special syntax for creating the product or selecting/changing its parts?
- Are the components of the product indexed by position or by name? If by position, is indexing 0 or 1 based?
- When accessing a component, can its index be calculated or must an index be a manifest constant?
- Are the components values (as in call-by-value) or computations (as in call-by-name/call-by-need)?
- Are the components of the product immutable or mutable?
- Is the length of the product fixed or variable?
- Are all components of the product required to have the "same type," i.e., are products *homogeneous*?
- When products are nested, are the nested components all required to have the same size and/or "shape"?
- How are products passed as arguments, returned as results, and stored in assignments?
- Can the lifetime of a product exceed the lifetime of an invocation of a procedure in which it is created?

Pairs: The Simplest Product

- OCaml's immutable pairs has parts accessible via `fst` and `snd` or pattern matching:

```
# let p = (17,true);;
val p : int * bool = (17, true)

# if snd(p) then fst(p)*2 else 42;;
- : int = 34

# let (i,b) = p in if b then i*2 else 42;;
- : int = 34
```

- Standard ML of New Jersey (SMLNJ) has immutable pairs whose parts are accessible via #1 and #2 or pattern matching:

```
- val p = (17,true);
val p = (17,true) : int * bool

- if #2(p) then #1(p)*2 else 42;
val it = 34 : int

- let val (i,b) = p in if b then i*2 else 42 end;
val it = 34 : int
```

- Pair components can be made mutable via explicit cells. E.g.

```
# let p = (ref 17, true) in (fst p := (! (fst p) + 1); p);;
- : int ref * bool = ({contents = 18}, true)
```

Pairs in Scheme

- Dynamically typed mutable pairs in Scheme created via `cons`, selected via `car` and `cdr`, and changed via `set-car!` and `set-cdr!`.

```
(let* ((p (cons 17 #t))
       (a (car p)))
  (begin (set-car! p (cdr p)) (set-cdr! p a) p))
;Value 1: (#t . 17) ; A "dotted pair"
```

- Scheme lists are just `cdr`-linked pairs terminated with the empty list `'()` (equivalent to `#f` in some implementations).

```
(cons 17 (cons #t (cons "foo" '())))
;Value 2: (17 #t "foo") ; Abbreviation of (17 . (#t . ("foo" . ())))
```

```
(list 17 #t "foo")
;Value 3: (17 #t "foo")
```

- The quotation notation for symbols (`(quote` *`symbol`*`)`, abbreviated `'`*`symbol`*) is also used for dotted pairs and lists.

```
'((17 . #t) (fun (a b) (bind c (+ a b) (/ a b))))
;Value 4: ((17 . #t) (fun (a b) (bind c (+ a b) (/ a b))))
```


Pairs in Other Languages

- Can make immutable and mutable pairs in Java, but have type headaches:
  - *Approach 1*: make different class for every pair of component types. Yuck!
  - *Approach 2*: make one `Pair` class that holds two `Objects`, but then have to (1) wrap/unwrap small values like integers and characters and (2) cast components upon extraction. Yuck!
  - *Approach 3*: Java 1.5 supports *generic classes* that are parameterized by type. Can define a class `Pair<S,T>` that pairs objects of type `S` with those of type `T` and can instantiate these with any object types. Wrapping and casting is still necessary, but is handled automatically by Java in most cases.
- Can use C structs (records), but
  - need different struct type for every pair of component types;
  - we'll see that semantics is somewhat surprising.

Product Components can be Lazy

Parameter passing mechanisms can be applied to product components: they can be values (call-by-value), thunks (call-by-name), or memoized promised (call-by-lazy).
E.g. what is the behavior of the following example under CBV, CBN, and CBL?

```
(bind p (pair (println (+ 1 2))
              (println (+ 3 4)))
  (+ (snd p) (* (fst p) (fst p))))
```


Parameter Passing of Mutable Products

Consider passing mutable pairs in HOILIC-like language:

```
(bind p (pair 2 3)
  (bind f (fun (r) (seq (setfst r (+ (fst r) (snd r)))
                        (setsnd p (* (fst p) (snd p)))))
    (seq (f p) (println (fst p)) (println (snd p)))))
```

- In **call-by-value-sharing** (Ocaml, Scheme, Java, C arrays), parameter `r` shares the same mutable storage with p.
- In **call-by-value-copy** (C structs, Pascal arrays and records), parameter `r` has mutable slots distinct from p that are initialized to the contents of p's slots.

Positional Products

- Both OCaml and SMLNJ support arbitrary length statically typed immutable heterogeneous tuples and fixed-length mutable homogeneous arrays. In OCaml, a general tuple is only decomposable via pattern matching; in SMLNJ, the `#i` syntax may also be used.
- Scheme has dynamically typed arbitrary-length mutable heterogeneous lists and fixed-length mutable heterogeneous vectors.
- Java has statically typed mutable fixed-length homogeneous arrays and extensible heterogeneous (sort of, via `Object` subtyping) vectors.
- C has (weakly) statically typed mutable fixed-length homogeneous arrays, but since arrays don't "know" size, can access out-of-bounds array indices.
- Pascal has statically typed mutable fixed-length homogeneous arrays. Bizarrely, array size is part of type, so procedures aren't polymorphic over arrays of different size!

Products with Named Components

Many languages support products with named components. E.g.

- OCaml/SMLNJ records have immutable components by default. (SMLNJ tuples are just sugar for records.)
- Java class instances have mutable components.
- C structs have mutable components.
- Pascal records have mutable components.
- Common Lisps `defstruct` facility manipulates records with mutable components.

Records in OCaml

```
# type person = name: string; mutable age: int; sex: bool;;
type person =  name : string; mutable age : int; sex : bool;

# let wendy = name="Wendy Wellesley"; age=19; sex=true;;
val wendy : person = name = "Wendy Wellesley"; age = 19; sex = true

# let will = name="William Wellesley"; age=57; sex=false;;
val will : person = name = "William Wellesley"; age = 57; sex = false

# let wanda = wendy with name = "Wanda Wellesley";; (* new person record
val wanda : person = name = "Wanda Wellesley"; age = 19; sex = true

# wendy.age;;
- : int = 19

# wendy.age <- wendy.age + 1;;
- : unit = ()

# wendy.age;;
- : int = 20

# wanda.age;;
- : int = 19
```

Note: the names `name`, `age`, and `sex` are in a single global namespace for record field names. A declaration of a new record type with one of these names makes the name inaccessible in the old record type.

Stack vs. Heap

Programs typically manipulate two areas of memory:

Stack: (i.e., Java Execution Land in CS11/CS230) The stack typically holds *activation/execution frames* for function/procedure/method invocations. Variables and compound data whose lifetime does not outlast the invocation may be stored in the frame. C/Java execution frames and local C compound data (arrays/structs) are stored here. When the invocation returns, the frame is popped off the stack, implicitly deallocating any data in the frame.

Heap: (i.e., Java Object Land in CS11/CS230) The heap holds *data blocks* whose lifetime may outlast the function/procedure/method invocation in which it was created. Java objects and Ocaml/Haskell/Scheme data/closures/environments are stored here. Heap blocks can be deallocated manually (as in C/Ada/Pascal) or automatically (via garbage collection, as in Java/Ocaml/Haskell/Scheme).

Inspecting the C Stack with Invalid Array References

```
int test (int* a, int lo, int hi) {
  int i; for (i=lo; i<=hi; i++)
           printf("%x:a[%d]=%d (%x)\n", &a[i], i, a[i], a[i]); }
int main () { int b[] = {17,42}; test(b, -17, 1); }
```

```
linux> gcc -o arrayrefs arrayrefs.c; ./arrayrefs
bffff74c:a[-17]=1073823076 (40013d64)    # return addr. of printf (stale
bffff750:a[-16]=134513912 (80484f8)      # address of format string
bffff754:a[-15]=-1073744048 (bffff750)   # address of &a[i] (stale)
bffff758:a[-14]=-15 (fffffff1)           # i (stale)
bffff75c:a[-13]=-15 (fffffff1)           # a[i] (stale)
bffff760:a[-12]=-15 (fffffff1)           # a[i] (stale)
...                                      # unused slots
bffff774:a[-7]=-7 (fffffff9)             # i (for i = -7)
bffff778:a[-6]=-1073743976 (bffff798)    # saved base pointer for main
bffff77c:a[-5]=134513799 (8048487)       # return address of test call
bffff780:a[-4]=-1073743984 (bffff790)    # 1st arg to test = &b[0]
bffff784:a[-3]=-17 (ffffffef)            # 2nd arg to test
bffff788:a[-2]=1 (1)                     # 3rd arg to test
bffff78c:a[-1]=134513633 (80483e1)       # unused slot
bffff790:a[0]=17 (11)                    # b[0]
bffff794:a[1]=42 (2a)                    # b[1]
```


String Overwriting in C

```
// strings.c
// illustrates how one string can overwrite another in c
int main () {
  char a[] = "foo";
  char b[] = "bar";
  printf("a=%s; b=%s\n",a,b);
  strcpy(b,"bazquux");
    // strcpy(dest,src) is a built-in string copy function
  printf("a=%s; b=%s\n",a,b);
}

linux> gcc -o strings strings.c; ./strings
a=foo; b=bar
a=uux; b=bazquux
```

Stack-Allocated C Arrays Cannot be Returned

```
// stack-arrays.c
void printarray(char* s, int* a, int n) {
  int i; for (i = 0; i < n; i++) {printf("%s[%d] = %d\t", s, i, a[i]);}
  printf("\n");}

int* elts (int n, int scale) {
  int a[n]; // Stack allocated array
  int i; for (i = 0; i < n; i++) { a[i] = scale*i; }
  printarray("a",a,n); return a; }

int main () ] {
  int* b; int* c; b = elts(4,1); printarray("b",b,4);
  c = elts(4,2); printarray("b",b,4); printarray("c",c,4); }

linux> gcc -o stack-arrays stack-arrays.c; ./stack-arrays
stack-arrays.c: In function ‘elts’:
stack-arrays.c:20: warning: function returns address of local variable
a[0] = 0  a[1] = 1  a[2] = 2  a[3] = 3
b[0] = 0  b[1] = 1108542220  b[2] = -1073744264  b[3] = 134513720
a[0] = 0  a[1] = 2  a[2] = 4  a[3] = 6
b[0] = 0  b[1] = 1108542220  b[2] = -1073744264  b[3] = 134513720
c[0] = 0  a[1] = 1108542220  c[2] = -1073744264  c[3] = 134513720
```


One Way to Fix Problem

```
// stack-arrays2.c
void printarray(char* s, int* a, int n) {
  int i; for (i = 0; i < n; i++) {printf("%s[%d] = %d\t", s, i, a[i]);}
  printf("\n");}

int* elts (int* a, int n, int scale) {
  // Pass in already allocated array a to elts
  int i; for (i = 0; i < n; i++) { a[i] = scale*i;}
  printarray("a",a,n); return a;}

int main () {
  // Fix problem in stack-arrays.c by allocating arrays outside elts;
  int b[4]; int c[4]; elts(b,4,1); printarray("b",b,4);
  elts(c,4,2); printarray("b",b,4); printarray("c",c,4); }

linux> gcc -o stack-arrays2 stack-arrays2.c; ./stack-arrays2
a[0] = 0        a[1] = 1        a[2] = 2        a[3] = 3
b[0] = 0        b[1] = 1        b[2] = 2        b[3] = 3
a[0] = 0        a[1] = 2        a[2] = 4        a[3] = 6
b[0] = 0        b[1] = 1        b[2] = 2        b[3] = 3
c[0] = 0        c[1] = 2        c[2] = 4        c[3] = 6
```

Another Way to Fix Problem: Heap Allocation

C provides the following functions for manual heap storage management:

> `void *malloc (size_t size);`
> allocates `size` bytes and returns a pointer to the allocated memory. The memory is not cleared. Returns the `NULL` pointer if the request fails.
>
> `void free(void *ptr);`
> frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc()`. Otherwise, or if `free(ptr)` has already been called before, undefi ned behaviour occurs. If `ptr` is `NULL`, no operation is performed.

Heap Allocated Arrays in C

```
// heap-arrays.c
void printarray(char* s, int* a, int n) {
  int i; for (i = 0; i < n; i++) {printf("%s[%d] = %d\t", s, i, a[i]);}
  printf("\n");}

int* elts (int n, int scale) {
  int* a = (int *) malloc(n*sizeof(int)); // Heap allocated array:
  int i; for (i = 0; i < n; i++) { a[i] = scale*i;}
  printarray("a",a,n); return a;}

int main () {
  int* b; int* c; b = elts(4,1); printarray("b",b,4);
  c = elts(4,2); printarray("b",b,4); printarray("c",c,4);}

linux> gcc -o heap-arrays heap-arrays.c; ./heap-arrays
a[0] = 0        a[1] = 1        a[2] = 2        a[3] = 3
b[0] = 0        b[1] = 1        b[2] = 2        b[3] = 3
a[0] = 0        a[1] = 2        a[2] = 4        a[3] = 6
b[0] = 0        b[1] = 1        b[2] = 2        b[3] = 3
c[0] = 0        c[1] = 2        c[2] = 4        c[3] = 6
```

Problems with Manual Heap Storage Management

- **Storage Leak:** If do not `free` storage that is no longer accessible, can run out of heap space.
- **Dangling Pointer:** If `free` a pointer to a heap block that is still in use, unpredictable behavior can result.

```
// dangling.c
int main () {
  int* a; int *b;
  a = (int *) malloc(10);
  a[0] = 17;
  free(a); // Any reference to a after this is dangling
  b = (int *) malloc(10);
  b[0] = 42;
  printf("a[0]=%d; b[0]=%d\n", a[0], b[0]);}

linux> gcc -o dangling dangling.c; ./dangling
a[0]=42; b[0]=42
```


Automatic Heap Storage Management: Garbage Collection

- Can automatically reclaim storage that is no longer accessible from a program a via a process called **garbage collection (GC)**.
- All storage blocks reachable from the **root set** (typically processor registers) are **live** and are preserved. All others are **dead** and are reclaimed.
- We'll consider several approaches to GC in the context of the following Scheme example:

```
(let* ((a (cons 1 (cons 2 '())))
       (b (cons 3 (cons 4 '())))
       (c (cons a b)))
  (begin (set-cdr! (cdr b) b)
         (set-cdr! (cdr a) (cdr a))
         (set-car! c (cdr b))
         c))
```

  Assume the returned pair is the GC root.
- Some languages allow specifying actions to perform when a storage block is reclaimed. E.g., Java `finalize` method and C++ destructors.
- Garbage collection is essential to program modularity. Without it, how can we know in a large system when it's safe to free memory?

GC: Reference Counting

- *Idea:* Keep track of the number of pointers to each heap-allocated block and reclaim the block when this number reaches 0;
- Some C++ implementations use reference counting for GC.
- *Advantage:* Easy to perform incrementally.
- *Disadvantages:*
  - Need space to maintain the reference counts.
  - Reference counts must be updated at every allocation and assignment;
  - Doesn't reclaim cyclic data.

GC: Mark-Sweep

- *Idea:* Maintain a free list of all storage blocks from which new storage is allocated. For simplicity, assume all blocks are pairs. Each block has a **mark bit** that is initially false. When free list is exhausted, perform GC in two phases:
  1. **Mark phase** Trace through all blocks accessible from root set, setting the mark bit of every accessible block.
  2. **Sweep phase** Sweep through all blocks. Unmarked blocks are reclaimed by adding them to the free list. Marked blocks have their mark bit unset.
- *Advantages:* (1) Easy to understand and (2) only requires one bit per block.
- *Disadvantages:*
  - Storage for mark bits.
  - Cleverness needed to avoid recursion stack in mark phase.
  - System must pause while GC takes place.
  - Sweep phase touches all memory (mark phase touches only live memory).
  - Memory fragmentation.

GC: Stop and Copy

- *Idea:* Split memory into two equal-sized **semispaces**. Allocate blocks from "current" semispace (other used only for collection). When current semispace is exhausted, copy only accessible blocks to other semispace, and make it the new "current" semispace.
- *Advantages:*
  - Simple to allocate and trace arbitrary sized blocks.
  - Copy phase touches only live memory.
  - Copy phase compacts memory, avoiding fragmentation.
- *Disadvantages:*
  - Half of memory is unused!
  - Need to pause for GC (but there are incremental versions).

GC: Stop and Copy Algorithm

Call exhausted semispace **from-space** and the other semispace **to-space**. GC copies live blocks in from-space to to-space using two pointers into to-space named `scan` and `free`. Invariants: (1) `scan` $\leq$ `free`; (2) pointers before `scan` point to to-space; (3) pointers between `scan` and `free` point to from-space; (4) from-space blocks already moved to to-space contain a forwarding address to to-space in first slot.

```
// Pseudocode
copy k root pointers beginning of to-space
scan = beginning of to-space
free = scan + k
while scan != free
  if mem[scan] is pointer to not-yet-moved from-space block then
    copy block to mem[free .. free+(n-1)]; // assume n is block size
    mem[mem[scan]] = free; // Leave forwarding address
    mem[scan] = free; // Update pointer to to-space.
    free = free + n;
  else if mem[scan] is pointer to already moved from-space block then
    mem[scan] = mem[mem[scan]]; // Use forwarding address
  // Do nothing if mem[scan] is a non-pointer
  scan = scan + 1
// When scan = free, collection is done. Start allocating from free.
```

GC: Conservative GC

- Precise GC requires distinguishing pointers and non-pointers.
- In some language implementations (esp. C, C++) this is not possible.
- **Conservative GC** treats everything that *might be* a pointer as a pointer. Will preserve some blocks that are reclaimed in precise systems.

C Points as Structs

```
// points-struct.c
typedef struct P {int x; int y;} point;
point scaledCopy (int s, point p) {
  point q; q.x = s * p.x; q.y = s * p.y; return q;}
void scale1 (int s, point p) { // Call by copy, not sharing!
  p.x = s * p.x; p.y = s * p.y; }
void scale2 (int s, point* p) {
  (*p).x = s * (*p).x; (*p).y = s * (*p).y; }
void printPoint (point p) {
  printf("x=%d;y=%d\n", p.x, p.y); }
int main () {
  point a,b; a.x = 1; a.y = 2;
  b = scaledCopy(3,a); printPoint(a); printPoint(b);
  scale1(4,a); scale2(5,&b); printPoint(a); printPoint(b);}
```

```
linux> gcc -o points-struct points-struct.c; ./points-struct
x=1;y=2
x=3;y=6
x=1;y=2
x=15;y=30
```

C Points as Stack-Allocated Arrays

```
// points-array.c
/* Represent a point as a 2-slot integer stack array,
   with x in slot 0 and y in slot 1. */

typedef int point[2];

void scaledCopy (int s, point p, point q) { // Must pass in result array
  q[0] = s * p[0]; q[1] = s * p[1]; }

void scale1 (int s, point p) { // Call by sharing!
  p[0] = s * p[0]; p[1] = s * p[1]; }

void scale2 (int s, point* p) {
  (*p)[0] = s * (*p)[0]; (*p)[1] = s * (*p)[1]; }

void printPoint (point p) {printf("x=%d;y=%d\n", p[0], p[1]); }

int main () { point a,b; a[0] = 1; a[1] = 2;
  scaledCopy(3,a,b); printPoint(a); printPoint(b);
  scale1(4,a); scale2(5,&b); printPoint(a); printPoint(b); }
```

```
linux> gcc -o points-array points-sarray.c; ./points-sarray
x=1;y=2
x=3;y=6
x=4;y=8
x=15;y=30
```


C Points as Heap-Allocated Arrays

```
// points-harray.c
/* Represent a point as a 2-slot integer heap array,
   with x in slot 0 and y in slot 1. */

typedef int* point;

point makePoint (int x, int y) {
  point p = (point) malloc(2*sizeof(int));
  p[0] = x; p[1] = y; return p;}

point scaledCopy (int s, point p) { return makePoint(s*p[0], s*p[1]);}

void scale (int s, point p) { p[0] = s*p[0]; p[1] = s*p[1]; }

void printPoint (point p) { printf("x=%d;y=%d\n", p[0], p[1]); }

int main () { point a,b; a = makePoint(1,2);
  b = scaledCopy(3,a); printPoint(a); printPoint(b);
  scale(4,a); scale(5,b); printPoint(a); printPoint(b);}
```

```
linux> gcc -o points-harray points-harray.c; ./points-harray
x=1;y=2
x=3;y=6
x=4;y=8
x=15;y=30
```

Integer Lists in C

```
// sumlist.c
#include <stddef.h>
typedef struct IL {int head; struct IL *tail;} intlist;
int sumlist (intlist* lst) {
  if (lst == NULL) return 0;
  else return (*lst).head + sumlist((*lst).tail);}
intlist* fromTo (int lo, int hi) {
  intlist* result;
  if (lo > hi) return NULL;
  else { result  = (intlist*) malloc(sizeof(intlist));
         (*result).head = lo;
         (*result).tail = fromTo(lo + 1, hi);
         return result; } }
int main () {
  printf("sumlist(fromTo(1,10))=%d\n", sumlist(fromTo(1,10))); }
linux> gcc -o sumlist sumlist.c; ./sumlist
sumlist(fromTo(1,10))=55
```