Abstraction of memory states

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

Xavier Rival

INRIA

November, 16th. 2012

Overview of the lecture

How to reason about memory properties

$\Rightarrow$ **a very broad topic...**

- This lecture:
  - **overview most common problems**
  - discuss **arrays**, **strings**
  - introduction to **shape analysis**
- Next lecture: deeper study of a **family of shape analyses**

Programs

Previous lectures

- Programs can be viewed as labelled transition systems
- Transition relation:

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

where $\mathbb{S}$ is the set of states

- To design a static analysis, we need to abstract sets of states

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

A state = a label + a memory state

- **Label** ($l \in \mathbb{L}$): *control state*
  value of the program counter, current instruction...
- **Memory** ($m \in \mathbb{M}$): *memory state*
  description of the computer's memory contents

Abstraction of program semantics

How to abstract $\mathcal{P}(\mathbb{S}^\star)$?

- For each control state, we collect a set of states
- We need to abstract the corresponding set of memory states

Steps of the context sensitive abstraction

- **Partitioning** guided by the control state

$$\begin{array}{rcll} \gamma_{\rm ctxt} : & (\mathbb{L} \rightarrow \mathcal{P}(\mathbb{M})) & \longrightarrow & \mathcal{P}(\mathbb{S}^\star) \\ & \Phi & \longmapsto & \{\langle (l_0, m_0), \ldots, (l_n, m_n) \rangle \mid \forall i,\ m_i \in \Phi(l_i)\} \end{array}$$

- **Pointwise composition** with an abstraction for sets of memory states

$$\gamma_{\rm mem} : \mathbb{D}^\sharp \longrightarrow \mathcal{P}(\mathbb{M})$$

- **Resulting abstraction**

$$\begin{array}{rcll} \gamma : & (\mathbb{L} \rightarrow \mathbb{D}^\sharp) & \longrightarrow & \mathcal{P}(\mathbb{S}^\star) \\ & \Phi^\sharp & \longmapsto & \{\langle (l_0, m_0), \ldots, (l_n, m_n) \rangle \mid \forall i,\ m_i \in \gamma_{\rm mem}(\Phi^\sharp(l_i))\} \end{array}$$

Abstraction of homogeneous memory states

How to describe memory states $m \in \mathbb{M}$?

A simple model

- **Finite set of variables** $\mathbb{X}$: e.g., $\mathbb{X} = \{x, y, z \ldots\}$
- **Set of values** $\mathbb{V}$
- **Memory states**: functions from variables to values

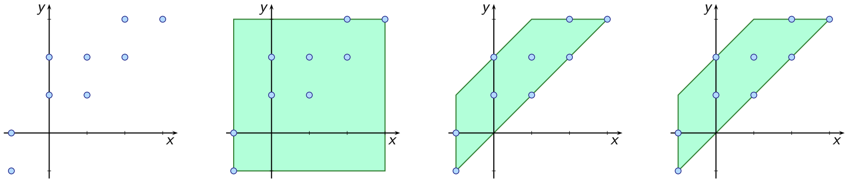
$$\mathbb{M} = \mathbb{X} \to \mathbb{V}$$

Homegenous memory states and abstraction

Homogenous 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 — octagon domain

Heterogeneous memory states

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

Heterogeneous memory states

- **Values are not all of the same kind**
  $\mathbb{V} = \mathbb{V}_{t_0} \uplus \mathbb{V}_{t_1} \uplus \ldots$
- Finite set of variables; each variable has a fixed type
  $$\mathbb{X} = \mathbb{X}_{t_0} \uplus \mathbb{X}_{t_1} \uplus \ldots$$
- $\mathbb{M} = \mathbb{X} \to \mathbb{V}$

Example:

- Values are either (machine) integers ($\mathbb{V}_{\rm int}$), floating point ($\mathbb{V}_{\rm float}$) or booleans ($\mathbb{V}_{\rm bool}$)
- $\mathbb{V} = \mathbb{V}_{\rm int} \uplus \mathbb{V}_{\rm float} \uplus \mathbb{V}_{\rm bool}$
- $\mathbb{X} = \mathbb{X}_{\rm int} \uplus \mathbb{X}_{\rm float} \uplus \mathbb{X}_{\rm bool}$

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

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

- Then, can be **pointwisely composed** with other abstraction

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

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

- intervals
- polyhedra...

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

- lattice of boolean constants
- relational abstraction with BDDs

Heterogeneous memory states: relational abstraction

- The non relational solution **abstracts away all relations** between data of distinct types
- In many cases, **such relations are necessary**

Relational abstraction of heterogeneous memory states

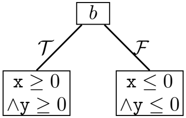
- Build on a per case basis an abstraction of $\mathcal{P}(\mathbb{V}_{t_0}^{N_0} \times \mathbb{V}_{t_0}^{N_0} \times \ldots)$

Concrete states, with
$\mathbb{X}_{\mathrm{bool}} = \{\mathtt{b}\}, \mathbb{X}_{\mathrm{int}} = \{\mathtt{x}, \mathtt{y}\}$

Set of stores characterized by

$$\left\{ \begin{array}{ll} & \mathtt{b} \Rightarrow (\mathtt{x} \geq \mathtt{y} \wedge \mathtt{y} \geq 0) \\ \wedge & \neg \mathtt{b} \Rightarrow (\mathtt{x} \leq \mathtt{y} \wedge \mathtt{y} \leq 0) \end{array} \right.$$

Non relational abstraction with boolean trees and intervals



Memory structures

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

Common structures

- **Structures, records, tuples**
  sequences of cells accessed with fields
- **Arrays**, similar as 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*)

Many other structures can be found:
e.g., closures in functional languages (not studied in this lecture)

Specific kinds of errors

Memory safety

Absence of memory errors

Pointer errors

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

Access errors

- Access to an array **out of its bounds**
- **Buffer overrun**

Specific properties to verify

Many other classes of properties, beyond memory safety

Example:
program closing a list of file descriptors

```
//l points to a list
c = l;
while(c ≠ NULL){
  close(c → FD);
  c = c → next;
}
```

Program specific correctness questions

- `l` is supposed to store all file descriptors at all times
  Will its structure be preserved ?
  Yes, no breakage of a `next` link

Structural preservation properties

- Algorithms manipulating **trees**, **lists**...
- Libraries of algorithms on **balanced trees**
- **Not guaranteed by the language** !

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

A better model

Not all memory cells correspond to a variable

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

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

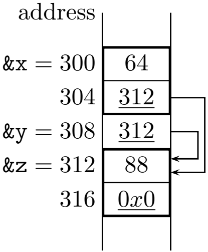
- h is actually only a partial function

Example of a concrete memory state

- x and z are two list elements containing values 64 and 88, and where the former points to the latter
- y stores a pointer to z

Memory layout

(pointer values underlined)



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

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 situation**
- 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`
  - null pointer exception...

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

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...

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

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**:
  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 $\star$)** with a **terminal zero character**
- **Direct access** to string elements (array dereference)
- String copy operation **strcpy**(s, "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

- **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

Outline

Programs: syntax and semantics

- We start with a **basic language**, to be extended with arrays, strings, pointers...
- Memory states comprise an environment and a heap: $\mathbb{M} = \mathbb{E} \times \mathbb{H}$

Basic language

- **L-values**: $l \ ::= \ \mathrm{x} \ (\mathrm{x} \in \mathbb{X})$
- **Expressions**: $e \ ::= \ l \mid c \ (c \in \mathbb{V}) \mid e \oplus e$
- **Statements**: $s \ ::= \ l := e \mid \mathbf{if}(e)\,\{s\}\,\mathbf{else}\,\{s\} \mid \mathbf{while}(e)\,\{s\} \mid s; s$

Semantics

- **L-values**: $[\![l]\!] : \mathbb{M} \rightarrow \mathbb{V}_{\mathrm{addr}}$
- **Expressions**: $[\![e]\!] : \mathbb{M} \rightarrow \mathbb{V}$
- **Programs and statements**:
  - we assume a label **before each statement**
  - each statement defines a **set of transition** ($\rightarrow$)

Programs: extension with arrays

Syntax extension

- A new kind of **l-value**: $l ::= \ldots \mid \mathrm{x}[e]$
- Other constructions **remain the same**

This language is restrictive (no arrays of arrays)
It is sufficient to show the main analysis issues

Semantics extension

- We add a special **"error value"** Ω (propagates)
- **L-values**: $[\![l]\!] : \mathbb{M} \to \mathbb{V}_{\mathrm{addr}}$
  Case of l-value $\mathrm{x}[e]$:
  - if x is a variable of type array, of length n and if $[\![e]\!](e, h) = v \in \mathbb{V}_{\mathrm{int}} \cap [0, n-1]$, then:
    $$[\![\mathrm{x}[e]]\!](e, h) = e(\mathrm{x}) + n$$
  - otherwise $\quad [\![\mathrm{x}[e]]\!](e, h) = \Omega$
- Similar extension for the assignment to an array cell

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. The program will not crash: no index out of bound
2. If the values in the array are in $[0, 100]$ before, they are also in that range after
3. At the end, the array is sorted

Expressing correctness of array operations

Static analysis problem

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

Safety verification

- At label l, analysis computes local abstraction of the set of reachable memory states $\Phi^\sharp(l)$
- 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
$$\forall m \in \gamma_{\mathrm{mem}}(\Phi^\sharp(l)),\ [\![e]\!](m) \in [0, n-1]$$
- In many cases, this can be done with an **interval abstraction**
… but not always

For now, we do not treat the contents of the array

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

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

Elementwise abstraction

Static analysis problem

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** $\mathtt{x}[\mathtt{y}[i+j]+k]$

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:

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

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 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**

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

Array smashing

- All cells of the array are mapped into **one abstract cell** $\bar{\mathtt{t}}$
- **Abstract cells**: $\mathbb{C}^\sharp = \{\&\bar{\mathtt{t}}\} \cup \{\&\mathtt{x}_0, \ldots, \&\mathtt{x}_{m-1}\}$
- $\mathbb{D}^\sharp$ 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 $\mathtt{t}$ is represented using one abstract cell whathever the value of $\mathtt{n}$

Weak updates: transfer function

What is the loss of precision induced by smashing ?

- **Smashing abstraction**, with the **interval abstract domain**
- Array t is supposed **of known length** $n \geq 2$
- We consider statement $l_0 : \mathtt{t[i]} = e;\ l_1$
- Given $m_0^\sharp : \mathbb{C}^\sharp \rightarrow \mathcal{I} \backslash \sqcup^\sharp$, describing a set of states at l, we wish to compute an over-approximation $m_1^\sharp$ of
$$\{m_1 \mid \exists m_0 \in \gamma_{\mathrm{mem}}(m_0^\sharp),\ (l_0, m_0) \rightarrow (l_1, m_1)\}$$
- **Solution**, **assuming the analysis computes** $[a, b]$ as a range over-approximating the value of e ($\forall m_0 \in \gamma_{\mathrm{mem}}(m_0^\sharp),\ [\![e]\!] m_0 \in [a, b]$):
$$\left\{ \begin{array}{rcl} m_1^\sharp(\bar{\mathtt{t}}) & = & m_0^\sharp(\bar{\mathtt{t}}) \sqcup [a, b] \\ m_1^\sharp(\&\mathtt{x}_i) & = & m_0^\sharp(\&\mathtt{x}_i) \end{array} \right.$$
no better solution: the array has several cells, some of which are not affected
- $m_1^\sharp(\bar{\mathtt{t}})$ **always less precise than** $m_0^\sharp(\bar{\mathtt{t}})$

Weak updates

Notion of weak update

Udpate

- **where the modified cell** cannot be computed precisely in the abstract
- **that must be over-approximated in a coarse manner**

In the case of $\mathtt{t[i]} := e$, that may happen:

- 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: 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; at the end: $m^\sharp(\bar{\mathtt{t}}) = \top$

- Array abstractions are fine for coarse properties of array elements
- Weak updates may cause a **serious loss of precision**
  More complex array abstractions are needed

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:
$$\{\& + k \cdot i \cdot s \mid k \cdot i < n\}$$
- useful for arrays of structures
- issue: determine k by analysis

Example array properties

Static analysis problem

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 the manual 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

Composing sortedness predicates

Predicates

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

As part of the proof, predicates need be composed

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

For sorting, **bounds are needed**; for $[0; 3; 9; 2; 4; 8]$, we have:

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

Alternate predicate: $\mathbf{sort}_{\mathtt{t}}(i, j, \min, \max)$, with bounds

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**:

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

- **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

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

- **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

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 *à la C* strings

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

- strcpy(**char** ⋆ d, **char** ⋆ s): copies s into d, including terminating 0, provided there is enough space (unspecified otherwise)
- strncpy(**char** ⋆ d, **char** ⋆ s, **int** 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 declarations

Abstraction of string buffers

Static analysis problem

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

- $\mathtt{t}_{\mathrm{n}}$: size of buffer t
- $\mathtt{t}_{\mathrm{z}}$: position of the first 0 in t if any (otherwise, we let $\mathtt{t}_{\mathrm{z}} = \mathtt{t}_{\mathrm{n}}$)

Abstraction of string buffers

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

Abstract domain

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

$$\begin{array}{rlcl} \gamma_{\mathrm{buf}} : & \mathbb{D}^{\sharp}_{\mathrm{num}} & \longrightarrow & \mathcal{P}(\mathbb{M}) \\ & X^{\sharp} & \longmapsto & \{m \in \mathbb{M} \mid \mathbf{abs}(m) \in \gamma_{\mathrm{num}}(X^{\sharp})\} \end{array}$$

Typical choice: polyhedra

Example

- **Example**: | 'h' | 'e' | 'l' | 'l' | 'o' | '/0' | 'b' | '/0' | 'a' | 'x' | gets abstracted 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**:

```
char q[2];
char s[2];
char t[4];
strcpy(t, "bon");
strncpy(s, t, 2);

strcpy(q, s);
printf("nres:   %s/n", q);
```

$\longmapsto$

$$
\left\{
\begin{array}{l}
q_n = 2; \\
s_n = 2; \\
t_n = 2; \\
t_z = 3; \\
\textbf{if}(t_z < 2)\{s_z = t_z;\} \\
\textbf{else if}(s_z < t_n)\{s_z = s_n\} \\
\textbf{assert}(s_z < q_n); q_z = s_z; \\
\textbf{assert}(q_z < q_n);
\end{array}
\right.
$$

Outline

Programs: syntax and semantics

Syntax extension

- Two new kinds of **l-value**: $l \ ::= \ \ldots \mid \star e \mid l \cdot \mathtt{f}$ ($\mathtt{f}$ structure field)
- A new kind of **expression**: $e \ ::= \ \ldots \mid \& l$

We do not consider **pointer arithmetics here**

Semantics extension

- Case of **l-value** $\star e$:
  - if $[\![e]\!](m) \in \mathbb{V}_{\mathrm{addr}}$ and $m([\![e]\!](m)) = v \in \mathbb{V}$, then $[\![\star e]\!](m) = v$
- Case of **l-value** $l \cdot \mathtt{f}$:
  if $[\![l]\!](m) = v \in \mathbb{V}_{\mathrm{addr}}$, and o is the offset of field $\mathtt{f}$, then
  $[\![l \cdot \mathtt{f}]\!](m) = v + o$
- Case of **expression** $\& l$:
  if $[\![l]\!](m) \in \mathbb{V}_{\mathrm{addr}}$, then $[\![\& l]\!](m) = [\![l]\!](m)$
- Case of **statement** $\star l = e$: similar to that of l-value $\star e$

Example

$$\begin{array}{l}
\textbf{int}\ \mathtt{x}, \mathtt{y};\\
\textbf{int} \star \mathtt{p} = \mathtt{NULL};\\
\textbf{if}(\ldots)\{\\
\quad\quad \mathtt{p} = \&\mathtt{x};\\
\}\textbf{else}\{\\
\quad\quad \mathtt{p} = \&\mathtt{y};\\
\}\\
\mathtt{printf}(''\%d'', \star\mathtt{p});\\
\star\mathtt{p} = \ldots;
\end{array}$$

- The dereference $\star$p will not result in a **null pointer dereference** (interesting to verify by static analysis)
- What cells may be impacted by $\star\mathtt{p} = \ldots$? this is a **weak update**, as p may point to x or y...

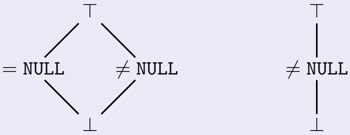
Null pointer analysis

Static analysis problem

Can the program perform a null pointer dereference ? (and crash)

The analysis

- Possible lattices:



- A trivial **non relational abstraction** of pointer values

```
list ⋆ l;
list ⋆ c = l;
//l points to a list
while(c ≠ NULL){
    ... op. on the first element
    c = c -> next;
}
```

- Limited scope, but very light analysis
- **Detection of invalid pointers:** same principle

Dealing with pointers in static analysis

Static analysis problem

- Weak updates must be treated **conservatively**
- How to **resolve weak updates** and **avoid a loss in precision** ?
- Help static analyses in presence of pointers

Examples **of static analyzers that need a pointer analysis**:

- **Astrée**: value analysis, mostly aimed at inferring numerical invariants
- **CSSV**: analyzer for buffer overruns

Pointer aliasing

Aliasing relation

- Given a memory state $m = (e, h)$, pointers p and q are **aliases** if:

$$h(e(\mathrm{p})) = h(e(\mathrm{q}))$$

- Aliasing also extends to references to variables, structure fields...
  Example: $\mathrm{p} := \mathrm{x} \cdot \mathrm{f}$

When **pointers may be aliases**, static analyses have to perform **weak updates**

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

Best result of the analysis ?

- range for x
- range for y

Equivalence relations over access paths

Abstraction to infer pointer aliasing properties

- A notion of **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
- **Example**: $\{\star\mathtt{p}, \mathtt{x}, \mathtt{y}\}$

Examples of aliasing abstractions:

- **set abstractions**: map from access paths to their equivalence class (implementation with union find structures)
- **numerical relations**, to describe aliasing among paths of the form $\mathtt{x}(\text{->}\mathtt{n})^k$

Limitation of pointer analyses

- Pointer analyses hardly work on unbounded memory regions
- Pointer analyses will not capture structural invariants

light algorithms, but not very strong results

Outline

An abstract representation of memory states: shape graphs

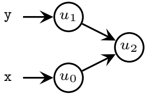
Static analysis problem

Discover complex invariants of programs that manipulate unbounded heap

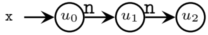
Observation: representation of memory states by shape graphs

- **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:



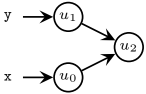
⇒ Basically, we need to over-approximate sets of shape graphs

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

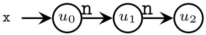
Two alias pointers:



	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:



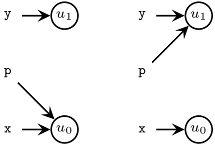
	x
u_0	1
u_1	0
u_2	0

$\cdot\mathtt{n} \mapsto$	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

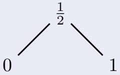
How to abstract away some information ?
i.e., to abstract several graphs into one ?

Example: pointer variable p alias with x or y

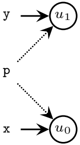


A boolean lattice

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



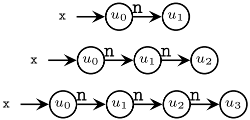
- Graph representation: dotted edges
- **Abstract graph**:



Summary nodes

We cannot talk about unbounded memory states with finitely many nodes

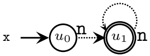
Lists of lengths 1, 2, 3:



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

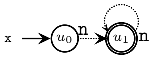
A **summary node** is an atom that may denote several concrete atoms

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:

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

Predicates defined by transitive closure

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

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

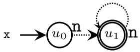
- **Acyclicity**: $\underline{\mathrm{acy}}(v)$
  similar, with a negation

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



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

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

n	u_0	u_1
u_0	0	1
u_1	0	0

Embedding

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

The embedding principle

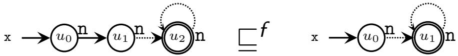
Let $\mathcal{S}_0 = (\mathcal{U}_0, \mathcal{P}, \phi_0)$ and $\mathcal{S}_1 = (\mathcal{U}_1, \mathcal{P}, \phi_1)$ be two three structures, with the same sets of predicates.
Let $f : \mathcal{U}_0 \to \mathcal{U}_1$, surjective.
We say that f **embeds $\mathcal{S}_0$ into $\mathcal{S}_1$** iff

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

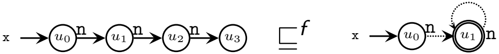
Then, **we write $\mathcal{S}_0 \sqsubseteq^f \mathcal{S}_1$**

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

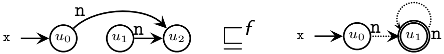
Embedding examples



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

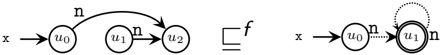
- **2-structure**: a 3-structure $(\mathcal{U}, \mathcal{P}, \phi)$ is a 2-structure, if and only if ϕ always returns in $\{0, 1\}$
- A **2-structure** corresponds to a set of concrete memory states (environment, heap):
  - we simply need to take into account all mappings of addresses into the memory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

Concretization

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

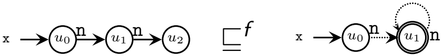
Concretization examples

- **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**:



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

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

Principle for the design of sound transfer functions

Transfer functions for static analysis

- Semantics of concrete statements encoded into boolean formulas
  - example: assignment $\mathrm{y} := \mathrm{x}$
  - new predicate $\mathrm{y}'(u) = \mathrm{x}(u)$
- Evaluation in the abstract is sound (embedding theorem)

Advantages:

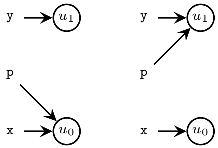
- **abstract transfer functions** derive directly from the concrete transfer functions
  intuition: $\alpha \circ f \circ \gamma$...
- the same solution works for **weakest pre-conditions**

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

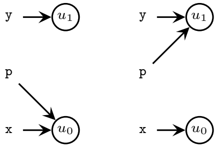
- We assume there are n variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$
  Thus the number of unary predicates is finite
- **Sub-lattice**: structures with atoms **distinguished by the values of the unary predicates** (or *abstraction predicates*) $\mathtt{x}_1, \ldots, \mathtt{x}_n$

Canonical abstraction

Canonical abstraction by truth blurring

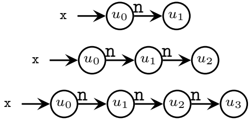
1. **Identify** nodes that **have different abstraction predicates**
2. When several nodes have the **same abstraction predicate** introduce a **summary node**
3. **Compute new predicate values** by doing a join over truth values

Elements not merged:

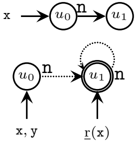


Elements merged:

Lists of lengths 1, 2, 3:



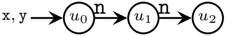
Abstract into:



Assignment: a simple case

Statement $l_0 : \mathtt{y} = \mathtt{y} \texttt{->} \mathtt{n}; l_1 : \ldots$

Pre-condition $\mathcal{S}$

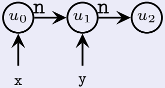


Transfer function

- 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:

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

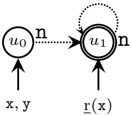
- Result:



This was exactly what we expected

Assignment: a more involved case

Statement $l_0 : \mathtt{y = y \text{->} n};\ l_1 : \ldots$



Pre-condition $\mathcal{S}$

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

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

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

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

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

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

Focus

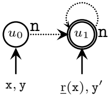
Focusing on a formula

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

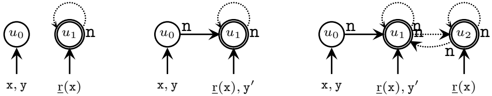
- **f evaluates to 0 or 1** on all elements of $\hat{\mathcal{S}}$
- **precision was gained**: $\forall \mathcal{S}' \in \hat{\mathcal{S}},\ \mathcal{S}' \sqsubseteq \mathcal{S}$
- **soundness is preserved**: $\gamma(\mathcal{S}) = \bigcup \{\gamma(\mathcal{S}') \mid \mathcal{S}' \in \hat{\mathcal{S}}\}$

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

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

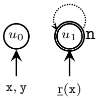


We obtain (we show y and y′):

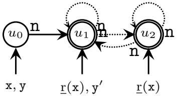


Focus and coerce

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**

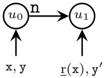


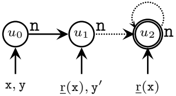
u_0 has an n-field to u_1 so u_1 **cannot be a summary node**

Coerce operation

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

Result:

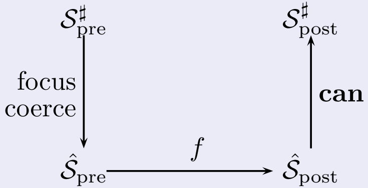




Focus, transfer, abstract...

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 should yield an over-approximation of its input
3. canonicalization over-approximates graph (truth blurring weakening)

A common picture in shape analysis

Outline

Programme overview

November, 23rd. 2012

- **Another family** of shape analyses
- **Combination** of shape abstraction and numerical abstract domains
- Design of **widening** operators in shape analysis

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.
- **[S'96]**: **Points-to analysis in almost linear time.**
  Bjarne Steensgaard. In POPL'96, pages 32–41, 1996.
- **[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.