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

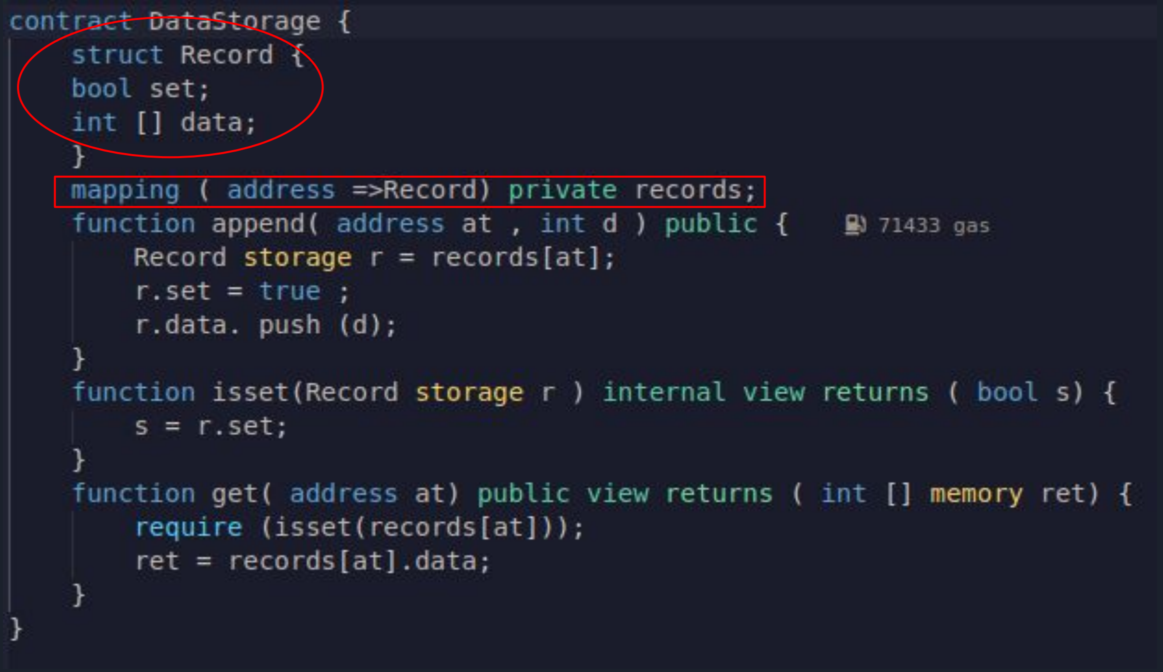
Contract storage

- 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



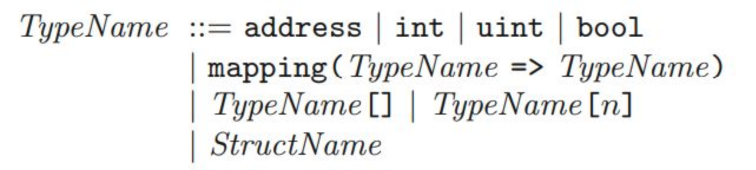
```
contract DataStorage {
    struct Record {
    bool set;
    int [] data;
    }
    mapping ( address =>Record) private records;
    function append( address at , int d ) public {    71433 gas
        Record storage r = records[at];
        r.set = true ;
        r.data. push (d);
    }
    function 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

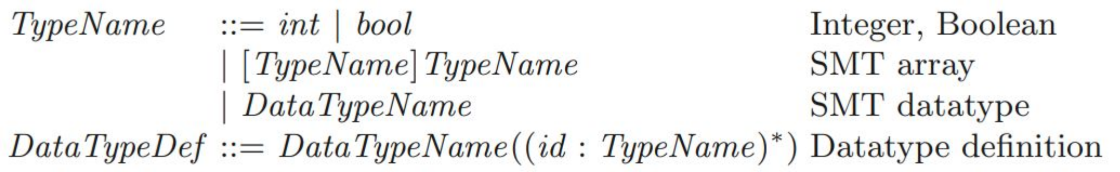
- Motivation
- Background
- **Formalization**
    - **Types**
    - Local storage pointers
    - State variables, function, memory and defval
    - Assignments
    - Expressions
    - Statements
- Summary

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

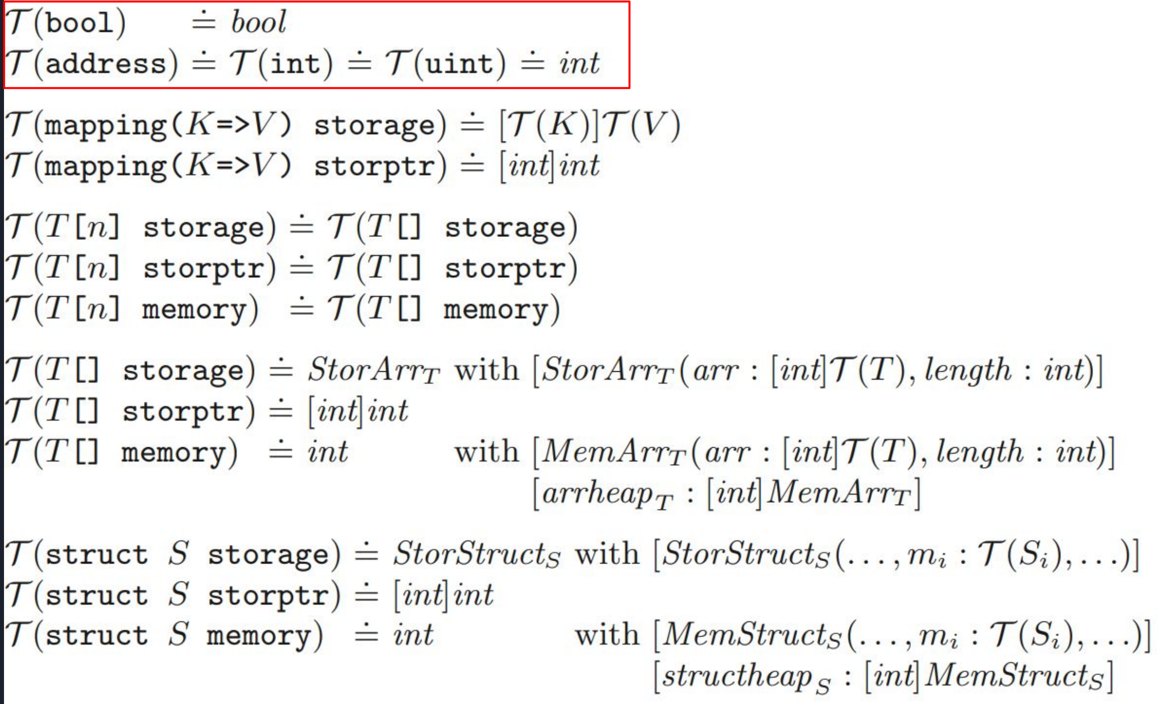


$$\begin{aligned} TypeName \ ::= &\ \texttt{address} \mid \texttt{int} \mid \texttt{uint} \mid \texttt{bool}\\ &\mid \texttt{mapping(}TypeName\ \texttt{=>}\ TypeName\texttt{)}\\ &\mid TypeName\texttt{[]} \mid TypeName\texttt{[}n\texttt{]}\\ &\mid StructName\end{aligned}$$



$TypeName$	$::= int \mid bool$	Integer, Boolean
	$\mid [TypeName]\,TypeName$	SMT array
	$\mid DataTypeName$	SMT datatype
$DataTypeDef$	$::= DataTypeName((id : TypeName)^*)$	Datatype definition

Value types



$$\boxed{\begin{aligned}
\mathcal{T}(\texttt{bool}) &\doteq \mathit{bool}\\
\mathcal{T}(\texttt{address}) &\doteq \mathcal{T}(\texttt{int}) \doteq \mathcal{T}(\texttt{uint}) \doteq \mathit{int}
\end{aligned}}$$

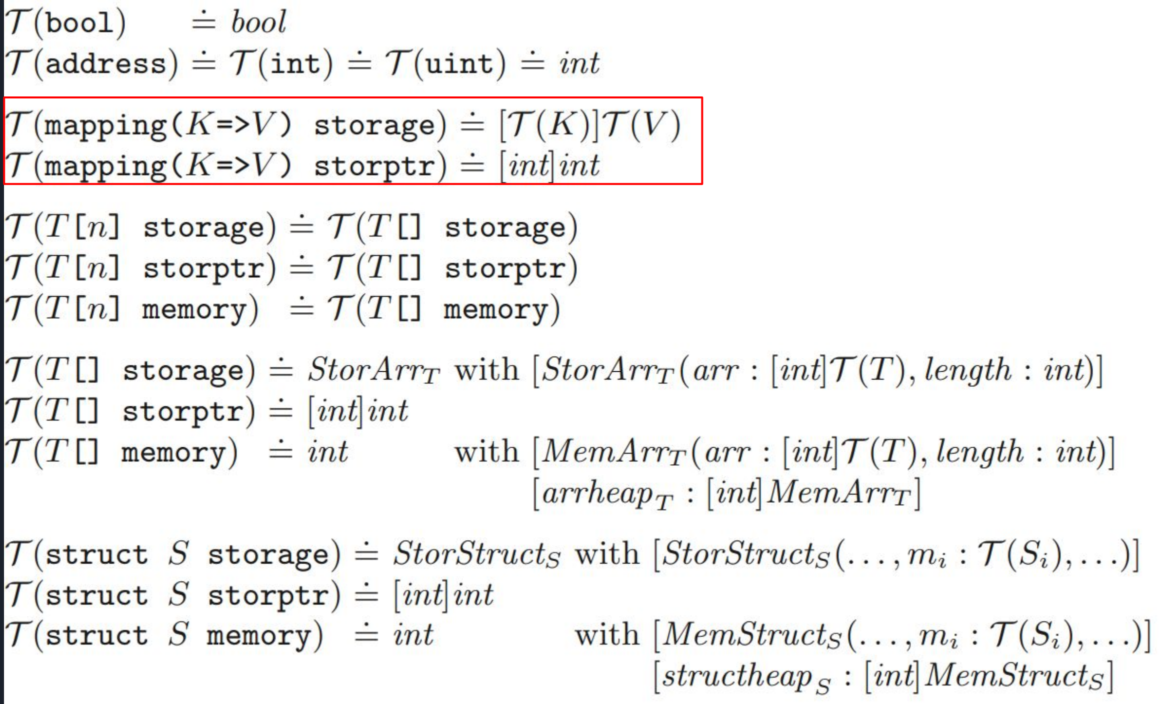
$$\begin{aligned}
\mathcal{T}(\texttt{mapping(}K\texttt{=>}V\texttt{) storage}) &\doteq [\mathcal{T}(K)]\mathcal{T}(V)\\
\mathcal{T}(\texttt{mapping(}K\texttt{=>}V\texttt{) storptr}) &\doteq [\mathit{int}]\mathit{int}
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}(T\texttt{[}n\texttt{] storage}) &\doteq \mathcal{T}(T\texttt{[] storage})\\
\mathcal{T}(T\texttt{[}n\texttt{] storptr}) &\doteq \mathcal{T}(T\texttt{[] storptr})\\
\mathcal{T}(T\texttt{[}n\texttt{] memory}) &\doteq \mathcal{T}(T\texttt{[] memory})
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}(T\texttt{[] storage}) &\doteq \mathit{StorArr}_T \text{ with } [\mathit{StorArr}_T(\mathit{arr} : [\mathit{int}]\mathcal{T}(T), \mathit{length} : \mathit{int})]\\
\mathcal{T}(T\texttt{[] storptr}) &\doteq [\mathit{int}]\mathit{int}\\
\mathcal{T}(T\texttt{[] memory}) &\doteq \mathit{int} \qquad\quad \text{with } [\mathit{MemArr}_T(\mathit{arr} : [\mathit{int}]\mathcal{T}(T), \mathit{length} : \mathit{int})]\\
&\qquad\qquad\qquad\quad\; [\mathit{arrheap}_T : [\mathit{int}]\mathit{MemArr}_T]
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}(\texttt{struct } S \texttt{ storage}) &\doteq \mathit{StorStruct}_S \text{ with } [\mathit{StorStruct}_S(\ldots, m_i : \mathcal{T}(S_i), \ldots)]\\
\mathcal{T}(\texttt{struct } S \texttt{ storptr}) &\doteq [\mathit{int}]\mathit{int}\\
\mathcal{T}(\texttt{struct } S \texttt{ memory}) &\doteq \mathit{int} \qquad\quad \text{with } [\mathit{MemStruct}_S(\ldots, m_i : \mathcal{T}(S_i), \ldots)]\\
&\qquad\qquad\qquad\quad\; [\mathit{structheap}_S : [\mathit{int}]\mathit{MemStruct}_S]
\end{aligned}$$

Mappings



$$\begin{aligned}
&\mathcal{T}(\texttt{bool}) \doteq \mathit{bool}\\
&\mathcal{T}(\texttt{address}) \doteq \mathcal{T}(\texttt{int}) \doteq \mathcal{T}(\texttt{uint}) \doteq \mathit{int}
\end{aligned}$$

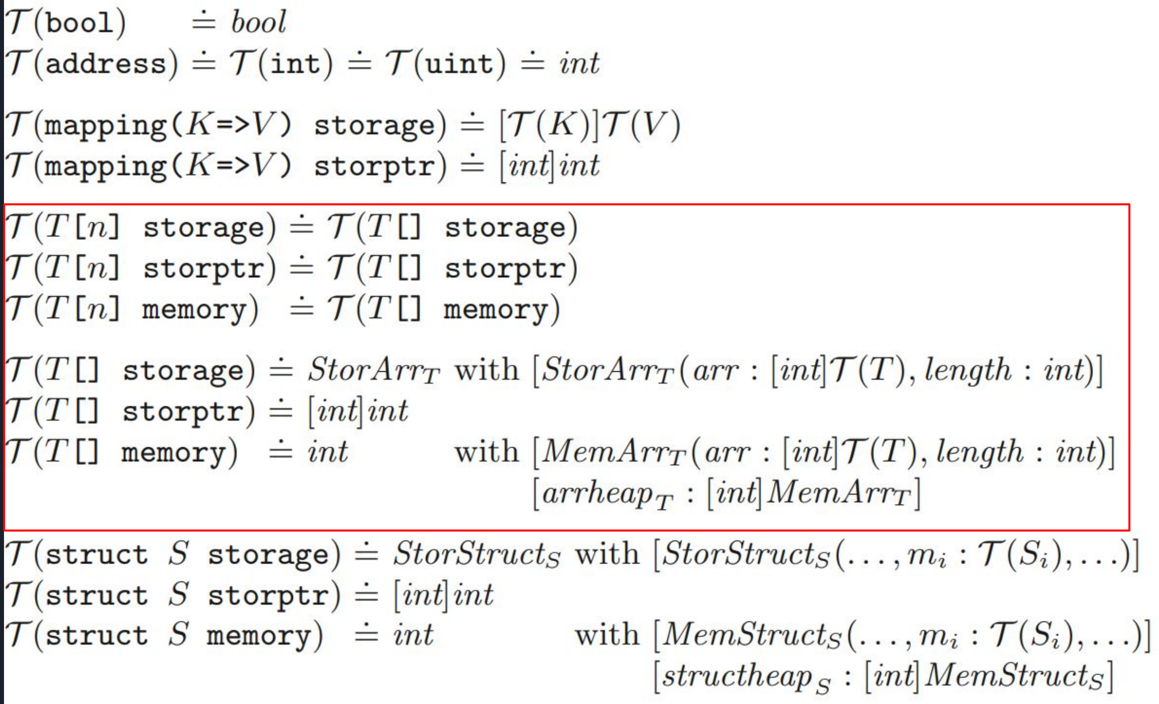
$$\boxed{\begin{aligned}
&\mathcal{T}(\texttt{mapping(}K\texttt{=>}V\texttt{) storage}) \doteq [\mathcal{T}(K)]\mathcal{T}(V)\\
&\mathcal{T}(\texttt{mapping(}K\texttt{=>}V\texttt{) storptr}) \doteq [\mathit{int}]\mathit{int}
\end{aligned}}$$

$$\begin{aligned}
&\mathcal{T}(T\texttt{[}n\texttt{] storage}) \doteq \mathcal{T}(T\texttt{[] storage})\\
&\mathcal{T}(T\texttt{[}n\texttt{] storptr}) \doteq \mathcal{T}(T\texttt{[] storptr})\\
&\mathcal{T}(T\texttt{[}n\texttt{] memory}) \doteq \mathcal{T}(T\texttt{[] memory})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{T}(T\texttt{[] storage}) \doteq \mathit{StorArr}_T \text{ with } [\mathit{StorArr}_T(\mathit{arr} : [\mathit{int}]\mathcal{T}(T), \mathit{length} : \mathit{int})]\\
&\mathcal{T}(T\texttt{[] storptr}) \doteq [\mathit{int}]\mathit{int}\\
&\mathcal{T}(T\texttt{[] memory}) \doteq \mathit{int} \quad \text{with } [\mathit{MemArr}_T(\mathit{arr} : [\mathit{int}]\mathcal{T}(T), \mathit{length} : \mathit{int})]\\
&\qquad\qquad\qquad\qquad\qquad [\mathit{arrheap}_T : [\mathit{int}]\mathit{MemArr}_T]
\end{aligned}$$

$$\begin{aligned}
&\mathcal{T}(\texttt{struct } S \texttt{ storage}) \doteq \mathit{StorStruct}_S \text{ with } [\mathit{StorStruct}_S(\ldots, m_i : \mathcal{T}(S_i), \ldots)]\\
&\mathcal{T}(\texttt{struct } S \texttt{ storptr}) \doteq [\mathit{int}]\mathit{int}\\
&\mathcal{T}(\texttt{struct } S \texttt{ memory}) \doteq \mathit{int} \quad \text{with } [\mathit{MemStruct}_S(\ldots, m_i : \mathcal{T}(S_i), \ldots)]\\
&\qquad\qquad\qquad\qquad\qquad [\mathit{structheap}_S : [\mathit{int}]\mathit{MemStruct}_S]
\end{aligned}$$

Arrays



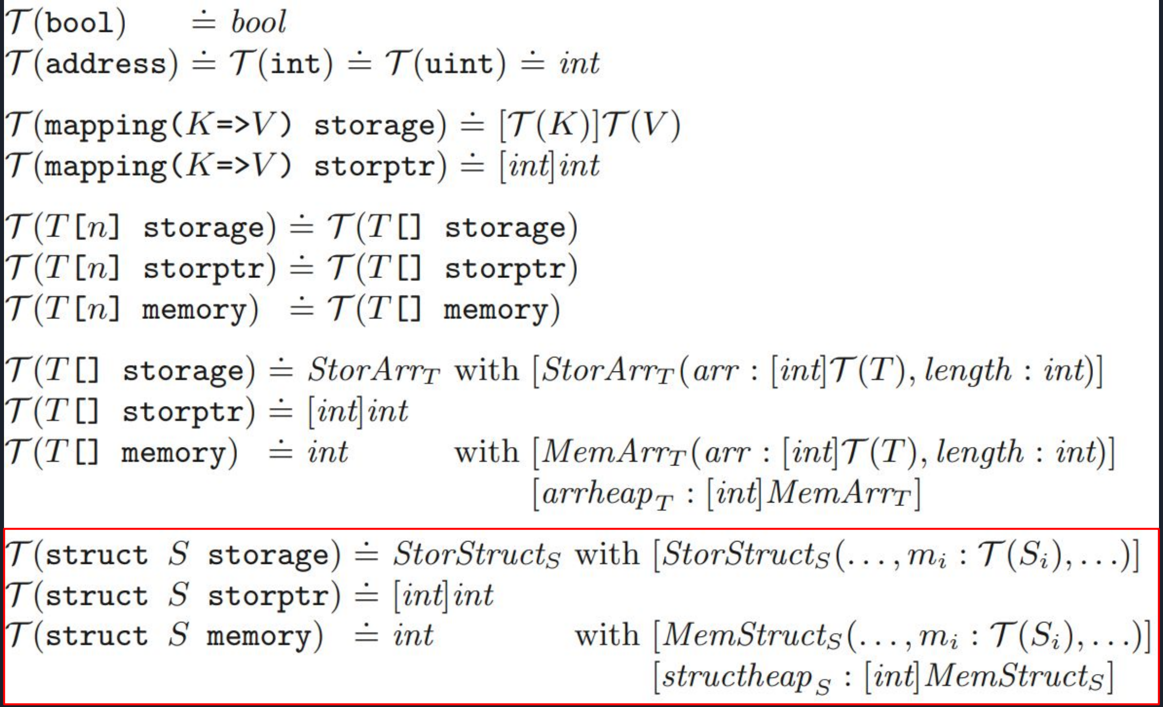
$$
\begin{aligned}
&\mathcal{T}(\texttt{bool}) \doteq \mathit{bool}\\
&\mathcal{T}(\texttt{address}) \doteq \mathcal{T}(\texttt{int}) \doteq \mathcal{T}(\texttt{uint}) \doteq \mathit{int}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{T}(\texttt{mapping(}K\texttt{=>}V\texttt{) storage}) \doteq [\mathcal{T}(K)]\mathcal{T}(V)\\
&\mathcal{T}(\texttt{mapping(}K\texttt{=>}V\texttt{) storptr}) \doteq [\mathit{int}]\mathit{int}
\end{aligned}
$$

$$
\boxed{
\begin{aligned}
&\mathcal{T}(T\texttt{[}n\texttt{] storage}) \doteq \mathcal{T}(T\texttt{[] storage})\\
&\mathcal{T}(T\texttt{[}n\texttt{] storptr}) \doteq \mathcal{T}(T\texttt{[] storptr})\\
&\mathcal{T}(T\texttt{[}n\texttt{] memory}) \doteq \mathcal{T}(T\texttt{[] memory})\\
\\
&\mathcal{T}(T\texttt{[] storage}) \doteq \mathit{StorArr}_T \text{ with } [\mathit{StorArr}_T(\mathit{arr} : [\mathit{int}]\mathcal{T}(T), \mathit{length} : \mathit{int})]\\
&\mathcal{T}(T\texttt{[] storptr}) \doteq [\mathit{int}]\mathit{int}\\
&\mathcal{T}(T\texttt{[] memory}) \doteq \mathit{int} \qquad \text{with } [\mathit{MemArr}_T(\mathit{arr} : [\mathit{int}]\mathcal{T}(T), \mathit{length} : \mathit{int})]\\
&\qquad\qquad\qquad\qquad\qquad\quad [\mathit{arrheap}_T : [\mathit{int}]\mathit{MemArr}_T]
\end{aligned}
}
$$

$$
\begin{aligned}
&\mathcal{T}(\texttt{struct } S \texttt{ storage}) \doteq \mathit{StorStruct}_S \text{ with } [\mathit{StorStruct}_S(\ldots, m_i : \mathcal{T}(S_i), \ldots)]\\
&\mathcal{T}(\texttt{struct } S \texttt{ storptr}) \doteq [\mathit{int}]\mathit{int}\\
&\mathcal{T}(\texttt{struct } S \texttt{ memory}) \doteq \mathit{int} \qquad \text{with } [\mathit{MemStruct}_S(\ldots, m_i : \mathcal{T}(S_i), \ldots)]\\
&\qquad\qquad\qquad\qquad\qquad\qquad [\mathit{structheap}_S : [\mathit{int}]\mathit{MemStruct}_S]
\end{aligned}
$$

Structs



$$
\begin{aligned}
&\mathcal{T}(\texttt{bool}) \doteq \mathit{bool}\\
&\mathcal{T}(\texttt{address}) \doteq \mathcal{T}(\texttt{int}) \doteq \mathcal{T}(\texttt{uint}) \doteq \mathit{int}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{T}(\texttt{mapping(}K\texttt{=>}V\texttt{) storage}) \doteq [\mathcal{T}(K)]\mathcal{T}(V)\\
&\mathcal{T}(\texttt{mapping(}K\texttt{=>}V\texttt{) storptr}) \doteq [\mathit{int}]\mathit{int}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{T}(T\texttt{[}n\texttt{] storage}) \doteq \mathcal{T}(T\texttt{[] storage})\\
&\mathcal{T}(T\texttt{[}n\texttt{] storptr}) \doteq \mathcal{T}(T\texttt{[] storptr})\\
&\mathcal{T}(T\texttt{[}n\texttt{] memory}) \doteq \mathcal{T}(T\texttt{[] memory})
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{T}(T\texttt{[] storage}) \doteq \mathit{StorArr}_T \text{ with } [\mathit{StorArr}_T(\mathit{arr} : [\mathit{int}]\mathcal{T}(T), \mathit{length} : \mathit{int})]\\
&\mathcal{T}(T\texttt{[] storptr}) \doteq [\mathit{int}]\mathit{int}\\
&\mathcal{T}(T\texttt{[] memory}) \doteq \mathit{int} \quad \text{ with } [\mathit{MemArr}_T(\mathit{arr} : [\mathit{int}]\mathcal{T}(T), \mathit{length} : \mathit{int})]\\
&\qquad\qquad\qquad\qquad\qquad [\mathit{arrheap}_T : [\mathit{int}]\mathit{MemArr}_T]
\end{aligned}
$$

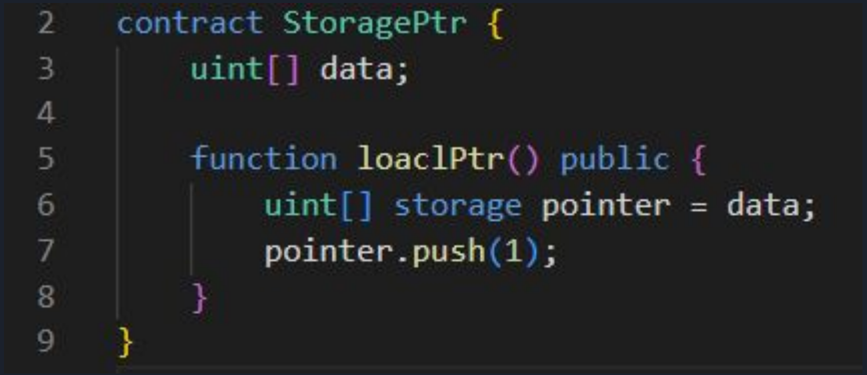
$$
\begin{aligned}
&\mathcal{T}(\texttt{struct } S \texttt{ storage}) \doteq \mathit{StorStruct}_S \text{ with } [\mathit{StorStruct}_S(\ldots, m_i : \mathcal{T}(S_i), \ldots)]\\
&\mathcal{T}(\texttt{struct } S \texttt{ storptr}) \doteq [\mathit{int}]\mathit{int}\\
&\mathcal{T}(\texttt{struct } S \texttt{ memory}) \doteq \mathit{int} \quad \text{ with } [\mathit{MemStruct}_S(\ldots, m_i : \mathcal{T}(S_i), \ldots)]\\
&\qquad\qquad\qquad\qquad\qquad [\mathit{structheap}_S : [\mathit{int}]\mathit{MemStruct}_S]
\end{aligned}
$$

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

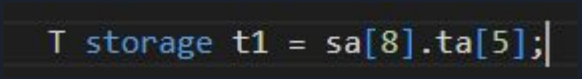


```
contract StoragePtr {
    uint[] data;

    function loaclPtr() public {
        uint[] storage pointer = data;
        pointer.push(1);
    }
}
```

Local storage pointers

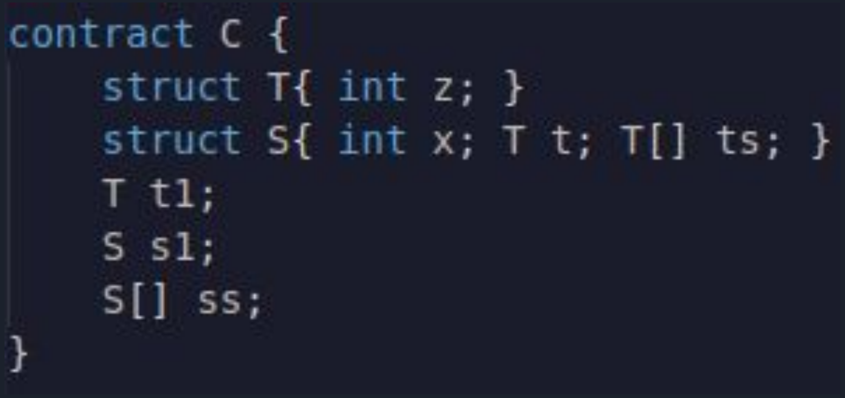
- Question: How can we encode local storage pointers with SMT?
- Partial solution: Substitute each occurrence of the local pointer with the expression that is assigned to
- Downsides: storage pointer can be reassigned, received as a function argument and more.



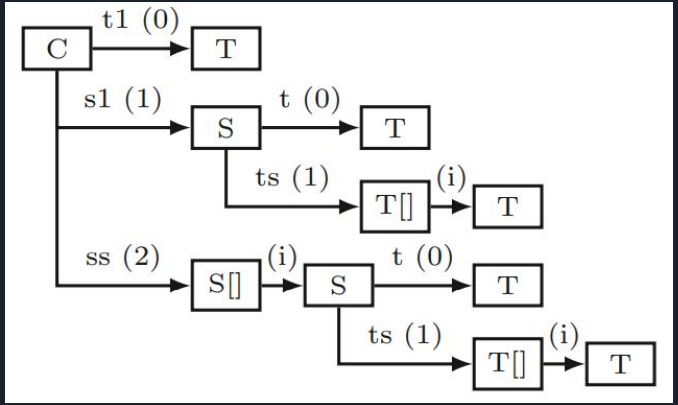
```
T storage t1 = sa[8].ta[5];
```

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*

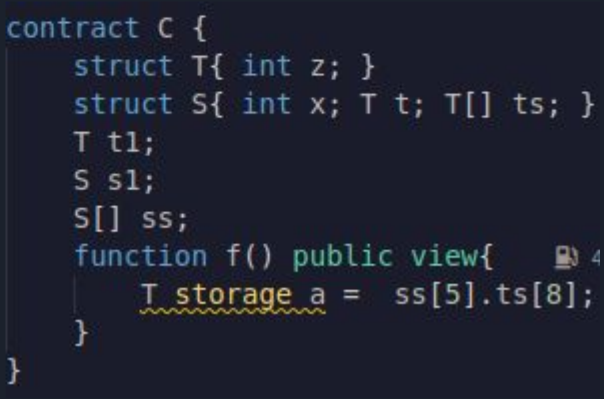


```
contract C {
    struct T{ int z; }
    struct S{ int x; T t; T[] ts; }
    T t1;
    S s1;
    S[] ss;
}
```



Local storage pointers - solution

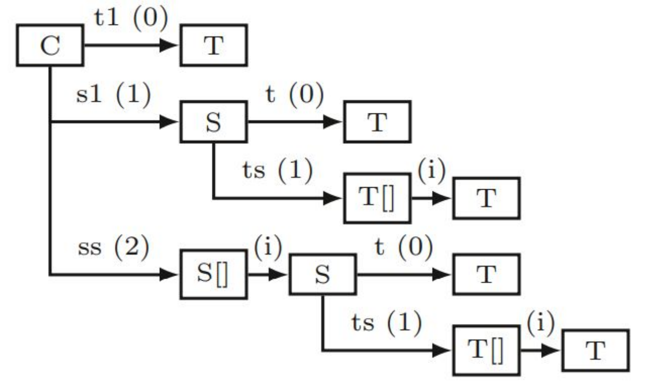
- Local storage pointer’s SMT type is always [int]int
- The array will be the finite path from the tree of values of the contract



```
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) = [int]int

a ⟶ [2,5,1,8]



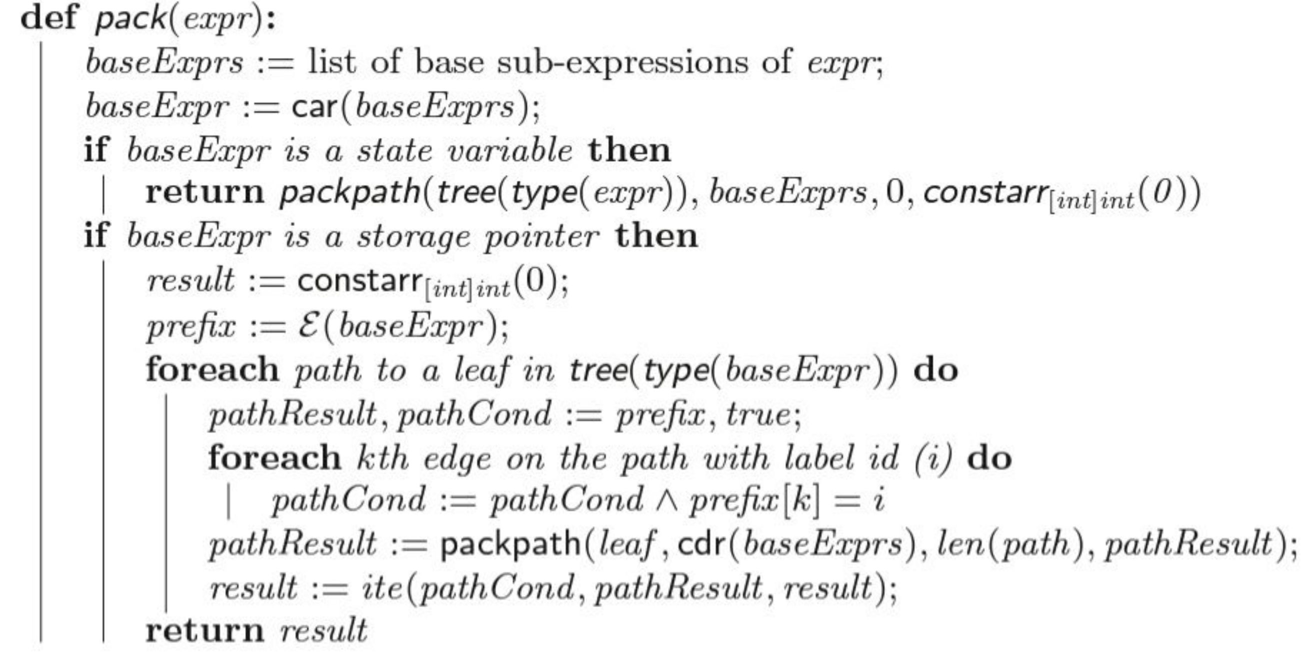
Usage

- 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

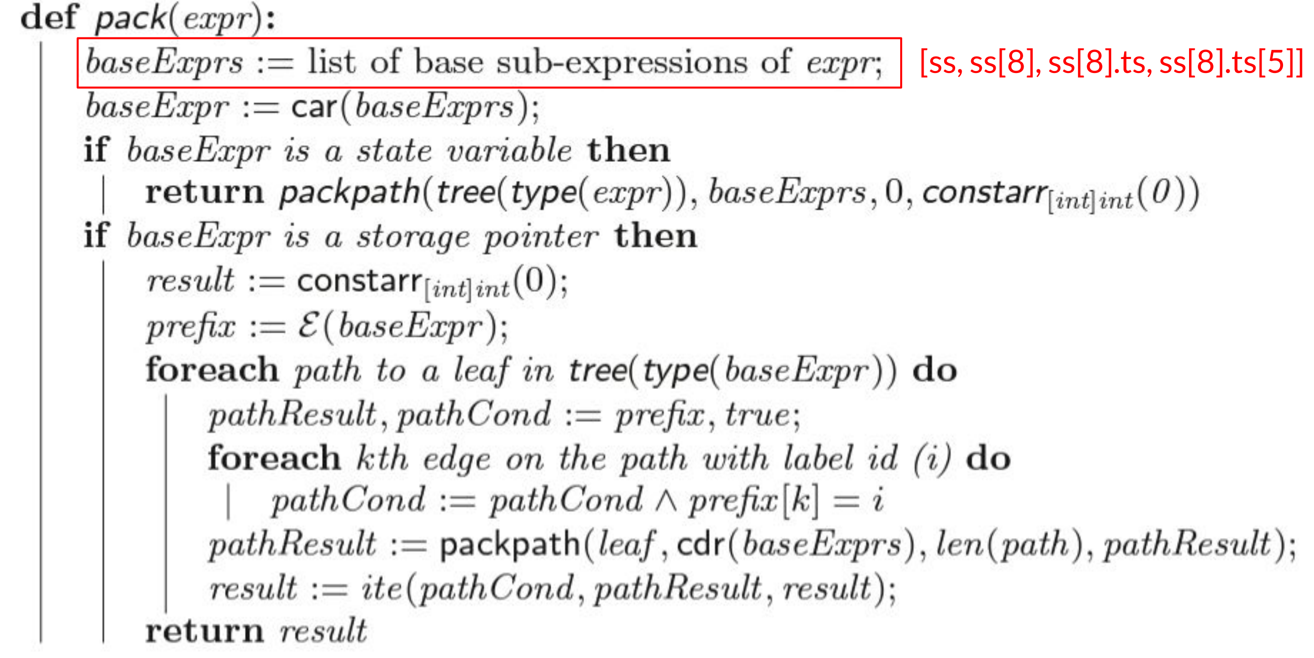
Pack function



```
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, $\mathsf{constarr}_{[int]int}(0)$)
    if baseExpr is a storage pointer then
        result := $\mathsf{constarr}_{[int]int}(0)$;
        prefix := $\mathcal{E}$(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
                $pathCond := pathCond \wedge prefix[k] = i$
            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]

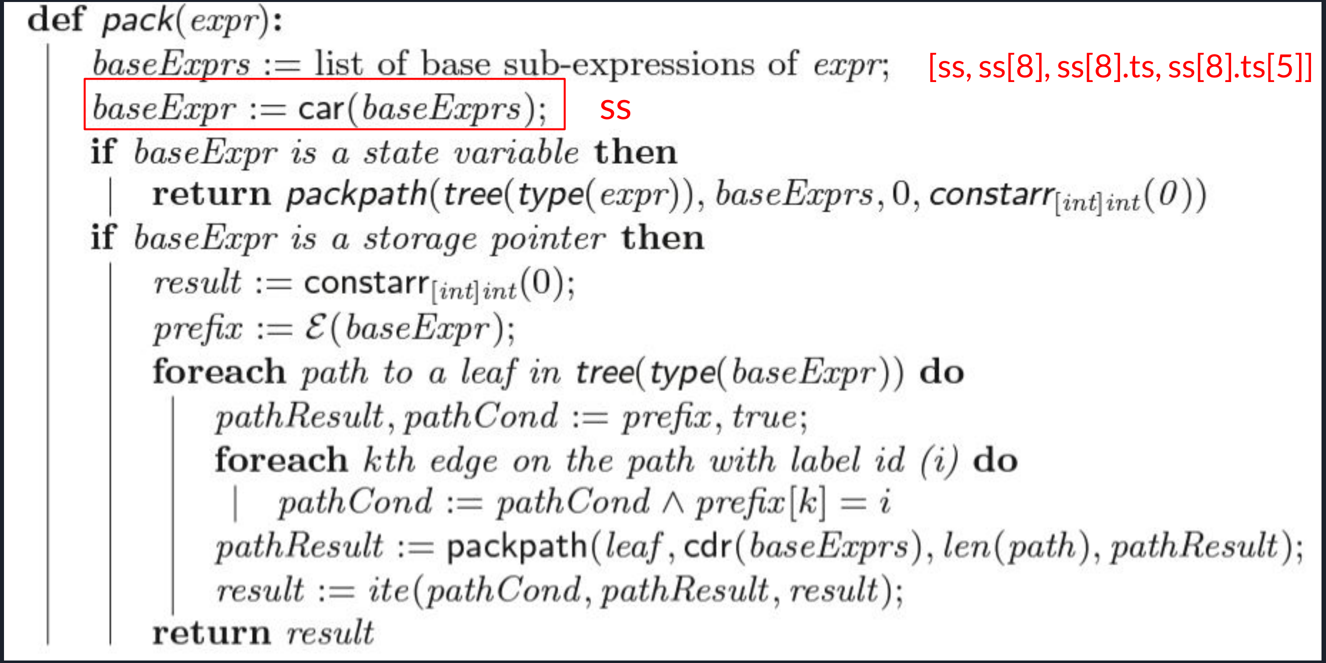


```
def pack(expr):
    baseExprs := list of base sub-expressions of expr;
    baseExpr := car(baseExprs);
    if baseExpr is a state variable then
        return packpath(tree(type(expr)), baseExprs, 0, $\mathsf{constarr}_{[int]\,int}(0)$)
    if baseExpr is a storage pointer then
        result := $\mathsf{constarr}_{[int]\,int}(0)$;
        prefix := $\mathcal{E}$(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 := $pathCond \wedge prefix[k] = i$
            pathResult := packpath(leaf, cdr(baseExprs), len(path), pathResult);
            result := ite(pathCond, pathResult, result);
        return result
```

[ss, ss[8], ss[8].ts, ss[8].ts[5]]

Pack function - run example

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



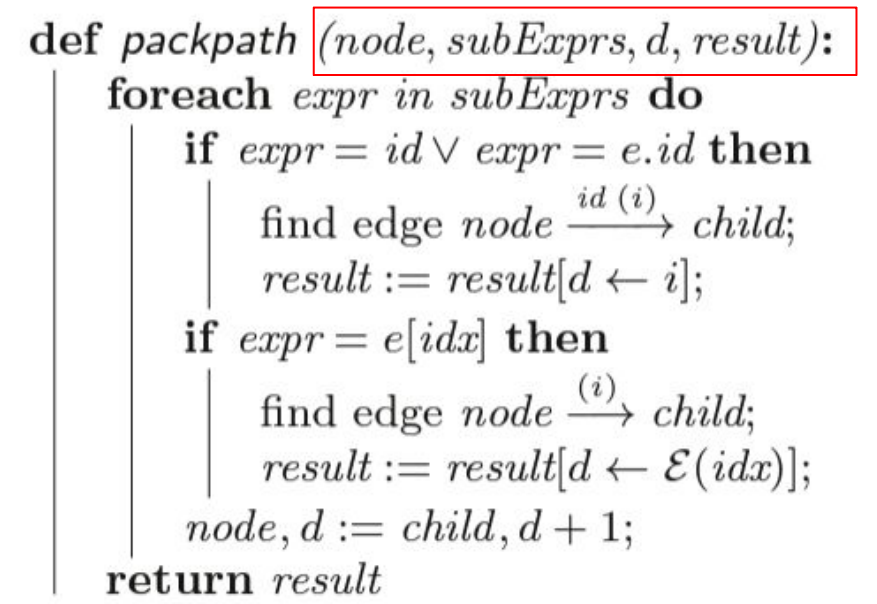
```
def pack(expr):
    baseExprs := list of base sub-expressions of expr;
    baseExpr := car(baseExprs);
    if baseExpr is a state variable then
        return packpath(tree(type(expr)), baseExprs, 0, $constarr_{[int]int}(0)$)
    if baseExpr is a storage pointer then
        result := $constarr_{[int]int}(0)$;
        prefix := $\mathcal{E}$(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 $\wedge$ prefix[k] = i
            pathResult := packpath(leaf, cdr(baseExprs), len(path), pathResult);
            result := ite(pathCond, pathResult, result);
        return result
```

[ss, ss[8], ss[8].ts, ss[8].ts[5]]

ss

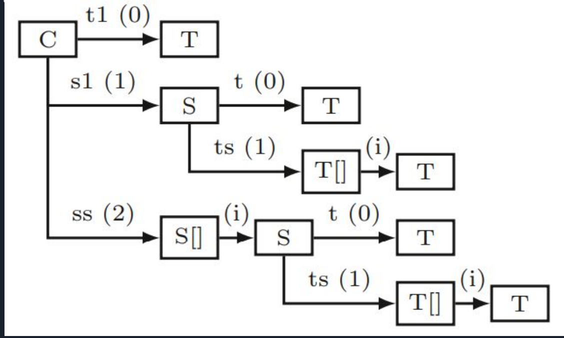
Pack function - run example

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



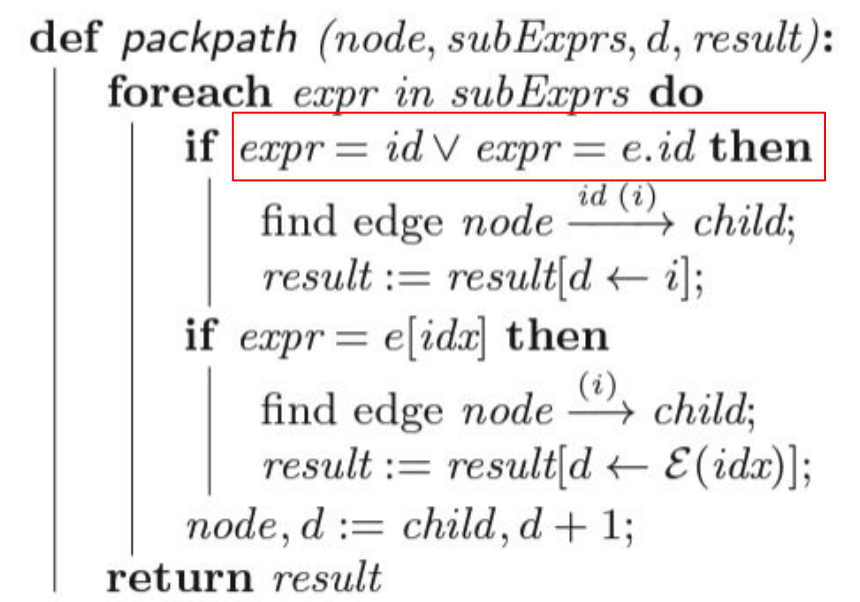
```
def packpath (node, subExprs, d, result):
    foreach expr in subExprs do
        if $expr = id \lor expr = e.id$ then
            find edge $node \xrightarrow{id\ (i)} child$;
            $result := result[d \leftarrow i]$;
        if $expr = e[idx]$ then
            find edge $node \xrightarrow{(i)}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

node = contract (tree)
subExprs =[ss, ss[8], ss[8].ts, ss[8].ts[5]]
d = 0
result = []



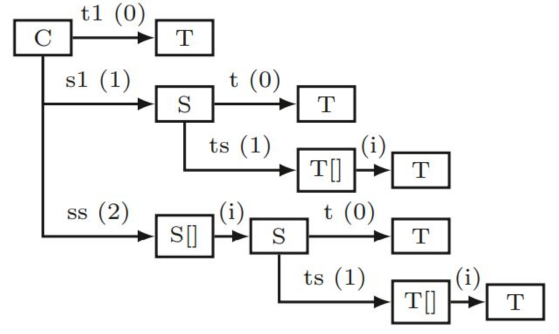
Pack function - run example

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



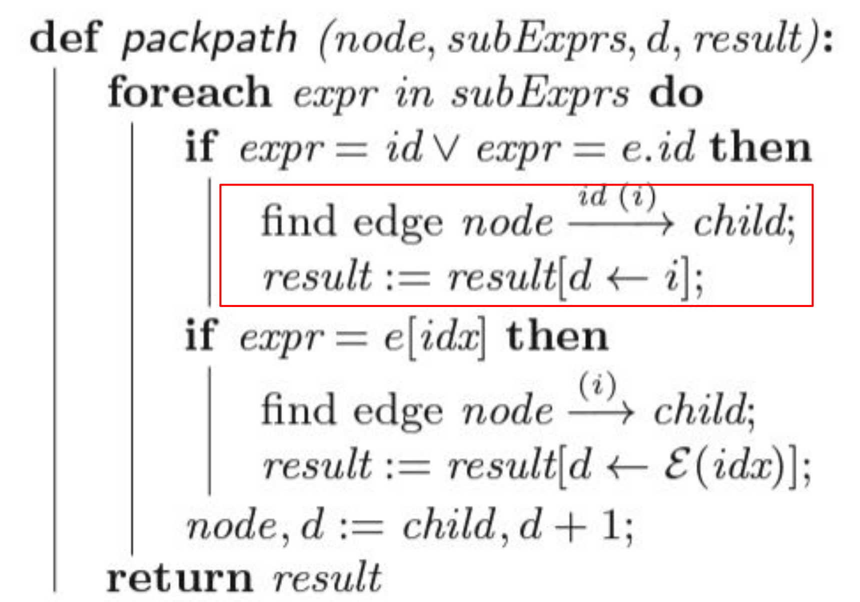
```
def packpath (node, subExprs, d, result):
    foreach expr in subExprs do
        if $\boxed{expr = id \lor expr = e.id \textbf{ then}}$
            find edge $node \xrightarrow{id\ (i)} child$;
            $result := result[d \leftarrow i]$;
        if $expr = e[idx]$ then
            find edge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

node = contract (tree)
subExprs =[ss, ss[8], ss[8].ts, ss[8].ts[5]]
d = 0
result = []
expr = ss



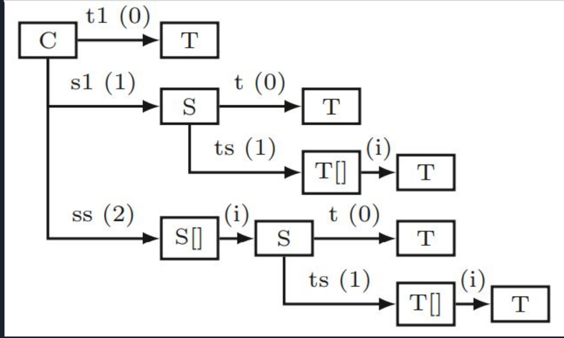
Pack function - run example

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



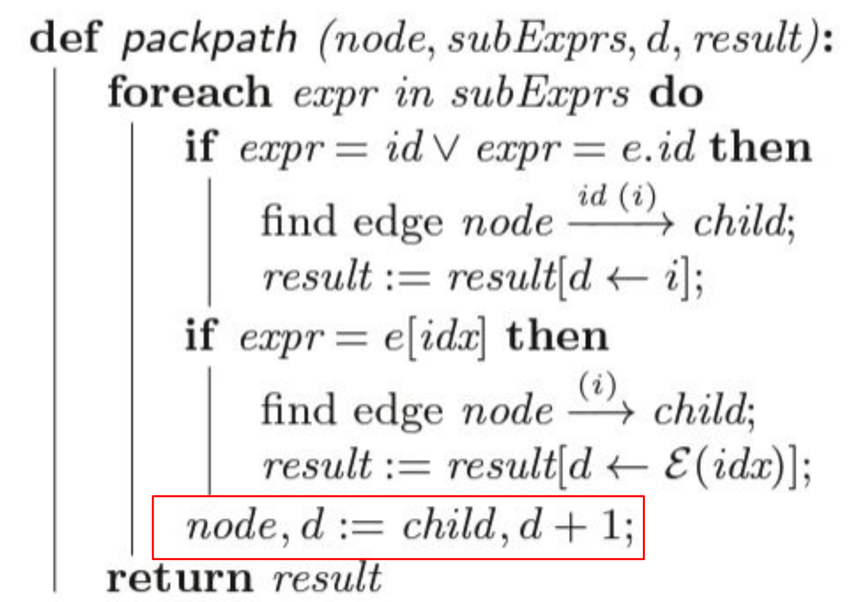
```
def $packpath\ (node, subExprs, d, result)$:
    foreach $expr$ in $subExprs$ do
        if $expr = id \lor expr = e.id$ then
            find edge $node \xrightarrow{id\ (i)} child$;
            $result := result[d \leftarrow i]$;
        if $expr = e[idx]$ then
            find edge $node \xrightarrow{(i)}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

node = contract (tree)
subExprs =[ss, ss[8], ss[8].ts, ss[8].ts[5]]
d = 0
result = [2]
expr = ss
i = 2
child = S[]



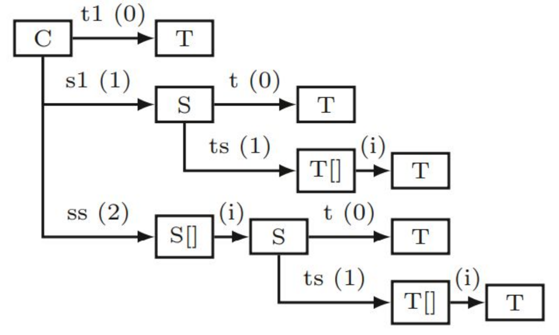
Pack function - run example

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



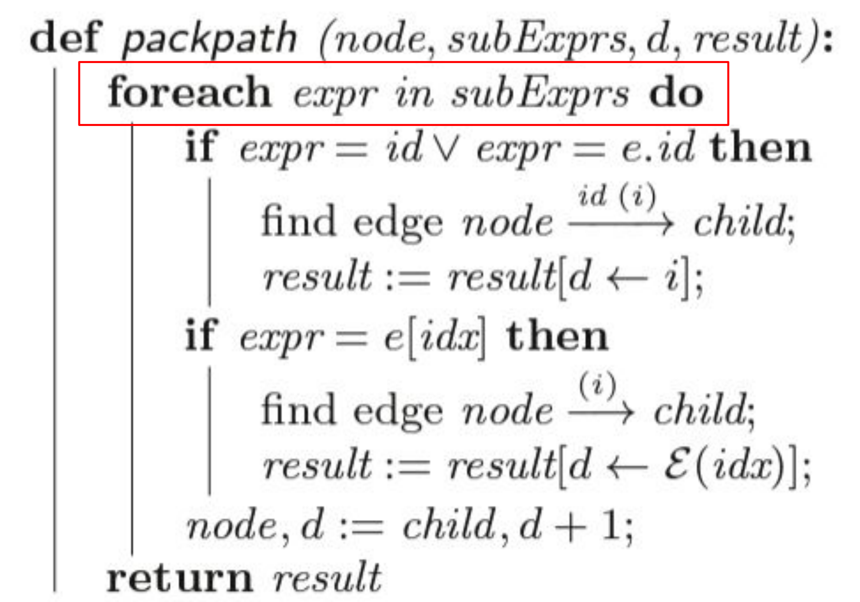
```
def packpath (node, subExprs, d, result):
    foreach expr in subExprs do
        if $expr = id \lor expr = e.id$ then
            find edge $node \xrightarrow{id\ (i)} child$;
            $result := result[d \leftarrow i]$;
        if $expr = e[idx]$ then
            find edge $node \xrightarrow{(i)}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

node = S[]
subExprs =[ss, ss[8], ss[8].ts, ss[8].ts[5]]
d = 1
result = [2]
expr = ss



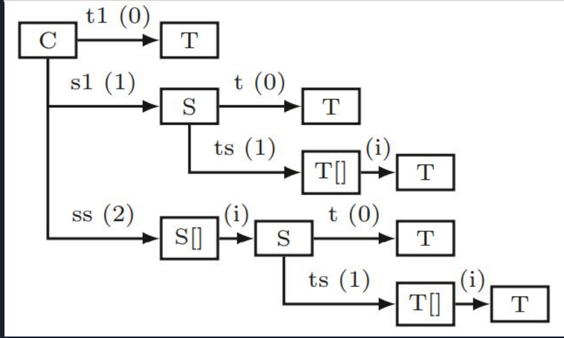
Pack function - run example

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



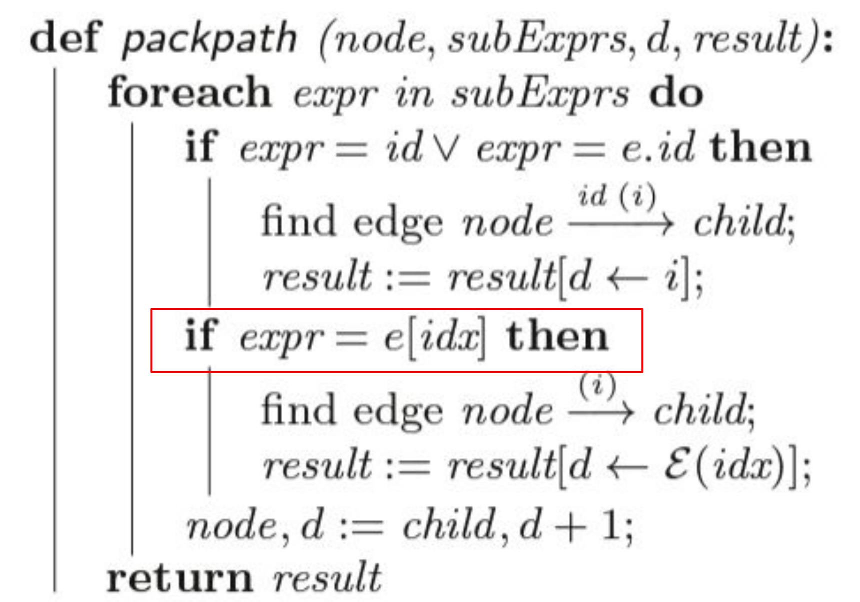
```
def packpath $(node, subExprs, d, result)$:
    foreach $expr$ in $subExprs$ do
        if $expr = id \lor expr = e.id$ then
            find edge $node \xrightarrow{id\ (i)} child$;
            $result := result[d \leftarrow i]$;
        if $expr = e[idx]$ then
            find edge $node \xrightarrow{(i)}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

node = S[]
subExprs =[ss, ss[8], ss[8].ts, ss[8].ts[5]]
d = 1
result = [2]
expr = ss[8]



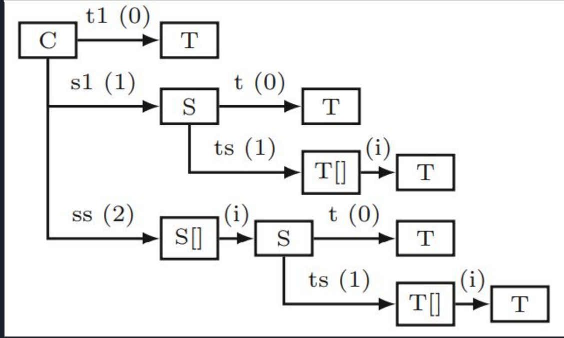
Pack function - run example

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



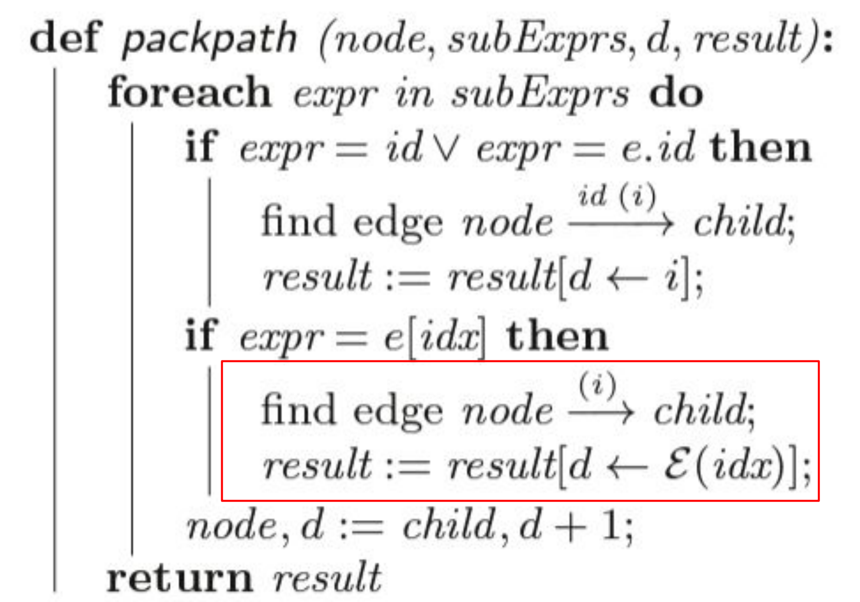
```
def packpath $(node, subExprs, d, result)$:
    foreach $expr$ in $subExprs$ do
        if $expr = id \vee expr = e.id$ then
            find edge $node \xrightarrow{id\ (i)} child$;
            $result := result[d \leftarrow i]$;
        if $expr = e[idx]$ then
            find edge $node \xrightarrow{(i)}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

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



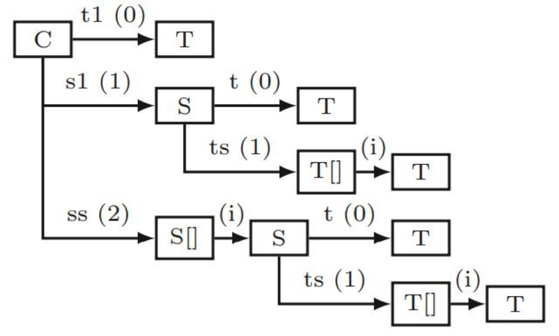
Pack function - run example

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

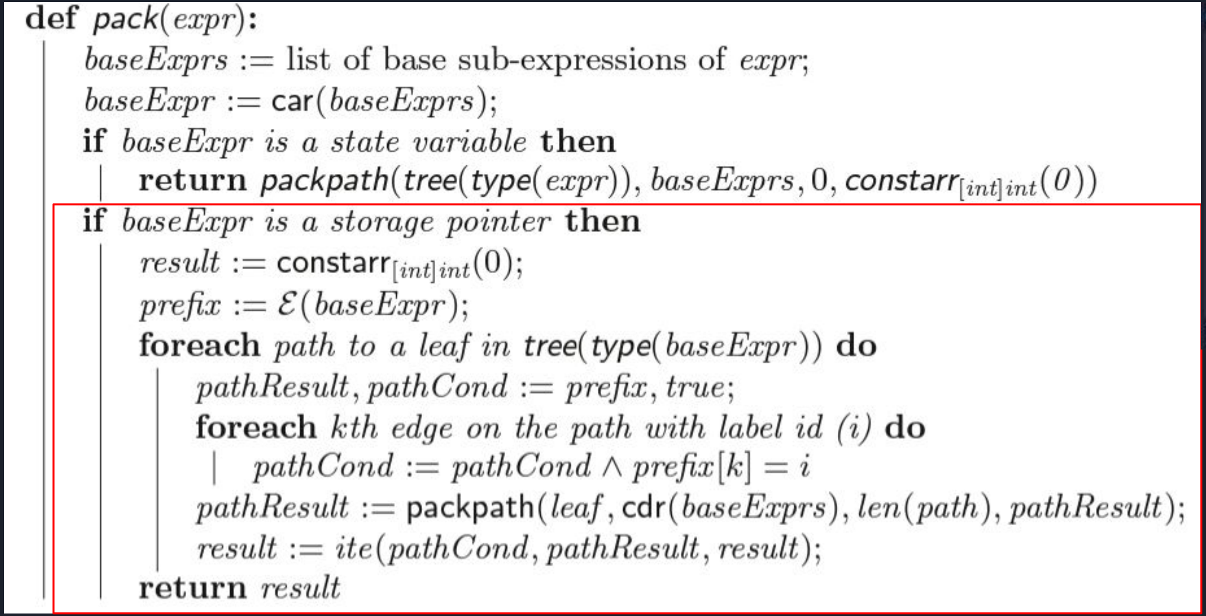


```
def packpath $(node, subExprs, d, result)$:
    foreach $expr$ in $subExprs$ do
        if $expr = id \vee expr = e.id$ then
            find edge $node \xrightarrow{id\ (i)} child$;
            $result := result[d \leftarrow i]$;
        if $expr = e[idx]$ then
            find edge $node \xrightarrow{(i)}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

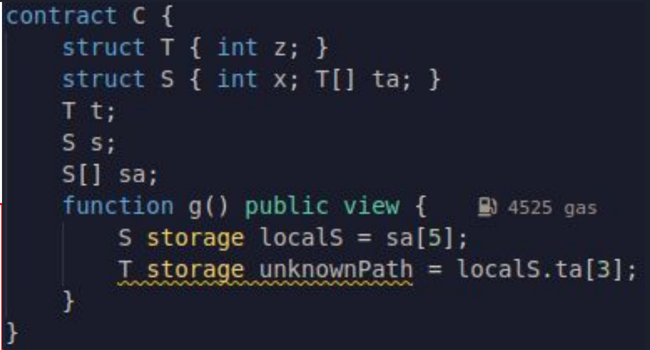
node = S[]
subExprs =[ss, ss[8], ss[8].ts, ss[8].ts[5]]
d = 1
result = [2,8]
expr = ss[8]
(idx = 8)
child = S



Pack function



```
def $pack(expr)$:
    $baseExprs$ := list of base sub-expressions of $expr$;
    $baseExpr := \mathsf{car}(baseExprs)$;
    if $baseExpr$ is a state variable then
        return $\mathit{packpath}(\mathit{tree}(\mathit{type}(expr)), baseExprs, 0, \mathit{constarr}_{[int]int}(0))$
    if $baseExpr$ is a storage pointer then
        $result := \mathsf{constarr}_{[int]int}(0)$;
        $prefix := \mathcal{E}(baseExpr)$;
        foreach path to a leaf in $\mathit{tree}(\mathit{type}(baseExpr))$ do
            $pathResult, pathCond := prefix, true$;
            foreach $k$th edge on the path with label id $(i)$ do
                $pathCond := pathCond \land prefix[k] = i$
            $pathResult := \mathsf{packpath}(leaf, \mathsf{cdr}(baseExprs), len(path), pathResult)$;
            $result := ite(pathCond, pathResult, result)$;
        return $result$
```

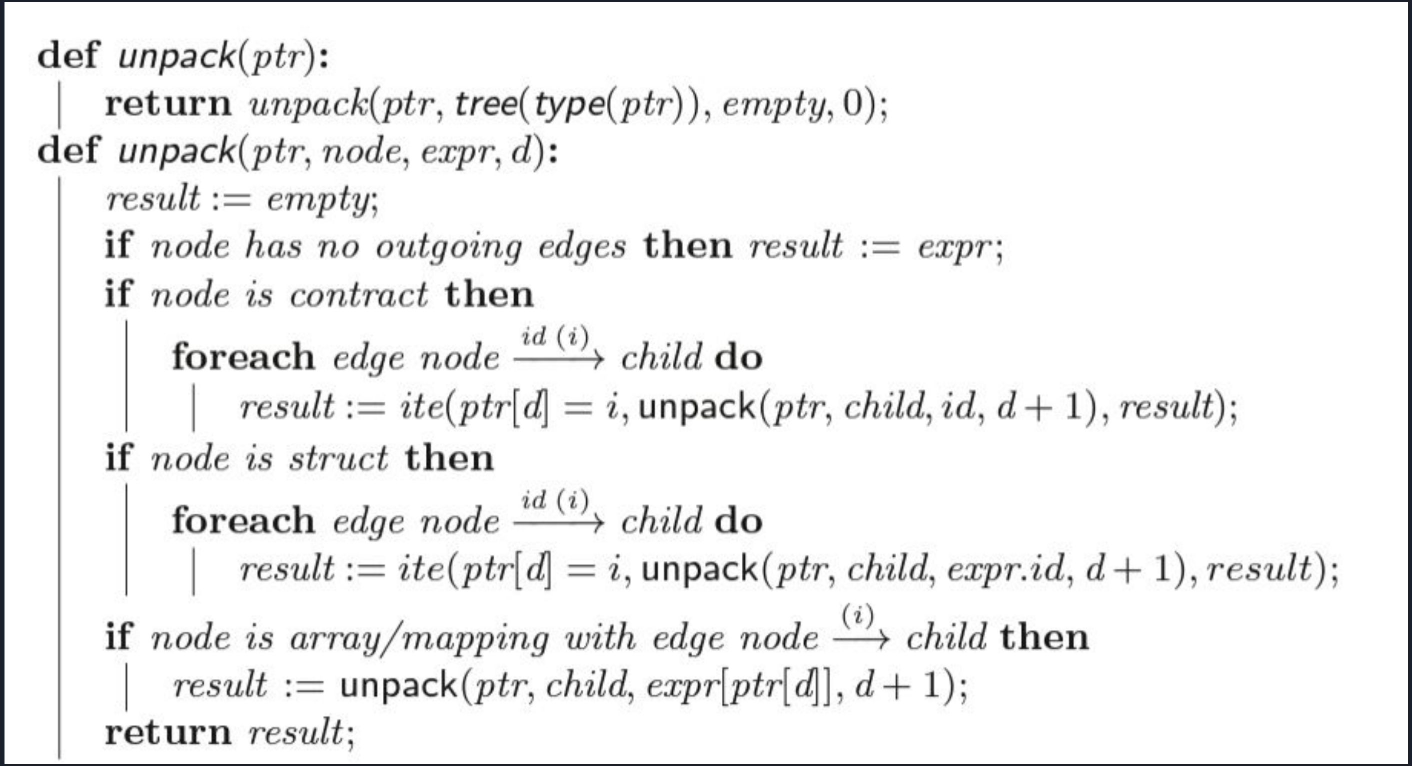


```
contract C {
    struct T { int z; }
    struct S { int x; T[] ta; }
    T t;
    S s;
    S[] sa;
    function g() public view {
        S storage localS = sa[5];
        T storage unknownPath = localS.ta[3];
    }
}
```

Unpack function

- The function takes a storage pointer (of type [int]int) and produces a conditional expression that decodes any given path in to one of the leaves of the storage tree
- The SMT equivalent to dereference

Unpack function



```
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
- Ae add the declaration s_i : T(*type*(s_i) storage)
- Wlog, we assume they are assigned in the constructor

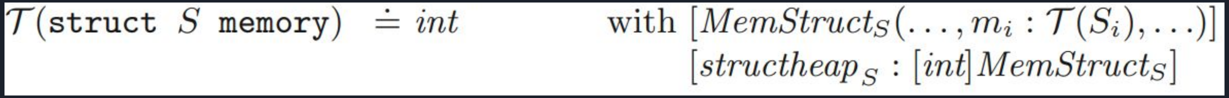
Function calls

- 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

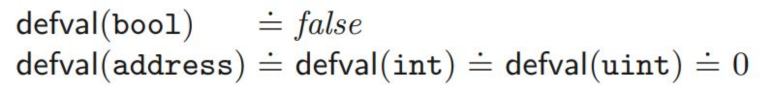
$$\begin{aligned}\mathcal{T}(T\,\texttt{[] memory}) \doteq int \qquad & \text{with } [MemArr_T(arr : [int]\mathcal{T}(T), length : int)] \\ & [arrheap_T : [int]MemArr_T]\end{aligned}$$



$$\begin{aligned}\mathcal{T}(\texttt{struct}\ S\ \texttt{memory}) \doteq int \qquad & \text{with } [MemStruct_S(\ldots, m_i : \mathcal{T}(S_i), \ldots)] \\ & [structheap_S : [int]MemStruct_S]\end{aligned}$$

Default values

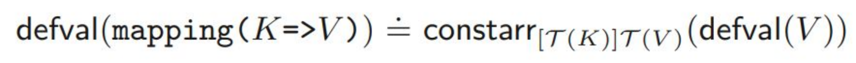
- The defval function maps a solidity type to its default value in smt
- Trivial for value types



$$\begin{aligned}\mathsf{defval}(\texttt{bool}) &\doteq \mathit{false}\\ \mathsf{defval}(\texttt{address}) &\doteq \mathsf{defval}(\texttt{int}) \doteq \mathsf{defval}(\texttt{uint}) \doteq 0\end{aligned}$$

Default values - mappings

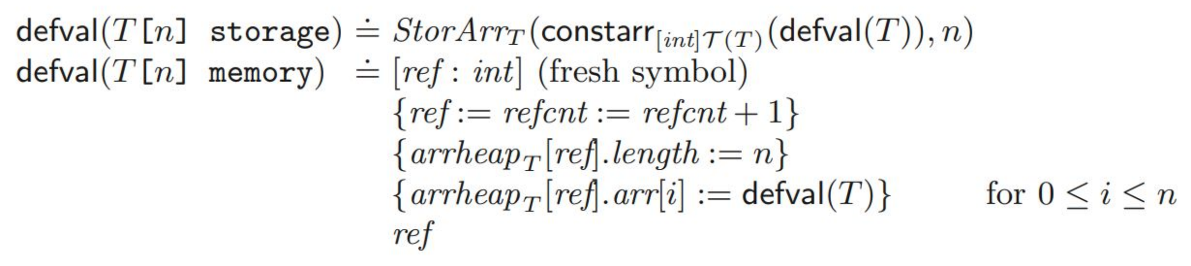
- Mappings can only be stored in storage or storptr



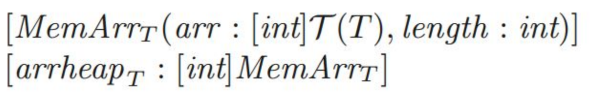
$$\mathsf{defval}(\texttt{mapping}(K\texttt{=>}V)) \doteq \mathsf{constarr}_{[\mathcal{T}(K)]\mathcal{T}(V)}(\mathsf{defval}(V))$$

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



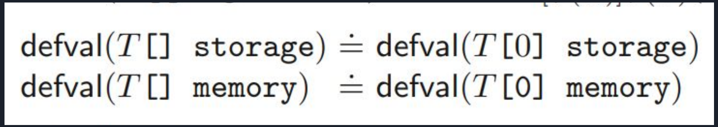
$$\begin{aligned}
\mathsf{defval}(T\,[n]\ \texttt{storage}) &\doteq \mathit{StorArr}_T(\mathsf{constarr}_{[int]\mathcal{T}(T)}(\mathsf{defval}(T)), n)\\
\mathsf{defval}(T\,[n]\ \texttt{memory}) &\doteq [\mathit{ref} : \mathit{int}]\ \text{(fresh symbol)}\\
&\quad \{\mathit{ref} := \mathit{refcnt} := \mathit{refcnt} + 1\}\\
&\quad \{\mathit{arrheap}_T[\mathit{ref}].\mathit{length} := n\}\\
&\quad \{\mathit{arrheap}_T[\mathit{ref}].\mathit{arr}[i] := \mathsf{defval}(T)\} \qquad \text{for } 0 \leq i \leq n\\
&\quad \mathit{ref}
\end{aligned}$$



$$\begin{aligned}
&[\mathit{MemArr}_T(\mathit{arr} : [\mathit{int}]\mathcal{T}(T), \mathit{length} : \mathit{int})]\\
&[\mathit{arrheap}_T : [\mathit{int}]\mathit{MemArr}_T]
\end{aligned}$$

Default values - dynamic arrays

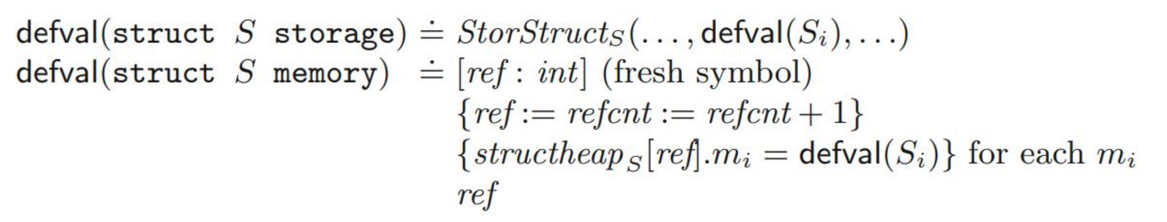
- Initialized as a 0 length fixed size array



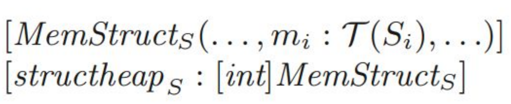
$$
\begin{aligned}
\mathsf{defval}(T\,\texttt{[] storage}) &\doteq \mathsf{defval}(T\,\texttt{[0] storage})\\
\mathsf{defval}(T\,\texttt{[] memory}) &\doteq \mathsf{defval}(T\,\texttt{[0] memory})
\end{aligned}
$$

Default values - structs

- Similar to arrays
- Initialization can be done without loops



$$\begin{aligned}
\mathsf{defval}(\texttt{struct}\ S\ \texttt{storage}) &\doteq \mathit{StorStruct}_S(\ldots, \mathsf{defval}(S_i), \ldots)\\
\mathsf{defval}(\texttt{struct}\ S\ \texttt{memory}) &\doteq [\mathit{ref} : \mathit{int}]\ \text{(fresh symbol)}\\
&\ \{\mathit{ref} := \mathit{refcnt} := \mathit{refcnt} + 1\}\\
&\ \{\mathit{structheap}_S[\mathit{ref}].m_i = \mathsf{defval}(S_i)\}\ \text{for each } m_i\\
&\ \mathit{ref}
\end{aligned}$$



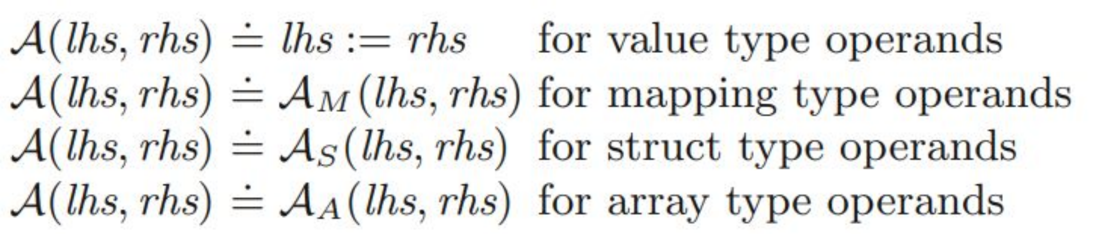
$$\begin{aligned}
&[\mathit{MemStruct}_S(\ldots, m_i : \mathcal{T}(S_i), \ldots)]\\
&[\mathit{structheap}_S : [\mathit{int}]\mathit{MemStruct}_S]
\end{aligned}$$

Agenda

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

The A(.,.) function

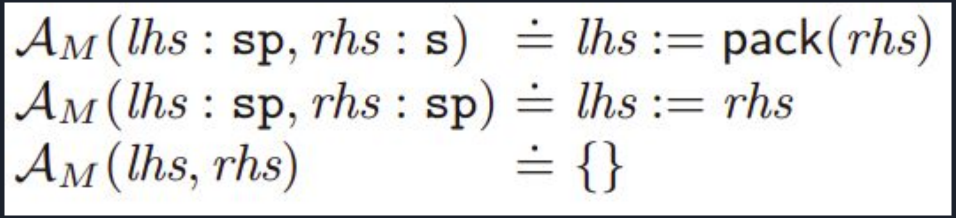
- Reference type assignments can be either pointer assignments or value assignments
    - value assignments can create new allocations
- A(lhs,rhs) denotes assigning rhs to lhs as SMT expressions
- Value type assignments are simple to convert to smt



$$\begin{array}{lll}
\mathcal{A}(lhs, rhs) & \doteq lhs := rhs & \text{for value type operands} \\
\mathcal{A}(lhs, rhs) & \doteq \mathcal{A}_M(lhs, rhs) & \text{for mapping type operands} \\
\mathcal{A}(lhs, rhs) & \doteq \mathcal{A}_S(lhs, rhs) & \text{for struct type operands} \\
\mathcal{A}(lhs, rhs) & \doteq \mathcal{A}_A(lhs, rhs) & \text{for array type operands}
\end{array}$$

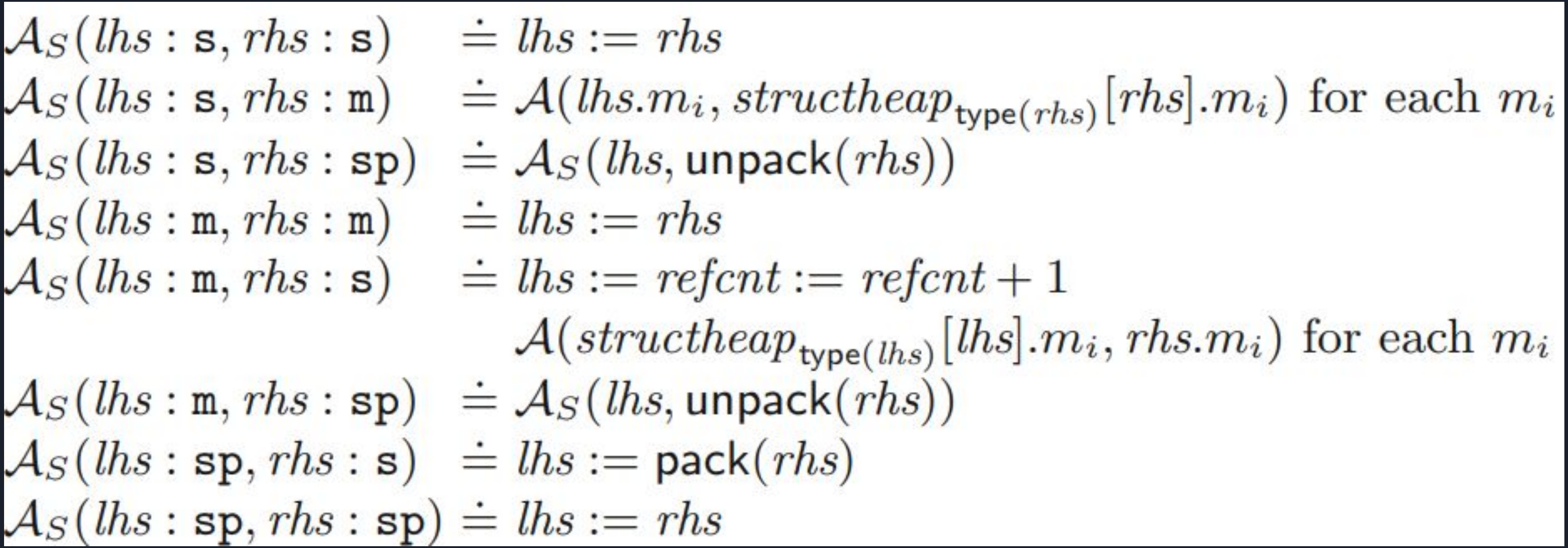
Mappings

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



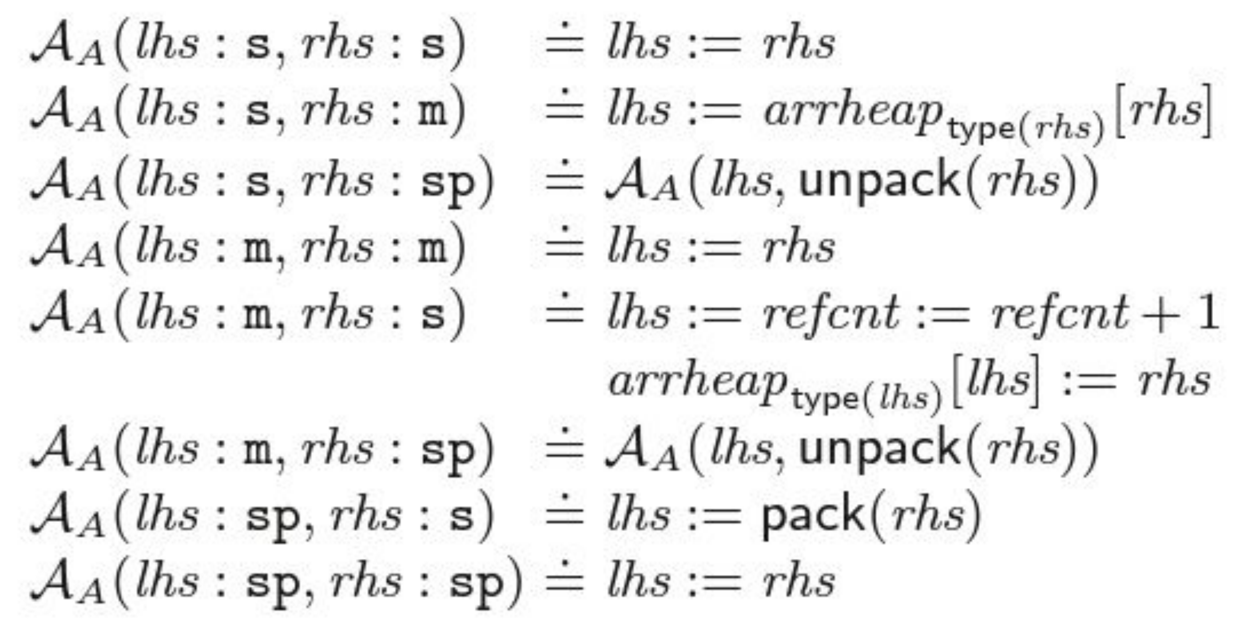
$$\begin{aligned}
\mathcal{A}_M(lhs : \mathsf{sp}, rhs : \mathsf{s}) &\doteq lhs := \mathsf{pack}(rhs)\\
\mathcal{A}_M(lhs : \mathsf{sp}, rhs : \mathsf{sp}) &\doteq lhs := rhs\\
\mathcal{A}_M(lhs, rhs) &\doteq \{\}
\end{aligned}$$

Structs



$$
\begin{aligned}
\mathcal{A}_S(\mathit{lhs} : \mathtt{s}, \mathit{rhs} : \mathtt{s}) &\doteq \mathit{lhs} := \mathit{rhs} \\
\mathcal{A}_S(\mathit{lhs} : \mathtt{s}, \mathit{rhs} : \mathtt{m}) &\doteq \mathcal{A}(\mathit{lhs}.m_i, \mathit{structheap}_{\mathsf{type}(\mathit{rhs})}[\mathit{rhs}].m_i) \text{ for each } m_i \\
\mathcal{A}_S(\mathit{lhs} : \mathtt{s}, \mathit{rhs} : \mathsf{sp}) &\doteq \mathcal{A}_S(\mathit{lhs}, \mathsf{unpack}(\mathit{rhs})) \\
\mathcal{A}_S(\mathit{lhs} : \mathtt{m}, \mathit{rhs} : \mathtt{m}) &\doteq \mathit{lhs} := \mathit{rhs} \\
\mathcal{A}_S(\mathit{lhs} : \mathtt{m}, \mathit{rhs} : \mathtt{s}) &\doteq \mathit{lhs} := \mathit{refcnt} := \mathit{refcnt} + 1 \\
&\quad\; \mathcal{A}(\mathit{structheap}_{\mathsf{type}(\mathit{lhs})}[\mathit{lhs}].m_i, \mathit{rhs}.m_i) \text{ for each } m_i \\
\mathcal{A}_S(\mathit{lhs} : \mathtt{m}, \mathit{rhs} : \mathsf{sp}) &\doteq \mathcal{A}_S(\mathit{lhs}, \mathsf{unpack}(\mathit{rhs})) \\
\mathcal{A}_S(\mathit{lhs} : \mathsf{sp}, \mathit{rhs} : \mathtt{s}) &\doteq \mathit{lhs} := \mathsf{pack}(\mathit{rhs}) \\
\mathcal{A}_S(\mathit{lhs} : \mathsf{sp}, \mathit{rhs} : \mathsf{sp}) &\doteq \mathit{lhs} := \mathit{rhs}
\end{aligned}
$$

Arrays



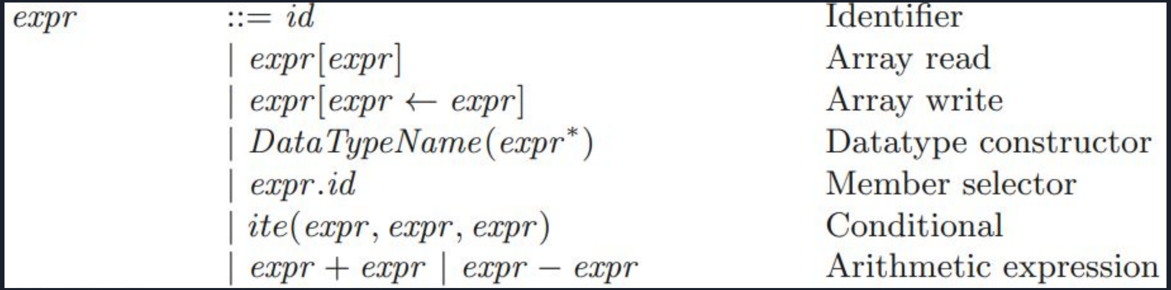
$$
\begin{aligned}
\mathcal{A}_A(lhs:\mathtt{s}, rhs:\mathtt{s}) &\doteq lhs := rhs\\
\mathcal{A}_A(lhs:\mathtt{s}, rhs:\mathtt{m}) &\doteq lhs := arrheap_{\mathsf{type}(rhs)}[rhs]\\
\mathcal{A}_A(lhs:\mathtt{s}, rhs:\mathtt{sp}) &\doteq \mathcal{A}_A(lhs, \mathsf{unpack}(rhs))\\
\mathcal{A}_A(lhs:\mathtt{m}, rhs:\mathtt{m}) &\doteq lhs := rhs\\
\mathcal{A}_A(lhs:\mathtt{m}, rhs:\mathtt{s}) &\doteq lhs := refcnt := refcnt + 1\\
&\; arrheap_{\mathsf{type}(lhs)}[lhs] := rhs\\
\mathcal{A}_A(lhs:\mathtt{m}, rhs:\mathtt{sp}) &\doteq \mathcal{A}_A(lhs, \mathsf{unpack}(rhs))\\
\mathcal{A}_A(lhs:\mathtt{sp}, rhs:\mathtt{s}) &\doteq lhs := \mathsf{pack}(rhs)\\
\mathcal{A}_A(lhs:\mathtt{sp}, rhs:\mathtt{sp}) &\doteq lhs := rhs
\end{aligned}
$$

Agenda

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

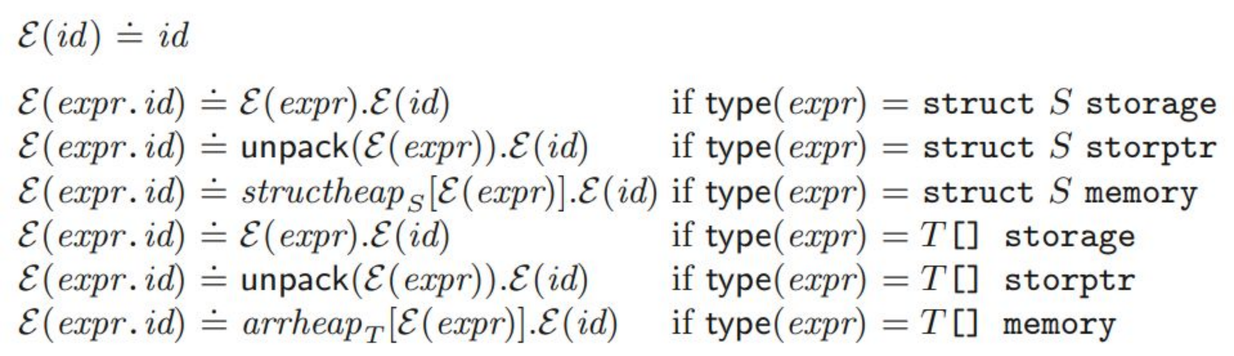
The ε(.) function

- Translates a Solidity expression to an SMT expression
- Can introduce side effects (declarations and statements)



$expr$	$::= id$	Identifier
	$\mid expr[expr]$	Array read
	$\mid expr[expr \leftarrow expr]$	Array write
	$\mid DataTypeName(expr^*)$	Datatype constructor
	$\mid expr.id$	Member selector
	$\mid ite(expr, expr, expr)$	Conditional
	$\mid expr + expr \mid expr - expr$	Arithmetic expression

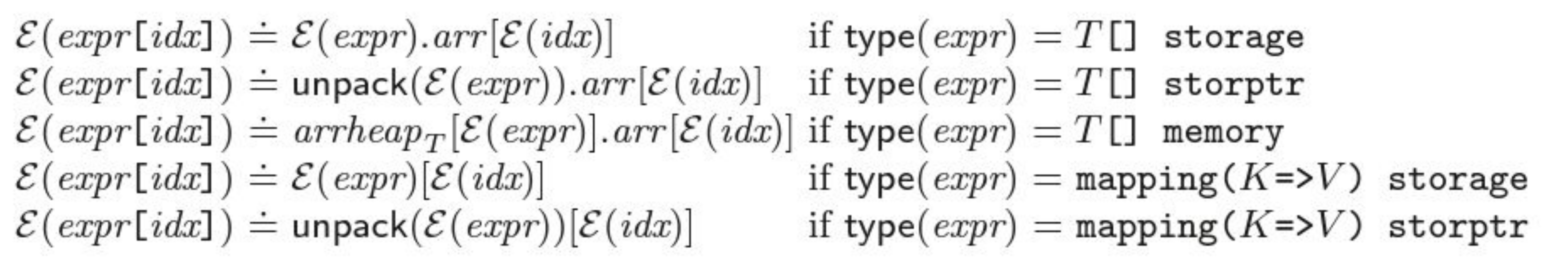
The ε(.) function - member access



$$\mathcal{E}(id) \doteq id$$

$$\begin{aligned}
\mathcal{E}(expr.id) &\doteq \mathcal{E}(expr).\mathcal{E}(id) && \text{if } \mathsf{type}(expr) = \texttt{struct } S \texttt{ storage} \\
\mathcal{E}(expr.id) &\doteq \mathsf{unpack}(\mathcal{E}(expr)).\mathcal{E}(id) && \text{if } \mathsf{type}(expr) = \texttt{struct } S \texttt{ storptr} \\
\mathcal{E}(expr.id) &\doteq structheap_S[\mathcal{E}(expr)].\mathcal{E}(id) && \text{if } \mathsf{type}(expr) = \texttt{struct } S \texttt{ memory} \\
\mathcal{E}(expr.id) &\doteq \mathcal{E}(expr).\mathcal{E}(id) && \text{if } \mathsf{type}(expr) = T\texttt{[] storage} \\
\mathcal{E}(expr.id) &\doteq \mathsf{unpack}(\mathcal{E}(expr)).\mathcal{E}(id) && \text{if } \mathsf{type}(expr) = T\texttt{[] storptr} \\
\mathcal{E}(expr.id) &\doteq arrheap_T[\mathcal{E}(expr)].\mathcal{E}(id) && \text{if } \mathsf{type}(expr) = T\texttt{[] memory}
\end{aligned}$$

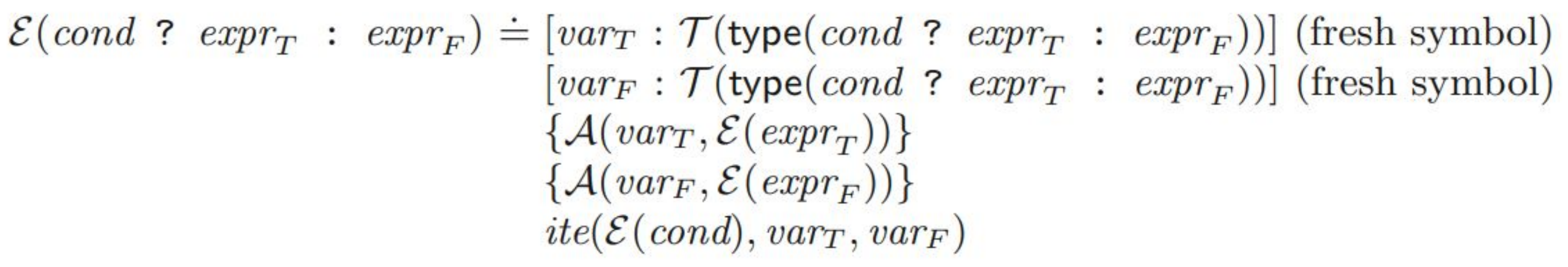
The ε(.) function - index access



$$
\begin{array}{lll}
\mathcal{E}(expr\texttt{[}idx\texttt{]}) \doteq \mathcal{E}(expr).arr[\mathcal{E}(idx)] & \text{if } \mathsf{type}(expr) = T\texttt{[]} & \texttt{storage} \\
\mathcal{E}(expr\texttt{[}idx\texttt{]}) \doteq \mathsf{unpack}(\mathcal{E}(expr)).arr[\mathcal{E}(idx)] & \text{if } \mathsf{type}(expr) = T\texttt{[]} & \texttt{storptr} \\
\mathcal{E}(expr\texttt{[}idx\texttt{]}) \doteq arrheap_T[\mathcal{E}(expr)].arr[\mathcal{E}(idx)] & \text{if } \mathsf{type}(expr) = T\texttt{[]} & \texttt{memory} \\
\mathcal{E}(expr\texttt{[}idx\texttt{]}) \doteq \mathcal{E}(expr)[\mathcal{E}(idx)] & \text{if } \mathsf{type}(expr) = \texttt{mapping(}K\texttt{=>}V\texttt{)} & \texttt{storage} \\
\mathcal{E}(expr\texttt{[}idx\texttt{]}) \doteq \mathsf{unpack}(\mathcal{E}(expr))[\mathcal{E}(idx)] & \text{if } \mathsf{type}(expr) = \texttt{mapping(}K\texttt{=>}V\texttt{)} & \texttt{storptr}
\end{array}
$$

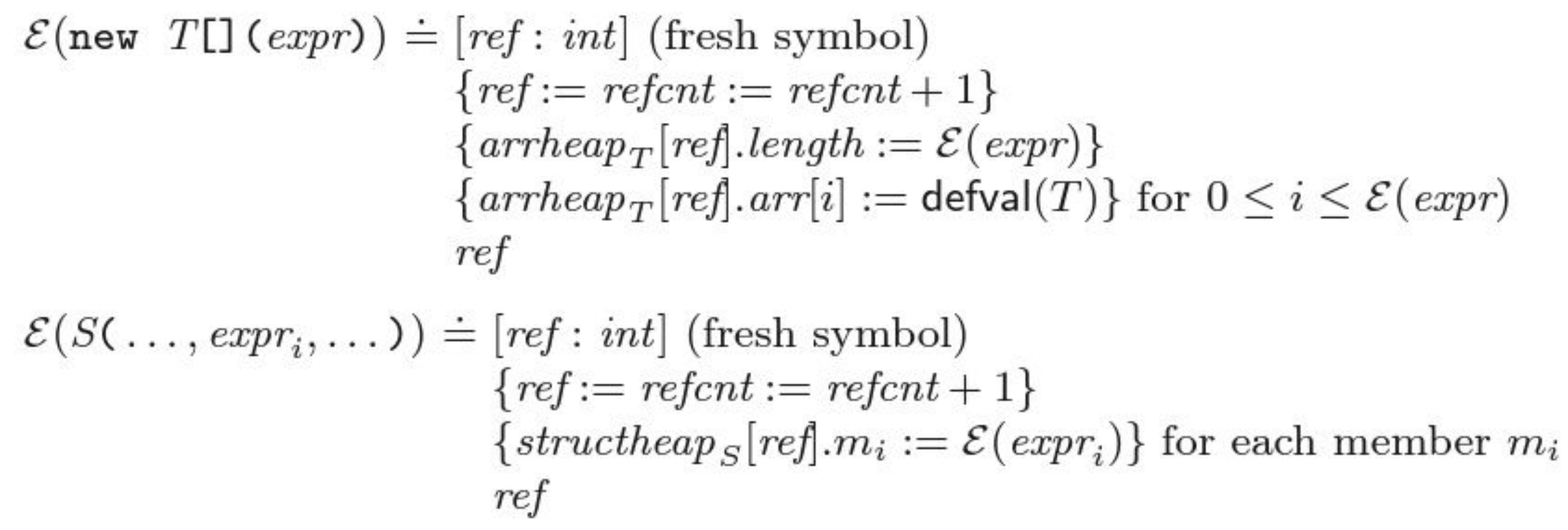
The ε(.) function - conditionals

- 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



$$
\begin{aligned}
\mathcal{E}(cond \text{ ? } expr_T \text{ : } expr_F) \doteq\ & [var_T : \mathcal{T}(\mathsf{type}(cond \text{ ? } expr_T \text{ : } expr_F))] \text{ (fresh symbol)} \\
& [var_F : \mathcal{T}(\mathsf{type}(cond \text{ ? } expr_T \text{ : } expr_F))] \text{ (fresh symbol)} \\
& \{\mathcal{A}(var_T, \mathcal{E}(expr_T))\} \\
& \{\mathcal{A}(var_F, \mathcal{E}(expr_F))\} \\
& ite(\mathcal{E}(cond), var_T, var_F)
\end{aligned}
$$

The ε(.) function - memory allocation



$$\begin{aligned}
\mathcal{E}(\texttt{new}\ \ T\texttt{[]}\,\texttt{(}expr\texttt{)}) \doteq\ & [ref : int]\ \text{(fresh symbol)}\\
& \{ref := refcnt := refcnt + 1\}\\
& \{arrheap_T[ref].length := \mathcal{E}(expr)\}\\
& \{arrheap_T[ref].arr[i] := \mathsf{defval}(T)\}\ \text{for}\ 0 \leq i \leq \mathcal{E}(expr)\\
& ref
\end{aligned}$$

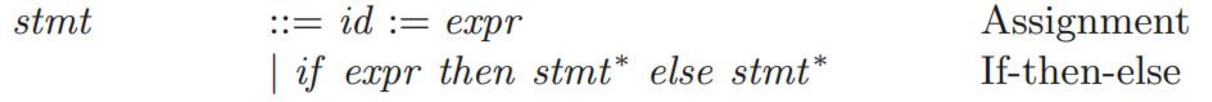
$$\begin{aligned}
\mathcal{E}(S\texttt{(}\ldots, expr_i, \ldots\texttt{)}) \doteq\ & [ref : int]\ \text{(fresh symbol)}\\
& \{ref := refcnt := refcnt + 1\}\\
& \{structheap_S[ref].m_i := \mathcal{E}(expr_i)\}\ \text{for each member}\ m_i\\
& ref
\end{aligned}$$

Agenda

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

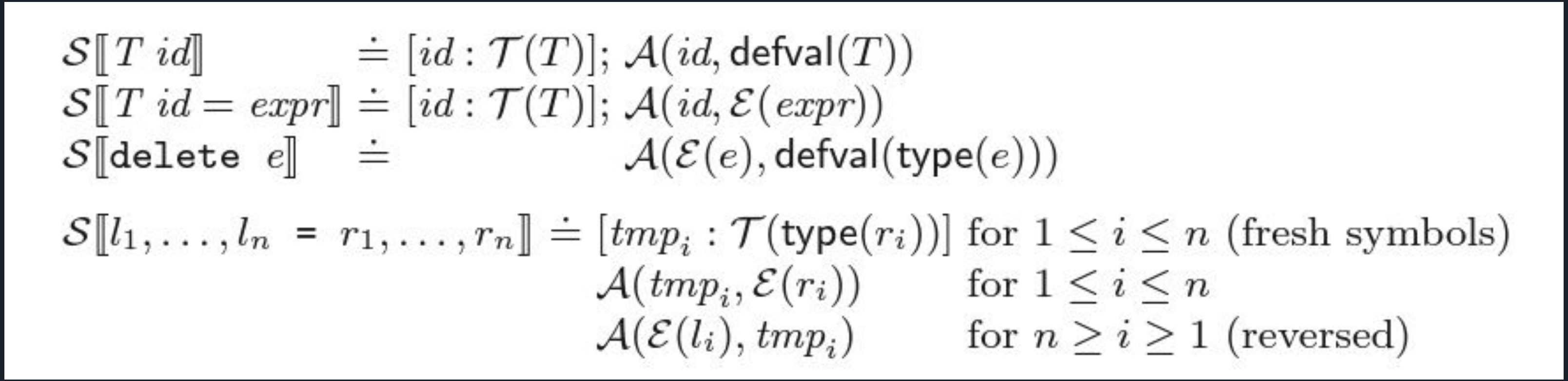
The S[.] function

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



stmt	$::= id := expr$	Assignment
	$\mid$ *if expr then* $stmt^*$ *else* $stmt^*$	If-then-else

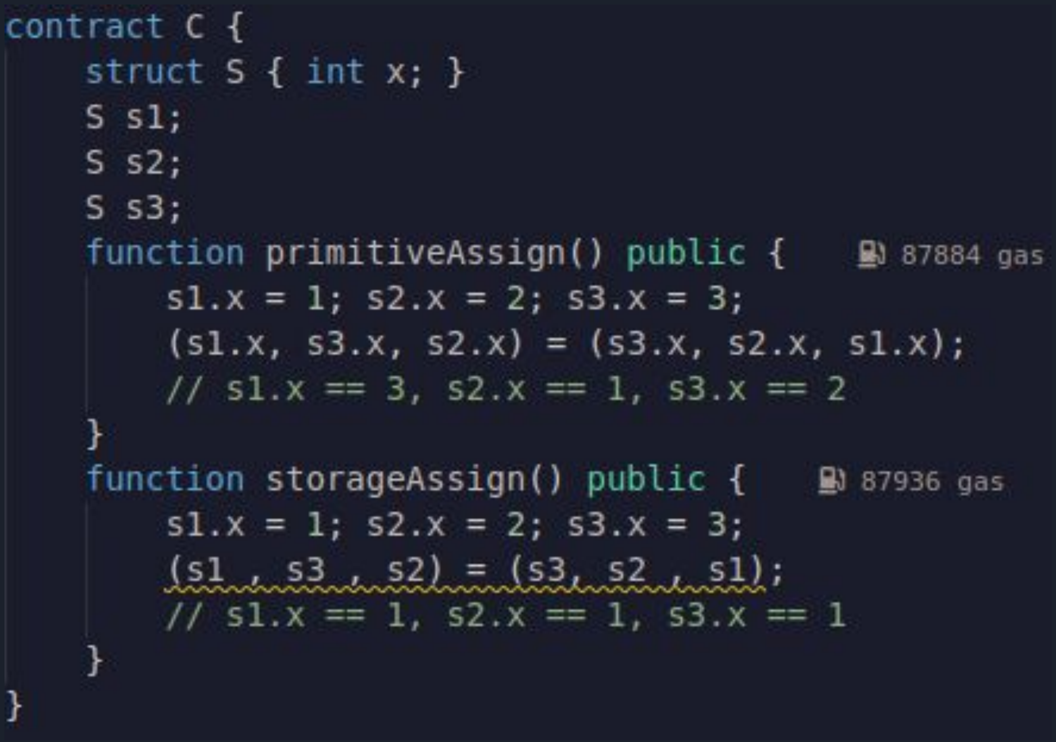
The S[.] function



$$
\begin{aligned}
\mathcal{S}[\![T\ id]\!] &\doteq [id : \mathcal{T}(T)];\ \mathcal{A}(id, \mathsf{defval}(T))\\
\mathcal{S}[\![T\ id = expr]\!] &\doteq [id : \mathcal{T}(T)];\ \mathcal{A}(id, \mathcal{E}(expr))\\
\mathcal{S}[\![\texttt{delete}\ \ e]\!] &\doteq \qquad\qquad\qquad\ \ \mathcal{A}(\mathcal{E}(e), \mathsf{defval}(\mathsf{type}(e)))
\end{aligned}
$$

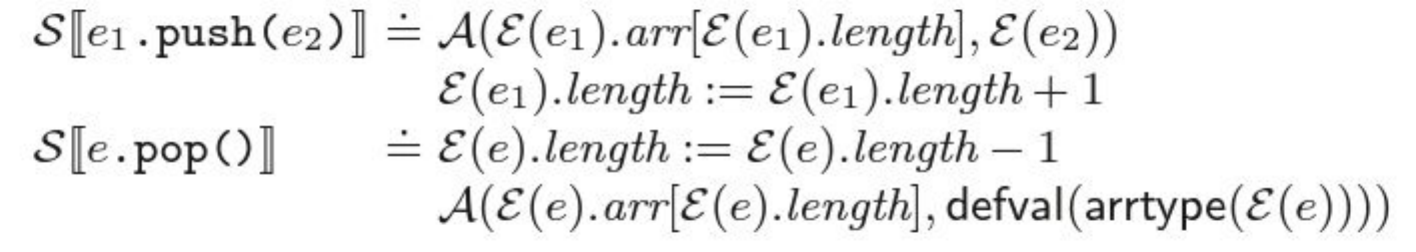
$$
\begin{array}{lll}
\mathcal{S}[\![l_1, \ldots, l_n\ \texttt{=}\ r_1, \ldots, r_n]\!] \doteq & [tmp_i : \mathcal{T}(\mathsf{type}(r_i))] & \text{for } 1 \leq i \leq n \text{ (fresh symbols)}\\
 & \mathcal{A}(tmp_i, \mathcal{E}(r_i)) & \text{for } 1 \leq i \leq n\\
 & \mathcal{A}(\mathcal{E}(l_i), tmp_i) & \text{for } n \geq i \geq 1 \text{ (reversed)}
\end{array}
$$

Reverse assignment example



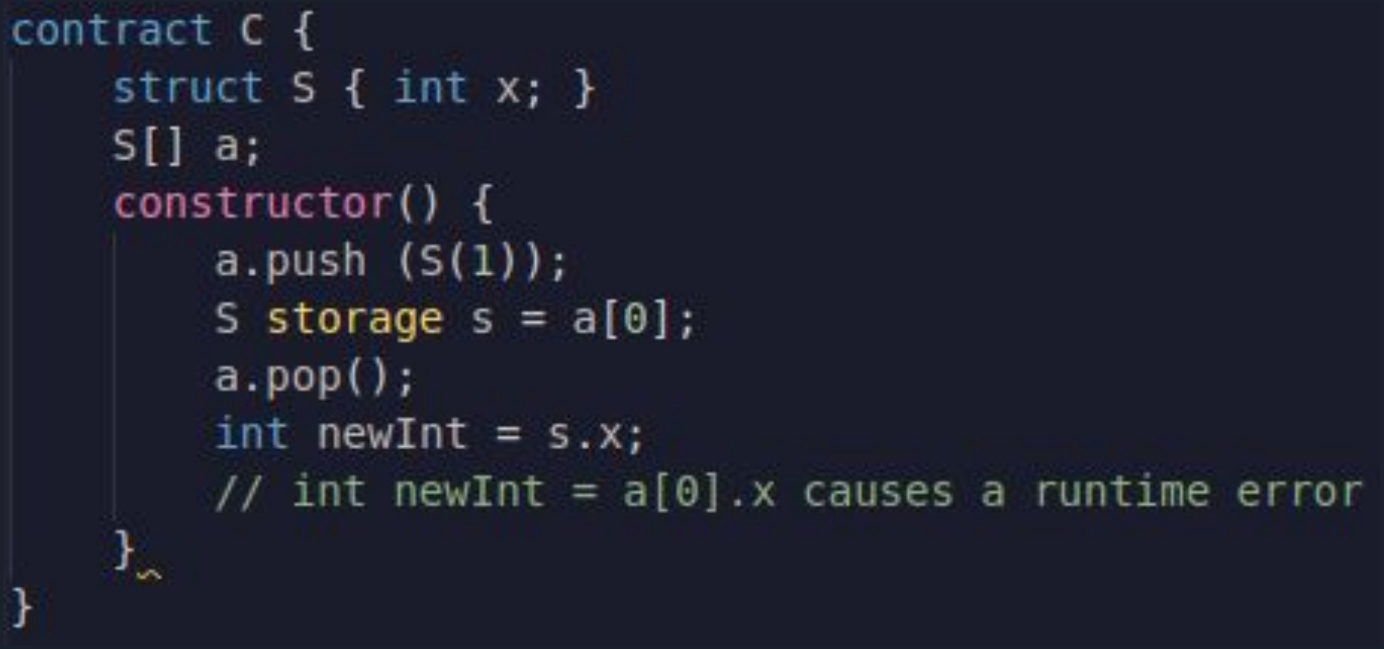
```
contract C {
    struct S { int x; }
    S s1;
    S s2;
    S s3;
    function primitiveAssign() public {    87884 gas
        s1.x = 1; s2.x = 2; s3.x = 3;
        (s1.x, s3.x, s2.x) = (s3.x, s2.x, s1.x);
        // s1.x == 3, s2.x == 1, s3.x == 2
    }
    function storageAssign() public {    87936 gas
        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



$$\begin{aligned}
\mathcal{S}[\![e_1\,\texttt{.push(}e_2\texttt{)}]\!] &\doteq \mathcal{A}(\mathcal{E}(e_1).arr[\mathcal{E}(e_1).length], \mathcal{E}(e_2)) \\
&\ \mathcal{E}(e_1).length := \mathcal{E}(e_1).length + 1 \\
\mathcal{S}[\![e\,\texttt{.pop()}]\!] &\doteq \mathcal{E}(e).length := \mathcal{E}(e).length - 1 \\
&\ \mathcal{A}(\mathcal{E}(e).arr[\mathcal{E}(e).length], \mathsf{defval}(\mathsf{arrtype}(\mathcal{E}(e))))
\end{aligned}$$

Dangling pointer example



```
contract 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?

⭐⭐