Lecture 14
Generic Data Structures

15-122: Principles of Imperative Computation (Spring 2017)
Rob Simmons

1 Introduction

Our story about client interfaces in the previous lecture was incomplete. We were able to use client interfaces to implement queues and hash tables that treat the client's `elem` type as abstract, but any given program could only have a *single* type of element, a *single* way of hashing.

To solve these problems, we will have to move beyond the C0 language to a language we call C1. C1 gives us two important features that aren't available in C0. The first new feature is *function pointers*, which allow us to augment hash sets with *methods*, an idea that is connected to Java and object-oriented programming. The second new feature is a *void pointer*, which acts as a generic pointer.

Starting in this lecture, we will be working in an extension of C0 called C1. To get the `cc0` compiler to recognize C1, you need to use a `.c1` extension. Coin does not currently accept C1.

Relating to our learning goals, we have

Computational Thinking: Structs with function pointers that can be used to modify the data contained within the struct is an important idea from object oriented programming.

Algorithms and Data Structures: We will revisit the idea of hash sets in a new setting.

Programming: We explore function pointers and void pointers, which are necessary for creating truly generic data structures in C0/C1.

2 Hash Set Review

In the last lecture, we talked about a client interface for hash sets that allowed us to treat the type `elem` of hash table elements as *abstract* to the library, and therefore changeable by the client. Recall this interface, as we developed it in the last lecture:

```
/*** Client interface ***/
// typedef _______ elem;           // Supplied by client
bool elem_equiv(elem x, elem y);   // Supplied by client
int elem_hash(elem x);             // Supplied by client

/*** Library interface ***/
// typedef ______* hset_t;

hset_t hset_new(int capacity)
/*@requires capacity > 0; @*/
/*@ensures \result != NULL; @*/ ;

elem hset_contains(hset_t H, elem x)
/*@requires H != NULL; @*/ ;

void hset_add(hset_t H, elem x)
/*@requires H != NULL; @*/
/*@ensures hset_contains(H, x); @*/ ;
```

There is still a significant problem with this structure of a client/library interface. Within a given program, we could only instantiate the client interface type and the client functions *once*. This is a problem: even if we are okay committing to a single type of element, like a struct with two fields `color` and `fruit`

```
1 struct produce {
2   string color;
3   string fruit;
4 };
```

there are multiple reasonable ways of instantiating the two client functions `elem_equiv` and `elem_hash`.

If we treat elements as equivalent only if all fields are equal, then we can have a set containing red, yellow, and green apples, red and blue berries, and yellow and green bananas.

```
6  bool produce_equiv_all(struct produce* x, struct produce* y)
7  //@requires x != NULL && y != NULL;
8  {
9    return string_equal(x->color, y->color)
10       && string_equal(x->fruit, y->fruit);
11 }
12
13 int produce_hash_all(struct produce* x)
14 //@requires x != NULL;
15 {
16   return hash_string(string_join(x->color,
17                      string_join(" ", x->fruit)));
18 }
```

Maybe we only care about which fruits the table contains (so that red apples and green apples are the same from the perspective of the set).

```
20 bool produce_equiv_color(struct produce* x, struct produce* y)
21 //@requires x != NULL && y != NULL;
22 {
23   return string_equal(x->color, y->color);
24 }
25
26 int produce_hash_color(struct produce* x)
27 //@requires x != NULL;
28 {
29   return hash_string(x->color);
30 }
```

Our first goal will be to allow, *within a single program,* the use of hash sets that treat all fruits the as equivalent, all colors as equivalent, and sets that only contain fruits with the same color and name as equivalent. We will accomplish this with function pointers.

3 Function Pointers

Although we saw two different functions for equivalence checking above, as long as we have

```
typedef struct produce* elem;
```

they both match the hash set client interface declaration for `elem_equiv`; in fact, any function that checks two `struct produce` values for equality has a declaration of the form:

```
bool equiv(struct produce* x, struct produce* y);
```

The function name can be anything we choose, and so can the parameter names, but the return type and the parameter types are fixed (as well as the contract, with parameter names suitably adjusted).

In C1, we can define a *type* capturing all such functions by just adding the word **typedef** to the beginning of the example declaration above:

```
typedef bool equiv_fn(struct produce* x, struct produce* y);
```

By convention, we use `_fn` as a suffix for function types, just like we used `_t` as a suffix for client-side abstract data types.

When we call a function by writing `equiv(a,b)`, the code that executes is stored in memory in compiled form. Since the amount of memory is large (compared to C0 *small types*), it is illegal to write code that treats functions like other values, as below:

```
equiv_fn f;                                 // NOT ALLOWED
f = produce_equiv_all;                      // NOT ALLOWED
println(f(a,b) ? "equiv" : "not equiv");
f = produce_equiv_color;                    // NOT ALLOWED
println(f(a,b) ? "equiv" : "not equiv");
```

Instead, in the extended language C1, we can declare *pointers* to functions of type `equiv_fn`.

```
equiv_fn* g;
```

We don't create function pointers by dynamically allocating them the way we do structs: all the functions we could possibly have in our program are already known when we compile the program. Instead, we use a new operator, `&` (read as *address-of*). If we write `&produce_equiv_all`, we obtain a pointer to the function `produce_equiv_all`. So we can assign the pointer to `g` as follows, as long as `produce_equiv_all` has the correct type.

```
equiv_fn* g = &produce_equiv_all;
```

When we call a function using a function pointer, we have to use parentheses like this: `(*g)(a,b)`, because `*g(a,b)` is parsed the same way as `*(g(a,b))`.

We can correct the illegal example above to:

```
equiv_fn* f;
f = &produce_equiv_all;
println((*f)(a,b) ? "equiv" : "not equiv");
f = &produce_equiv_color;
println((*f)(a,b) ? "equiv" : "not equiv");
```

Like all other pointers, function pointers can be `NULL`, and it is a safety violation to dereference a `NULL` function pointer.

4 Methods For Hash Sets

Remember our goal: to allow, *within a single program,* the use of hash sets that act like sets of colored fruits, and other hash sets that act like maps from colors to fruits. To achieve this, we will make hash tables more generic.

The first step is to change the client interface from declaring two particular functions `elem_equiv` and `elem_hash` to declaring two function *types*, `elem_equiv_fn` and `elem_hash_fn`.

```
1 /*** Client interface ***/
2 typedef struct produce* elem;
3
4 typedef bool elem_equiv_fn(elem x, elem y);
5
6 typedef int elem_hash_fn(elem x);
```

Now our old code for checking that an element is in a hash set is broken, because the functions `elem_hash` and `elem_equiv` are no longer in the client interface.

```
8  bool hset_contains(hset* H, elem x)
9  //@requires is_hset(H);
10 {
11   int i = abs(elem_hash(x) % H->capacity); // BROKEN
12   for (chain* p = H->table[i]; p != NULL; p = p->next) {
13     if (elem_equiv(p->data, x)) {          // BROKEN
14       return true;
15     }
16   }
17
18   return false;
19 }
```

In C1 with function pointers, there are at least two options: one option is that we could pass along pointers to the needed functions as extra arguments to the `hset_contains`.

```
8  bool hset_contains(hset* H, elem x,
9                     elem_hash_fn* hash, elem_equiv_fn* equiv)
10 //@requires is_hset(H, hash, equiv);
11 //@requires x != NULL && hash != NULL && equiv != NULL;
12 {
13   int i = abs((*hash)(x) % H->capacity);
14   for (chain* p = H->table[i]; p != NULL; p = p->next) {
15     if ((*equiv)(p->data, x)) {
16       return true;
17     }
18   }
19
20   return false;
21 }
```

This is not the best option, though: the data structure invariants of our hash set implementation require that the placement of each element into a chain *makes sense with respect to the hash function*. We might want to have two different hash sets during the course of a program: `H1` using `produce_hash_all` and `H2` using `produce_hash_color`, but it will never make sense to have a single hash set change which hash function it is using. So the right ap-

proach is to store the two functions as fields of the **struct** hset_header that is created when we call hset_new.

```
typedef struct hset_header hset;
struct hset_header {
  int size;              // 0 <= size
  int capacity;          // 0 < capacity
  chain*[] table;        // \length(table) == capacity
  elem_equiv_fn* equiv;  // non-NULL
  elem_hash_fn* hash;    // non-NULL
};
```

Now our data structure invariant has to ensure that the function pointers are not NULL, and hset_new has to store the function pointers in the struct:

```
bool is_hset(hset* H) {
  return  H != NULL
    && H->capacity > 0
    && H->size >= 0
    && H->equiv != NULL
    && H->hash != NULL
    && is_table_expected_length(H->table, H->capacity)
 /* && each element satisfies its own representation invariants */
 /* && there aren't equivalent elements */
 /* && the number of elements matches the size */
 /* && every element in H->table[i] hashes to i */
}

hset* hset_new(int capacity,
               elem_equiv_fn* equiv, elem_hash_fn* hash)
//@requires capacity > 0 && equiv != NULL && hash != NULL;
//@ensures is_hset(\result);
{
  hset* H = alloc(hset);
  H->size = 0;
  H->capacity = capacity;
  H->table = alloc_array(chain*, capacity);
  H->equiv = equiv;
  H->hash = hash;
  return H;
}
```

The function pointers `H->equiv` and `H->hash` can be thought of as *methods* in an *object-oriented* language like Java: they are functions bundled with a particular hash set object that help us interpret the data in that object. Syntactically, compared to object-oriented languages, it's a little bit cumbersome to call these functions with the C1 notation. To avoid a lot of calls that use the cumbersome notation `(*H->equiv)(x,y)`, we will write some helper functions so we can use calls that look like `elemequiv(H,x,y)`.

```
37 bool elemequiv(hset* H, elem x, elem y)
38 //@requires H != NULL && H->equiv != NULL;
39 {
40   return (*H->equiv)(x, y);
41 }
42
43 int elemhash(hset* H, elem x)
44 //@requires H != NULL && H->capacity > 0 && H->hash != NULL;
45 //@ensures 0 <= \result && \result < H->capacity;
46 {
47   return abs((*H->hash)(x) % H->capacity);
48 }
49
50 bool hset_contains(hset* H, elem x)
51 //@requires is_hset(H, hash, equiv);
52 //@requires hash != NULL && equiv != NULL;
53 {
54   int i = elemhash(H, x);
55   for (chain* p = H->table[i]; p != NULL; p = p->next) {
56     if (elemequiv(H, x, p->data)) {
57       return true;
58     }
59   }
60
61   return false;
62 }
```

Now we can achieve the generic behavior we wanted by creating two hash tables, one that acts like a set of colored fruits (`H1`) and one that acts like an associative array where keys are colors and values are fruits (`H2`).

```
hset_t H1 = hset_new(100, &produce_equiv_all, &produce_hash_all);
hset_t H2 = hset_new(100, &produce_equiv_color, &produce_hash_color);
hset_t H3 = hset_new(100, &produce_equiv_all, &produce_hash_color);
```

The third hash table, `H3`, works just like `H1`, but it may be much less efficient, because we know that any two distinct fruits with the same color will *definitely* collide. This is bad, but it is not as bad as a hash table which has the equivalence function `produce_equiv_color` and the hash function `produce_hash_all`. Such a hash set would be fundamentally broken; the invariants of a hash table require that if any two elements are equivalent, they *must* have the same hash value.

5 Generic Pointers: `void*`

Our different types of hash sets can co-exist in one program, which was our goal, but this is only true because they are storing the same type of element, **`struct`** `produce`. But we often need more generality than this: for instance, in the Clac program, we needed two stacks storing different types of elements. We can generalize our hash sets, abstracting away the element type using another new feature of C1, the *void pointer* (written **`void`**`*`.) Using this feature, we can avoid the error-prone and wasteful copying of the hash set library that would otherwise be needed for each type of hash set element used in a program.

It should be said that calling it **`void`**`*` is a terrible name! (Blame C.) A variable `p` of type **`void`**`*` is allowed to hold a pointer to *anything*, not just to **`void`**. Any pointer can be turned into a void pointer by a *cast*:

```
void* p1 = (void*)alloc(int);
void* p2 = (void*)alloc(string);
void* p3 = (void*)alloc(struct produce);
void* p4 = (void*)alloc(int**);
```

When we have a void pointer, we can turn it back into the type it came from by casting in the other direction:

```
int* x = (int*)p1;
string x = *(string*)p2;
```

At runtime, a non-`NULL` void pointer has a *tag*: casting incorrectly, like trying to run `(char*)p1` in the example above, is a safety violation: it will cause a memory error just like a `NULL` dereference or array-out-of-bounds error.

These tags make void pointers a bit like values in Python: a void pointer carries the information about its true pointer type, and an error is raised if we treat a pointer to an integer like a pointer to a string or vice versa. Inside of contracts, we can check that type with the `\hastag(ty,p)` function:

```
//@assert \hastag(int*, p1);
//@assert \hastag(string*, p2);
//@assert \hastag(int***, p4);

//@assert !\hastag(string*, p1);
//@assert !\hastag(int**, p1);
//@assert !\hastag(int***, p1);
```

One quirk: the `NULL` void pointer is just `NULL`, so `\hastag(ty, NULL)` always returns `true` and we can do slightly strange things like this without any error:

```
void* p = NULL;
void* x = (void*)(int*)(void*)(string*)(void*)(struct produce*)p;
```

6 Generic Hash Sets

Function pointers allowed us to implement hash sets without committing to a single implementation of our hashing and equivalence functions, but without `void*`, C1 still requires us to commit to a single implementation of the `elem` type. But if we let the type of an element be `void*`, then that's no commitment at all: we can use *any* pointer type as our element.

```
1  /************************/
2  /*** Client interface ***/
3  /************************/
4
5  // Library treats this like typedef _______ elem;
6  typedef void* elem;
7  typedef bool elem_equiv_fn(elem x, elem y);
8  typedef int elem_hash_fn(elem x);
```

The cost of this approach to generic data structures is that, within our client functions, we need to cast the generic pointers back to their original types before we use them.

```
10 bool produce_equiv_all(void* x, void* y)
11 //@requires x != NULL && \hastag(struct produce*, x);
12 //@requires y != NULL && \hastag(struct produce*, y);
13 {
14   struct produce* a = (struct produce*)x;
15   struct produce* b = (struct produce*)y;
```

```
16   return string_equal(a->color, b->color)
17       && string_equal(a->fruit, b->fruit);
18 }
```

Because the hash set implementation still treats the `elem` type as abstract, as long as we only give the hashtable **`struct`** `produce` pointers that have been cast to **`void`**`*`, we can be sure that only generic pointers that are correctly tagged with **`struct`** `produce*` will ever get passed to the equivalence function, meaning that whenever the method `produce_equiv_all` is called by the hash set implementation, the precondition should always be satisfied.

```
hset_t H1 = hset_new(10, &produce_equiv_color, &produce_hash_color);

struct produce* redapple = alloc(struct produce);
redapple->color = "red";
redapple->fruit = "apple";

struct produce* redberry = alloc(struct produce);
redberry->color = "red";
redberry->fruit = "berry";

hset_insert(H1, (void*)redapple);
assert(hset_lookup(H1, (void*)redberry));
```

With this new interface, it's now possible for one program to use hash sets storing different types of elements without duplicating the hash set implementation code. For example, the following code could be added to the "produce" client.

```
20 struct prod_count {
21   string fruit;
22   int count;
23 };
24
25 bool prod_count_equiv(void* x, void* y)
26 //@requires x != NULL && \hastag(struct prod_count*, x);
27 //@requires y != NULL && \hastag(struct prod_count*, y);
28 {
29   return string_equal(((struct prod_count*)x)->fruit,
30                       ((struct prod_count*)y)->fruit);
31 }
```

```
32
33 int prod_count_hash(void* x)
34 //@requires x != NULL && \hastag(struct prod_count*, x);
35 {
36   return hash_string(((struct prod_count*) x)->fruit);
37 }
38
39 void* prod_count(string fruit, int count)
40 //@ensures \result != NULL;
41 //@ensures \hastag(struct prod_count*, \result);
42 {
43   struct prod_count* x = alloc(struct prod_count);
44   x->fruit = fruit;
45   x->count = count;
46   return (void*)x;
47 }
48
49 int main() {
50   elem greenapple = produce("green", "apple");
51   elem redapple = produce("red", "apple");
52   elem redberry = produce("red", "berry");
53   elem blueberry = produce("blue", "berry");
54   elem apple_count = prod_count("apple", 2);
55   elem berry_count = prod_count("berry", 2);
56
57   hset_t H1 = hset_new(10, &produce_equiv_all,
58                        &produce_hash_all);
59   hset_t H3 = hset_new(10, &prod_count_equiv,
60                        &prod_count_hash);
61
62   // some test code here
63   return 0;
64 }
```

Notice that even though `produce()` and `prod_count()` create different structures, from the hash set's point of view both produce things of type `elem`, i.e., `void*`. The type of the hash set elements has been abstracted away, as it should be, since the hash set implementation should work for any element type as long as it's a pointer type.