SMT-Friendly formalization of the solidity memory model

Akos Hajdu and Dejan Jovanovi´c

Omer and Yoav
Agenda

- **Motivation**
- **Background**
- **Formalization**
  - Types
  - Local storage pointers
  - State variables, function, memory and defval
  - Assignments
  - Expressions
  - Statements
- **Summary**
Solidity

- Object Oriented
- Runs on the EVM
- By now, you are probably familiar with it
Smart Contracts

- Deployed on the ethereum network
- EVM bytecode
- Typically written in a high level language (e.g. Solidity)
- Cannot be modified
- Communication via transactions
- Two kinds of memory locations
- Don’t support null pointers
The problem

- Contracts are prone to errors
- Errors can lead to devastating losses
- DAO, Bitrue, Deus and many more
- We want to use formal verification
Our end goal

- Convert solidity to an smt based program (Boogie, why3 etc.)
- Convert solidity programs to smt-based syntax
Agenda

- Motivation
- Background
- Formalization
  - Types
  - Local storage pointers
  - State variables, function, memory and defval
  - Assignments
  - Expressions
  - Statements
- Summary
Contract storage

- Persistent
- Stored on the blockchain
- Array of up to $2^{256}$ slots
  - Each slot is 32 bytes
  - Most data is allocated on a fixed number of slots starting from 0
  - Fixed size arrays
  - Dynamic arrays and mappings are implemented as a hash table
Contract memory

- Accessible only on executions
- Deleted after each transaction
- Stores function arguments and return values
- Heap-like
Reference vs value types

class DataStorage {
  struct Record {
    bool set;
    int [] data;
  }
  mapping ( address =>Record) private records;
  function append( address at , int d ) public {
    Record storage r = records[at];
    r.set = true ;
    r.data. push (d);
  }
  function isset(Record storage r ) internal view returns ( bool s ) {
    s = r.set;
  }
  function get( address at) public view returns ( int [] memory ret) {
    require (isset(records[at]));
    ret = records[at].data;
  }
}
Agenda

- Motivation
- Background
- Formalization
  - Types
  - Local storage pointers
  - State variables, function, memory and defval
  - Assignments
  - Expressions
  - Statements
- Summary
The $T$ function

- A mapping function from Solidity types to SMT types
- Ignores side effects
- Assumes each declaration has a unique name
- Assumes data location of reference type is a part of the type

```
TypeName ::= address | int | uint | bool
    | mapping(TypeName => TypeName)
    | TypeName[] | TypeName[n]
    | StructName
```

```
TypeName ::= int | bool                Integer, Boolean
    | [TypeName]typeName             SMT array
    | DataTypeName                   SMT datatype
DataTypeDef ::= DataTypeName((id : TypeName)*) Datatype definition
```
Value types

\[ T(\text{bool}) \doteq \text{bool} \]
\[ T(\text{address}) \doteq T(\text{int}) \doteq T(\text{uint}) \doteq \text{int} \]

\[ T(\text{mapping}(K=\rightarrow V) \text{ storage}) \doteq [T(K)]T(V) \]
\[ T(\text{mapping}(K=\rightarrow V) \text{ storptr}) \doteq [\text{int}]\text{int} \]

\[ T(T[n] \text{ storage}) \doteq T(T[] \text{ storage}) \]
\[ T(T[n] \text{ storptr}) \doteq T(T[] \text{ storptr}) \]
\[ T(T[n] \text{ memory}) \doteq T(T[] \text{ memory}) \]

\[ T(T[] \text{ storage}) \doteq \text{StorArr}_T \text{ with } [\text{StorArr}_T(\text{arr} : [\text{int}]T(T), \text{length} : \text{int})] \]
\[ T(T[] \text{ storptr}) \doteq [\text{int}]\text{int} \]
\[ T(T[] \text{ memory}) \doteq \text{int} \text{ with } [\text{MemArr}_T(\text{arr} : [\text{int}]T(T), \text{length} : \text{int})] \]
\[ \text{arrheap}_T : [\text{int}]\text{MemArr}_T \]

\[ T(\text{struct } S \text{ storage}) \doteq \text{StorStruct}_S \text{ with } [\text{StorStruct}_S(\ldots, m_i : T(S_i), \ldots)] \]
\[ T(\text{struct } S \text{ storptr}) \doteq [\text{int}]\text{int} \]
\[ T(\text{struct } S \text{ memory}) \doteq \text{int} \text{ with } [\text{MemStruct}_S(\ldots, m_i : T(S_i), \ldots)] \]
\[ \text{structheap}_S : [\text{int}]\text{MemStruct}_S \]
Mappings

\[ T(\text{bool}) \triangleq \text{bool} \]
\[ T(\text{address}) \triangleq T(\text{int}) \triangleq T(\text{uint}) \triangleq \text{int} \]

\[ T(\text{mapping(K=\rightarrow V) storage}) \triangleq [T(K)]T(V) \]
\[ T(\text{mapping(K=\rightarrow V) storptr}) \triangleq [\text{int}]\text{int} \]

\[ T(T[n] \text{ storage}) \triangleq T(T[]) \text{ storage} \]
\[ T(T[n] \text{ storptr}) \triangleq T(T[]) \text{ storptr} \]
\[ T(T[n] \text{ memory}) \triangleq T(T[]) \text{ memory} \]

\[ T(T[] \text{ storage}) \triangleq \text{StorArr}_T \text{ with } [\text{StorArr}_T(\text{arr : } \text{int}T(T), \text{length : } \text{int})] \]
\[ T(T[] \text{ storptr}) \triangleq [\text{int}]\text{int} \]
\[ T(T[] \text{ memory}) \triangleq \text{int} \text{ with } [\text{MemArr}_T(\text{arr : } \text{int}T(T), \text{length : } \text{int})] \]
\[ \quad [\text{arrheap}_T : [\text{int}]\text{MemArr}_T] \]

\[ T(\text{struct } S \text{ storage}) \triangleq \text{StorStruct}_S \text{ with } [\text{StorStruct}_S(\ldots, m_i : T(S_i), \ldots)] \]
\[ T(\text{struct } S \text{ storptr}) \triangleq [\text{int}]\text{int} \]
\[ T(\text{struct } S \text{ memory}) \triangleq \text{int} \text{ with } [\text{MemStruct}_S(\ldots, m_i : T(S_i), \ldots)] \]
\[ \quad [\text{structheap}_S : [\text{int}]\text{MemStruct}_S] \]
Arrays

\[ T(\text{bool}) \equiv \text{bool} \]
\[ T(\text{address}) \equiv T(\text{int}) \equiv T(\text{uint}) \equiv \text{int} \]
\[ T(\text{mapping}(K\Rightarrow V) \text{ storage}) \equiv [T(K)]T(V) \]
\[ T(\text{mapping}(K\Rightarrow V) \text{ storptr}) \equiv [\text{int}]\text{int} \]
\[ T(T[n] \text{ storage}) \equiv T(T[] \text{ storage}) \]
\[ T(T[n] \text{ storptr}) \equiv T(T[] \text{ storptr}) \]
\[ T(T[n] \text{ memory}) \equiv T(T[] \text{ memory}) \]
\[ T(T[] \text{ storage}) \equiv \text{StorArr}_T \text{ with } [\text{StorArr}_T(\text{arr} : [\text{int}]T(T), \text{length} : \text{int})] \]
\[ T(T[] \text{ storptr}) \equiv [\text{int}]\text{int} \]
\[ T(T[] \text{ memory}) \equiv \text{int} \text{ with } [\text{MemArr}_T(\text{arr} : [\text{int}]T(T), \text{length} : \text{int})] \]
\[ \quad \quad \quad \quad \quad \quad \text{[arrheap}_T : [\text{int}]\text{MemArr}_T] \]
\[ T(\text{struct } S \text{ storage}) \equiv \text{StorStruct}_S \text{ with } [\text{StorStruct}_S(\ldots, m_i : T(S_i), \ldots)] \]
\[ T(\text{struct } S \text{ storptr}) \equiv [\text{int}]\text{int} \]
\[ T(\text{struct } S \text{ memory}) \equiv \text{int} \text{ with } [\text{MemStruct}_S(\ldots, m_i : T(S_i), \ldots)] \]
\[ \quad \quad \quad \quad \quad \quad \text{[structheap}_S : [\text{int}]\text{MemStruct}_S] \]
Structs

\[ T(\text{bool}) \doteq \text{bool} \]
\[ T(\text{address}) \doteq T(\text{int}) \doteq T(\text{uint}) \doteq \text{int} \]
\[ T(\text{mapping}(K\rightarrow V) \text{ storage}) \doteq [T(K)]T(V) \]
\[ T(\text{mapping}(K\rightarrow V) \text{ storptr}) \doteq [\text{int}]\text{int} \]
\[ T(T[n] \text{ storage}) \doteq T(T[] \text{ storage}) \]
\[ T(T[n] \text{ storptr}) \doteq T(T[] \text{ storptr}) \]
\[ T(T[n] \text{ memory}) \doteq T(T[] \text{ memory}) \]
\[ T(T[] \text{ storage}) \doteq \text{StorArr}_T \text{ with } [\text{StorArr}_T(arr : [\text{int}]T(T), length : \text{int})] \]
\[ T(T[] \text{ storptr}) \doteq [\text{int}]\text{int} \]
\[ T(T[] \text{ memory}) \doteq \text{int} \text{ with } [\text{MemArr}_T(arr : [\text{int}]T(T), length : \text{int})] \]
\[ \text{arrheap}_T : [\text{int}]\text{MemArr}_T \]

\[ T(\text{struct } S \text{ storage}) \doteq \text{StorStruct}_S \text{ with } [\text{StorStruct}_S(\ldots, m_i : T(S_i), \ldots)] \]
\[ T(\text{struct } S \text{ storptr}) \doteq [\text{int}]\text{int} \]
\[ T(\text{struct } S \text{ memory}) \doteq \text{int} \text{ with } [\text{MemStruct}_S(\ldots, m_i : T(S_i), \ldots)] \]
\[ \text{structheap}_S : [\text{int}]\text{MemStruct}_S \]
Agenda

- Motivation
- Background
- **Formalization**
  - Types
  - **Local storage pointers**
  - State variables, function, memory and defval
  - Assignments
  - Expressions
  - Statements
- **Summary**
Local storage pointers

- Pointers to storage that are used in a local context
- Function parameters or local variables that reference storage
- We denote it as storptr
Local storage pointers

- Question: How can we encode local storage pointers with SMT?
- Partial solution: Substitute each occurrence of the local pointer with the expression that is assigned to
- Downsides: storage pointer can be reassigned, received as a function argument and more.

```
T storage t1 = sa[8].ta[5];
```
tree(.) function

- Given a contract and a type $T$, returns a tree of its variables that includes:
  - Storage variables
  - Variables that lead to a sub variable of type $T$
Local storage pointers - solution

- Local storage pointer’s SMT type is always `[int]int`
- The array will be the finite path from the tree of values of the contract

```
contract C {
    struct T{ int z; }
    struct S{ int x; T t; T[] ts; }
    T t1;
    S s1;
    S[] ss;
    function f() public view{
        T storage a = ss[5].ts[8];
    }
}
```

\[ T(a) = [\text{int}]\text{int} \]

\[ a \rightarrow [2,5,1,8] \]
Usage

- We got a representation of storage pointers, but how do we use it?
- On initialization, we use the `pack` function
- On dereferencing we use the `unpack` function
Pack function

- Given an expression, pack(.) uses the storage tree
- Encodes the expression to an array
- Fits the expression into the tree
def pack(expr):
    baseExprs := list of base sub-expressions of expr;
    baseExpr := car(baseExprs);
    if baseExpr is a state variable then
        return packpath(tree(type(expr)), baseExprs, 0, constarr[int][int](0))
    if baseExpr is a storage pointer then
        result := constarr[int][int](0);
        prefix := E(baseExpr);
        foreach path to a leaf in tree(type(baseExpr)) do
            pathResult, pathCond := prefix, true;
            foreach kth edge on the path with label id (i) do
                pathCond := pathCond \ pathCond[k] = i
                pathResult := packpath(leaf, cdr(baseExprs), len(path), pathResult);
            result := ite(pathCond, pathResult, result);
        return result
Pack function - run example

- Lets run the pack(.) function on - ss[8].ts[5]
Pack function - run example

- Let's run the pack(.) function on - ss[8].ts[5]

```python
def pack(expr):
    baseExprs := list of base sub-expressions of expr;  # [ss, ss[8], ss[8].ts, ss[8].ts[5]]
    baseExpr := car(baseExprs);
    ss
    if baseExpr is a state variable then
        return packpath(tree(type(expr)), baseExprs, 0, constarr[int,int](0))
    if baseExpr is a storage pointer then
        result := constarr[int,int](0);
        prefix := E(baseExpr);
        foreach path to a leaf in tree(type(baseExpr)) do
            pathResult, pathCond := prefix, true;
            foreach kth edge on the path with label id (i) do
                pathCond := pathCond ∧ prefix[k] = i
            pathResult := packpath(leaf, cdr(baseExprs), len(path), pathResult);
            result := ite(pathCond, pathResult, result);
        return result
```
Pack function - run example

- Lets run the pack(.) function on - ss[8].ts[5]

```python
def packpath(node, subExprs, d, result):
    foreach expr in subExprs do
        if expr = id \lor expr = e.id then
            find edge node \(\xrightarrow{\text{id}}\) child;
            result := result[d ↦ i];
        if expr = e[idx] then
            find edge node \(\xrightarrow{\text{i}}\) child;
            result := result[d ↦ E(idx)];
        node, d := child, d + 1;
    return result
```

```
node = contract (tree)
subExprs = [ss, ss[8], ss[8].ts, ss[8].ts[5]]
d = 0
result = []
```
Pack function - run example

- Lets run the pack(.) function on - ss[8].ts[5]
Pack function - run example

- Lets run the pack(.) function on - ss[8].ts[5]

```python
def packpath (node, subExprs, d, result):
    foreach expr in subExprs do
        if expr = id \lor expr = e.id then
            find edge node \xrightarrow{id (i)} child;
            result := result[d \leftarrow i];
        end
        if expr = e[idx] then
            find edge node \xrightarrow{(i)} child;
            result := result[d \leftarrow E(idx)];
            node, d := child, d + 1;
        end
    end
    return result
```

node = contract (tree)
subExprs = [ss, ss[8], ss[8].ts, ss[8].ts[5]]
d = 0
result = [2]
expr = ss
i = 2
child = S[]
Let's run the pack(.) function on - ss[8].ts[5]

```python
def packpath(node, subExprs, d, result):
    for expr in subExprs:
        if expr == id \\= e.id then
            find edge node \(i\) \(\rightarrow\) child;
            result := result[d \(\leftarrow\) i];
        if expr == e[idx] then
            find edge node \(i\) \(\rightarrow\) child;
            result := result[d \(\leftarrow\) E(idx)];
        node, d := child, d + 1;
    return result
```

node = S[]
subExprs = [ss, ss[8], ss[8].ts, ss[8].ts[5]]
d = 1
result = [2]
expr = ss
Pack function - run example

- Let's run the pack(.) function on - ss[8].ts[5]

def packpath(node, subExprs, d, result):
    foreach expr in subExprs do
        if expr = id \lor expr = e.id then
            find edge node \xrightarrow{id (i)} child;
            result := result[d \leftarrow i];
        if expr = e[idx] then
            find edge node \xrightarrow{(i)} child;
            result := result[d \leftarrow E(idx)];
        node, d := child, d + 1;
    return result

node = S[]
subExprs = [ss, ss[8], ss[8].ts, ss[8].ts[5]]
d = 1
result = [2]
expr = ss[8]
Pack function - run example

- Lets run the pack(.) function on - ss[8].ts[5]

```python
def packpath (node, subExprs, d, result):
    foreach expr in subExprs do
        if expr = id ∨ expr = e.id then
            find edge node −→ child;
            result := result[d ← i];
            if expr = e[idx] then
                find edge node −→ child;
                result := result[d ← E(idx)];
                node, d := child, d + 1;
    return result
```

node = S[]
subExprs = [ss, ss[8], ss[8].ts, ss[8].ts[5]]
d = 1
result = [2]
expr = ss[8]
(idx = 8)
Pack function - run example

- Lets run the pack(.) function on - ss[8].ts[5]

```python
def packpath (node, subExprs, d, result):
    foreach expr in subExprs do
        if expr = id ∨ expr = e.id then
            find edge node \(i\) \(\rightarrow\) child;
            result := result[d \leftarrow i];
        if expr = e[idx] then
            find edge node \(i\) \(\rightarrow\) child;
            result := result[d \leftarrow E(idx)];
        node, d := child, d + 1;
    return result
```

node = S[]
subExprs = [ss, ss[8], ss[8].ts, ss[8].ts[5]]
d = 1
result = [2,8]
expr = ss[8]
(idx = 8)
child = S
def pack(expr):
    baseExprs := list of base sub-expressions of expr;
    baseExpr := car(baseExprs);

    if baseExpr is a state variable then
        return packpath(tree(type(expr)), baseExprs, 0, constarr[int][int](0))
    
    if baseExpr is a storage pointer then
        result := constarr[int][int](0);
        prefix := E(baseExpr);
        foreach path to a leaf in tree(type(baseExpr)) do
            pathResult, pathCond := prefix, true;
            foreach kth edge on the path with label id (i) do
                pathCond := pathCond ∧ prefix[k] = i
            pathResult := packpath(leaf, cdr(baseExprs), len(path), pathResult);
            result := ite(pathCond, pathResult, result);
        
        return result

contract C {
    struct T { int z; }
    struct S { int x; T[] ta; }
    T t;
    S s;
    S[] sa;
    function g() public view {
        s storage locals = sa[5];
        T storage unknownPath = locals.ta[3];
    }
}
Unpack function

- The function takes a storage pointer (of type \([\text{int}\times\text{int}]\)) and produces a conditional expression that decodes any given path into one of the leaves of the storage tree.
- The SMT equivalent to dereference.
def unpack(ptr):
    return unpack(ptr, tree(type(ptr)), empty, 0);
def unpack(ptr, node, expr, d):
    result := empty;
    if node has no outgoing edges then result := expr;
    if node is contract then
        foreach edge node \( \xrightarrow{id (i)} \) child do
            result := ite(ptr[d] = i, unpack(ptr, child, id, d + 1), result);
    if node is struct then
        foreach edge node \( \xrightarrow{id (i)} \) child do
            result := ite(ptr[d] = i, unpack(ptr, child, expr.id, d + 1), result);
    if node is array/mapping with edge node \( \xrightarrow{i} \) child then
        result := unpack(ptr, child, expr[ptr[d]], d + 1);
    return result;
Agenda

- Motivation
- Background
- Formalization
  - Types
  - Local storage pointers
  - State variables, functions, memory and defval
  - Assignments
  - Expressions
  - Statements
- Summary
State variables

- Always stored in storage
- Add the declaration `s_i : T(type(s_i) storage)`
- Wlog, we assume they are assigned in the constructor
Function calls

- The types of function variables and return values can be either memory or storage ptr
- We can treat them as regular assignments
- For each parameter and return value, we add $p_i : T(type(p_i))$, $r_i : T(type(r_i))$
Memory allocation

- We use arrays as heaps
- We keep track of an allocation counter called `refcnt`
- In each declaration, `refcnt` is incremented

\[
\tau(T[] \text{ memory}) \doteq \text{ int} \quad \text{with} \quad \begin{cases} 
\text{MemArr}_T(\text{arr : int} | T(T), \text{length : int}) \\
\text{arrheap}_T : [\text{int}]\text{MemArr}_T
\end{cases}
\]

\[
\tau(\text{struct } S \text{ memory}) \doteq \text{ int} \quad \text{with} \quad \begin{cases} 
\text{MemStruct}_S(\ldots, m_i : \tau(S_i), \ldots) \\
\text{structheap}_S : [\text{int}]\text{MemStruct}_S
\end{cases}
\]
Default values

- The `defval` function maps a solidity type to its default value in smt.
- Trivial for value types.

\[
\begin{align*}
\text{defval}(&\text{bool}) & \equiv false \\
\text{defval}(\text{address}) & \equiv \text{defval}(\text{int}) \equiv \text{defval}(\text{uint}) \equiv 0
\end{align*}
\]
Default values - mappings

- Mappings can only be stored in storage or storptr

\[
defval(mapping(K \rightarrow V)) = \text{constarr}_{T(K) \mid T(V)}(\text{defval}(V))\]
Default values - fixed size arrays

- Storage arrays get a value of a n sized array with recursive defval
- Memory arrays cause an int declaration, and refcnt increment
- Initialization can be done without loop

\[
\text{defval}(T[n] \text{ storage}) \triangleq \text{StorArr}_{T}(\text{constarr}_{\text{int}T(T)}(\text{defval}(T)), n) \\
\text{defval}(T[n] \text{ memory}) \triangleq \{ \text{ref} : \text{int} \} \text{ (fresh symbol)} \\
\{ \text{ref} := \text{refcnt} := \text{refcnt} + 1 \} \\
\{ \text{arrheap}_T[\text{ref}].\text{length} := n \} \\
\{ \text{arrheap}_T[\text{ref}].\text{arr}[i] := \text{defval}(T) \} \quad \text{for } 0 \leq i \leq n \\
\text{ref}
\]

\[
[\text{MemArr}_T(\text{arr} : [\text{int}]T(T), \text{length} : \text{int})] \\
[\text{arrheap}_T : [\text{int}]\text{MemArr}_T]
\]
Default values - dynamic arrays

- Initialized as a 0 length fixed size array

```c
defval(T[] storage) ≅ defval(T[0] storage)
defval(T[] memory) ≅ defval(T[0] memory)
```
Default values - structs

- Similar to arrays
- Initialization can be done without loops

\[
\text{defval}(\text{struct } S \text{ storage}) \triangleq \text{StorStructs}(\ldots, \text{defval}(S_i), \ldots) \\
\text{defval}(\text{struct } S \text{ memory}) \triangleq [\text{ref : int}] \text{ (fresh symbol)} \\
\{\text{ref} := \text{refcnt} := \text{refcnt} + 1\} \\
\{\text{structheap}_S[\text{ref}].m_i = \text{defval}(S_i)\} \text{ for each } m_i \\
\text{ref}
\]

\[
[\text{MemStruct}_S(\ldots, m_i : \mathcal{T}(S_i), \ldots)] \\
[\text{structheap}_S : [\text{int}]\text{MemStruct}_S]
\]
Agenda

- Motivation
- Background
- Formalization
  - Types
  - Local storage pointers
  - State variables, functions, memory and defval
  - Assignments
  - Expressions
  - Statements
- Summary
The A(.,.) function

- Reference type assignments can be either pointer assignments or value assignments
  - value assignments can create new allocations
- $A(lhs, rhs)$ denotes assigning rhs to lhs as SMT expressions
- Value type assignments are simple to convert to smt

\[
\begin{align*}
A(lhs, rhs) & \doteq lhs := rhs \quad \text{for value type operands} \\
A(lhs, rhs) & \doteq A_M(lhs, rhs) \quad \text{for mapping type operands} \\
A(lhs, rhs) & \doteq A_S(lhs, rhs) \quad \text{for struct type operands} \\
A(lhs, rhs) & \doteq A_A(lhs, rhs) \quad \text{for array type operands}
\end{align*}
\]
Mappings

- Solidity disables mapping assignments
- Storage pointers can be assigned, either from a pointer or a storage variable.

\[
\begin{align*}
A_M(lhs : sp, rhs : s) & \iff lhs := \text{pack}(rhs) \\
A_M(lhs : sp, rhs : sp) & \iff lhs := rhs \\
A_M(lhs, rhs) & \iff \{\}
\end{align*}
\]
\( \mathcal{A}_S(\text{lhs} : s, \text{rhs} : s) \quad \vdash \text{lhs} := \text{rhs} \)

\( \mathcal{A}_S(\text{lhs} : s, \text{rhs} : m) \quad \vdash \mathcal{A}(\text{lhs}.m_i, \text{structheap}_{\text{type}(\text{rhs})}[\text{rhs}].m_i) \text{ for each } m_i \)

\( \mathcal{A}_S(\text{lhs} : s, \text{rhs} : sp) \quad \vdash \mathcal{A}_S(\text{lhs, unpack}(\text{rhs})) \)

\( \mathcal{A}_S(\text{lhs} : m, \text{rhs} : m) \quad \vdash \text{lhs} := \text{rhs} \)

\( \mathcal{A}_S(\text{lhs} : m, \text{rhs} : s) \quad \vdash \text{lhs} := \text{refcnt} := \text{refcnt} + 1 \)

\[ \mathcal{A}(\text{structheap}_{\text{type}(\text{lhs})}[\text{lhs}].m_i, \text{rhs}.m_i) \text{ for each } m_i \]

\( \mathcal{A}_S(\text{lhs} : m, \text{rhs} : sp) \quad \vdash \mathcal{A}_S(\text{lhs, unpack}(\text{rhs})) \)

\( \mathcal{A}_S(\text{lhs} : sp, \text{rhs} : s) \quad \vdash \text{lhs} := \text{pack}(\text{rhs}) \)

\( \mathcal{A}_S(\text{lhs} : sp, \text{rhs} : sp) \quad \vdash \text{lhs} := \text{rhs} \)
Arrays

\[
\begin{align*}
\mathcal{A}_A(lhs : s, rhs : s) & \triangleq lhs := rhs \\
\mathcal{A}_A(lhs : s, rhs : m) & \triangleq lhs := arrheap_{type(rhs)}[rhs] \\
\mathcal{A}_A(lhs : s, rhs : sp) & \triangleq \mathcal{A}_A(lhs, \text{unpack}(rhs)) \\
\mathcal{A}_A(lhs : m, rhs : m) & \triangleright lhs := rhs \\
\mathcal{A}_A(lhs : m, rhs : s) & \triangleright lhs := refcnt := refcnt + 1 \\
& \quad arrheap_{type(lhs)}[lhs] := rhs \\
\mathcal{A}_A(lhs : m, rhs : sp) & \triangleright \mathcal{A}_A(lhs, \text{unpack}(rhs)) \\
\mathcal{A}_A(lhs : sp, rhs : s) & \triangleright lhs := \text{pack}(rhs) \\
\mathcal{A}_A(lhs : sp, rhs : sp) & \triangleright lhs := rhs
\end{align*}
\]
Agenda

- Motivation
- Background
- Formalization
  - Types
  - Local storage pointers
  - State variables, functions, memory and defval
  - Assignments
  - Expressions
  - Statements
- Summary
The $\varepsilon(.)$ function

- Translates a Solidity expression to an SMT expression
- Can introduce side effects (declarations and statements)
The $\varepsilon(.)$ function - member access

\[
\begin{align*}
\varepsilon(id) & \equiv id \\
\varepsilon(expr.id) & \equiv \varepsilon(expr).\varepsilon(id) & \text{if type(expr) = struct } S \text{ storage} \\
\varepsilon(expr.id) & \equiv \text{unpack}(\varepsilon(expr)).\varepsilon(id) & \text{if type(expr) = struct } S \text{ storptr} \\
\varepsilon(expr.id) & \equiv \text{structheap}_S[\varepsilon(expr)].\varepsilon(id) & \text{if type(expr) = struct } S \text{ memory} \\
\varepsilon(expr.id) & \equiv \varepsilon(expr).\varepsilon(id) & \text{if type(expr) = } T[] \text{ storage} \\
\varepsilon(expr.id) & \equiv \text{unpack}(\varepsilon(expr)).\varepsilon(id) & \text{if type(expr) = } T[] \text{ storptr} \\
\varepsilon(expr.id) & \equiv \text{arrheap}_T[\varepsilon(expr)].\varepsilon(id) & \text{if type(expr) = } T[] \text{ memory}
\end{align*}
\]
The $\varepsilon(.)$ function - index access

\[
\begin{align*}
\varepsilon(expr[idx]) & \triangleq \varepsilon(expr).arr[\varepsilon(idx)] & \text{if } \text{type}(expr) = T[] & \text{storage} \\
\varepsilon(expr[idx]) & \triangleq \text{unpack}(\varepsilon(expr)).arr[\varepsilon(idx)] & \text{if } \text{type}(expr) = T[] & \text{storptr} \\
\varepsilon(expr[idx]) & \triangleq \text{arrheap}_T[\varepsilon(expr)].arr[\varepsilon(idx)] & \text{if } \text{type}(expr) = T[] & \text{memory} \\
\varepsilon(expr[idx]) & \triangleq \varepsilon(expr)[\varepsilon(idx)] & \text{if } \text{type}(expr) = \text{mapping}(K=>V) & \text{storage} \\
\varepsilon(expr[idx]) & \triangleq \text{unpack}(\varepsilon(expr))[\varepsilon(idx)] & \text{if } \text{type}(expr) = \text{mapping}(K=>V) & \text{storptr}
\end{align*}
\]
The $\varepsilon(.)$ function - conditionals

- Evaluates both expressions, uses memory if at least one is in memory, storptr otherwise
- Creates the variables and calls the side effects before checking the conditional

\[
\varepsilon(\text{cond} \ ? \ \text{expr}_T \ : \ \text{expr}_F) = \begin{cases} 
\text{var}_T : T(\text{type}(\text{cond} \ ? \ \text{expr}_T \ : \ \text{expr}_F)) & \text{(fresh symbol)} \\
\text{var}_F : T(\text{type}(\text{cond} \ ? \ \text{expr}_T \ : \ \text{expr}_F)) & \text{(fresh symbol)} \\
\{A(\text{var}_T, \varepsilon(\text{expr}_T))\} \\
\{A(\text{var}_F, \varepsilon(\text{expr}_F))\} \\
\text{ite}(\varepsilon(\text{cond}), \text{var}_T, \text{var}_F)
\end{cases}
\]
The $\varepsilon(.)$ function - memory allocation

$$\varepsilon(\text{new } T[] (expr)) \equiv [\text{ref} : \text{int}] \text{ (fresh symbol)}$$

$$\{ \text{ref} := \text{refcnt} := \text{refcnt} + 1 \}$$

$$\{ \text{arrheap}_T[\text{ref}].\text{length} := \varepsilon(\text{expr}) \}$$

$$\{ \text{arrheap}_T[\text{ref}].\text{arr}[i] := \text{defval}(T) \} \text{ for } 0 \leq i \leq \varepsilon(\text{expr})$$

$$\text{ref}$$

$$\varepsilon(S(\ldots, expr_i, \ldots)) \equiv [\text{ref} : \text{int}] \text{ (fresh symbol)}$$

$$\{ \text{ref} := \text{refcnt} := \text{refcnt} + 1 \}$$

$$\{ \text{structheap}_S[\text{ref}].m_i := \varepsilon(\text{expr}_i) \} \text{ for each member } m_i$$

$$\text{ref}$$
Agenda

- Motivation
- Background
- Formalization
  - Types
  - Local storage pointers
  - State variables, functions, memory and defval
  - Assignments
  - Expressions
  - Statements
- Summary
The S[.] function

- Translates Solidity statements to a list of statements in the SMT program

\[
stmt ::= id := expr \\
| if expr then stmt* else stmt* \quad \text{Assignment} \\
| \text{If-then-else}
\]
The $S[.]$ function

\[
S[T \ id] \doteq [id : T(T)]; A(id, defval(T))
\]
\[
S[T \ id = expr] \doteq [id : T(T)]; A(id, E(expr))
\]
\[
S[\text{delete } e] \doteq A(E(e), \text{defval(type(e)))}
\]
\[
S[l_1, \ldots, l_n = r_1, \ldots, r_n] \doteq [tmp_i : T(\text{type}(r_i))] \text{ for } 1 \leq i \leq n \text{ (fresh symbols)}
\]
\[
A(tmp_i, E(r_i)) \text{ for } 1 \leq i \leq n
\]
\[
A(E(l_i), tmp_i) \text{ for } n \geq i \geq 1 \text{ (reversed)}
\]
Reverse assignment example

```solidity
contract C {
    struct S { int x; }
    S s1;
    S s2;
    S s3;

    function primitiveAssign() public {
        s1.x = 1; s2.x = 2; s3.x = 3;
        (s1.x, s3.x, s2.x) = (s3.x, s2.x, s1.x);
        // s1.x == 3, s2.x == 1, s3.x == 2
    }

    function storageAssign() public {
        s1.x = 1; s2.x = 2; s3.x = 3;
        (s1, s3, s2) = (s3, s2, s1);
        // s1.x == 1, s2.x == 1, s3.x == 1
    }
}
```
The $S[.]$ function

\[
S[e_1.push(e_2)] \doteq A(\mathcal{E}(e_1).arr[\mathcal{E}(e_1).length], \mathcal{E}(e_2)) \\
\mathcal{E}(e_1).length := \mathcal{E}(e_1).length + 1
\]

\[
S[e.pop()] \doteq \mathcal{E}(e).length := \mathcal{E}(e).length - 1 \\
A(\mathcal{E}(e).arr[\mathcal{E}(e).length], \text{defval}(\text{arrtype}(\mathcal{E}(e))))
\]
Dangling pointer example

```solidity
class C {
  struct S {
    int x;
  }
  S[] a;
  constructor() {
    a.push (S(1));
    S storage s = a[0];
    a.pop();
    int newInt = s.x;
    // int newInt = a[0].x causes a runtime error
  }
}
```
Agenda

- Motivation
- Background
- Formalization
  - Types
  - Local storage pointers
  - State variables, function, memory and defval
  - Assignments
  - Expressions
  - Statements
- Summary
SOLC-VERIFY

- SOLC-VERIFY is a verification tool that uses this approach
- Converts to boogie
- Better results than other existing tools
Summary

- The solidity memory model - storage and memory
- High-level SMT-based formalization of the Solidity memory model semantics.
  - Covers both memory and storage locations
  - Uses the packing method for storage pointers
  - Allows deep copies
Questions?