Shape analysis based on separation logic

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

Xavier Rival

INRIA

Nov, 23th. 2012

Overview of the lecture

How to reason about memory properties (bis)

- Last lecture:
  - a broad **overview** of problems and techniques
  - concrete and abstract memory models
  - an introduction to shape analysis: TVLA
- Today:
  - a logic to describe properties of memory states
  - abstract domain
  - static analysis algorithms
  - combination with numerical domains
  - widening operators...

Outline

Our model

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

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

- h is actually only a partial function
- **Memory states**:

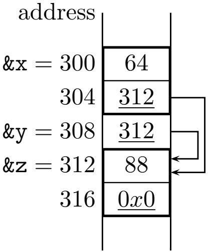
$$\mathbb{M} = \mathbb{E} \times \mathbb{H}$$

Example of a concrete memory state

- 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)



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

Weak updates

Notion of weak update

Udpate

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

We remarked many cases of weak-updates:

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

Best result of the analysis ?

- range for x
- range for y

- Weak updates are a curse for the static analysis
- Huge loss in precision incurred by weak updates

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**
- Use a **special separating conjunction operator** $*$
- **Local reasoning**:
  powerful principle, which allows to consider only part of the program memory
- Separation logic has been used in many contexts, including manual verification, static analysis, etc...

Separation logic

- **Logic made of a set of formulas**
- inference rules...

Pure formulas

- Set of **pure formulas**, similar to first order logics

$$
\begin{array}{rcll}
e & ::= & n & (n \in \mathbb{N}) \\
 & | & l & \text{l-value} \\
 & | & e' + e'' & \text{binary} \\
 & | & \ldots & \\
P & ::= & e = e' \mid P' \vee P'' \mid P' \wedge P'' \ldots &
\end{array}
$$

- Denote numerical properties among the values

Heap formulas (syntax on the next slide)

- Set of formulas to describe concrete heaps
- Concretization relation of the form $(e, h) \in \gamma(F)$

Heap formulas

Main connectors

Each formula describes a **heap region**

$$
\begin{array}{rcll}
F & ::= & \mathbf{emp} & \text{empty region} \\
 & | & \mathbf{true} & \text{empty region} \\
 & | & l \mapsto v & \text{memory cell} \\
 & | & F' * F'' & \text{separating conjunction} \\
 & | & F' \wedge F'' & \text{classical conjunction} \\
 & | & \ldots & \text{many other connectors (see biblio)}
\end{array}
$$

Denotations: the usual stuff...

- $\gamma(\mathbf{emp}) = \emptyset$; $\quad \gamma(\mathbf{true}) = \mathbb{M}$
- $(e, h) \in \gamma(l \mapsto v)$ if and only if $h([\![l]\!](e, h)) = v$
- $(e, h) \in \gamma(F' \wedge F'')$ if and only if $(e, h) \in \gamma(F')$ and $(e, h) \in \gamma(F'')$

Separating conjunction: next slide...

The separating conjunction

Merge of concrete stores

Let $h_0, h_1 \in (\mathbb{V}_{\rm addr} \rightarrow \mathbb{V})$, such that $\mathbf{dom}(h_0) \cap \mathbf{dom}(h_1) = \emptyset$.
Then, we let $h_0 \circledast h_1$ be defined by:

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

Concretization of separating conjunction

- Logical formulas denote sets of heaps; concretization γ
- **Binary logical connector on formulas** $*$ defined by:

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

- Exercise: concretization of $a \mapsto \&b \wedge b \mapsto \&a$? of $a \mapsto \&b * b \mapsto \&a$?

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 $a = b$

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

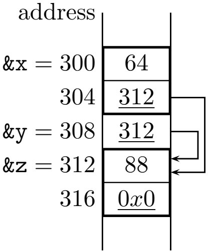
- two **separate** sub-heaps respectively satisfy $a \mapsto \&b$ and $b \mapsto \&a$
- thus $a \neq b$

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

An example

Concrete memory layout

(pointer values underlined)



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

A formula that abstracts away the addresses:

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

Separating and non separating conjunction

- There are two conjunction operators $\wedge$ and $*$
- How to relate them ?

Separating conjunction vs normal conjunction

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

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

Programs: syntax and semantics

Basic language

- **L-values**: $l \ ::= \ \mathtt{x}\ (\mathtt{x} \in \mathbb{X})\ \ldots \mid \star e \mid l \cdot \mathtt{f}$
- **Expressions**: $e \ ::= \ l \mid c\ (c \in \mathbb{V}) \mid e \oplus e \mid \& l \mid \mathbf{malloc}(n)$
- **Statements**:
  $s \ ::= \ l := e \mid \mathbf{if}(e)\,\{s\}\,\mathbf{else}\,\{s\} \mid \mathbf{while}(e)\,\{s\} \mid s; s \mid \mathbf{free}(l);$

Semantics

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

Separating logic triple

Program proofs based on triples

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

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

Hoare triples

- Application: formalize proofs of programs

A few rules (straightforward proofs):

$$\frac{F_0 \Longrightarrow F_0' \qquad \{F_0'\}p\{F_1'\} \qquad F_1' \Longrightarrow F_0'}{\{F_0\}p\{F_1\}}\ \textit{consequence}$$

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

$$\frac{}{\{x \mapsto ? * F\}x := e\{x \mapsto e * F\}}\ \textit{mutation} - 2$$

The frame rule

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

Theorem: the frame rule

$$\frac{\{F_0\}s\{F_1\}}{\{F_0 * F\}s\{F_1 * F\}}\ \textit{frame}$$

- Proof by induction on the rules
  (see biblio for a more complete set of rules)
- Rules are proved by case analysis on the program syntax

We can reason locally about programs

Application of the frame rule

Let us consider the program below:

$$
\begin{array}{ll}
\textbf{int}\,\texttt{i}; & \\
\textbf{int} \star \texttt{x}; & \\
\textbf{int} \star \texttt{y}; & \{\texttt{i} \mapsto ? * \texttt{x} \mapsto ? * \texttt{y} \mapsto ?\} \\
\texttt{x} = \texttt{\&i}; & \{\texttt{i} \mapsto ? * \texttt{x} \mapsto \texttt{\&i} * \texttt{y} \mapsto ?\} \\
\texttt{y} = \texttt{\&i}; & \{\texttt{i} \mapsto ? * \texttt{x} \mapsto \texttt{\&i} * \texttt{y} \mapsto \texttt{\&i}\} \\
\star\texttt{x} = 42; & \{\texttt{i} \mapsto 42 * \texttt{x} \mapsto \texttt{\&i} * \texttt{y} \mapsto \texttt{\&i}\}
\end{array}
$$

- Each step impacts a disjoint memory region
- This case is easy
  See biblio for more complex applications
  (verification of the Deutsch-Shorr-Waite algorithm)

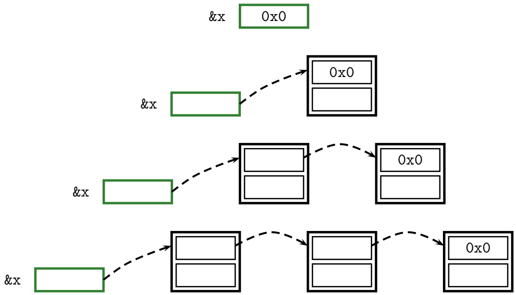
Summarization and inductive definitions

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 **one formula** i.e., no infinite disjunction ?

Inductive definitions in separation logic

List definition

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

- Formula abstracting our set of structures:

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

- **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

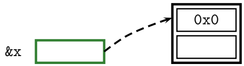
- Inductive predicates can be **unfolded**, by unrolling their definitions
  Syntactic unfolding is noted $\xrightarrow{\mathcal{U}}$
- A formula F with inductive predicates describes all stores described by all formulas F' such that $F \xrightarrow{\mathcal{U}} F'$

Example:

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

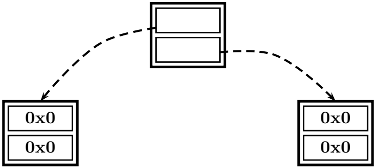
$$\begin{array}{ll} & \&\mathtt{x} \mapsto \alpha_0 * \alpha_0 \cdot \textbf{list} \\ \xrightarrow{\mathcal{U}} & \&\mathtt{x} \mapsto \alpha_0 * \alpha_0 \cdot \mathtt{next} \mapsto \alpha_1 * \alpha_0 \cdot \mathtt{data} \mapsto \beta_1 * \alpha_1 \cdot \textbf{list} \\ \xrightarrow{\mathcal{U}} &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

- We get the concrete state below:



Example: tree

- **Example:**



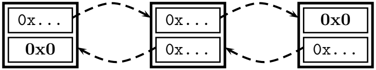
Inductive definition

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

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

Example: doubly linked list

- **Example:** binary tree



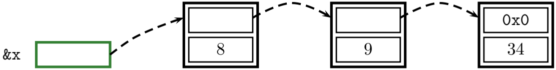
Inductive definition

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

$$\begin{array}{rcll} \alpha \cdot \mathbf{dll}(p) & := & & \alpha = 0 \wedge \mathbf{emp} \\ & & \vee & \alpha \neq 0 \wedge \alpha \cdot \mathtt{next} \mapsto \gamma * \alpha \cdot \mathtt{prev} \mapsto p * \gamma \cdot \mathbf{dll}(\alpha) \end{array}$$

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

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

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

A new overview of the remaining part of the lecture

How to apply separation logic to static analysis and design abstract interpretation algorithms based on it ?

In this lecture, we will:

- choose a **small but expressive set of separation logic formulas**
- define wide **families of abstract domains**
- study algorithms for **local concretization** (equivalent to focus) and **global abstraction** (widening...)

Outline

Choice of a set of formulas

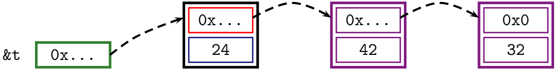
Our set of predicates

- An abstract value is a **separating conjunction** of terms
- Each term describes
  - either a **contiguous region**
  - or a **summarized region**, described by an **inductive defintion**

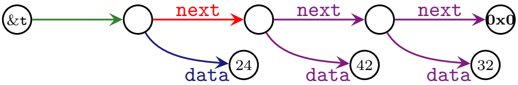
- Abstract elements have a straightforward interpretation as a **shape graph**
- Unless necessary, we omit environments (concretization into sets of heaps)

Abstraction into separating shape graphs

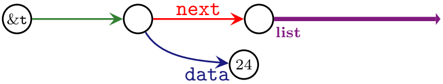
- **Memory splitting into regions**



- **Graph abstraction:** $\begin{cases} \text{values, addresses} & \longrightarrow & \text{nodes} \\ \text{cells} & \longrightarrow & \text{edges} \end{cases}$



- **Region summarization:**



  - abstraction **parameterized** by a **set of inductive definitions**

- Defines a **concretization relation**
- Let us formalize this...

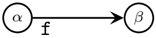
Contiguous regions

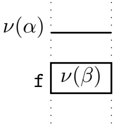
Shape graphs

- **Edges**: denote memory regions
- **Nodes**: denote values, i.e. addresses or cell contents

Points-to edge, denote **contiguous** memory regions

- **Separation logic formula**: $\alpha \cdot \mathtt{f} \mapsto \beta$
- **Abstract and concrete views**:





- **Concretization**:

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

  - ν: **bridge** between memory and values

Separation

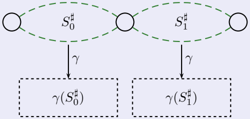
- A graph = a set of edges
- Denotes the separating conjunction of the edges

Empty graph **emp**

$\gamma_{\mathrm{S}}(\mathbf{emp}) = \{(\emptyset, \nu) \mid \nu : \mathbf{nodes} \to \mathbb{V}\}$ i.e., empty store

Separating conjunction

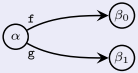
$$\gamma_{\mathrm{S}}(S_0^\sharp * S_1^\sharp) \quad = \quad \{(h_0 \circledast h_1, \nu) \mid (h_0, \nu) \in \gamma_{\mathrm{S}}(S_0^\sharp) \wedge (h_1, \nu) \in \gamma_{\mathrm{S}}(S_1^\sharp)\}$$



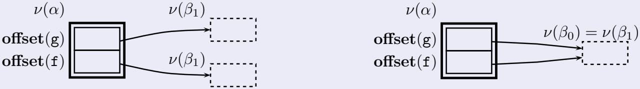
Separation example

Field splitting model

- Separation impacts edges / fields, *not pointers*
- Shape graph



accounts for both abstract states below:



In other words, separation

- asserts addresses are distinct
- says nothing about contents

Inductive edges

List definition

$$\begin{array}{rcl} \alpha \cdot \textbf{list} & ::= & (\textbf{emp}, \alpha = 0) \\ & | & (\alpha \cdot \texttt{next} \mapsto \beta_0 * \alpha \cdot \texttt{data} \mapsto \beta_1 * \beta_0 \cdot \textbf{list}, \alpha \neq 0) \end{array}$$

where **emp** denotes the **empty heap**

Concretization as a least fixpoint

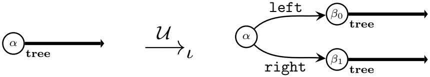
Given an inductive def ι

$$\gamma_{\mathrm{S}}(\alpha \cdot \iota) = \bigcup \left\{ \gamma_{\mathrm{S}}(F) \mid \alpha \cdot \iota \xrightarrow{\mathcal{U}} F \right\}$$

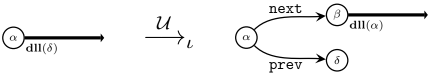
- **Alternate approach**:
  index inductive applications with **induction depth**
  allows to reason on **length of structures**

Inductive structures IV: a few instances

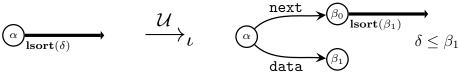
- More complex shapes: trees



- Relations among pointers: doubly-linked lists

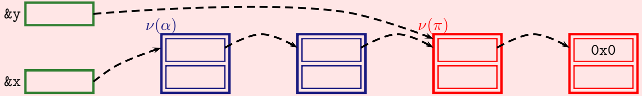


- Relations between pointers and numerical: sorted lists



Inductive segments

A frequent pattern



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

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

- This definition would **derive from list**
  Thus, we make **segments** part of the **fundamental predicates of the domain**

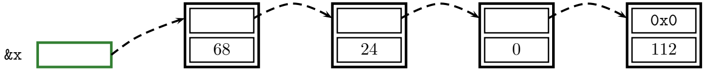
- **Multi-segments**: possible, but harder for analysis

Outline

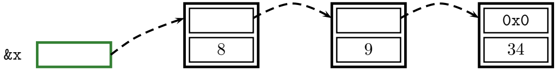
Example

How to express both shape **and** numerical properties ?

- **List of even elements:**



- **Sorted list:**



- **Many other examples:**
  - **list of open filed descriptors**
  - **tries**
  - **balanced trees**: red-black, AVL...
- Note: **inductive definitions also talk about data**

A first approach to domain combination

Basis

- Graphs form a **shape domain** $\mathbb{D}^\sharp_\mathrm{S}$
  abstract stores together with a **physical mapping** of nodes
  $$\gamma_\mathrm{S} : \mathcal{P}((\mathbb{D}^\sharp_\mathrm{S} \to \mathbb{M}) \times (\textbf{nodes} \to \mathbb{V}))$$
- Numerical values are taken in a **numerical domain** $\mathbb{D}^\sharp_\mathrm{num}$
  abstracts physical mapping of nodes
  $$\gamma_\mathrm{num} : \mathbb{D}^\sharp_\mathrm{num} \to \mathcal{P}((\textbf{nodes} \to \mathbb{V}))$$

Concretization of the combined domain **[CR]**

$$\gamma(S^\sharp, N^\sharp) = \{\sigma \in \mathbb{M} \mid \exists \nu \in \gamma_\mathrm{num}(N^\sharp),\ (\sigma, \nu) \in \gamma_\mathrm{S}(S^\sharp)\}$$

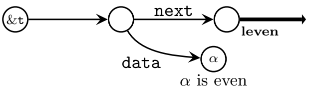
- Quite similar to a **reduced product**

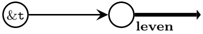
Combination by reduced product

Reduced product

- **Product abstraction:** $\mathbb{D}^\sharp = \mathbb{D}^\sharp_0 \times \mathbb{D}^\sharp_1$
  $\gamma(x_0, x_1) = \gamma(x_0) \cap \gamma(x_1)$
- **Reduction:** $\mathbb{D}^\sharp_r$ is the quotient of $\mathbb{D}^\sharp$ by the equivalence relation $\equiv$ defined by $(x_0, x_1) \equiv (x'_0, x'_1) \iff \gamma(x_0, x_1) = \gamma(x'_0, x'_1)$
- Domain operations (join, transfer functions) are **pairwise** (are usually composed with reduction)
- Why not to use a product of the shape domain with a numerical domain ?

- How to compare / join the following two elements ?





Towards a more adapted combination operator

Why does this fail here ?

- The set of nodes / symbolic variables **is not fixed**
- Variables represented in the numerical domain depend on the shape abstraction

$\Rightarrow$ Thus the product is **not** symmetric

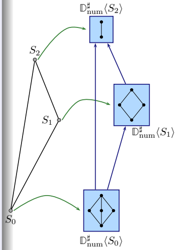
Intuitions

- Graphs form a **shape domain** $\mathbb{D}^{\sharp}_{\mathrm{S}}$
- For **each** graph $S^{\sharp} \in \mathbb{D}^{\sharp}_{\mathrm{S}}$, we have a **numerical lattice** $\mathbb{D}^{\sharp}_{\mathbf{num}\langle S^{\sharp}\rangle}$
  - example: if graph $S^{\sharp}$ contains nodes $\alpha_0, \alpha_1, \alpha_2$, $\mathbb{D}^{\sharp}_{\mathbf{num}\langle S^{\sharp}\rangle}$ should abstract $\{\alpha_0, \alpha_1, \alpha_2\} \rightarrow \mathbb{V}$
- **An abstract value is a pair $(S^{\sharp}, N^{\sharp})$, such that $N^{\sharp} \in \mathbb{D}^{\sharp}_{\mathbf{num}\langle N^{\sharp}\rangle}$**

Cofibered domain

Definition [AV]

- **Basis:** abstract domain $(\mathbb{D}_0^\sharp, \sqsubseteq_0^\sharp)$, with concretization $\gamma_0 : \mathbb{D}_0^\sharp \to \mathbb{D}$
- **Function:** $\phi : \mathbb{D}_0^\sharp \to \mathcal{D}_1$, where each element of $\mathcal{D}_1$ is an abstract domain $(\mathbb{D}_1^\sharp, \sqsubseteq_1^\sharp)$, with a concretization $\gamma_{\mathbb{D}_1^\sharp} : \mathbb{D}_1^\sharp \to \mathbb{D}$
- **Lift functions:** $\forall x^\sharp, y^\sharp \in \mathbb{D}_0^\sharp$, such that $x^\sharp \sqsubseteq_0^\sharp y^\sharp$, there exists a function $\Pi_{x^\sharp, y^\sharp} : \phi(x^\sharp) \to \phi(y^\sharp)$, that is monotone for $\gamma_{x^\sharp}$ and $\gamma_{y^\sharp}$
- **Domain:** $\mathbb{D}^\sharp$ is the set of **pairs $(x_0^\sharp, x_1^\sharp)$ where** $x_1^\sharp \in \phi(x_0^\sharp)$



- **Generic product**, where the second lattice depends on the first
- Provides a generic scheme for **widening**, **comparison**

Domain operations

- **Lift functions** allow to **switch domain when needed**

Comparison of $(x_0^\sharp, x_1^\sharp)$ and $(y_0^\sharp, y_1^\sharp)$

1. First, **compare** $x_0^\sharp$ **and** $y_0^\sharp$ in $\mathbb{D}_0^\sharp$
2. If $x_0^\sharp \sqsubseteq_0^\sharp y_0^\sharp$, **compare** $\Pi_{x_0^\sharp, y_0^\sharp}(x_1^\sharp)$ **and** $y_1^\sharp$

Widening of $(x_0^\sharp, x_1^\sharp)$ and $(y_0^\sharp, y_1^\sharp)$

1. First, compute the **widening in the basis** $z_0^\sharp = x_0^\sharp \triangledown y_0^\sharp$
2. Then **move to** $\phi(z_0^\sharp)$, by computing $x_2^\sharp = \Pi_{x_0^\sharp, z_0^\sharp}(x_1^\sharp)$ and $y_2^\sharp = \Pi_{y_0^\sharp, z_0^\sharp}(y_1^\sharp)$
3. Last **widen in** $\phi(z_0^\sharp)$: $z_1^\sharp = x_2^\sharp \triangledown_{z_0^\sharp} y_2^\sharp$

$$(x_0^\sharp, x_1^\sharp) \triangledown (y_0^\sharp, y_1^\sharp) = (z_0^\sharp, z_1^\sharp)$$

Domain operations

Transfer functions, e.g., assignment

- Require memory location be **materialized** in the graph
  - i.e., the graph may have to be modified
  - the numerical component should be updated with lift functions
- Require **update** in the graph and the numerical domain
  - i.e., the numerical component should be kept coherent with the graph

Proofs of soundness of transfer functions rely on:

- the soundness of the lift functions
- the soundness of both domain transfer functions

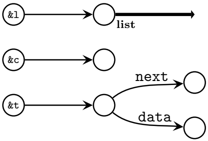
Outline

Static analysis overview

A list insertion function:

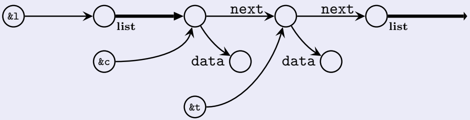
```
list ⋆ l assumed to point to a list
list ⋆ t assumed to point to a list element
list ⋆ c = l;
while(c != NULL && c -> next != NULL && (...)){
    c = c -> next;
}
t -> next = c -> next;
c -> next = t;
```

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



Result of the (interprocedural) analysis

- **Over-approximations** of reachable concrete states
  e.g., at the loop exit:



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**

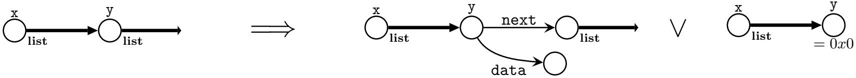
Should be **sound** i.e., not forget any concrete behavior

Abstract operators

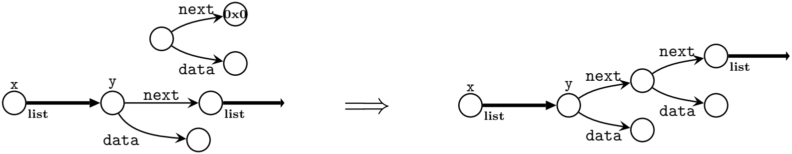
- **Join** and **widening**: over-approximation
- **Inclusion checking**: check stabilization of abstract iterates

The algorithms underlying the transfer functions

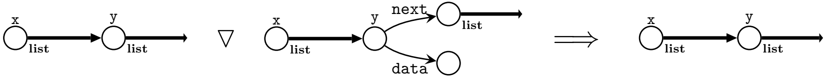
- **Unfolding**: cases analysis on summaries



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



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

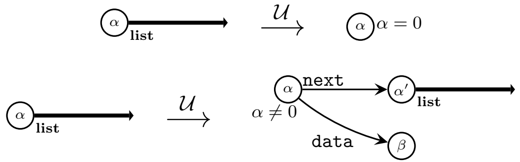


Unfolding as a local case analysis

Unfolding principle

- **Case analysis**, based on the inductive definition
- Generates **symbolic disjunctions**
  analysis performed in a **disjunction domain**

- Example, for lists:



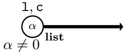
- **Numeric predicates: approximated in the numerical domain**

Soundness: by definition of the concretization of inductive structures

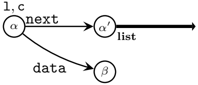
$$\gamma_{\mathrm{S}}(S) \subseteq \bigcup \{\gamma_{\mathrm{S}}(S_0) \mid S \xrightarrow{\mathcal{U}} S_0\}$$

Local reasoning

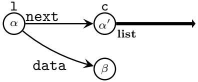
Before the assignment $\mathtt{c} = \mathtt{c} \rightarrow \mathtt{next};$:



1. **Result of the unfolding:** two rules to consider
   - **empty list** does **not** need be considered
     contradiction with num. invariant $\alpha \neq 0$
   - **non-empty list** case:



2. **Result of the assignment:**

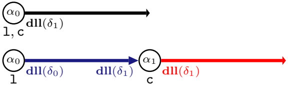


note: sound analysis of the assignment in itself is **trivial** (**frame rule**)

Unfolding and degenerated cases

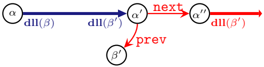
```
assume(l points to a dll)
c = l;
① while(c ≠ NULL && condition)
      c = c -> next;
② if(c ≠ 0 && c -> prev ≠ 0)
      c = c -> prev → prev;
```

- at ①:
- at ②:

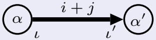


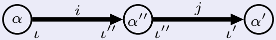
⇒ **non trivial unfolding**

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

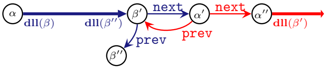


Segment splitting lemma: basis for segment unfolding



describes the same set of stores as


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



- **Implementation issue**: discover **which inductive edge** to unfold
  non decidable !

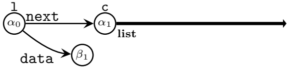
Need for a folding operation

- Back to the **list traversal** example...

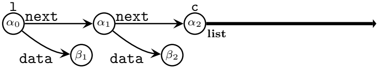
```
assume(l points to a list)
c = l;
while(c ≠ NULL){
  c = c → next;
}
```

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

  - at **iteration 1**:



  - at **iteration 2**:



- 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 $\triangledown$

- **Over-approximates join:**

$$\begin{cases} X^\sharp & \subseteq & \gamma(X^\sharp \triangledown Y^\sharp) \\ Y^\sharp & \subseteq & \gamma(X^\sharp \triangledown Y^\sharp) \end{cases}$$

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

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

Canonicalization

Upper closure operator

$\rho : \mathbb{D}^\sharp \longrightarrow \mathbb{D}^\sharp_{\mathbf{can}} \subseteq \mathbb{D}^\sharp$ 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 $\rho_\vee : \mathbb{D}^\sharp_\vee \longrightarrow \mathbb{D}^\sharp_{\mathbf{can}\vee}$ and $\rho_\vee(X^\sharp) = \{\rho(x^\sharp) \mid x^\sharp \in X^\sharp\}$ where ρ is an uco and $\mathbb{D}^\sharp_{\mathbf{can}}$ has finite height

- Usually **more simple to compute**
- Canonicalization is used in **many shape analysis tools**: **TVLA**, most **separation logic** based analysis tools
- However **less powerful** than widening: does not exploit history of computation

Per region weakening

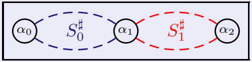
The weakening principles shown in the following apply both in canonicalization and widening approaches

We can apply the **local reasoning principle** to weakening

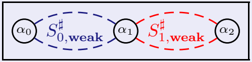
- inclusion test (comparison)
- canonicalization
- join / widening

Application: inclusion test

- Operator $\sqsubseteq^\sharp$ should satisfy $X^\sharp \sqsubseteq^\sharp Y^\sharp \Longrightarrow \gamma(X^\sharp) \subseteq \gamma(Y^\sharp)$
- If $S_0^\sharp \sqsubseteq^\sharp S_{0,\mathbf{weak}}^\sharp$ and $S_1^\sharp \sqsubseteq^\sharp S_{1,\mathbf{weak}}^\sharp$



$\sqsubseteq^\sharp$



Inductive weakening

Weakening identity

- $X^\sharp \sqsubseteq^\sharp X^\sharp$...
- Trivial, but useful, when a graph region appears in both widening arguments

Weakening unfolded region

- If $S_0^\sharp \xrightarrow{\mathcal{U}} S_1^\sharp$, $\gamma(S_1^\sharp) \subseteq \gamma(S_0^\sharp)$
- Soundness follows the the **soundness of unfolding**

- **Application to a simple example:**

Comparison operator in the shape domain

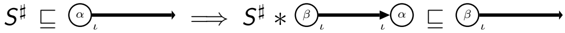
Algorithm structure

- **Based on separation** and **local reasoning**:

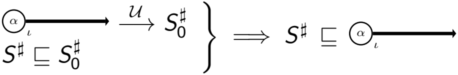
$$\gamma(S^\sharp_0) \subseteq \gamma(S^\sharp_1) \Longrightarrow \gamma(S^\sharp_0 * S^\sharp) \subseteq \gamma(S^\sharp_1 * S^\sharp)$$

- **Algorithm:**
  - applies local rules and "**consumes**" graph regions proved weaker
  - keeps discovering new rule applications

- **Structural rules** such as:
  - **segment splitting:**



  - **inductive folding:**

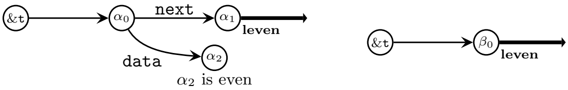


Correctness:

$$S^\sharp_0 \sqsubseteq S^\sharp_1 \Longrightarrow \gamma(S^\sharp_0) \subseteq \gamma(S^\sharp_1)$$

Comparison operator in the combined domain

We need to tackle the fact nodes names may differ (cofibered domain)



Instrumented comparison in the shape domain

- Comparison $S_0^\sharp \sqsubseteq^\sharp S_1^\sharp$: rules should compute **a physical mapping** $\Psi : \mathbf{nodes}_1 \longrightarrow \mathbf{nodes}_0$
- **Soundness condition:** $(\sigma, \nu) \in \gamma_S(S_0^\sharp) \Longrightarrow (\sigma, \nu \circ \Psi) \in \gamma_S(S_0^\sharp)$

Comparison in the cofibered domain

- **Lift function for numerical abstract values:** $\Pi_{S_0^\sharp, S_1^\sharp}(N_0^\sharp) = N_0^\sharp \circ \Psi$
- Thus, we simply need to compare $N_0^\sharp \circ \Psi$ **and** $N_1^\sharp$

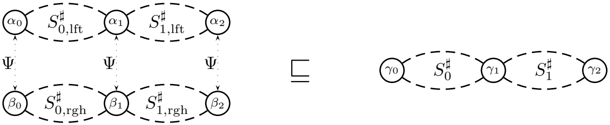
Join operator

- Similar iterative scheme, based on local rules
- But needs to reason locally on **two graphs** in the same time: each rule maps two regions into a common over-approximation

Graph partitioning and mapping

- **Inputs:** $S_0^\sharp, S_1^\sharp$
- Performed by a function Ψ : nodes$_0 \times$ nodes$_1 \rightarrow$ nodes$_\sqcup$
- Ψ is computed at the same time as the join

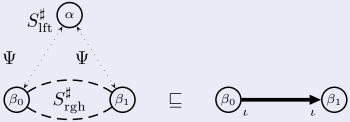
If $\forall i \in \{0, 1\},\ \forall s \in \{\mathrm{lft}, \mathrm{rgh}\},\ S_{i,s}^\sharp \sqsubseteq^\sharp S_s^\sharp$,



Segment introduction

Rule

if



then

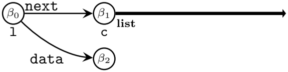
$S^\sharp_{\rm lft} \nabla S^\sharp_{\rm rgh} =$ 

$$(\alpha, \beta_0) \overset{\Psi}{\longleftrightarrow} \gamma_0$$

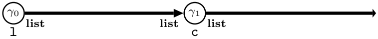
$$(\alpha, \beta_1) \overset{\Psi}{\longleftrightarrow} \gamma_1$$

- **Application to list traversal**, at the end of iteration 1:
  - **before iteration 0:**

  - **end of iteration 0:**



  - **join, before iteration 1:**

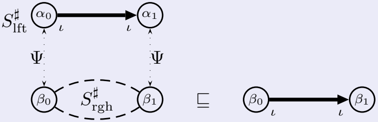


$$\begin{cases} \Psi(\alpha_0, \beta_0) = \gamma_0 \\ \Psi(\alpha_0, \beta_1) = \gamma_1 \end{cases}$$

Segment extension

Rule

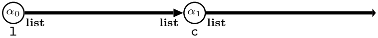
if



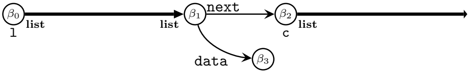
then

$$\begin{cases} S^\sharp_{\mathrm{lft}} \nabla S^\sharp_{\mathrm{rgh}} = \\ (\alpha_0, \beta_0) \overset{\Psi}{\longleftrightarrow} \gamma_0 \\ (\alpha_1, \beta_1) \overset{\Psi}{\longleftrightarrow} \gamma_1 \end{cases}$$

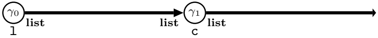
- **Application to list traversal**, at the end of iteration 1:
  - **previous invariant before iteration 1:**



  - **end of iteration 1:**



  - **join, before iteration 1:**



$$\begin{cases} \Psi(\alpha_0, \beta_0) = \gamma_0 \\ \Psi(\alpha_1, \beta_2) = \gamma_1 \end{cases}$$

Rewrite system properties

- Comparison, canonicalization and widening algorithms can be considered **rewriting systems over tuples of graphs**
- Each step applies a rule / computation step

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

Properties

Inclusion checking is sound

If $S_0^\sharp \sqsubseteq^\sharp S_1^\sharp$, then $\gamma(S_0^\sharp) \subseteq \gamma(S_1^\sharp)$

Canonicalization is sound

$$\gamma(S^\sharp) \subseteq \gamma(\rho_{\mathbf{can}}(S^\sharp))$$

Widening is sound and terminating

$$\gamma(S_0^\sharp) \subseteq \gamma(S_0^\sharp \nabla S_1^\sharp)$$
$$\gamma(S_1^\sharp) \subseteq \gamma(S_0^\sharp \nabla S_1^\sharp)$$

∇ ensures termination of abstract iterates

- **Soundness** of **local reasoning** and of **local rules**
- **Termination of widening**: ∇ can introduce only segments, and may not introduce infintely many of them

Outline

Interprocedural analysis

- Analysis of programs that consist in several functions (or procedures)
- Difficulty: how to cope with multiple (possibly recursive) calls

Relational approach

- analyze each function **once**
- compute **function summaries** abstraction of input-output relations
- analysis of a **function call**: instantiate the function summary (**hard**)

Inlining approach

- **inline functions** at function calls
- just an extension of **intraprocedural analysis**

- In this section, we study the **inlining approach** for recursion
- Side result: a **widening for inductive definitions**

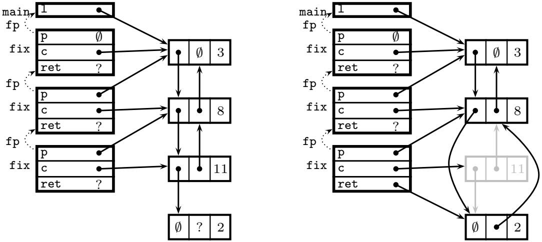
Approaches to interprocedural analysis

“relational” approach	“inlining” approach
analyze each definition **abstracts $\mathcal{P}(\bar{\mathbb{S}} \rightarrow \bar{\mathbb{S}})$**	**analyze each call** **abstracts $\mathcal{P}(\mathbb{S})$**
+ modularity + reuse of invariants - deals with state relations - complex higher order iteration strategy	- not modular - re-analysis in $\neq$ contexts + deals with states + straightforward iteration
challenge: frame problem	**challenge: unbounded calls**

Challenges in interprocedural analysis

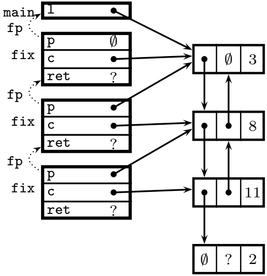
```
void main(){
   dll ⋆ l; //assume l points to a sll
   l = fix(l, NULL);
}
dll ⋆ fix(dll ⋆ c, dll ⋆ p){
   dll ⋆ ret;
   if(c != NULL){
      c -> prev = p;
      ①c -> next = fix(c -> next, c);
      if(check(c -> data)){
         ret = c -> next;
         remove(c);
      ②} else ret = c;
   }
   return ret;
}
```

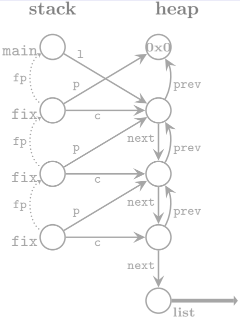
- turns a linked list into a doubly linked list
- removes some elements



- **Heap** is **unbounded**, **needs abstraction** (shape analysis)
- But **stack** may **also** grow **unbounded**, **needs abstraction**
- **Complex relations** between both **stack** and **heap**

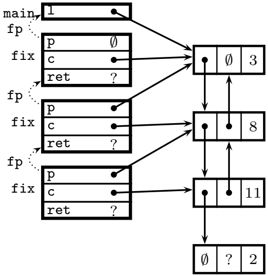
Calling contexts as shape graphs

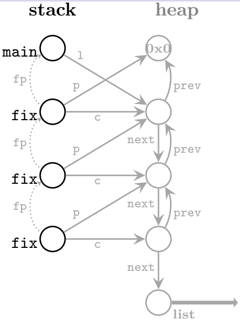




- **Concrete assembly call stack** modelled in a **separating shape graph** together with the **heap**
  - **one node** per **activation record address**

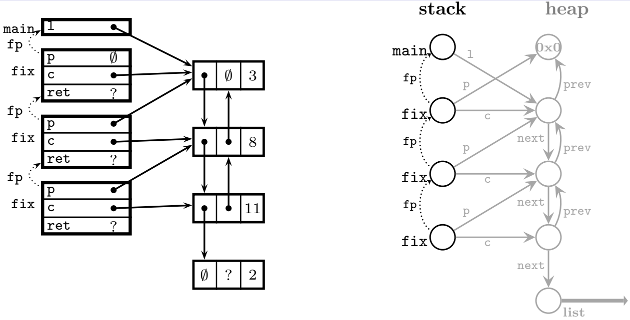
Calling contexts as shape graphs





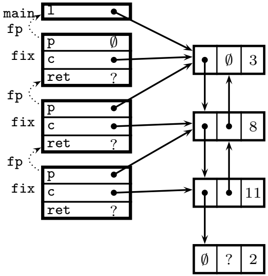
- **Concrete assembly call stack** modelled in a **separating shape graph** together with the **heap**
  - **one node** per **activation record address**

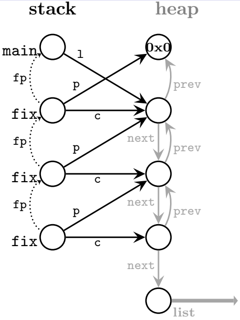
Calling contexts as shape graphs



- **Concrete assembly call stack** modelled in a **separating shape graph** together with the **heap**
  - **one node** per **activation record address**
  - **explicit edges** for **frame pointers**

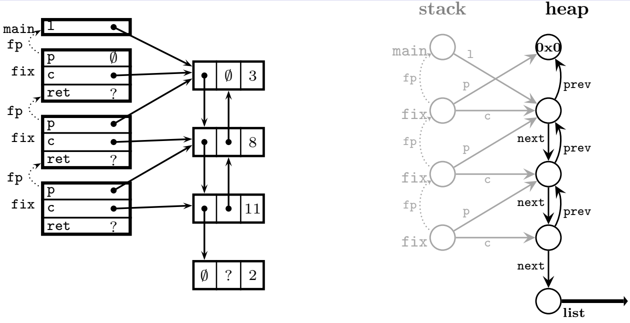
Calling contexts as shape graphs





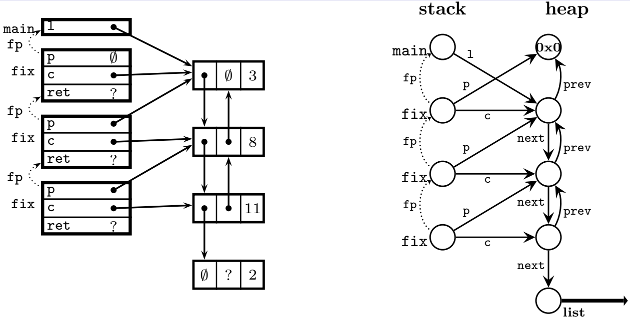
- **Concrete assembly call stack** modelled in a **separating shape graph** together with the **heap**
  - **one node** per **activation record address**
  - **explicit edges** for **frame pointers**
  - **local variables** turn into **activation record fields**

Calling contexts as shape graphs



- **Concrete assembly call stack** modelled in a **separating shape graph** together with the **heap**
  - **one node** per **activation record address**
  - **explicit edges** for **frame pointers**
  - **local variables** turn into **activation record fields**

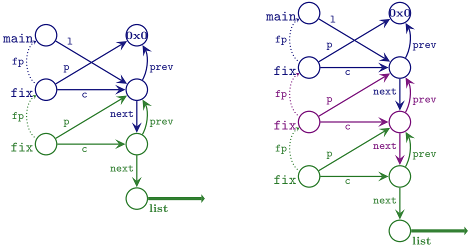
Calling contexts as shape graphs



- **Concrete assembly call stack** modelled in a **separating shape graph** together with the **heap**
  - **one node** per **activation record address**
  - **explicit edges** for **frame pointers**
  - **local variables** turn into **activation record fields**

Inference of a call-stack inductive structure

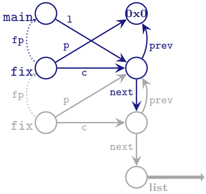
- **Second** and **third iterates**: **a repeating pattern**

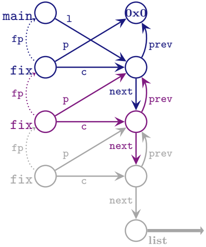


- **Computing an inductive rule** for **summarization**: **subtraction**

Inference of a call-stack inductive structure

- **Second** and **third iterates**: **a repeating pattern**

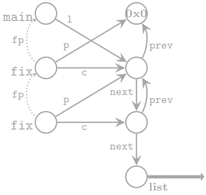


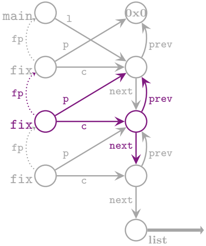


- **Computing an inductive rule** for **summarization**: **subtraction**
  - subtract **top-most activation record**

Inference of a call-stack inductive structure

- **Second** and **third iterates**: **a repeating pattern**

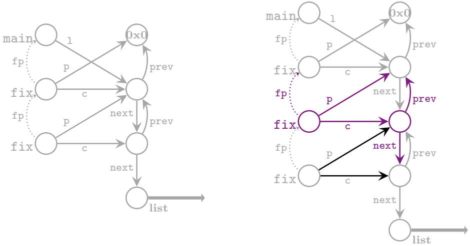




- **Computing an inductive rule** for **summarization**: **subtraction**
  - subtract **top-most activation record**
  - subtract **common stack region**

Inference of a call-stack inductive structure

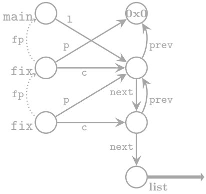
- **Second** and **third iterates**: **a repeating pattern**

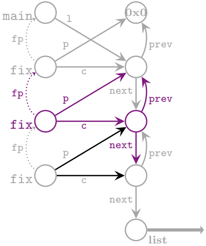


- **Computing an inductive rule** for **summarization**: **subtraction**
  - subtract **top-most activation record**
  - subtract **common stack region**
  - gather **relations** with next activation records: additional **parameters**
  - collect **numerical constraints**

Inference of a call-stack inductive structure

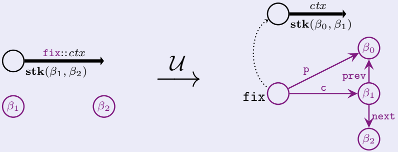
- **Second** and **third iterates**: **a repeating pattern**





- **Computing an inductive rule** for **summarization**: **subtraction**

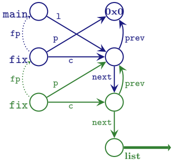
Inferred inductive rule



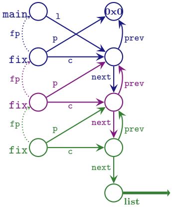
Inference of a call-stack summary: widening iterates

- **Fixpoint at function entry**:

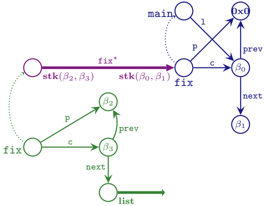
first iterate:



second iterate:



widened iterate:



Fixpoint reached

- **Fixpoint upon function return**:
  - function return involves **unfolding** of stack summaries
  - **simpler widening sequence**: no rule to infer

Widening over inductive definitions

Domain structure

An abstract value should comprise:

- a set S of **unfolding rules for the stack inductive**
- a **shape graph** G
- a **numeric abstract value** N

Shape graph G is defined in a lattice specified by S,
thus, this is an instance of the **cofibered abstraction**

- **Lift functions** are trivial:
  - adding rules simply weakens shape graphs
  - i.e., no need to change them syntactically, their concretization just gets weaker
- **Termination** in the lattice of rules:
  limiting of the number of rules that can be generated to some given bound

Outline

Abstraction choices

Many families of symbolic abstractions including TVLA and separation logic based approaches

- Variants: region logic, ownership, fractional permissions

Common ingredients

- **Splitting** of the heap in regions
  - **TVLA**: covering, via embedding
  - **Separation logic**: partitioning, enforced at the concrete level
- Use of **induction** in order to summarize **large regions**

- More **limited pointer analyses**: no inductives, no summarization...

Algorithms

Rather different process, compared to numerical domains

From abstract to concrete (**locally**)

- **Unfold** abstract properties in a local maner
- Allows **quasi-exact** analysis of usual operations (assignment, condition test...)

From concrete to abstract (**globally**)

- Guarantees **termination**
- Allows to **infer** pieces of code **build complex structures**
- **Intuition:**
  - static analysis involves **post-fixpoint** computations (over program traces)
  - widening produces a **fixpoint** over memory cells

Open problems

Many opportunities for research:

- Improving **expressiveness**
  e.g., **sharing in data-structures**
  - new abstractions
  - combining several abstractions into more powerful ones
- Improving **scalability**
  - shape analyses remain expensive analyses, with few "cheap" and useful abstractions
  - cut down the cost of complex algorithms
  - isolate smaller families of predicates
- **Applications**, beyond software safety:
  - security
  - verification of functional properties

Internships

Several topics possible, soon to be announced on the lecture webpage:

Internal reduction operator

- inductive definitions are very expressive thus tricky to reason about
- design of an **internal reduction operator** on abstract elements with inductive definitions

Modular inter-procedural analysis

- a relation between pre-conditions and post-conditions can be formalized in a **single shape graph**
- design of an abstract domain **for relations between states**

Bibliography

- **[SRW]**: Parametric Shape Analysis via 3-Valued Logic.
  Shmuel Sagiv, Thomas W. Reps et Reinhard Wilhelm.
  In POPL'99, pages 105–118, 1999.
- **[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 et Hongseok Yang.
  In TACAS'06, pages 287–302.
- **[AV]**: Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs.
  Arnaud Venet. In SAS'96, pages 366–382.
- **[CR]**: Relational inductive shape analysis.
  Bor-Yuh Evan Chang et Xavier Rival.
  In POPL'08, pages 247–260, 2008.