Advanced Memory and Shape Analyses

Matthieu Lemerre, Xavier Rival, Hugo Illous,
Olivier Nicole, Julien Simonnet, Mihaela Sighireanu
Université Paris-Saclay, CEA LIST and INRIA and Département d'informatique de l'ENS

Frama-C Days 2024

Outline

1. Introduction

2. Relational shape analysis based on separation logic

3. Type-based analysis

4. Comparison and conclusion

Why is memory analysis important

Reason 1: Memory is a key program property

Structural invariants on memory are the backbone of the proof in systems programs.

Why is memory analysis important

Reason 1: Memory is a key program property

Structural invariants on memory are the backbone of the proof in systems programs.

“*Much of the kernel-call code is directed at maintaining [data-structure] invariants*”
– Walker et al., *Specification and Verification of the UCLA Unix Security Kernel*, 1980

“*There are four main categories of invariants in our proof: 1. low-level memory invariants, 2. typing invariants, 3. data structure invariants, and 4. algorithmic invariants. [...] 80% of the effort [...] went into establishing invariants.*”
– Klein et al., *Comprehensive Formal Verification of an OS Microkernel*, 2015

Why is memory analysis important

Reason 2: A key safety and cybersecurity property

Memory safety is key for safety and security of systems software

- Memory corruption is what makes C programming painful (crash, complex debugging, etc.)
- Main cybersecurity attack vector (buffer overflows, use-after-free, etc.)

Why is memory analysis important

Reason 2: A key safety and cybersecurity property

Memory safety is key for safety and security of systems software

- Memory corruption is what makes C programming painful (crash, complex debugging, etc.)
- Main cybersecurity attack vector (buffer overflows, use-after-free, etc.)

"70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues." *Microsoft*

"63% of 2019's exploited 0-day vulnerabilities fall under memory corruption." *Google project0*

" Future Software Should Be Memory Safe" *White House Press Release, Feb. 2024*

Why is memory analysis important

Reason 3: General purpose analysis of C

Without a good memory abstraction, the analysis is limited to situations where the abstract state is a finite list of known memory cells.

→ In practice:

- *embedded systems programs* (no dynamic memory allocation, no recursion), and
- *whole-program* analysis (that cannot analyze parts of the program in isolation).

Why is memory analysis important

Reason 3: General purpose analysis of C

Without a good memory abstraction, the analysis is limited to situations where the abstract state is a finite list of known memory cells.

➜ In practice:

- *embedded systems programs* (no dynamic memory allocation, no recursion), and
- *whole-program* analysis (that cannot analyze parts of the program in isolation).

Counter examples:

- Analyzing a function which is not `main` (e.g., a library function).
- Analyzing a program that calls unknown functions pointers, or large/unknown libraries.
- Analyzing a program with an unbounded recursion;
- Analyzing a program that allocates an array with a variable length;
- Analyzing a program that calls `malloc` in a loop;

➜ Ubiquitous situations!

Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\},\ x \in \{0\}$

Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\}, \; x \in \{0\}$

$i \in \{100\}, \; x \in \{0\}$

Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

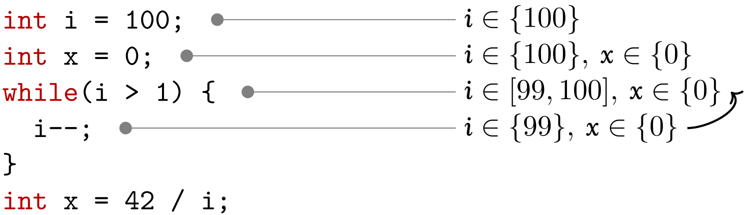
$i \in \{100\}, x \in \{0\}$

$i \in \{100\}, x \in \{0\}$

$i \in \{99\}, x \in \{0\}$

Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.



```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\},\ x \in \{0\}$

$i \in [99, 100],\ x \in \{0\}$

$i \in \{99\},\ x \in \{0\}$

Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

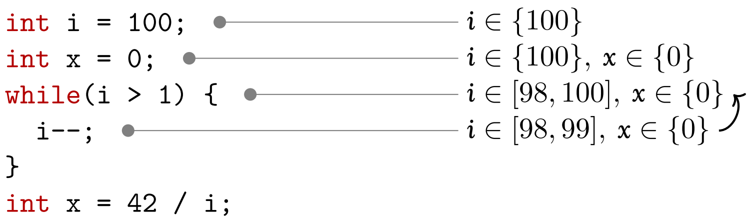
$i \in \{100\}$, $x \in \{0\}$

$i \in [99, 100]$, $x \in \{0\}$

$i \in [98, 99]$, $x \in \{0\}$

Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.



```c
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\}$, $x \in \{0\}$

$i \in [98, 100]$, $x \in \{0\}$

$i \in [98, 99]$, $x \in \{0\}$

Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.

`int i = 100;`	$i \in \{100\}$
`int x = 0;`	$i \in \{100\}$, $x \in \{0\}$
`while(i > 1) {`	$i \in [98, 100]$, $x \in \{0\}$
`  i--;`	$i \in [97, 99]$, $x \in \{0\}$
`}`	
`int x = 42 / i;`	

Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.

```c
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\}$, $x \in \{0\}$

$i \in [2, 100]$, $x \in \{0\}$

$i \in [1, 99]$, $x \in \{0\}$

Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.

```c
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\}, \ x \in \{0\}$

$i \in [2, 100], \ x \in \{0\}$

$i \in [1, 99], \ x \in \{0\}$

$i \in \{1\}, \ x \in \{0\}$

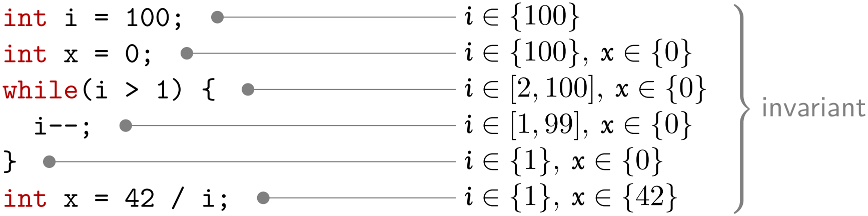
Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.

`int i = 100;`	$i \in \{100\}$
`int x = 0;`	$i \in \{100\}, x \in \{0\}$
`while(i > 1) {`	$i \in [2, 100], x \in \{0\}$
`  i--;`	$i \in [1, 99], x \in \{0\}$
`}`	$i \in \{1\}, x \in \{0\}$
`int x = 42 / i;`	$i \in \{1\}, x \in \{42\}$

Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.



```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\},\ x \in \{0\}$

$i \in [2, 100],\ x \in \{0\}$

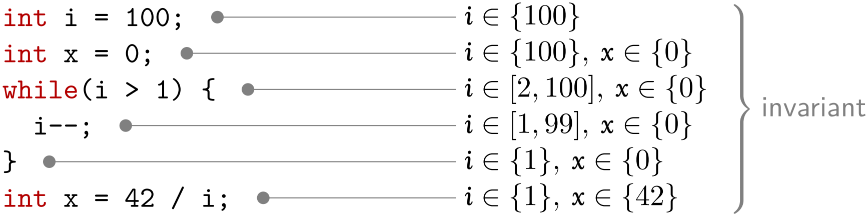
$i \in [1, 99],\ x \in \{0\}$

$i \in \{1\},\ x \in \{0\}$

$i \in \{1\},\ x \in \{42\}$

Abstract interpretation basics (example: interval analysis)

Abstracts each *numeric variable* by an *interval* that *over-approximate* its possible values.



```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\},\ x \in \{0\}$

$i \in [2, 100],\ x \in \{0\}$

$i \in [1, 99],\ x \in \{0\}$

$i \in \{1\},\ x \in \{0\}$

$i \in \{1\},\ x \in \{42\}$

invariant

- Abstract interpretation automatically infers invariants of the program.
- This can be used to *automatically prove* program properties.
  - Here: no division by zero (an example of *run-time error*)
  - Other runtime errors: integer overflow
  - Memory-related errors: buffer overflow, null pointer dereference, invalid cast...

Numerical and memory abstractions

Abstract interpretation = *automatic* computation of *abstractions* representing program properties.

Some numeric abstractions:

Example (Intervals)

```
// $x \in [-9, 4]$
y := x * x
// $x \in [-9, 4] \land y \in [0, 81]$
```

Example (Linear equations)

```
//
y := x * x
// $y \geq x \land y \geq -x$
```

What about memory abstractions? A hard, unsolved problem:

> *However, while for numerical domains, we have nice open-source libraries that can easily be embedded into larger use-cases, it was noted that this is hardly the case for [data structures] domains*
>
> – *Dagstuhl seminar on Theoretical Advances and Emerging Applications in Abstract Interpretation*

➜How to structure and interface with memory abstractions?

Three structures of abstract interpreters

```c
struct point { int x; int y; } p;
```

Non-relational		
$\mathtt{p} \mapsto \{\mathrm{x}{=}[1{-}3], \mathrm{y}{=}[2{-}4]\}$		
Value, TIS-Analyzer		
– No relations		
+ Expr-composable $[1{-}3] + [2{-}4] = [3{-}7]$		
+ Known structure		

$$\mathrm{p.x} := \mathrm{p.x} + \mathrm{p.y}$$

$\mathtt{p} \mapsto \{\mathrm{x}{=}[3{-}7], \mathrm{y}{=}[2{-}4]\}$		

Value-based analysis allows clean separation between the numeric and memory abstractions.

Three structures of abstract interpreters

```
struct point { int x; int y; } p;
```

Non-relational	Variable/Lvalue-based	
$p \mapsto \{x=[1-3], y=[2-4]\}$	$p.x \in [1-3] \wedge p.y \in [2-4]$	
Value, TIS-Analyzer	Eva, Astrée,MOPSA	
– No relations	+ Relations between vars	
+ Expr-composable $[1-3] + [2-4] = [3-7]$	– Not expr-composable $p.x + p.y = ?$	
+ Known structure	+ Known structure	

$$p.x := p.x + p.y$$

$p \mapsto \{x=[3-7], y=[2-4]\}$	$p.x \in [3-7] \wedge p.y \in [2-4]$	

Value-based analysis allows clean separation between the numeric and memory abstractions.

Three structures of abstract interpreters

```
struct point { int x; int y; } p;
```

Non-relational	Variable/Lvalue-based	Value/SSA-based
$\mathtt{p} \mapsto \{\mathtt{x}{=}[1{-}3], \mathtt{y}{=}[2{-}4]\}$	$\mathtt{p.x} \in [1{-}3] \wedge \mathtt{p.y} \in [2{-}4]$	$\mathtt{p} \mapsto \{\mathtt{x}{=}\alpha, \mathtt{y}{=}\beta\} \wedge$ $\alpha \in [1{-}3] \wedge \beta \in [2{-}4]$
Value, TIS-Analyzer	Eva, Astrée,MOPSA	Codex, RMA, MemCAD
– No relations	+ Relations between vars	+ Relations between α
+ Expr-composable	– Not expr-composable	+ Expr-composable
$[1{-}3] + [2{-}4] = [3{-}7]$	$\mathtt{p.x} + \mathtt{p.y} = ?$	$\alpha + \beta = (\alpha + \beta)$
+ Known structure	+ Known structure	– Need study

$$\mathtt{p.x} := \mathtt{p.x} + \mathtt{p.y}$$

$\mathtt{p} \mapsto \{\mathtt{x}{=}[3{-}7], \mathtt{y}{=}[2{-}4]\}$	$\mathtt{p.x} \in [3{-}7] \wedge \mathtt{p.y} \in [2{-}4]$	$\mathtt{p} \mapsto \{\mathtt{x}{=}\alpha + \beta, \mathtt{y}{=}\beta\} \wedge$ $(\alpha + \beta) \in [3{-}7] \wedge \beta \in [2{-}4]$ $\wedge\ \alpha \in [1{-}3]$

Value-based analysis allows clean separation between the numeric and memory abstractions.

Variable vs value-based analysis for memory

Example (Microsoft CheckedC)

```
int n;
int *ptr : count(n);
if(...)
  n++; // ptr: count(n-1)
else if(...)
  n = n * n; //ptr: count(sqrt(n))

else if(...)
  n = n / 2; // ptr: count(2*n) ||
             // ptr:(count(2*n+1))
else if(...)
  ptr++ // ptr: count(n-1);
}
```

- Complicated relation
  - ➜ forbid changing `ptr` or `n`

Example (Codex)

```
(int with self = α) n;
int[α]* ptr;
if(...)
  n++; // ptr:int[α]*, n = α + 1
else if(...)
  n = n * n; // ptr:int[α]*
             // n = α * α
else if(...)
  n = n / 2; // ptr:int[α]*
             // n = α / 2
else if(...)
  ptr++ // ptr:int[α]* + 1,n = α
}
```

- Simple relation
- Memory abstraction not involved

Problem: renaming values when joining execution paths

Example

```
// $p \mapsto \{x = \alpha, y = \beta\}$
if(...) {
  p.x := p.x + 1;
  // $p \mapsto \{x = \alpha + 1, y = \beta\}$
 }
else {
 p.x := p.x;
  // $p \mapsto \{x = \alpha, y = \beta\}$
 }
  // $p \mapsto \{x = \phi(\alpha, \alpha + 1), y = \beta\}$
```

- What does ϕ mean? Is this related to the ϕ of SSA? (Was unclear for several years...)
  - Recent advances: SSA Translation Is an Abstract Interpretation [POPL 2023], Compiling with Abstract Interpretation [PLDI 2024]
  - ➜ Allows future integration of Codex memory domains in Eva

Outline

1. Introduction

2. **Relational shape analysis based on separation logic**

3. Type-based analysis

4. Comparison and conclusion

State vs transformations

- State analyses: Computes a set of reachable states to verify state properties:
  - Can this program perform a null pointer dereference?
  - Does this program preserve structural invariants of data structures?

State vs transformations

- State analyses: Computes a set of reachable states to verify state properties:
  - Can this program perform a null pointer dereference?
  - Does this program preserve structural invariants of data structures?
- Transformation analyses: Compute abstract transformations, i.e. relations between program input state and output state:
  - Does this program modify the linked list received as an argument?
  - Is this sorting algorithm in-place?

State vs transformations

- State analyses: Computes a set of reachable states to verify state properties:
  - Can this program perform a null pointer dereference?
  - Does this program preserve structural invariants of data structures?
- Transformation analyses: Compute abstract transformations, i.e. relations between program input state and output state:
  - Does this program modify the linked list received as an argument?
  - Is this sorting algorithm in-place?

The RMA (Relational Memory Analysis) plugin

- Abstract transformations as procedure summaries

State vs transformations

- State analyses: Computes a set of reachable states to verify state properties:
  - Can this program perform a null pointer dereference?
  - Does this program preserve structural invariants of data structures?
- Transformation analyses: Compute abstract transformations, i.e. relations between program input state and output state:
  - Does this program modify the linked list received as an argument?
  - Is this sorting algorithm in-place?

The RMA (Relational Memory Analysis) plugin

- Abstract transformations as procedure summaries
- Applied to shape analysis using separation logic.

Overview

```
double_append(list* k0,list* k1,list* k2){



  append(k0,k1);



  append(k0,k2);



}

append(list* l0,list* l1){


  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

State analysis by inlining

```
double_append(list* k_0,list* k_1,list* k_2){
```

$h^\sharp_0$

```
  append(k_0,k_1);


  append(k_0,k_2);


}
append(list* l_0,list* l_1){


  while(l_0→n≠0x0){l_0=l_0→n;}
  l_0→n = l_1;
```

State analysis by inlining

```
double_append(list* k0,list* k1,list* k2){
```

$h_0^\sharp$

```
  append(k0,k1);


  append(k0,k2);


}
append(list* l0,list* l1){
```

$h_1^\sharp$

```
  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

State analysis by inlining

```
double_append(list* k0,list* k1,list* k2){
```

$h^\sharp_0$

```
  append(k0,k1);



  append(k0,k2);



}
```

```
append(list* l0,list* l1){
```

$h^\sharp_1$

```
  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

$h^\sharp_{19}$

State analysis by inlining

```
double_append(list* k0,list* k1,list* k2){
```

$$h^{\sharp}_{0}$$

```
  append(k0,k1);
```

$$h^{\sharp}_{20}$$

```
  append(k0,k2);



}
```

```
append(list* l0,list* l1){


  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

State analysis by inlining

```
double_append(list* k0,list* k1,list* k2){
```

$h^\sharp_0$

```
  append(k0,k1);
```

$h^\sharp_{20}$

```
  append(k0,k2);



}
```

```
append(list* l0,list* l1){
```

$h^\sharp_{21}$

```
  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

State analysis by inlining

```
double_append(list* $k_0$,list* $k_1$,list* $k_2$){
                        $h^\sharp_0$
  append($k_0$,$k_1$);
                        $h^\sharp_{20}$
  append($k_0$,$k_2$);


}
append(list* $l_0$,list* $l_1$){
                                  $h^\sharp_{21}$
  while($l_0$→n≠0x0){$l_0$=$l_0$→n;}
  $l_0$→n = $l_1$;
                                  $h^\sharp_{39}$
```

State analysis by inlining

```
double_append(list* k0,list* k1,list* k2){

                      $h^{\sharp}_{0}$

  append(k0,k1);

                      $h^{\sharp}_{20}$

  append(k0,k2);

                      $h^{\sharp}_{40}$

}

append(list* l0,list* l1){


  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

State analysis by inlining

```
double_append(list* k0,list* k1,list* k2){


  append(k0,k1);


  append(k0,k2);


}

append(list* l0,list* l1){


  while(l0→n≠0x
  l0→n = l1;
```

- + Precise analysis of procedures
- – Analysis of append is repeated for each calling context
- – Cannot handle recursive procedures

Abstract transformations as procedure summaries

```
double_append(list* k_0,list* k_1,list* k_2){


  append(k_0,k_1);


  append(k_0,k_2);


}
append(list* l_0,list* l_1){


  while(l_0→n≠0x0){l_0=l_0→n;}
  l_0→n = l_1;
```

Abstract transformations as procedure summaries

```
double_append(list* k0,list* k1,list* k2){


  append(k0,k1);


  append(k0,k2);


}

append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

$\downarrow t^\sharp$

Abstract transformations as procedure summaries

```
double_append(list* k0,list* k1,list* k2){
```

$h_0^\sharp$

```
  append(k0,k1);


  append(k0,k2);


}

append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

$\Big\downarrow t^\sharp$

Abstract transformations as procedure summaries

```
double_append(list* k0,list* k1,list* k2){
```

$$h_0^\sharp$$

```
  append(k0,k1);
```

$$h_1^\sharp = t^\sharp(h_0^\sharp)$$

```
  append(k0,k2);



}
append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

$$\Big\downarrow t^\sharp$$

Abstract transformations as procedure summaries

```
double_append(list* $k_0$,list* $k_1$,list* $k_2$){
                         $h_0^\sharp$
  append($k_0$,$k_1$);
                         $h_1^\sharp = t^\sharp(h_0^\sharp)$
  append($k_0$,$k_2$);
                         $h_2^\sharp = t^\sharp(h_1^\sharp)$
}
append(list* $l_0$,list* $l_1$){

  while($l_0$→n≠0x0){$l_0$=$l_0$→n;}  $\Big\downarrow t^\sharp$
  $l_0$→n = $l_1$;
```

Abstract transformations as procedure summaries

```
double_append(list* k0,list* k1,list* k2){


  append(k0,k1);


  append(k0,k2);


}

append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;} | t♯
  l0→n = l1;
```

- Applying an abstract transformation can speed up a state analysis.

Modular analysis by composition of abstract transformations [SAS 2020, FMSD 2021]

```
double_append(list* k0,list* k1,list* k2){


  append(k0,k1);


  append(k0,k2);


}
append(list* l0,list* l1){


  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

Modular analysis by composition of abstract transformations [SAS 2020, FMSD 2021]

```
double_append(list* k0,list* k1,list* k2){


  append(k0,k1);


  append(k0,k2);


}
append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

$\Big\downarrow t^\sharp$

Modular analysis by composition of abstract transformations [SAS 2020, FMSD 2021]

```
double_append(list* k0,list* k1,list* k2){
```

$\downarrow \mathtt{Id}(h_0^\sharp)$

```
  append(k0,k1);


  append(k0,k2);


}
append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

$\downarrow t^\sharp$

Modular analysis by composition of abstract transformations [SAS 2020, FMSD 2021]

```
double_append(list* k0,list* k1,list* k2){

  append(k0,k1);

  append(k0,k2);

}

append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

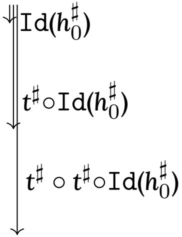
$\Downarrow \mathtt{Id}(h_0^\sharp)$

$t^\sharp \circ \mathtt{Id}(h_0^\sharp)$

$t^\sharp$

Modular analysis by composition of abstract transformations [SAS 2020, FMSD 2021]

```
double_append(list* k0,list* k1,list* k2){

  append(k0,k1);

  append(k0,k2);

}
```

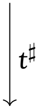


$\mathtt{Id}(h_0^\sharp)$

$t^\sharp \circ \mathtt{Id}(h_0^\sharp)$

$t^\sharp \circ t^\sharp \circ \mathtt{Id}(h_0^\sharp)$

```
append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```



$t^\sharp$

Modular analysis by composition of abstract transformations [SAS 2020, FMSD 2021]

```
double_append(list* k0,list* k1,list* k2){


  append(k0,k1);


  append(k0,k2);


}
append(list* l0,list* l1){

  while(l0→n≠0x
  l0→n = l1;
```

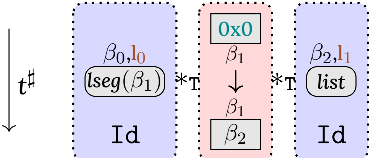
- Composition of relations can produce a new summary from summaries of callee functions.
- Summary was created for a given input state (context)

Modular analysis by composition of abstract transformations [SAS 2020, FMSD 2021]

```
double_append(list* k0,list* k1,list* k2){
```

$$\overset{\alpha_0,k_0}{\boxed{lseg(\alpha_1)}} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1}{\boxed{lseg(\alpha_3)}} * \overset{\alpha_3}{\boxed{0x0}} * \overset{\alpha_4,k_2}{\boxed{list}}$$

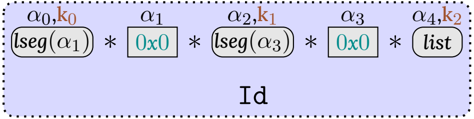
```
  append(k0,k1);


  append(k0,k2);


}

append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```



Modular analysis by composition of abstract transformations [SAS 2020, FMSD 2021]

```
double_append(list* k0,list* k1,list* k2){
```

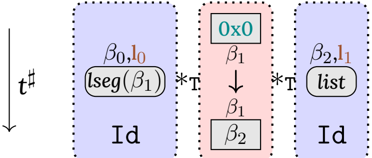
$\downarrow \mathtt{Id}(h_0^\sharp)$



α_0,k0 $\mathit{lseg}(\alpha_1)$ $*$ α_1 0x0 $*$ α_2,k1 $\mathit{lseg}(\alpha_3)$ $*$ α_3 0x0 $*$ α_4,k2 *list*

Id

```
  append(k0,k1);

  append(k0,k2);

}
append(list* l0,list* l1){
  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

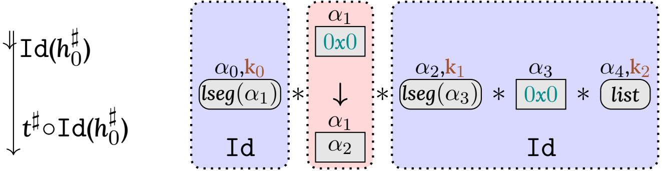


$t^\sharp$

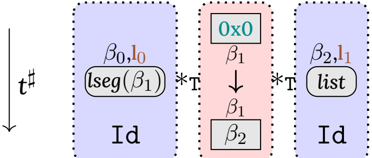
β_0,l0 $\mathit{lseg}(\beta_1)$ Id $*_{\mathrm{T}}$ 0x0 β_1 $\downarrow$ β_1 β_2 $*_{\mathrm{T}}$ β_2,l1 *list* Id

Modular analysis by composition of abstract transformations [SAS 2020, FMSD 2021]

```
double_append(list* k0,list* k1,list* k2){

  append(k0,k1);

  append(k0,k2);


}
```

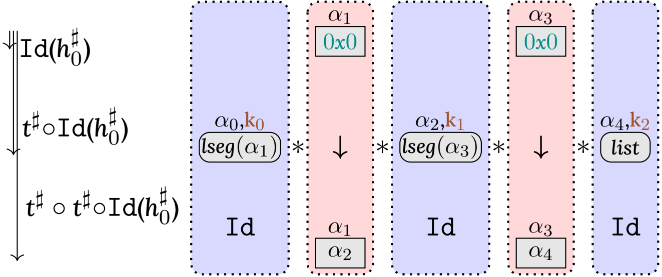


```
append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

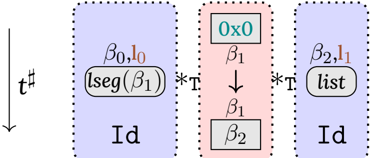


Modular analysis by composition of abstract transformations [SAS 2020, FMSD 2021]

```
double_append(list* k0,list* k1,list* k2){
  append(k0,k1);
  append(k0,k2);
}
```



```
append(list* l0,list* l1){
  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```



Outline

1 Introduction

2 Relational shape analysis based on separation logic

3 Type-based analysis

4 Comparison and conclusion

Separation logic vs type-based analysis

Shape analysis based on separation logic

- Allows very precise invariant
- Only need to provide initial state and inductive predicates (list...)
- Sometimes matching may fail...
- Need to reason about whether lists are separated or cyclic

Type-based analysis

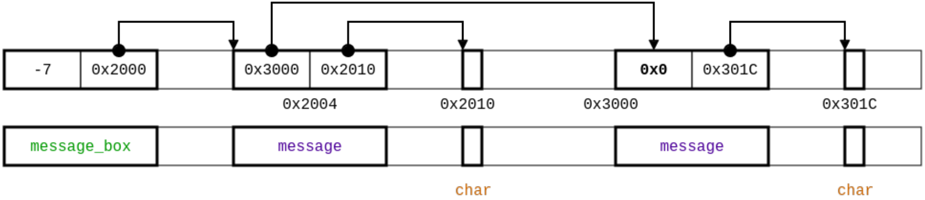
- Weaker invariant
- But need to know less about the memory invariants
- Type-based analysis: lightweight formal method
  - Goal: prove absence of undefined behaviour, including memory corruption
  - Per-function analysis that can scale to large program
  - Without rewriting in Rust or annotating the function body

Record & array types

Types represent a memory layout. **Record types** $\tau_1 \times \tau_2$ and **array types** $\tau[e]$ concatenate types.

```
def int := byte[4]
def char := byte

def message :=
   message* ×
   char*

def message_box :=
  int ×
  message*
```

```
// corresponding C type
//

struct message {
  struct message *next;
  char *buffer };

struct message_box {
  int length;
  struct message *first };
```

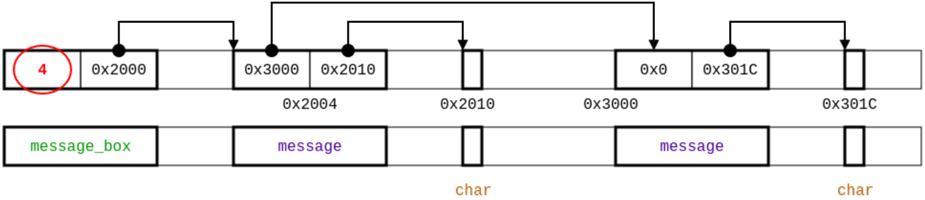


Refinement types

Types also represent values. Values in a **refinement type** τ `with` p fulfill predicate p.

```
def int := byte[4]
def char := byte

def message :=
   message* ×
   char*

def message_box :=
  byte[4] with self >= 0 ×
  message*
```

```
// corresponding C type
//

struct message {
  struct message *next;
  char *buffer };

struct message_box {
  int length;
  struct message *first };
```

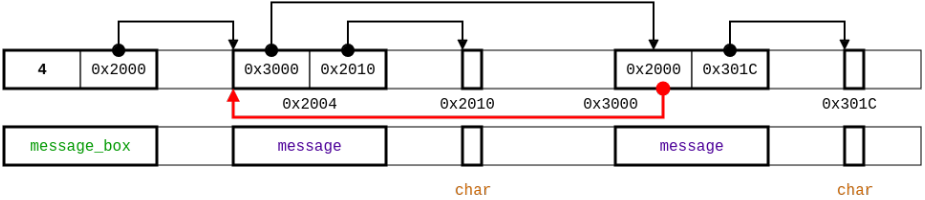


Non-null pointer types

Pointer types $\eta\star$ denote **non null** addresses. Actually, $\eta*$ is $\eta\star \cup (\mathtt{byte}[\mathcal{W}]$ with self $= 0)$

```
def int := byte[4]
def char := byte

def message :=
   message⋆ ×
   char⋆

def message_box :=
  byte[4] with self >= 0 ×
  message⋆
```

```
// corresponding C type
//

struct message {
  struct message *next;
  char *buffer };

struct message_box {
  int length;
  struct message *first };
```

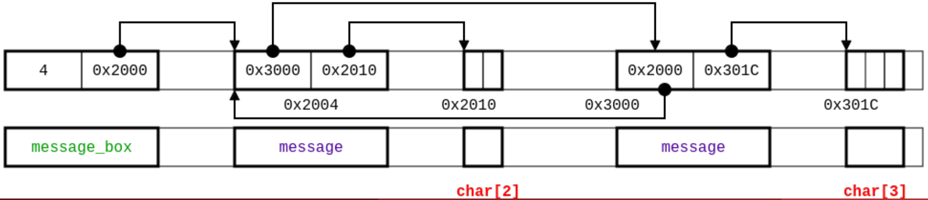


Existential types

Existential types: $\exists\alpha : \tau_1.\tau_2$ bind in τ_2 a **symbolic variable** α of type τ_1.

```
def int := byte[4]
def char := byte

def message :=
∃ len:byte[4] with self >= 0.
   message⋆ ×
   char[len]⋆

def message_box :=
  byte[4] with self >= 0 ×
  message⋆
```

```
// corresponding C type
//

struct message {
  struct message *next;
  char *buffer };

struct message_box {
  int length;
  struct message *first };
```

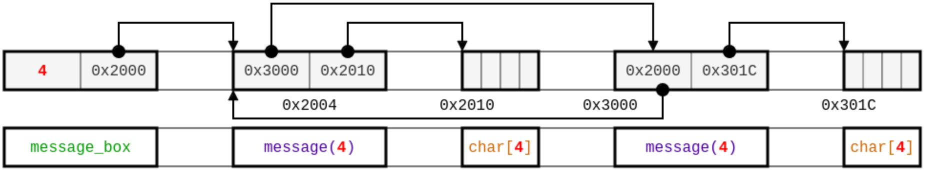


Parameterized types

Parameterized types $n(e_1, \ldots, e_\ell)$ use symbolic variables as parameters.

```
def int  := byte[4]
def char  := byte

def message(len:int) :=
   message(len)⋆ ×
   char[len]⋆

def message_box :=
∃ mlen:byte[4] with self >= 0.
  byte[4] with self = mlen ×
  message(mlen)⋆
```

```
// corresponding C type
//

struct message {
  struct message *next;
  char *buffer };

struct message_box {
  int length;
  struct message *first };
```



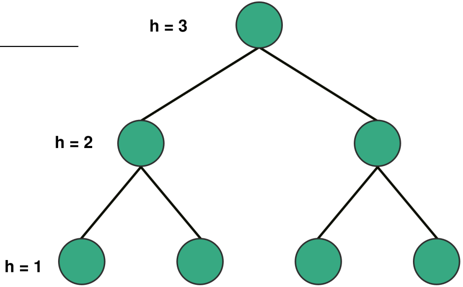
Union types

Union types $\tau_1 \cup \tau_2$ specify that a value belongs to one type τ_i or both.

```
nullptr ≜ byte[W] with self=0

def node(h:byte[4]) :=
  byte[4] ×
  ( (node(h-1)⋆ × node(h-1)⋆) with h > 1
  ∪ (nullptr × nullptr) with h <= 1)

def nodeptr :=
  ∃ h:byte[4] with self > 0. node(h)⋆
```

```c
struct node {
  int value;
  struct node *left;
  struct node *right;
};
```

This specifies a perfect binary tree



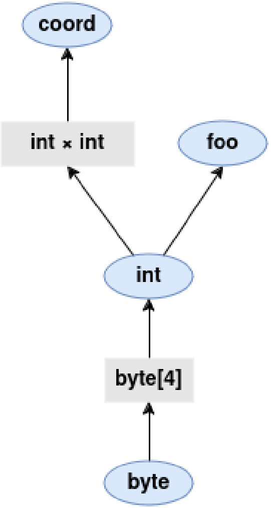
Nominal type system

Pointer types $\eta\star$ point to a **region name** η

```
def int := byte[4]
def coord := int × int
def foo := int
```

Derivation rules

- $(\!|\texttt{coord}\star|\!) \subseteq (\!|\texttt{int}\star|\!)$
- $(\!|\texttt{foo}\star|\!) \subseteq (\!|\texttt{int}\star|\!)$
- $(\!|\texttt{coord}\star|\!) \cap (\!|\texttt{foo}\star|\!) = \emptyset$



Outline

1. Introduction

2. Relational shape analysis based on separation logic

3. Type-based analysis

4. Comparison and conclusion

Conclusion

- Memory analysis : a fundamental problem that limits applicability of static analysers
- Was an exploratory solution now with a strong theory and fast becoming mature
  - Codex: a library of composable memory abstractions
  - Plan on integrating more with Eva analysis
- Important applications:
  - Modular analysis (integrate abstract interpretation during the development phase, analysis of libraries...)
  - Automated analysis of programs with complex memory invariants
  - Automated verification of memory safety
- Different memory abstractions
  - Separation logic: for precise invariant on data structures
  - Types: a simpler, scalable analysis that handles low-level code
  - Future work: integrate separation logic and type-based analysis