Memory abstraction 1

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

Xavier Rival

INRIA, ENS, CNRS

Dec, 10th. 2014

Overview of the lecture

So far, we have shown **numeric abstract domains**

- non relational: intervals, congruences...
- relational: polyhedra, octagons, ellipsoids...

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

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

This lecture:

- **overview most common problems**
- discuss **arrays**, **strings**
- introduction to **shape analysis**

Next lecture: deeper study of a **family of shape analyses**

Assumptions

Programs can be viewed as **transition systems**:

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

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

- a program is described by a **transition relation:**

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

Abstraction: described by a domain $\mathbb{D}^\sharp$ and a concretization:

$$\gamma : (\mathbb{D}^\sharp, \sqsubseteq^\sharp) \longrightarrow (\mathcal{P}(\mathbb{S}), \subseteq)$$

Programs: syntax

We start with a **minimal language**, to be extended with arrays, strings, pointers...

A minimal imperative language

$$\begin{array}{rcll}
\mathtt{l} & ::= & \textbf{l-valules} & \\
 & | & \mathtt{x} & (\mathtt{x} \in \mathbb{X}) \\
\mathtt{e} & ::= & \textbf{expressions} & \\
 & | & c & (c \in \mathbb{V}) \\
 & | & \mathtt{l} & (\textit{lvalue}) \\
 & | & \mathtt{e} \oplus \mathtt{e} & (\textit{arithoperation}, \textit{comparison}) \\
\mathtt{s} & ::= & \textbf{statements} & \\
 & | & \mathtt{l} = \mathtt{e} & (\text{assignment}) \\
 & | & \mathtt{s}; \ldots \mathtt{s}; & (\text{sequence}) \\
 & | & \textbf{if}(\mathtt{e})\{\mathtt{s}\} & (\text{condition}) \\
 & | & \textbf{while}(\mathtt{e})\{\mathtt{s}\} & (\text{loop})
\end{array}$$

Programs: semantics

We assume **classical definitions for:**

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

We rely on the usual:

Reachable states semantics

The reachable states are computed as $[\![\mathcal{S}]\!]_{\mathcal{R}} = \mathbf{lfp} F$ where

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

Programs: semantics abstraction

We assume a **memory abstraction**:

- memory abstract domain $\mathbb{D}^\sharp_{\rm mem}$
- concretization function $\gamma_{\rm mem} : \mathbb{D}^\sharp_{\rm mem} \rightarrow \mathcal{P}(\mathbb{M})$

Reachable states abstraction

We construct $\mathbb{D}^\sharp = \mathbb{L} \rightarrow \mathbb{D}^\sharp_{\rm mem}$ and:

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

The whole question is how do we choose $\mathbb{D}^\sharp_{\rm mem}, \gamma_{\rm mem}$...

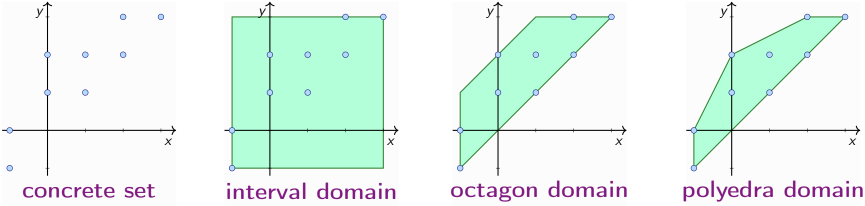
- previous lectures: $\mathbb{X}$ **is fixed and finite** and, usually, $\mathbb{V}$ **is integers**
- thus, $\mathbb{M} \equiv \mathbb{V}^n$

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

- **types:** $t_0, t_1, \ldots$
- **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:**

$$\mathbb{M} = \mathbb{X}_{t_0} \to \mathbb{V}_{t_0} \times \mathbb{X}_{t_1} \to \mathbb{V}_{t_1} \ldots$$

- At a later point, we will add **pointers**:
  t_0 **denotes pointers,** $\mathbb{V} = \ldots \uplus \mathbb{V}_{\mathrm{addr}}$
- For a moment, we let t_0 be integers, and t_1 be booleans

Heterogeneous memory states: non relational abstraction

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

Non relational abstraction of heterogeneous memory states

- $\mathbb{M} \equiv \mathbb{M}_{t_0} \times \mathbb{M}_{t_1} \times \ldots$ where $\mathbb{M}_{t_i} = \mathbb{X}_{t_i} \rightarrow \mathbb{V}_{t_i}$
- **Concretization function** (case with two types)

$$\begin{array}{rrcl} \gamma_{\rm nr} : & \mathcal{P}(\mathbb{M}_{t_0}) \times \mathcal{P}(\mathbb{M}_{t_1}) & \longrightarrow & \mathcal{P}(\mathbb{M}) \\ & (m_0^\sharp, m_1^\sharp) & \longmapsto & \{(m_{t_0}, m_{t_1}) \mid \forall i,\ m_{t_i} \in \gamma_i(m_i^\sharp)\} \end{array}$$

Example: $\mathbb{V} = \mathbb{V}_{\rm int} \uplus \mathbb{V}_{\rm bool}$, thus, $\mathbb{M} = \mathbb{M}_{\rm int} \times \mathbb{M}_{\rm bool}$

Abstraction of $\mathcal{P}(\mathbb{X}_{\rm int} \rightarrow \mathbb{V}_{\rm int})$:

- intervals
- polyhedra...

Abstraction of $\mathcal{P}(\mathbb{X}_{\rm bool} \rightarrow \mathbb{V}_{\rm bool})$:

- lattice of boolean constants
- relational abstraction with BDDs

How about a relational analysis ?

Memory structures

- The definition $\mathbb{M} = \mathbb{X} \to \mathbb{V}$ is **too restrictive**
- It ignores many ways of organizing data in the memory states

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**:
  numeric values corresponding to the address of a memory cell
- **Strings and buffers**:
  blocks with a sequence of elements and a terminating element (e.g., *null character*)
- **Closures** (functional languages):
  pointer to function code and (partial) list of arguments)

Specific properties to verify

Memory safety

Absence of memory errors (crashes, or undefined behaviors)

Pointer errors:

- Dereference of a **null pointer**
- Dereference of an **invalid pointer**

Access errors:

- Access to an array **out of its bounds**
- **Buffer overrun** (very commonly used for attacks)

Invariance properties

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

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

- memory safety
- `l` is supposed to store all file descriptors at all times
  Will its structure be preserved ?
  Yes, no breakage of a `next` link
- 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., balancing of `Maps` was wrong in the OCaml standard library...

Issues to consider in this lecture

- Propose a **concrete model**: expressive, intuitive...
- Abstract the **layout of memory states**
  i.e., what is the structure of the data
- Abstract the **contents of data structures**
- Express **relations** among various elements
  e.g., structural properties and properties of the contents of the structures
- Desgin **abstract interpretation algorithms**
  - transfer functions
  - widening

Outline

1. Towards memory properties

2. Memory models
   - Formalizing concrete memory states
   - Treatment of errors
   - Language semantics

3. Abstraction of arrays

4. Abstraction of strings and buffers

5. Basic pointer analyses

6. Three valued logic heap abstraction

7. Conclusion

A more realistic model

Not all memory cell corresponds to a variable

- a variable may correspond to **several cells**
- **heap allocated cells** correspond to no variable at all...

Environment + Heap

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

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

  h is actually only a partial function

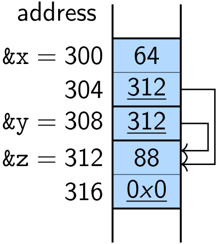
- **Memory states:** $\mathbb{M} = \mathbb{E} \times \mathbb{H}$

Example of a concrete memory state (variables)

- `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)

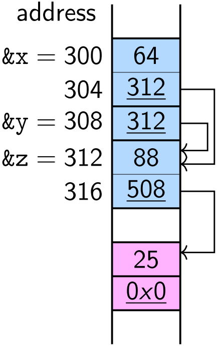


$$
\begin{array}{rlcl}
e: & \mathtt{x} & \mapsto & 300 \\
 & \mathtt{y} & \mapsto & 308 \\
 & \mathtt{z} & \mapsto & 312 \\
 & & & \\
m: & 300 & \mapsto & 64 \\
 & 304 & \mapsto & 312 \\
 & 308 & \mapsto & 312 \\
 & 312 & \mapsto & 88 \\
 & 316 & \mapsto & 0
\end{array}
$$

Example of a concrete memory state (variables + heap)

- same configuration
- + z points to a heap allocated list element (in purple)

Memory layout



$$
\begin{array}{rlll}
e: & \mathtt{x} & \mapsto & 300 \\
& \mathtt{y} & \mapsto & 308 \\
& \mathtt{z} & \mapsto & 312 \\
\\
m: & 300 & \mapsto & 64 \\
& 304 & \mapsto & 312 \\
& 308 & \mapsto & 312 \\
& 312 & \mapsto & 88 \\
& 316 & \mapsto & 508 \\
& 508 & \mapsto & 25 \\
& 512 & \mapsto & 0
\end{array}
$$

Extending the language syntax

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

$$
\begin{array}{rcll}
\mathtt{l} & ::= & \textbf{l-valules} & \\
 & \mid & \mathtt{x} & (\mathtt{x} \in \mathbb{X}) \\
 & \mid & \ldots & \textbf{other kinds of l-values} \\
 & & & \textbf{pointers, array dereference...} \\
\mathtt{e} & ::= & \textbf{expressions} & \\
 & \mid & c & (c \in \mathbb{V}) \\
 & \mid & \mathtt{l} & (\mathit{lvalue}) \\
 & \mid & \mathtt{e} \oplus \mathtt{e} & (\mathit{arithoperation}, \mathit{comparison}) \\
\mathtt{s} & ::= & \textbf{statements} & \\
 & \mid & \mathtt{l} = \mathtt{e} & (\text{assignment}) \\
 & \mid & \mathtt{s}; \ldots \mathtt{s}; & (\text{sequence}) \\
 & \mid & \textbf{if}(\mathtt{e})\{\mathtt{s}\} & (\text{condition}) \\
 & \mid & \textbf{while}(\mathtt{e})\{\mathtt{s}\} & (\text{loop})
\end{array}
$$

Extending the language semantics

Some slight modifications to the semantics of the initial language:

- **Values are addresses:** $\mathbb{V}_{\mathrm{addr}} \subseteq \mathbb{V}$
- **L-values evaluate into addresses:** $[\![\mathtt{l}]\!] : \mathbb{M} \to \mathbb{V}_{\mathrm{addr}}$

$$[\![\mathtt{x}]\!](e, h) \;=\; e(\mathtt{x})$$

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

$$[\![\mathtt{l}]\!](e, h) \;=\; m([\![\mathtt{l}]\!](e, h))$$

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

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

where

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

Extensions of the symbolic model

Our model is still not quite realistic

- 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

- **Division** of the memory in **blocks** with a **base address** and a **size**
- **Division** 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 concrete memory states:
see **CompCert** project source files (Coq formalization)

Language semantics: program crash

- In an abnormal situation, **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:

- out-of-bound array access: exception `java.lang.ArrayIndexOutOfBoundsException`

Language semantics: undefined behaviors

- The behavior of the program is **not specified** when an abnormal situation is encountered
- Advantage: easy implementation (often architecture driven)
- Disadvantage: unintuitive semantics, errors hard to reproduce

Modeling of undefined behavior

- Very hard to capture what a program operation may modify
- Abnormal situation at (l_0, m_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 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 its 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

- Our purpose is not to select a language for programming
- It is to remark salient language features, and their impact on abstractions

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...

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

Pointer arithmetics

What kind of operations can be performed on a pointer ?

Classical pointer operations

- Pointer **dereference**:
  $\star$p returns the contents of the cell pointed to by 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**:
  $\mathrm{p} + n$: address contained in $\mathrm{p} + n$ times the size of the type of p
  Interaction with pointer casts...
- **Pointer subtraction**: returns a numeric offset

String operations

- Many **data-structures** can be handled in very different ways depending on the languages
- **Strings** are just one example

OCaml strings

- **Abstract type**: representation not part of the language definition
- **Type safe** implementation
  - no buffer orverrun
  - exception for out of bound accesses
    i.e., like arrays
- Most operations **generate new string structures**

C strings

- A **string** is an **array of characters** (**char** $\star$) with a **terminal zero character**
- **Direct access** to string elements (array dereference)
- String copy operation **strcpy**$(\mathrm{s}, \text{"foo\_bar"})$:
  - copies "foo_bar" into s
  - **undefined behavior** if length of s < 7

Manual memory management

Allocation of unbounded memory space

- How are new memory blocks made available to 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

List of choices:

- **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 objct fields**: allowed or not
- **Pointer arithmetic**: allowed or not
  i.e., are pointer values symbolic values or numeric values
- **Memory management**: automatic or manual

We will generally assume a simple model,
unless considering specific features

Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
    - A micro language for manipulating arrays
    - Verifying safety of array operations
    - Abstraction of array contents
    - Abstraction of array properties
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction

Programs: extension with arrays

Extension of the syntax:

$$\begin{array}{rcll} \texttt{l} & ::= & \textbf{l-valules} & \\ & | & \ldots & \text{previous constructions} \\ & | & \mathtt{x}[\mathrm{e}] & \text{cell of array } \mathtt{x} \\ \ldots & ::= & \ldots & \text{the rest is unchanged} \end{array}$$

Extension of the semantics:

- if x is an array variable, and corresponds to an array of length N, we have N cells corresponding to it, with addresses

  $$\{e(\mathrm{x}) + 0, e(\mathrm{x}) + s, \ldots, e(\mathrm{x}) + (N-1)s$$

  where s is the size of an array cell (e.g., 8 bytes for a 64-bit int)
- **evaluation of an array cell read:**

$$[\![\mathrm{x}[\mathrm{e}]]\!](e, h) = \begin{cases} e(\mathrm{x}) + is & \text{if } [\![\mathrm{e}]\!](e, h) = i \in [0, N-1] \\ \Omega & \text{otherwise} \end{cases}$$

Example

```
// a is an integer array of length n
bool s;
do{
    s = false;
    for(int i = 0; i < n - 1; i = i + 1){
        if(a[i] < a[i + 1]){
            swap(a[i] < a[i + 1]);
            s = true;
        }
    }
} while(s);
```

Properties to verify by static analysis

1. **Safety property:** the program will not crash (no index out of bound)
2. **Contents property:** if the values in the array are in $[0, 100]$ before, they are also in that range after
3. **Global array property:** at the end, the array is sorted

Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
    - A micro language for manipulating arrays
    - Verifying safety of array operations
    - Abstraction of array contents
    - Abstraction of array properties
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction

Expressing correctness of array operations

Goal of the analysis: establish safety

Prove the absence of runtime error due to array reads / writes
i.e., **that no Ω will ever arise**

Safety verification:

- At label ℓ, the analysis computes a **local abstraction of the set of reachable memory states** $\Phi^\sharp(\ell)$
- If a statement at label ℓ performs array read or write operation $\mathtt{x}[e]$, where $\mathtt{x}$ is an array of length n, the analysis simply needs to establish
$$\forall m \in \gamma_{\mathrm{mem}}(\Phi^\sharp(\ell)),\ [\![e]\!](m) \in [0, n-1]$$
- In many cases, this can be done with an **interval abstraction**
  ... but not always (Exercise: when would it not be enough ?)

For now, we ignore the contents of the array
(Exercise: when does this fail ?)

Verifying correctness of array operations

Case where intervals are enough:

```
//x array of length 40
int i = 0;
while(i < 40){
    printf("%d;", x[i]);
    i = i + 1;
}
```

- interval analysis establishes that $\mathtt{i} \in [0;39]$ at the loop head
- this allows the verification of the code

Case where intervals cannot represent precise enough invariants:

```
//x array of length 40
int i, j;
if(0 ≤ i && i < j)
    if(j < 41)
        printf("%d;", x[i]);
```

- in the concrete, $\mathtt{i} \in [0,39]$ at the array access point
- to establish this in the abstract, after the first test, relation $\mathtt{i} < \mathtt{j}$ need be represented
- e.g., octagon abstract domain

Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
    - A micro language for manipulating arrays
    - Verifying safety of array operations
    - Abstraction of array contents
    - Abstraction of array properties
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction

Elementwise abstraction

Goal of the analysis: abstract contents

Inferring invariants about the **contents** of the array

- e.g., that the values in the array **are in a given range**
- e.g., in order to verify the **safety of** $\mathtt{x}[\mathtt{y}[i+j]+k]$ or $\mathtt{y} = n/\mathtt{x}[\mathtt{i}]$

Assumption:

- **One array** $\mathtt{t}$, of **known, fixed length** n (element size s)
- Scalar variables $\mathtt{x}_0, \mathtt{x}_1, \ldots, \mathtt{x}_{m-1}$

Concrete memory cell addresses:

$$\mathbb{V}_{\mathrm{addr}} = \{\&\mathtt{x}_0, \ldots, \&\mathtt{x}_{m-1}\} \cup \{\&\bar{\mathtt{t}}, \&\bar{\mathtt{t}} + 1 \cdot s, \ldots, \&\bar{\mathtt{t}} + (n-1) \cdot s\}$$

Elementwise abstraction

- **Each** concrete cell is **mapped into one abstract cell**
- $\mathbb{D}^\sharp$ should simply be an **abstraction of** $\mathcal{P}(\mathbb{V}^{m+n})$

Array smashing abstraction: abstraction into one cell

The elementwise abstraction is **too costly**:

- **high number of abstract cells** if the arrays are big
- **will not work** if the size of arrays is **not known statically**

Alternative: **use fewer abstract cells**, e.g., **a single cell**

Assumption: m scalar variables, $\bar{\mathtt{t}}$ array of length n

Array smashing

- All cells of the array are mapped into **one abstract cell** $\bar{\mathtt{t}}$
- **Abstract cells**: $\mathbb{C}^\sharp = \{\&\mathtt{x}_0, \ldots, \&\mathtt{x}_{m-1}\} \cup \{\&\bar{\mathtt{t}}\}$
- $\mathbb{D}^\sharp$ should simply be an **abstraction of $\mathcal{P}(\mathbb{V}^{m+1})$**

This also works **if the size of the array is not known statically**:

```
int n = ...;
int t[n];
```

The contents of t is represented using one abstract cell whathever the value of n

Array smashing abstraction

Definition

- **Abstract domain** $\mathcal{P}(\mathbb{C}^\sharp \to \mathcal{P}(\mathbb{V}))$
- **Abstraction function:**

$$\alpha_{\mathrm{smash}}(H) = \left\{ \begin{array}{ccc} \&\mathtt{x}_i & \mapsto & \{\hbar(\mathtt{x}_i)\} \\ \&\overline{\mathtt{t}} & \mapsto & \{\hbar(\&\mathtt{t}+0), \ldots, \hbar(\&\mathtt{t}+n-1)\} \end{array} \middle| \hbar \in H \right\}$$

Example:

- No variable, array of length 2
- **Set of concrete states:**

$$\left\{ \begin{array}{ccc} \mathtt{t}[0] & \mapsto & 0 \\ \mathtt{t}[1] & \mapsto & 10 \end{array} \right\}, \left\{ \begin{array}{ccc} \mathtt{t}[0] & \mapsto & 2 \\ \mathtt{t}[1] & \mapsto & 11 \end{array} \right\}, \left\{ \begin{array}{ccc} \mathtt{t}[0] & \mapsto & 1 \\ \mathtt{t}[1] & \mapsto & 12 \end{array} \right\}$$

- **Abstract state**, using interval abstraction: $\&\overline{\mathtt{t}} \mapsto [0, 12]$

Weak updates: an imprecision in the analysis

Assumptions:

- **Smashing abstraction**, with the **interval abstract domain**
- Array `t` is supposed **of known length** $n \geq 2$
- We consider statement $l_0 : \mathtt{t[i]} = 0;\ l_1$
- Given $m_0^\sharp$, using intervals to describe a set of states at l, we wish to compute an over-approximation $m_1^\sharp$ of
$$\{m_1 \mid \exists m_0 \in \gamma_{\mathrm{mem}}(m_0^\sharp),\ (l_0, m_0) \rightarrow (l_1, m_1)\}$$
- **Abstract pre-condition:** $m_0^\sharp(\&\mathtt{i}) = [0, 0], m_0^\sharp(\&\overline{\mathtt{t}}) = [a, b]$

Post-condition:

- in the concrete level:
$$\begin{cases} \&\mathtt{t} + 0 & \longmapsto & 0 & \text{(cell just modified)} \\ \&\mathtt{t} + 1 & \longmapsto & v & \text{where } v \in [a, b] \text{ (cell not modified)} \end{cases}$$
- in the abstract level, we **only lose precision**:
$$\&\overline{\mathtt{t}} \longmapsto [0, 0] \sqcup [a, b] = [\min(a, 0), \max(b, 0)]$$

Weak updates

Summary:

- `i` was known very precisely
- `&t̄` **stands for several concrete cells**
- The assignment will modify only **one cell**
  the others will keep their old value
- The abstraction cannot distinguish unmodified values from the modified cell
- As a consequence, the range for `&t̄` **may only grow**

Weak updates

- It would only be worse if the value of `i` was not known precisely
- This is a significant loss in precision
- This is a **limitation of all smashing analyses**

Weak updates and strong updates

Definitions

- **Strong update:** modified abstract cell **fully materialized**, and **old value fully discarded**
- **Weak update:** modified abstract cell **not fully materialized**, and **new value "joined" with old values**

In the case of $\mathtt{t}[\mathtt{i}] := e$, weak updates may arise in the following cases:

- using a **smashing abstraction**:
  $\overline{\mathtt{t}}$ denotes several concrete cells; only one gets modified, so we must keep old values
- using a **pointwise abstraction**, if $m_0^\sharp(\mathtt{i}) = [i, i']$ where $i < i'$:
  - one cell in $\{\&\mathtt{t} + i \cdot s, \ldots, \&\mathtt{t} + i' \cdot s\}$ gets modified
  - the other cells in that set remain the same
  - so we must also keep old values

Weak updates and strong updates: example

```
//x uninitialized array of length n
int i = 0;
while(i < n){
    x[i] = 0;
    i = i + 1;
}
```

Pointwise abstraction:

- initially $\forall i,\ m^\sharp(\&\mathtt{t} + i \cdot s) = \top$
- if loop unrolled completely, at the end, $\forall i,\ m^\sharp(\&\mathtt{t} + i \cdot s) = [0, 0]$
- weak updates, if the loop is not unrolled; then, at the end $\forall i,\ m^\sharp(\&\mathtt{t} + i \cdot s) = \top$

Smashing abstraction:

- initially $m^\sharp(\bar{\mathtt{t}}) = \top$
- weak updates at each step (whatever the unrolling that is performed); at the end: $m^\sharp(\bar{\mathtt{t}}) = \top$

- Weak updates may cause a serious loss of precision
- Workaround ahead: more complex array abstractions may help

Other forms of array smashing

- **Smashing does not have to affect the whole array**
- Efficient smashing strategies can be found

Segment smashing:

- abstraction of the array cells into $\{\bar{\mathtt{t}}_0, \ldots, \bar{\mathtt{t}}_{k-1}\}$ where $\bar{\mathtt{t}}_i$ corresponds to **a segment of the array**
- useful when sub-segments have interesting properties
- **issue**: determine the segment by analysis

Modulo smashing:

- abstraction of the array cells into $\{\bar{\mathtt{t}}_0, \ldots, \bar{\mathtt{t}}_{k-1}\}$ where $\bar{\mathtt{t}}_i$ corresponds to **a repeating set of offsets** $\{\&\bar{\mathtt{t}} + k \cdot i \cdot s \mid k \cdot i < n\}$
- useful for arrays of structures
- **issue**: determine k by analysis

Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
    - A micro language for manipulating arrays
    - Verifying safety of array operations
    - Abstraction of array contents
    - Abstraction of array properties
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction

Example array properties

Goal of the analysis: precisely abstract contents

Discover non trivial properties of **array regions**

- Initialization to a constant (e.g., 0)
- Sortedness

An array initialization loop:

```
// t integer array of length n
int i = 0;
while(i < n){
    t[i] = 0;
    i = i + 1;
}
```

Sketch of a hand proof:

- At iteration i, $\mathtt{i} = i$ and the segment $\mathtt{t}[0], \ldots \mathtt{t}[i-1]$ is initialized
- At the loop exit, $\mathtt{i} = n$ and the whole array is initialized

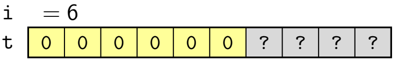
We need to express properties on segments;
otherwise the proof cannot be completed

Array segment properties

An array initialization loop:

```
// t integer array of length $n$
int i = 0;
while(i < $n$){
    t[i] = 0;
    i = i + 1;
}
```

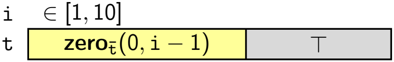
Concrete state after 6 iterations:



$\mathtt{i} \quad = 6$

t	0	0	0	0	0	0	?	?	?	?

Corresponding abstract state:



$\mathtt{i} \quad \in [1, 10]$

t	$\textbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i}-1)$	$\top$

Array segment predicates

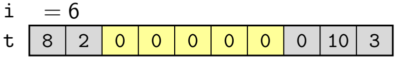
Definition

An **array segment predicate** is an abstract predicate that describes the contents of a contiguous series of cells in the array, such as:

- **Initialization**: $\mathbf{zero}_{\mathtt{t}}(i,j)$ iff t initialized to 0 between i and j
- **Sortedness**: $\mathbf{sort}_{\mathtt{t}}(i,j)$ iff t sorted between i and j

Examples:

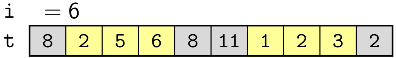
- array satisfying $\mathbf{zero}_{\mathtt{t}}(2,6)$:



i = 6

t	8	2	0	0	0	0	0	0	10	3

- array satisfying $\mathbf{sort}_{\mathtt{t}}(1,4)$ **and** $\mathbf{sort}_{\mathtt{t}}(6,8)$:



i = 6

t	8	2	5	6	8	11	1	2	3	2

Composing sortedness predicates

As part of the proof, predicates need be composed

$$\begin{array}{rcl}
\mathbf{zero}_{\mathtt{t}}(i,j) \wedge \mathbf{zero}_{\bar{\mathtt{t}}}(j+1,k) & \Rightarrow & \mathbf{zero}_{\mathtt{t}}(i,k) \\
\mathbf{zero}_{\mathtt{t}}(i,j) \wedge \mathtt{t}[j+1] = 0 & \Rightarrow & \mathbf{zero}_{\mathtt{t}}(i,j+1) \\
\mathbf{sort}_{\mathtt{t}}(i,j) \wedge \mathbf{sort}_{\bar{\mathtt{t}}}(j+1,k) & \not\Rightarrow & \mathbf{sort}_{\mathtt{t}}(i,k) \\
\mathtt{t}[j] \leq \mathtt{t}[j+1] \wedge \mathbf{sort}_{\mathtt{t}}(i,j) \wedge \mathbf{sort}_{\bar{\mathtt{t}}}(j+1,k) & \Rightarrow & \mathbf{sort}_{\mathtt{t}}(i,k)
\end{array}$$

- **counter example** for the third line: for $[0;3;9;2;4;8]$, we have:

$$\mathbf{sort}_{\mathtt{t}}(0,2) \wedge \mathbf{sort}_{\mathtt{t}}(3,5) \qquad \text{but not} \qquad \mathbf{sort}_{\mathtt{t}}(0,5)$$

Another sortedness predicate: $\mathbf{sort}_{\mathtt{t}}(i,j,\mathrm{min},\mathrm{max})$

$$B \leq C \wedge \mathbf{sort}_{\mathtt{t}}(i,j,A,B) \wedge \mathbf{sort}_{\bar{\mathtt{t}}}(j+1,k,C,D) \;\Rightarrow\; \mathbf{sort}_{\mathtt{t}}(i,k,A,D)$$

Analysis operators (for predicate **zero**)

Assignment transfer function:

1. Identify segments that may be modified
2. If a single segment is impacted, split it
3. Do a strong update

$$\begin{array}{rcl} \mathbf{zero}_{\mathtt{t}}(0,\mathtt{n}) \wedge 0 \leq \mathtt{i} < \mathtt{n} & \xrightarrow{\mathtt{t[i]}=?} & \mathbf{zero}_{\mathtt{t}}(0,\mathtt{i}-1) \wedge \mathbf{zero}_{\mathtt{t}}(\mathtt{i}+1,\mathtt{n}) \wedge 0 \leq \mathtt{i} < \mathtt{n} \\ \top \wedge 0 \leq \mathtt{i} < \mathtt{n} & \xrightarrow{\mathtt{t[i]}=0} & \mathbf{zero}_{\mathtt{t}}(\mathtt{i},\mathtt{i}) \wedge 0 \leq \mathtt{i} < \mathtt{n} \end{array}$$

Abstract join operator: generalizes bounds

$$(\top \wedge \mathtt{i} = 0 < n) \sqcup^{\sharp} (\mathbf{zero}_{\mathtt{t}}(0,0) \wedge \mathtt{i} = 1 < n) \quad = \quad (\mathbf{zero}_{\mathtt{t}}(0,\mathtt{i}-1) \wedge 0 \leq \mathtt{i} < n)$$

Array analysis: example

```
// t integer array of length n > 0
            t [        ⊤        ]        i  ⊤
int i = 0;
            t [        ⊤        ]        i  ⊤
while(i < n){
            t [        ⊤        ]        i  ⊤
    t[i] = 0;
            t [        ⊤        ]        i  ⊤
    i = i + 1;
            t [        ⊤        ]        i  ⊤
}
            t [        ⊤        ]        i  ⊤
```

Array analysis: example

// `t` integer array of length $n > 0$

`t` $\top$ `i` $\top$

int `i` $= 0$;

`t` $\top$ `i` $[0, 0]$

while(`i` $< n$){

`t` $\top$ `i` $\top$

`t[i]` $= 0$;

`t` $\top$ `i` $\top$

`i` $=$ `i` $+ 1$;

`t` $\top$ `i` $\top$

}

`t` $\top$ `i` $\top$

Array analysis: example

// `t` integer array of length $n > 0$

`t` $\top$ `i` $\top$

int `i` $= 0$;

`t` $\top$ `i` $[0, 0]$

while(`i` $< n$){

`t` $\top$ `i` $[0, 0]$

`t[i]` $= 0$;

`t` $\top$ `i` $\top$

`i` $=$ `i` $+ 1$;

`t` $\top$ `i` $\top$

}

`t` $\top$ `i` $\top$

Array analysis: example

// `t` integer array of length $n > 0$

`t` $\top$ `i` $\top$

int `i` $= 0$;

`t` $\top$ `i` $[0, 0]$

while(`i` $< n$){

`t` $\top$ `i` $[0, 0]$

  `t[i]` $= 0$;

`t` | **zero**$_{\bar{t}}(0, 1)$ | $\top$ | `i` $[0, 0]$

  `i` $=$ `i` $+ 1$;

`t` $\top$ `i` $\top$

}

`t` $\top$ `i` $\top$

Array analysis: example

// `t` integer array of length $n > 0$

t $\top$ i $\top$

int `i` $= 0$;

t $\top$ i $[0, 0]$

while$(\mathtt{i} < n)\{$

t $\top$ i $[0, 0]$

  $\mathtt{t}[\mathtt{i}] = 0$;

t **zero**$_{\bar{\mathtt{t}}}(0, 1)$ | $\top$ i $[0, 0]$

  $\mathtt{i} = \mathtt{i} + 1$;

t **zero**$_{\bar{\mathtt{t}}}(0, 1)$ | $\top$ i $[1, 1]$

$\}$

t $\top$ i $\top$

Array analysis: example

// t integer array of length $n > 0$

t $\boxed{\top}$ i $\top$

int i = 0;

t $\boxed{\mathbf{zero}_{\bar{t}}(0, i-1)}\boxed{\top}$ i $[0, 1]$

while(i < n){

t $\boxed{\top}$ i $[0, 0]$

t[i] = 0;

t $\boxed{\mathbf{zero}_{\bar{t}}(0, 1)}\boxed{\top}$ i $[0, 0]$

i = i + 1;

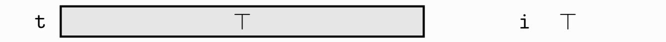
t $\boxed{\mathbf{zero}_{\bar{t}}(0, 1)}\boxed{\top}$ i $[1, 1]$

}

t $\boxed{\top}$ i $\top$

Array analysis: example

// `t` integer array of length $n > 0$



int $\mathtt{i} = 0$;

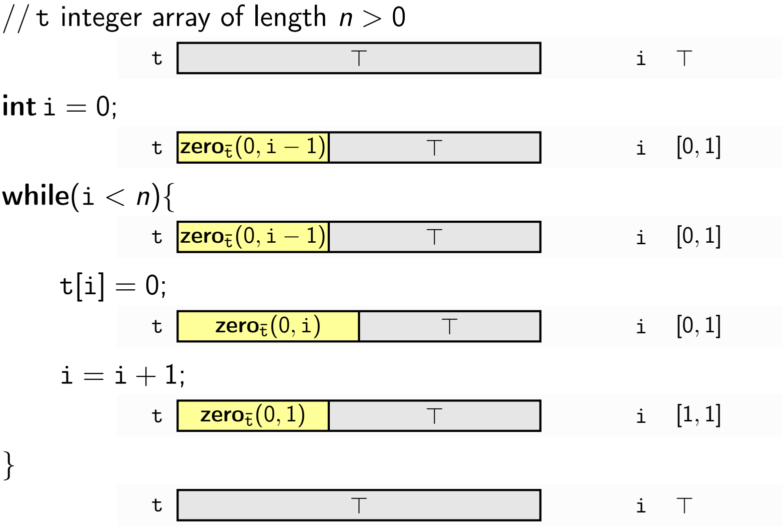
while$(\mathtt{i} < n)\{$

$\mathtt{t}[\mathtt{i}] = 0$;

$\mathtt{i} = \mathtt{i} + 1$;

$\}$

Array analysis: example



// $\mathtt{t}$ integer array of length $n > 0$

$\mathtt{t}$ $\top$ $\mathtt{i}$ $\top$

$\textbf{int}\ \mathtt{i} = 0;$

$\mathtt{t}$ $\textbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i} - 1)$ $\top$ $\mathtt{i}$ $[0, 1]$

$\textbf{while}(\mathtt{i} < n)\{$

$\mathtt{t}$ $\textbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i} - 1)$ $\top$ $\mathtt{i}$ $[0, 1]$

$\mathtt{t}[\mathtt{i}] = 0;$

$\mathtt{t}$ $\textbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i})$ $\top$ $\mathtt{i}$ $[0, 1]$

$\mathtt{i} = \mathtt{i} + 1;$

$\mathtt{t}$ $\textbf{zero}_{\bar{\mathtt{t}}}(0, 1)$ $\top$ $\mathtt{i}$ $[1, 1]$

$\}$

$\mathtt{t}$ $\top$ $\mathtt{i}$ $\top$

Array analysis: example

$// \; \mathtt{t}$ integer array of length $n > 0$

$$\mathtt{t}\ \boxed{\top} \qquad \mathtt{i}\ \ \top$$

int $\mathtt{i} = 0;$

$$\mathtt{t}\ \boxed{\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i}-1)}\boxed{\top} \qquad \mathtt{i}\ \ [0, 1]$$

while$(\mathtt{i} < n)\{$

$$\mathtt{t}\ \boxed{\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i}-1)}\boxed{\top} \qquad \mathtt{i}\ \ [0, 1]$$

$\mathtt{t}[\mathtt{i}] = 0;$

$$\mathtt{t}\ \boxed{\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i})}\boxed{\top} \qquad \mathtt{i}\ \ [0, 1]$$

$\mathtt{i} = \mathtt{i} + 1;$

$$\mathtt{t}\ \boxed{\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i}-1)}\boxed{\top} \qquad \mathtt{i}\ \ [1, 2]$$

$\}$

$$\mathtt{t}\ \boxed{\top} \qquad \mathtt{i}\ \ \top$$

Array analysis: example

// $\mathtt{t}$ integer array of length $n > 0$

$\mathtt{t}$ | $\top$ | $\mathtt{i}$ $\top$

int $\mathtt{i} = 0;$

$\mathtt{t}$ | $\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i} - 1)$ | $\top$ | $\mathtt{i}$ $[0, \mathtt{n}]$

while$(\mathtt{i} < n)\{$

$\mathtt{t}$ | $\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i} - 1)$ | $\top$ | $\mathtt{i}$ $[0, 1]$

$\mathtt{t}[\mathtt{i}] = 0;$

$\mathtt{t}$ | $\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i})$ | $\top$ | $\mathtt{i}$ $[0, 1]$

$\mathtt{i} = \mathtt{i} + 1;$

$\mathtt{t}$ | $\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i} - 1)$ | $\top$ | $\mathtt{i}$ $[1, 2]$

$\}$

$\mathtt{t}$ | $\top$ | $\mathtt{i}$ $\top$

Array analysis: example

// $\mathtt{t}$ integer array of length $n > 0$

$\mathtt{t}$ $\boxed{\top}$ $\quad$ $\mathtt{i}$ $\top$

int $\mathtt{i} = 0;$

$\mathtt{t}$ $\boxed{\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i} - 1)}\boxed{\top}$ $\quad$ $\mathtt{i}$ $[0, \mathtt{n}]$

while$(\mathtt{i} < n)\{$

$\mathtt{t}$ $\boxed{\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i} - 1)}\boxed{\top}$ $\quad$ $\mathtt{i}$ $[0, \mathtt{n} - 1]$

$\quad\mathtt{t}[\mathtt{i}] = 0;$

$\mathtt{t}$ $\boxed{\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i})}\boxed{\top}$ $\quad$ $\mathtt{i}$ $[0, 1]$

$\quad\mathtt{i} = \mathtt{i} + 1;$

$\mathtt{t}$ $\boxed{\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i} - 1)}\boxed{\top}$ $\quad$ $\mathtt{i}$ $[1, 2]$

$\}$

$\mathtt{t}$ $\boxed{\top}$ $\quad$ $\mathtt{i}$ $\top$

Array analysis: example

// $\mathtt{t}$ integer array of length $n > 0$

$\mathtt{t}\ \boxed{\top}$ $\quad\mathtt{i}\ \top$

int $\mathtt{i} = 0;$

$\mathtt{t}\ \boxed{\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i}-1)}\boxed{\top}$ $\quad\mathtt{i}\ [0, \mathtt{n}]$

while$(\mathtt{i} < n)\{$

$\mathtt{t}\ \boxed{\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i}-1)}\boxed{\top}$ $\quad\mathtt{i}\ [0, \mathtt{n}-1]$

$\quad\mathtt{t}[\mathtt{i}] = 0;$

$\mathtt{t}\ \boxed{\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i})}\boxed{\top}$ $\quad\mathtt{i}\ [0, \mathtt{n}-1]$

$\quad\mathtt{i} = \mathtt{i} + 1;$

$\mathtt{t}\ \boxed{\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i}-1)}\boxed{\top}$ $\quad\mathtt{i}\ [1, 2]$

$\}$

$\mathtt{t}\ \boxed{\top}$ $\quad\mathtt{i}\ \top$

Array analysis: example

// $\mathtt{t}$ integer array of length $n > 0$

$\mathtt{t}$ $\top$ $\mathtt{i}$ $\top$

int $\mathtt{i} = 0;$

$\mathtt{t}$ $\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i} - 1)$ $\top$ $\mathtt{i}$ $[0, \mathtt{n}]$

while$(\mathtt{i} < n)\{$

$\mathtt{t}$ $\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i} - 1)$ $\top$ $\mathtt{i}$ $[0, \mathtt{n} - 1]$

$\mathtt{t}[\mathtt{i}] = 0;$

$\mathtt{t}$ $\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i})$ $\top$ $\mathtt{i}$ $[0, \mathtt{n} - 1]$

$\mathtt{i} = \mathtt{i} + 1;$

$\mathtt{t}$ $\mathbf{zero}_{\bar{\mathtt{t}}}(0, \mathtt{i} - 1)$ $\top$ $\mathtt{i}$ $[1, \mathtt{n}]$

$\}$

$\mathtt{t}$ $\top$ $\mathtt{i}$ $\top$

Array analysis: example

$//\ \mathtt{t}$ integer array of length $n > 0$

$\mathtt{t}$ $\top$ $\qquad$ $\mathtt{i}$ $\top$

$\textbf{int}\ \mathtt{i} = 0;$

$\mathtt{t}$ $\mathbf{zero}_{\overline{\mathtt{t}}}(0, \mathtt{i} - 1)$ $\top$ $\qquad$ $\mathtt{i}$ $[0, \mathtt{n}]$

$\textbf{while}(\mathtt{i} < n)\{$

$\mathtt{t}$ $\mathbf{zero}_{\overline{\mathtt{t}}}(0, \mathtt{i} - 1)$ $\top$ $\qquad$ $\mathtt{i}$ $[0, \mathtt{n} - 1]$

$\quad \mathtt{t}[\mathtt{i}] = 0;$

$\mathtt{t}$ $\mathbf{zero}_{\overline{\mathtt{t}}}(0, \mathtt{i})$ $\top$ $\qquad$ $\mathtt{i}$ $[0, \mathtt{n} - 1]$

$\quad \mathtt{i} = \mathtt{i} + 1;$

$\mathtt{t}$ $\mathbf{zero}_{\overline{\mathtt{t}}}(0, \mathtt{i} - 1)$ $\top$ $\qquad$ $\mathtt{i}$ $[1, \mathtt{n}]$

$\}$

$\mathtt{t}$ $\mathbf{zero}_{\overline{\mathtt{t}}}(0, \mathtt{n} - 1)$ $\qquad$ $\mathtt{i}$ $[\mathtt{n}, \mathtt{n}]$

Partitioning of arrays

Array partitions

A partition of an array `t` of length n is a sequence $\mathcal{P} = \{e_0, \ldots, e_k\}$ of symbolic expressions where

- e_i denotes the lower (resp., upper) bound of element i (resp. $i-1$) of the partition
- e_0 should be equal to 0 (and e_k to n)

Example:

- set of four **concrete states**:

$$\left\{\begin{array}{llcll} \mathtt{i} = 1 & [0,4,1,2,3,5] & & \mathtt{i} = 3 & [2,2,4,5,1,8] \\ \mathtt{i} = 2 & [0,1,5,2,3,4] & & \mathtt{i} = 5 & [0,2,4,6,7,9] \end{array}\right.$$

- **partition**: $\{0, \mathtt{i}+1, 6\}$
- note that the array is always
  - sorted between 0 and $\mathtt{i}$
  - sorted between $\mathtt{i}+1$ and 5

Abstraction based on array partitions

Segment and array abstraction

An array segmentation is given by a partition $\mathcal{P} = \{e_0, \ldots, e_k\}$ and a set of abstract properties $\{P_0, \ldots, P_{k-1}\}$.
Its concretization is the set of memory states $m = (e, h)$ such that

$$\forall i,\ [\mathtt{t}[v], \mathtt{t}[v+1], \ldots, \mathtt{t}[w-1]] \text{ satisfies } P_i,\ \text{where} \left\{ \begin{array}{lcl} v & = & [\![e_i]\!](m) \\ w & = & [\![e_{i+1}]\!](m) \end{array} \right.$$

- **Partitions can be**:
  - **static**, i.e., pre-computed by another analysis **[HP'08]**
  - **dynamic**, i.e., computed as part of the analysis **[CCL'11]** (more complex abstract domain structure with partitions *and* predicates)
- **Example**: array initialization

Outline

1 Towards memory properties

2 Memory models

3 Abstraction of arrays

4 Abstraction of strings and buffers
  - A micro-language with strings
  - Abstraction

5 Basic pointer analyses

6 Three valued logic heap abstraction

7 Conclusion

Strings in programming languages

- In **high-level programming languages**:
  - **high-level** API, like OCaml `String` module or Java `String` classes
  - a set of **exceptions** in case of an invalid operation
  - **no security** risk in case of a crash
- In **C**:
  - **arrays of characters**
  - integration in other structures **with no protection**
  - **direct access**, with **no protection**

We focus on the case of languages with strings *à la C*

Programs: syntax and semantics

We extend our simple language with strings...

Encoding of strings in C

- **Strings** are represented by **character arrays**, with a **terminating 0**
- Only characters to the first zero are meaningful
- Example of a **string buffer** of length 10 containing string "hello"

'h'	'e'	'l'	'l'	'o'	'/0'	'b'	'/0'	'a'	'x'

Thus, **the language is essentially the same as for arrays**:

- data-types remain the same; we include a **char** type;
- expressions and l-values remain the same too
- we consider a set of **string operations** (typically, library functions)

Programs: string operations

String operations

- $\mathtt{strcpy}(\textbf{char} \star \mathtt{d}, \textbf{char} \star \mathtt{s})$: copies s into d, including terminating 0, provided there is enough space (unspecified otherwise)
- $\mathtt{strncpy}(\textbf{char} \star \mathtt{d}, \textbf{char} \star \mathtt{s}, \textbf{int}\ \mathtt{n})$: copies exactly n characters at most, from s into d
- `printf`: interprets "%s" as a string placeholder; displays up to the terminating 0 (unspecified if there is none)

```
char q[2];
char s[2];
char t[4];
strcpy(t, "bon");
strncpy(s, t, 2);
strcpy(q, s);
printf("nres:  %s/n", q);
```

Result ?

- not fully defined
- depends on the order of memory blocks in memory...

Abstraction of string buffers

Goal of static analysis

Prove the absence of runtime errors in string buffer operations

Such errors could:

- cause **abrupt crashes** (segmentation fault) or undefined behaviors
- make **exploits** possible (e.g., by overwriting other program data)

We remark that:

- the **positions of "zero" characters** matters
- the **value of the other characters** usually does not matter
  exception: cases where the program decides what to do depending on non zero characters, and where that impacts the error behavior of the program

Numeric abstraction of strings

String characters abstractions

We consider the character abstraction below:

$$\phi : \emptyset \mapsto \emptyset \qquad \phi : c \mapsto '?'$$

$$\dot{\phi} : c_0 \cdots c_{n-1} \mapsto \phi(c_0) \cdots \phi(c_{n-1})$$

$$\alpha_{\mathrm{string}} : \mathcal{S} \mapsto \{\dot{\phi}\,(s) \mid s \in \mathcal{S}\}$$

- α_{string} abstracts unneeded characters information

Numerical abstraction

We consider memory states that comprise only one string buffer `t`. We can abstract each such state using two numbers

- $\mathtt{t_n}$: size of buffer `t`
- $\mathtt{t_z}$: position of the first 0 in `t` if any (otherwise, we let $\mathtt{t_z} = \mathtt{t_n}$)

Abstraction of string buffers

We consider a program with integer variables $\mathbb{X}_{\rm int} = \{\mathtt{x}, \mathtt{y}, \ldots\}$ and string buffer variables $\mathbb{X}_{\rm buf} = \{\mathtt{t}, \mathtt{u}, \ldots\}$

Abstract domain

- We let $\mathbb{X}' = \mathbb{X}_{\rm int} \uplus \{\mathtt{t}_{\rm n}, \mathtt{t}_{\rm z}, \mathtt{u}_{\rm n}, \mathtt{u}_{\rm z}, \ldots\}$
- Each memory state m gets abstracted into a state $m' = \textbf{abs}(m)$ over $\mathbb{X}'$
- Given an abstract domain $(\mathbb{D}^\sharp_{\rm num}, \sqsubseteq_{\rm num})$ of $\mathcal{P}(\mathbb{X}' \rightarrow \mathbb{Z})$, we can build an abstraction of $(\mathcal{P}(\mathbb{M}), \subseteq)$:

$$\begin{array}{rccl} \gamma_{\rm buf} : & \mathbb{D}^\sharp_{\rm num} & \longrightarrow & \mathcal{P}(\mathbb{M}) \\ & X^\sharp & \longmapsto & \{m \in \mathbb{M} \mid \textbf{abs}(m) \in \gamma_{\rm num}(X^\sharp)\} \end{array}$$

Typical choice: polyhedra

Example

- **Example**: abstraction of

'h'	'e'	'l'	'l'	'o'	'/0'	'b'	'/0'	'a'	'x'

  into $t_n = 10, t_z = 5$
- **Practical implementation**:
  - either as a classical static analysis
  - or using a **transformation into an integer program**
- **Code transformation approach**:

$$\left.\begin{array}{l}
\textbf{char}\; q[2];\\
\textbf{char}\; s[2];\\
\textbf{char}\; t[4];\\
\mathtt{strcpy}(t, \texttt{"bon"});\\
\mathtt{strncpy}(s, t, 2);\\
\\
\mathtt{strcpy}(q, s);\\
\mathtt{printf}(\texttt{"nres:\ \ \%s/n"}, q);
\end{array}\right\}
\longmapsto
\left\{\begin{array}{l}
q_n = 2;\\
s_n = 2;\\
t_n = 2;\\
t_z = 3;\\
\textbf{if}(t_z < 2)\{s_z = t_z;\}\\
\textbf{else if}(s_z < t_n)\{s_z = s_n\}\\
\textbf{assert}(s_z < q_n); q_z = s_z;\\
\textbf{assert}(q_z < q_n);
\end{array}\right.$$

Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
4. Abstraction of strings and buffers
5. Basic pointer analyses
   - A micro-language with pointers
6. Three valued logic heap abstraction
7. Conclusion

Programs with pointers: syntax

Syntax extension: quite a few additional constructions

$\mathtt{l}$	$::=$	**l-valules**	
	$\mid$	$\mathtt{x}$	$(\mathtt{x} \in \mathbb{X})$
	$\mid$	$\ldots$	
	$\mid$	$*\mathtt{e}$	pointer dereference
	$\mid$	$\mathtt{l} \cdot \mathtt{f}$	field read
$\mathtt{e}$	$::=$	**expressions**	
	$\mid$	$\mathtt{l}$	
	$\mid$	$\ldots$	
	$\mid$	$\&\mathtt{l}$	"address of" operator
$\mathtt{s}$	$::=$	**statements**	
	$\mid$	$\ldots$	
	$\mid$	$\mathtt{x} = \mathbf{malloc}(c)$	allocation of c bytes
	$\mid$	$\mathbf{free}(\mathtt{x})$	deallocation of the block pointed to by $\mathtt{x}$

We do not consider pointer arithmetics here

Programs with pointers: semantics

Case of l-values:

$$\begin{array}{rcl}
[\![\mathtt{x}]\!](e, \hbar) & = & e(\mathtt{x}) \\
[\![*\mathsf{e}]\!](e, \hbar) & = & \begin{cases} \hbar([\![\mathsf{e}]\!](e, \hbar)) & \text{if } [\![\mathsf{e}]\!](e, \hbar) \neq 0 \wedge [\![\mathsf{e}]\!](e, \hbar) \in \mathbf{Dom}(\hbar) \\ \Omega & \text{otherwise} \end{cases} \\
[\![\mathtt{l} \cdot \mathtt{f}]\!](e, heap) & = & [\![\mathtt{l}]\!](e, \hbar) + \mathbf{offset}(\mathtt{f}) \text{ (numeric offset)}
\end{array}$$

Case of expressions:

$$\begin{array}{rcl}
[\![\mathtt{l}]\!](e, heap) & = & \hbar([\![\mathtt{l}]\!](e, heap)) \\
[\![\&\mathtt{l}]\!](e, heap) & = & [\![\mathtt{l}]\!](e, heap)
\end{array}$$

Case of statements:

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

Pointer non relational abstraction: null pointers

The dereferece of a null pointer will cause programs to crash

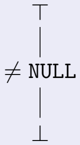
We go back to the **non relational abstraction of heterogeneous states**

- $\mathbb{V} = \mathbb{V}_{\rm addr} \uplus \mathbb{V}_{\rm int}$, $\mathbb{X} = \mathbb{X}_{\rm addr} \uplus \mathbb{X}_{\rm int}$
- we apply a non relational abstraction to pointer variables, based on $\mathbb{D}^\sharp_{\rm addr}$ and $\gamma_{\rm addr} : \mathbb{D}^\sharp_{\rm addr} \rightarrow \mathcal{P}(\mathbb{V}_{\rm addr})$

Null pointer analysis

Abstract lattice for addresses:

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



- 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 programs to crash

This requires a **similar abstraction:**

Null pointer analysis

Abstract lattice for addresses:

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


$\top$

Not dangling

$\bot$

- very lightweight, can typically resolve rather trivial cases
- useful for C
- in Java, superseded by the requirement that any variable be initialized

Pointer non relational abstraction: pointer aliasing

Determine where a pointer may store a reference to

Very useful to **support client analyses**:

```
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 `x` ?
  0, since **it is not modified at line 5**...

Basic pointer abstraction

- We assume a set of **abstract memory locations** $\mathbb{A}^\sharp$ is fixed:
  $$\mathbb{A}^\sharp = \{\&\mathtt{x}, \&\mathtt{y}, \ldots, \&\mathtt{t}, a_0, a_1, \ldots, a_N\}$$
- All concrete addresses are abstracted into $\mathbb{A}^\sharp$
- A pointer value is abstracted by the abstraction of the addresses it may point to (example, for `p`: $\{\&\mathtt{x}\}$)

Pointer aliasing based on equivalence on access paths

Aliasing relation

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

Abstraction to infer pointer aliasing properties

- An **access path** describes a sequence of operations to compute an l-value (i.e., an address); e.g.:

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

- 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 $\mathtt{x}(\texttt{->}\mathtt{n})^{\mathtt{k}}$ (ex: $\{\{\mathtt{x}(\texttt{->}\mathtt{n})^{\mathtt{k}}, \&(\mathtt{x}(\texttt{->}\mathtt{n})^{\mathtt{k}+1}) \mid k \in \mathbb{N}\}$)

Weak update problems

```
x ∈ [−10, −5]; y ∈ [5, 10]
int ⋆ p;
if(?)
    p = &x;
else
    p = &y;
⋆p = 0;
```

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

Weak update problems

```
x ∈ [−10, −5]; y ∈ [5, 10]
int ⋆ p;
if(?)
    p = &x;
else
    p = &y;
⋆p = 0;
```

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

- After the **if** statement, p may contain any address in $\{\&x, \&y\}$
- Thus, the assignment must consider all cases, in a conservative way
- Thus, x may receive a new value (0) or keep its old value
- Conclusion: $x \in [-10, 0]$, $y \in [0, 10]$

Weak updates

Any imprecision in the analysis may lead to weak updates...

Limitation of basic pointer analyses

- **Weak updates**:

  imprecisions for pointer values quickly spread out

- Many programs with pointers address **unbounded memory**

  e.g., to create lists, trees and other dynamically allocated structures
  most pointer analyses do not deal with this well...

- Pointer analyses do not nicely capture **structural invariants**
  e.g., lists, trees, but also nested structures

Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction
    - Basic principles
    - Building an abstract domain
    - Weakening abstract elements
    - Computation of transfer functions

An abstract representation of memory states: shape graphs

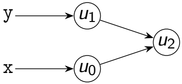
Goal of the static analysis

Discover complex invariants of programs that manipulate unbounded heap

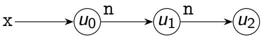
Observation: representation of memory states by shape graphs

- **Nodes** (aka, atoms) denote **memory locations**
- **Edges** denote **properties**, such as:
  - "field `f` of location u points to v"
  - "variable `x` is stored at location u"

Two alias pointers:



A list of length 2:



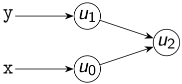
⇒ **We need to over-approximate sets of shape graphs**

Shape graphs and their representation

Description with predicates

- **Boolean encoding**: nodes are atoms $u_0, u_1, \ldots$
- **Predicates over atoms**:
  - $\mathtt{x}(u)$: variable `x` contains the address of u
  - $\mathtt{n}(u, v)$: field of u points to v
- **Truth values**: traditionally noted 0 and 1 in the TVLA litterature

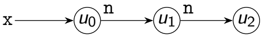
Two alias pointers:



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



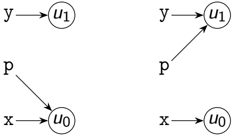
	$\mathtt{x}$
u_0	1
u_1	0
u_2	0

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

Unknown value: three valued logic

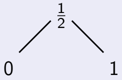
How to abstract away some information ?
i.e., to abstract several graphs into one ?

Example: pointer variable p alias with x or y

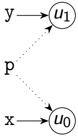


A boolean lattice

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



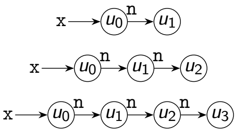
- Graph representation: dotted edges
- **Abstract graph**:



Summary nodes

We cannot talk about unbounded memory states with finitely many nodes

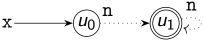
Lists of lengths 1, 2, 3:



We would like to **summarize** the lists

An idea

- Choose a node to represent **several** concrete nodes
- Similar to **smashing**



- Edges to u_1 are dotted

Definition: summary node

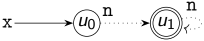
A **summary node** is an atom that may denote several concrete atoms

A few interesting predicates

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 $\frac{1}{2}$)

The properties of lists are not well-captured in



"Is shared"

$\underline{\mathrm{sh}}(u)$ ssi:

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

Predicates defined by transitive closure

- **Reachability**: $\underline{\mathrm{r}}(u, v)$ ssi

$$u = v \ \vee \ \exists u_0,\ \mathtt{n}(u, u_0) \wedge \underline{\mathrm{r}}(u_0, v)$$

- **Acyclicity**: $\underline{\mathrm{acy}}(v)$
  similar, with a negation

Outline

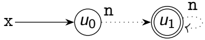
1. Towards memory properties
2. Memory models
3. Abstraction of arrays
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction
    - Basic principles
    - Building an abstract domain
    - Weakening abstract elements
    - Computation of transfer functions

Three structures

Definition: 3-structures

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

- a set $\mathcal{U} = \{u_0, u_1, \ldots, p_m\}$ of **atoms**
- a set $\mathcal{P} = \{p_0, p_1, \ldots, p_n\}$ of **predicates** (we write k_i for the arity of predicate p_i)
- 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 $\{0, \frac{1}{2}, 1\}$



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

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

n	u_0	u_1
u_0	0	1
u_1	0	0

Embedding

- How to compare two 3-structures ?
- How to describe the concretization of 3-structures ?

The embedding principle

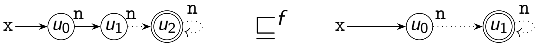
Let $\mathcal{S}_0 = (\mathcal{U}_0, \mathcal{P}, \phi_0)$ and $\mathcal{S}_1 = (\mathcal{U}_1, \mathcal{P}, \phi_1)$ be two three structures, with the same sets of predicates.
Let $f : \mathcal{U}_0 \to \mathcal{U}_1$, surjective.
We say that f **embeds** $\mathcal{S}_0$ **into** $\mathcal{S}_1$ iff

$$\begin{array}{c}\text{for all predicate } p \in \mathcal{P} \text{ or arity } k, \\ \text{for all } u_{l_1}, \ldots, u_{l_{k_i}} \in \mathcal{U}_0, \\ \phi_0(u_{l_1}, \ldots, u_{l_{k_i}}) \sqsubseteq \phi_0(f(u_{l_1}), \ldots, f(u_{l_{k_i}}))\end{array}$$

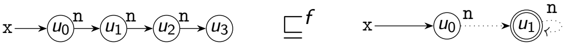
Then, **we write** $\mathcal{S}_0 \sqsubseteq^f \mathcal{S}_1$

Note: we use the order $\sqsubseteq$ of the lattice $\{0, \frac{1}{2}, 1\}$

Embedding examples



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

Note on the last example

- Reachability would be necessary to constrain it be a list
- Alternatively: cells should not be shared

Two structures and concretization

Concrete states correspond to 2-structures

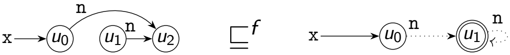
- **2-structure**: a 3-structure $(\mathcal{U}, \mathcal{P}, \phi)$ is a 2-structure, if and only if ϕ always returns in $\{0, 1\}$
- A **2-structure** corresponds to a set of concrete memory states (environment, heap):
  - we simply need to take into account all mappings of addresses into the memory
    we let **stores**$(\mathcal{S})$ denote the stores corresponding to 2-structure $\mathcal{S}$
  - more on this in the next lecture; here we keep it informal

Concretization

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

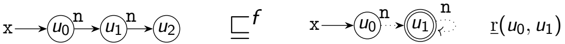
Concretization examples

- **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**:



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

Principle for the design of sound transfer functions

How to carry out static analysis using 3-structures ?

Embedding theorem

- Let $\mathcal{S}_0 = (\mathcal{U}_0, \mathcal{P}, \phi_0)$ and $\mathcal{S}_1 = (\mathcal{U}_1, \mathcal{P}, \phi_1)$ be two three structures, with the same sets of predicates
- Let $f : \mathcal{U}_0 \rightarrow \mathcal{U}_1$, such that $\mathcal{S}_0 \sqsubseteq^f \mathcal{S}_1$
- Let Ψ be a logical formula, with variables in X and $g : X \rightarrow \mathcal{U}_0$ be an assignment for the variables of Ψ

Then, $\llbracket \Psi_{|g} \rrbracket(\mathcal{S}_0) \sqsubseteq \llbracket \Psi_{|f \circ g} \rrbracket(\mathcal{S}_1)$

Principle for the design of sound transfer functions

Transfer functions for static analysis

- Semantics of concrete statements encoded into boolean formulas
- **Example:** assignment $\mathrm{y} := \mathrm{x}$
  - let y' denote the *new* value of y
  - add the constraint $\mathrm{y}'(u) = \mathrm{x}(u)$
  - rename y' into y

  Full examples of transfer functions computation in a few slides...
- Evaluation in the abstract is sound (embedding theorem)

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**

Outline

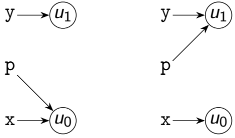
1. Towards memory properties
2. Memory models
3. Abstraction of arrays
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction
   - Basic principles
   - Building an abstract domain
   - Weakening abstract elements
   - Computation of transfer functions

A powerset abstraction

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

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

After the if statement:



abstracting here would be imprecise

Powerset abstraction

- Shape analyzers usually rely on a **powerset abstract domain**
  i.e., TVLA manipulates **finite disjunctions** of 3-structures
- How to ensure disjunctions will not grow infinite ?

Canonical abstraction

Canonicalization principle

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

- **can** called a **canonicalization** if it is an upper closure operator
- then, **can** extends into a canonicalization operator of $\mathcal{P}(\mathcal{L})$, into $\mathcal{P}(\mathcal{L}')$:
$$\mathbf{can}(\mathcal{E}) = \{\mathbf{can}(x) \mid x \in \mathcal{E}\}$$

To make the powerset domain work, we simply need a **can** over 3-structures

A canonicalization over 3-structures

- We assume there are n variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$
  Thus the number of unary predicates is finite
- **Sub-lattice**: structures with atoms **distinguished by the values of the unary predicates** (or *abstraction predicates*) $\mathtt{x}_1, \ldots, \mathtt{x}_n$

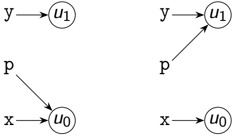
We may choose another set of predicates for the sub-lattice representation

Canonical abstraction

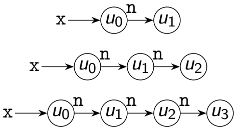
Canonical abstraction by truth blurring

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

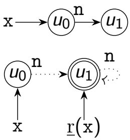
Elements not merged:



Elements merged:
Lists of lengths 1, 2, 3:



Abstract into:

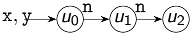


Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction
   - Basic principles
   - Building an abstract domain
   - Weakening abstract elements
   - Computation of transfer functions

Assignment: a simple case

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

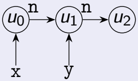


Transfer function

- Should yield an over-approximation of $\{m_1 \in \mathbb{M} \mid (l_0, m_0) \rightarrow (l_1, m_1)\}$
- We let **"primed predicates"** denote predicates after evaluation of the assignment, to evaluate them in the same structure
- Then:

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

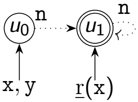
- Result:



This was exactly what we expected

Assignment: a more involved case

Statement $\ell_0 : \mathtt{y} = \mathtt{y} \mathtt{->} \mathtt{n}; \ell_1 : \ldots$



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

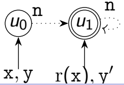
Focusing on a formula

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 $\hat{\mathcal{S}}$ such that:

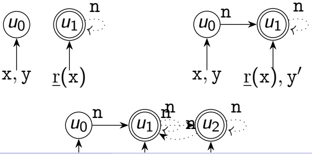
- **f evaluates to 0 or 1** on all elements of $\hat{\mathcal{S}}$
- **precision was gained**: $\forall \mathcal{S}' \in \hat{\mathcal{S}},\ \mathcal{S}' \sqsubseteq \mathcal{S}$
- **soundness is preserved**: $\gamma(\mathcal{S}) = \bigcup\{\gamma(\mathcal{S}') \mid \mathcal{S}' \in \hat{\mathcal{S}}\}$

- Focusing algorithms are complex and tricky (see biblio)
- Involves splitting of summary nodes, solving of boolean constraints

Example: focusing on
$\mathtt{y}'(u) = \exists v,\ \mathtt{y}(v) \wedge \mathtt{n}(v, u)$

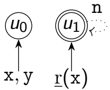


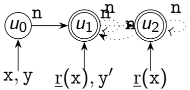
We obtain (we show y and y′):



Focus and coerce

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





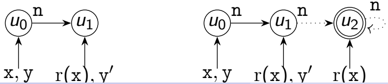
u_1 is reachable from x, but there is no sequence of n fields: this structure has **empty concretization**

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:



Focus, transfer, abstract...

Computation of a transfer function

We consider a transfer function encoded into boolean formula f

$$\begin{tikzcd}
\mathcal{S}^{\sharp}_{\mathrm{pre}} \arrow[d, swap, "\text{focus coerce}"] & \mathcal{S}^{\sharp}_{\mathrm{post}} \\
\hat{\mathcal{S}}_{\mathrm{pre}} \arrow[r, "f"] & \hat{\mathcal{S}}_{\mathrm{post}} \arrow[u, swap, "\textbf{can}"]
\end{tikzcd}$$

Soundness proof steps:

1. sound encoding of the semantics of program statements into formulas
   typically, no loss of precision at this stage
2. focusing should yield an over-approximation of its input
3. canonicalization over-approximates graph (truth blurring weakening)

A common picture in shape analysis

Outline

1. Towards memory properties
2. Memory models
3. Abstraction of arrays
4. Abstraction of strings and buffers
5. Basic pointer analyses
6. Three valued logic heap abstraction
7. Conclusion

Conclusion

Concrete semantics:

- Splitting environment and heap
- Taking into account of the representaton of data

Many families of domain specific abstractions:

- Based on numerical methods
  path based pointer analyses, array segment analyses, string analyses
- Symbolic abstractions based on pointer sets, structural predicates
- Locally concretize / globally abstract pattern (TVLA, arrays...)
  More on this during the next lecture...

Bibliography

- **[HP'08]**: **Discovering properties about arrays in simple programs.**
  Nicolas Halbwachs, Mathias Péron. In PLDI'08, pages 339-348, 2008.
- **[CCL'11]**: **A parametric segmentation functor for fully automatic and scalable array content analysis.**
  Patrick Cousot, Radhia Cousot, Francesco Logozzo. In POPL'11, pages 105-118, 2011.
- **[AD'94]**: **Interprocedural may alias analysis for pointers: beyond k-limiting.**
  Alain Deutsch. In PLDI'94, pages 230–241, 1994.
- **[CSSV'03]**: **CSSV: towards a realistic tool for statically detecting all buffer overflows in C.**
  Nurit Dor, Michael Rodeh, Shmuel Sagiv. In PLDI'03, pages 155-167.
- **[SRW'99]**: **Parametric Shape Analysis via 3-Valued Logic.**
  Shmuel Sagiv, Thomas W. Reps et Reinhard Wilhelm. In POPL'99, pages 105–118, 1999.