What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Even C is hell

Conclusion

Proving the security of an embedded operating system using abstract interpretation

Marc Chevalier

April 29, 2018

What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Even C is hell

Conclusion

What it's all about?
The need of certifications
What we prove
What I want to prove
Abstract interpretation saves the day
Add assembly into the soup
Even C is hell
Conclusion

What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Even C is hell

Conclusion

The need of certifications – Cost of software failure

Bugs have various annoying consequences:

- Deaths (Patriot, Toyota)
- A lot of money: Ariane V, $60 billion/year in the US
- Privacy
- …

The need of certifications – What we usually do

How developers think they can avoid bugs:

- High level/safe language
- Tests
- Strict code style

Still, Ariane V crashed. . . . "And here, poor fool[s], with all [their] lore, [they] stand no wiser than before".

What we prove

Usually, no runtime error:

- Signed integer overflow
- Out of bound access
- Invalid pointer dereference
- ...

Better:

- The result satisfies some property
- The execution path does not depend on some secret data

What it's all about?
The need of certifications
What we prove
What I want to prove
Abstract interpretation saves the day
Add assembly into the soup
Even C is hell
Conclusion

What I want to prove

Study case: the OS of an host platform in planes at the border between trusted (flight control) and untrusted (potentially hostile) world.

We want to prove some security properties: memory isolation, hosted applications don't get more privileges. . . .

Properties are not visible from C (check some CPU's registers, mainly): inline assembly $\Rightarrow$ analyze assembly.

What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Even C is hell

Conclusion

What it's all about?

Abstract interpretation saves the day

Introduction

An example

Let's generalize

The incompleteness

Other domains

Add assembly into the soup

Even C is hell

Conclusion

Introduction

- Check an execution: test, limited.
- Check all executions at once: ok, but not computable.
- Check an over-approximation of all execution: sound, not complete.

What it's all about?

Abstract interpretation saves the day

- Introduction
- **An example**
- Let's generalize
- The incompleteness
- Other domains

Add assembly into the soup

Even C is hell

Conclusion

An example

```
int f(int x)
{                          // x ∈ [−2^31, 2^31 − 1]
    int y = abs(x);        // y ∈ [0, 2^31 − 1] ∨ x = −2^31
    int z = y + 1;         // z ∈ [1, 2^31 − 1] ∨ y = 2^31 − 1
    return 1/z;            // 0 ∉ [1, 2^31 − 1] ⇒ OK!
}
```

What it's all about?

Abstract interpretation saves the day

Introduction

An example

Let's generalize

The incompleteness

Other domains

Add assembly into the soup

Even C is hell

Conclusion

Let's generalize

$(D, \subseteq, \wedge, \vee, \bot, \top)$ a too big complete lattice (typically, set of memory environments).

$$[\![P]\!] = f_1 \circ \cdots \circ f_n$$

We want that $c \subseteq$ *specification* holds at every program point.

What it's all about?

Abstract interpretation saves the day
- Introduction
- An example
- Let's generalize
- The incompleteness
- Other domains

Add assembly into the soup

Even C is hell

Conclusion

Let's generalize

Abstract domain:

- $(D^\sharp, \subseteq^\sharp, \wedge^\sharp, \vee^\sharp, \perp^\sharp, \top^\sharp)$: complete lattice (eg. $\overline{\mathbb{Z}}^2$)
- $\gamma : D^\sharp \to D$: concretization (eg. $(a, b) \mapsto \{x \in \mathbb{Z} \mid a \leqslant x \leqslant b\}$)

Sound if for all program point, $c \subseteq \gamma(a)$: we don't miss any behavior by executing in the abstract (but we lose precision).

Sound abstract operator: $f_i \circ \gamma \subseteq \gamma \circ f_i^\sharp$.

And we want $\gamma(a) \subseteq$ *specification*.

The incompleteness

```c
/*@ requires -10 <= x <= 10; */
int g(int x)
{                          // x ∈ [−10, 10]
    int y = x;             // y ∈ [−10, 10]
    int z = x * y;
    /* z ∈ Interval({a × b | a ∈ [−10, 10], b ∈ [−10, 10]})
    z ∈ [−100, 100]
    */
    int t = z + 1;         // t ∈ [−99, 101]
    return 1/t;            // 0 ∈ [−99, 101] ⇒ Alarm!
}
```

But this program is clearly safe.
What happens? This abstract domain cannot understand the relation between x and y.

Other domains

- Numerical:
  Non relational:
  - Modulo: $x_i \equiv c_i[n_i]$
  - Bitwise: $x_i = 0?1??100010111????$
  - Sign: $x_i < 0$, $x_i > 0$, $x_i \leqslant 0 \dots$

  Relational:
  - Polytope: $\sum a_i x_i \leqslant c_i$
  - Octagon: $\pm x_i \pm x_j \leqslant c_i$

  And combination of domains.
- Memory: some value points to another, memory structures, separation logic. . . .
- Partitioning: $(x > 0 \Rightarrow \dots) \wedge (x \leqslant 0 \Rightarrow \dots)$

What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Even C is hell

Conclusion

Back on security

OS $\Rightarrow$ assembly (Intel x86).

Some properties:

- Memory isolation $\Rightarrow$ register CR3 are correctly set and not modified (paging).
- "Sandboxing" $\Rightarrow$ applications stay in ring 3.
- Static code $\Rightarrow$ a writable segment never become executable.

What it's all about?
Abstract interpretation saves the day
Add assembly into the soup
Back on security
Inline assembly
Difficulties
Computing the destination
Local jumps
Distant and return jumps
Mixed calls
Even C is hell
Conclusion

Inline assembly

```
int a;
void f()
{
    // C code
    asm {
        ; assembly code
        mov a, 4
    }
}
```

What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Back on security

Inline assembly

Difficulties

Computing the destination

Local jumps

Distant and return jumps

Mixed calls

Even C is hell

Conclusion

Difficulties

Majority of C: need to analyze x86 in a analysis designed for C.

Why x86 is really different from C:

- Jumps across functions vs local goto and blocks,
- Computed jump destinations vs static CFG,
- Type-agnostic registers vs statically typed programs,
- Intensive usage of stack, register... vs independent from architecture and implementation.

What it's all about?
Abstract interpretation saves the day
Add assembly into the soup
Back on security
Inline assembly
Difficulties
Computing the destination
Local jumps
Distant and return jumps
Mixed calls
Even C is hell
Conclusion

Difficulties – Control Flow

Let's take a look at the control flow problem.

C: cfg, a lot of structured control flow (while, for, if...), gotos
x86: basically, only jumps. (For experts: only near/short jmp/call)

Problems (in increasing difficulty):

- Compute jumps local to a C function.
- Compute the destination.
- Compute jumps leading to anywhere else.

What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

- Back on security
- Inline assembly
- Difficulties
- **Computing the destination**
- Local jumps
- Distant and return jumps
- Mixed calls

Even C is hell

Conclusion

Computing the destination

```
mov EBX, 0
mov EAX, label
add EAX, 3
jmp EAX
label:
add EBX, 1   ; This instruction has 3 bytes: 83 C3 01
add EBX, 2
; Here EBX == 2
```

Computing the destination

Program point: (block number, statement number). An instruction. Useful in analysis
Code pointer: (label, offset) where the label and the offset can be imprecise. An address. How the assembly works.

We have to compute the byte length of each assembly instruction to reinterpret code pointer as program point.

A jump in the middle of an instruction is considered as an error.

What it's all about?
Abstract interpretation saves the day
Add assembly into the soup
Back on security
Inline assembly
Difficulties
Computing the destination
Local jumps
Distant and return jumps
Mixed calls
Even C is hell
Conclusion

Computing the destination

```
void f() {
    register int p;
    asm {
        mov p, [ESP+4] ; return address
    }
    int n = (p-zero)/(one-zero); // call number
}
void h() {
    asm {
        zero: call f
        one: call f
        ...
        call f
    }
}
```

Local jumps

```
int f()
{
    int x = 1;  // x = 1
    goto l;     // ⊥,    l ↦ {x = 1}
    m:          // ...
    return x;   // ...
    l:          // x = 1
    goto m;     // ⊥,    m ↦ {x = 1}
}               // ⊥
```

What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

- Back on security
- Inline assembly
- Difficulties
- Computing the destination
- **Local jumps**
- Distant and return jumps
- Mixed calls

Even C is hell

Conclusion

Local jumps

```
int f()
{
	int x = 1; // x = 1, m ↦ {x = 1}
	goto l;    // ⊥,    l ↦ {x = 1}, m ↦ {x = 1}
	m:         // x = 1, l ↦ {x = 1}
	return x;  // return = 1, l ↦ {x = 1}
	l:         // x = 1
	goto m;    // ⊥,    m ↦ {x = 1}
}              // return = 1
```

What it's all about?
Abstract interpretation saves the day
Add assembly into the soup
Back on security
Inline assembly
Difficulties
Computing the destination
Local jumps
Distant and return jumps
Mixed calls
Even C is hell
Conclusion

Local jumps – Termination

We have a super-union that accelerate convergence: widening.

- sound: $\forall (a, b) \in {D^\sharp}^2, \gamma(a) \cup \gamma(b) \subseteq \gamma(a \nabla b)$
- termination: for all sequence $(a_n) \in {D^\sharp}^{\mathbb{N}}$, the sequence

$$b_0 = a_0$$
$$b_{n+1} = b_n \nabla a_{n+1}$$

is stationary.

Make sure there is at least a widening (and not only abstract union) when we iterate until fixpoint.

What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

- Back on security
- Inline assembly
- Difficulties
- Computing the destination
- Local jumps
- **Distant and return jumps**
- Mixed calls

Even C is hell

Conclusion

Distant and return jumps

A bit of context:

- No recursion (call stack abstraction).
- Inlined analysis (functions always return where they were called).

2 kinds of jumps:

- To a new function (not in the stack): distant jump.
- To a function which is in the call stack: return jump.

Distant and return jumps

```
void f() {
    asm {        ; P
        jmp pp ; ⊥
        ...
        pp2:
    }
}
void g() {        // ⊥, pp ↦ P
    asm {
        pp:       ; P
        ...
        ; Q
        jmp pp2 ; ⊥, ret: pp2 ↦ Q
    }
}
```

Distant and return jumps

```
void f() {
    asm {          ; P
        jmp pp     ; ⊥, pp2 ↦ Q
        ...
        pp2:       ; Q
    }
}
void g() {           // ⊥, pp ↦ P
    asm {
        pp:          ; P
        ...
        ; Q
        jmp pp2  ; ⊥, ret: pp2 ↦ Q
    }
}
```

Distant and return jumps

Why so complicated?

C structure keep most of the control flow: essential for precision.
Syntactic information (`call`, `ret`, `jmp`) are absolutely not reliable!

And there is worse....

Mixed calls

```
int m;
int g(int a, int b) {
    int c = a + b;
    a = 1; b = 2;
    return c;
}
void f() {
    asm {
        push 22
        push 20
        call g
        mov m, EAX
        add ESP, 8
    }
}
```

```
extern int l(int);
int a = 0, param = 0;
void g() {
    asm {
        l:
        mov EAX, 4[ESP]
        mov param, EAX
        add EAX, 42
        ret
    }
}
void f() {
    a = l(1);
}
```

What it's all about?

Abstract interpretation saves the day

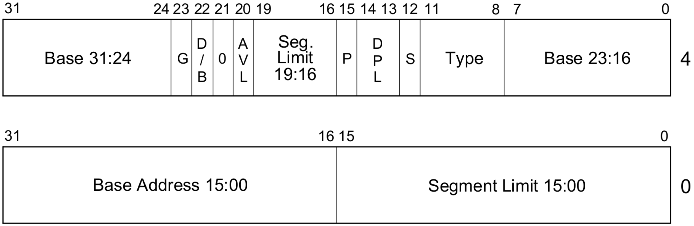
Add assembly into the soup

Even C is hell

Conclusion

What it's all about?
Abstract interpretation saves the day
Add assembly into the soup
Even C is hell
A common idiom
Virtual variables
Astrée on the inside
The new product in action
Dialectic
Current status
Conclusion

A common idiom



AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 1: A segment descriptor

Proving the security of an embedded operating system using abstract interpretation

Marc Chevalier

What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Even C is hell

A common idiom

Virtual variables

Astrée on the inside

The new product in action

Dialectic

Current status

Conclusion

A common idiom

```c
void set_base(struct seg_desc *seg, long* base)
{
        seg->low_base = base & 0xffff;
        seg->middle_base = (base >> 16) & 0xff;
        seg->high_base = base >> 24;
}
```

⇒ we want to remember slices of pointers.

What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Even C is hell

A common idiom

Virtual variables

Astrée on the inside

The new product in action

Dialectic

Current status

Conclusion

Virtual variables

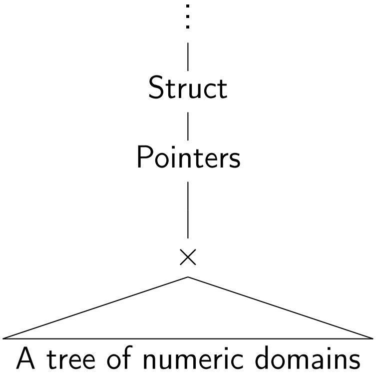
Variables used to represent an abstract value but that do not concretely exist.

```c
int t[4];
int* p = &t[2];
```

We want to say $\mathtt{p} = \mathtt{t} + o_p$. o_p: offset of p.

What it's all about?
Abstract interpretation saves the day
Add assembly into the soup
Even C is hell
A common idiom
Virtual variables
Astrée on the inside
The new product in action
Dialectic
Current status
Conclusion

Astrée on the inside



Downsides

- Variables ids are given by Struct: missing ids for virtual variables.
- There is no way to add another Pointer-like domain.
- Nodes in the tree can't do global iteration: delegated to Struct and Pointers

Astrée on the inside – Reduced Product

Given $(D_1^\sharp, \subseteq_1^\sharp), (D_2^\sharp, \subseteq_2^\sharp)$, abstract domains for the same concrete domain.

Product: $D_{1\times 2}^\sharp = D_1^\sharp \times D_2^\sharp$ with pointwise operations.
$\gamma_{1\times 2}(a1, a2) = \gamma_1(a_1) \cap \gamma_2(a_2)$

$\rho(a_1, a_2) = (b_1, b_2)$ with

$$\gamma_{1\times 2}(a1, a2) \subseteq \gamma_{1\times 2}(b1, b2) \qquad \text{(sound)}$$

$$\text{Morally:} \quad b_1 \subseteq_1^\sharp a_1 \wedge b_2 \subseteq_2^\sharp a_2 \qquad \text{(better)}$$

What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Even C is hell

A common idiom

Virtual variables

Astrée on the inside

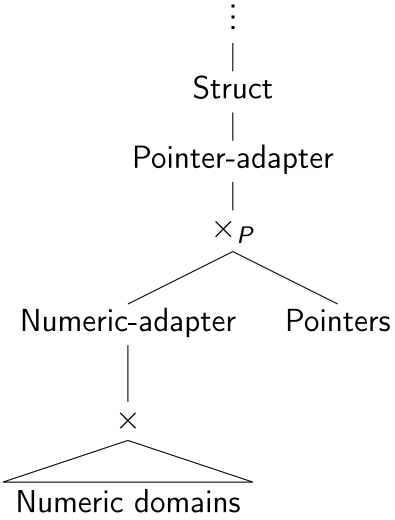
The new product in action

Dialectic

Current status

Conclusion

Astrée on the inside



New $\times_P$:

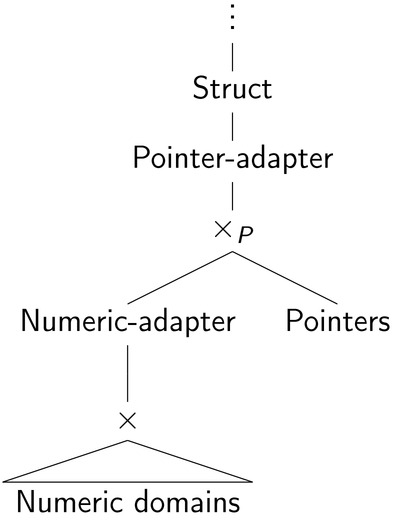
- Id translation by Pointer-adapter.
- Can add domains for pointer slices, linear combinations. . . .
- Cleaner interfaces.
- Each domain can ask everybody to store a virtual variable and do computations on it.

The new product in action

```
1 int t[4], *p, *q;
2 p = &t[0];
3 q = p + 1;
```

Before line 3:

- t has size 4.
- $p = t + o_p$.
- $o_p = 0$.



What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Even C is hell

- A common idiom
- Virtual variables
- Astrée on the inside
- **The new product in action**
- Dialectic
- Current status

Conclusion

The new product in action

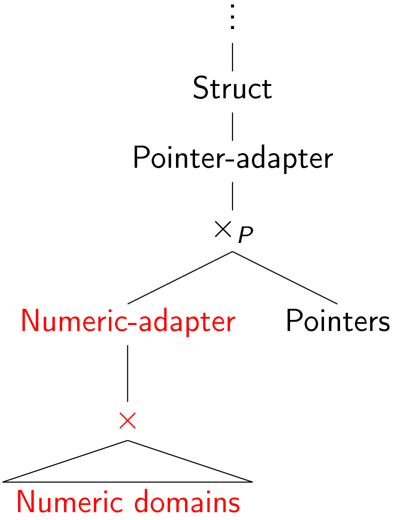
```c
int t[4], *p, *q;
p = &t[0];
q = p + 1;
```

$q \leftarrow p + 1$:

$p = \top$

$\Downarrow$

$q = \top$



What it's all about?
Abstract interpretation saves the day
Add assembly into the soup
Even C is hell
A common idiom
Virtual variables
Astrée on the inside
The new product in action
Dialectic
Current status
Conclusion

The new product in action

```c
int t[4], *p, *q;
p = &t[0];
q = p + 1;
```

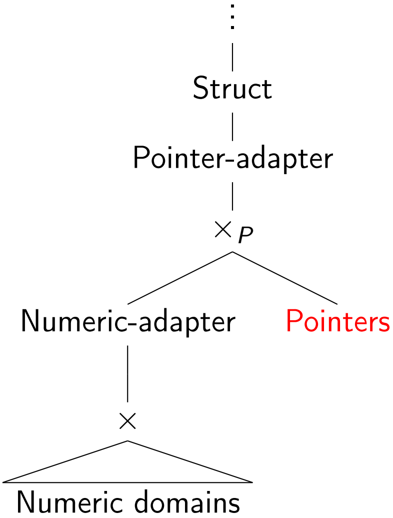
$q \leftarrow p + 1$:

$p = t + o_p$

$\Downarrow$

$o_q \leftarrow o_p + 1 \times 4$

context

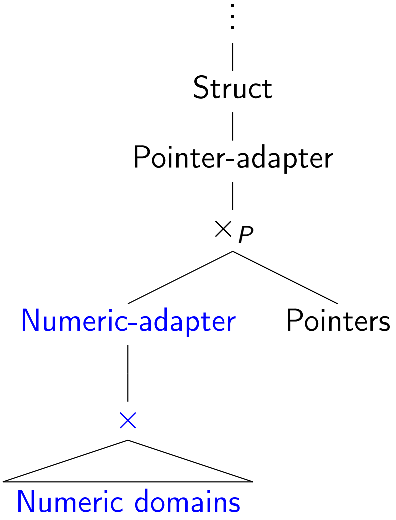


The new product in action

```
int t[4], *p, *q;
p = &t[0];
q = p + 1;
```

$o_q \leftarrow o_p + 1 \times 4$:

$o_p = 0$
$\Downarrow$
$o_q = 4$



What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Even C is hell

A common idiom

Virtual variables

Astrée on the inside

The new product in action

Dialectic

Current status

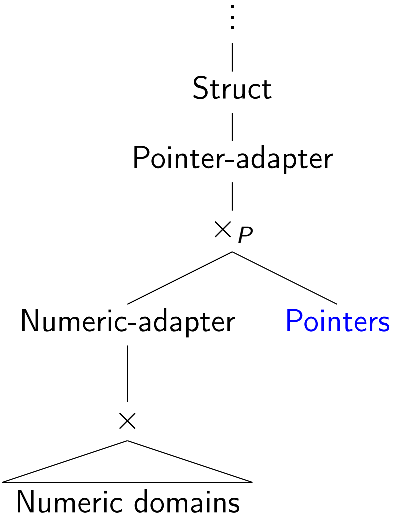
Conclusion

The new product in action

```c
int t[4], *p, *q;
p = &t[0];
q = p + 1;
```

$o_\mathtt{q} \leftarrow o_\mathtt{p} + 1 \times 4$:

$$o_\mathtt{p} \in \mathrm{NUM}$$
$$\Downarrow$$
$$o_\mathtt{q} \in \mathrm{NUM}$$



What it's all about?
Abstract interpretation saves the day
Add assembly into the soup
Even C is hell
A common idiom
Virtual variables
Astrée on the inside
The new product in action
Dialectic
Current status
Conclusion

The new product in action

```
int t[4], *p, *q;
p = &t[0];
q = p + 1;
```

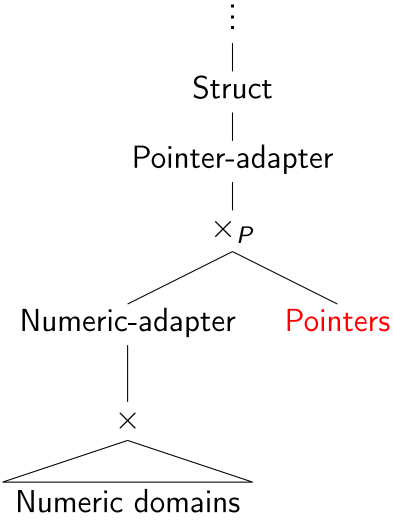
$\mathtt{q} \leftarrow \mathtt{p} + 1$:

$\mathtt{p} = \mathtt{t} + o_{\mathtt{p}}$

context

$\Downarrow$

$\mathtt{q} = \mathtt{t} + o_{\mathtt{q}}$



What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Even C is hell

A common idiom

Virtual variables

Astrée on the inside

The new product in action

Dialectic

Current status

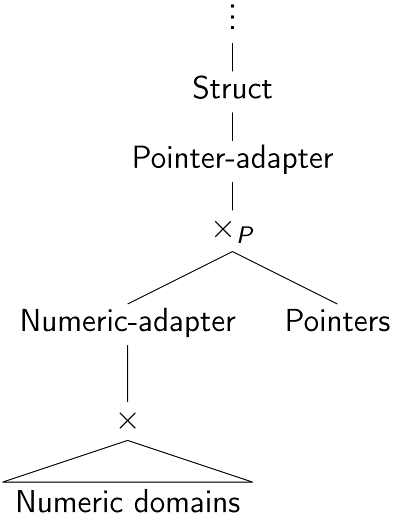
Conclusion

The new product in action

```c
1 int t[4], *p, *q;
2 p = &t[0];
3 q = p + 1;
```

After line 3:

- t has size 4.
- $\mathtt{p} = \mathtt{t} + o_{\mathtt{p}}$.
- $\mathtt{q} = \mathtt{t} + o_{\mathtt{q}}$.
- $o_{\mathtt{p}} = 0$.
- $o_{\mathtt{q}} = 4$.



What it's all about?
Abstract interpretation saves the day
Add assembly into the soup
Even C is hell
A common idiom
Virtual variables
Astrée on the inside
The new product in action
Dialectic
Current status
Conclusion

Dialectic

+ More general
+ Solve my problem
+ Solve older problems
+ Remove some hacks
+ Cleaner code (more parametric, more abstraction)

- More internal instructions for each real one: slower (I don't know how much)
- Very tricky
- Termination of each instruction is not ensured by local argument

And opportunistically: clean and optimize some old code I adapted.

What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Even C is hell
- A common idiom
- Virtual variables
- Astrée on the inside
- The new product in action
- Dialectic
- **Current status**

Conclusion

Current status

Astrée: 200klo OCaml.

All my modifications: Astrée: +84k -30k
Pointer product: Astrée: +30k -14k; Coproduct: 10k OCaml, 9k Python

Pointer product: big. Seems to work, but still testing. Log module to ease debugging.
Still to write: pointer slices and linear combinations.

What it's all about?

Abstract interpretation saves the day

Add assembly into the soup

Even C is hell

Conclusion

Conclusion

OK:

- Parsing, preprocessing, (dis)assembling, everything before the analysis.
- Register/stack abstraction.
- Control flow.

Still to do:

- Test (and debug) the new reduced product.
- Analysis: write stubs or model the environment.
- Some abstractions may need more precision, but the backbone is there.