Storage and memory

Mattias Heinl

1 Heap/memory

The memory model for Solidity is in large part equivalent to the memory model of Java. Thus the changes required to the heap are minimal. Firstly, Solidity lacks the *null* value, thus any rules in reference to it are obsolete. We can assume that nothing is null. Secondly, dynamic allocation exists for all possible values not only some primitive cases. For primitive data types the select of heap for these are expanded to include defaults and more complex data types are covered in section 3. Lastly, the memory does not allow for all data types of Solidity to be stored, namely those including mappings. This however is not expressed with explicit rules but in stead with the lack of coverage for these types. The memory can only contain primitives, structs, and arrays, thus simplifying the possible valid pairs $o,\ f$.

The structure of the memory (in the rules referred to as *heap* for continuity) is maintained except for the definition of wellformedness and allocation (see section 2). The new definition of wellformedness is as follows.

onlyCreatedObjectsAreReferenced
$wellformed(h) \to select_{boolean}(h, selectA(h,o,f), created) = TRUE$

onlyCreatedObjectsAreInLocSets
$wellformed(h) \wedge \varepsilon(o2, f2, select_{LocSet}(h,o,f)) \to select_{boolen}(h, o2, created) = TRUE$

narrowSelectType, where type of f is A and $A \sqsubseteq B$
$wellFormed(h) \wedge select_B(h,o,f) \to select_A(h,o,f)$

narrowSelectArrayType, where type of o is $A[]$ and $A \sqsubseteq B$
$wellFormed(h) \wedge select_B(h,o,arr(i)) \to select_A(h,o,arr(i))$

wellFormedStoreObject, where type of f is A
$wellFormed(h) \wedge ((select_{boolean}(h,x,created) = TRUE \wedge instanceA(x) = TRUE)) \to wellFormed(store(h,o,f,x))$

wellFormedStoreArray
$wellFormed(h) \wedge ((select_{boolean}(h, x, created) = TRUE \wedge arrayStoreValid(o, x))) \rightarrow wellFormed(store(h, o, arr(idx), x)))$

wellFormedStoreLocSet, where type of f is A and $LocSet \sqsubseteq A$
$wellFormed(h) \wedge \forall ov; \forall fv; (\varepsilon(ov, fv, y) \rightarrow select_{boolean}(h, ov, created) = TRUE) \rightarrow wellFormed(store(h, o, f, y))$

wellFormedStorePrimitive, where f is a field of type A, x is of type B, and $B \sqsubseteq A$, $B \not\sqsubseteq Object$, $B \not\sqsubseteq LocSet$
$wellFormed(h) \rightarrow wellFormed(store(h, o, f, x))$

wellFormedStorePrimitiveArray, where o is of sort A, x is of sort B, $B \sqsubseteq A$, $B \not\sqsubseteq Object$, $B \not\sqsubseteq LocSet$
$wellFormed(h) \rightarrow wellFormed(store(h, o, arr(idx), x))$

wellFormedCreate
$wellFormed(h) \rightarrow wellFormed(create(h, o))$

wellFormedAnon
$wellFormed(h) \wedge wellFormed(h2) \rightarrow wellFormed(anon(h, y, h2))$

2 Allocation

For clarity and continuity the keyword *"new"* is kept even in places where not applicable in true Solidity. Allocation is handled in the same manner with the exception to changes of the following rule. The original rules can be found on starting at page 89 (chapter 3.6.6.3).

Start by noting that since the same type of objects do not exists we can exclude everything except $<$ *allocate* $>$ which is the only part needed by the original $<$ *createObject* $>$, and that we assume that there exists a function $allFields()$ that works as a quantifier of said type in order of the constructor. I.e. for structs the declared fields and for arrays the numbered order (ignoring the need for *arr()*). Also note that arrays are converted to implicit structs, and since the only allowed parameter is the length field that is the only parameter. A more in depth definition can be given at a later date with the main ideas summarised in section 3.2. The new versions are as follows.

$$\text{instanceCreation}\ \frac{\Rightarrow \langle T\ v0 = T. < allocation > ();\ Ti\ ai = ei;\ v = v0;\ v.allFields() = ai;}{\Rightarrow \langle \pi\ v = newT(e1, ..., en); \omega \rangle}$$

Note that for dynamic allocation we can ignore the lines $Ti...$ and $v.allFields....$ The method *allocate* is in turn handled by the following rule.

$$\text{allocateInstance}\ \frac{\begin{array}{l} exactInstance(o') = TRUE,\ (wellFormed(heap) \\ \rightarrow selectboolean(heap, o', created) = FALSE), \\ \{heap := create(heap, o')\},\ \{lhs := o'\}, \langle \pi\omega \rangle \phi \end{array}}{\Rightarrow \langle \pi\ lhs = T. < allocate >; \omega \rangle \phi}$$

where $o' : T \in FSym$

The implication for how declarations of variables are to be handled is partly covered in section 4. However, since Solidity does not include objects (from Java) in the same manner (namely external classes included in the same manner), these changes should not affect the existing work done on variables and storing them in memory/heap.

Note that since the only two types of objects (in memory) are structs and arrays, there is no need for more complex allocation rules. However, this rule only covers completely structs, or any type with nested object fields, where each of the nested object fields is already allocated in the parameters. Note that this does not break any of the rules previously stated, and that there is no way of circumventing having to first allocate the object and then assigning the fields. See section 3 (and comment on *instanceCreation*) for how the parameterless constructor should be handled, i.e. when an object with nested object fields is created without supplying parameter. In short, these two rules cover allocation when all fields are assigned by the constructor/parameters, the empty constructor case requires the rules covered in section 3.

3 Dynamic allocation

For primitive types, such as *"uint"* and *"bool"* the dynamic allocation is handled with the select rules later on in this section.

We start with only dynamic allocation for structs, with arrays later on in the section. For dynamic allocation we should have the following equivalence "$T\ s; \leftrightarrow T\ s = new\ T()$", however "$new\ T()$" is not valid Solidity syntax but instead shorthand to make use of existing Java work done. We can't make use of the exact same rule *instanceCreation*, now the assignment of parameters does not happen, instead it is equivalent to only the ¡allocation¿ call. The following rule is suggested.

Note the inclusion of quantification over fields r_i for type T (variable *se*), this is not exactly possible with taclets (but implementable with the proposed *allFields()* function). It is however a logical shorthand for readability. The implementation would be several rules.

$$\text{newStruct}\ \frac{\Rightarrow \forall r_i.\ isRef(r_i)\langle \pi\ T_{r_i0}\ r_i0; ...;\ T\ se = new\ T();\ se.r_i = r_i0; ...; \omega\rangle\phi}{\Rightarrow \langle \pi\ T\ se; \omega\rangle\phi}$$

Note that r_i is the field and r_i0 is the equivalent dynamically allocated value for said field, and that isRef() is technically just isStruct or isArray. Since the line $typeof(r_i0)\ r_i0;$ is in turn translated to $T'\ r_i0;$, which again in turn is handled by the same *newStruct* rule, this is a recursive bottom up approach. Thus at the end of application of *newStruct* rule applications we have a new variable *se* with all the object fields (equivalent to isRef() fields) assigned a new valid value. What remains is the special considerations for primitives and arrays.

3.1 Primitives

Dynamic allocation of primitives, such as *uint* and *bool*, is not handled with assignment of values but instead with additions to selects on empty memory.

$$\text{primitiveDefault}\ \frac{default(T)}{\Rightarrow select :: T(o, f, mtm)}$$

So for example $default(uint) = 0$ and $default(bool) = false$, these are their own rules.

3.2 Arrays

There are two types of arrays that have slight differences in allocation. Since arrays have the length given by the type, there are additional considerations.

Next, the allFields() for arrays is the defined in the following manner. Let $T[n]$ be the type where $n > 0$, then $allFields() = [arr(0), ..., arr(n)]$. Let $T[]$ be the type, then $allFields() = \emptyset$. Since the quantification over r_i in *newStruct* is going to make use of *allFields()*, then the only addition needed are the two following rules.

$$\text{staticNew}\ \frac{\Rightarrow \langle \pi\ T[n]se; se.length = n; \omega\rangle\phi}{\Rightarrow \langle \pi\ T[n]se; \omega\rangle\phi}$$

$$\text{dynamicNew}\ \frac{\Rightarrow \langle \pi\ T[]se; se.length = 0; \omega\rangle\phi}{\Rightarrow \langle \pi\ T[]se; \omega\rangle\phi}$$

Technically the second rule (*dynamicNew*) is not necessary since the default select should handle this. Also not that arrays have an additional type of constructor which is right now not handled, namely the case $T[]\ a = new\ T[](n);$. Also note that right now *staticNew/dynamicNew* are recursively applicable.

As we can see, with the addition of the function *allFields()* defined in the manner above, we can treat arrays the same a structs with some minor additions (handling the length).

4 Copying

Start by noting that assigning from storage to memory means that the value is a deep copy, meaning that there is no aliasing between them. Since the code that is to be handled here is $T\ mv = sv;$, where any variable starting with m is memory and any variable starting with s is storage, we can break it down into two parts. Firstly, the memory variable must be allocated as a new reference object if applicable, and secondly each part of the storage variable must be copied.

Note that rules are now on the following form, where an instance of *Pre* can be replaced with *Post* given that *Condition* is true.

$$\text{example}\ \frac{Condition}{Pre \rightsquigarrow Post}$$

Start by breaking $T\ mv = sv;$ into separate parts, an allocation and assignment. Note that if mv is not a struct then the allocation has no effect.

$$\text{copyBreakdown}\ \frac{}{\langle T\ mv = sv; \rangle \rightsquigarrow \langle T\ mv;\ mv = sv; \rangle}$$

Since dynamic allocation is already described, we divide the assignment into two cases, either T is a primitive (such as *uint*) or it has fields (i.e. *isRef()* meaning it is a struct of array). Starting with the primitives.

$$\text{primitiveCopy}\ \frac{isPrimitive(mv)}{\langle mv = sv; \rangle \rightsquigarrow \{mv := sv\}\langle\rangle}$$

No special considerations for primitives, sv is in turn translated to a read from storage.

$$\text{referenceCopy}\ \frac{\begin{array}{l} isRef(mv). \\ \forall mf_i \in mv.allFields().isRef(mv.mf_i). \\ \forall mp_i \in mv.allFields().isPrimitive(mv.mp_i). \end{array}}{\langle mv = sv; \rangle \rightsquigarrow \langle T_{mf_i}\ mf_i0 = sv.mf_i;\ mv.mf_i = mf_i0;\ mv.mp_i = sv.mp_i; \rangle}$$

Here mf_i is the field name for type T which is a reference, mf_i0 is the temporary variable, and mp_i are the primitive fields. Note that the first line in the Post is handled recursively by rule copyBreakdown, meaning that we have a bottom up approach.

Note that this rule again assumes that arrays are reduced to structs with allFields() to avoid using $arr()$ to convert to fields. Also note that the implementation of this rule wouldn't have to be as divided in terms of primitive and references, the exact order of them is irrelevant. As long as the references are recursively copied before assignment the approach is bottom up.