Mappings

Mattias Heinl

1 In storage

Mappings are functions between two data-types. There is one key type, analogous to the domain, and a value type, analogous to range. A mapping has the following form, $mapping(key => value)m;$. The value type can be any type, while the key type is limited to *"elementary types, user defined value types, contract types or enums"* which for our purpose is primitives. For the purpose of accessing values in the map it follows the same structure as arrays, meaning that every key (index for arrays) actually maps to a value. This is called *dynamic allocation*, however the size of mappings may be for all intents and purposes infinite depending on the key type. For example, if the key type is addresses, where each address is 160 bits, then there are a total of 2^{160} keys. Thus we can't dynamically allocate in the same manner as we do with other types. Most notably however, we can only have mappings in storage, meaning that anything that contains or is a mapping can only exist in storage and never be copied to memory. Also note that since we are only concerned with storage there is no aliasing that needs to be considered only reading and writing which is equivalent to copying.

Note that technically anything could be in storage before function call, but we have to assume defaults unless otherwise stated in pre-condition for function.

2 Mapping as heap

The proposed solution to handling mappings is to treat them the same as heaps. Each map is built like a list (in the functional programming sense). There are simplifications that we can make to the heap/list, such needing fewer constructors and functions for access. From now on to describe the heap/list for mappings the word *meap* is used. There exists two constructors, the empty meap *mtm* and *store* which has the following type declaration $store :: meap \times any_2 \times any_3 \rightarrow meap$, the same as with heap where the second parameter (any_2) is the key and the third parameter (any_3) is the stored value. Similarly there is select which is equivalent to the heap version, except the additions made with defaults which are handled differently in storage. Note however since a meap may have a nested meap in the value there may be some changes needed for it to be proper or some other way to handle having a meap be returned.

(Note that perhaps we could want a un-allocate constructor for some special cases, then we would have to make some changes to defaults as well. This would correspond to anon in heap but unsure if needed.)

3 Defaults

To implement defaults, use different approach from memory. Since we now need dynamic allocation for more than primitives, in fact we need to be able to dynamically allocate anything in storage. We generate actual default values and link them before starting the execution of the proof.

Say that we have a contract with two types of structs, A and B, and a map in storage on the following form:

$struct\ A$ {
$uint\ i$;
}

$struct\ B$ {
$A\ a$;
}

$mapping(k => v)\ m$;

This means that we first generate a $default\_A$ where $default\_A.i == default\_uint$, $default\_B$ where $default\_B.a = default\_A$, and $default\_map$ where $default\_m == mtm$ (any mapping is by default the mtm constructor for meaps). In turn values like $default\_uint$ (and others like bool) are predefined like in memory. These are actual values that exist in storage and can be read, however we never change these values, only read and copy them (we should never update them).

Next we add rules equivalent to the select on empty in memory/heap for the read. The $read_T$ of an empty storage returns the corresponding $default\_T$. For example if we define a variable $A\ myA$; in the contract then access that struct in a function then we get $default\_A$. Note that this doesn't cover access to meaps.

For select on meaps, we add corresponding rules to those of memory/heap on empty, meaning that for meaps, $T :: select(mtm, o) = default\_T$. This covers every kind of dynamic allocation for mappings.

3.1 Considerations

In order to ensure correctness we would need to consider additional things. Such as comparison, "==" does not exist for structs/non-primitives so this is not an issue. As long as reads from storage are copies, nothing should be broken when it comes to changes/updates on storage.

3.1.1 Assignment

Say that we have a contract with the following types.

```
struct A {
Coordinates coordinates;
...
}

struct Coordinates {
int x;
int y;
...
}

mapping(uint => A) m;
```

Using a meap to model m, we want to ensure that assignments are handled properly. Say that in a function we have "$m[i].coordinates.x = 1$" and before that $m[i].coordinates == default\_Coordinates$. We never create a $coordinates_i$ value, and we don't change the default, so how can we be sure that we access the correct x in the future? In essence, in storage, there is no explicit struct *coordinates* only the correct structure with fields that corresponds to $coordinates_i$. We need to store $coordinates_i.x == 1$ somewhere in the meap which should be done with store() for meaps. But if we in the future decide to first copy $coordinates_i$, and then read $coordinates_i.x$ how can we get 1 unless we change $coordinates_i$ which is supposed to be $default\_Coordinates$? Either we need some special rules for reading $coordinates_i$ or we have to update $coordinates_i$ which means it can't be the default.

There are two possible/proposed solutions, either reading from map is only concerned with copying fields in a manner that ignores non-existent structs, or assignment (e.g. "$m[i].coordinates.x = 1$") is handled in a different manner than expected. Propose that option two is better/simpler.

$$\text{assignInMapExample}\ \frac{}{\langle "m[i].coordinates.x = n\rangle \rightsquigarrow \langle T\ t = m[i];\ t.coordinates.x = n;\ m[i] = t\rangle}$$

Where t is in memory. In essence on rule is needed for "objects" (read, structs, arrays, mappings) and primitives. I.e. if you assign primitive then we first create a corresponding object (by reading from map), update it, and then assign that new object, and if you assign na object then that is handled in the expected way (by copying from heap to meap).

How to handle nested mappings? For example

$$mapping(uint8 => mapping(uint8 => uint8))\ m;$$

is a valid map in solidity. Remember that default for meap/map is mtm. Propose that it should look something like this.

$m[i][j] = x;$
$\rightsquigarrow$
$T_{map} j0 = m[i];$

$m[i] = j0;$

in turn need rule for changing which map is pointed at.

$j0[j] = x;$
$\rightsquigarrow$
$\{newJ = store(j0, j, x)\}$
$\{m[i] = newJ\};$

Alternatively there needs to be a better way to assign something of type map which would mean that, at least temporarily, mappings can exist in memory/heap.