Memory abstraction 1

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

Xavier Rival

INRIA, ENS, CNRS

Jan, 22nd. 2021

Overview of the lecture

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

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

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

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

This lecture

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

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

Outline

Assumptions for the two lectures on memory abstraction

Imperative programs viewed as **transition systems**:

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

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

- **transition relation:**

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

Abstraction of sets of states

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

Assumptions: syntax of programs

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

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

Assumptions: semantics of programs

We assume **classical definitions for:**

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

In this course, we rely on the usual **reachable states semantics**

Reachable states semantics

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

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

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

Reachable states abstraction

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

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

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

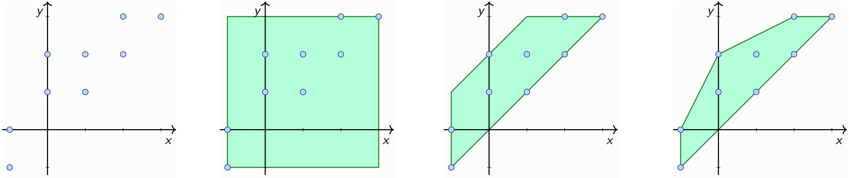
- previous lectures:
  $\mathbb{X}$ **is fixed and finite** and, $\mathbb{V}$ **is scalars** (integers or floats), thus, $\mathbb{M} \equiv \mathbb{V}^n$
- today:
  we will **extend the language** thus, also **need to extend** $\mathbb{D}^\sharp_{\mathrm{mem}}, \gamma_{\mathrm{mem}}$

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

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

Non relational abstraction of heterogeneous memory states

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

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

Memory structures

Common structures (non exhaustive list)

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

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

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

Specific properties to verify

Memory safety

Absence of memory errors (crashes, or undefined behaviors)

Pointer errors:

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

Access errors:

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

Invariance properties

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

Examples:

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

Properties to verify: examples

A program closing a list of file descriptors

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

Correctness properties

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

Examples of structure preservation properties

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

A more realistic model

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

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

Environment + Heap

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

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

  h is actually only a partial function
- **Memory states** (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")**

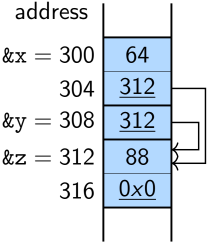
Example of a concrete memory state (variables)

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)



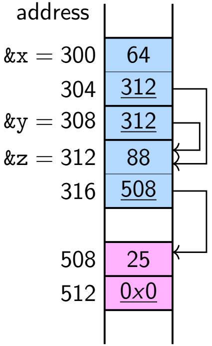
$$
\begin{array}{rlcl}
e: & \mathtt{x} & \mapsto & 300 \\
 & \mathtt{y} & \mapsto & 308 \\
 & \mathtt{z} & \mapsto & 312 \\
 & & & \\
h: & 300 & \mapsto & 64 \\
 & 304 & \mapsto & 312 \\
 & 308 & \mapsto & 312 \\
 & 312 & \mapsto & 88 \\
 & 316 & \mapsto & 0
\end{array}
$$

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

Example setup:

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

Memory layout



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

Extending the semantic domains

Some slight modifications to the semantics of the initial language:

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

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

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

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

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

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

where

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

Realistic definitions of memory states

Our model is still not very accurate for most languages

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

Other refined models

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

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

Language semantics: program crash

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

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

Error state

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

OCaml:

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

Java:

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

Language semantics: undefined behaviors

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

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

Modeling of undefined behavior

- Very hard to capture what a program operation may modify
- Abnormal situation at (l_0, m_0) such that $\forall m_1 \in \mathbb{M},\ (l_0, m_0) \rightarrow (l_1, m_1)$

- **In C**:
  array out-of-bound accesses and dangling pointer dereferences lead to undefined behavior (and potentially, memory corruption) whereas a null-pointer dereference always result into a crash

Composite objects

How are contiguous blocks of information organized ?

Java objects, OCaml struct types

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

C composite structures and unions

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

Pointers and records / structures / objects

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

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

Pointers only to records / structures / objects

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

Pointers to fields

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

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

Pointer arithmetics

What kind of operations can be performed on a pointer ?

Classical pointer operations

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

Arithmetics on pointers, requiring a more precise model

- **Addition of a numeric constant**:
  $\mathtt{p} + n$: address contained in $\mathtt{p} + n$ 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

Abstraction for pointers and dynamic data-structures:

- **pointer abstractions**
- **separation logic**-based abstraction for **dynamic structures**
- **three-valued logic**-based abstraction for **dynamic structures**
- **combination** of **value** and **structure** abstractions

Abstract operations:

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

Outline

Programs with pointers: syntax

Syntax extension: we add pointer operations

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

We do not consider **pointer arithmetics here**

Programs with pointers: semantics

Case of l-values:

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

Case of expressions:

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

Case of statements:

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

Pointer non relational abstractions

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

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

Non relational pointer abstraction

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

We will see **several instances** of this kind of abstraction, and show how such abstraction lift into abstraction for sets of heaps

Pointer non relational abstraction: null pointers

The dereference of a null pointer will cause a crash

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

Null pointer analysis

Abstract domain for addresses:

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

$$\begin{array}{c} \top \\ | \\ \neq \texttt{NULL} \\ | \\ \bot \end{array}$$

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

Pointer non relational abstraction: dangling pointers

The dereferece of a null pointer will cause a crash

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

Null pointer analysis

Abstract domain for addresses:

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

$\top$

Not dangling

$\perp$

- 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** $\mathbb{A}^\sharp$ is fixed:
  $$\mathbb{A}^\sharp = \{\&\mathtt{x}, \&\mathtt{y}, \ldots, \&\mathtt{t}, a_0^\sharp, a_1^\sharp, \ldots, a_N^\sharp\}$$
  where $a_0^\sharp, \ldots, a_N^\sharp$ is a collection of $N+1$ fixed abstract addresses
- **Concrete addresses** are **abstracted into** $\mathbb{A}^\sharp$ by $\phi_{\mathbb{A}} : \mathbb{V}_{\mathrm{addr}} \to \mathbb{A}^\sharp \uplus \{\top\}$
- A pointer value is abstracted by the abstraction of the addresses it may point to, *i.e.*, $\mathbb{D}^\sharp_{\mathrm{ptr}} = \mathcal{P}(\mathbb{A}^\sharp)$
  and $\gamma_{\mathrm{ptr}}(a^\sharp) = \{a \in \mathbb{V}_{\mathrm{addr}} \mid \phi_{\mathbb{A}}(a) = a^\sharp\}$

Abstraction of pointer states

We consider all values are of pointer type, *i.e.*, heaps are of the form $\hbar : \mathbb{V}_{\mathrm{addr}} \to \mathbb{V}_{\mathrm{addr}}$.

Intuition:

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

Lifting a pointer abstraction to heap abstraction

We let $\mathbb{D}^\sharp_{\mathrm{mem}} = \mathbb{A}^\sharp \to \mathbb{D}^\sharp_{\mathrm{ptr}}$ and define

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

Examples of properties described by this abstraction:

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

Points-to sets computation example

Example code:

$$
\begin{array}{ll}
1: & \textbf{int}\; x, y;\\
2: & \textbf{int} * p;\\
3: & y = 9;\\
4: & p = \&x;\\
5: & *p = 0;\\
6: & \ldots
\end{array}
$$

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 : ...
```

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

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

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

Imprecise results

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

Weak-updates

We can formalize this imprecision a bit more:

Weak updates

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

Effect in pointer analysis, in the case of an **assignment**:

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

Pointer aliasing based on equivalence on access paths

Aliasing relation

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

Abstraction to infer pointer aliasing properties

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

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

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

Examples of aliasing abstractions:

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

Limitation of basic pointer analyses seen so far

Weak updates:

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

Unsatisfactory abstraction of unbounded memory:

- common assumption that $\mathbb{C}^\sharp$ **be finite**
- programs using **dynamic allocations** often perform **unbounded** numbers of **malloc** calls (*e.g.*, allocation of a list)

Unable to express well structural invariants:

- for instance, that a structure should be a **list**, a **tree**...
- **very indirect** abstraction in numeric / path equivalence abstration

A common solution:
shape abstraction

Outline

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

$$
\begin{array}{rcll}
\mathtt{e} & ::= & n \quad (n \in \mathbb{N}) & \text{constants} \\
 & \mid & \mathtt{l} & \text{l-value} \\
 & \mid & \mathtt{e}_0 + \mathtt{e}_1 & \text{binary operations} \\
 & \mid & \ldots & \\
\mathtt{P} & ::= & \mathtt{e}_0 = \mathtt{e}_1 \mid \mathtt{P}' \vee \mathtt{P}'' \mid \mathtt{P}' \wedge \mathtt{P}'' \ldots & \text{pure predicates}
\end{array}
$$

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

Separation logic: points-to predicates

The next slides introduce the main **separation logic formulas** $\mathtt{F} ::= \ldots$

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

Points-to predicate

- **Predicate:**

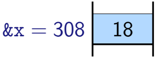
$$\mathtt{F} ::= \ldots \mid a \mapsto \mathtt{v} \qquad \text{where } a \text{ is an address and } \mathtt{v} \text{ is a value}$$

- **Concretization:**

$$(e, h) \in \gamma(\mathtt{l} \mapsto \mathtt{v}) \quad \text{if and only if} \quad h = [[\![\mathtt{l}]\!](e, h) \mapsto \mathtt{v}]$$

- **Example:**

$$\mathtt{F} = \&\mathtt{x} \mapsto 18$$



- We also note $\mathtt{l} \mapsto \mathtt{e}$, as an l-value $\mathtt{l}$ denotes an address

Separation logic: separating conjunction

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

$$h_0 \circledast h_1 : \begin{array}{lcl} \mathbf{dom}(h_0) \cup \mathbf{dom}(h_1) & \longrightarrow & \mathbb{V} \\ x \in \mathbf{dom}(h_0) & \longmapsto & h_0(x) \\ x \in \mathbf{dom}(h_1) & \longmapsto & h_1(x) \end{array}$$

Separating conjunction

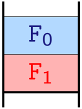
- **Predicate:**

$$\mathtt{F} ::= \ldots \mid \mathtt{F}_0 * \mathtt{F}_1$$

- **Concretization:**

$$\gamma(\mathtt{F}_0 * \mathtt{F}_1) = \{(e, h_0 \circledast h_1) \mid (e, h_0) \in \gamma(\mathtt{F}_0) \wedge (e, h_1) \in \gamma(\mathtt{F}_1)\}$$

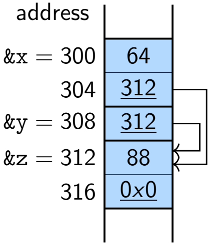
$\mathtt{F}_0 * \mathtt{F}_1$



An example

Concrete memory layout

(pointer values underlined)



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

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:**

$$\mathtt{F} ::= \ldots \mid \mathtt{F}_0 \wedge \mathtt{F}_1$$

- **Concretization:**

$$\gamma(\mathtt{F}_0 \wedge \mathtt{F}_1) = \gamma(\mathtt{F}_0) \cap \gamma(\mathtt{F}_1)$$

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

- $\texttt{\&a} \mapsto \texttt{\&b} \wedge \texttt{\&b} \mapsto \texttt{\&a}$
- $\texttt{\&a} \mapsto \texttt{\&b} * \texttt{\&b} \mapsto \texttt{\&a}$

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:**

$$\frac{(e, h_0) \in \gamma(\mathtt{F}_0) \qquad (e, h_1) \in \gamma(\mathtt{F}_1)}{(e, h_0 \circledast h_1) \in \gamma(\mathtt{F}_0 * \mathtt{F}_1)}$$

- **Non separating conjunction:**

$$\frac{(e, h) \in \gamma(\mathtt{F}_0) \qquad (e, h) \in \gamma(\mathtt{F}_1)}{(e, h) \in \gamma(\mathtt{F}_0 \wedge \mathtt{F}_1)}$$

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

Separation logic: empty store

Empty store

- **Predicate:**

$$\mathrm{F} ::= \ldots \mid \mathsf{emp}$$

- **Concretization:**

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

where [] denotes the empty store

- emp is the **neutral element for $*$** (monoid structure induced by $*$)
- by contrast the **neutral element for $\wedge$** is TRUE, with concretization:

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

Separation logic: other connectors

Disjunction:

- $\mathtt{F} ::= \ldots \mid \mathtt{F}_0 \vee \mathtt{F}_1$
- concretization:

$$\gamma(\mathtt{F}_0 \vee \mathtt{F}_1) = \gamma(\mathtt{F}_0) \cup \gamma(\mathtt{F}_1)$$

Spatial implication (*aka*, magic wand):

- $\mathtt{F} ::= \ldots \mid \mathtt{F}_0 \mathrel{-\!\!*} \mathtt{F}_1$
- concretization:

$$\gamma(\mathtt{F}_0 \mathrel{-\!\!*} \mathtt{F}_1) = \{(e, h) \mid \forall h_0 \in \mathbb{H},\ (e, h_0) \in \gamma(\mathtt{F}_0) \Longrightarrow (e, h \circledast h_0) \in \gamma(\mathtt{F}_1)\}$$

- 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

$$
\begin{array}{rcl}
\gamma(\mathsf{emp}) & = & \mathbb{E} \times \{[]\} \\
\gamma(\mathtt{TRUE}) & = & \mathbb{E} \times \mathbb{H} \\
\gamma(\mathtt{l} \mapsto \mathtt{v}) & = & \{(e, [[\![\mathtt{l}]\!](e, h) \mapsto \mathtt{v}]) \mid e \in \mathbb{E}\} \\
\gamma(\mathtt{F}_0 * \mathtt{F}_1) & = & \{(e, h_0 \circledast h_1) \mid (e, h_0) \in \gamma(\mathtt{F}_0) \wedge (e, h_1) \in \gamma(\mathtt{F}_1)\} \\
\gamma(\mathtt{F}_0 \wedge \mathtt{F}_1) & = & \gamma(\mathtt{F}_0) \cap \gamma(\mathtt{F}_1) \\
\gamma(\mathtt{F}_0 -\!\!* \mathtt{F}_1) & = & \{(e, h) \mid \forall h_0 \in \mathbb{H},\ (e, h_0) \in \gamma(\mathtt{F}_0) \Longrightarrow (e, h \circledast h_0) \in \gamma(\mathtt{F}_1)\}
\end{array}
$$

Concretization of pure formulas is standard

How does this help for program reasoning ?

Separation logic triple

Program proofs based on Hoare triples

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

$$\forall s, s' \in \mathbb{S},\ s \in \gamma(\mathtt{F}) \wedge s' \in [\![p]\!](s) \Longrightarrow s' \in \gamma(\mathtt{F}')$$

- Application: **formalize proofs of programs**

A few rules (straightforward proofs):

$$\frac{\mathtt{F}_0 \Longrightarrow \mathtt{F}_0' \qquad \{\mathtt{F}_0'\}\mathtt{b}\{\mathtt{F}_1'\} \qquad \mathtt{F}_1' \Longrightarrow \mathtt{F}_1}{\{\mathtt{F}_0\}\mathtt{b}\{\mathtt{F}_1\}}\ \text{consequence}$$

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

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

(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

$$\frac{\{\mathtt{F}_0\}\mathtt{b}\{\mathtt{F}_1\} \qquad \mathtt{freevar}(\mathtt{F}) \cap \mathtt{write}(\mathtt{b}) = \emptyset}{\{\mathtt{F}_0 * \mathtt{F}\}\mathtt{b}\{\mathtt{F}_1 * \mathtt{F}\}} \text{ frame}$$

- 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**:

$$
\begin{array}{l}
\quad \textbf{int}\ \mathtt{i};\\
\quad \textbf{int} * \mathtt{x};\\
\quad \textbf{int} * \mathtt{y};\\
\{\&\mathtt{i} \mapsto ? * \&\mathtt{x} \mapsto ? * \&\mathtt{y} \mapsto ?\}\\
\quad \mathtt{x} = \&\mathtt{i};\\
\{\&\mathtt{i} \mapsto ? * \&\mathtt{x} \mapsto \&\mathtt{i} * \&\mathtt{y} \mapsto ?\}\\
\quad \mathtt{y} = \&\mathtt{i};\\
\{\&\mathtt{i} \mapsto ? * \&\mathtt{x} \mapsto \&\mathtt{i} * \&\mathtt{y} \mapsto \&\mathtt{i}\}\\
\quad * \mathtt{x} = 42;\\
\{\&\mathtt{i} \mapsto 42 * \&\mathtt{x} \mapsto \&\mathtt{i} * \&\mathtt{y} \mapsto \&\mathtt{i}\}
\end{array}
$$

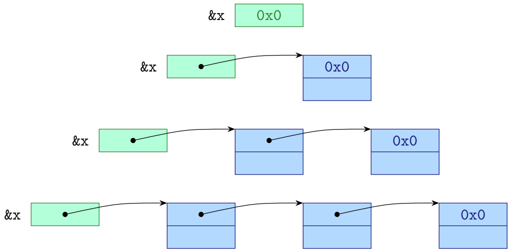
Many other program proofs done using separation logic
e.g., verification of the Deutsch-Shorr-Waite algorithm (biblio)

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

$$\begin{array}{rcll} \alpha \cdot \mathsf{list} & := & & \alpha = 0 \wedge \mathsf{emp} \\ & & \vee & \alpha \neq 0 \wedge \alpha \cdot \mathtt{next} \mapsto \delta * \alpha \cdot \mathtt{data} \mapsto \beta * \delta \cdot \mathsf{list} \end{array}$$

- Formula abstracting our set of structures:

$$\&\mathtt{x} \mapsto \alpha * \alpha \cdot \mathsf{list}$$

- **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

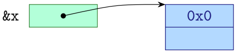
- Inductive predicates can be **unfolded**, by **unrolling their definitions**
  Syntactic unfolding is noted $\xrightarrow{\mathcal{U}}$
- A formula $\mathtt{F}$ with inductive predicates describes all stores described by all formulas $\mathtt{F}'$ such that $\mathtt{F} \xrightarrow{\mathcal{U}} \mathtt{F}'$

Example:

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

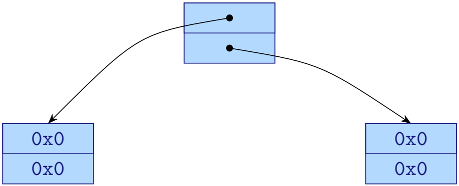
$$\begin{array}{ll} \multicolumn{2}{l}{\&\mathtt{x} \mapsto \alpha_0 * \alpha_0 \cdot \mathsf{list}} \\ \xrightarrow{\mathcal{U}} & \&\mathtt{x} \mapsto \alpha_0 * \alpha_0 \cdot \mathtt{next} \mapsto \alpha_1 * \alpha_0 \cdot \mathtt{data} \mapsto \beta_1 * \alpha_1 \cdot \mathsf{list} \\ \xrightarrow{\mathcal{U}} & \&\mathtt{x} \mapsto \alpha_0 * \alpha_0 \cdot \mathtt{next} \mapsto \alpha_1 * \alpha_0 \cdot \mathtt{data} \mapsto \beta_1 * \mathsf{emp} \wedge \alpha_1 = \mathbf{0x0} \end{array}$$

- We get the concrete state below:



Example: tree

- **Example:**



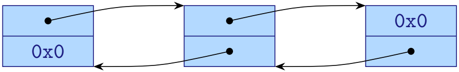
Inductive definition

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

$$\begin{array}{rcl} \alpha \cdot \mathsf{tree} & := & \alpha = 0 \wedge \mathsf{emp} \\ & \vee & \alpha \neq 0 \wedge \alpha \cdot \mathtt{left} \mapsto \beta * \alpha \cdot \mathtt{right} \mapsto \delta \\ & & * \, \beta \cdot \mathsf{tree} * \delta \cdot \mathsf{tree} \end{array}$$

Example: doubly linked list

- **Example:**



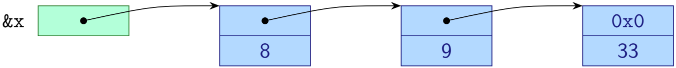
Inductive definition

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

$$\begin{array}{rcl}\alpha\cdot\mathsf{dll}(\delta) & := & \quad\ \alpha = 0 \wedge \mathsf{emp} \\ & & \vee\quad \alpha \neq 0 \wedge \alpha\cdot\mathtt{next} \mapsto \beta * \alpha\cdot\mathtt{prev} \mapsto \delta \\ & & \quad\ * \beta\cdot\mathsf{dll}(\alpha)\end{array}$$

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

$$
\begin{array}{rcl}
\alpha \cdot \mathsf{lsort}_{\mathrm{aux}}(n) & := & \quad \alpha = 0 \wedge \mathsf{emp} \\
 & \vee & \quad \alpha \neq 0 \wedge n \leq \beta \wedge \alpha \cdot \mathtt{next} \mapsto \delta \\
 & & \quad * \, \alpha \cdot \mathtt{data} \mapsto \beta * \delta \cdot \mathsf{lsort}_{\mathrm{aux}}(\beta)
\end{array}
$$

$$
\alpha \cdot \mathsf{lsort}() \quad := \quad \alpha \cdot \mathsf{lsort}_{\mathrm{aux}}(-\infty)
$$

Outline

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:

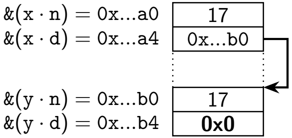
- unstructured formulas can be represented as **ASTs**,
  but this representation is **not easy to manipulate efficiently**
- intuition over memory states typically involves **graphs**

Analysis algorithms:

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

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

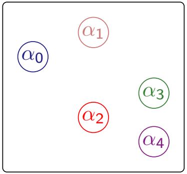
- **Concrete memory states**
  - very **low level** description
    numeric offsets / field names
  - pointers, numeric values:
    raw sequences of bits

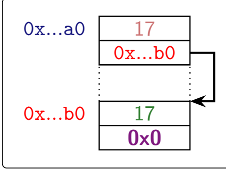


Basic abstraction: structures and their contents (1/2)

- **Concrete memory states**

- **Abstraction of values into symbolic variables** (nodes)



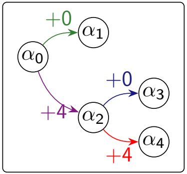


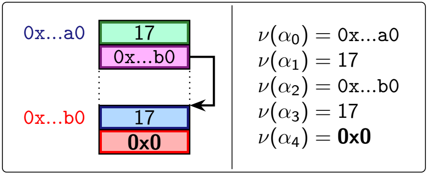
$$\nu(\alpha_0) = \texttt{0x...a0}$$
$$\nu(\alpha_1) = 17$$
$$\nu(\alpha_2) = \texttt{0x...b0}$$
$$\nu(\alpha_3) = 17$$
$$\nu(\alpha_4) = \mathbf{0x0}$$

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

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

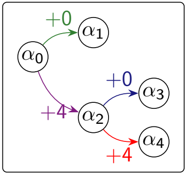
- **Concrete memory states**
- **Abstraction of values into symbolic variables** / nodes
- **Abstraction of regions into points-to edges**

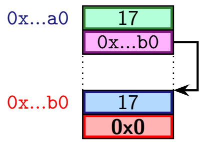




Basic abstraction: structures and their contents (1/2)

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





$$\nu(\alpha_0) = \texttt{0x...a0}$$
$$\nu(\alpha_1) = 17$$
$$\nu(\alpha_2) = \texttt{0x...b0}$$
$$\nu(\alpha_3) = 17$$
$$\nu(\alpha_4) = \mathbf{0x0}$$

- **Shape graph concretization**

$$\gamma_{\mathsf{sh}}(G) = \{(\hbar, \nu) \mid \ldots\}$$

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: $\mathbb{V}^\sharp = \{\alpha, \beta, \delta, \ldots\}$

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): $\mathsf{Val} = \mathbb{V}^\sharp \longrightarrow \mathbb{V}$

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

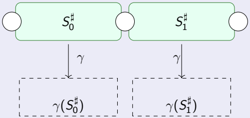
Structure of shape graphs

Distinct edges describe separate regions

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

Separating conjunction

$$\gamma_{\mathsf{sh}}(S_0^\sharp * S_1^\sharp) \quad = \quad \{(\hbar_0 \circledast \hbar_1, \nu) \mid (\hbar_0, \nu) \in \gamma_{\mathsf{sh}}(S_0^\sharp) \wedge (\hbar_1, \nu) \in \gamma_{\mathsf{sh}}(S_1^\sharp)\}$$



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

$$\gamma_{\mathsf{sh}}(\mathsf{emp}) = \{(\emptyset, \nu) \mid \nu : \mathbb{V}^\sharp \to \mathbb{V}\}$$

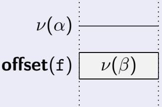
Abstraction of contiguous regions

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**

Graph edge	Separation logic formula	Concrete view
$\alpha \xrightarrow{\mathtt{f}} \beta$	$\alpha \cdot \mathtt{f} \mapsto \beta$	

- **Concretization**:

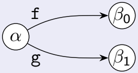
$$\gamma_{\mathsf{sh}}(\alpha \cdot \mathtt{f} \mapsto \beta) = \{([\nu(\alpha) + \mathbf{offset}(\mathtt{f}) \mapsto \nu(\beta)], \nu) \mid \nu : \{\alpha, \beta, \ldots\} \to \mathbb{N}\}$$

Abstraction of contiguous regions

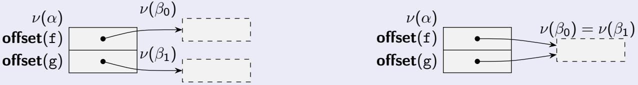
Contiguous regions are described by adjacent points-to edges

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:

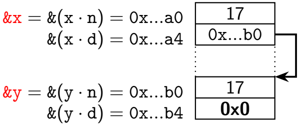


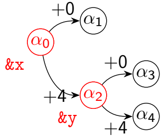
In other words, in a field splitting model, separation:

- asserts addresses are distinct
- says nothing about contents

Abstraction of the environment

Environments bind variables to their (concrete / abstract) address





$$\nu: \begin{array}{ccc} \alpha_0 & \mapsto & \texttt{0x...a0} \\ \alpha_2 & \mapsto & \texttt{0x...b0} \\ \ldots & \mapsto & \ldots \end{array}$$

$$e^\sharp: \begin{array}{ll} \texttt{x} \mapsto \alpha_0 & (\overset{\nu}{\mapsto} \texttt{0x...a0}) \\ \texttt{y} \mapsto \alpha_2 & (\overset{\nu}{\mapsto} \texttt{0x...b0}) \end{array}$$

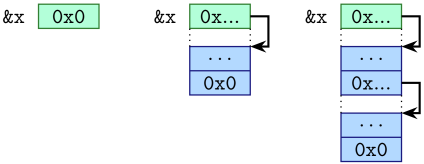
Abstract environments

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

$$\gamma_{\mathsf{mem}}(e^\sharp, S^\sharp) = \{(e, \hbar, \nu) \mid (\hbar, \nu) \in \gamma_{\mathsf{sh}}(S^\sharp) \wedge e = \nu \circ e^\sharp\}$$

Basic abstraction: summarization

Set of all lists of any length:

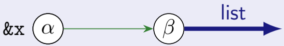


Well-founded list inductive def.

$$\begin{array}{rl} \alpha \cdot \mathsf{list} := & \\ & (\mathsf{emp} \wedge \alpha = \mathbf{0x0}) \\ \vee & (\alpha \cdot \mathtt{d} \mapsto \beta_0 * \alpha \cdot \mathtt{n} \mapsto \beta_1 \\ & \quad * \beta_1 \cdot \mathsf{list} \wedge \alpha \neq \mathbf{0x0}) \end{array}$$

well-founded predicate

Inductive summary predicates



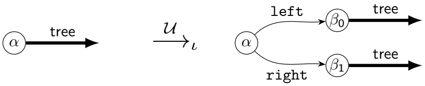
Concretization based on **unfolding** and **least-fixpoint**:

- $\xrightarrow{\mathcal{U}}$ replaces **an** $\alpha \cdot \mathsf{list}$ **predicate** with **one of its premises**
- $\gamma(S^\sharp, \mathtt{F}) = \bigcup \{\gamma(S^\sharp_u, \mathtt{F}_u) \mid (S^\sharp, \mathtt{F}) \xrightarrow{\mathcal{U}} (S^\sharp_u, \mathtt{F}_u)\}$

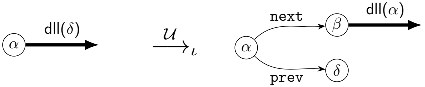
Inductive structures: a few instances

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

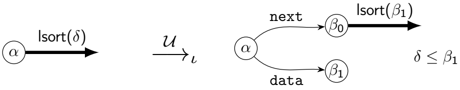
- **More complex shapes**: **trees**



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

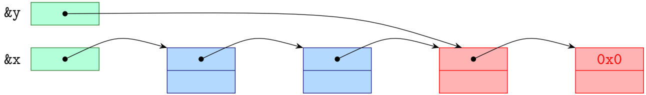


- **Relations between pointers and numerical**: **sorted lists**



Inductive segments

A frequent pattern:



A first attempt:

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

However, the following **does not hold**

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

Why ? **violation of separation!**

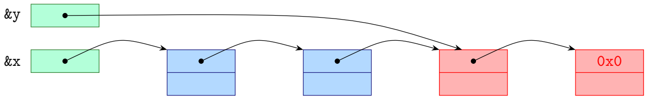
A second attempt:

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

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

Inductive segments

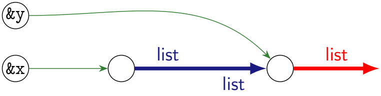
A frequent pattern:



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

$$\begin{array}{rcl}\alpha \cdot \mathsf{list\_endp}(\pi) & ::= & (\mathsf{emp}, \alpha = \pi) \\ & | & (\alpha \cdot \mathtt{next} \mapsto \beta_0 * \alpha \cdot \mathtt{data} \mapsto \beta_1 \\ & & \quad * \beta_0 \cdot \mathsf{list\_endp}(\pi), \alpha \neq 0)\end{array}$$

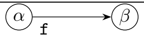
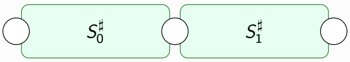
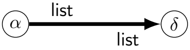
This definition **straightforwardly derives** from list
Thus, we make **segments** part of the **fundamental predicates of the domain**



Multi-segments: possible, but harder for analysis

Shape graphs and separation logic

Semantic preserving translation Π of graphs into separation logic formulas:

Graph $S^\sharp \in \mathbb{D}^\sharp_{\mathrm{sh}}$	Translated formula $\Pi(S^\sharp)$
	$\alpha \cdot \mathtt{f} \mapsto \beta$
	$\Pi(S_0^\sharp) * \Pi(S_1^\sharp)$
	$\alpha \cdot \mathsf{list}$
	$\alpha \cdot \mathsf{list\_endp}(\delta)$
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

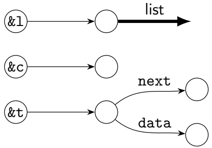
Outline

Static analysis overview

A list insertion function:

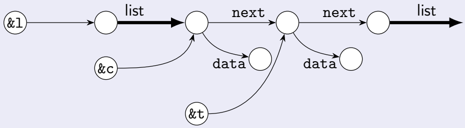
```
list * l assumed to point to a list
list * t assumed to point to a list element
list * c = l;
while(c != NULL && c -> next != NULL && (...)){
    c = c -> next;
}
t -> next = c -> next;
c -> next = t;
```

- 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:** $\mathrm{x} \rightarrow f = \mathrm{y} \rightarrow g$ or $\mathrm{x} \rightarrow f = e_{\mathrm{arith}}$
- **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** $[\![\mathtt{b}]\!] : \mathbb{S} \longrightarrow \mathcal{P}(\mathbb{S})$
- **Abstract post-condition** $[\![\mathtt{b}]\!]^\sharp(\mathsf{S})$, computed by the analysis:

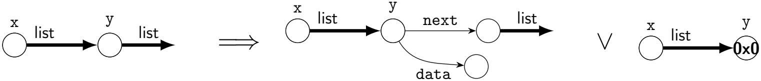
$$s \in \gamma(\mathsf{S}) \Longrightarrow [\![\mathtt{b}]\!](s) \subseteq \gamma([\![\mathtt{b}]\!]^\sharp(\mathsf{S}))$$

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

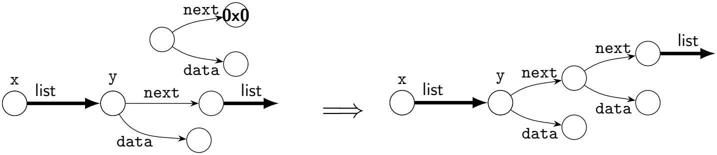
$$\begin{array}{rcl}
[\![\mathtt{b}_0; \mathtt{b}_1]\!]^\sharp(\mathsf{S}) & = & [\![\mathtt{b}_1]\!]^\sharp \circ [\![\mathtt{b}_0]\!]^\sharp(\mathsf{S}) \\
[\![\mathtt{l} = \mathtt{e}]\!]^\sharp(\mathsf{S}) & = & \mathit{assign}(\mathtt{l}, \mathtt{e}, \mathsf{S}) \\
[\![\mathtt{l} = \mathbf{malloc}(n)]\!]^\sharp(\mathsf{S}) & = & \mathit{alloc}(\mathtt{l}, n, \mathsf{S}) \\
[\![\mathbf{free}(\mathtt{l})]\!]^\sharp(\mathsf{S}) & = & \mathit{free}(\mathtt{l}, n, \mathsf{S}) \\
[\![\mathbf{if}(\mathtt{e})\ \mathtt{b}_t\ \mathbf{else}\ \mathtt{b}_f]\!]^\sharp(\mathsf{S}) & = & \left\{ \begin{array}{l} \mathit{join}([\![\mathtt{b}_t]\!]^\sharp(\mathit{test}(\mathtt{e}, \mathsf{S})), \\ \qquad\qquad [\![\mathtt{b}_f]\!]^\sharp(\mathit{test}(\mathtt{e} = \mathrm{false}, \mathsf{S}))) \end{array} \right. \\
[\![\mathbf{while}(\mathtt{e})\mathtt{b}]\!]^\sharp(\mathsf{S}) & = & \mathit{test}(\mathtt{e} = \mathrm{false}, \mathbf{lfp}^\sharp{}_{\mathsf{S}} F^\sharp) \\
\multicolumn{3}{c}{\text{where, } F^\sharp : \mathsf{S}_0 \mapsto [\![\mathtt{b}]\!]^\sharp(\mathit{test}(\mathtt{e}, \mathsf{S}_0))}
\end{array}$$

The algorithms underlying the transfer functions

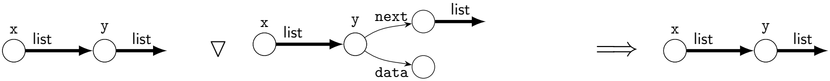
- **Unfolding**: **cases analysis on summaries**



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



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



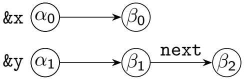
Outline

Analysis of an assignment in the graph domain

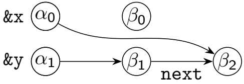
Steps for analyzing `x = y -> next` (local reasoning)

1. Evaluate **l-value** `x` into **points-to edge** $\alpha \mapsto \beta$
2. Evaluate **r-value** `y -> next` into **node** β'
3. Replace points-to edge $\alpha \mapsto \beta$ with **points-to edge** $\alpha \mapsto \beta'$

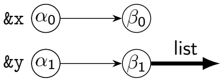
With pre-condition:



- 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**

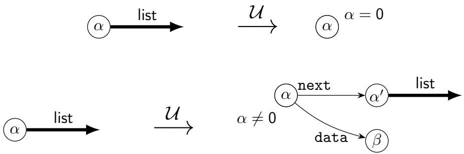
- Abstract state **too abstract**
- We need to **refine it**

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

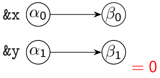
$$\gamma_{\mathsf{sh}}(S^\sharp) \subseteq \bigcup \{\gamma_{\mathsf{sh}}(S^\sharp_0) \mid S^\sharp \xrightarrow{\mathcal{U}} S^\sharp_0\}$$

Analysis of an assignment, with unfolding

Principle

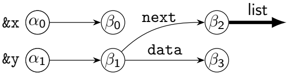
- We have $\gamma_{\mathsf{sh}}(\alpha \cdot \iota) = \bigcup \{ \gamma_{\mathsf{sh}}(S^\sharp) \mid \alpha \cdot \iota \xrightarrow{\mathcal{U}} S^\sharp \}$
- Replace $\alpha \cdot \iota$ with a finite number of disjuncts and continue

Disjunct 1:

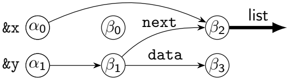


- Step 1 produces $\alpha_0 \mapsto \beta_0$
- **Step 2 fails: Null pointer !**
- In a **correct** program, would be ruled out by a **condition** $\mathtt{y} \neq 0$ *i.e.*, $\beta_1 \neq 0$ in $\mathbb{D}^\sharp_{\mathsf{num}}$

Disjunct 2:

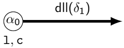


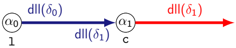
- Step 1 produces $\alpha_0 \mapsto \beta_0$
- Step 2 produces β_2
- **End result**:



Unfolding and degenerated cases

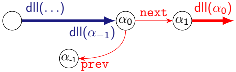
```
assume(l points to a dll)
c = l;
① while(c ≠ NULL && condition)
      c = c -> next;
② if(c ≠ 0 && c -> prev ≠ 0)
      c = c -> prev → prev;
```

- at ①: 

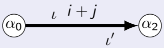
- at ②: 

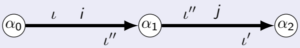
$\Rightarrow$ **non trivial unfolding**

- **Materialization of** c -> prev:

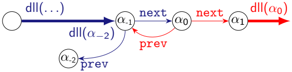


Segment splitting lemma: basis for segment unfolding



describes the same set of stores as


- **Materialization of** c -> prev -> prev:



- **Implementation issue**: discover **which inductive edge** to unfold **very hard !**

Outline

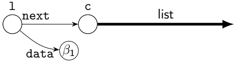
Need for a folding operation

Back to the **list traversal** example:

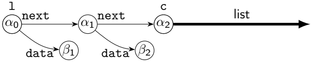
First iterates in the loop:

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

- at **iteration 1**:

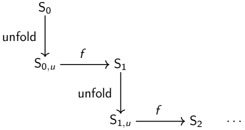


- at **iteration 2**:




```
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:**

$$\left\{ \begin{array}{lcl} \gamma(X^\sharp) & \subseteq & \gamma(X^\sharp \nabla Y^\sharp) \\ \gamma(Y^\sharp) & \subseteq & \gamma(X^\sharp \nabla Y^\sharp) \end{array} \right.$$

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

$$\left\{ \begin{array}{rcl} Y^\sharp_0 & = & X^\sharp_0 \\ \forall n \in \mathbb{N},\ Y^\sharp_{n+1} & = & Y^\sharp_n \nabla X^\sharp_{n+1} \end{array} \right.$$

Canonicalization

Upper closure operator

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

Canonicalization

- **Disjunctive completion:** $\mathbb{D}^\sharp_\vee$ = finite disjunctions over $\mathbb{D}^\sharp$
- **Canonicalization operator** $\rho_\vee$ defined by $\rho_\vee : \mathbb{D}^\sharp_\vee \longrightarrow \mathbb{D}^\sharp_{\text{can}\vee}$ and $\rho_\vee(X^\sharp) = \{\rho(x^\sharp) \mid x^\sharp \in X^\sharp\}$ where ρ is an uco and $\mathbb{D}^\sharp_{\text{can}}$ is finite

- 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

$$\forall(\hbar, \nu) \in \gamma_{\mathsf{sh}}(S_0^\sharp),\ \exists \nu' \in \mathsf{Val},\ (\hbar, \nu') \in \gamma_{\mathsf{sh}}(S_1^\sharp)$$

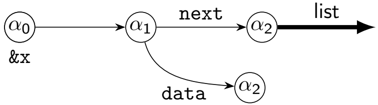
We then note $S_0^\sharp \preccurlyeq S_1^\sharp$

Applications:

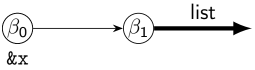
- **inclusion test** (comparison) inputs $S_0^\sharp, S_1^\sharp$; if returns true $S_0^\sharp \preccurlyeq S_1^\sharp$
- **canonicalization** (unary weakening) inputs $S_0^\sharp$ and returns $\rho(S_0^\sharp)$ such that $S_0^\sharp \preccurlyeq \rho(S_0^\sharp)$
- **widening** / **join** (binary weakening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

Weakening: example

We consider $S_0^\sharp$ defined by:



and $S_1^\sharp$ defined by:



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

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

- 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 $S_0^\sharp \preccurlyeq S_{0,\mathrm{weak}}^\sharp$ and $S_1^\sharp \preccurlyeq S_{1,\mathrm{weak}}^\sharp$ then:

Separating conjunction rule ($\preccurlyeq_*$)

Let us assume that

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

then:

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

Local weakening: unfolding rule, identity rule

Weakening unfolded region ($\preccurlyeq_{\mathcal{U}}$)

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

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

- 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 ($\preccurlyeq_{\mathbf{Id}}$)

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

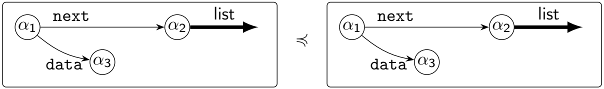
- the proof is trivial:

$$\gamma_{\mathrm{sh}}(S^\sharp) \subseteq \gamma_{\mathrm{sh}}(S^\sharp)$$

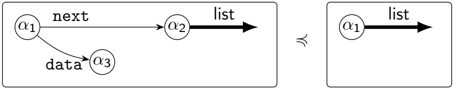
- on itself, this principle is not very useful, but it can be applied locally, and combined with ($\preccurlyeq_{\mathcal{U}}$) on graph regions that are not equal

Local weakening: example

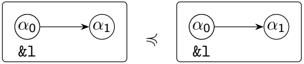
By **rule** ($\preccurlyeq_{\mathbf{Id}}$):



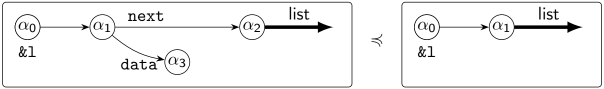
Thus, by **rule** ($\preccurlyeq_{\mathcal{U}}$):



Additionally, by **rule** ($\preccurlyeq_{\mathbf{Id}}$):



Thus, by **rule** ($\preccurlyeq_{*}$):



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:**

$$\begin{array}{rcrcl} \forall i,\ \Psi(\beta_i) = \alpha_i & \Longrightarrow & \alpha_0 \cdot \mathtt{f} \mapsto \alpha_1 & \sqsubseteq^\sharp{}_\Psi & \beta_0 \cdot \mathtt{f} \mapsto \beta_1 \\ \Psi(\beta) = \alpha & \Longrightarrow & \alpha \cdot \mathsf{list} & \sqsubseteq^\sharp{}_\Psi & \beta \cdot \mathsf{list} \\ \forall i,\ \Psi(\beta_i) = \alpha_i & \Longrightarrow & \alpha_0 \cdot \mathsf{list\_endp}(\alpha_1) & \sqsubseteq^\sharp{}_\Psi & \beta_0 \cdot \mathsf{list\_endp}(\beta_1) \end{array}$$

- **Rules on inductives:**

$$\begin{array}{rcrcl} \forall i,\ \Psi(\beta_i) = \alpha & \Longrightarrow & \mathsf{emp} & \sqsubseteq^\sharp{}_\Psi & \beta_0 \cdot \mathsf{list\_endp}(\beta_1) \\ S_0^\sharp \sqsubseteq^\sharp{}_\Psi S_1^\sharp \wedge \beta \cdot \iota \xrightarrow{\mathcal{U}} S_1^\sharp & \Longrightarrow & S_0^\sharp & \sqsubseteq^\sharp{}_\Psi & \beta \cdot \iota \end{array}$$

$$\text{if } \beta_1 \text{ fresh },\ \Psi' = \Psi[\beta_1 \mapsto \alpha_1] \text{ and } \Psi(\beta_0) = \alpha_0 \text{ then,}$$

$$S_0^\sharp \sqsubseteq^\sharp{}_{\Psi'} \beta_1 \cdot \mathsf{list} \Longrightarrow \alpha_0 \cdot \mathsf{list\_endp}(\alpha_1) * S_0^\sharp \sqsubseteq^\sharp{}_\Psi \beta_0 \cdot \iota$$

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 $\mathtt{x}$ such that $e_0^\sharp(\mathtt{x}) = \alpha \wedge e_1^\sharp(\mathtt{x}) = \beta$
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

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

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 $\forall i \in \{0, 1\},\ \forall s \in \{\mathrm{lft}, \mathrm{rgh}\},\ S^\sharp_{i,s} \preccurlyeq S^\sharp_s$,

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

$$\Psi : \mathbb{V}^\sharp(S^\sharp_{\mathrm{lft}}) \times \mathbb{V}^\sharp(S^\sharp_{\mathrm{rgh}}) \longrightarrow \mathbb{V}^\sharp(S^\sharp)$$

- 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** $(\preccurlyeq_{\mathbf{Id}})$:

- If $S^\sharp_{\mathrm{lft}} = \alpha_0 \cdot \mathtt{f} \mapsto \alpha_1$
  and $S^\sharp_{\mathrm{lft}} = \beta_0 \cdot \mathtt{f} \mapsto \beta_1$
  and $\Psi(\alpha_0, \beta_0) = \delta_0$, $\Psi(\alpha_1, \beta_1) = \delta_1$, then:

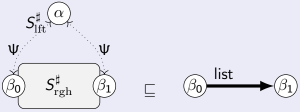
$$S^\sharp_{\mathrm{lft}} \nabla S^\sharp_{\mathrm{rgh}} = \delta_0 \cdot \mathtt{f} \mapsto \delta_1$$

- If $S^\sharp_{\mathrm{lft}} = \alpha_0 \cdot \mathsf{list}$
  and $S^\sharp_{\mathrm{lft}} = \beta_0 \cdot \mathsf{list}_1$
  and $\Psi(\alpha_0, \beta_0) = \delta_0$, then:

$$S^\sharp_{\mathrm{lft}} \nabla S^\sharp_{\mathrm{rgh}} = \delta_0 \cdot \mathsf{list}$$

Over-approximation of union: segment introduction rule

Rule

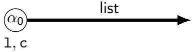
if 

then

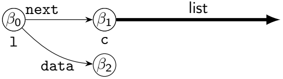
$$\left\{ \begin{array}{l} S^\sharp_{\rm lft} \mathbin{\triangledown} S^\sharp_{\rm rgh} \ = \ \delta_0 \xrightarrow{\text{list}} \delta_1 \\ (\alpha, \beta_0) \stackrel{\Psi}{\longleftrightarrow} \delta_0 \\ (\alpha, \beta_1) \stackrel{\Psi}{\longleftrightarrow} \delta_1 \end{array} \right.$$

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

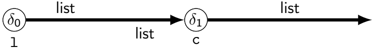
- **before iteration 0:**



- **end of iteration 0:**



- **join, before iteration 1:**

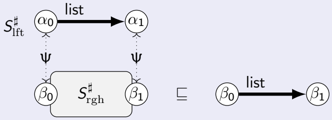


$$\left\{ \begin{array}{lcl} \Psi(\alpha_0, \beta_0) & = & \delta_0 \\ \Psi(\alpha_0, \beta_1) & = & \delta_1 \end{array} \right.$$

Over-approximation of union: segment extension rule

Rule

if



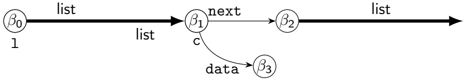
then

$$\left\{ \begin{array}{l} S^\sharp_{\rm lft} \triangledown S^\sharp_{\rm rgh} \; = \; \delta_0 \xrightarrow{\text{list}} \delta_1 \\ (\alpha_0, \beta_0) \stackrel{\Psi}{\longleftrightarrow} \delta_0 \\ (\alpha_1, \beta_1) \stackrel{\Psi}{\longleftrightarrow} \delta_1 \end{array} \right.$$

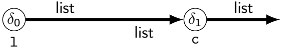
Application to list traversal, at the end of iteration 1:

- **previous invariant before iteration 1:**

- **end of iteration 1:**



- **join, before iteration 1:**



$$\left\{ \begin{array}{lcl} \Psi(\alpha_0, \beta_0) & = & \delta_0 \\ \Psi(\alpha_1, \beta_2) & = & \delta_1 \end{array} \right.$$

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 $\Psi(\alpha, \beta) = e(\mathrm{x}) = \delta$ (where δ is a fresh node) if and only if $e_0^\sharp(\mathrm{x}) = \alpha \wedge e_1^\sharp(\mathrm{x}) = \beta$
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

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 $\mathtt{b}$:

$$\begin{array}{rl} (\mathcal{P}(\mathbb{M}), \subseteq) & \text{is a complete lattice, hence a CPO} \\ F : \mathcal{P}(\mathbb{M}) \rightarrow \mathcal{P}(\mathbb{M}) & \text{is the concrete semantic ("post") function of } \mathtt{b} \end{array}$$

  thus, the concrete semantics writes down as $[\![\mathtt{b}]\!] = \mathsf{lfp}_{\emptyset} F$

- **Assumptions in the abstract level:**

$$\begin{array}{rl} \mathbb{M}^{\sharp} & \text{set of abstract elements, no order a priori} \\ & m^{\sharp} ::= (e^{\sharp}, S^{\sharp}) \\ \gamma_{\mathrm{mem}} : \mathbb{M}^{\sharp} \rightarrow \mathcal{P}(\mathbb{M}) & \text{concretization} \\ F^{\sharp} : \mathbb{M}^{\sharp} \rightarrow \mathbb{M}^{\sharp} & \text{sound abstract semantic function} \\ & \textit{i.e.}\text{, such that } F \circ \gamma_{\mathrm{mem}} \subseteq \gamma_{\mathrm{mem}} \circ F^{\sharp} \\ \triangledown : \mathbb{M}^{\sharp} \times \mathbb{M}^{\sharp} \rightarrow \mathbb{M}^{\sharp} & \text{widening operator, terminates, and such that} \\ & \gamma_{\mathrm{mem}}(m_0^{\sharp}) \cup \gamma_{\mathrm{mem}}(m_1^{\sharp}) \subseteq \gamma_{\mathrm{mem}}(m_0^{\sharp} \triangledown m_1^{\sharp}) \end{array}$$

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:

$$\begin{cases} m^\sharp_0 & = & \bot \\ m^\sharp_{n+1} & = & m^\sharp_n \triangledown F^\sharp(m^\sharp_n) \end{cases}$$

Soundness proof:

- by induction over n, $\bigcup_{k \leq n} F^k(\emptyset) \subseteq \gamma_{\mathrm{mem}}(m^\sharp_n)$
- by the property of widening, the abstract sequence converges at a rank N: $\forall k \geq N,\ m^\sharp_k = m^\sharp_N$, thus

$$\mathsf{lfp}_\emptyset F = \bigcup_k F^k(\emptyset) \subseteq \gamma_{\mathrm{mem}}(m^\sharp_N)$$

Discussion on the abstract ordering

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

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

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

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

$$\mathsf{is\_le}(m_0^\sharp, m_1^\sharp) = \mathsf{true} \quad \Longrightarrow \quad \gamma_{\mathrm{mem}}(m_0^\sharp) \subseteq \gamma_{\mathrm{mem}}(m_1^\sharp)$$

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

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

Internships on memory abstraction

Several topics are possible, for instance:

Summarization based on universal quantification:

- memory abstractions use **summaries**
  today, we consider inductive linked structures; we will also see arrays...
- **another form of summarization** based on an **unbounded set** E

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

  requires the definition of fold / unfold, analysis operations...
- towards a parametric abstract domain:
  - generic dictionary abstraction
  - arrays (generalization of existing)
  - union finds and DAGs