

Memory abstraction 1

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

Xavier Rival

INRIA, ENS, CNRS

Oct, 16th. 2018

Overview of the lecture

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

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

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

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

This lecture

- **overview memory models** and **memory properties**
- abstraction of **pointer structures** and **separation logic based shape analysis**

Next lecture: arrays, shape/numerical abstraction, composition of shape abstractions

Outline

- 1 Memory models
 - Towards memory properties
 - Formalizing concrete memory states
 - Treatment of errors
 - Language semantics
- 2 Pointer Abstractions
- 3 Separation Logic
- 4 A shape abstract domain relying on separation
- 5 Standard static analysis algorithms
- 6 Conclusion
- 7 Internships

Assumptions for the two lectures

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: $\mathbb{M} = \mathbb{X} \rightarrow \mathbb{V}$)
- **error state**: Ω
- **states**: \mathbb{S}

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

- **transition relation**:

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

Abstraction of sets of states

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

Assumptions: syntax of programs

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

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

Assumptions: semantics of programs

We assume **classical definitions for**:

- **l-values**: $\llbracket l \rrbracket : M \rightarrow X$
- **expressions**: $\llbracket e \rrbracket : M \rightarrow V$
- **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 $\llbracket S \rrbracket_{\mathcal{R}} = \mathbf{lfp} F$ where

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

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

Assumptions: general form of the abstraction

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

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

Reachable states abstraction

We construct $\mathbb{D}^{\#} = \mathbb{L} \rightarrow \mathbb{D}_{\text{mem}}^{\#}$ and:

$$\begin{aligned} \gamma : \mathbb{D}^{\#} &\longrightarrow \mathcal{P}(\mathbb{S}) \\ X^{\#} &\longmapsto \{(l, m) \in \mathbb{S} \mid m \in \gamma_{\text{mem}}(X^{\#}(l))\} \end{aligned}$$

The whole question is how do we choose $\mathbb{D}_{\text{mem}}^{\#}, \gamma_{\text{mem}} \dots$

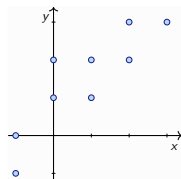
- previous lectures:
 - \mathbb{X} is fixed and finite and, \mathbb{V} is scalars (integers or floats), thus, $\mathbb{M} \equiv \mathbb{V}^n$
- today:
 - we will **extend the language** thus, also **need to extend** $\mathbb{D}_{\text{mem}}^{\#}, \gamma_{\text{mem}}$

Abstraction of purely numeric memory states

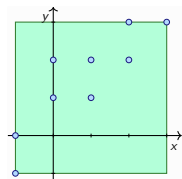
Purely numeric case

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

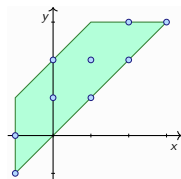
Example: $N = 2$, $\mathbb{X} = \{x, y\}$



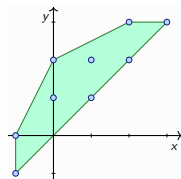
concrete set



interval domain



octagon domain



polyedra domain

Heterogeneous memory states

In real life languages, there are many kinds of values:

- **scalars** (integers of various sizes, boolean, floating-point values)...
- **pointers, arrays**...

Heterogeneous memory states and non relational abstraction

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

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

Non relational abstraction of heterogeneous memory states

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

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

Memory structures

Common structures (non exhaustive list)

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

To describe memories, the definition $\mathbb{M} = \mathbb{X} \rightarrow \mathbb{V}$ is **too restrictive**

Generally, non relational, heterogeneous abstraction cannot handle many such structures all at once: relations are needed!

Specific properties to verify

Memory safety

Absence of memory errors (crashes, or undefined behaviors)

Pointer errors:

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

Access errors:

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

Invariance properties

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

Examples:

- **Preservation of structures**, e.g., lists should remain connected
- **Preservation of invariants**, e.g., of balanced trees

Properties to verify: examples

A program closing a list of file descriptors

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

Correctness properties

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

Examples of structure preservation properties

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

A more realistic model

No one-to-one relation between memory cells and program variables

- a variable may indirectly reference **several cells** (structures...)
- **dynamically allocated cells** correspond to no variable at all...

Environment + Heap

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

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

h is actually only a partial function

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

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

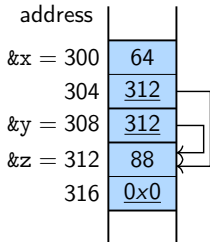
Example of a concrete memory state (variables)

Example setup:

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

Memory layout

(pointer values underlined)



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

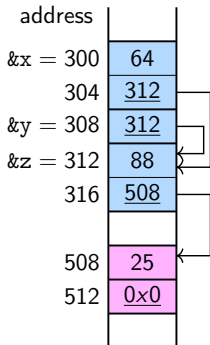
$$\begin{array}{lcl}
 h : & 300 & \mapsto 64 \\
 & 304 & \mapsto 312 \\
 & 308 & \mapsto 312 \\
 & 312 & \mapsto 88 \\
 & 316 & \mapsto 0
 \end{array}$$

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

Example setup:

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

Memory layout



```

e :  x   ↦ 300
     y   ↦ 308
     z   ↦ 312

```

```

hi : 300 ↦ 64
      304 ↦ 312
      308 ↦ 312
      312 ↦ 88
      316 ↦ 508
      508 ↦ 25
      512 ↦ 0

```

Extending the semantic domains

Some slight modifications to the semantics of the initial language:

- **Addresses are values:** $\mathbb{V}_{\text{addr}} \subseteq \mathbb{V}$
- **L-values evaluate into addresses:** $\llbracket \mathbf{l} \rrbracket : \mathbb{M} \rightarrow \mathbb{V}_{\text{addr}}$

$$\llbracket \mathbf{x} \rrbracket(e, h) = e(\mathbf{x})$$

- **Semantics of expressions** $\llbracket e \rrbracket : \mathbb{M} \rightarrow \mathbb{V}$, mostly unchanged

$$\llbracket \mathbf{l} \rrbracket(e, h) = h(\llbracket \mathbf{l} \rrbracket(e, h))$$

- **Semantics of assignment** $l_0 : l := e; l_1 : \dots :$

$$(l_0, e, h_0) \longrightarrow (l_1, e, h_1)$$

where

$$h_1 = h_0[\llbracket \mathbf{l} \rrbracket(e, h_0) \leftarrow \llbracket e \rrbracket(e, h_0)]$$

Realistic definitions of memory states

Our model is still not very accurate for most languages

- Memory cells do not all have the same **size**
- **Memory management algorithms** usually do not treat cells one by one, e.g., **malloc** returns a pointer to a **block** applying **free** to that pointer will dispose the *whole block*

Other refined models

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

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

Language semantics: program crash

In an abnormal situation, we assume that **the program will crash**

- advantage: **very clear semantics**
- disadvantage (for the compiler designer): **dynamic checks** are required

Error state

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

OCaml:

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

Java:

- exception in case of out-of-bound array access, null dereference:
`java.lang.ArrayIndexOutOfBoundsException`

Language semantics: undefined behaviors

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

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

Modeling of undefined behavior

- Very hard to capture what a program operation may modify
- Abnormal situation at (ℓ_0, m_0) such that $\forall m_1 \in \mathbb{M}, (\ell_0, m_0) \rightarrow (\ell_1, m_1)$
- **In C:**
array out-of-bound accesses and dangling pointer dereferences lead to undefined behavior (and potentially, memory corruption) whereas a null-pointer dereference always result into a crash

Composite objects

How are contiguous blocks of information organized ?

Java objects, OCaml struct types

- sets of fields
- each field has a type
- **no assumption** on physical storage, **no pointer arithmetics**

C composite structures and unions

- **physical mapping** defined by the norm
- each field has a specified **size** and a specified **alignment**
- **union types / casts**:
implementations may allow several views

Pointers and records / structures / objects

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

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

Pointers only to records / structures / objects

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

Pointers to fields

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

```
struct {int a; int b} x;  
int * y = &(x · b);
```

Pointer arithmetics

What kind of operations can be performed on a pointer ?

Classical pointer operations

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

Arithmetics on pointers, requiring a more precise model

- **Addition of a numeric constant**:
 $p + n$: address contained in $p + n$ times the size of the type of p
Interaction with pointer casts...
- **Pointer subtraction**: returns a numeric offset

Manual memory management

Allocation of unbounded memory space

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

OCaml memory management

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

C memory management

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

Manual memory management is not safe:

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

Summary on the memory model

Language dependent items

- **Clear error cases** or **undefined behaviors**
for analysis, a semantics with clear error cases is preferable
- **Composite objects**: structure fully exposed or not
- **Pointers to object fields**: allowed or not
- **Pointer arithmetic**: allowed or not
i.e., are pointer values symbolic values or numeric values
- **Memory management**: automatic or manual

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

Rest of the course

Structures for which we introduce abstractions:

- **arrays** (in a few weeks)
- **pointers** and **dynamically allocated pointer structures** (today)

Abstract operations:

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

Outline

- 1 Memory models
- 2 Pointer Abstractions**
- 3 Separation Logic
- 4 A shape abstract domain relying on separation
- 5 Standard static analysis algorithms
- 6 Conclusion
- 7 Internships

Programs with pointers: syntax

Syntax extension: we add pointer operations

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

We do not consider **pointer arithmetics here**

Programs with pointers: semantics

Case of l-values:

$$\begin{aligned} \llbracket \mathbf{x} \rrbracket(e, h) &= e(\mathbf{x}) \\ \llbracket *e \rrbracket(e, h) &= \begin{cases} h(\llbracket e \rrbracket(e, h)) & \text{if } \llbracket e \rrbracket(e, h) \neq 0 \wedge \llbracket e \rrbracket(e, h) \in \mathbf{Dom}(h) \\ \Omega & \text{otherwise} \end{cases} \\ \llbracket \mathbf{l} \cdot \mathbf{f} \rrbracket(e, h) &= \llbracket \mathbf{l} \rrbracket(e, h) + \mathbf{offset}(\mathbf{f}) \text{ (numeric offset)} \end{aligned}$$

Case of expressions:

$$\begin{aligned} \llbracket \mathbf{l} \rrbracket(e, h) &= h(\llbracket \mathbf{l} \rrbracket(e, h)) && \text{(evaluates into the contents)} \\ \llbracket \&\mathbf{l} \rrbracket(e, h) &= \llbracket \mathbf{l} \rrbracket(e, h) && \text{(evaluates into the address)} \end{aligned}$$

Case of statements:

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

Pointer non relational abstractions

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

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

Non relational pointer abstraction

- Set of **pointer abstract values** $\mathbb{D}_{\text{ptr}}^\#$
- **Concretization** $\gamma_{\text{ptr}} : \mathbb{D}_{\text{ptr}}^\# \rightarrow \mathcal{P}(\mathbb{V}_{\text{addr}})$ into pointer sets

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

Pointer non relational abstraction: null pointers

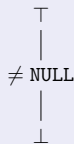
The dereference of a null pointer will cause a crash

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

Null pointer analysis

Abstract domain for addresses:

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



- we may also use a lattice with a fourth element = **NULL**
exercise: what do we gain using this lattice ?
- very **lightweight**, can typically resolve rather trivial cases
- useful for **C**, but also for **Java**

Pointer non relational abstraction: dangling pointers

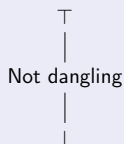
The dereference of a null pointer will cause a crash

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

Null pointer analysis

Abstract domain for addresses:

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



- 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 y ?
9, since **it is not modified at line 5...**

Basic pointer abstraction

- We assume a set of **abstract memory locations** $\mathbb{A}^\#$ is fixed:

$$\mathbb{A}^\# = \{\&x, \&y, \dots, \&t, a_0, a_1, \dots, a_N\}$$

- **Concrete addresses** are **abstracted into** $\mathbb{A}^\#$ by $\phi_{\mathbb{A}} : \mathbb{A} \rightarrow \mathbb{A}^\# \uplus \{\top\}$
- A pointer value is abstracted by the abstraction of the addresses it may point to, i.e., $\mathbb{D}_{\text{ptr}}^\# = \mathcal{P}(\mathbb{A}^\#)$
and $\gamma_{\text{ptr}}(a^\#) = \{a \in \mathbb{A} \mid \phi_{\mathbb{A}}(a) = a^\#\}$

- **example:** p may point to $\{\&x\}$

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;
8: ...

```

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

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

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

Imprecise results

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

Weak-updates

We can formalize this imprecision a bit more:

Weak updates

- **The modified concrete cell cannot be 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**
as in the first example: $p \mapsto \{\&x\}$
- if the points-to set may contain **more than one element**, the analysis needs to perform a **weak-update**
as in the second example: $p \mapsto \{\&x, \&y\}$

Pointer aliasing based on equivalence on access paths

Aliasing relation

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

Abstraction to infer pointer aliasing properties

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

$$a ::= x \mid a \cdot f \mid *a$$

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

Examples of aliasing abstractions:

- **set abstractions**: map from access paths to their equivalence class
(**ex**: $\{\{p_0, p_1, \&x\}, \{p_2, p_3\}, \dots\}$)
- **numerical relations**, to describe aliasing among paths of the form $x(->n)^k$
(**ex**: $\{\{x(->n)^k, \&(x(->n)^{k+1}) \mid k \in \mathbb{N}\}$)

Limitation of basic pointer analyses seen so far

Weak updates:

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

Unsatisfactory abstraction of unbounded memory:

- common assumption that $\mathbb{C}^\#$ **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 abstraction

**A common solution:
shape abstraction**

Outline

- 1 Memory models
- 2 Pointer Abstractions
- 3 Separation Logic**
- 4 A shape abstract domain relying on separation
- 5 Standard static analysis algorithms
- 6 Conclusion
- 7 Internships

Separation logic principle: avoid weak updates

How to deal with weak updates ?

Avoid them !

- Always materialize exactly the cell that needs be modified
- Can be very costly to achieve, and not always feasible
- Notion of property that holds **over a memory region**:
special separating conjunction operator *
- **Local reasoning**:
powerful principle, which allows to consider only part of the memory
- Separation logic has been used in **many contexts**, including **manual verification**, **static analysis**, etc...

Separation logic

Two kinds of formulas:

- **pure formulas** behave like formulas in first-order logic *i.e.*, are not attached to a memory region
- **spatial formulas** describe properties attached to a memory region

Pure formulas denote value properties

$e ::= n$	$(n \in \mathbb{N})$	constants	
	l	l-value	
	$e_0 + e_1$	binary operations	
	\dots		
$P ::= e_0 = e_1$	$ P' \vee P''$	$ P' \wedge P'' \dots$	pure predicates

Pure formulas semantics: $\gamma(P) \subseteq \mathbb{E} \times \mathbb{M}$

Separation logic: points-to predicates

The next slides introduce the main **separation logic formulas** $F ::= \dots$

We start with the most basic predicate, that **describes a single cell**:

Points-to predicate

- **Predicate:**

$F ::= \dots \mid a \mapsto v$ where a is an address and v is a value

- **Concretization:**

$(e, h) \in \gamma(1 \mapsto v)$ if and only if $h = [[1]](e, h) \mapsto v$

- **Example:**

$F = \&x \mapsto 18$ $\&x = 308$

18

- We also note $1 \mapsto e$, as an l-value 1 denotes an address

Separation logic: separating conjunction

Merge of concrete heaps: let $h_0, h_1 \in (\mathbb{V}_{\text{addr}} \rightarrow \mathbb{V})$, such that $\text{dom}(h_0) \cap \text{dom}(h_1) = \emptyset$; then, we let $h_0 \circledast h_1$ be defined by:

$$\begin{aligned} h_0 \circledast h_1 : \quad & \text{dom}(h_0) \cup \text{dom}(h_1) & \longrightarrow & \mathbb{V} \\ & x \in \text{dom}(h_0) & \longmapsto & h_0(x) \\ & x \in \text{dom}(h_1) & \longmapsto & h_1(x) \end{aligned}$$

Separating conjunction

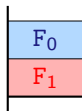
- Predicate:**

$$F ::= \dots \mid F_0 * F_1$$

- Concretization:**

$$\gamma(F_0 * F_1) = \{(e, h_0 \circledast h_1) \mid (e, h_0) \in \gamma(F_0) \wedge (e, h_1) \in \gamma(F_1)\}$$

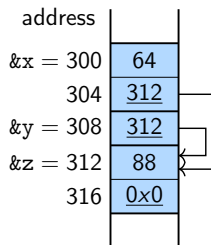
$F_0 * F_1$



An example

Concrete memory layout

(pointer values underlined)



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

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

A formula that abstracts away the addresses:

$$\&x \mapsto \langle 64, \&z \rangle * \&y \mapsto \&z * \&z \mapsto \langle 88, 0 \rangle$$

Separation logic: non separating conjunction

We can also add the **conventional conjunction operator**, with its **usual concretization**:

Non separating conjunction

- **Predicate:**

$$F ::= \dots \mid F_0 \wedge F_1$$

- **Concretization:**

$$\gamma(F_0 \wedge F_1) = \gamma(F_0) \cap \gamma(F_1)$$

Exercise: **describe** and **compare** the concretizations of

- $\&a \mapsto \&b \wedge \&b \mapsto \&a$
- $\&a \mapsto \&b * \&b \mapsto \&a$

Separating conjunction vs non separating conjunction

- **Classical conjunction**: properties for the same memory region
- **Separating conjunction**: properties for **disjoint** memory regions

$$\&a \mapsto \&b \wedge \&b \mapsto \&a$$

- the same heap verifies $\&a \mapsto \&b$ and $\&b \mapsto \&a$
- there can be only **one cell**
- thus $a = b$

$$\&a \mapsto \&b * \&b \mapsto \&a$$

- two **separate** sub-heaps respectively satisfy $\&a \mapsto \&b$ and $\&b \mapsto \&a$
- thus $a \neq b$

- Separating conjunction and non-separating conjunction have **very different properties**
- Both **express very different properties**
e.g., no ambiguity on weak / strong updates

Separating and non separating conjunction

Logic rules of the two conjunction operators of SL:

- **Separating conjunction:**

$$\frac{(e, h_0) \in \gamma(F_0) \quad (e, h_1) \in \gamma(F_1)}{(e, h_0 \circledast h_1) \in \gamma(F_0 * F_1)}$$

- **Non separating conjunction:**

$$\frac{(e, h) \in \gamma(F_0) \quad (e, h) \in \gamma(F_1)}{(e, h) \in \gamma(F_0 \wedge F_1)}$$

**Reminiscent of Linear Logic [Girard87]:
resource aware / non resource aware conjunction operators**

Separation logic: empty store

Empty store

- **Predicate:**

$$F ::= \dots \mid \mathbf{emp}$$

- **Concretization:**

$$\gamma(\mathbf{emp}) = \{(e, []) \mid e \in \mathbb{E}\} = \mathbb{E} \times \{[]\}$$

where $[]$ denotes the empty store

- **emp** is the **neutral element for** $*$
- by contrast the **neutral element for** \wedge is **TRUE**, with concretization:

$$\gamma(\mathbf{TRUE}) = \mathbb{E} \times \mathbb{H}$$

Separation logic: other connectors

Disjunction:

- $F ::= \dots \mid F_0 \vee F_1$
- concretization:

$$\gamma(F_0 \vee F_1) = \gamma(F_0) \cup \gamma(F_1)$$

Spatial implication (*aka, magic wand*):

- $F ::= \dots \mid F_0 \multimap F_1$
- concretization:

$$\gamma(F_0 \multimap F_1) = \{(e, h) \mid \forall h_0 \in \mathbb{H}, (e, h_0) \in \gamma(F_0) \implies (e, h \circledast h_0) \in \gamma(F_1)\}$$

- very powerful connector to describe **structure segments**, used in complex SL proofs

Separation logic

Summary of the main separation logic constructions seen so far:

Separation logic main connectors

$$\begin{aligned}
 \gamma(\mathbf{emp}) &= \mathbb{E} \times \{\emptyset\} \\
 \gamma(\mathbf{TRUE}) &= \mathbb{E} \times \mathbb{H} \\
 \gamma(\mathbf{1} \mapsto \mathbf{v}) &= \{(e, [\![\mathbf{1}]\!] (e, h) \mapsto \mathbf{v}) \mid e \in \mathbb{E}\} \\
 \gamma(\mathbf{F}_0 * \mathbf{F}_1) &= \{(e, h_0 \otimes h_1) \mid (e, h_0) \in \gamma(\mathbf{F}_0) \wedge (e, h_1) \in \gamma(\mathbf{F}_1)\} \\
 \gamma(\mathbf{F}_0 \wedge \mathbf{F}_1) &= \gamma(\mathbf{F}_0) \cap \gamma(\mathbf{F}_1)
 \end{aligned}$$

Concretization of pure formulas is standard

How does this help for program reasoning ?

Separation logic triple

Program proofs based on Hoare triples

- **Notation:** $\{F\}p\{F'\}$ if and only if:

$$\forall s, s' \in \mathbb{S}, s \in \gamma(F) \wedge s' \in \llbracket p \rrbracket(s) \implies s' \in \gamma(F')$$

- Application: **formalize proofs of programs**

A few rules (straightforward proofs):

$$\frac{F_0 \implies F'_0 \quad \{F'_0\}b\{F'_1\} \quad F'_1 \implies F_1}{\{F_0\}b\{F_1\}} \text{ consequence}$$

$$\frac{}{\{\&x \mapsto ?\}x := e\{&x \mapsto e\}} \text{ mutation}$$

$$\frac{x \text{ does not appear in } F}{\{\&x \mapsto ? * F\}x := e\{&x \mapsto e * F\}} \text{ mutation-2}$$

(we assume that e does not allocate memory)

The frame rule

What about the resemblance between rules “mutation” and “mutation-2” ?

Theorem: the frame rule

$$\frac{\{F_0\}b\{F_1\} \quad \text{freevar}(F) \cap \text{write}(b) = \emptyset}{\{F_0 * F\}b\{F_1 * F\}} \text{ frame}$$

- Proof by induction on the logical rules on program statements, *i.e.*, essentially a large case analysis (see biblio for a more complete set of rules)
- Rules are proved by case analysis on the program syntax

The frame rule allows to reason locally about programs

Application of the frame rule

A program with **intermittent invariants**, derived using **the frame rule**, since each step **impacts a disjoint region**:

```

int i;
int * x;
int * y;
{&i ↦? * &x ↦? * &y ↦?}
  x = &i;
{&i ↦? * &x ↦ &i * &y ↦?}
  y = &i;
{&i ↦? * &x ↦ &i * &y ↦ &i}
  *x = 42;
{&i ↦ 42 * &x ↦ &i * &y ↦ &i}

```

Many other program proofs done using separation logic
 e.g., verification of the Deutsch-Shorr-Waite algorithm (biblio)

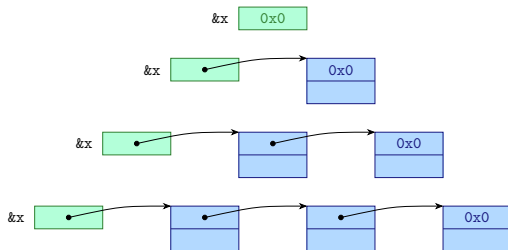
Summarization and inductive definitions

What do we still miss ?

So far, formulas denote **fixed sets of cells**

Thus, no summarization of unbounded regions...

- **Example** all lists pointed to by x , such as:



- How to precisely abstract these stores with **a single formula** *i.e.*, no infinite disjunction ?

Inductive definitions in separation logic

List definition

$$\alpha \cdot \mathbf{list} := \alpha = 0 \wedge \mathbf{emp} \vee \alpha \neq 0 \wedge \alpha \cdot \mathbf{next} \mapsto \delta * \alpha \cdot \mathbf{data} \mapsto \beta * \delta \cdot \mathbf{list}$$

- Formula abstracting our set of structures:

$$\&x \mapsto \alpha * \alpha \cdot \mathbf{list}$$

- **Summarization:**
this formula is finite and describe infinitely many heaps
- **Concretization:** next slide...

Practical implementation in verification/analysis tools

- **Verification:** hand-written definitions
- **Analysis:** either built-in or user-supplied, or partly inferred

Concretization by unfolding

Intuitive semantics of inductive predicates

- Inductive predicates can be **unfolded**, by **unrolling their definitions**
Syntactic unfolding is noted $\xrightarrow{\mathcal{U}}$
- A formula F with inductive predicates describes all stores described by all formulas F' such that $F \xrightarrow{\mathcal{U}} F'$

Example:

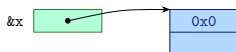
- Let us start with $x \mapsto \alpha_0 * \alpha_0 \cdot \mathbf{list}$; we can unfold it as follows:

$$\&x \mapsto \alpha_0 * \alpha_0 \cdot \mathbf{list}$$

$$\xrightarrow{\mathcal{U}} \quad \&x \mapsto \alpha_0 * \alpha_0 \cdot \mathbf{next} \mapsto \alpha_1 * \alpha_0 \cdot \mathbf{data} \mapsto \beta_1 * \alpha_1 \cdot \mathbf{list}$$

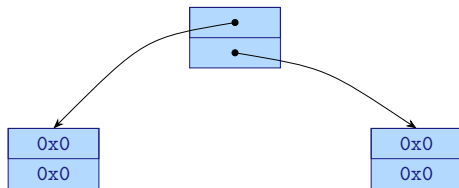
$$\xrightarrow{\mathcal{U}} \quad \&x \mapsto \alpha_0 * \alpha_0 \cdot \mathbf{next} \mapsto \alpha_1 * \alpha_0 \cdot \mathbf{data} \mapsto \beta_1 * \mathbf{emp} \wedge \alpha_1 = \mathbf{0x0}$$

- We get the concrete state below:



Example: tree

- **Example:**



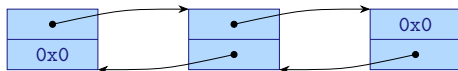
Inductive definition

- **Two recursive calls** instead of one:

$$\begin{aligned}
 \alpha \cdot \mathbf{tree} &:= && \alpha = 0 \wedge \mathbf{emp} \\
 &\vee && \alpha \neq 0 \wedge \alpha \cdot \mathbf{left} \mapsto \beta * \alpha \cdot \mathbf{right} \mapsto \delta \\
 &&& * \beta \cdot \mathbf{tree} * \delta \cdot \mathbf{tree}
 \end{aligned}$$

Example: doubly linked list

- **Example:**



Inductive definition

- We need to propagate the prev pointer as an additional parameter:

$$\alpha \cdot \mathbf{dll}(\delta) \quad := \quad \alpha = 0 \wedge \mathbf{emp} \\ \vee \quad \alpha \neq 0 \wedge \alpha \cdot \mathbf{next} \mapsto \beta * \alpha \cdot \mathbf{prev} \mapsto \delta \\ * \beta \cdot \mathbf{dll}(\alpha)$$

Example: sortedness

- **Example:** sorted list



Inductive definition

- Each element should be greater than the previous one
- The first element simply needs be greater than $-\infty$...
- We need to propagate **the lower bound**, using a scalar parameter

$$\begin{aligned}
 \alpha \cdot \mathbf{lsort}_{\text{aux}}(n) &:= && \alpha = 0 \wedge \mathbf{emp} \\
 &\vee && \alpha \neq 0 \wedge n \leq \beta \wedge \alpha \cdot \mathbf{next} \mapsto \delta \\
 &&& * \alpha \cdot \mathbf{data} \mapsto \beta * \delta \cdot \mathbf{lsort}_{\text{aux}}(\beta)
 \end{aligned}$$

$$\alpha \cdot \mathbf{lsort}() := \alpha \cdot \mathbf{lsort}_{\text{aux}}(-\infty)$$

Outline

- 1 Memory models
- 2 Pointer Abstractions
- 3 Separation Logic
- 4 A shape abstract domain relying on separation**
- 5 Standard static analysis algorithms
- 6 Conclusion
- 7 Internships

Design of an abstract domain

A lot of things are missing to turn SL into an abstract domain

Set of logical predicates:

- separation logic formulas are **very expressive**
e.g., arbitrary **alternations** of \wedge and $*$
- such expressiveness is not necessarily required in static analysis

Representation:

- unstructured formulas can be represented as **ASTs**,
but this representation is **not easy to manipulate efficiently**
- intuition over memory states typically involves **graphs**

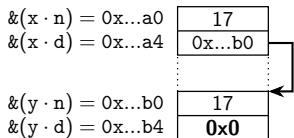
Analysis algorithms:

- inference of “optimal” invariants in SL, with numerical predicates obviously **not computable**

Basic abstraction: structures and their contents (1/2)

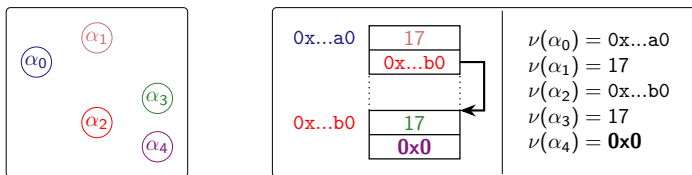
- Concrete memory states

- ▶ very **low level** description
numeric offsets / field names
- ▶ pointers, numeric values:
raw sequences of bits



Basic abstraction: structures and their contents (1/2)

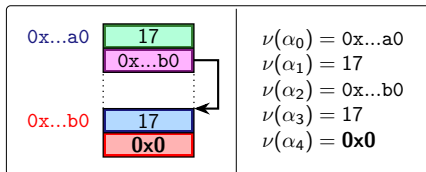
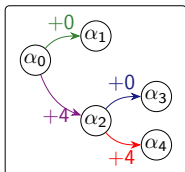
- Concrete memory states
- Abstraction of values into symbolic variables (nodes)



- ▶ characterized by **valuation** ν
- ▶ ν maps **symbolic variables** into **concrete addresses**

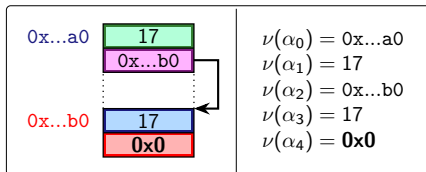
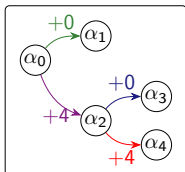
Basic abstraction: structures and their contents (1/2)

- Concrete memory states
- Abstraction of values into symbolic variables / nodes
- Abstraction of regions into points-to edges



Basic abstraction: structures and their contents (1/2)

- Concrete memory states
- Abstraction of values into symbolic variables / nodes
- Abstraction of regions into points-to edges



- Shape graph concretization

$$\gamma_{\text{sh}}(G) = \{(h, \nu) \mid \dots\}$$

valuation ν plays an important role to combine abstraction...

Structure of shape graphs

Valuations bridge the gap between nodes and values

Symbolic variables / **nodes** and intuitively abstract concrete values:

Symbolic variables

We let \mathbb{V}^\sharp denote a countable set of **symbolic variables**; we usually let them be denoted by Greek letters in the following: $\mathbb{V}^\sharp = \{\alpha, \beta, \delta, \dots\}$

When concretizing a shape graph, we need to **characterize how the concrete instance evaluates each symbolic variable**, which is the purpose of the **valuation functions**:

Valuations

A **valuation** is a function from **symbolic variables** into **concrete values** (and is often denoted by ν): $\mathbf{Val} = \mathbb{V}^\sharp \longrightarrow \mathbb{V}$

Note that valuations treat **in the same way addresses** and **raw values**

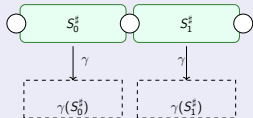
Structure of shape graphs

Distinct edges describe separate regions

In particular, if we **split** a graph into **two parts**:

Separating conjunction

$$\gamma_{\text{sh}}(S_0^\# * S_1^\#) = \{(h_0 \otimes h_1, \nu) \mid (h_0, \nu) \in \gamma_{\text{sh}}(S_0^\#) \wedge (h_1, \nu) \in \gamma_{\text{sh}}(S_1^\#)\}$$



Similarly, when considering the **empty set of edges**, we get the empty heap (where $\mathbb{V}^\#$ is the set of nodes):

$$\gamma_{\text{sh}}(\mathbf{emp}) = \{(\emptyset, \nu) \mid \nu : \mathbb{V}^\# \rightarrow \mathbb{V}\}$$

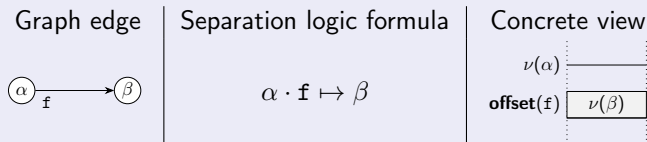
Abstraction of contiguous regions

A single points-to edge represents one heap cell

A **points-to edge** encodes **basic points to predicate in separation logic**:

Points-to edges

- Syntax**



- Concretization:**

$$\gamma_{\text{sh}}(\alpha \cdot \mathbf{f} \mapsto \beta) = \{([\nu(\alpha) + \text{offset}(\mathbf{f}) \mapsto \nu(\beta)], \nu) \mid \nu : \{\alpha, \beta, \dots\} \rightarrow \mathbb{N}\}$$

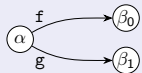
Abstraction of contiguous regions

Contiguous regions are described by adjacent points-to edges

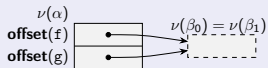
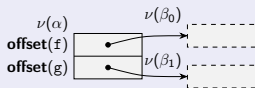
To describe **blocks** containing series of **cells** (e.g., in a **C structure**), shape graphs utilize several outgoing edges from the node representing the base address of the block

Field splitting model

- Separation impacts edges / fields, *not pointers*



- Shape graph accounts for both abstract states below:

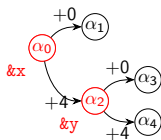
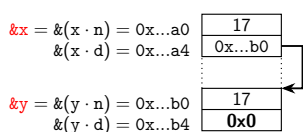


In other words, in a field splitting model, separation:

- asserts addresses are distinct
- says nothing about contents

Abstraction of the environment

Environments bind variables to their (concrete / abstract) address



$$\nu : \alpha_0 \mapsto 0x\dots a0$$

$$\alpha_2 \mapsto 0x\dots b0$$

$$\dots \mapsto \dots$$

$$e^\# : x \mapsto \alpha_0 \quad (\mapsto^\nu 0x\dots a0)$$

$$y \mapsto \alpha_2 \quad (\mapsto^\nu 0x\dots b0)$$

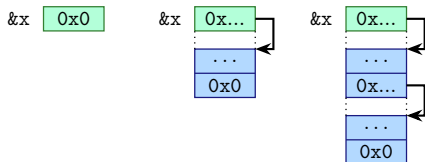
Abstract environments

- An **abstract environment** is a function $e^\#$ from **variables** to **symbolic nodes**
- The **concretization** extends as follows:

$$\gamma_{\text{mem}}(e^\#, S^\#) = \{(e, h, \nu) \mid (h, \nu) \in \gamma_{\text{sh}}(S^\#) \wedge e = \nu \circ e^\#\}$$

Basic abstraction: summarization

Set of all lists of any length:



Well-founded list inductive def.

$$\alpha \cdot \mathbf{list} :=$$

$$(\mathbf{emp} \wedge \alpha = \mathbf{0x0})$$

$$\vee (\alpha \cdot \mathbf{d} \mapsto \beta_0 * \alpha \cdot \mathbf{n} \mapsto \beta_1$$

$$* \beta_1 \cdot \mathbf{list} \wedge \alpha \neq \mathbf{0x0})$$

well-founded predicate

Inductive summary predicates

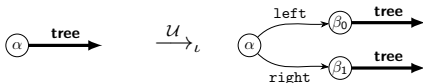
Concretization based on **unfolding** and **least-fixpoint**:

- $\xrightarrow{\mathcal{U}}$ replaces **an $\alpha \cdot \mathbf{list}$ predicate** with **one of its premises**
- $\gamma(S^\#, F) = \bigcup \{ \gamma(S_u^\#, F_u) \mid (S^\#, F) \xrightarrow{\mathcal{U}} (S_u^\#, F_u) \}$

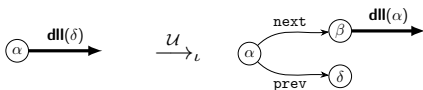
Inductive structures: a few instances

As before, **many interesting inductive predicates** encode nicely into graph inductive definitions:

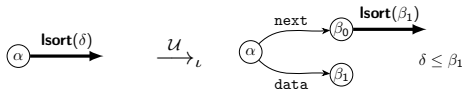
- **More complex shapes: trees**



- **Relations among pointers: doubly-linked lists**

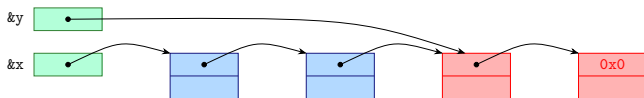


- **Relations between pointers and numerical: sorted lists**



Inductive segments

A frequent pattern:



A first attempt:

- x points to a list, so $\&x \mapsto \alpha * \alpha \cdot \mathbf{list}$ holds
- y points to a list, so $\&y \mapsto \beta * \beta \cdot \mathbf{list}$ holds

However, the following **does not hold**

$$\&x \mapsto \alpha * \alpha \cdot \mathbf{list} * \&y \mapsto \beta * \beta \cdot \mathbf{list}$$

Why ? **violation of separation!**

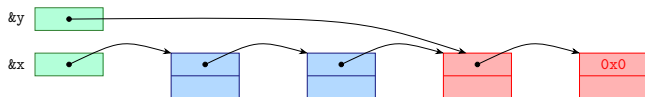
A second attempt:

$$(\&x \mapsto \alpha * \alpha \cdot \mathbf{list} * \mathbf{TRUE}) \wedge (\&y \mapsto \beta * \beta \cdot \mathbf{list} * \mathbf{TRUE})$$

Why is it still not all that good ? **relation lost!**

Inductive segments

A frequent pattern:



Could be **expressed directly** as an inductive with a parameter:

$$\alpha \cdot \text{list_endp}(\pi) ::= (\text{emp}, \alpha = \pi) \\ \quad \mid (\alpha \cdot \text{next} \mapsto \beta_0 * \alpha \cdot \text{data} \mapsto \beta_1 \\ \quad * \beta_0 \cdot \text{list_endp}(\pi), \alpha \neq 0)$$

This definition **straightforwardly derives** from **list**

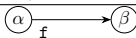
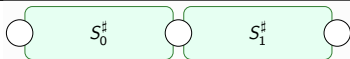
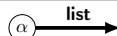
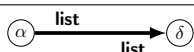
Thus, we make **segments** part of the **fundamental predicates of the domain**



Multi-segments: possible, but harder for analysis

Shape graphs and separation logic

Semantic preserving translation Π of graphs into separation logic formulas:

Graph $S^\# \in \mathbb{D}_{sh}^\#$	Translated formula $\Pi(S^\#)$
	$\alpha \cdot f \mapsto \beta$
	$\Pi(S_0^\#) * \Pi(S_1^\#)$
	$\alpha \cdot \mathbf{list}$
	$\alpha \cdot \mathbf{list_endp}(\delta)$
other inductives and segments	similar

Note that:

- **shape graphs can be encoded into separation logic formula**
- **the opposite is usually not true**

Value information:

- discussed in the next course
- intuitively, assume we maintain numerical information next to shape graphs

Outline

- 1 Memory models
- 2 Pointer Abstractions
- 3 Separation Logic
- 4 A shape abstract domain relying on separation
- 5 Standard static analysis algorithms**
 - Overview of the analysis
 - Post-conditions and unfolding
 - Folding: widening and inclusion checking
 - Abstract interpretation framework: assumptions and results
- 6 Conclusion
- 7 Internships

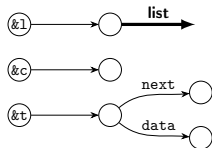
Static analysis overview

A list insertion function:

```

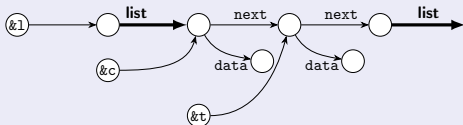
list * l assumed to point to a list
list * t assumed to point to a list element
list * c = l;
while(c != NULL && c -> next != NULL && (...)){
    c = c -> next;
}
t -> next = c -> next;
c -> next = t;
  
```

- **list** inductive structure def.
- Abstract precondition:



Result of the (interprocedural) analysis

- **Over-approximations** of reachable concrete states
e.g., after the insertion:



Transfer functions

Abstract interpreter design

- **Follows the semantics** of the language under consideration
- The abstract domain should provide **sound transfer functions**

Transfer functions:

- **Assignment:** $x \rightarrow f = y \rightarrow g$ or $x \rightarrow f = e_{\text{arith}}$
- **Test:** analysis of conditions (if, while)
- Variable **creation** and **removal**
- **Memory management:** **malloc**, **free**

Abstract operators:

- **Join** and **widening:** over-approximation
- **Inclusion checking:** check stabilization of abstract iterates

Should be **sound** *i.e.*, not forget any concrete behavior

Abstract operations

Denotational style abstract interpreter

- Concrete **denotational semantics** $\llbracket b \rrbracket : \mathbb{S} \longrightarrow \mathcal{P}(\mathbb{S})$
- Abstract post-condition** $\llbracket b \rrbracket^\#(\mathbf{S})$, computed by the analysis:

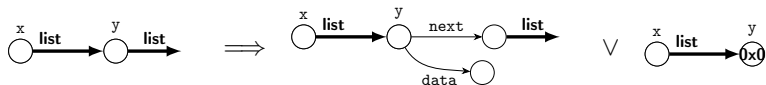
$$s \in \gamma(\mathbf{S}) \implies \llbracket b \rrbracket(s) \subseteq \gamma(\llbracket b \rrbracket^\#(\mathbf{S}))$$

Analysis by induction on the syntax using **domain operators**

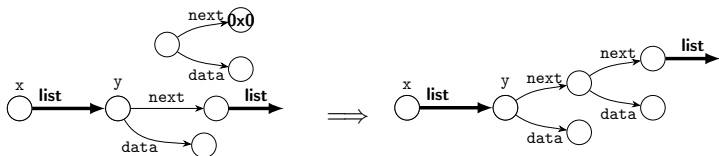
$$\begin{aligned}
 \llbracket b_0; b_1 \rrbracket^\#(\mathbf{S}) &= \llbracket b_1 \rrbracket^\# \circ \llbracket b_0 \rrbracket^\#(\mathbf{S}) \\
 \llbracket l = e \rrbracket^\#(\mathbf{S}) &= \textit{assign}(l, e, \mathbf{S}) \\
 \llbracket l = \text{malloc}(n) \rrbracket^\#(\mathbf{S}) &= \textit{alloc}(l, n, \mathbf{S}) \\
 \llbracket \text{free}(l) \rrbracket^\#(\mathbf{S}) &= \textit{free}(l, n, \mathbf{S}) \\
 \llbracket \text{if}(e) b_t \text{ else } b_f \rrbracket^\#(\mathbf{S}) &= \begin{cases} \textit{join}(\llbracket b_t \rrbracket^\#(\textit{test}(e, \mathbf{S})), \\ \llbracket b_f \rrbracket^\#(\textit{test}(e = \text{false}, \mathbf{S}))) \end{cases} \\
 \llbracket \text{while}(e)b \rrbracket^\#(\mathbf{S}) &= \textit{test}(e = \text{false}, \textit{lfp}_s^\# F^\#) \\
 &\text{ where, } F^\# : \mathbf{S}_0 \mapsto \llbracket b \rrbracket^\#(\textit{test}(e, \mathbf{S}_0))
 \end{aligned}$$

The algorithms underlying the transfer functions

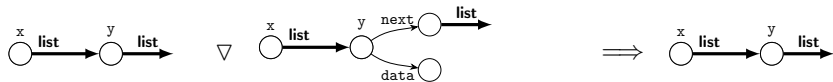
- **Unfolding:** cases analysis on summaries



- **Abstract postconditions,** on “exact” regions, e.g. insertion



- **Widening:** builds summaries and ensures termination



Outline

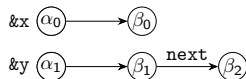
- 1 Memory models
- 2 Pointer Abstractions
- 3 Separation Logic
- 4 A shape abstract domain relying on separation
- 5 Standard static analysis algorithms**
 - Overview of the analysis
 - Post-conditions and unfolding
 - Folding: widening and inclusion checking
 - Abstract interpretation framework: assumptions and results
- 6 Conclusion
- 7 Internships

Analysis of an assignment in the graph domain

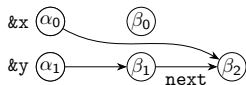
Steps for analyzing $x = y \rightarrow \text{next}$ (local reasoning)

- 1 Evaluate **l-value** x into **points-to edge** $\alpha \mapsto \beta$
- 2 Evaluate **r-value** $y \rightarrow \text{next}$ into **node** β'
- 3 Replace points-to edge $\alpha \mapsto \beta$ with **points-to edge** $\alpha \mapsto \beta'$

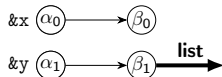
With pre-condition:



- Step 1 produces $\alpha_0 \mapsto \beta_0$
- Step 2 produces β_2
- End result:



With pre-condition:



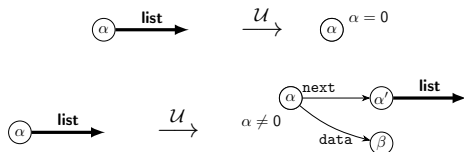
- Step 1 produces $\alpha_0 \mapsto \beta_0$
- **Step 2 fails**

- Abstract state **too abstract**
- We need to **refine it**

Unfolding as a local case analysis

Unfolding principle

- **Case analysis**, based on the inductive definition
- Generates **symbolic disjunctions** (analysis performed in a **disjunction domain**, e.g., trace partitioning)
- Example, for lists:



- **Numeric predicates:** next course on shape + value abstraction

Soundness: by definition of the concretization of inductive structures

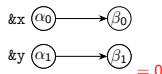
$$\gamma_{\text{sh}}(S^\#) \subseteq \bigcup \{ \gamma_{\text{sh}}(S_0^\#) \mid S^\# \xrightarrow{\mathcal{U}} S_0^\# \}$$

Analysis of an assignment, with unfolding

Principle

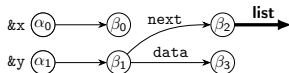
- We have $\gamma_{\text{sh}}(\alpha \cdot \iota) = \bigcup \{ \gamma_{\text{sh}}(S^\#) \mid \alpha \cdot \iota \xrightarrow{\mathcal{U}} S^\# \}$
- Replace $\alpha \cdot \iota$ with a finite number of disjuncts and continue

Disjunct 1:

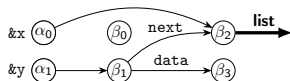


- Step 1 produces $\alpha_0 \mapsto \beta_0$
- **Step 2 fails: Null pointer !**
- In a **correct** program, would be ruled out by a **condition** $y \neq 0$ i.e., $\beta_1 \neq 0$ in $\mathbb{D}_{\text{num}}^\#$

Disjunct 2:



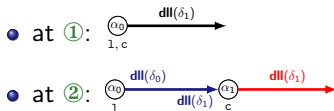
- Step 1 produces $\alpha_0 \mapsto \beta_0$
- Step 2 produces β_2
- **End result:**



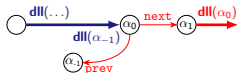
Unfolding and degenerated cases

```

assume(l points to a dll)
c = l;
① while(c ≠ NULL && condition)
  c = c -> next;
② if(c ≠ 0 && c -> prev ≠ 0)
  c = c -> prev -> prev;
  
```

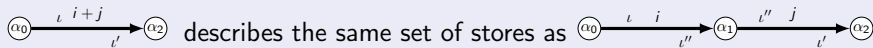


⇒ **non trivial unfolding**



- **Materialization of $c \rightarrow \text{prev}$:**

Segment splitting lemma: basis for segment unfolding



- **Materialization of $c \rightarrow \text{prev} \rightarrow \text{prev}$:**
- **Implementation issue:** discover **which inductive edge** to unfold **very hard !**

Outline

- 1 Memory models
- 2 Pointer Abstractions
- 3 Separation Logic
- 4 A shape abstract domain relying on separation
- 5 Standard static analysis algorithms**
 - Overview of the analysis
 - Post-conditions and unfolding
 - Folding: widening and inclusion checking
 - Abstract interpretation framework: assumptions and results
- 6 Conclusion
- 7 Internships

Need for a folding operation

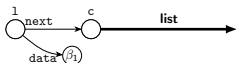
Back to the **list traversal** example:

First iterates in the loop:

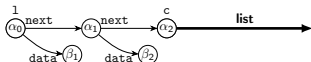
- at **iteration 0** (before entering the loop):



- at **iteration 1**:

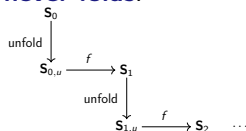


- at **iteration 2**:



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

The analysis **unfolds**, but **never folds**:



- How to guarantee **termination** of the analysis ?
- How to **introduce segment edges** / perform **abstraction** ?

Widening

- The lattice of shape abstract values has **infinite height**
- Thus iteration sequences **may not terminate**

Definition of a widening operator ∇

- **Over-approximates join:**

$$\begin{cases} \gamma(X^\#) \subseteq \gamma(X^\# \nabla Y^\#) \\ \gamma(Y^\#) \subseteq \gamma(X^\# \nabla Y^\#) \end{cases}$$

- **Enforces termination:** for all sequence $(X_n^\#)_{n \in \mathbb{N}}$, the **sequence $(Y_n^\#)_{n \in \mathbb{N}}$ defined below is ultimately stationary**

$$\begin{cases} Y_0^\# = X_0^\# \\ \forall n \in \mathbb{N}, Y_{n+1}^\# = Y_n^\# \nabla X_{n+1}^\# \end{cases}$$

Canonicalization

Upper closure operator

$\rho : \mathbb{D}^\# \longrightarrow \mathbb{D}_{\text{can}}^\# \subseteq \mathbb{D}^\#$ is an **upper closure operator** (uco) iff it is monotone, extensive and idempotent.

Canonicalization

- **Disjunctive completion:** $\mathbb{D}_\vee^\# =$ finite disjunctions over $\mathbb{D}^\#$
- **Canonicalization operator** ρ_\vee defined by $\rho_\vee : \mathbb{D}_\vee^\# \longrightarrow \mathbb{D}_{\text{can}\vee}^\#$ and $\rho_\vee(X^\#) = \{\rho(x^\#) \mid x^\# \in X^\#\}$ where ρ is an uco and $\mathbb{D}_{\text{can}}^\#$ has finite height
- Canonicalization is used in **many shape analysis tools**
- **Easier to compute** but **less powerful** than widening: does not exploit history of computation

Weakening: definition

To design **inclusion test**, **join** and **widening** algorithms, we first study a more general notion of **weakening**:

Weakening

We say that S_0^\sharp **can be weakened into** S_1^\sharp if and only if

$$\forall (h, \nu) \in \gamma_{\text{sh}}(S_0^\sharp), \exists \nu' \in \mathbf{Val}, (h, \nu') \in \gamma_{\text{sh}}(S_1^\sharp)$$

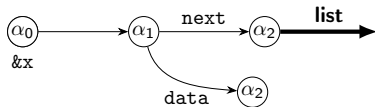
We then note $S_0^\sharp \preceq S_1^\sharp$

Applications:

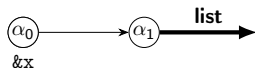
- **inclusion test** (comparison) inputs S_0^\sharp, S_1^\sharp ; if returns true $S_0^\sharp \preceq S_1^\sharp$
- **canonicalization** (unary weakening) inputs S_0^\sharp and returns $\rho(S_0^\sharp)$ such that $S_0^\sharp \preceq \rho(S_0^\sharp)$
- **widening / join** (binary weakening ensuring termination or not) inputs S_0^\sharp, S_1^\sharp and returns S_{up}^\sharp such that $S_i^\sharp \preceq S_{\text{up}}^\sharp$

Weakening: example

We consider S_0^\sharp defined by:



and S_1^\sharp defined by:



Then, we have **the weakening** $S_0^\sharp \preceq S_1^\sharp$ **up-to a renaming in** S_1^\sharp :

$$\Psi : \begin{array}{l} \beta_0 \longmapsto \alpha_0 \\ \beta_1 \longmapsto \alpha_1 \end{array}$$

- weakening **up-to renaming** is generally required as graphs do not have the same name space
- formalized a bit later...

Local weakening: separating conjunction rule

We can apply the local reasoning principle to weakening

If $S_0^\# \preceq S_{0,\text{weak}}^\#$ and $S_1^\# \preceq S_{1,\text{weak}}^\#$ then:

Separating conjunction rule (\preceq_*)

Let us assume that

- $S_0^\#$ and $S_1^\#$ have distinct set of **source nodes**
- we can weaken $S_0^\#$ into $S_{0,\text{weak}}^\#$
- we can weaken $S_1^\#$ into $S_{1,\text{weak}}^\#$

then:

we can weaken $S_0^\# * S_1^\#$ into $S_{0,\text{weak}}^\# * S_{1,\text{weak}}^\#$

Local weakening: unfolding rule, identity rule

Weakening unfolded region (\preceq_u)

Let us assume that $S_0^\# \xrightarrow{u} S_1^\#$. Then, by definition of the concretization of unfolding

we can weaken $S_1^\#$ into $S_0^\#$

- the proof follows from the definition of unfolding
- it can be applied locally, on graph regions that differ due to unfolding of inductive definitions

Identity weakening (\preceq_{Id})

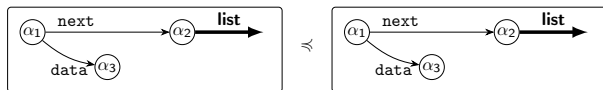
we can weaken $S^\#$ into $S^\#$

- the proof is trivial:

$$\gamma_{\text{sh}}(S^\#) \subseteq \gamma_{\text{sh}}(S^\#)$$

- on itself, this principle is not very useful, but it can be applied locally, and combined with (\preceq_u) on graph regions that are not equal

Local weakening: example

By **rule** (\approx_{Id}):Thus, by **rule** (\approx_{U}):Additionally, by **rule** (\approx_{Id}):Thus, by **rule** (\approx_{*}):

Inclusion checking rules in the shape domain

Graphs to compare have distinct sets of nodes, thus inclusion check should carry out a **valuation transformer** $\Psi : \mathbb{V}^\sharp(S_1^\sharp) \longrightarrow \mathbb{V}^\sharp(S_0^\sharp)$ (important when dealing also with content values)

Using (and extending) the weakening principles, we obtain the following rules (considering only inductive definition **list**, though these rules would extend to other definitions straightforwardly):

- **Identity rules:**

$$\begin{aligned} \forall i, \Psi(\beta_i) = \alpha_i &\implies \alpha_0 \cdot f \mapsto \alpha_1 \sqsubseteq^\sharp_\Psi \beta_0 \cdot f \mapsto \beta_1 \\ \Psi(\beta) = \alpha &\implies \alpha \cdot \mathbf{list} \sqsubseteq^\sharp_\Psi \beta \cdot \mathbf{list} \\ \forall i, \Psi(\beta_i) = \alpha_i &\implies \alpha_0 \cdot \mathbf{list_endp}(\alpha_1) \sqsubseteq^\sharp_\Psi \beta_0 \cdot \mathbf{list_endp}(\beta_1) \end{aligned}$$

- **Rules on inductives:**

$$\begin{aligned} \forall i, \Psi(\beta_i) = \alpha &\implies \mathbf{emp} \sqsubseteq^\sharp_\Psi \beta_0 \cdot \mathbf{list_endp}(\beta_1) \\ S_0^\sharp \sqsubseteq^\sharp_\Psi S_1^\sharp \wedge \beta \cdot \iota \xrightarrow{u} S_1^\sharp &\implies S_0^\sharp \sqsubseteq^\sharp_\Psi \beta \cdot \iota \\ \text{if } \beta_1 \text{ fresh, } \Psi' = \Psi[\beta_1 \mapsto \alpha_1] \text{ and } \Psi(\beta_0) = \alpha_0 \text{ then,} \\ S_0^\sharp \sqsubseteq^\sharp_{\Psi'} \beta_1 \cdot \mathbf{list} &\implies \alpha_0 \cdot \mathbf{list_endp}(\alpha_1) * S_0^\sharp \sqsubseteq^\sharp_\Psi \beta_0 \cdot \iota \end{aligned}$$

Inclusion checking algorithm

Comparison of (e_0^\sharp, S_0^\sharp) and (e_1^\sharp, S_1^\sharp)

- 1 start with Ψ defined by $\Psi(\beta) = \alpha$ if and only if there exists a variable x such that $e_0^\sharp(x) = \alpha \wedge e_1^\sharp(x) = \beta$
 - 2 iteratively **apply local rules**, and extend Ψ when needed
 - 3 return **true** when both shape graphs become empty
- the first step ensures both environments are consistent

This algorithm is sound:

Soundness

$$(e_0^\sharp, S_0^\sharp) \sqsubseteq^\sharp (e_1^\sharp, S_1^\sharp) \implies \gamma(e_0^\sharp, S_0^\sharp) \subseteq \gamma(e_1^\sharp, S_1^\sharp)$$

Over-approximation of union

The principle of join and widening algorithm **is similar to that of \sqsubseteq^\sharp** :

- It can be computed **region by region**, as for weakening in general:
If $\forall i \in \{0, 1\}, \forall s \in \{\text{left}, \text{right}\}, S_{i,s}^\sharp \preceq S_s^\sharp$,



The partitioning of inputs / different nodes sets requires a **node correspondence function**

$$\Psi : \mathbb{V}^\sharp(S_{\text{left}}^\sharp) \times \mathbb{V}^\sharp(S_{\text{right}}^\sharp) \longrightarrow \mathbb{V}^\sharp(S^\sharp)$$

- The computation of the shape join progresses by the application of **local join rules**, that produce a **new (output) shape graph, that weakens both inputs**

Over-approximation of union: syntactic identity rules

In the next few slides, we focus on ∇
though the abstract union would be defined similarly in the shape domain

Several rules derive **from** (\preceq_{id}) :

- If $S_{\text{lft}}^{\#} = \alpha_0 \cdot \mathbf{f} \mapsto \alpha_1$
and $S_{\text{lft}}^{\#} = \beta_0 \cdot \mathbf{f} \mapsto \beta_1$
and $\Psi(\alpha_0, \beta_0) = \delta_0$, $\Psi(\alpha_1, \beta_1) = \delta_1$, then:

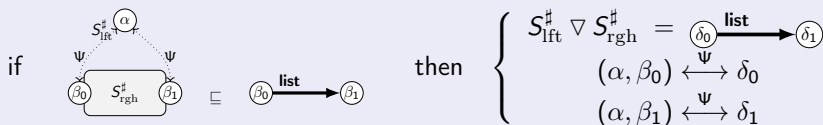
$$S_{\text{lft}}^{\#} \nabla S_{\text{rgh}}^{\#} = \delta_0 \cdot \mathbf{f} \mapsto \delta_1$$

- If $S_{\text{lft}}^{\#} = \alpha_0 \cdot \mathbf{list}$
and $S_{\text{lft}}^{\#} = \beta_0 \cdot \mathbf{list}_1$
and $\Psi(\alpha_0, \beta_0) = \delta_0$, then:

$$S_{\text{lft}}^{\#} \nabla S_{\text{rgh}}^{\#} = \delta_0 \cdot \mathbf{list}$$

Over-approximation of union: segment introduction rule

Rule

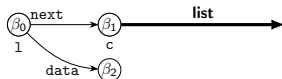


Application to list traversal, at the end of iteration 1:

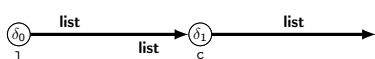
- before iteration 0:



- end of iteration 0:



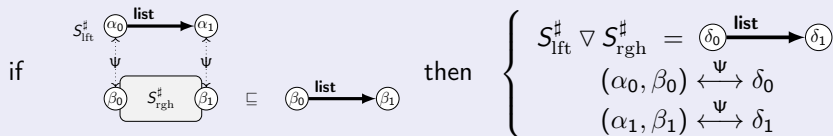
- join, before iteration 1:



$$\left\{ \begin{array}{l} \Psi(\alpha_0, \beta_0) = \delta_0 \\ \Psi(\alpha_0, \beta_1) = \delta_1 \end{array} \right.$$

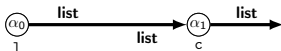
Over-approximation of union: segment extension rule

Rule

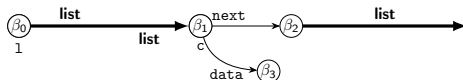


Application to list traversal, at the end of iteration 1:

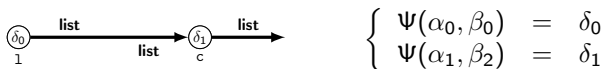
- previous invariant before iteration 1:



- end of iteration 1:



- join, before iteration 1:



Over-approximation of union: rewrite system properties

- Comparison, canonicalization and widening algorithms can be considered **rewriting systems over tuples of graphs**
- **Success configuration**: weakening applies on all components, *i.e.*, the inputs are fully **“consumed”** in the weakening process
- **Failure configuration**: some components **cannot be weakened** *i.e.*, the algorithm should return the conservative answer (*i.e.*, \top)

Termination

- The systems are **terminating**
- This ensures comparison, canonicalization, widening are **computable**

Non confluence !

- The results depends on the order of application of the rules
- Implementation requires the choice of an **adequate strategy**

Over-approximation of union in the combined domain

Widening of (e_0^\sharp, S_0^\sharp) and (e_1^\sharp, S_1^\sharp)

- 1 define Ψ, e by $\Psi(\alpha, \beta) = e(x) = \delta$ (where δ is a fresh node) if and only if $e_0^\sharp(x) = \alpha \wedge e_1^\sharp(x) = \beta$
- 2 iteratively **apply join local rules**, and extend Ψ when new relations are inferred (for instance for points-to edges)
- 3 return the result obtained when all regions of both inputs are approximated in the output graph

This algorithm is sound:

Soundness

$$\gamma(e_0^\sharp, S_0^\sharp) \cup \gamma(e_1^\sharp, S_1^\sharp) \subseteq \gamma(e^\sharp, S^\sharp)$$

Widening also enforces **termination** (it only introduces segments, and the growth induced by the introduction of segments is bounded)

Outline

- 1 Memory models
- 2 Pointer Abstractions
- 3 Separation Logic
- 4 A shape abstract domain relying on separation
- 5 Standard static analysis algorithms**
 - Overview of the analysis
 - Post-conditions and unfolding
 - Folding: widening and inclusion checking
 - Abstract interpretation framework: assumptions and results
- 6 Conclusion
- 7 Internships

Assumptions

What assumptions do we make ?
How do we prove soundness of the analysis of a loop ?

- **Assumptions in the concrete level**, and for block b :

$(\mathcal{P}(\mathbb{M}), \subseteq)$ is a complete lattice, hence a CPO

$F : \mathcal{P}(\mathbb{M}) \rightarrow \mathcal{P}(\mathbb{M})$ is the concrete semantic (“post”) function of b

thus, the concrete semantics writes down as $\llbracket b \rrbracket = \text{lfp}_\emptyset F$

- **Assumptions in the abstract level:**

$\mathbb{M}^\#$ set of abstract elements, no order a priori

$m^\# ::= (e^\#, S^\#)$

$\gamma_{\text{mem}} : \mathbb{M}^\# \rightarrow \mathcal{P}(\mathbb{M})$ concretization

$F^\# : \mathbb{M}^\# \rightarrow \mathbb{M}^\#$ sound abstract semantic function

i.e., such that $F \circ \gamma_{\text{mem}} \subseteq \gamma_{\text{mem}} \circ F^\#$

$\nabla : \mathbb{M}^\# \times \mathbb{M}^\# \rightarrow \mathbb{M}^\#$ widening operator, terminates, and such that

$\gamma_{\text{mem}}(m_0^\#) \cup \gamma_{\text{mem}}(m_1^\#) \subseteq \gamma_{\text{mem}}(m_0^\# \nabla m_1^\#)$

Computing a loop abstract post-condition

Loop abstract semantics

The abstract semantics of loop **while**(**rand**())**{b}** is calculated as the limit of the sequence of abstract iterates below:

$$\begin{cases} m_0^\# &= \perp \\ m_{n+1}^\# &= m_n^\# \nabla F^\#(m_n^\#) \end{cases}$$

Soundness proof:

- by induction over n , $\bigcup_{k \leq n} F^k(\emptyset) \subseteq \gamma_{\text{mem}}(m_n^\#)$
- by the property of widening, the abstract sequence converges at a rank N :
 $\forall k \geq N$, $m_k^\# = m_N^\#$, thus

$$\text{Ifp}_\emptyset F = \bigcup_k F^k(\emptyset) \subseteq \gamma_{\text{mem}}(m_N^\#)$$

Discussion on the abstract ordering

How about the abstract ordering ? We assumed *NONE* so far...

- **Logical ordering**, induced by concretization, used for **proofs**

$$m_0^\sharp \sqsubseteq m_1^\sharp \quad ::= \quad " \gamma_{\text{mem}}(m_0^\sharp) \subseteq \gamma_{\text{mem}}(m_1^\sharp) "$$

- **Approximation of the logical ordering**, **implemented** as a function **is_le** : $\mathbb{M}^\sharp \times \mathbb{M}^\sharp \rightarrow \{\mathbf{true}, \top\}$, used to **test the convergence of abstract iterates**

$$\mathbf{is_le}(m_0^\sharp, m_1^\sharp) = \mathbf{true} \quad \implies \quad \gamma_{\text{mem}}(m_0^\sharp) \subseteq \gamma_{\text{mem}}(m_1^\sharp)$$

Abstract semantics is not assumed (and is actually most likely NOT) monotone with respect to either of these orders...

- Also, **computational ordering** would be used for **proving widening termination**

Outline

- 1 Memory models
- 2 Pointer Abstractions
- 3 Separation Logic
- 4 A shape abstract domain relying on separation
- 5 Standard static analysis algorithms
- 6 Conclusion**
- 7 Internships

Updates and summarization

**Weak updates cause significant precision loss...
Separation logic makes updates strong**

Separation logic

Separating conjunction combines properties on disjoint stores

- Fundamental idea: * **forces to identify what is modified**
- Before an **update** (or a **read**) takes place, memory cells need to be **materialized**
- **Local reasoning**: properties on unmodified cells pertain

Summaries

Inductive predicates describe unbounded memory regions

- Last lecture: **array segments** and **transitive closure** (TVLA)

Bibliography

- **[JR]: Separation Logic: A Logic for Shared Mutable Data Structures.**
John C. Reynolds.
In LICS'02, pages 55–74, 2002.
- **[DHY]: A Local Shape Analysis Based on Separation Logic.**
Dino Distefano, Peter W. O'Hearn and Hongseok Yang.
In TACAS'06, pages 287–302.
- **[CR]: Relational inductive shape analysis.**
Bor-Yuh Evan Chang and Xavier Rival.
In POPL'08, pages 247–260, 2008.

Assignment and paper reading

The Frame rule:

- formalize the Hoare logic rules for a language with pointer assignments and condition tests
- prove the Frame rule by induction over the syntax of programs

Reading:

Separation Logic: A Logic for Shared Mutable Data Structures.

John C. Reynolds.

In LICS'02, pages 55–74, 2002.

Formalizes the Frame rule, among others

Outline

- 1 Memory models
- 2 Pointer Abstractions
- 3 Separation Logic
- 4 A shape abstract domain relying on separation
- 5 Standard static analysis algorithms
- 6 Conclusion
- 7 Internships**

Internship on memory abstraction

Summarization based on universal quantification:

- memory abstractions use **summarization** for arrays, arrays segments, linked structures...
- **another form of summarization** based on an **unbounded set E**

$$*\{P(x) \mid x \in E\}$$

definition of fold / unfold, analysis operations...

- analysis of new kinds of structures, e.g., **union finds**

Proof of correctness properties of a process scheduler for Linux

(PhD funding available):

- schedulers rely on very complex structures
overlaid lists / balanced trees
- relations between memory properties and the **high level scheduling policy**
- part of a project for the **generation of verified schedulers**