Memory abstraction 1

MPRI — Cours 2.6 “Interprétation abstraite : application à la vérification et à l’analyse statique”

Xavier Rival

INRIA, ENS, CNRS

Jan, 30th. 2023
Overview of the lecture

So far, we have shown **numerical abstract domains**
- non relational: intervals, congruences...
- relational: polyhedra, octagons, ellipsoids...

How to deal with non purely numerical states ?
How to reason about complex data-structures ?

⇒ **a very broad topic**, and two lectures:

This lecture
- overview memory models and memory properties
- non relational **pointer structures abstraction**
- **predicates based shape abstraction**

Next lecture: separation logic and shape abstraction, shape/numerical abstraction
Outline

1. Memory models
   - Towards memory properties
   - Formalizing concrete memory states
   - Treatment of errors
   - Language semantics

2. Pointer Abstractions

3. Shape analysis in Three-Valued Logic (TVL)

4. Conclusion
Assumptions for the two lectures on memory abstraction

Imperative programs viewed as \textit{transition systems}:

- set of control states: \( L \) (program points)
- set of variables: \( X \) (all assumed globals)
- set of values: \( V \) (so far: \( V \) consists of integers (or floats) only)
- set of memory states: \( M \) (so far: \( M = X \to V \))
- error state: \( \Omega \)
- states: \( S \)

\[
S = L \times M
\]
\[
S_\Omega = S \cup \{ \Omega \}
\]

- transition relation:

\[
(\to) \subseteq S \times S_\Omega
\]

\textbf{Abstraction} of sets of states

- abstract domain \( \mathbb{D}^\# \)
- concretization \( \gamma : (\mathbb{D}^\#, \subseteq^\#) \rightarrow (\mathcal{P}(S), \subseteq) \)
Assumptions: syntax of programs

We start from the same language syntax and will extend l-values:

\[
\begin{align*}
  l &::= \text{l-values} \\
  &| \quad x \quad (x \in X) \\
  &| \quad \ldots \\
  &| \quad \text{we will add other kinds of l-values} \\
  &| \quad \text{pointers, array dereference...} \\
  e &::= \text{expressions} \\
  &| \quad c \quad (c \in V) \\
  &| \quad l \quad (l\text{value}) \\
  &| \quad e \oplus e \quad (\text{arith operation, comparison}) \\
  s &::= \text{statements} \\
  &| \quad l = e \quad (\text{assignment}) \\
  &| \quad s; \ldots s; \quad (\text{sequence}) \\
  &| \quad \text{if}(e)\{s\} \quad (\text{condition}) \\
  &| \quad \text{while}(e)\{s\} \quad (\text{loop})
\end{align*}
\]
Assumptions: semantics of programs

We assume **classical definitions for:**

- **l-values:** \([l] : M \rightarrow X\)
- **expressions:** \([e] : M \rightarrow V\)
- **programs and statements:**
  - we assume a label **before each statement**
  - each statement defines a **set of transitions** (\(\rightarrow\))

In this course, we rely on the usual **reachable states semantics**

**Reachable states semantics**

The reachable states are computed as \([S]_R = \text{lfp} F\) where

\[
F : \mathcal{P}(S) \longrightarrow \mathcal{P}(S)
\]

\[X \longmapsto S_I \cup \{s \in S | \exists s' \in X, s' \rightarrow s\}\]

and \(S_I\) denotes the set of initial states.
Assumptions: general form of the abstraction

We assume an abstraction for sets of memory states:
- memory abstract domain $M^\#$
- concretization function $\gamma_M : M^\# \rightarrow \mathcal{P}(M)$

Reachable states abstraction

We construct $D^\# = L \rightarrow M^\#$ and:

$$\begin{align*}
\gamma : & \quad D^\# \quad \mapsto \quad \mathcal{P}(S) \\
X^\# & \quad \longmapsto \quad \{(l, m) \in S \mid m \in \gamma_M(X^\#(l))\}
\end{align*}$$

The whole question is how do we choose $M^\#, \gamma_M$...

- previous lectures: $X$ is fixed and finite and, $V$ is scalars (integers or floats), thus, $M \equiv V^n$
- today: we will extend the language thus, also need to extend $M^\#, \gamma_M$
Abstraction of purely numeric memory states

Purely numeric case

- $\mathbb{V}$ is a set of values of the same kind
- e.g., integers ($\mathbb{Z}$), machine integers ($\mathbb{Z} \cap [-2^{63}, 2^{63} - 1]$)...
- If the set of variables is fixed, we can use any abstraction for $\mathbb{V}^N$

Example: $N = 2$, $\mathbb{X} = \{x, y\}$

- concrete set
- interval domain
- octagon domain
- polyedra domain
Heterogeneous memory states

In real life languages, there are many kinds of values:
- **scalars** (integers of various sizes, boolean, floating-point values)...
- **pointers, arrays**...

Heterogeneous memory states and non relational abstraction

- **types** $t_0, t_1, \ldots$ and **values** $\mathbb{V} = \mathbb{V}_{t_0} \cup \mathbb{V}_{t_1} \cup \ldots$
- finitely many **variables**; each has a **fixed type**: $\mathbb{X} = \mathbb{X}_{t_0} \cup \mathbb{X}_{t_1} \cup \ldots$
- **memory states**: $\mathbb{M} = \mathbb{X}_{t_0} \rightarrow \mathbb{V}_{t_0} \times \mathbb{X}_{t_1} \rightarrow \mathbb{V}_{t_1} \ldots$

**Principle:** compose abstractions for sets of memory states of each type

Non relational abstraction of heterogeneous memory states

- $\mathbb{M} \equiv \mathbb{M}_0 \times \mathbb{M}_1 \times \ldots$ where $\mathbb{M}_i = \mathbb{X}_i \rightarrow \mathbb{V}_i$

- **Concretization function** (case with two types)

$$\gamma_{nr} : \mathcal{P}(\mathbb{M}_0) \times \mathcal{P}(\mathbb{M}_1) \rightarrow \mathcal{P}(\mathbb{M})$$

$$(m_0^\#, m_1^\#) \mapsto \{(m_0, m_1) \mid \forall i, \ m_i \in \gamma_i(m_i^\#)\}$$
Memory structures

Common structures (non exhaustive list)

- **Structures, records, tuples**: sequences of cells accessed with fields
- **Arrays**: similar to structures; indexes are integers in \([0, n - 1]\)
- **Pointers**: numerical values corresponding to the address of a memory cell
- **Strings and buffers**: blocks with a sequence of elements and a terminating element (e.g., 0x0)
- **Closures** (functional languages): pointer to function code and (partial) list of arguments

To describe memories, the definition \(M = X \rightarrow V\) is too restrictive

**Generally, non relational, heterogeneous abstraction cannot handle many such structures all at once: relations are needed!**
Specific properties to verify

**Memory safety**

Absence of memory errors (crashes, or undefined behaviors)

**Pointer errors:**
- Dereference of a null pointer / of an invalid pointer

**Access errors:**
- Out of bounds array access, buffer overruns (often used for attacks)

**Invariance properties**

Data should not become corrupted (values or structures...)

**Examples:**
- Preservation of structures, e.g., lists should remain connected
- Preservation of invariants, e.g., of balanced trees
Properties to verify: examples

A program closing a list of file descriptors

```c
// l points to a list
int c = l;
while (c != NULL) {
    close(c -> FD);
    c = c -> next;
}
```

Correctness properties

1. memory safety
2. l is supposed to store all file descriptors at all times. Will its structure be preserved? yes, no breakage of a next link
3. closure of all the descriptors

Examples of structure preservation properties

- Algorithms manipulating trees, lists...
- Libraries of algorithms on balanced trees
- Not guaranteed by the language! e.g., the balancing of Maps in the OCaml standard library was incorrect for years (performance bug)
A more realistic model

No one-to-one relation between memory cells and program variables
- A variable may indirectly reference several cells (structures...)
- Dynamically allocated cells correspond to no variable at all...

Environment + Heap
- Addresses are values: $\mathbb{V}_{\text{addr}} \subseteq \mathbb{V}$
- Environments $e \in \mathbb{E}$ map variables into their addresses
- Heaps $(h \in \mathbb{H})$ map addresses into values

$$\begin{align*}
\mathbb{E} & = \mathbb{X} \rightarrow \mathbb{V}_{\text{addr}} \\
\mathbb{H} & = \mathbb{V}_{\text{addr}} \rightarrow \mathbb{V}
\end{align*}$$

$\mathbb{h}$ is actually only a partial function
- Memory states (or memories): $\mathbb{M} = \mathbb{E} \times \mathbb{H}$

Note: Avoid confusion between heap (function from addresses to values) and dynamic allocation space (often referred to as “heap”
Example of a concrete memory state (variables)

Example setup:
- \( x \) and \( z \) are two list elements containing values 64 and 88, and where the former points to the latter
- \( y \) stores a pointer to \( z \)

Memory layout
(pointer values underlined)

\[
\begin{array}{ccc}
\text{address} & & \\
\&x = 300 & 64 & \\
304 & 312 & \\
\&y = 308 & 312 & \\
\&z = 312 & 88 & \\
316 & 0x0 & \\
\end{array}
\]

\[
\begin{array}{ccc}
e : & x & \mapsto 300 \\
y & \mapsto 308 \\
z & \mapsto 312 \\
\end{array}
\]

\[
\begin{array}{ccc}
h : & 300 & \mapsto 64 \\
304 & \mapsto 312 \\
308 & \mapsto 312 \\
312 & \mapsto 88 \\
316 & \mapsto 0 \\
\end{array}
\]
Example of a concrete memory state (variables + dyn. cell)

**Example setup:**
- same configuration
- + second field of z points to a dynamically allocated list element (in purple)

**Memory layout**

<table>
<thead>
<tr>
<th>address</th>
<th>&amp;x = 300</th>
<th>&amp;y = 308</th>
<th>&amp;z = 312</th>
</tr>
</thead>
<tbody>
<tr>
<td>304</td>
<td>312</td>
<td>312</td>
<td>316</td>
</tr>
<tr>
<td>300</td>
<td>64</td>
<td>312</td>
<td>88</td>
</tr>
<tr>
<td>312</td>
<td>508</td>
<td>312</td>
<td>508</td>
</tr>
<tr>
<td>316</td>
<td>508</td>
<td>312</td>
<td>512</td>
</tr>
<tr>
<td>508</td>
<td>25</td>
<td>316</td>
<td>512</td>
</tr>
<tr>
<td>512</td>
<td>0x0</td>
<td>508</td>
<td>512</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
e &: x &\mapsto& 300 \\
   & y &\mapsto& 308 \\
   & z &\mapsto& 312 \\
\end{align*}
\]

\[
\begin{align*}
h &: 300 &\mapsto& 64 \\
   & 304 &\mapsto& 312 \\
   & 308 &\mapsto& 312 \\
   & 312 &\mapsto& 88 \\
   & 316 &\mapsto& 508 \\
   & 508 &\mapsto& 25 \\
   & 512 &\mapsto& 0 \\
\end{align*}
\]
Extending the semantic domains

Some slight modifications to the semantics of the initial language:

- **Addresses are values:** \( V_{addr} \subseteq V \)

- **L-values evaluate into addresses:** \([l] : M \rightarrow V_{addr}\)
  (after extension of \( l \), the evaluation of l-values will also require memory states, thus we do not simply let \([l] \in V_{addr}\) )

\[
[x](e, h) = e(x)
\]

- **Semantics of expressions** \([e] : M \rightarrow V\), mostly unchanged

\[
[l](e, h) = h([l](e, h))
\]

- **Semantics of assignment** \( l_0 : l := e; l_1 : \ldots : \)

\[
(l_0, (e, h_0)) \rightarrow (l_1, (e, h_1))
\]

where

\[
h_1 = h_0[l](e, h_0) \leftarrow [e](e, h_0)
\]
Realistic definitions of memory states

Our model is still not very accurate for most languages
- Memory cells do not all have the same size
- **Memory management algorithms** usually do not treat cells one by one, e.g., `malloc` returns a pointer to a **block**, applying `free` to that pointer will dispose the **whole block**

Other refined models
- **Partition of the memory** in **blocks** with a **base address** and a **size**
- **Partition of blocks** into **cells** with a **size**
- Description of **fields** with an **offset**
- Description of **pointer values** with a **base address** and an **offset**...

For a **very formal** description of such concrete memory states: see **CompCert** project source files (Coq formalization)
Language semantics: program crash

In an abnormal situation, we assume that the program will crash
- advantage: very clear semantics
- disadvantage (for the compiler designer): dynamic checks are required

Error state
- $\Omega$ denotes an error configuration
- $\Omega$ is a blocking: $\rightarrow \subseteq S \times (\{\Omega\} \cup S)$

OCaml:
- out-of-bound array access:
  - Exception: Invalid_argument "index out of bounds".
- no notion of a null pointer

Java:
- exception in case of out-of-bound array access, null dereference:
  - java.lang.ArrayIndexOutOfBoundsException
Language semantics: undefined behaviors

Alternate choice: leave the behavior of the program **unspecified** when an abnormal situation is encountered

- advantage: easy implementation (often architecture driven)
- disadvantage: unintuitive semantics, errors hard to reproduce
  different compilers may make different choices...
  or in fact, make no choice at all (= let the program evaluate even when performing invalid actions)

Modeling of undefined behavior

- Very hard to capture what a program operation may modify
- Abnormal situation at \((l_0, m_0)\) such that \(\forall m_1 \in M, (l_0, m_0) \rightarrow (l_1, m_1)\)

- In C:
  array out-of-bound accesses and dangling pointer dereferences lead to undefined behavior (and potentially, memory corruption) whereas a null-pointer dereference always result into a crash
Memory models
Language semantics

Composite objects

How are contiguous blocks of information organized?

Java objects, OCaml struct types
- sets of fields
- each field has a type
- **no assumption** on physical storage, **no pointer arithmetics**

C composite structures and unions
- **physical mapping** defined by the norm
- each field has a specified **size** and a specified **alignment**
- **union types / casts:** implementations may allow several views
Pointers and records / structures / objects

Many languages provide **pointers** or **references** and allow to manipulate **addresses**, but with different levels of expressiveness.

What kind of objects can be referred to by a pointer?

Pointers only to records / structures / objects

- **Java**: only pointers to objects
- **OCaml**: only pointers to records, structures...

Pointers to fields

- **C**: pointers to any valid cell...

```c
struct {int a; int b} x;
int * y = &(x.b);
```
What kind of operations can be performed on a pointer?

Classical pointer operations:

- **Pointer dereference**: \( *p \) returns the contents of the cell of address \( p \)
- **“Address of” operator**: \( &x \) returns the address of variable \( x \)
- Can be analyzed with a rather coarse pointer model, e.g., symbolic base + symbolic field

Arithmetics on pointers, requiring a more precise model:

- **Addition of a numeric constant**: \( p + n \): address contained in \( p + n \) times the size of the type of \( p \)
  Interaction with pointer casts...
- **Pointer subtraction**: returns a numeric offset
Manual memory management

Allocation of unbounded memory space

- How are new memory blocks **created** by the program?
- How do old memory blocks get **freed**?

OCaml memory management

- **implicit allocation** when declaring a new object
- **garbage collection**: purely automatic process, that frees unreachable blocks

C memory management

- **manual allocation**: `malloc` operation returns a pointer to a new block
- **manual de-allocation**: `free` operation (block base address)

Manual memory management is not safe:

- **memory leaks**: growing unreachable memory region; memory exhaustion
- **dangling pointers** if freeing a block that is still referred to
Summary on the memory model

Language dependent items

- **Clear error cases** or **undefined behaviors**
  for analysis, a semantics with clear error cases is preferable

- **Composite objects**: structure fully exposed or not

- **Pointers to object fields**: allowed or not

- **Pointer arithmetic**: allowed or not
  *i.e.*, are pointer values symbolic values or numeric values

- **Memory management**: automatic or manual

In this course, we start with a simple model, and study specific features one by one and in isolation from the others
Rest of these two lectures

Abstraction for pointers and dynamic data-structures:

- pointer abstractions
- three-valued logic-based abstraction for dynamic structures
- separation logic-based abstraction for dynamic structures
- combination of value and structure abstractions

Abstract operations:

- post-condition for the **reading** of a cell defined by an l-value
  e.g., \( x = a[i] \) or \( x = *p \)
- post-condition for the **writing of a heap cell**
  e.g., \( a[i] = p \) or \( p \rightarrow f = x \)
- abstract join, that approximates unions of concrete states
Outline

1. Memory models
2. Pointer Abstractions
3. Shape analysis in Three-Valued Logic (TVL)
4. Conclusion
Syntax extension: we add pointer operations

\[ l ::= \text{l-values} \]
\[ \quad x \quad (x \in X) \]
\[ \quad \ldots \]
\[ \quad *e \quad \text{pointer dereference} \]
\[ \quad l \cdot f \quad \text{field read} \]

\[ e ::= \text{expressions} \]
\[ \quad l \]
\[ \quad \ldots \]
\[ \quad &l \quad \text{"address of" operator} \]

\[ s ::= \text{statements} \]
\[ \quad \ldots \]
\[ \quad x = \text{malloc}(c) \quad \text{allocation of } c \text{ bytes} \]
\[ \quad \text{free}(x) \quad \text{deallocation of the block pointed to by } x \]

We do not consider pointer arithmetics here
Programs with pointers: semantics

Case of l-values:

\[
\begin{align*}
\llbracket x \rrbracket(e, \bar{h}) &= e(x) \\
\llbracket *e \rrbracket(e, \bar{h}) &= \llbracket e \rrbracket(e, \bar{h}) \\
\llbracket 1 \cdot f \rrbracket(e, \bar{h}) &= \llbracket 1 \rrbracket(e, \bar{h}) + \text{offset}(f) \quad (\text{numeric offset})
\end{align*}
\]

Case of expressions:

\[
\begin{align*}
\llbracket 1 \rrbracket(e, \bar{h}) &= \begin{cases} 
\bar{h}(\llbracket 1 \rrbracket(e, \bar{h})) & \text{if } \llbracket 1 \rrbracket(e, \bar{h}) \neq 0 \land \llbracket 1 \rrbracket(e, \bar{h}) \in \text{Dom}(\bar{h}) \\
\Omega & \text{otherwise}
\end{cases} \\
\llbracket \& 1 \rrbracket(e, \bar{h}) &= \llbracket 1 \rrbracket(e, \bar{h}) \quad (\text{evaluates into the address})
\end{align*}
\]

Case of statements:

- **memory allocation** \( x = \text{malloc}(c) \): \((e, \bar{h}) \rightarrow (e, \bar{h}')\) where
  \[
  \bar{h}' = \bar{h}[e(x) \leftarrow k] \uplus \{ k \mapsto v_k, k+1 \mapsto v_{k+1}, \ldots, k+c-1 \mapsto v_{k+c-1} \} \quad \text{and}
  \]
  \( k, \ldots, k+c-1 \) are fresh and unused in \( \bar{h} \)

- **memory deallocation** \( \text{free}(x) \): \((e, \bar{h}) \rightarrow (e, \bar{h}')\) where \( k = e(x) \) and
  \[
  \bar{h} = \bar{h}' \uplus \{ k \mapsto v_k, k+1 \mapsto v_{k+1}, \ldots, k+c-1 \mapsto v_{k+c-1} \} 
  \]
We rely on the **non relational abstraction of heterogeneous states** that was introduced earlier, with a few changes:

- we let $\mathcal{V} = \mathcal{V}_{\text{addr}} \cup \mathcal{V}_{\text{int}}$ and $\mathcal{X} = \mathcal{X}_{\text{addr}} \cup \mathcal{X}_{\text{int}}$
- **concrete memory cells** now include **structure fields**, and fields of **dynamically allocated regions**
- **abstract cells** finitely summarize concrete cells
- we apply a **non relational abstraction**:

**Non relational pointer abstraction**

- Fix a set $\mathcal{A}^\#$ of **abstract addresses**, representing sets of concrete addresses
- Set of **pointer abstract values** $\mathcal{D}^\#_{\text{ptr}}$
- **Concretization** $\gamma_{\text{ptr}} : \mathcal{D}^\#_{\text{ptr}} \to \mathcal{P}(\mathcal{V}_{\text{addr}})$ into pointer sets

We will see **several instances** of this kind of abstraction, and show how such abstraction lift into abstractions for sets of heaps
Pointer non relational abstraction: null pointers

The dereference of a null pointer will cause a crash.

To establish safety: compute which pointers may be null.

Null pointer analysis

Abstract domain for addresses:

- $\gamma_{\text{ptr}}(\bot) = \emptyset$
- $\gamma_{\text{ptr}}(\top) = \mathbb{V}_{\text{addr}}$
- $\gamma_{\text{ptr}}(\neq \text{NULL}) = \mathbb{V}_{\text{addr}} \setminus \{0\}$

- we may also use a lattice with a fourth element = NULL
  
  **exercise**: what do we gain using this lattice?

- very lightweight, can typically resolve rather trivial cases

- useful for C, but also for Java
Pointer non relational abstraction: dangling pointers

The dereference of a null pointer will cause a crash

To establish safety: compute which pointers may be dangling

Null pointer analysis

Abstract domain for addresses:
- $\gamma_{\text{ptr}}(\bot) = \emptyset$
- $\gamma_{\text{ptr}}(\top) = \mathbb{V}_{\text{addr}} \times \mathbb{H}$
- $\gamma_{\text{ptr}}(\text{Not dangling}) = \{(v, h) \mid h \in \mathbb{H} \land v \in \text{Dom}(h)\}$

- very lightweight, can typically resolve rather trivial cases
- useful for C, useless for Java (initialization requirement + GC)
Determine where a pointer may store a reference to

1: int x, y;
2: int * p;
3: y = 9;
4: p = &x;
5: *p = 0;

- what is the final value for x?
  0, since it is modified at line 5...
- what is the final value for y?
  9, since it is not modified at line 5...

Basic pointer abstraction

- We assume a finite set of abstract memory locations $A^\#$ is fixed:
  \[ A^\# = \{ \&x, \&y, \ldots, \&t, a^0_0, a^0_1, \ldots, a^N_N \} \]
  where $a^0_0, \ldots, a^N_N$ is a fixed collection of $N + 1$ abstract addresses

- Concrete addresses are abstracted into $A^\#$ by $\phi_A: V_{addr} \rightarrow A^\# \cup \{\top\}$
  Assumption: $\phi_A$ surjective (no useless abstract address).

- A pointer value is abstracted by the abstraction of the addresses it may point to, i.e.,
  \[ D^\#_{ptr} = P(A^\#) \]
  and \[ \gamma_{ptr}(a^\#) = \{ a \in V_{addr} \mid \phi_A(a) \in a^\# \} \]
Abstraction of pointer states

We consider all values are of pointer type, i.e., heaps are of the form
\[ h : V_{addr} \rightarrow V_{addr} \] (recall \( M = E \times H \), \( E = X \rightarrow V_{addr} \) and \( H = V_{addr} \rightarrow V \))

**Intuition:**
- collect information separately for each element of \( A^\# \)
- use a pointer value abstract element for each abstract address

**Lifting a pointer abstraction to heap abstraction**

We let \( H^\# = A^\# \rightarrow D^\#_{ptr} \) and \( M^\# = H^\# \) and define

\[
\gamma_H(h^\#) = \{ h \in H \mid \forall a \in V_{addr}, \forall a^\# \in A^\#, \\
\phi_A(a) \in a^\# \implies \phi_A(h(a)) \in \gamma_{ptr}(h^\#(a^\#)) \}\n\]

\[
\gamma_M(h^\#) = \{ (e, h) \mid h \in \gamma_H(h^\#) \land \forall x \in X, e(x) \in h^\#(&x) \}\n\]

**Examples** of properties described by this abstraction:
- \( p \) may point to \( \{&x\} \)
- \( p \) points to some address described by \( a^\# \) and, at all addresses described by \( a^\# \), we can read another address described by \( a^\# \)
Example: points-to sets computation

Example code:

```
1:   int x, y;
2:   int * p;
3:   y = 9;
4:   p = &x;
5:   *p = 0;
6:   ...
```

Abstract locations: \{&x, &y, &p\}

Analysis results:

<table>
<thead>
<tr>
<th></th>
<th>&amp;x</th>
<th>&amp;y</th>
<th>&amp;p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>[9,9]</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>[9,9]</td>
<td>{&amp;x}</td>
</tr>
<tr>
<td>6</td>
<td>[0,0]</td>
<td>[9,9]</td>
<td>{&amp;x}</td>
</tr>
</tbody>
</table>
Example: call site abstraction

Abstraction of **memory locations**:

- for variable x: &x
- for dynamically allocated memory locations: **not discussed so far**...

**Allocation site abstraction**

**One abstract address for each malloc statement.**

Many possible refinements, e.g., using context/path sensitivity

**Example:**

```
1:   int * p, *q;
2:   p = malloc(sizeof(int))  ( point 1)
3:   q = malloc(sizeof(int))  ( point 2)
4:   *p = 0;
5:   *q = 1;
6:   q = p;
7:   ...
```

<table>
<thead>
<tr>
<th></th>
<th>&amp;p</th>
<th>&amp;q</th>
<th>a1</th>
<th>a2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>{a1}</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>{a1}</td>
<td>{a2}</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>{a1}</td>
<td>{a2}</td>
<td>[0, 0]</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>{a1}</td>
<td>{a2}</td>
<td>[0, 0]</td>
<td>[1, 1]</td>
</tr>
<tr>
<td>7</td>
<td>{a1}</td>
<td>{a1}</td>
<td>[0, 0]</td>
<td>[1, 1]</td>
</tr>
</tbody>
</table>

**Note:** memory leak at line 5
Points-to sets computation and imprecision

x ∈ [−10, −5]; y ∈ [5, 10]
1: int * p;
2: if(?){
3: p = &x;
4: } else {
5: p = &y;
6: }
7: *p = 0;
8: ...

What is the final range for x?
What is the final range for y?

Abstract locations: {&x, &y, &p}

<table>
<thead>
<tr>
<th></th>
<th>&amp;x</th>
<th>&amp;y</th>
<th>&amp;p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[−10, −5]</td>
<td>[5, 10]</td>
<td>⊤</td>
</tr>
<tr>
<td>2</td>
<td>[−10, −5]</td>
<td>[5, 10]</td>
<td>⊤</td>
</tr>
<tr>
<td>3</td>
<td>[−10, −5]</td>
<td>[5, 10]</td>
<td>⊤</td>
</tr>
<tr>
<td>4</td>
<td>[−10, −5]</td>
<td>[5, 10]</td>
<td>{&amp;x}</td>
</tr>
<tr>
<td>5</td>
<td>[−10, −5]</td>
<td>[5, 10]</td>
<td>⊤</td>
</tr>
<tr>
<td>6</td>
<td>[−10, −5]</td>
<td>[5, 10]</td>
<td>{&amp;y}</td>
</tr>
<tr>
<td>7</td>
<td>[−10, −5]</td>
<td>[5, 10]</td>
<td>{&amp;x, &amp;y}</td>
</tr>
<tr>
<td>8</td>
<td>[−10, 0]</td>
<td>[0, 10]</td>
<td>{&amp;x, &amp;y}</td>
</tr>
</tbody>
</table>

Imprecise results
- The abstract information about both x and y are weakened
- The fact that x ≠ y is lost
Weak updates

We can formalize this imprecision a bit more:

- The modified concrete cell cannot be uniquely mapped into a well identified abstract cell that describes only it.
- The resulting abstract information is obtained by joining the new value and the old information.

Effect in pointer analysis, in the case of an assignment:

- If the points-to set contains exactly one element, the analysis can perform a strong update as in the first example: $p \Rightarrow \{&x\}$.
- If the points-to set may contain more than one element, the analysis needs to perform a weak-update as in the second example: $p \Rightarrow \{&x, &y\}$. 
Weak updates

We recall:

- \( A^\# = \{ &x, &y, \ldots, &t, a_0^\#, a_1^\#, \ldots, a_N^\# \} \)
- \( \phi_A : \mathbb{V}_{\text{addr}} \to A^\# \cup \{ \top \} \), surjective

Moreover, we assume an abstract state \( h^\# \) and an assignment \( l := c \) where \( l \) is an l-value. We note the abstract evaluation of the l-value:

\[
L := \phi_A^{-1}([1][h^#]) = \{ a \in A^\# | \phi_A(a) \in [1][h^#] \}
\]

We have two cases, based on the cardinality of \( L \):

1. \( |L| \leq 1 \):
   then, exactly one abstract value needs to be updated (\( \phi_A(a) \) if \( L = \{ a \} \))

2. \( |L| > 1 \):
   then, there exists two distinct addresses \( a_0, a_1 \in L \); since the assignment overwrites one cell exactly:
   
   - if the content of \( a_0 \) is modified, then that of \( a_1 \) stays the same...
   - the other way around too, of course
   
   thus the post-condition need to map \( \phi_A(a_0) \) to something weaker than \( h^#(a_0) \), and the same for \( a_1 \), which means we have a weak update
Weak updates

We consider:
- abstract heap $h^\#$
- assignment $1 := c$
- the abstract evaluation of the l-value:

$$L ::= \phi^{-1}_A([1]^\#(h^\#)) = \{ a \in A^\# | \phi_A(a) \in [1]^\#(h^\#) \}$$

So, **when does the weak update happen?**

There are two (non exclusive) situations:

1. **when** $|[1]^\#(h^\#)| > 1$:
   this includes that the evaluation of 1 is not precise in the abstract

2. **when there exists** $a \in [1]^\#(h^\#)$ **such that** $|\phi^{-1}_A(\{a\})| > 1$:
   this means that one of the addresses 1 may evaluate to corresponds to several distinct concrete cells

Both cases can be expected to happen frequently in pointer analysis...
Pointer aliasing based on equivalence on access paths

**Aliasing relation**

Given \( m = (e, h) \), pointers \( p \) and \( q \) are **aliases** iff \( h(e(p)) = h(e(q)) \)

**Abstraction to infer pointer aliasing properties**

- An **access path** describes a sequence of dereferences to resolve an l-value (i.e., an address); e.g.:
  
  \[
  a ::= x \mid a \cdot f \mid *a
  \]

- An **abstraction for aliasing** is an over-approximation for **equivalence relations** over access paths

**Examples of aliasing abstractions:**

- **set abstractions:** map from access paths to their equivalence class (ex: \( \{\{p_0, p_1, &x\}, \{p_2, p_3\}, \ldots\} \))

- **numerical relations**, to describe aliasing among paths of the form \( x(-\rightarrow n)^k \) (ex: \( \{x(-\rightarrow n)^k, & x(-\rightarrow n)^{k+1} \mid k \in \mathbb{N} \} \)
Limitation of basic pointer analyses seen so far

Weak updates:
- **imprecision in updates** that spread out as soon as points-to set contain several elements
- impact **client analyses** severely (e.g., low precision on numerical)

Unsatisfactory abstraction of unbounded memory:
- common assumption that there are finitely many abstract cells
- programs using **dynamic allocations** often perform **unbounded** numbers of **malloc** calls (e.g., allocation of a list)

Unable to express well structural invariants:
- for instance, that a structure should be a **list**, a **tree**...
- **very indirect** abstraction in numeric / path equivalence abstraction

A common solution: shape abstraction
Outline

1. Memory models

2. Pointer Abstractions

3. Shape analysis in Three-Valued Logic (TVL)
   - Principles of Three-Valued Logic based abstraction
   - Comparing and concretizing Three-Valued Logic abstractions
   - Weakening Three-Valued Logic abstractions
   - Transfer functions
   - Focusing

4. Conclusion
Observation: representation of memory states by graphs

- **Nodes** (aka, atoms) denote **variables, memory locations**
- **Edges** denote **properties of addresses / pointers**, such as:
  - “field \( f \) of location \( u \) points to \( v \)”
  - “variable \( x \) is stored at location \( u \)”
- This representation is also relevant in the case of **separation logic** based shape abstraction

A couple of examples:

**Two alias pointers:**

```
   y ----> u₁
     |        |
     v       v
   x ----> u₀  u₂
```

**A list of length 2 or 3:**

```
x -- u₀  u₁  u₂
```

```
x -- u₀  u₁  u₂  u₃
```

We need to over-approximate **sets of shape graphs**
Memory graphs and predicates: variables

Before we apply some abstraction, we formalize memory graphs using some predicates, such as:

“Variable content” predicate

We note $x(u) = 1$ if node $u$ represents the contents of $x$.

Examples:

- Two alias pointers:

  ![Graph](image)

  Then, we have $x(u_0) = 1$ and $y(u_1) = 1$, and $x(u) = 0$ (resp., $y(u) = 0$) in all the other cases.

- A list of length 2:

  ![Graph](image)

  Then, we have $x(u_0) = 1$ and $x(u) = 0$ in all the other cases.
Memory graphs and predicates: (field) pointers

“Field content pointer” predicate

- We note $n(u, v)$ if the field $n$ of $u$ stores a pointer to $v$
- We note $0(u, v)$ if $u$ stores a pointer to $v$ (base address field is at offset 0)

Examples:

- **Two alias pointers:**

  \[
  y \rightarrow u_1 \\
  x \rightarrow u_0 \\
  \]

  Then, we have $0(u_0, u_2) = 1$ and $0(u_1, u_2) = 1$, and $0(u, v) = 0$ in all the other cases

- **A list of length 2:**

  \[
  x \rightarrow u_0 \stackrel{n}{\rightarrow} u_1 \stackrel{n}{\rightarrow} u_2 \\
  \]

  Then, we have $n(u_0, u_1) = 1$ and $n(u_1, u_2) = 1$, and $n(u, v) = 0$ in all the other cases
2-structures and concretization

We can represent the memory graphs using **tables of predicate values**: 

**Two-structures and concretization**

We assume a set \( \mathcal{P} = \{p_0, p_1, \ldots, p_n\} \) of **predicate symbols** (we write \( k_i \) for the arity of predicate \( p_i \)). A formal representation of a memory graph is a **2-structure** \((\mathcal{U}, \phi) \in \mathbb{D}_2^\#\) defined by:

- a set \( \mathcal{U} = \{u_0, u_1, \ldots, u_m\} \) of **atoms**
- a **truth table** \( \phi \) such that \( \phi(p_i, u_{l_1}, \ldots, u_{l_{k_i}}) \) denotes the truth value of \( p_i \) for \( u_{l_1}, \ldots, u_{l_{k_i}} \) (where arities of predicates are respected)

Then, \( \gamma_2(\mathcal{U}, \phi) \) is the set of \((e, h, \nu)\) where \( \nu : \mathcal{U} \rightarrow V_{\text{addr}} \) and that satisfy exactly the truth tables defined by \( \phi \):

- \((e, h, \nu)\) satisfies \( x(u) \) iff \( e(x) = \nu(u) \)
- \((e, h, \nu)\) satisfies \( f(u, v) \) iff \( h(\nu(u), f) = \nu(v) \)

- the name “two-structure” will become clear (very) soon
- the set of two-structures is parameterized by the data of a set of predicates \( x(.), y(.), 0(., .), n(., .) \) (additional predicates will be added soon...)
Examples of two-structures

Two alias pointers:

A list of length 2:

A list of length 2:

Lists of arbitrary length? More on this later
Unknown value: three valued logic

How to abstract away some information?

i.e., how to abstract several graphs into one?

Example: pointer variable p alias with x or y

A boolean lattice

- Use predicate tables
- Add a \( \top \) boolean value;
  (denoted to by \( \frac{1}{2} \) in TVLA papers)

Graph representation:
dotted edges

Abstract graph:
Summary nodes

At this point, we cannot talk about **unbounded memory states** with **finitely many** nodes, since one node represents at most one memory cell.

**An idea:** Choose a node to represent several concrete nodes.

**Definition: summary node**

A **summary node** is an atom that may denote several concrete atoms. Formally: a **unary predicate** \( \text{sum} \) (convention: 0 for non summary nodes; otherwise \( \frac{1}{2} \)).

- **Intuition:** we are using a **non injective function** \( \phi_A : V_{\text{addr}} \rightarrow A^\# \).
- **Representation:** double circled nodes.

**Lists of lengths 1, 2, 3:**

- \( x \rightarrow u_0 \overset{n}{\rightarrow} u_1 \)
- \( x \rightarrow u_0 \overset{n}{\rightarrow} u_1 \overset{n}{\rightarrow} u_2 \)
- \( x \rightarrow u_0 \overset{n}{\rightarrow} u_1 \overset{n}{\rightarrow} u_2 \overset{n}{\rightarrow} u_3 \)

**Attempt at a summary graph:**

- Edges to \( u_1 \) are dotted.
Additional graph predicate: sharing

We now define a few **higher level predicates** based on the previously seen **atomic predicates** describing the graphs.

Example: a cell is **shared** if and only if there exists several distinct pointers to it.

**“Is shared” predicate**

The predicate $\text{sh}(u)$ holds if and only if

$$\exists v_0, v_1, \left\{ \begin{array}{l} v_0 \neq v_1 \\
\& n(v_0, u) \\
\& n(v_1, u) \end{array} \right.$$  

(for concision, we assume only $n$ pointers)

\[ \text{sh}(u_0) = \text{sh}(u_1) = \text{sh}(u_3) = 0 \]
\[ \text{sh}(u_2) = 1 \]
Additional graph predicate: reachability

We can also define higher level predicates using induction:

For instance, a cell is **reachable** from \( u \) if and only if it is \( u \) or it is reachable from a cell pointed to by \( u \).

### “Reachability” predicate

The predicate \( r(u, v) \) holds if and only if:

\[
\begin{align*}
u &= v \\
\lor & \exists u_0, \ n(u, u_0) \land r(u_0, v)
\end{align*}
\]

(for concision, we assume only \( n \) pointers)

\[
\begin{array}{l}
\text{x} \\
\bigcirc_0^n \\
\bigcirc_1^n \\
\bigcirc_2^n \\
\bigcirc_3^n
\end{array}
\]

\begin{itemize}
  \item \( r(u_1, u_0) = r(u_2, u_0) = r(u_3, u_1) = 0 \)
  \item \( r(u_0, u_0) = r(u_0, u_2) = r(u_0, u_3) = 1 \)
\end{itemize}

### “Acyclicity” predicate

The predicate \( \text{acy}(u) \) holds iff \( \exists v, \ v \neq u \land r(u, v) \land r(v, u) \) does not hold.
Three structures

As for 2-structures, we assume a set $\mathcal{P} = \{p_0, p_1, \ldots, p_n\}$ of predicates fixed and write $k_i$ for the arity of predicate $p_i$.

**Definition: 3-structures**

A **3-structure** is a tuple $(U, \phi)$ defined by:

- a set $U = \{u_0, u_1, \ldots, u_m\}$ of atoms
- a truth table $\phi$ such that $\phi(p_i, u_{l_1}, \ldots, u_{l_{k_i}})$ denotes the truth value of $p_i$ for $u_{l_1}, \ldots, u_{l_{k_i}}$

  note: truth values are elements of the lattice $\{0, \frac{1}{2}, 1\}$

We write $D_3^\#$ for the set of three-structures.

In the following we build up an abstract domain of 3-structures (but a bit more work is needed for the definition of the concretization)
Main predicates and concretization

We have already seen:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x(u)$</td>
<td>variable $x$ contains the address of $u$</td>
</tr>
<tr>
<td>$n(u, v)$</td>
<td>field of $u$ points to $v$</td>
</tr>
<tr>
<td>$\text{sum}(u)$</td>
<td>whether $u$ is a summary node (convention: either 0 or $\frac{1}{2}$)</td>
</tr>
<tr>
<td>$\text{sh}(u)$</td>
<td>whether there exists several distinct pointers to $u$</td>
</tr>
<tr>
<td>$r(u, v)$</td>
<td>whether $v$ is reachable starting from $u$</td>
</tr>
<tr>
<td>$\text{acy}(v)$</td>
<td>$v$ may not be on a cycle</td>
</tr>
</tbody>
</table>

Concretization for 2 structures:

$\left( e, h, \nu \right) \in \gamma_2(U, \phi) \iff \bigwedge_{p \in \mathcal{P}} (\text{env}, h, \nu) \text{ evaluates } p \text{ as specified in } \phi$

Concretization for 3 structures:

- predicates with value $\frac{1}{2}$ may concretize either to true or to false
- but the concretization of summary nodes is still unclear...
Outline

1. Memory models

2. Pointer Abstractions

3. Shape analysis in Three-Valued Logic (TVL)
   - Principles of Three-Valued Logic based abstraction
   - Comparing and concretizing Three-Valued Logic abstractions
   - Weakening Three-Valued Logic abstractions
   - Transfer functions
   - Focusing

4. Conclusion
Embedding

Reasons why we need to set up a relation among structures:

- learn how to compare two 3-structures
- describe the concretization of 3-structures into 2-structures

The embedding principle

Let $S_0 = (U_0, \phi_0)$ and $S_1 = (U_1, \phi_1)$ be two three-structures, with the same sets of predicates $P$. Let $f: U_0 \rightarrow U_1$, surjective. We say that $f$ embeds $S_0$ into $S_1$ iff

$$\phi_0(p, u_{l_1}, \ldots, u_{l_{k_i}}) \sqsubseteq \phi_1(p, f(u_{l_1}), \ldots, f(u_{l_{k_i}}))$$

for all predicate $p \in P$ of arity $k$, for all $u_{l_1}, \ldots, u_{l_{k_i}} \in U_0$.

Then, we write $S_0 \sqsubseteq^f S_1$

- Note: we use the order $\sqsubseteq$ of the lattice $\{0, \frac{1}{2}, 1\}$
- Intuition: embedding defines an abstract pre-order i.e., when $S_0 \sqsubseteq^f S_1$, any property that is satisfied by $S_0$ is also satisfied by $S_1$
Embedding examples

A few examples of the embedding relation:

where $f : u_0 \mapsto u_0; u_1 \mapsto u_1; u_2 \mapsto u_1$

where $f : u_0 \mapsto u_0; u_1 \mapsto u_1; u_2 \mapsto u_1; u_3 \mapsto u_1$

where $f : u_0 \mapsto u_0; u_1 \mapsto u_1; u_2 \mapsto u_1$

The last example shows summary nodes are not enough to capture just lists:

- **reachability** would be necessary to constrain it be a list
- alternatively: list cells **should not be shared**
Concretization of three-structures

**Intuitions:**
- concrete memory states correspond to 2-structures
- embedding applies uniformly to 2-structures and 3-structures (in fact, 2-structures are a subset of 3-structures)
- 2-structures can be embedded into 3-structures, that abstract them

This suggests **a concretization of 3-structures in two steps:**
1. turn it into a set of 2-structures that can be embedded into it
2. concretize these 2-structures

**Concretization of 3-structures**

Let $S$ be a 3-structure. Then:

$$
\gamma_3(S) = \bigcup \{ \gamma_2(S') \mid S' \text{ 2-structure s.t. } \exists f, S' \sqsubseteq^f S \}
$$
Concretization examples

**Without reachability:**

\[
x \xrightarrow{n} u_0 \xrightarrow{n} u_1 \xrightarrow{n} u_2
\]

\[
x \xrightarrow{n} u_0 \xrightarrow{n} u_1 \xrightarrow{n} u_2 \xrightarrow{n} u_3
\]

where \( f : u_0 \mapsto u_0; u_1 \mapsto u_1; u_2 \mapsto u_1; u_3 \mapsto u_1 \)

**With reachability:**

\[
x \xrightarrow{n} u_0 \xrightarrow{n} u_1 \xrightarrow{n} u_2 \xrightarrow{n} u_1
\]

\[
x \xrightarrow{n} u_0 \xrightarrow{n} u_1 \xrightarrow{n} u_1 \left( u_0, u_1 \right)
\]

where \( f : u_0 \mapsto u_0; u_1 \mapsto u_1; u_2 \mapsto u_1 \)

Note the first item of the above case does not work here
Disjunctive completion

- Do 3-structures allow for a **sufficient level of precision**?
- How to **over-approximate a set of 2-structures**?

```c
int * x; int * y;...
int * p = NULL;
if(...){
    p = x;
} else{
    p = y;
}
printf("%d", *p);
*p = ...;
```

After the if statement:
abstracting would be imprecise

---

Abstraction based on disjunctive completion

- In the following, we use **partial disjunctive completion**;
i.e., TVLA manipulates **finite disjunctions** of 3-structures.
We write $\mathbb{D}_P^3$ for the abstract domain made of finite sets of 3-structures in $\mathbb{D}_3^3$

- How to ensure disjunctions **will not grow infinite**?
the set of atoms is **unbounded**, so it is not necessarily true!
Outline

1. Memory models

2. Pointer Abstractions

3. Shape analysis in Three-Valued Logic (TVL)
   - Principles of Three-Valued Logic based abstraction
   - Comparing and concretizing Three-Valued Logic abstractions
   - Weakening Three-Valued Logic abstractions
   - Transfer functions
   - Focusing

4. Conclusion
Canonical abstraction

To **prevent disjunctions** from growing infinite, we propose to **normalize (in a precision losing manner)** abstract states:

- the analysis may use all 3-structures at most points
- at selected points (including loop heads), only 3-structures in a finite set $D_{\text{can}(3)}$ are allowed
- there is a function to coarsen 3-structures into elements of $D_{\text{can}(3)}$

**Canonicalization function**

Let $L$ be a lattice, $L' \subseteq L$ be a finite sub-lattice and $\text{can} : L \rightarrow L'$:

- operator $\text{can}$ is called **canonicalization** if and only if it defines an **upper closure operator**
- then it extends into a **canonicalization operator** $\text{can} : \mathcal{P}(L) \rightarrow \mathcal{P}(L')$ for the disjunctive completion domain:
  \[
  \text{can}(E) = \{ \text{can}(x) \mid x \in E \}
  \]

proof of the extension to disjunctive completion domains: left as an exercise
to make the powerset domain work, we simply need a $\text{can}$ over 3-structures
Canonical abstraction

**Definition of a finite lattice $\mathbb{D}_{\text{can}}^\#(3)$**

We partition the set of predicates $P$ into two subsets $P_a$ and $P_o$:

- $P_a$ and defines **abstraction predicates** and should contain only unary predicates and have a finite truth table whatever the number of atoms.
- $P_o$ denotes **non-abstraction predicates**, and may define truth tables of unbounded size.

Then, we let $\mathbb{D}_{\text{can}}^\#(3)$ be the set of 3-structures such that **no pair of atoms have the same value of the $P_a$ predicates**. It defines a finite set of 3-structures.

This sub-lattice defines a clear “canonicalization” algorithm:

**Canonical abstraction by truth blurring**

1. **Identify** nodes that **have different abstraction predicates**
2. When several nodes have the **same abstraction predicate** introduce a summary node
3. **Compute new predicate values** by doing a **join over truth values**
Canonical abstraction examples

Most common TVLA instantiation:

- assume there are $n$ variables $x_1, \ldots, x_n$
- thus the number of unary predicates is finite, and provides a good choice for $P_a$
- sub-lattice: structures with atoms distinguished by the values of the unary predicates $x_1, \ldots, x_n$

Examples:

<table>
<thead>
<tr>
<th>Elements not merged:</th>
<th>Elements merged:</th>
<th>Abstract into:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y \rightarrow u_1$</td>
<td>$x \rightarrow u_0 \rightarrow u_1$</td>
<td>$x \rightarrow u_0 \rightarrow u_1$</td>
</tr>
<tr>
<td>$p \rightarrow u_0$</td>
<td>$x \rightarrow u_0 \rightarrow u_1 \rightarrow u_2$</td>
<td>$u_0 \rightarrow u_1, u_2, u_3$</td>
</tr>
<tr>
<td>$x \rightarrow u_0$</td>
<td>$x \rightarrow u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow u_3$</td>
<td>$r(x)$</td>
</tr>
</tbody>
</table>
Outline

1 Memory models

2 Pointer Abstractions

3 Shape analysis in Three-Valued Logic (TVL)
   - Principles of Three-Valued Logic based abstraction
   - Comparing and concretizing Three-Valued Logic abstractions
   - Weakening Three-Valued Logic abstractions
   - Transfer functions
   - Focusing

4 Conclusion
Principle for the design of sound transfer functions

- Intuitively, **concrete states** correspond to **2-structures**
- The **analysis** should track **3-structures**, thus the analysis and its soundness proof need to **rely on the embedding relation**

### Embedding theorem

We assume that

- $S_0 = (\mathcal{U}_0, \phi_0)$ and $S_1 = (\mathcal{U}_1, \phi_1)$ define a pair of 3-structures
- $f : \mathcal{U}_0 \rightarrow \mathcal{U}_1$, is such that $S_0 \sqsubseteq^f S_1$ (embedding)
- $\Psi$ is a logical formula, with variables in $X$
- $g : X \rightarrow \mathcal{U}_0$ is an assignment for the variables of $\Psi$

Then, the semantics (evaluation) of logical formulae is such that

$$\llbracket \Psi \rvert_g \rrbracket (S_0) \sqsubseteq \llbracket \Psi \rvert_{f \circ g} \rrbracket (S_1)$$

**Intuition**: this theorem ties the evaluation of conditions in the concrete and in the abstract in a general manner
Principle for the design of sound transfer functions

Transfer functions for static analysis

- Semantics of concrete statements is encoded into boolean formulas
- Evaluation in the abstract is sound (embedding theorem)

Example: analysis of an assignment \( y := x \)

1. let \( y' \) be a new predicate that denotes the new value of \( y \)
2. then we can add the constraint \( y'(u) = x(u) \)
   (using the embedding theorem to prove soundness)
3. rename \( y' \) into \( y \)

Advantages:
- abstract transfer functions derive directly from the concrete transfer functions (intuition: \( \alpha \circ f \circ \gamma \ldots \))
- the same solution works for weakest pre-conditions

Disadvantage: precision will require some care, more on this later!
Assignment: a simple case

<table>
<thead>
<tr>
<th>Statement</th>
<th>Pre-condition $S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_0 : y = y \rightarrow n; l_1 : \ldots$</td>
<td>$x, y \xrightarrow{n} u_0 \xrightarrow{n} u_1 \xrightarrow{n} u_2$</td>
</tr>
</tbody>
</table>

Transfer function computation:

- It should produce an over-approximation of $\{m_1 \in \mathbb{M} | (l_0, m_0) \rightarrow (l_1, m_1)\}$
- **Encoding** using “primed predicates” to denote predicates after the evaluation of the assignment, to evaluate them in the same structure (non primed variables are removed afterwards and primed variables renamed):

\[
\begin{align*}
x'(u) & = x(u) \\
y'(u) & = \exists v, \ y(v) \land n(v, u) \\
n'(u, v) & = n(u, v)
\end{align*}
\]

- Resulting structure:

\[
x, y \xrightarrow{n} u_0 \xrightarrow{n} u_1 \xrightarrow{n} u_2
\]

This is exactly the expected result.
Outline

1. Memory models

2. Pointer Abstractions

3. Shape analysis in Three-Valued Logic (TVL)
   - Principles of Three-Valued Logic based abstraction
   - Comparing and concretizing Three-Valued Logic abstractions
   - Weakening Three-Valued Logic abstractions
   - Transfer functions
   - Focusing

4. Conclusion
Assignment: a more involved case

<table>
<thead>
<tr>
<th>Statement</th>
<th>Pre-condition $S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_0: y = y \rightarrow n; l_1: \ldots$</td>
<td>$u^0 \rightarrow u_1^1$</td>
</tr>
</tbody>
</table>

- Let us try to **resolve the update in the same way as before**:
  
  $$x'(u) = x(u)$$
  $$y'(u) = \exists v, y(v) \land n(v, u)$$
  $$n'(u, v) = n(u, v)$$

- We **cannot resolve** $y'$:
  
  $$\begin{cases}
  y'(u_0) = 0 \\
  y'(u_1) = \frac{1}{2}
  \end{cases}$$

  **Imprecision**: after the statement, $y$ may point to anywhere in the list, save for the first element...

- The assignment transfer function **cannot be computed immediately**
- **We need to refine the 3-structure first**
Focus

Focusing on a formula

We assume a 3-structure $S$ and a boolean formula $f$ are given, we call a focusing $S$ on $f$ the generation of a set $\hat{S}$ of 3-structures such that:

- $f$ evaluates to 0 or 1 on all elements of $\hat{S}$
- precision was gained: $\forall S' \in \hat{S}, S' \sqsubseteq S$ (embedding)
- soundness is preserved: $\gamma(S) = \bigcup\{\gamma(S') | S' \in \hat{S}\}$

Details of focusing algorithms are rather complex: not detailed here

They involve splitting of summary nodes, solving of boolean constraints

Example: focusing on $y'(u) = \exists v, y(v) \land n(v, u)$

We obtain (we show $y$ and $y'$):
Focus and coerce

Some of the 3-structures generated by focus are not precise

\[ u_0 \xrightarrow{n} u_1 \]
\[ x, y \xrightarrow{r(x)} \]

\[ u_0 \xrightarrow{n} u_1 \xrightarrow{n} u_2 \]
\[ x, y \xrightarrow{r(x), y'} \xrightarrow{r(x)} \]

- \( u_1 \) is reachable from \( x \), but there is no sequence of \( n \) fields: this structure has empty concretization
- \( u_0 \) has an \( n \)-field to \( u_1 \) so \( u_1 \) denotes a unique atom and cannot be a summary node

Coerce operation

It enforces logical constraints among predicates and discards 3-structures with an empty concretization

Result: one case removed (bottom), two possibly summary nodes non summary

\[ u_0 \xrightarrow{n} u_1 \]
\[ x, y \xrightarrow{r(x), y'} \]

\[ u_0 \xrightarrow{n} u_1 \xrightarrow{n} u_2 \]
\[ x, y \xrightarrow{r(x), y'} \xrightarrow{r(x)} \]
Focus, transfer, abstract...

Computations of a transfer function

We consider a transfer function encoded into boolean formula \( f \)

\[
\begin{align*}
S_{\text{pre}}^\# & \quad \text{focus} \quad \text{coerce} \quad S_{\text{pre}} \quad f \quad S_{\text{post}} \quad \text{can} \quad S_{\text{post}}^\#
\end{align*}
\]

Soundness proof steps:

1. sound encoding of the semantics of program statements into formulas (typically, no loss of precision at this stage)
2. focusing produces a refined over-approximation (disjunction)
3. canonicalization over-approximates graphs (truth blurring)

A common picture in shape analysis
Shape analysis with three valued logic

Abstract states; two abstract domains are used:

- **infinite domain** \( \mathbb{D}^\#_P(3) \): finite disjunctions of 3-structures in \( \mathbb{D}^\#_3 \) for general abstract computations
- **finite domain** \( \mathbb{D}^\#_P(\text{can}(3)) \): disjunctions of finite domain \( \mathbb{D}^\#_{\text{can}(3)} \) to simplify abstract states and for loop iteration
- **concretization** via \( \mathbb{D}^\#_2 \)

Abstract post-conditions:

1. start from \( \mathbb{D}^\#_P(3) \) or \( \mathbb{D}^\#_{\text{can}(3)} \)
2. focus and coerce when needed
3. apply the concrete transformation
4. apply can to weaken abstract states; result in \( \mathbb{D}^\#_P(\text{can}(3)) \)

Analysis of loops:

- iterations in \( \mathbb{D}^\#_P(\text{can}(3)) \) terminate, as it is finite
Outline

1. Memory models
2. Pointer Abstractions
3. Shape analysis in Three-Valued Logic (TVL)
4. Conclusion
Updates and summarization

- Weak updates cause significant precision loss...

- Basic pointer abstractions suffer weak update issues leading to high precision loss
- Various techniques exist to mitigate this effect
- Today, we saw shape analysis based on three-valued predicates as a way to circumvent it
  Next week, another technique will be presented...

A novel family of abstract interpretation based static analyses:
- Some analysis operations require local concretization of abstract predicates
- A reverse operation makes abstract states more abstract
Assignment: formalization and paper reading

Formalization of the concretization of 2-structures:

- describe the concretization formula, assuming that we consider the predicates discussed in the course
- run it on the list abstraction example (from the 3-structure to a few select 2-structures, and down to memory states)

Reading:

**Parametric Shape Analysis via 3-Valued Logic.**
Shmuel Sagiv, Thomas W. Reps et Reinhard Wilhelm.
Assignment: a simple analysis in TVLA

1, k assumed to be disjoint lists

\textbf{while}(1 \neq 0)\
\begin{align*}
t & = l -> n; \\
l -> n & = k; \\
k & = l; \\
l & = t;
\end{align*}
\}