Abstract domains for the static analysis of programs manipulating complex data-structures

Xavier Rival

INRIA

August, 26th. 2011

Software verification

My research topic: software verification

- **Safety** (absence of runtime errors, non desired behaviors), **functional properties**...
- Those are all **undecidable properties**

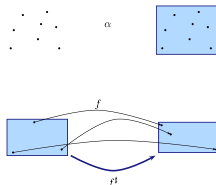
Abstract interpretation approach: **sound**, **automatic** but **incomplete**

- **Abstraction**:

  use of **conservative approximations**

- **Abstract transfer functions**,

  preserving conservative approximations



- **Widening**: **terminating** over-approximation of **concrete join**

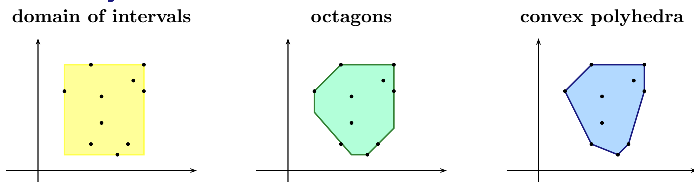
Abstract domain: abstraction + transfer function + widening

Verification of numerical properties

Numerical abstraction

- **Abstraction** of $\mathcal{P}(\mathbb{X} \to \mathbb{Z})$ or in $\mathcal{P}(\mathbb{X} \to \mathbb{F})$
- **Transfer functions**: **assignments**, **numerical conditions**...

- A **great variety of abstractions available**:



- Available as **libraries** of domains: **Apron**...
- **Astrée** analyzer: verification of highly critical softwares: avionics
  - up to 1 Million lines C applications
  - modular numerical abstract domain: **reduced product of simpler abstractions**, **tailored to applications**

Memory abstraction difficulties

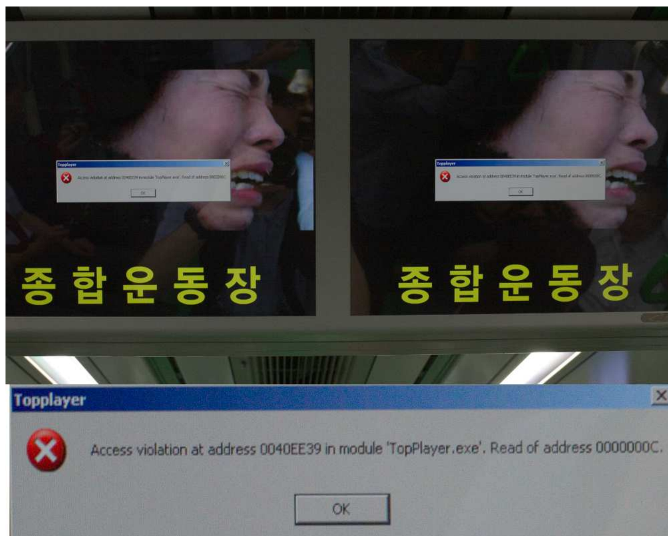
Memory abstraction

- **Abstraction** of $\mathcal{P}($ **Environment** $\times$ **Stack** $\times$ **Heap** $)$
- **Operations to analyze**: C statements including **pointer arithmetic**, **memory management**...
- **Operations to verify**: **memory safety**, **structure preservation**...

Difficulties:

1. **Complex concrete semantics**: memory states, pointers...
2. **Huge variety of properties to abstract**:
   - **linked structures**: lists, trees...
   - **arrays**
   - structures involving **relations** (sorted lists, balanced trees) or **sharing**
   - **combination** of numerical and memory properties
3. **Expensive analysis algorithms** to infer such abstractions

...


종 합 운 동 장
종 합 운 동 장
Topplayer
Access violation at address 0040EE39 in module 'TopPlayer.exe'. Read of address 0000000C.
OK

Main contributions

Long term goal

- **Set up a general purpose library of memory abstract domains**

1. Choice of a **concrete semantics**
   should not limit the scope of the analysis
2. **Abstraction** based on **inductive structural properties**
   expresses a wide range of properties, incl. shape + numerical
3. **Static analysis algorithms** to infer complex properties
   instantiations for low level programs and recursive procedures
   prototype implementation

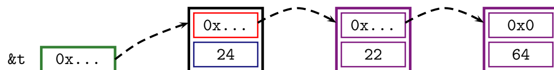
- Collaboration with Bor-Yuh Evan Chang (U. of Colorado, USA)
- Internships of Vincent Laviron, Suzanne Renard and Antoine Toubhans

Outline

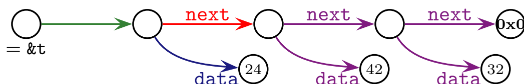
1. Introduction
2. A parametric abstraction
3. Static analysis algorithms
4. Instantiation to the analysis of low-level programs
5. Instantiation to interprocedural analysis
6. Perspectives and implementation

Overview of the abstraction

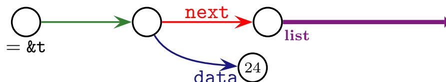
- **Memory partitioning into regions**



- **Graph abstraction:** $\left\{ \begin{array}{lcl} \text{values, addresses} & \longrightarrow & \text{nodes} \\ \text{cells} & \longrightarrow & \text{edges} \end{array} \right.$



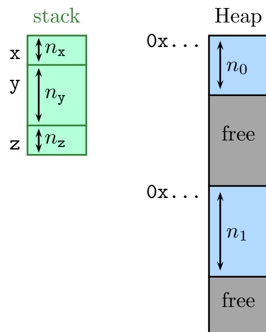
- **Region summarization:**



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

- Defines a **concretization relation**

Concrete semantics



A very concrete concrete semantics:

- division of the heap in **blocks**
  either **allocated** or **free**
- **memory cells** have a **numeric address** and a **numeric size**
- **pointers** are **numeric values**
  base + offset

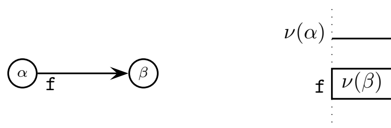
Contiguous regions

Shape graphs

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

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

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



- **Concretization**:

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

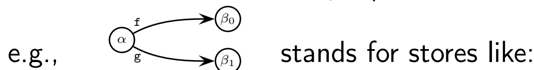
  - ν: **bridge** between memory and values

Separating conjunction

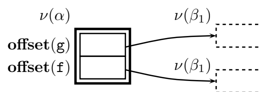
Shape graphs and separation

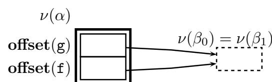
- **Distinct edges** denote **disjoint heap regions** $*$ **conjunction**

- **Advantage**: allows **local reasoning**
  e.g., easier analysis of **updates**
- **Disadvantage**: **maintaining separation** makes some analysis operations **harder** to design
- We use a **field splitting model**
  i.e., separation impacts edges / fields, *not pointers*

e.g., 

stands for stores like:

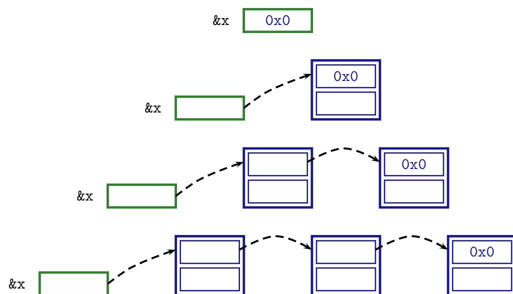




Inductive structures I: need for summarization

- Infinitely many concrete states
- A global structure invariant: x points to a list

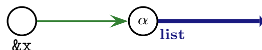
Concrete: all lists pointed to by x



- **Abstraction**:

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

- **Graph**:



Inductive structures II: inductive definitions in separation logic

List definition

$$\alpha \cdot \mathbf{list} ::= (\mathbf{emp}, \alpha = 0) \mid (\alpha \cdot \mathtt{next} \mapsto \beta_0 * \alpha \cdot \mathtt{data} \mapsto \beta_1 * \beta_0 \cdot \mathbf{list}, \alpha \neq 0)$$

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

- List structures abstracted by $\alpha \cdot$ **list**
- **Summarization:** a finite formula describes infinitely many heaps

Practical implementation in verification/analysis tools

- **Verification:** hand-written definitions
- **Analysis:** either built-in (Smallfoot, Space Invader) or user-supplied (TVLA, Xisa, [POPL'08]), or partly inferred (Xisa, [POPL'11])

Inductive structures III: concretization

Concretization as a least fixpoint

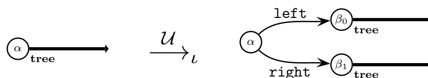
Given an inductive def ι

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

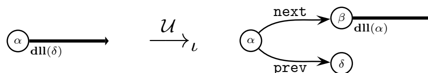
- **Alternate approach**:
  index inductive applications with **induction depth**
  allows to reason on **length of structures**

Inductive structures IV: a few instances

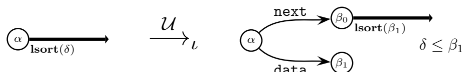
- **More complex shapes**: trees



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



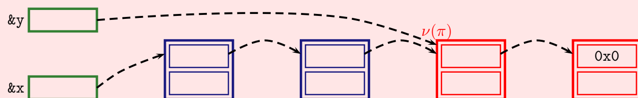
- **Relations between pointers and numerical**: sorted lists



All together: red-black trees with parent pointers [POPL'08]

Inductive segments

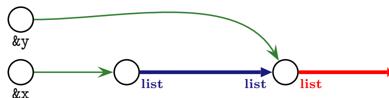
A frequent pattern



- 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**



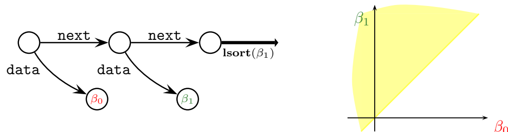
- **Multi-segments**: possible, but harder for analysis

Product abstraction

How to abstract both memory and contents properties ?

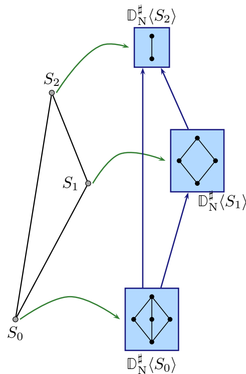
- We need a **sort of a product abstraction**

- **Example**: **partly unfolded sorted list**



Numerical domain variables: (**some**) **nodes**

- **Cofibered domain** structure [Venet, 1996]
  - a **lattice** $\mathbb{D}^\sharp_{\mathrm{S}}$ **of shape graphs**
  - for each shape graph $S \in \mathbb{D}^\sharp_{\mathrm{S}}$, a distinct **numerical lattice** $\mathbb{D}^\sharp_{\mathrm{N}}\langle S \rangle$
  - abstract values are **pairs** (S, N) where $N \in \mathbb{D}^\sharp_{\mathrm{N}}\langle S \rangle$
  - analysis should use **conversion functions**



Outline

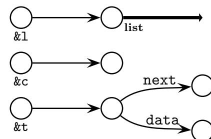
1. Introduction
2. A parametric abstraction
3. Static analysis algorithms
4. Instantiation to the analysis of low-level programs
5. Instantiation to interprocedural analysis
6. Perspectives and implementation

Static analysis overview

A list insertion function:

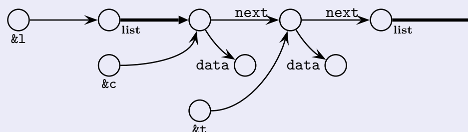
```
list ⋆ l assumed to point to a list
list ⋆ t assumed to point to a list element
list ⋆ c = l;
while(c != NULL && c -> next != NULL && (...)){
    c = c -> next;
}
t -> next = c -> next;
c -> next = t;
```

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



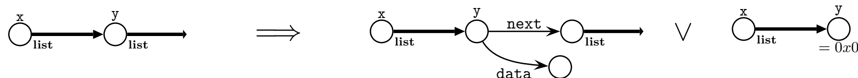
Result of the (interprocedural) analysis

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

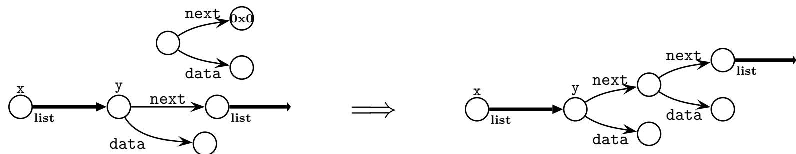


The algorithms for static analysis

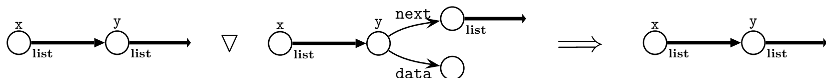
- **Unfolding**: cases analysis on summaries



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



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

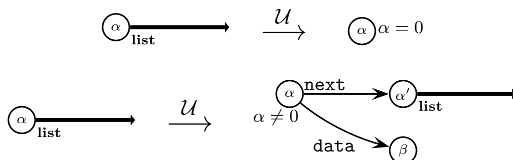


Unfolding as a local case analysis

Unfolding principle

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

- Example, for lists:



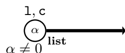
- **Numeric predicates: approximated in the numerical domain**

Soundness: by definition of the concretization of inductive structures

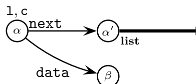
$$\gamma_{\mathrm{S}}(S) \subseteq \bigcup \{\gamma_{\mathrm{S}}(S_0) \mid S \xrightarrow{\mathcal{U}} S_0\}$$

Local reasoning

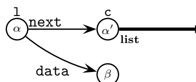
Before the assignment $\mathtt{c} = \mathtt{c} \rightarrow \mathtt{next}$;:



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:**

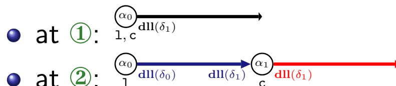


note: sound analysis of the assignment in itself is **trivial** (**frame rule**)

Unfolding and degenerated cases

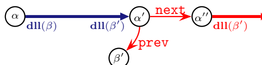
```
assume(l points to a dll)
c = l;
① while(c ≠ NULL && condition)
      c = c -> next;
② if(c ≠ 0 && c -> prev ≠ 0)
      c = c -> prev → prev;
```

- at ①:
- at ②:



⇒ non trivial unfolding

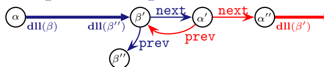
- **Materialization of** c -> prev:



Segment splitting lemma: basis for segment unfolding

$\alpha \xrightarrow[\iota \quad \iota']{i+j} \alpha'$ describes the same set of stores as $\alpha \xrightarrow[\iota \quad \iota'']{i} \alpha'' \xrightarrow[\iota'' \quad \iota']{j} \alpha'$

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



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

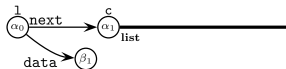
Widening I: need for a folding operation

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

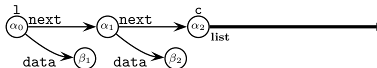
```
assume(l points to a list)
c = l;
while(c ≠ NULL){
  c = c → next;
}
```

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

  - at **iteration 1**:



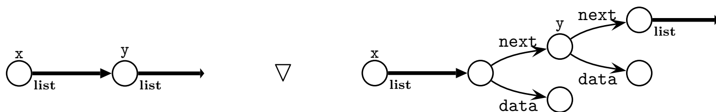
  - at **iteration 2**:



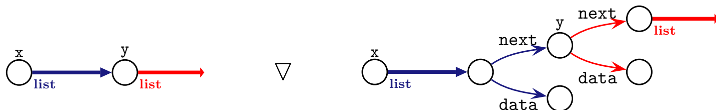
- How to guarantee **termination** of the analysis ?
- How to **introduce segment edges** / perform **abstraction** ?

Widening II: algorithm overview

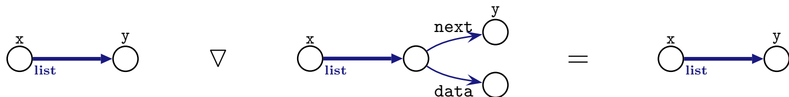
- Takes **two abstract values as inputs**



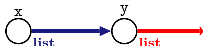
- **Region matching** (non unique choice: use of **strategies**)



- **Semantic rules** for **per region weakening**

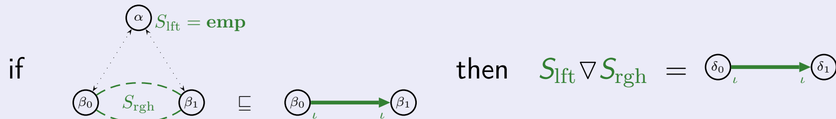


- **Widening:**



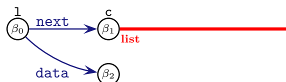
Widening III: an example of a widening meta-rule

Segment introduction meta rule (*for all* ι)

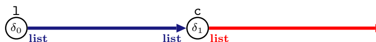


- **Application to list traversal**, at the end of iteration 1:
  - **before iteration 0:**

  - **end of iteration 0:**



  - **join, before iteration 1:**



Widening IV: properties

Rewrite system properties

- **A set of sound pair-wise weakening rules**
- Weakening **terminates**: widening is **computable**
- Weakening rules are **not confluent**: need for **adequate strategies** otherwise: risk of very imprecise results

Widening properties

- **Sound**: returns an **over approximation** of concrete join
- **Terminating**:
  - introduction of segments / inductive edges consumes points-to edges
  - after finitely many iterates, the size of graphs decreases
- **Soundness** and **termination** also hold in the **cofibered domain** assumption: use of a **widening** in the numeric domain

Widening and other folding operators

Canonicalization (TVLA, SmallFoot)

Another **kind of widening operator**:

- **Ignores its first argument**: **unary** weakening operator
  thus, does not depend on history
- Returns values in a **finite lattice**
- Applies **not only for fixpoint computation**: incremental weakening

- **Widening** (Xisa) exploits the **analysis history**
  - **quick convergence**, **few disjuncts**
  - applies mainly **on fixpoint computation**
- Ideally, a **local abstraction** would be **useful** together with history dependeng widening
  - e.g., to **weaken** locally abstract elements
  - no requirement to return in a finite lattice
  - still benefit from **widening increased precision** at loop heads

Outline

1. Introduction
2. A parametric abstraction
3. Static analysis algorithms
4. Instantiation to the analysis of low-level programs
5. Instantiation to interprocedural analysis
6. Perspectives and implementation

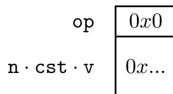
Issues inherent in the analysis of C programs

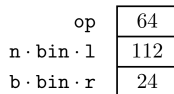
Toward the analysis of a more complex language

- So far, we assumed a **simple, Java-like memory model**
- Yet, we are interested in **complex C applications**...

An abstract syntax tree type

```
typedef struct arith {
    char op;
    union {
        struct{double v; }cst
        struct{
            struct arith * l;
            struct arith * r;
        } bin;
    } n;
} arith;
```





- Nested aggregates
- Size of fields to take into account
- Existence of pointers to fields
- Memory management

- Vincent Laviron's master internship, [ESOP'10]

Abstraction of contiguous regions (II)

- **Points-to edges** are of the form $\alpha \cdot \mathtt{f} \mapsto \beta$
- How to choose instantiate our framework / choose $\mathtt{f}$?

```
typedef struct {
    int a;
    struct {
        int x;
        int y;
    } b;
} tt
```

a	$0x...$
b · x	$0x...$
b · y	$0x...$

int[]

0	$0x...$
4	$0x...$
8	$0x...$

Offsets as sequences

$$\begin{array}{lcll} \mathtt{o} & ::= & \epsilon & \text{null offset} \\ & | & \mathtt{o} \cdot \mathtt{f} & \text{field dereference} \end{array}$$

Numeric offsets

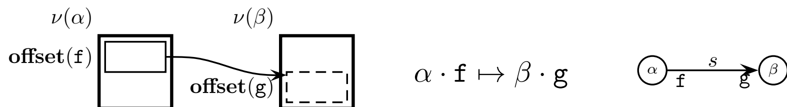
$$\mathtt{o} \in \mathbb{N} \quad \text{numerical value}$$

- **Only offsets** need be modified (instead of **field names**)
- Changes impact **points-to edges only**
- **Inductive structures** and **analysis algorithms** are unchanged

Pointer models

- How to model precisely and concisely **pointers to fields** ?

- Solution 1 (poor): rely on the numerical domain for offset relations
- **Our solution**: label edges destinations with offsets



- **Similar techniques** used by Kreiker for an analysis based on TVLA

Classification, with Pascal Sotin and Bertrand Jeannet [NSAD'10]

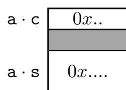
- **Two independent choices**, four models:
  - **numeric or symbolic** offsets
  - **pointers to fields** allowed/disallowed
- **Xisa** allows all four models

Abstracting multiple views

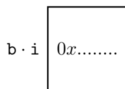
Analysis of unions/pointer casts in real applications

- **Distinct access** may not invalidate other views in user semantics
- How to maintain **several views** on a memory chunk ?

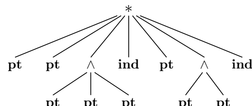
```
struct {
    char c;
    short s;
} a;
```




```
struct {
    int i;
} b;
```



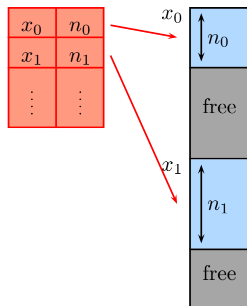
- Mine [LCTES'06], Astrée: analysis maintains **dynamic sets of views**
- Extends to **our abstraction**: **local non separating conjunction** i.e., $\wedge$ applies **only to points-to edges**
- **Limited fragment** of separation logic with inductive structures
  - local conjunctions preserve analysis efficiency



Dynamic memory management

- How to verify the safety of free(p) ?
  Only base addresses of allocated regions can be freed safely

Concrete states:



Static analysis:

- Abstract the allocation table
- Track the location of nodes representing valid addresses
- Track the size and base address of heap allocated regions

Inductive structures also need to summarize such properties

Benefit from the very concrete base semantics

Outline

1 Introduction

2 A parametric abstraction

3 Static analysis algorithms

4 Instantiation to the analysis of low-level programs

5 Instantiation to interprocedural analysis

6 Perspectives and implementation

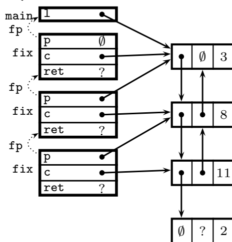
Approaches to interprocedural analysis

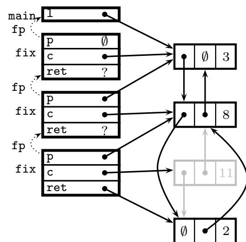
"relational" approach	"inlining" approach
analyze each definition abstracts $\mathcal{P}(\bar{\mathbb{S}} \rightarrow \bar{\mathbb{S}})$	analyze each call abstracts $\mathcal{P}(\mathbb{S})$
+ modularity + reuse of invariants - deals with state relations - complex higher order iteration strategy	- not 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;
▶   c -> next = fix(c -> next, c);
    if(check(c -> data)){
      ret = c -> next;
      remove(c);
▷   } else ret = c;
  }
  return ret;
}
```

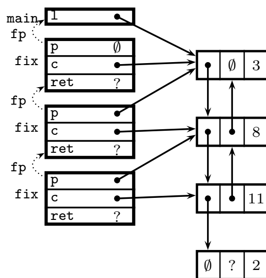
- turns a linked list into a doubly linked list
- removes some elements

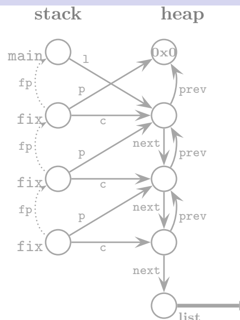




- Heap is unbounded, needs abstraction (shape analysis)
- But stack may also grow unbounded, needs abstraction
- Complex relations between both stack and heap

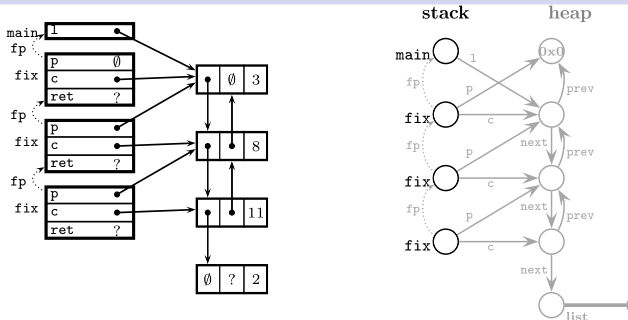
Calling contexts as shape graphs





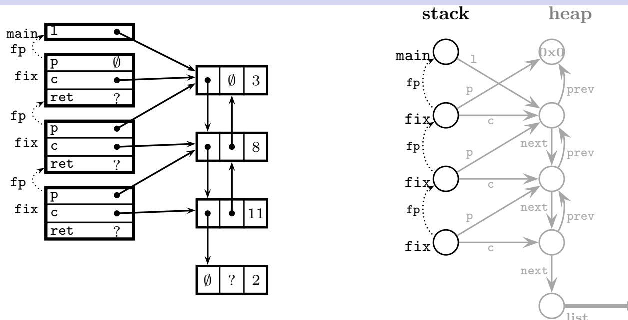
- **Concrete assembly call stack** modelled in a **separating shape graph** together with the **heap**
  - **one node** per **activation record address**

Calling contexts as shape graphs



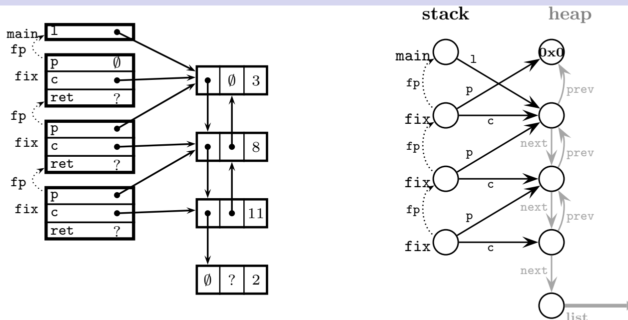
- **Concrete assembly call stack** modelled in a **separating shape graph** together with the **heap**
  - **one node** per **activation record address**

Calling contexts as shape graphs



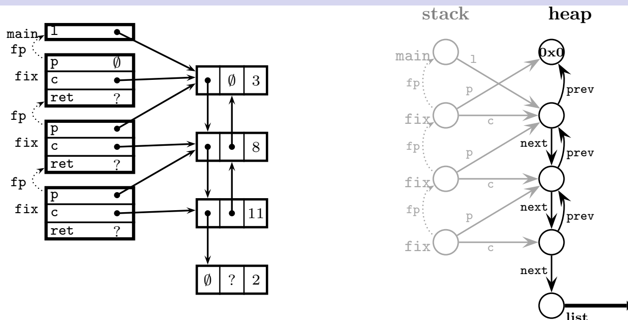
- **Concrete assembly call stack** modelled in a **separating shape graph** together with the **heap**
  - **one node** per **activation record address**
  - **explicit edges** for **frame pointers**

Calling contexts as shape graphs



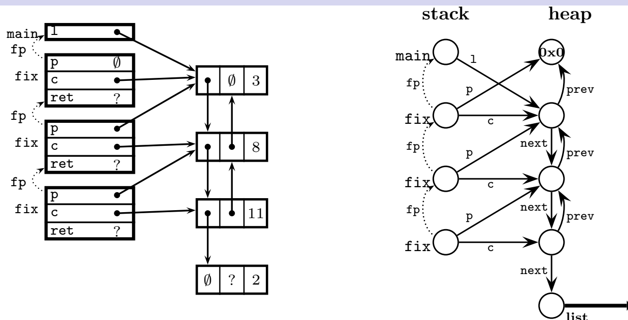
- **Concrete assembly call stack** modelled in a **separating shape graph** together with the **heap**
  - **one node** per **activation record address**
  - **explicit edges** for **frame pointers**
  - **local variables** turn into **activation record fields**

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

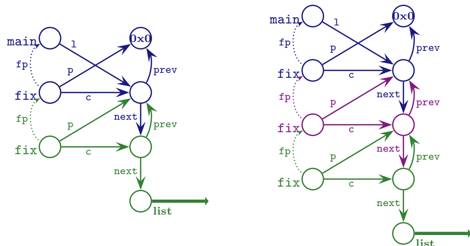
Calling contexts as shape graphs



- **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

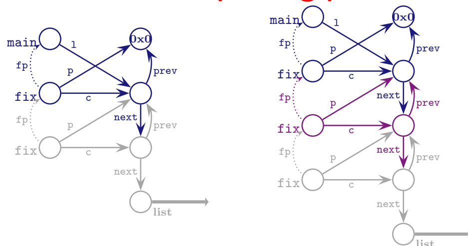
- **Second** and **third iterates**: **a repeating pattern**



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

Inference of a call-stack inductive structure

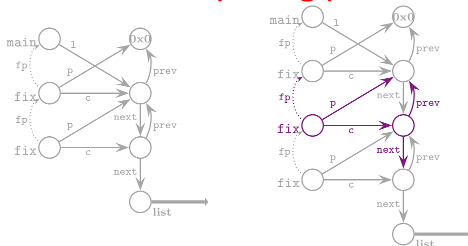
- **Second** and **third iterates**: **a repeating pattern**



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

Inference of a call-stack inductive structure

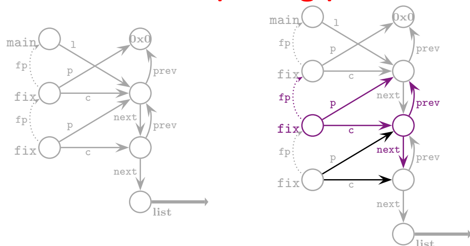
- **Second** and **third iterates**: **a repeating pattern**



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

Inference of a call-stack inductive structure

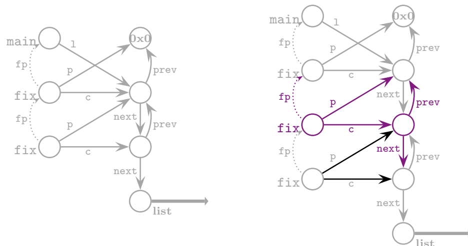
- **Second** and **third iterates**: **a repeating pattern**



- **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**

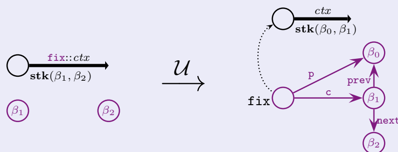
Inference of a call-stack inductive structure

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



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

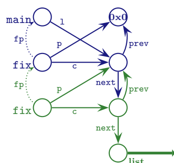
Inferred inductive rule



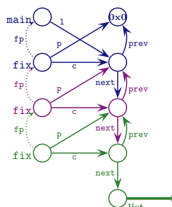
Inference of a call-stack summary: widening iterates

- **Fixpoint at function entry**:

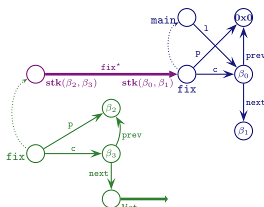
first iterate:



second iterate:



widened iterate:



Fixpoint reached

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

Outline

1. Introduction
2. A parametric abstraction
3. Static analysis algorithms
4. Instantiation to the analysis of low-level programs
5. Instantiation to interprocedural analysis
6. Perspectives and implementation

Foundations of our analysis

- **Concrete level choices have a huge impact** on the analysis
  Components that do no actively contribute to abstraction (summarization)
  - **separation**: allows simpler analysis algorithms
  - **contiguous regions**: very concrete model
- **High expressive power of inductive structures**
  - wide range of structures with or without pointer/numerical relations
  - though, **not adapted for everything**: arrays, graphs...
    (more on this later)
- **Induction in the abstraction and in the analysis**
  - **unfolding** (aka focus, local abstraction): *case analysis*
  - **widening** (aka folding): *reverse* operation and *support for induction*

References

- **Abstraction**: [SAS'07], [POPL'08]
- **Analysis algorithms**: [SAS'07], [POPL'08]
- **C memory model**: [ESOP'10], [NSAD'10]
- **Call stack abstraction**: [POPL'10]

Implementation

Xisa prototype (eXtensible Inductive Shape Analyzer)

- takes user-supplied inductive definitions [SAS'07,POPL'08,ESOP'10]
- infers inductive definitions for call-stack [POPL'11]

Example	size (locs)	time (ms)	peak $\vee$	iters
list reverse	19	7	1	3
list remove element	27	16	4	6
list insertion sort	56	21	4	7
dll copy	50	53	2	3
dll insert	40	38	2	4
binary search tree find	23	10	2	4
binary search tree insert	150	83	5	5
distribution over arithmetic trees	41	144	14	2
move negations up in arith. trees	120	488	38	2
scull driver	894	9 710	16	4

Infering appropriate inductive definitions

Choice of inductive definitions

- **User-supplied** in Xisa
- Ideally: **it should be automatic**

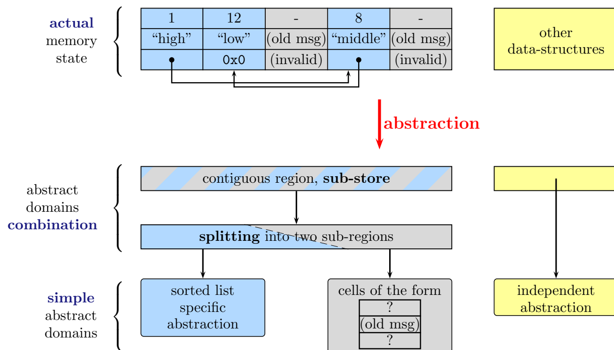
- **Solution 1**: **scan type definitions** and **assume no sharing**
  - e.g., type definition `t`, with **one pointer** to type `t`: assumed **list**...
  - **sound** yet **imprecise**; the analysis will **fail** in presence of sharing
- **Solution 2**: **infer inductive definitions**
  - **generalize** the call-stack inference algorithm
  - should apply well **to constructor code**, e.g. in object oriented programs
  - instance of a **cofibered** abstract domain [Venet, SAS, 1996]
    two levels: one for inductive rules, and one for numerical values

Ongoing works and extensions

- **Decide implication** between **different, folded inductive predicates** an **abstract interpretation** algorithm
- **Internal reduction**:
  convert equivalent abstract predicates
- **Generalize inductive definitions inference scheme**:
  e.g., for object languages, use **constructor code**

Longer term: towards modular heap abstraction

- Notion of **reduced product** (non separating conjunction): Ongoing Master internship by Antoine Toubhans
- **Splitting combination** (separating conjunction)...



- MemCAD project: five years contract, Post-Doc positions