Memory abstraction 1

MPRI — Cours 2.6 "Interprétation abstraite :
application à la vérification et à l'analyse statique"

Xavier Rival

INRIA, ENS, CNRS

Jan, 27th. 2025

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 ?**

$\Rightarrow$ **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

Assumptions for the two lectures on memory abstraction

Imperative programs viewed as **transition systems**:

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

$$\begin{array}{rcl} \mathbb{S} & = & \mathbb{L} \times \mathbb{M} \\ \mathbb{S}_\Omega & = & \mathbb{S} \uplus \{\Omega\} \end{array}$$

- **transition relation:**

$$(\rightarrow) \subseteq \mathbb{S} \times \mathbb{S}_\Omega$$

Abstraction of sets of states

- **abstract domain** $\mathbb{D}^\sharp$
- **concretization** $\gamma : (\mathbb{D}^\sharp, \sqsubseteq^\sharp) \longrightarrow (\mathcal{P}(\mathbb{S}), \subseteq)$

Assumptions: syntax of programs

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

l	::=	**l-values**	
	|	x	(x $\in \mathbb{X}$)
	|	...	**we will add other kinds of l-values pointers, array dereference...**
e	::=	**expressions**	
	|	c	($c \in \mathbb{V}$)
	|	l	(lvalue)
	|	e $\oplus$ e	(arith operation, comparison)
s	::=	**statements**	
	|	l = e	(assignment)
	|	s; ... s;	(sequence)
	|	**if**(e){s}	(condition)
	|	**while**(e){s}	(loop)

Assumptions: semantics of programs

We assume **classical definitions for:**

- **l-values**: $[\![\mathtt{l}]\!]_{\mathrm{l}} : \mathbb{M} \to \mathbb{X}$
- **expressions**: $[\![\mathtt{e}]\!]_{\mathrm{e}} : \mathbb{M} \to \mathbb{V}$
- **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 $[\![\mathcal{S}]\!]_{\mathcal{R}} = \mathbf{lfp} F$ where

$$\begin{array}{rccl} F: & \mathcal{P}(\mathbb{S}) & \longrightarrow & \mathcal{P}(\mathbb{S}) \\ & X & \longmapsto & \mathbb{S}_{\mathcal{I}} \cup \{ s \in \mathbb{S} \mid \exists s' \in X,\ s' \rightarrow s \} \end{array}$$

and $\mathbb{S}_{\mathcal{I}}$ denotes the set of initial states.

Assumptions: general form of the abstraction

We assume an **abstraction for sets of memory states**:

- memory abstract domain $\mathbb{M}^\sharp$
- concretization function $\gamma_{\mathbb{M}} : \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{M})$

Reachable states abstraction

We construct $\mathbb{D}^\sharp = \mathbb{L} \to \mathbb{M}^\sharp$ and:

$$\begin{array}{rccl} \gamma : & \mathbb{D}^\sharp & \longrightarrow & \mathcal{P}(\mathbb{S}) \\ & X^\sharp & \longmapsto & \{(\ell, m) \in \mathbb{S} \mid m \in \gamma_{\mathbb{M}}(X^\sharp(\ell))\} \end{array}$$

The whole question is how do we choose $\mathbb{M}^\sharp, \gamma_{\mathbb{M}}$...

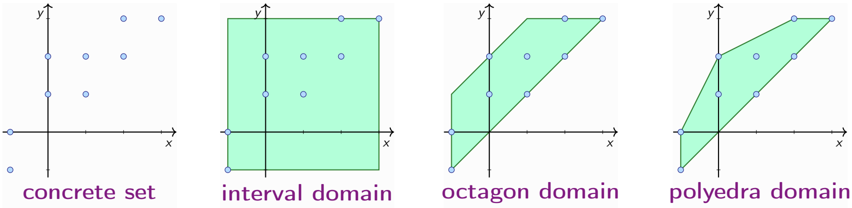
- previous lectures:
  $\mathbb{X}$ **is fixed and finite** and, $\mathbb{V}$ **is scalars** (integers or floats), thus, $\mathbb{M} \equiv \mathbb{V}^n$
- today:
  we will **extend the language** thus, also **need to extend** $\mathbb{M}^\sharp, \gamma_{\mathbb{M}}$

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



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** $\mathbb{V} = \mathbb{V}_{t_0} \uplus \mathbb{V}_{t_1} \uplus \ldots$
- finitely many **variables**; each has a **fixed type:** $\mathbb{X} = \mathbb{X}_{t_0} \uplus \mathbb{X}_{t_1} \uplus \ldots$
- **memory states:** $\mathbb{M} = \mathbb{X}_{t_0} \to \mathbb{V}_{t_0} \times \mathbb{X}_{t_1} \to \mathbb{V}_{t_1} \ldots$

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 $\mathbb{M}_i = \mathbb{X}_i \to \mathbb{V}_i$
- **Concretization function** (case with two types)

$$\begin{array}{rrcl} \gamma_{\mathrm{nr}}: & \mathcal{P}(\mathbb{M}_0) \times \mathcal{P}(\mathbb{M}_1) & \longrightarrow & \mathcal{P}(\mathbb{M}) \\ & (m_0^\sharp, m_1^\sharp) & \longmapsto & \{(m_0, m_1) \mid \forall i,\ m_i \in \gamma_i(m_i^\sharp)\} \end{array}$$

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 $\mathbb{M} = \mathbb{X} \rightarrow \mathbb{V}$ 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

```
//l points to a list
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

$$\begin{array}{rcl} \mathbb{E} & = & \mathbb{X} \to \mathbb{V}_{\text{addr}} \\ \mathbb{H} & = & \mathbb{V}_{\text{addr}} \to \mathbb{V} \end{array}$$

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

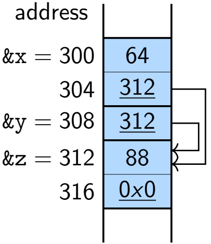
Note: **Avoid confusion between heap (function from addresses to values) and dynamic allocation space (often referred to as "heap")**

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)



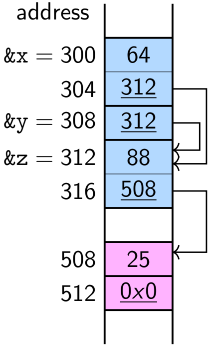
$$
\begin{array}{llll}
e: & \mathtt{x} & \mapsto & 300 \\
 & \mathtt{y} & \mapsto & 308 \\
 & \mathtt{z} & \mapsto & 312 \\
 & & & \\
h: & 300 & \mapsto & 64 \\
 & 304 & \mapsto & 312 \\
 & 308 & \mapsto & 312 \\
 & 312 & \mapsto & 88 \\
 & 316 & \mapsto & 0
\end{array}
$$

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



e :	x	$\mapsto$	300
	y	$\mapsto$	308
	z	$\mapsto$	312
h :	300	$\mapsto$	64
	304	$\mapsto$	312
	308	$\mapsto$	312
	312	$\mapsto$	88
	316	$\mapsto$	508
	508	$\mapsto$	25
	512	$\mapsto$	0

Extending the semantic domains

Some slight modifications to the semantics of the initial language:

- **Addresses are values:** $\mathbb{V}_{\mathrm{addr}} \subseteq \mathbb{V}$
- **L-values evaluate into addresses:** $[\![\mathtt{l}]\!]_{\mathrm{l}} : \mathbb{M} \to \mathbb{V}_{\mathrm{addr}}$
  (after extension of $\mathtt{l}$, the evaluation of l-values will also require memory states, thus we do not simply let $[\![\mathtt{l}]\!]_{\mathrm{l}} \in \mathbb{V}_{\mathrm{addr}}$)

$$[\![\mathtt{x}]\!]_{\mathrm{l}}(e, h) \quad = \quad e(\mathtt{x})$$

- **Semantics of expressions** $[\![\mathtt{e}]\!]_{\mathrm{e}} : \mathbb{M} \to \mathbb{V}$, mostly unchanged

$$[\![\mathtt{l}]\!]_{\mathrm{e}}(e, h) \quad = \quad h([\![\mathtt{l}]\!]_{\mathrm{l}}(e, h))$$

- **Semantics of assignment** $\ell_0 : \mathtt{l} := \mathtt{e}; \ell_1 : \ldots$:

$$(\ell_0, (e, h_0)) \longrightarrow (\ell_1, (e, h_1))$$

  where

$$h_1 = h_0[[\![\mathtt{l}]\!]_{\mathrm{l}}(e, h_0) \leftarrow [\![\mathtt{e}]\!]_{\mathrm{e}}(e, h_0)]$$

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

- Ω denotes an **error configuration**
- Ω is a **blocking**: $(\rightarrow) \subseteq \mathbb{S} \times (\{\Omega\} \uplus \mathbb{S})$

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 $\forall m_1 \in \mathbb{M},\ (l_0, m_0) \rightarrow (l_1, m_1)$

- **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

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...

$$\textbf{struct}\ \{\textbf{int}\ \mathtt{a};\ \textbf{int}\ \mathtt{b}\}\ \mathtt{x};$$
$$\textbf{int} * \mathtt{y} = \&(\mathtt{x}\cdot\mathtt{b});$$

Pointer arithmetics

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., $\mathtt{x} = \mathtt{a}[\mathtt{i}]$ or $\mathtt{x} = *\mathtt{p}$
- post-condition for the **writing of a heap cell**
  e.g., $\mathtt{a}[\mathtt{i}] = \mathtt{p}$ or $\mathtt{p}$ -> $\mathtt{f} = \mathtt{x}$
- **abstract join**, that approximates unions of concrete states

Outline

Programs with pointers: syntax

Syntax extension: we add pointer operations

$$
\begin{array}{rcll}
\mathtt{l} & ::= & \textbf{l-values} & \\
& | & \mathtt{x} & (\mathtt{x} \in \mathbb{X}) \\
& | & \ldots & \\
& | & *\mathtt{e} & \text{pointer dereference} \\
& | & \mathtt{l} \cdot \mathtt{f} & \text{field read} \\
\mathtt{e} & ::= & \textbf{expressions} & \\
& | & \mathtt{l} & \\
& | & \ldots & \\
& | & \&\mathtt{l} & \text{"address of" operator} \\
\mathtt{s} & ::= & \textbf{statements} & \\
& | & \ldots & \\
& | & \mathtt{x} = \textbf{malloc}(c) & \text{allocation of } c \text{ bytes} \\
& | & \textbf{free}(\mathtt{x}) & \text{deallocation of the block pointed to by } \mathtt{x}
\end{array}
$$

We do not consider **pointer arithmetics here**

Programs with pointers: semantics

Case of l-values:

$$\begin{array}{rcl} [\![\mathtt{x}]\!]_{\mathrm{l}}(e, h) & = & e(\mathtt{x}) \\ [\![*\mathtt{e}]\!]_{\mathrm{l}}(e, h) & = & [\![\mathtt{e}]\!]_{\mathrm{e}}(e, h) \\ [\![\mathtt{l} \cdot \mathtt{f}]\!]_{\mathrm{l}}(e, h) & = & [\![\mathtt{l}]\!]_{\mathrm{l}}(e, h) + \textbf{offset}(\mathtt{f}) \text{ (numeric offset)} \end{array}$$

Case of expressions:

$$[\![\mathtt{l}]\!]_{\mathrm{e}}(e, h) = \begin{cases} h([\![\mathtt{l}]\!]_{\mathrm{l}}(e, h)) & \text{if } [\![\mathtt{l}]\!]_{\mathrm{l}}(e, h) \neq 0 \wedge [\![\mathtt{l}]\!]_{\mathrm{l}}(e, h) \in \mathrm{Dom}(h) \\ \Omega & \text{otherwise} \end{cases}$$

(evaluates into the contents)

$$[\![\&\mathtt{l}]\!]_{\mathrm{l}}(e, h) = [\![\mathtt{l}]\!]_{\mathrm{l}}(e, h) \quad \text{(evaluates into the address)}$$

Case of statements:

- **memory allocation** $\mathtt{x} = \textbf{malloc}(c)$: $(e, h) \to (e, h')$ where $h' = h[e(\mathtt{x}) \leftarrow k] \uplus \{k \mapsto v_k, k+1 \mapsto v_{k+1}, \ldots, k+c-1 \mapsto v_{k+c-1}\}$ and $k, \ldots, k+c-1$ are fresh and unused in h
- **memory deallocation** $\textbf{free}(\mathtt{x})$: $(e, h) \to (e, h')$ where $k = e(\mathtt{x})$ and $h = h' \uplus \{k \mapsto v_k, k+1 \mapsto v_{k+1}, \ldots, k+c-1 \mapsto v_{k+c-1}\}$

Is this formalization faithful to the C semantics? what is missing?

Non relational pointer abstractions

We rely on the **non relational abstraction of heterogeneous states** that was introduced earlier, with a few changes:

- **values** are either **scalars** or **pointers**
  we assume $\mathbb{V} = \mathbb{V}_{\mathrm{addr}} \uplus \mathbb{V}_{\mathrm{int}}$ and $\mathbb{X} = \mathbb{X}_{\mathrm{addr}} \uplus \mathbb{X}_{\mathrm{int}}$
- **concrete memory cells** now include **structure fields**, and fields of **dynamically allocated regions**

In thhe formalization, we temporarily leave out scalars...

Non relational pointer abstraction relies on two ingredients:

1. a **partitioning** of the memory in **finitely many sets of cells**
   each of these sets will be described by one piece of abstract information
2. a **pointer abstraction** that allows to describe pointer values

Abstraction of addresses: abstract addresses

Definition: addresses abstraction

An **address abstraction** is defined by:

- A set of **abstract memory locations** $\mathbb{A}^\sharp$
- A concretization defined by a **pointwise abstraction of concrete addresses** $\phi_{\mathbb{A}} : \mathbb{V}_{\mathrm{addr}} \rightarrow \mathbb{A}^\sharp \uplus \{\top\}$:

$$a^\sharp \mapsto \{a \in \mathbb{V}_{\mathrm{addr}} \mid \phi_{\mathbb{A}}(a) = a^\sharp\}$$

Typical choice of abstract addreses $\mathbb{A}^\sharp$:

- one abstract address per scalar or pointer variable
- a finite set of abstract addresses for allocated regions,
  e.g., one per allocation site
  e.g., one per allocation site + path/context (path/context sensitivity)

In the following:

$$\mathbb{A}^\sharp = \{\&\mathtt{x}, \&\mathtt{y}, \ldots, \&\mathtt{t}, a_0^\sharp, a_1^\sharp, \ldots, a_N^\sharp\}$$

Abstraction of pointer values

Definition: pointer abstraction

A **pointer abstraction** is defined by a set of **pointer abstract values** $\mathbb{D}^{\sharp}_{\mathrm{ptr}}$ and a **concretization** $\gamma_{\mathrm{ptr}} : \mathbb{D}^{\sharp}_{\mathrm{ptr}} \to \mathcal{P}(\mathbb{V}_{\mathrm{addr}})$

We will see a few examples of such abstractions shortly, which express properties such as:

- may a pointer be null or is it definitely non-null ?
- may a pointer be invalid or is it definitely valid ?
- which addresses may a pointer refer to ?

Before this, we finish the definition of the **non relational pointer abstraction**

Non-relational pointer abstraction

Assuming a set of abstract addresses and a pointer abstraction, we build an abstraction for memory states

Intuition:

- collect information separately for each element of $\mathbb{A}^\sharp$
- use a pointer value abstract element for each abstract address

Lifting a pointer abstraction to heap abstraction

We let the **heap abstract domain** be $\mathbb{H}^\sharp = \mathbb{A}^\sharp \rightarrow \mathbb{D}^\sharp_{\mathrm{ptr}}$ and the **memory abstract domain** be $\mathbb{M}^\sharp = \mathbb{H}^\sharp$

We define the concretizations:

$$\begin{array}{rcl}\gamma_{\mathbb{H}}(h^\sharp) & = & \{\hbar \in \mathbb{H} \mid \forall a \in \mathbb{V}_{\mathrm{addr}}, \forall a^\sharp \in \mathbb{A}^\sharp, \\ & & \qquad \phi_{\mathbb{A}}(a) \in a^\sharp \Longrightarrow \phi_{\mathbb{A}}(\hbar(a)) \in \gamma_{\mathrm{ptr}}(h^\sharp(a^\sharp))\} \\ \gamma_{\mathbb{M}}(h^\sharp) & = & \{(e, \hbar) \mid \hbar \in \gamma_{\mathbb{H}}(h^\sharp) \wedge \forall \mathrm{x} \in \mathbb{X},\ e(\mathrm{x}) \in h^\sharp(\&\mathrm{x})\}\end{array}$$

Pointer non relational abstraction: null pointers

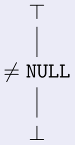
The dereference of a null pointer will cause a crash

To establish **safety**: compute **which pointers may be null**

Null pointer analysis

Abstract domain for addresses:

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



- We may also use a lattice with a fourth element $= \texttt{NULL}$
  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 dereferece of a null pointer will cause a crash

To establish **safety**: compute **which pointers may be dangling**

Null pointer analysis

Abstract domain for addresses:

- $\gamma_{\mathrm{ptr}}(\bot) = \emptyset$
- $\gamma_{\mathrm{ptr}}(\top) = \mathbb{V}_{\mathrm{addr}} \times \mathbb{H}$
- $\gamma_{\mathrm{ptr}}(\text{Not dangling}) = \{(v, h) \mid h \in \mathbb{H} \wedge v \in \text{Dom}(h)\}$

$\top$
|
Not dangling
|
$\bot$

- Very **lightweight**, can typically resolve rather trivial cases
- Useful for **C**, useless for Java (initialization requirement + GC)

Exercise: a slight update of γ_{ptr} is needed; define it...

Pointer non relational abstraction: points-to sets

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

- $\mathbb{D}^\sharp_{\mathrm{ptr}} = \mathcal{P}(\mathbb{A}^\sharp)$
- $\gamma_{\mathrm{ptr}}(a^\sharp) = \{a \in \mathbb{V}_{\mathrm{addr}} \mid \phi_{\mathbb{A}}(a) \in a^\sharp\}$

Examples of properties described by this abstraction:

- p may point to {&x}
- p points to some address described by $a_0^\sharp$ and, at all addresses described by $a_0^\sharp$, we can read another address described by $a_1^\sharp$

Points-to sets computation: example

Abstraction of **pointer+scalar values**: heterogeneous memory abstraction

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:

	&x	&y	&p
1	$\top$	$\top$	$\top$
2	$\top$	$\top$	$\top$
3	$\top$	$\top$	$\top$
4	$\top$	$[9, 9]$	$\top$
5	$\top$	$[9, 9]$	$\{\&x\}$
6	$[0, 0]$	$[9, 9]$	$\{\&x\}$

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 refinments, *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 :  ...
```

	&p	&q	a_1	a_2
1	$\top$	$\top$	$\top$	$\top$
2	$\top$	$\top$	$\top$	$\top$
3	$\{a_1\}$	$\top$	$\top$	$\top$
4	$\{a_1\}$	$\{a_2\}$	$\top$	$\top$
5	$\{a_1\}$	$\{a_2\}$	$[0, 0]$	$\top$
6	$\{a_1\}$	$\{a_2\}$	$[0, 0]$	$[1, 1]$
7	$\{a_1\}$	$\{a_1\}$	$[0, 0]$	$[1, 1]$

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 : ...
```

	&x	&y	&p
1	$[-10, -5]$	$[5, 10]$	$\top$
2	$[-10, -5]$	$[5, 10]$	$\top$
3	$[-10, -5]$	$[5, 10]$	$\top$
4	$[-10, -5]$	$[5, 10]$	{&x}
5	$[-10, -5]$	$[5, 10]$	$\top$
6	$[-10, -5]$	$[5, 10]$	{&y}
7	$[-10, -5]$	$[5, 10]$	{&x, &y}
8	$[-10, 0]$	$[0, 10]$	{&x, &y}

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

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

Imprecise results

- The abstract information about both x and y are weakened
- The fact that $x \neq y$ is lost

Weak updates

We can formalize this imprecision a bit more:

Weak updates

- **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: $\mathrm{p} \Mapsto \{\&\mathrm{x}\}$
- if the points-to set may contain **more than one element**, the analysis needs to perform a **weak-update**
  as in the second example: $\mathrm{p} \Mapsto \{\&\mathrm{x}, \&\mathrm{y}\}$

Weak updates

We recall:

- $\mathbb{A}^\sharp = \{\&\mathtt{x}, \&\mathtt{y}, \ldots, \&\mathtt{t}, a_0^\sharp, a_1^\sharp, \ldots, a_N^\sharp\}$
- $\phi_{\mathbb{A}} : \mathbb{V}_{\mathrm{addr}} \to \mathbb{A}^\sharp \uplus \{\top\}$, surjective

Moreover, we assume an abstract state $h^\sharp$ and an assignment $\mathtt{l} := c$ where $\mathtt{l}$ is an l-value. We note the abstract evaluation of the l-value $\mathtt{l}$:

$$\mathcal{L} ::= \phi_{\mathbb{A}}^{-1}(\llbracket \mathtt{l} \rrbracket^\sharp(h^\sharp)) = \{a \in \mathbb{A}^\sharp \mid \phi_{\mathbb{A}}(a) \in \llbracket \mathtt{l} \rrbracket^\sharp(h^\sharp)\}$$

We have **two cases**, based on **the cardinality of $\mathcal{L}$**:

1. $|\mathcal{L}| \leq 1$**:**
   then, exactly one abstract value needs to be updated ($\phi_{\mathbb{A}}(a)$ if $\mathcal{L} = \{a\}$)
2. $|\mathcal{L}| > 1$**:**
   then, there exist two distinct addresses $a_0, a_1 \in \mathcal{L}$; 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 $\phi_{\mathbb{A}}(a_0)$ to something weaker than $h^\sharp(a_0)$, and the same for a_1, which means we have a weak update

Weak updates

We consider:

- abstract heap $h^\sharp$
- assignment $\mathtt{l} := c$
- the abstract evaluation of the l-value:

$$\mathcal{L} ::= \phi_{\mathbb{A}}^{-1}(\llbracket \mathtt{l} \rrbracket^\sharp(h^\sharp)) = \{a \in \mathbb{A}^\sharp \mid \phi_{\mathbb{A}}(a) \in \llbracket \mathtt{l} \rrbracket^\sharp(h^\sharp)\}$$

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

There are two (non exclusive) situations:

1. **when** $|\llbracket \mathtt{l} \rrbracket^\sharp(h^\sharp)| > 1$:
   this includes that the evaluation of $\mathtt{l}$ is not precise in the abstract
2. **when there exists** $a \in \llbracket \mathtt{l} \rrbracket^\sharp(h^\sharp)$ **such that** $|\phi_{\mathbb{A}}^{-1}(\{a\})| > 1$:
   this means that one of the addresses $\mathtt{l}$ 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 $h(e(\mathrm{p})) = h(e(\mathrm{q}))$

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.*:

$$a ::= \mathtt{x} \mid a \cdot f \mid * a$$

- 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:** $\{\{\mathtt{p}_0, \mathtt{p}_1, \&\mathtt{x}\}, \{\mathtt{p}_2, \mathtt{p}_3\}, \ldots\}$)
- **numerical relations**, to describe aliasing among paths of the form $\mathtt{x(\text{->}n)}^{\mathtt{k}}$ (**ex:** $\{\{\mathtt{x(\text{->}n)}^{\mathtt{k}}, \&(\mathtt{x(\text{->}n)}^{\mathtt{k+1}}) \mid k \in \mathbb{N}\}$)

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 abstration

A common solution:
shape abstraction

Outline

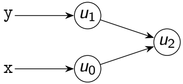
Representation of memory states: memory graphs

Observation: representation of memory states by graphs

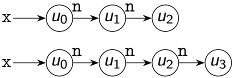
- **Nodes** (aka, atoms) denote **variables, memory locations**
- **Edges** denote **properties of addresses / pointers**, such as:
  - "field `f` of location u points to v"
  - "variable `x` is stored at location u"
- This representation is also relevant in the case of **separation logic** based shape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

A couple of examples:

Two alias pointers:



A list of length 2 or 3:



We need to over-approximate **sets of shape graphs**

Intuition: nodes represent $\mathbb{V}_{\mathrm{addr}}$ and later $\mathbb{A}^\sharp$ too

Memory graphs and predicates: variables

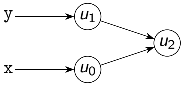
Before we apply some abstraction, we **formalize memory graphs** using some **predicates**, such as:

"Variable content" predicate

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

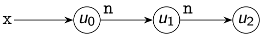
Examples:

- **Two alias pointers**:



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

- **A list of length 2**:



  Then, we have $\mathtt{x}(u_0) = 1$ and $\mathtt{x}(u) = 0$ in all the other cases

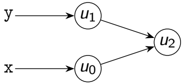
Memory graphs and predicates: (field) pointers

"Field content pointer" predicate

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

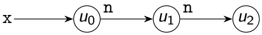
Examples:

- **Two alias pointers**:



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

- **A list of length 2**:



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

2-structures and conretization

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** $(\mathcal{U}, \phi) \in \mathbb{D}_2^\sharp$ defined by:

- a set $\mathcal{U} = \{u_0, u_1, \ldots, u_m\}$ of **atoms**
- a **truth table** ϕ 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)

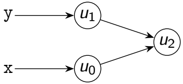
Then, $\gamma_2(\mathcal{U}, \phi)$ is the set of (e, h, ν) where $\nu : \mathcal{U} \to \mathbb{V}_{\mathrm{addr}}$ and that satisfy exactly the truth tables defined by ϕ:

- (e, h, ν) satisfies $\mathtt{x}(u)$ iff $e(\mathtt{x}) = \nu(u)$
- (e, h, ν) satisfies $\mathtt{f}(u, v)$ iff $h(\nu(u), \mathtt{f}) = \nu(v)$

- the name "two-structure" will become clear (very) soon
- the set of two-structures is parameterized by the data of a set of predicates $\mathtt{x}(.), \mathtt{y}(.), \underline{0}(.,.), \mathtt{n}(.,.)$ (additional predicates will be added soon...)

Examples of two-structures

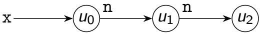
Two alias pointers:



	x	y
u_0	1	0
u_1	0	1
u_2	0	0

$\mapsto$	u_0	u_1	u_2
u_0	0	0	1
u_1	0	0	1
u_2	0	0	0

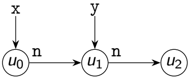
A list of length 2:



	x
u_0	1
u_1	0
u_2	0

$\cdot\mathtt{n} \mapsto$	u_0	u_1	u_2
u_0	0	1	0
u_1	0	0	1
u_2	0	0	0

A list of length 2:



	x	y
u_0	1	0
u_1	0	1
u_2	0	0

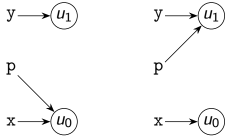
$\cdot\mathtt{n} \mapsto$	u_0	u_1	u_2
u_0	0	1	0
u_1	0	0	1
u_2	0	0	0

Lists of **arbitrary length** ? More on this **later**

Unknown value: three valued logic

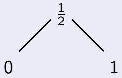
How to abstract away some information ?

i.e., how to abstract several graphs into one ?
Example: pointer variable `p` alias with `x` or `y`

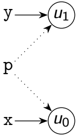


A boolean lattice

- 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

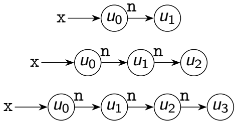
An idea: Choose a node to represent several concrete nodes

Definition: summary node

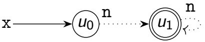
A **summary node** is an atom that may denote several concrete atoms.
Formally: a **unary predicate** $\underline{\mathrm{sum}}$ (convention: 0 for non summary nodes; otherwise $\frac{1}{2}$)

- intuition: we are using a **non injective function** $\phi_{\mathbb{A}} : \mathbb{V}_{\mathrm{addr}} \longrightarrow \mathbb{A}^{\sharp}$
- representation: double circled nodes

Lists of lengths 1, 2, 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.

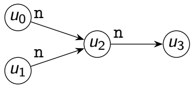
Example: a cell is **shared** if and only if there exists several distinct pointers to it

"Is shared" predicate

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

$$\exists v_0, v_1, \begin{cases} & v_0 \neq v_1 \\ \wedge & \mathtt{n}(v_0, u) \\ \wedge & \mathtt{n}(v_1, u) \end{cases}$$

(for concision, we assume only $\mathtt{n}$ pointers)



- $\underline{\mathrm{sh}}(u_0) = \underline{\mathrm{sh}}(u_1) = \underline{\mathrm{sh}}(u_3) = 0$
- $\underline{\mathrm{sh}}(u_2) = 1$

Additional graph predicate: reachability

We can also define higher level predicates **using induction**:

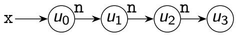
For instance, a cell is **reachable** from u if and only it is u or it is reachable from a cell pointed to by u.

"Reachability" predicate

The predicate $\underline{\mathrm{r}}(u, v)$ holds if and only if:

$$\begin{array}{ll} & u = v \\ \vee & \exists u_0,\ \mathtt{n}(u, u_0) \wedge \underline{\mathrm{r}}(u_0, v) \end{array}$$

(for concision, we assume only $\mathtt{n}$ pointers)



- $\underline{\mathrm{r}}(u_1, u_0) = \underline{\mathrm{r}}(u_2, u_0) = \underline{\mathrm{r}}(u_3, u_1) = 0$
- $\underline{\mathrm{r}}(u_0, u_0) = \underline{\mathrm{r}}(u_0, u_2) = \underline{\mathrm{r}}(u_0, u_3) = 1$

"Acyclicity" predicate

The predicate $\underline{\mathrm{acy}}(u)$ holds iff $\exists v,\ v \neq u \wedge \underline{\mathrm{r}}(u, v) \wedge \underline{\mathrm{r}}(v, u)$ does not hold

Three structures

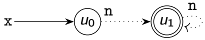
As for 2-structures, we assume a set $\mathcal{P} = \{p_0, p_1, \ldots, p_n\}$ of **predicates** fixed and write k_i for the arity of predicate p_i.

Definition: 3-structures

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

- a set $\mathcal{U} = \{u_0, u_1, \ldots, u_m\}$ of **atoms**
- a **truth table** ϕ 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 $\mathbb{D}_3^\sharp$ for the set of three-structures.



$$\left\{ \begin{array}{l} \mathcal{U} = \{u_0, u_1\} \\ \mathcal{P} = \{\mathtt{x}(\cdot), \mathtt{n}(\cdot,\cdot), \underline{\mathtt{sum}}(\cdot)\} \end{array} \right.$$

	x	<u>sum</u>
u_0	1	0
u_1	0	$\frac{1}{2}$

n	u_0	u_1
u_0	0	$\frac{1}{2}$
u_1	0	$\frac{1}{2}$

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:

$\mathtt{x}(u)$	variable x contains the address of u
$\mathtt{n}(u, v)$	field of u points to v
$\underline{\mathrm{sum}}(u)$	whether u is a summary node (convention: either 0 or $\frac{1}{2}$)
$\underline{\mathrm{sh}}(u)$	whether there exists several distinct pointers to u
$\underline{\mathrm{r}}(u, v)$	whether v is reachable starting from u
$\underline{\mathrm{acy}}(v)$	v may not be on a cycle

Concretization for 2 structures:

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

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

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 $\mathcal{S}_0 = (\mathcal{U}_0, \phi_0)$ and $\mathcal{S}_1 = (\mathcal{U}_1, \phi_1)$ be two three structures, with the same sets of predicates $\mathcal{P}$. Let $f : \mathcal{U}_0 \to \mathcal{U}_1$, surjective.

We say that f **embeds** $\mathcal{S}_0$ **into** $\mathcal{S}_1$ iff

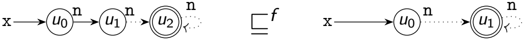
$$\text{for all predicate } p \in \mathcal{P} \text{ of arity } k, \quad \text{for all } u_{l_1}, \ldots, u_{l_{k_i}} \in \mathcal{U}_0, \\ \phi_0(p, u_{l_1}, \ldots, u_{l_{k_i}}) \sqsubseteq \phi_1(p, f(u_{l_1}), \ldots, f(u_{l_{k_i}}))$$

Then, **we write** $\mathcal{S}_0 \sqsubseteq^f \mathcal{S}_1$

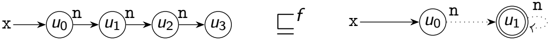
- Note: we use the order $\sqsubseteq$ of the lattice $\{0, \frac{1}{2}, 1\}$
- Intuition: **embedding** defines an **abstract pre-order**
  i.e., when $\mathcal{S}_0 \sqsubseteq^f \mathcal{S}_1$, any property that is satsfied by $\mathcal{S}_0$ is also satisfied by $\mathcal{S}_1$

Embedding examples

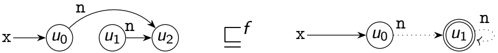
A few examples of the embedding relation:



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 uniformally 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

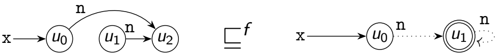
Concretization of 3-structures

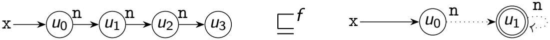
Let $\mathcal{S}$ be a 3-structure. Then:

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

Concretization examples

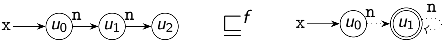
Without reachability:





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

With reachability:



$\underline{r}(u_0, u_1)$

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

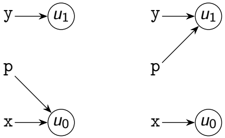
Note the first item of the above case does not work here

Disjunctive completion

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

```
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 $\mathbb{D}^\sharp_{\mathcal{P}(3)}$ for the abstract domain made of finite sets of 3-structures in $\mathbb{D}^\sharp_3$
- How to ensure disjunctions **will not grow infinite** ?
  the set of atoms is **unbounded**, so it is not necessarily true!

Outline

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 $\mathbb{D}^\sharp_{\mathsf{can}(3)}$ are allowed
- there is a function to coarsen 3-structures into elements of $\mathbb{D}^\sharp_{\mathsf{can}(3)}$

Canonicalization function

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

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

$$\mathsf{can}(\mathcal{E}) = \{\mathsf{can}(x) \mid x \in \mathcal{E}\}$$

- proof of the extension to disjunctive completion domains: left as an exercise
- to make the powerset domain work, we simply need a can over 3-structures

Canonical abstraction

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

We partition the set of predicates $\mathcal{P}$ into two subsets $\mathcal{P}_a$ and $\mathcal{P}_o$:

- $\mathcal{P}_a$ and defines **abstraction predicates** and should contains only unary predicates and have a finite truth table whatever the number of atoms
- $\mathcal{P}_o$ denotes **non-abstraction predicates**, and may define truth tables of unbounded size

Then, we let $\mathbb{D}^\sharp_{\text{can}(3)}$ be the set of 3-structures such that **no pair of atoms have the same value of the $\mathcal{P}_a$ 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**

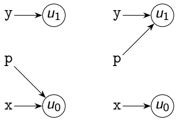
Canonical abstraction examples

Most common TVLA instantiation:

- ae assume there are n variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$
  thus the number of unary predicates is finite, and provides a good choice for $\mathcal{P}_a$
- **sub-lattice**: structures with atoms **distinguished by the values of the unary predicates** $\mathtt{x}_1, \ldots, \mathtt{x}_n$

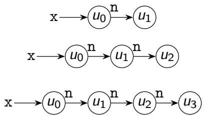
Examples:

Elements not merged:

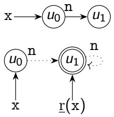


Elements merged:

Lists of lengths 1, 2, 3:



Abstract into:



Outline

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

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

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

$$[\![\Psi_{|g}]\!](\mathcal{S}_0) \sqsubseteq [\![\Psi_{|f \circ g}]\!](\mathcal{S}_1)$$

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 $\mathtt{y} := \mathtt{x}$

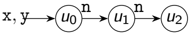
1. let $\mathtt{y}'$ be a new predicate that denotes the *new* value of $\mathtt{y}$
2. then we can add the constraint $\mathtt{y}'(u) = \mathtt{x}(u)$
   (using the embedding theorem to prove soundness)
3. rename $\mathtt{y}'$ into $\mathtt{y}$

Advantages:

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

Disadvantage: precision will require some care, more on this later!

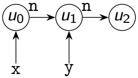
Assignment: a simple case

Statement l_0 : y = y -> n; l_1 : ...	**Pre-condition** $\mathcal{S}$ 

Transfer function computation:

- it should produce an over-approximation of $\{m_1 \in \mathbb{M} \mid (l_0, m_0) \rightarrow (l_1, m_1)\}$
- **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):

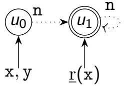
$$\begin{array}{rcl} \mathtt{x}'(u) & = & \mathtt{x}(u) \\ \mathtt{y}'(u) & = & \exists v,\ \mathtt{y}(v) \wedge \mathtt{n}(v, u) \\ \mathtt{n}'(u, v) & = & \mathtt{n}(u, v) \end{array}$$

- **resulting structure:** 

This is exactly the expected result

Outline

Assignment: a more involved case

Statement l_0 : $\mathtt{y = y \text{ -> } n};\ l_1 : \ldots$	**Pre-condition** $\mathcal{S}$ 

- Let us try to **resolve the update in the same way as before**:

$$\begin{array}{rcl} \mathtt{x}'(u) & = & \mathtt{x}(u) \\ \mathtt{y}'(u) & = & \exists v,\ \mathtt{y}(v) \wedge \mathtt{n}(v, u) \\ \mathtt{n}'(u, v) & = & \mathtt{n}(u, v) \end{array}$$

- We **cannot resolve** $\mathtt{y}'$:

$$\left\{ \begin{array}{rcl} \mathtt{y}'(u_0) & = & 0 \\ \mathtt{y}'(u_1) & = & \frac{1}{2} \end{array} \right.$$

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

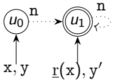
We assume a 3-structure $\mathcal{S}$ and a boolean formula f are given, we call a **focusing $\mathcal{S}$ on f** the generation of a set $\hat{\mathcal{S}}$ of 3-structures such that:

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

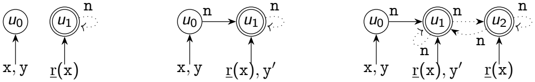
- Details of focusing algorithms are rather complex: not detailed here
- They involve splitting of summary nodes, solving of boolean constraints

Example: focusing on

$$\mathrm{y}'(u) = \exists v,\ \mathrm{y}(v) \wedge \mathrm{n}(v, u)$$

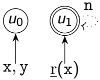


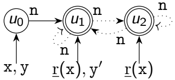
We obtain (we show y and y'):



Focus and coerce

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





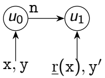
u_1 is reachable from x, but there is no sequence of n fields: this structure has **empty concretization**

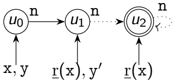
u_0 has an n-field to u_1 so u_1 denotes a unique atom and **cannot be a summary node**

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

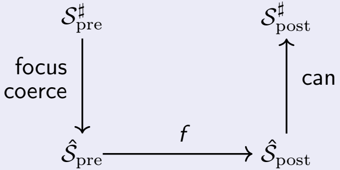




Focus, transfer, abstract...

Computation of a transfer function

We consider a transfer function encoded into boolean formula f



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** $\mathbb{D}^\sharp_{\mathcal{P}(3)}$: finite disjunctions of 3-structures in $\mathbb{D}^\sharp_3$ for general abstract computations
- **finite domain** $\mathbb{D}^\sharp_{\mathcal{P}(\mathsf{can}(3))}$: disjunctions of finite domain $\mathbb{D}^\sharp_{\mathsf{can}(3)}$ to simplify abstract states and for loop iteration
- **concretization** via $\mathbb{D}^\sharp_2$

Abstract post-conditions:

1. start from $\mathbb{D}^\sharp_{\mathcal{P}(3)}$ or $\mathbb{D}^\sharp_{\mathsf{can}(3)}$
2. focus and coerce when needed
3. apply the concrete transformation
4. apply can to weaken abstract states; result in $\mathbb{D}^\sharp_{\mathcal{P}(\mathsf{can}(3))}$

Analysis of loops:

- iterations in $\mathbb{D}^\sharp_{\mathcal{P}(\mathsf{can}(3))}$ terminate, as it is finite

Outline

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**
- The set of **abstract addresses** is dynamic, *i.e.*, changes during the analysis

Internships

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.
In POPL'99, pages 105–118, 1999.

Assignment: a simple analysis in TVLA

l, k assumed to be disjoint lists

```
while(l ≠ 0){
    t = l -> n;
    l -> n = k;
    k = l;
    l = t;
}
```