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 of SolidiKeY from the base KeY version are minimal. However, Solidity includes a storage model as well which is unlike anything that exists in Java. The memory model represents every variable declared within a function, the information is only saved in that context and is not persistent. The storage model is the equivalent to instance variables in Java, they are declared outside of any function, are persistent between function calls, and do not make use of references/addresses/object identifiers in the same manner as the memory model. This is to say that there is not aliasing in the storage model, every value which is assigned or read is treated as a copy.

Solidity has differences with regards to data types. For the purpose of this text, there exists two sorts of data types, primitive and references. What may be known as primitives in other languages is called *Value Types* in Solidity, such as booleans, numbers (Integers uint/int or Rational and Integer Literals), address (equivalent to Ethereum address), and more. The second sort of types are called *Reference Types* these consists in essence of structs, arrays, and mappings. For each of these types there are additional variation, such as static versus dynamic arrays in section 3.2, that for the purpose of this text are irrelevant. For the sake of clarity, the words **primitives** and **references** are used to describe these sorts to avoid confusion.

In correct Solidity, if a reference variable is declared within a function the "location" must be declared, meaning whether it is in the memory or storage model, outside a function the variable is always in the storage model. A reference in the memory model is the equivalent to a reference in Java or an address in C (without any of the valid operations on addresses), while in the storage model it is more so equivalent to a path to a location in the storage model. For the purpose of this text variables have the additional keyword information excluded from any example, but in the example below, the variable ExI is correct Solidity while MI is SolidiKeY.

While Solidity does not differentiate between the types of based on location, SolidiKeY does. For example:

```
    contract C {

    struct Integer {
            int i;
    }

    Integer SI;

    function f(){
            Integer MI;
            Integer memory ExI;
    }
}
```

While in Solidity the variables SI and MI have the same type, namely Integer, Solidity differentiates them, meaning that they should have the types $Integer^S$ for SI and $Integer^M$ for MI.

The changes made to SolidiKeY with regards to Solidity's memory/storage model are as follows. Firstly, Solidity lacks the *null* value, thus any rules with regards to that value is obsolete. In other words we can assume that every variable/field has a value and is never null. Secondly there exists something referred to as *dynamic allocation*, which means that every variable/field no matter the data type and whether any assignment has been made is to begin with a default value for said type, see section 3 for the full explanation. Lastly, the memory model 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. For the storage model this restriction does not exists. The memory model can only contain primitives (for examples bool and uint), structs, and arrays, thus simplifying the possible valid pairs o and f.

The structure of the memory model (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.

(changes pending discussion with Richard)

1.1 Wellformed rules

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

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

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

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

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

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

In order for a variable in the memory model to be correctly uniquely identifiable, space/symbol must be allocated for it. Based the original rules in the KeY book page 89 (chapter 3.6.6.3) the following are changes made to the rules. Note that only references require this sort of allocation.

For clarity and continuity the keyword *"new"* is kept even in places where not

applicable in real Solidity. For instance, in the example below the function SolidityTyntax is correct while NotSolidityTyntax is incorrect with regards to Solidity. For the purpose of SolidiKeY NotSolidityTyntax is also valid.

```
contract C {

        struct Coords{
        int x;
        int y;
        }

        function SolidityTyntax(){
                Coords pos = Coords(1,2)
        }

        function NotSolidityTyntax(){
                Coords pos = new Coords(1,2)
        }
}
```

Start by noting that since the same type of objects (in a Java sense, here a reference) 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() which would be equivalent to fieldOf()). 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.

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

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 the memory model.

Note that since the only two types of references in the memory model are structs and arrays, there is no need for more complex allocation rules. However, this rule only covers completely references where each of the parameter expressions ei which are in turn references have already been allocated before the assignment.

Note that this does not break any of the rules previously stated in section 1.1, and that there is no way of circumventing having to first allocate the reference and then assigning the fields of said reference.

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

In Solidity, since there can be no null for any variable, implicitly every variable has a corresponding value no matter whether it is primitive or a reference. In the example below, the assignment has to have a pre and post state with values for each variable.

```
contract C {

        struct A{
                int x;
                bool y;
        }

        function f(){
                Coords pos;
                uint q;
                q = pos.x;
        }

}
```

At the point of assignment the value of q is 0, pos.x is 0, and pos.y is false. This is in spite of there being no assignment previously of any variable or field.

For primitive types, such as *"uint"* and *"bool"* the dynamic allocation is handled with the select rules later on in section 3.1

We start with only dynamic allocation for structs, with arrays later on in the section. For dynamic allocation we should have the following equivalence "$T\ s; \leftrightarrow T^M\ s = new\ T^M()$", however as previously established "$new\ T^M()$" is not valid Solidity 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. Thus the rule for the empty parameter constructor is as follows.

$$\text{emptyConstructor}\ \frac{\Rightarrow \langle T^M\ v0 = T^M.<allocation>();\ v = v0;\ \rangle}{\Rightarrow \langle \pi\ T^M\ v = new\ T^M(); \omega \rangle}$$

With this we can construct the rule needed to dynamically allocate references. Note the inclusion of quantification over fields r_i for type T^M (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.

$$\text{newReference}\ \frac{\Rightarrow \forall r_i.\ isRef(r_i) \langle \pi\ T^M_{r_{i0}}\ r_{i0}; ...;\ T^M\ se = new\ T^M();\ se.r_i = r_{i0}; ...; \omega \rangle}{\Rightarrow \langle \pi\ T^M\ se; \omega \rangle}$$

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 in this context. Since the lines $T^M_{r_{i0}}\ r_{i0};$ are in turn translated to as a dynamic allocations where the rule newReference is applicable this is a recursive bottom up approach.

In plain words, for a reference (se), each field which is in turn a reference becomes a dynamically allocated variable, then se is allocated, and then the corresponding fields and variable is assigned.

Thus at the end of application of newReference rule applications we have a new variable se with all the reference 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 heap in the memory model.

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

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

3.2 Arrays

Arrays in SolidiKeY can be thought of as structs with integers as fields and the field length. The thing that distinguishes between the types of arrays are static and dynamic. Static arrays have a length set at declaration while dynamic arrays have a length that may change.

These two kinds of arrays are such that anything that holds for structs also holds for arrays. However arrays themselves have additional constraints that require additional considerations.

To start with, the allFields() for arrays is the defined in the following manner. Note that the function arr() is SolidiKeY specific and would better be named fieldOf(), it takes an integer and turns it into a field. 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 newReference 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

When assigning values between the two different storage/memory models, every pair is different. For the purpose of this text, the interaction from storage model to memory model is explored.

Start by noting that assigning from storage model to memory model 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 $T^M\ mv = sv;$, 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 model variable must be allocated as a new reference if applicable, and secondly each part of the storage model 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 $T^M\ mv = sv;$ into separate parts, an allocation and assignment. Note that if mv is not a reference then the allocation has no effect.

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

Since dynamic allocation is already described, we divide the assignment into two cases, either T^M is a primitive (such as *uint*) or it is a reference meaning that it has fields (in this case either struct or 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 the storage model.

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

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 recursive bottom up approach.

In plain words, when assigning a variable in memory model with a value from a variable in the storage model, either it is the initial declaration ($T^M\ mv = sv;$) or just the assignment where the both have previously been allocated ($mv = sv$). First we ensure that the memory model variable is allocated (with dynamic allocation) and only then to we assign values. When assigning the values (of fields) we first recursively allocated and assign the fields and then the primitive values. Since the fields are on the same for $T^M\ mv = sv;$ we have a bottom up approach similar to the rule newReference.

Note that this rule again assumes that arrays are reduced to structs with allFields() to avoid using arr() (or fieldOf()) to convert integers 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.