15-122: Principles of Imperative Computation — Spring 2024

Recitation 08: Generically Speaking — Thursday March 7th

Generic/Void Pointers

In C1, we introduce the concept of generic pointers, also known as **void pointers** (whose type is **void***). These are pointers that can have any underlying pointer type. We can create void pointers by casting other pointers to **void***:

```
int* int_pointer = alloc(int);
void* void_pointer = (void*)int_pointer;
```

We recover the original pointer by casting this void pointer back to an **int***:

```
//@assert \hastag(int*, void_pointer);
int* orig_int_pointer = (int*)void_pointer;
```

Note that we cannot cast a void pointer to a type other than that of the original pointer, which was **int*** here. We use the `\hastag` assertion to check the underlying type of a void pointer. `NULL` has all tags but no expression can have tag **void***.

We also cannot dereference or allocate void pointers: `*void_pointer` would cause a compilation error. To access the value that `void_pointer` points to, we need to first cast it back to an **int** pointer. We only cast *pointer* types to **void***! For example, you cannot cast an **int** to a **void*** — you would need to allocate an **int*** to be able to cast to **void***.

Function Pointers

In C1, we can also have pointers to functions. These **function pointers** hold the address of the function itself, and are associated with a function type. The type of a function is determined by the number and types of its inputs along with its return type. For example, the declarations:

```
typedef int one_input_fn(int a);         // Functions of this type take 1 input
typedef int two_input_fn(int a, int b);  // Functions of this type take 2 inputs
```

defines the **type** `one_input_fn` of functions that take a single argument of type **int** and return a result also of type **int**, and the type `two_input_fn` of functions that take two **int**s and return an **int**, respectively. We can then use these function types in our code:

```
one_input_fn* f = &fact;
one_input_fn* g = &fib;

int x = (*f)(5);     // This is calling fact(5), so x is now 120
int y = (*g)(5);     // This is calling fib(5), so y is now 8

two_input_fn* j = &pow;
int z = (*j)(2, 4);  // This is calling pow(2, 4), so z is now 16
```

But we can't assign `&pow` to a variable of type `one_input_fn*` because the number of arguments does not match. Same thing if trying to assign `&fact` to a variable of type `two_input_fn`:

```
one_input_fn* too_many = &pow;     // BAD: THIS WON'T TYPECHECK
two_input_fn* not_enough = &fact;  // BAD: THIS WON'T TYPECHECK
```

Checkpoint 0

Below is the interface for generic stacks in C1:

```
// typedef ______* stack_t;
typedef void* elem;          // the stack elements are generic pointers

int stack_size(stack_t S)                    void push(stack_t S, elem x)
/*@requires S != NULL; @*/;                  /*@requires S != NULL; @*/;
/*@ensures \result >= 0; @*/;                /*@ensures stack_size(S) > 0; @*/;

stack_t stack_new()                          elem pop(stack_t S)
/*@ensures \result != NULL; @*/              /*@requires S != NULL; @*/
/*@ensures stack_size(\result) == 0; @*/;    /*@requires stack_size(S) > 0; @*/;
```

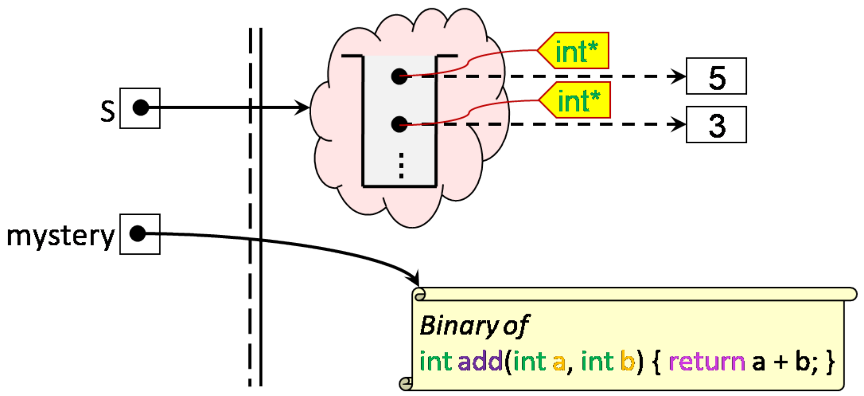
Note that these generic stacks can only contain pointers, as only pointers can be generic in C1.

Let's do some practice with some generic pointers and function pointers! Implement the function `apply_mystery_binary_fun`, which takes in a generic stack `S` and a binary function `mystery`, pops the topmost two elements, applies the mystery function, then pushes the result back onto the stack.[1]

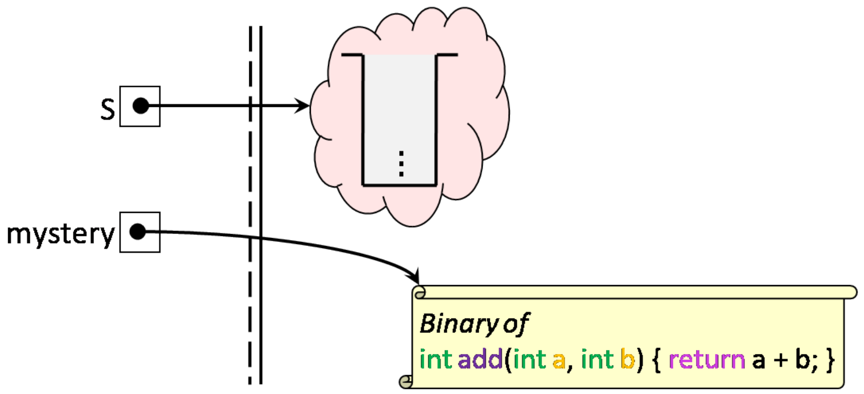
```
typedef int binop_fn(int a, int b);

void apply_mystery_binary_fun(stack_t S, binop_fn* mystery)
//@requires S != NULL && stack_size(S) >= 2;
//@requires mystery != NULL;
//@ensures stack_size(S) >= 1;
{
  void* generic_x = __________________________________________________________;
  //@assert \hastag(________________ , ________________) && _______________;
  int* x = ___________________________________________________________________;

  void* generic_y = __________________________________________________________;
  //@assert \hastag(________________ , ________________) && _______________;
  int* y = ___________________________________________________________________;

  ____________________________________________________________________________;
  ____________________________________________________________________________;
  push(S, ___________________________________________________________________);
}
```

[1]This may remind you of a previous programming assignment...

Assume we call `apply_mystery_binary_fun` with the following values for parameters `S` and `mystery`:



What does the memory diagram look like right before we exit the call to `apply_mystery_binary_fun`? Complete the following picture. (We omitted the variables `generic_x`, `x`, `generic_y` and `y` to avoid cluttering the diagram. You are welcome to do the same.)



Hash Dictionaries

A **hash dictionary** is a data structure that allows us to insert entries to be looked up later. Each **entry** in the dictionary is a data item that is associated with a unique **key**.

In lecture, we discussed how to use our newfound C1 capabilities to implement generic hash dictionaries, which can be used for any type of keys and entries. Let's give it a try below!

Farmer Lora is raising a horde of geese to participate in the local Honking Competition^{TM}. To help train her geese for the competition, she needs to keep track of the number of times each goose honks. Luckily for Lora, she can use a generic hash dictionary to help her do this!

Below is the interface for generic hash dictionaries in C1:

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

typedef void* entry;
typedef void* key;

typedef key entry_key_fn(entry x)           // Supplied by client
  /*@requires x != NULL; @*/ ;
typedef int key_hash_fn(key k);             // Supplied by client
typedef bool key_equiv_fn(key k1, key k2);  // Supplied by client

/**************************** Library interface ****************************/

// typedef ______* hdict_t;

hdict_t hdict_new(int capacity, entry_key_fn* entry_key,
                  key_hash_fn* hash, key_equiv_fn* equiv)
/*@requires capacity > 0; @*/
/*@requires entry_key != NULL && hash != NULL && equiv != NULL; @*/
/*@ensures \result != NULL; @*/ ;

entry hdict_lookup(hdict_t H, key k)    void hdict_insert(hdict_t H, entry x)
/*@requires H != NULL; @*/ ;            /*@requires H != NULL && x != NULL; @*/ ;
```

Farmer Lora's geese are represented by structs with the following definition.

```
typedef struct goose_header goose;
struct goose_header {
  string name;
  int num_honks;
};
```

Checkpoint 1

What should Farmer Lora use for the key and entry in her hash dictionary?

Key: ______________________________

Entry: ______________________________

What are their corresponding underlying types? (I.e., what tag would we assert that the keys and entries have?)

Key: ______________________________

Entry: ______________________________

Implement the client-side function `goose_key`, which takes in an entry and returns its key. Include sufficient contracts to ensure safety, both in your code and for the caller.

```
key goose_key(entry e)
//@requires ______________________________;
//@ensures ______________________________;
{
  ______________________________;
  ______________________________;
  return ______________________________;
}
```

Implement the client-side function `goose_equiv`, which takes in two keys and returns true if they are equivalent.

Checkpoint 2

Suppose you also have a `goose_hash` function of `type key_hash_fn`. Use the function `hdict_new` to create a new hash dictionary for storing Lora's geese, with an initial capacity of 122.

Checkpoint 3

Assume we have a hash dictionary `H` filled with geese. Implement the function `get_num_honks` which, given a dictionary and a goose name, returns the number of honks that goose has honked. If the goose doesn't exist, the function returns `-1`.

```
int get_num_honks(hdict_t H, string name)
//@requires H != NULL;
{
  ______________________________;
  ______________________________;
  ____________ lookup_result = hdict_lookup(____________________);
  if (lookup_result == NULL) return -1;
  //@assert \hastag(__________, lookup_result);
  ______________________________;
  return ______________________________;
}
```