Lecture Notes on
Generic Data Structures

15-122: Principles of Imperative Computation
Rob Simmons

Lecture 22
March 5, 2015

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 another 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;

bool elem_equiv(elem x, elem y)
  /*@requires x != NULL && y != NULL; @*/ ;

int elem_hash(elem x)
  /*@requires x != NULL; @*/ ;

/*** Library interface ***/
typedef _____* hset_t;

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

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

void hset_insert(hset_t H, elem x)
  /*@requires H != NULL && x != NULL; @*/
  /*@ensures hset_lookup(H, x) == 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`

```
struct produce {
  string color;
  string fruit;
};
```

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 end up with a `hset_t` that acts like a regular set: we can have a set containing red, yellow, and green apples, red and blue berries, and yellow and green bananas.

```
bool produce_equiv_all(struct produce* x, struct produce* y)
//@requires x != NULL && y != NULL;
{
  return string_equal(x->color, y->color)
    && string_equal(x->fruit, y->fruit);
}


int produce_hash_all(struct produce* x)
//@requires x != NULL;
{
  return hash_string(string_join(x->color,
                     string_join(" ", x->fruit)));
}
```

This is not the only option! If we have a hash function and equivalence function that is based only on the `color` field of the struct, that we end up with a `hset_t` that operates like an *associative array* – a hash set can only contain one red thing (which might be an apple or a berry) and one yellow thing (which might be a banana or an apple).

```
bool produce_equiv_color(struct produce* x, struct produce* y)
//@requires x != NULL && y != NULL;
{
  return string_equal(x->color, y->color);
}


int produce_hash_color(struct produce* x)
//@requires x != NULL;
{
  return hash_string(x->color);
}
```

Our first goal will be to allow, within a single C0 program to hash sets that act like sets of colored fruits, and other hash sets that act like maps from colors to fruits. We will accomplish this with function pointers.

3   Function Pointers

Any function that checks two "`struct produce`" pointers for equality has the same declaration:

```
bool equiv(struct produce* x, struct produce* y)
  /*@requires x != NULL && y != NULL; @*/ ;
```

The only difference is that the name may be different (it can be something besides `equiv`) and the name of the arguments can be different from `x` and `y`. The name of the function obviously matters a great deal, but the names of the function arguments is only relevant for the preconditions.

We can define a *type* capturing all such pointers by just adding the word `typedef` to the beginning of this declaration:

```
typedef bool equiv_fn(struct produce* x, struct produce* y)
  /*@requires x != NULL && y != NULL; @*/ ;
```

When we call a function normally by writing `equiv(a,b)`, the program actually has a bunch of code, stored in memory, that represents the compiled version of function `equiv`. This code represents a large chunk memory, much like how a struct is taken as a large chunk of memory. Therefore, C0 does not quite allow us to write code like this:

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

Instead, we can declare *pointers* to the function type `equiv_fn`.

```
equiv_fn* g;
```

When we call a function through the function pointer, we have to use parentheses `(*g)(a,b)`, because `*g(a,b)` is parsed by C0 the same way as `*(g(a,b))`. Like all other pointers, function pointers can be `NULL`, and it is a safety violation to dereference a `NULL` function pointer.

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, *address-of*. Taking the address of a function with `&produce_equiv_all` which takes a function that already exists, `produce_equiv_all`, and gives us a pointer to that function that has the correct type `equiv_fn`.

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

4   Methods For Hash Sets

We will make hash tables more generic by changing the client interface
from declaring two particular functions elem_equiv and elem_hash to declar-
ing two function *types*, elem_equiv_fn and elem_hash_fn.

```
/*** Client interface ***/
typedef struct produce* elem;

typedef bool elem_equiv_fn(elem x, elem y)
  /*@requires x != NULL && y != NULL; @*/ ;

typedef int elem_hash_fn(elem x)
  /*@requires x != NULL; @*/ ;
```

How will this change insertion into the hash table?

```
elem hset_lookup(hset* H, elem x)
//@requires is_hset(H);
//@requires x != NULL;
{
  int i = abs(elem_hash(x) % H->capacity);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    //@assert p->data != NULL;
    if (elem_equiv(p->data, x)) {
      return p->data;
    }
  }

  return NULL;
}
```

There are at least two options: one option is that we could pass along the relevant functions as extra arguments to the `hset_lookup`.

```
elem hset_lookup(hset* H, elem x,
                 elem_hash_fn* hash,
                 elem_equiv_fn* equiv)
//@requires is_hset(H, hash, eq);
//@requires x != NULL && hash != NULL && equiv != NULL;
{
  int i = abs((*hash)(x) % H->capacity);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    //@assert p->data != NULL;
    if ((*equiv)(p->data, x)) {
      return p->data;
    }
  }

  return NULL;
}
```

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 objects `hset_t` during the course of a program, `H1` that uses `produce_hash_all` and `H2` that uses `produce_hash_color`, but it will never make sense to have a single `hset_t` change which hash function it is using. As a result, the right approach 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;
  int capacity;              /* 0 < capacity */
  chain*[] table;            /* \length(table) == capacity */
  elem_equiv_fn* equiv;      /* non-NULL */
  elem_hash_fn* hash;        /* non-NULL */
};
```

```
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 is non-null */
 /* && 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 a language like Java: they are functions bundled with a particular hash set object that help us interpret the data in that object. It's a little bit cumbersome to call these functions with the C0/C1 notation. To avoid a lot of calls that use the cumbersome notation `(*H->equiv)(x,y)`, we will write some helper functions to make them calls that look like `elemequiv(H,x,y)`.

```
bool elemequiv(hset* H, elem x, elem y)
//@requires H != NULL && H->equiv != NULL;
{
  return (*H->equiv)(x, y);
}
```

```
int elemhash(hset* H, elem x)
//@requires H != NULL && H->capacity > 0 && H->hash != NULL;
//@requires x != NULL;
//@ensures 0 <= \result && \result < H->capacity;
{
  return abs((*H->hash)(x) % H->capacity);
}


elem hset_lookup(hset* H, elem x)
//@requires is_hset(H, hash, eq);
//@requires x != NULL && hash != NULL && eq != NULL;
{
  int i = elemhash(H, x);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    //@assert p->data != NULL;
    if (elemequiv(H, x, p->data)) {
      return p->data;
    }
  }

  return NULL;
}
```

Now, we can achieve the generic behavior we wanted by creating two hash tables, one that acts like a set of colored fruit (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*`

In addition to function pointers, we need to discuss one additional feature: the "void pointer" `void*`. It should be said that calling it `void*` is a terrible name! (Blame C.) A `void*` is just a generic type that is allowed to hold a pointer to *anything*. 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 C0/C1 still requires us to commit to a single implementation of the `elem` type. But if we commit to the type of an element being a generic `void*`, then that's no commitment at all: we can use *any* pointer type as our element.

```
/***********************/
/*** Client interface ***/
/***********************/

// typedef _____* elem;
typedef void* elem;

typedef bool elem_equiv_fn(elem x, elem y)
  /*@requires x != NULL && y != NULL; @*/ ;

typedef int elem_hash_fn(elem x)
  /*@requires x != NULL; @*/ ;
```

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.

```
bool produce_equiv_all(void* x, void* y)
//@requires x != NULL && \hastag(struct produce*, x);
//@requires y != NULL && \hastag(struct produce*, y);
{
  struct produce* a = (struct produce*)x;
  struct produce* b = (struct produce*)y;
  return string_equal(a->color, b->color)
    && string_equal(a->fruit, b->fruit);
}
```

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);
void* x = hset_lookup(H1, (void*)redberry);
assert(redapple == (struct produce*)x);
```