Lecture Notes on
Generic Data Structures

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 22
April 7, 2011

1 Introduction

Using `void*` to represent pointers to values of arbitrary type, we were able to implement *generic* stacks in that the types of the elements were arbitrary. The main remaining restriction was that they had to be pointers. Generic queues or unbounded arrays can be implemented in an analogous fashion. However, when considering, say, hash table or binary search trees, we run into difficulties because implementations of these data structures require operations on data provided by the client. For example, a hash table implementation requires a hash function and an equality function on keys. Similarly, binary search trees require a comparison function on keys with respect to an order. In this lecture we show how to overcome this limitation using function pointers as introduce in the previous lecture.

2 The Hash Table Interface Revisited

Recall the client-side interface for hash tables, in file ht.c0. The client must provide a type `elem` (which must be a pointer), a type `key` (which was arbitrary), a hash function on keys, an equality function on keys, and a function to extract a key from an element. We write `___` while a concrete type must be supplied there in the actual file.

```
/**********************************/
/* Hash table client-side interface */
/**********************************/
```

```
typedef ___* elem;
typedef ___ key;

int hash(key k, int m)
//@requires m > 0;
//@ensures 0 <= \result && \result < m;
  ;

bool key_equal(key k1, key k2);

key elem_key(elem e)
//@requires e != NULL;
  ;
```

We were careful to write the implementation so that it did not need to know what these types and functions were. But due to limitations in C0, we could not obtain multiple implementations of hash tables to be used in the same application, because once we fix `elem`, `key`, and the above three functions they cannot be changed.

Given the above the library provides a type `ht` of hash tables and means to create, insert, and search through a hash table.

```
/**************************************/
/* Hash table library side interface */
/**************************************/
struct ht;
typedef struct ht* ht;

ht ht_new(int m)
//@requires m > 0;
  ;
elem ht_search(ht H, key k);    /* O(1) avg. */
void ht_insert(ht H, elem e)    /* O(1) avg. */
//@requires e != NULL;
  ;
```

3 Generic Types

Since both keys and elements are defined by the clients, they turn into generic pointer types when we implement a truly generic structure in C.

We might try the following in a file `hashtable.c`, where we have added the function `ht_free` to the interface. As explained in the last lecture, we provide it with a function to apply to each element stored in the table.

```
#ifndef _HASHTABLE_H_
#define _HASHTABLE_H_

typedef void* ht_key;
typedef void* ht_elem;

/* Hash table interface */
typedef struct ht* ht;
ht ht_new (int init_size)
void ht_insert(ht H, ht_elem e);
ht_elem ht_search(ht H, ht_key k);
void ht_free(ht H, void (*elem_free)(ht_elem e));

#endif
```

We use type definitions instead of writing `void*` in this interface so the role of the arguments as keys or elements is made explicit (even if the compiler is blissfully unaware of this distinction).

However, this does not yet work. Before you read on, try to think about why not, and how we might solve it

4 Generic Operations via Function Pointers

The problem with the approach in the previous section is that the implementation of hashtables must call the functions `elem_key`, `key_equal`, and `key_hash`. Their types would now involve `void*` but in the environment in which the hash table implementation is compiled, there is can still only be one of each of these functions. This means the implementation cannot be truly generic.

Instead, we should pass pointers to these functions! But where do we pass them? We could pass all three to `ht_insert` and `ht_search`, where they are actually used. However, it is awkward to do this on every call. We notice that for a particular hash table, all three functions should be the same for all calls to insert into and search this table, because a single hash table stores elements of the same type and key. We can therefore pass these functions just once, when we first create the hash table, and store them with the table!

This gives us the following interface (in file `hashtable.h`):

```
#ifndef _HASHTABLE_H_
#define _HASHTABLE_H_

typedef void* ht_key;
typedef void* ht_elem;

/* Hash table interface */
typedef struct ht* ht;
ht ht_new (int init_size,
           ht_key (*elem_key)(ht_elem e),
           bool (*key_equal)(ht_key k1, ht_key k2),
           int (*key_hash)(ht_key k, int m));
void ht_insert(ht H, ht_elem e);
ht_elem ht_search(ht H, ht_key k);
void ht_free(ht H, void (*elem_free)(ht_elem e));

#endif
```

Storing the function for manipulating the data brings us closer to the realm of object-oriented programming where such functions are called *methods*, and the structure they are stored in are *objects*. We don't pursue this analogy further in this course, but you may see it in follow-up courses, specifically 15-214 *Software System Construction*.

5 Using Generic Hashtables

First, we see how the client code works with the above interface. We use here the example of word counts, which we also used to illustrate and test hash tables earlier. The structure contains a string and a count.

```
/* elements */
struct elem {
  char* word;                    /* key */
  int count;                     /* information */
};
typedef struct elem* elem;
```

As mentioned before, strings are represented as arrays of characters (type `char*`). The C function `strcmp` from library with header `string.h` compares strings. We then define:

```
bool key_equal(ht_key s1, ht_key s2) {
  return strcmp((char*)s1,(char*)s2) == 0;
}
```

Keep in mind that `ht_key` is defined to be `void*`. We therefore have to cast it to the appropriate type `char*` before we pass it to `strcmp`, which requires to strings as arguments. Similarly, when extracting a key from an element, we are given a pointer of type `void*` and have to cast it as of type `elem` (which is `struct elem*`).

```
/* extracting keys from elements */
ht_key elem_key(ht_elem e)
{ REQUIRES(e != NULL);
  return ((elem)e)->word;
}
```

The hash function is defined in a similar manner.

Here is an example where we insert strings created from integers (function `itoa`) into a hash table and then search for them.

```
int n = (1<<10);
ht H = ht_new(n/5, &elem_key, &key_equal, &key_hash);
for (int i = 0; i < n; i++) {
  elem e = xmalloc(sizeof(struct elem));
  e->word = itoa(i);
  e->count = i;
```

```
    ht_insert(H, e);
  }
  for (int i = 0; i < n; i++) {
    char* s = itoa(i);
    assert(((elem)ht_search(H, s))->count == i);
    free(s);
  }
```

Not the required cast when we receive an element from the table, while the arguments e and s do not need to be cast because the conversion from `t*` to `void*` is performed implicitly by the compiler.

6 Implementing Generic Hash Tables

The hash table structure, defined in file `hashtable.c` now needs to store the function pointers passed to it.

```
struct ht {
  int size;                     /* m */
  list* A;                      /* \length(A) == size */
  ht_key (*elem_key)(ht_elem e); /* extracting keys from elements */
  bool (*key_equal)(ht_key k1, ht_key k2); /* comparing keys */
  int (*key_hash)(ht_key k, int m);        /* hashing keys */
};

ht ht_new(int init_size,
          ht_key (*elem_key)(ht_elem e),
          bool (*key_equal)(ht_key k1, ht_key k2),
          int (*key_hash)(ht_key k, int m))
{ REQUIRES(init_size > 1);
  list* A = xcalloc(init_size, sizeof(list));
  ht H = xmalloc(sizeof(struct ht));
  H->size = init_size;
  H->A = A;                     /* all initialized to NULL; */
  H->elem_key = elem_key;
  H->key_equal = key_equal;
  H->key_hash = key_hash;
  ENSURES(is_ht(H));
  return H;
}
```

When we search for an element (and insertion is similar) we retrieve the functions from the hash table structure and call them. We exploit here that C allows function pointers to be directly applied to arguments, implicitly dereferencing the pointer.

```
/* ht_search(H, k) returns NULL if key k not present in H */
ht_elem ht_search(ht H, ht_key k)
{
  REQUIRES(is_ht(H));
  int h = H->key_hash(k, H->size);
  list l = H->A[h];
  while (l != NULL)
    //@loop_invariant is_chain(H, l, h);
    {
      if (H->key_equal(H->elem_key(l->data), k))
        return l->data;
      l = l->next;
    }
  return NULL;
}
```

This concludes this short discussion of generic implementations of libraries, exploiting `void*` and function pointers.

In more modern languages such ML, so-called *parametric polymorphism* can eliminate the need for checks when coercing from `void*`. The corresponding construct in object-oriented languages such as Java is usually called *generics*. We do not discuss these in this course.

7 Separate Compilation

Although the C language does not provide much support for modularity, convention helps. The convention rests on a distinction between *header files* (with extension `.h`) and *program files* (with extension `c`).

When we implement a data structure or other code, we provide not only `filename.c` with the code, but also a header file `filename.h` with declarations providing the interface for the code in `filename.c`. The implementation `filename.c` contains `#include "filename.h"` at its top, and client will have the same line. The fact that both implementation and client include the same header file provides a measure of consistency between the two.

Header files `filename.h` should never contain any function definitions (that is, code), only type definition, structure declarations, macros, and function declarations (so-called function prototypes). In contrast, program files `filename.c` can contain both declarations and definitions, with the understanding that the definitions are not available to other files.

We only ever `#include` header files, never program files, in order to maintain the separation between code and interface.

When `gcc` is invoked with multiple files, it behaves somewhat differently than `cc0`. It compiles each file *separately*, referring only to the included header files. Those come in two forms, `#include <syslib.h>` where `syslib` is a system library, and `#include "filename.h"`, where `filename.h` is provided in the local directory. Therefore, if the right header files are not included, the program file will not compiler correctly. We never pass a header file directly to `gcc`.

The compiler then produces a separate so-called *object file* `filename.o` for each `filename.c` that is compiled. All the object files and then *linked* together to create the executable. By default, that is `a.out`, but it can also be provided with the `-o executable` switch.

Let us summarize the most important conventions:

- Every file `filename`, except for the one with the `main` function, has a header file `filename.h` and a program file `filename.c`.
- The program `filename.c` and any client that would like to use it has a line `#include "filename.h"` at the beginning.
- The header file `filename.h` never contains any code, only macros, type definition, structure definitions, and functions header files. It has appropriate header guards to void problems if it is loaded more than once.
- We never `#include` any program files, only header files (with `.h` extension).
- We only pass program files (with `.c` extension) to `gcc` on the command line.

Exercises

Exercise 1 *Convert the interface and implementation for binary search trees from C0 to C and make them generic. Also convert the testing code, and verify that no memory is leaked in your tests. Make sure to adhere to the conventions described in Section 7.*