Memory abstraction 1

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

Xavier Rival

INRIA, ENS, CNRS

Jan, 6th. 2016

Outline

1. Memory models
   - Towards memory properties
   - Formalizing concrete memory states
   - Treatment of errors
   - Language semantics
2. Abstraction of arrays
3. Basic pointer analyses
4. Three valued logic heap abstraction
5. Conclusion

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 ?**

$\Rightarrow$ **a very broad topic**, and two lectures:

This lecture

- **overview memory models** and **memory properties**
- abstraction of **arrays**
- abstraction of **pointer structures** / **shape analysis**

Next lecture: abstractions based on **separation logic**

Assumptions

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:**

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

Abstraction of sets of states described by domain $\mathbb{D}^\sharp$ and concretization $\gamma : (\mathbb{D}^\sharp, \sqsubseteq^\sharp) \longrightarrow (\mathcal{P}(\mathbb{S}), \subseteq)$

Programs: syntax

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

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

Programs: semantics

We assume **classical definitions for:**

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

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

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

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 $\mathbb{D}^\sharp_{\rm mem}, \gamma_{\rm mem}$...

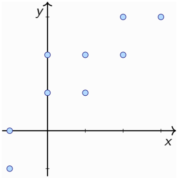
- previous lectures: $\mathbb{X}$ **is fixed and finite** and, $\mathbb{V}$ **is integers of floats**, thus, $\mathbb{M} \equiv \mathbb{V}^n$
- today, we will extend the language and the abstractions

Abstraction of purely numeric memory states

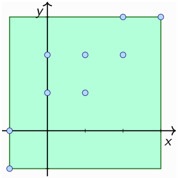
Purely numeric case

- $\mathbb{V}$ 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 $\mathbb{V}^N$**

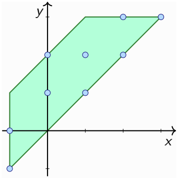
Example: $N = 2$, $\mathbb{X} = \{x, y\}$



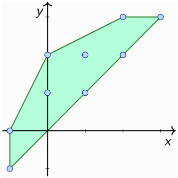
concrete set



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

- $\mathbb{M} \equiv \mathbb{M}_0 \times \mathbb{M}_1 \times \ldots$ where $\mathbb{M}_i = \mathbb{X}_i \to \mathbb{V}_i$
- **Concretization function** (case with two types)

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

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

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

- intervals
- polyhedra...

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

- lattice of boolean constants
- relational abstraction with BDDs

Memory structures

- To describe memories, 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** / 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...)

- **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., balancing of `Maps` was wrong in the OCaml standard library...

A more realistic model

Not all memory cell corresponds to a variable

- a variable may correspond to **several cells** (structures...)
- **dynamically 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

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

  h is actually only a partial function

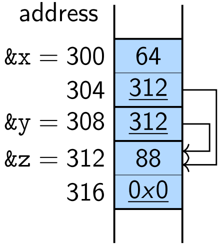
- **Memory states** (or **memories**): $\mathbb{M} = \mathbb{E} \times \mathbb{H}$

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)

- 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)



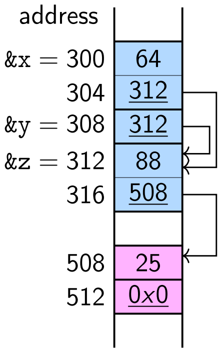
$$e: \begin{array}{lll} x & \mapsto & 300 \\ y & \mapsto & 308 \\ z & \mapsto & 312 \end{array}$$

$$h: \begin{array}{lll} 300 & \mapsto & 64 \\ 304 & \mapsto & 312 \\ 308 & \mapsto & 312 \\ 312 & \mapsto & 88 \\ 316 & \mapsto & 0 \end{array}$$

Example of a concrete memory state (variables + dyn. cell)

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

Memory layout



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

Extending the semantic domains

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:** $[\![\mathtt{l}]\!] : \mathbb{M} \to \mathbb{V}_{\mathrm{addr}}$

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

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

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

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

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

where

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

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 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:

- exception in case of out-of-bound array access, null dereference:

  `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) such that $\forall m_1 \in \mathbb{M},\ (l_0, m_0) \to (l_1, m_1)$

- **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 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

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

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 $*$) 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 **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

Choices to fix a memory model

- **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

In this course, we start with a simple model, and add specific features one by one (arrays, pointers) in order to study corresponding abstractions

Outline

1 Memory models

2 Abstraction of arrays
- A micro language for manipulating arrays
- Verifying safety of array operations
- Abstraction of array contents
- Abstraction of array properties

3 Basic pointer analyses

4 Three valued logic heap abstraction

5 Conclusion

Programs: extension with arrays

Extension of the syntax:

l	::=	**l-valules**	
	|	...	previous constructions
	|	x[e]	cell of array x
...	::=	...	the rest is unchanged

Extension of the states:

- 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 a base type value** (8 bytes for a 64-bit int)

Extension of the semantics, case of an **array cell read**:

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

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, 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 Memory models

2 Abstraction of arrays
- A micro language for manipulating arrays
- Verifying safety of array operations
- Abstraction of array contents
- Abstraction of array properties

3 Basic pointer analyses

4 Three valued logic heap abstraction

5 Conclusion

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

Outline

1. Memory models

2. Abstraction of arrays
   - A micro language for manipulating arrays
   - Verifying safety of array operations
   - Abstraction of array contents
   - Abstraction of array properties

3. Basic pointer analyses

4. Three valued logic heap abstraction

5. Conclusion

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** $\mathtt{x[y[i+j]+k]}$ or $\mathtt{y = n / x[i]}$

Assumption:

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

Elementwise abstraction

- **Each** concrete cell is **mapped into one abstract cell**
- $\mathbb{D}^\sharp$ should simply be an **abstraction of** $\mathcal{P}(\mathbb{V}^{m+n})$ (relational or not)

Abstract and concrete memory cell addresses:

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

Elementwise abstraction example

We consider the following **set of concrete states**:

i	t		
i : 1	0	1	0
i : 4	2	5	1
i : 8	5	8	3
i : 7	3	6	2

The **elementwise abstraction** produces the following vectors:

$$(1,0,1,0) \qquad (4,2,5,1)$$
$$(8,5,8,3) \qquad (7,3,6,2)$$

After applying the **interval abstraction**, we get:

$$([1,8],[0,5],[1,8],[0,3])$$

This is **precise** but **costly** if arrays are big

Post-condition for an assignment: example 1

Assignment $\mathtt{t}[0] = 6$	**Pre-condition:** t : $[0,1]$ | $[1,2]$

- **concrete pre-condition:**

t : 0 | 1   t : 0 | 2   t : 1 | 1   t : 1 | 2

- **effect** of the assignment **in the concrete** and post-condition:

t : 6 | 1   t : 6 | 2   t : 6 | 1   t : 6 | 2

Thus, we obtain the **abstract post-condition**:

t : $[6,6]$ | $[1,2]$

This analysis step is **precise**, but what if the index is not known so precisely ?

Post-condition for an assignment: example 2

Assignment $\mathtt{t[i]} = 6$	**Pre-condition:** $\mathtt{i} \in [0, 1] \wedge$ $\mathtt{t}$: | $[0, 0]$ | $[8, 8]$ |

- **concrete pre-condition:** $\mathtt{t}$: | 0 | 8 |
- **effect** of the assignment **in the concrete** and post-condition:

$\mathtt{t}$: | 0 | 8 | —— if $\mathtt{i} = 0$ ——> $\mathtt{t}$: | 6 | 8 |

$\mathtt{t}$: | 0 | 8 | —— if $\mathtt{i} = 1$ ——> $\mathtt{t}$: | 0 | 6 |

Thus, we obtain the **abstract post-condition**:

$\mathtt{t}$: | $[0, 6]$ | $[6, 8]$ |

This analysis step looks quite coarse, but it is actually fine here: each cell may get the new value or keep the old one...

Two kinds of abstract updates

Strong updates

- **One modified concrete cell abstracted by one, precisely known abstract cell**
- The effect of the update **is computed precisely** by the analysis

Strong updates are the **most favorable case**, as new information is computed precisely, and known information is not lost (example 1)

Weak updates

- **The modified concrete cell cannot be mapped into a well identified abstract cell**
- The resulting abstract information is obtained by **joining the new value and the old information**

In the example, the weak update loses no information...

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, one array $\bar{\mathtt{t}}$ of length n

Array smashing

- All cells of the array are mapped into **one abstract cell** $\bar{\mathtt{t}}$
- **Concrete cells:**
  $\mathbb{V}_{\mathrm{addr}} = \{\&\mathtt{x}_0, \ldots, \&\mathtt{x}_{m-1}\} \cup \{\&\bar{\mathtt{t}}, \&\bar{\mathtt{t}} + 1 \cdot s, \ldots, \&\bar{\mathtt{t}} + (n-1) \cdot s\}$
- **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 $\mathtt{n} = \ldots$; **int** $\mathtt{t}[\mathtt{n}]$;

Array smashing abstraction

Definition

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

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

Example, with no variable and an array of length 2:

- **Set of concrete states** H:

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

- **Smashing abstraction** produces $\{\{0, 10\}, \{2, 11\}, \{1, 12\}\}$
- After **non relational abstraction**, we obtain $\&\bar{\mathtt{t}} \mapsto \{0, 1, 2, 10, 11, 12\}$

Array smashing abstraction example

We consider the following **set of concrete states**:

i : 1 t : | 0 | 1 | 0 |

i : 4 t : | 2 | 5 | 1 |

i : 8 t : | 5 | 8 | 3 |

i : 7 t : | 3 | 6 | 2 |

The **smashing abstraction** produces the following vectors:

$$(\{1\},\{0,1,0\}) \qquad (\{4\},\{2,5,1\})$$
$$(\{8\},\{5,8,3\}) \qquad (\{7\},\{3,6,2\})$$

After **non relational abstraction**:

$$\&\mathtt{i} \longmapsto \{1,4,8,7\}$$
$$\&\bar{\mathtt{t}} \longmapsto \{0,1,2,3,5,6,8\}$$

After applying the **interval abstraction**, we get: $([1,8],[0,8])$

Post-condition for an assignment: example

Assignment $\mathtt{t}[0] = 6$	**Pre-condition:** $\mathtt{t} : \forall i,\ \mathtt{t}[i] : [0, 0]$

- **concrete pre-condition:** $\mathtt{t} : \boxed{0 \mid 0}$
- **effect** of the assignment **in the concrete** and post-condition:

$$\mathtt{t} : \boxed{0 \mid 0} \longrightarrow \mathtt{t} : \boxed{6 \mid 0}$$

Thus, we obtain the **abstract post-condition**:

$$\mathtt{t} : \forall i,\ \mathtt{t}[i] : [0, 6]$$

Consequence:
the analysis of $\mathtt{t}[0] = 6; \mathtt{t}[1] = 6;$
will also produce

$$\mathtt{t} : \forall i,\ \mathtt{t}[i] : [0, 6]$$

This is a another case of weak-update, resulting in significant precision loss

Weak-updates

Weak updates

- **The modified concrete cell cannot be mapped into a well identified abstract cell**
- The resulting abstract information is obtained by **joining the new value and the old information**

To summarize:

abstraction	$\mathtt{t}[0] = \ldots$	$\mathtt{t}[[a, b]] = \ldots$
element-wise	strong update	weak update
smashing	weak update	weak update

- relatively to the abstraction, a weak update may be precise (as in the examples)
- however, successions of weak updates will prevent from inferring invariants such as correctness of initialization

Weak updates and strong updates: example

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

Elementwise 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. Memory models

2. Abstraction of arrays
   - A micro language for manipulating arrays
   - Verifying safety of array operations
   - Abstraction of array contents
   - Abstraction of array properties

3. Basic pointer analyses

4. Three valued logic heap abstraction

5. Conclusion

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

Array initialization loop

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

Hand proof sketch:

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

To complete the proof, we need to express properties on segments

Array segment properties

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:

i $= 6$

t									
0	0	0	0	0	0	?	?	?	?

Corresponding abstract state:

i $\in [1, 10]$

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

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 $\mathbf{sort}_{\mathtt{t}}(1, 3)$ **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

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

- **counter example** for the fourth 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: $\mathbf{sort}_{\mathtt{t}}(i, j, \min, \max)$

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

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

For instance, for an array of length *n*:

$$\mathbf{zero}_{\mathtt{t}}(0, \mathtt{n}-1) \wedge 0 \leq \mathtt{i} < \mathtt{n} \overset{\mathtt{t[i]=?}}{\longrightarrow} \mathbf{zero}_{\mathtt{t}}(0, \mathtt{i}-1) \wedge \mathbf{zero}_{\mathtt{t}}(\mathtt{i}+1, \mathtt{n}-1)$$

$$\top \wedge 0 \leq \mathtt{i} < \mathtt{n} \overset{\mathtt{t[i]=0}}{\longrightarrow} \mathbf{zero}_{\mathtt{t}}(\mathtt{i}, \mathtt{i})$$

Abstract join operator: generalizes bounds

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

Array analysis: example

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

t $\top$ i $\top$

int $\texttt{i} = 0$;

t $\top$ i $\top$

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

t $\top$ i $\top$

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

t $\top$ i $\top$

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

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 $\top$

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 $[\ \mathbf{zero}_{\bar{t}}(0, 0) \mid \top\ ]$ 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 $\texttt{i} = 0$;

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

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

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

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

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

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

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

$\}$

t $\top$ i $\top$

Array analysis: example

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

t: [$\top$]   i: $\top$

int i = 0;

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

while(i < n){

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

    t[i] = 0;

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

    i = i + 1;

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

}

t: [$\top$]   i: $\top$

Array analysis: example

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

t [$\top$] i $\top$

int i $= 0$;

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

while$(i < n)\{$

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

t[i] $= 0$;

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

i $=$ i $+ 1$;

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

$\}$

t [$\top$] i $\top$

Array analysis: example

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

t	$\top$	i	$\top$

int i $= 0$;

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

while$(i < n)\{$

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

t[i] $= 0$;

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

i $=$ i $+ 1$;

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

$\}$

t	$\top$	i	$\top$

Array analysis: example

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

$\mathtt{t}$ [$\top$] $\quad \mathtt{i}\ \top$

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

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

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

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

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

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

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

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

$\}$

$\mathtt{t}$ [$\top$] $\quad \mathtt{i}\ \top$

Array analysis: example

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

t	$\top$	i	$\top$

int $\mathtt{i} = 0;$

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

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

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

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

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

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

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

$\}$

t	$\top$	i	$\top$

Array analysis: example

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

t: $\top$ | i: $\top$

int i = 0;

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

while(i < n){

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

    t[i] = 0;

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

    i = i + 1;

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

}

t: $\top$ | i: $\top$

Array analysis: example

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

$\mathtt{t}$ [$\top$] $\quad \mathtt{i} \quad \top$

int $\mathtt{i} = 0$;

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

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

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

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

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

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

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

$\}$

$\mathtt{t}$ [$\top$] $\quad \mathtt{i} \quad \top$

Array analysis: example

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

t	$\top$		i	$\top$

int $\mathtt{i} = 0$;

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

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

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

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

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

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

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

$\}$

t	$\top$		i	$\top$

Array analysis: example

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

$\mathtt{t}$: $[\ \top\ ]$ $\quad$ $\mathtt{i}$: $\top$

int i = 0;

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

while(i < n){

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

t[i] = 0;

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

i = i + 1;

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

}

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

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 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

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

- **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 Memory models

2 Abstraction of arrays

3 Basic pointer analyses

4 Three valued logic heap abstraction

5 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:

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

Case of expressions:

$$\begin{array}{rcll} [\![\mathtt{l}]\!](e, h) & = & h([\![\mathtt{l}]\!](e, h)) & \text{(evaluates into the contents)} \\ [\![\&\mathtt{l}]\!](e, h) & = & [\![\mathtt{l}]\!](e, h) & \text{(evaluates into the address)} \end{array}$$

Case of statements:

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

Pointer non relational abstractions

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

- $\mathbb{V} = \mathbb{V}_{\mathrm{addr}} \uplus \mathbb{V}_{\mathrm{int}}$
- $\mathbb{X} = \mathbb{X}_{\mathrm{addr}} \uplus \mathbb{X}_{\mathrm{int}} \uplus \ldots$
- **concrete memory cells** now include **structure fields**, and fields of **dynamically allocated regions**
- **abstract cells** $\mathbb{C}^\sharp$ finitely summarize concrete cells
- we apply a **non relational abstraction** to pointer locations, based on $\mathbb{D}^\sharp_{\mathrm{ptr}}$ and $\gamma_{\mathrm{ptr}} : \mathbb{D}^\sharp_{\mathrm{ptr}} \to \mathcal{P}(\mathbb{V}_{\mathrm{addr}})$ (other location abstracted in the same way as before, *e.g.*, non relationally)

We will see **several instances** of this kind of abstraction

Pointer non relational abstraction: null pointers

The dereference of a null pointer will cause a crash

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

Null pointer analysis

Abstract domain for addresses:

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

$$\top \;-\; \neq \texttt{NULL} \;-\; \bot$$

- 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:

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

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

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 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:
  $$\mathbb{A}^\sharp = \{\&\mathtt{x}, \&\mathtt{y}, \ldots, \&\mathtt{t}, a_0, a_1, \ldots, a_N\}$$
- **Concrete addresses** are **abstracted into** $\mathbb{A}^\sharp$ by $\phi_{\mathbb{A}} : \mathbb{A} \rightarrow \mathbb{A}^\sharp \uplus \{\top\}$
- A pointer value is abstracted by the abstraction of the addresses it may point to, *i.e.*, $\mathbb{D}^\sharp_{\mathrm{ptr}} = \mathcal{P}(\mathbb{A}^\sharp)$
  and $\gamma_{\mathrm{ptr}}(a^\sharp) = \{a \in \mathbb{A} \mid \phi_{\mathbb{A}}(a) = a^\sharp\}$

Points-to sets computation example

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	$[0, 0]$	$[9, 9]$	$\{\&x\}$

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;
```

	&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	$[-10, 0]$	$[0, 10]$	{&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

As in array analysis, we encounter:

Weak updates

- **The modified concrete cell cannot be mapped into a well identified abstract cell**
- 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**
- if the points-to set may contain **more than one element**, the analysis has to perform a **weak-update**

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.*:

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

- 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

Weak updates:

- **imprecision in updates** that spread out as soon as points-to set contain several elements
- impact **client analyses** severely (as for array analyses)

Unsatisfactory abstraction of unbounded memory:

- common assumption that $\mathbb{C}^\sharp$ **be finite**
- 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

Shape abstraction:
We will use similar ideas as for array segment analyses

Outline

1 Memory models

2 Abstraction of arrays

3 Basic pointer analyses

4 Three valued logic heap abstraction
- Basic principles
- Building an abstract domain
- Weakening abstract elements
- Computation of transfer functions

5 Conclusion

An abstract representation of memory states: shape graphs

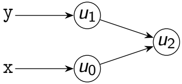
Goal of the static analysis

Infer structural invariants of programs using unbounded heap

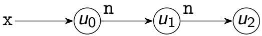
Observation: representation of memory states by shape graphs

- **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:



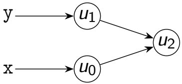
⇒ We need to over-approximate sets of shape graphs

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

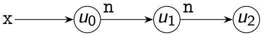
Two alias pointers:



	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:

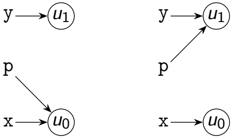


	x
u_0	1
u_1	0
u_2	0

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

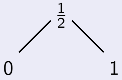
Unknown value: three valued logic

How to abstract away some information ?
i.e., to abstract several graphs into one ?
Example: pointer variable p alias with x or y

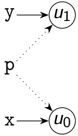


A boolean lattice

- 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

At this point, we cannot talk about **unbounded memory states** with **finitely many** nodes

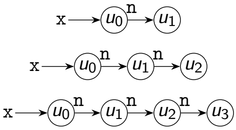
An idea

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

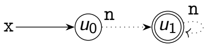
Definition: summary node

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

Lists of lengths 1, 2, 3:



Attempt at a **summary** graph:



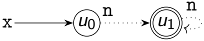
- Edges to u_1 are dotted

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



We need to **add more information**, *e.g.*, about **connectedness**

"Is shared"

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

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

Predicates defined by transitive closure

- **Reachability**: $\underline{\mathrm{r}}(u, v)$ if and only if
  $u = v \ \vee \ \exists u_0,\ \mathtt{n}(u, u_0) \wedge \underline{\mathrm{r}}(u_0, v)$
- **Acyclicity**: $\underline{\mathrm{acy}}(v)$
  similar, with a negation

Outline

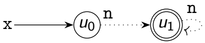
1. Memory models
2. Abstraction of arrays
3. Basic pointer analyses
4. Three valued logic heap abstraction
   - Basic principles
   - Building an abstract domain
   - Weakening abstract elements
   - Computation of transfer functions
5. Conclusion

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 $\{0, \frac{1}{2}, 1\}$



$$\begin{cases} \mathcal{U} = \{u_0, u_1\} \\ \mathcal{P} = \{\mathtt{x}(\cdot), \mathtt{n}(\cdot,\cdot), \underline{\mathrm{sum}}(\cdot)\} \end{cases}$$

	x	$\underline{\mathrm{sum}}$
u_0	1	0
u_1	0	$\frac{1}{2}$

n	u_0	u_1
u_0	0	1
u_1	0	0

In the following we build up an abstract domain of 3-structures

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

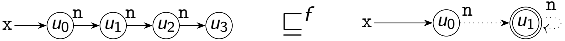
$$\text{for all predicate } p \in \mathcal{P} \text{ or arity } k, \quad \text{for all } u_{l_1}, \ldots, u_{l_{k_i}} \in \mathcal{U}_0, \\ \phi_0(u_{l_1}, \ldots, u_{l_{k_i}}) \sqsubseteq \phi_0(f(u_{l_1}), \ldots, f(u_{l_{k_i}}))$$

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

- **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

A 3-structure $(\mathcal{U}, \mathcal{P}, \phi)$ is a **2-structure**, if and only if ϕ always returns in $\{0, 1\}$

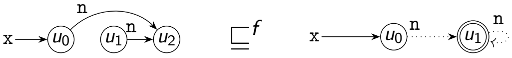
- A **2-structure** defines a set of **concrete memory states** (e, h) obtained by mapping symbols to addresses, that are **compatible with the predicates** of the structure
- We let **stores**$(\mathcal{S})$ denote the stores corresponding to 2-structure $\mathcal{S}$

Concretization of a 3-structure

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

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 ?

- **Concrete states** correspond to **2-structures**
- The **analysis** should track **3-structures**, thus the analysis correctness should **rely on the embedding relation**

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 $f : \mathcal{U}_0 \to \mathcal{U}_1$, such that $\mathcal{S}_0 \sqsubseteq^f \mathcal{S}_1$
- Let Ψ be a logical formula, with variables in X and $g : X \to \mathcal{U}_0$ be an assignment for the variables of Ψ

Then,

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

Principle for the design of sound transfer functions

Transfer functions for static analysis

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

Example: assignment $\mathrm{y} := \mathrm{x}$

1. let y' denote the *new* value of y
2. add the constraint $\mathrm{y}'(u) = \mathrm{x}(u)$
3. rename y' into 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**

Outline

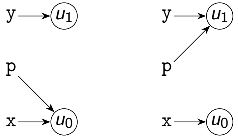
1. Memory models
2. Abstraction of arrays
3. Basic pointer analyses
4. Three valued logic heap abstraction
    - Basic principles
    - Building an abstract domain
    - Weakening abstract elements
    - Computation of transfer functions
5. Conclusion

A powerset abstraction

- 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



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** : $\mathcal{L} \rightarrow \mathcal{L}'$:

- operator **can** is called **canonicalization** if and only if it defines an **upper closure operator**
- then it defines a **canonicalization operator** **can** : $\mathcal{P}(\mathcal{L}) \rightarrow \mathcal{P}(\mathcal{L}')$:

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

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*) $\mathtt{x}_1, \ldots, \mathtt{x}_n$

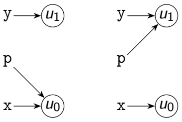
Canonical abstraction

We assume the analysis relies on unary predicates for canonicalization. The analysis design may choose another set of predicates than the unary predicates for the sub-lattice representation

Canonical abstraction by truth blurring

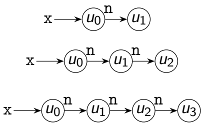
1. **Identify** nodes that **have different abstraction predicates**
2. When several nodes have the **same abstraction predicate introduce a summary node**
3. **Compute new predicate values** by doing a **join over truth values**

Elements not merged:

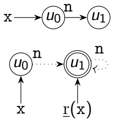


Elements merged:

Lists of lengths 1, 2, 3:



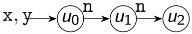
Abstract into:



Outline

1. Memory models
2. Abstraction of arrays
3. Basic pointer analyses
4. Three valued logic heap abstraction
   - Basic principles
   - Building an abstract domain
   - Weakening abstract elements
   - Computation of transfer functions
5. Conclusion

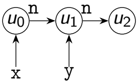
Assignment: a simple case

Statement $\ell_0 : \mathtt{y} = \mathtt{y} \text{ -> } \mathtt{n}; \ell_1 : \ldots$	**Pre-condition** $\mathcal{S}$ 

Transfer function computation:

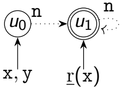
- It should produce an over-approximation of $\{m_1 \in \mathbb{M} \mid (\ell_0, m_0) \rightarrow (\ell_1, m_1)\}$
- **Encoding** using **"primed predicates"** to denote predicates **after** the evaluation of the assignment, to evaluate them in the same structure (non primed variables are removed afterwards and primed variables renamed):

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

- **Result:** 

This is exactly the expected result

Assignment: a more involved case

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

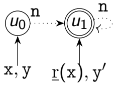
We assume a 3-structure $\mathcal{S}$ and a boolean formula f are given, we call a **focusing $\mathcal{S}$ on f** the generation of a set $\hat{\mathcal{S}}$ such that:

- **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**: $\gamma(\mathcal{S}) = \bigcup\{\gamma(\mathcal{S}') \mid \mathcal{S}' \in \hat{\mathcal{S}}\}$

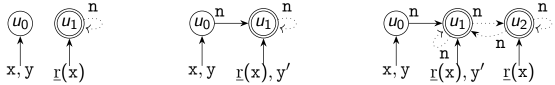
- Focusing algorithms are complex and tricky
- Involves splitting of summary nodes, solving of boolean constraints

Example: focusing on

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

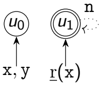


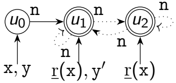
We obtain (we show y and y'):



Focus and coerce

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





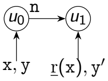
u_1 is reachable from x, but there is no sequence of n fields: this structure has **empty concretization**

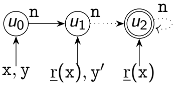
u_0 has an n-field to u_1 so u_1 denotes a unique atom and **cannot be a summary node**

Coerce operation

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

Result:

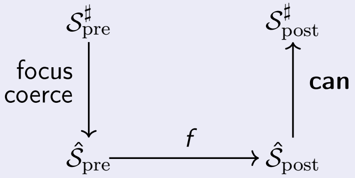




Focus, transfer, abstract...

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

Outline

1. Memory models
2. Abstraction of arrays
3. Basic pointer analyses
4. Three valued logic heap abstraction
5. Conclusion

Summarization: one abstract cell, many concrete cells

Large / unbounded numbers of concrete cells need to be abstracted

- **Array blocks** may have large number of elements
- **Dynamic memory allocation** functions may be called an unbounded number of times

Summary abstract cell

A **summary abstract cell** describes **several concrete cells**.
A **summary abstract variable** describes **several concrete values**.

- **Formalization** based on a function mapping **concrete cells** into the **abstract cells** that represent them:

$$\phi_{\mathbb{A}} : \mathbb{A} \to \mathbb{A}^{\sharp}$$

- Analysis operations should reason on abstract states **up-to** $\phi_{\mathbb{A}}$

Updates: weak vs strong

Memory updates may be very imprecise

Several typical cases:

1. update to a cell **that cannot be determined precisely**
   i.e., affecting an abstract cell among $A^\sharp \subseteq \mathbb{A}^\sharp$, where $|A^\sharp| > 1$
2. update to a **summary cell**

In those cases, the abstract update **joins previous values and new values**

Weak updates

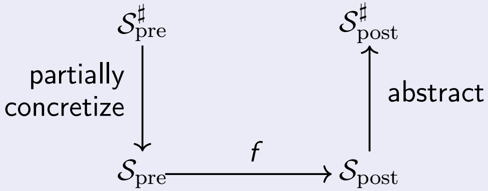
- **The modified concrete cell cannot be mapped into a well identified abstract cell**
- The resulting abstract information is obtained by **joining the new value and the old information**

Concretize partially, update, abstract

Summaries can be refined locally for better precision

- **Array segment predicates** can be split into predicates over smaller segments for abstract transfer functions
- The information over **TVLA summary nodes** can be refined using disjunctions for the computation of abstract post-conditions

A scheme to compute more precise post-conditions



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.
- **[SRW'99]**: **Parametric Shape Analysis via 3-Valued Logic.**
  Shmuel Sagiv, Thomas W. Reps et Reinhard Wilhelm. In POPL'99, pages 105–118, 1999.