Outline

Building a Certified Static Analyser (tutorial)

A small bytecode language

We will develop a certified analysis for a simple bytecode language.

Features:

- operand stack,
- integers (with infinite arithmetic),
- dynamically allocated arrays

Syntax

```
Definition pc := word.                          (* program counters          *)
Definition var := word.                         (* variable names            *)

Inductive binop := Add | Sub | Mult.            (* arith. binary operators   *)

Inductive cmp := Eq | Ne | Gt | Ge.             (* bool. binary operators    *)

Inductive instruction :=
  | Nop                                         (* do nothing                *)
  | Push (n:integer)| Pop | Dup                 (* op. stack manipulation    *)
  | Binop (op:binop)                            (* binary operation          *)
  | Load (x:var) | Store (x:var)                (* variable manipulation     *)
  | Newarray                                    (* array allocation          *)
  | Arraylength | Arrayload | Arraystore        (* array manipulation        *)
  | Input                                       (* get an arbitrary int.     *)
  | If (c:cmp) (jump:pc)                        (* conditional jump          *)
  | Goto (jump:pc).                             (* unconditional jump        *)

Definition program := list (pc * instruction).
```

The type `word` will be explained later...

Program example: Bubble sort

```
int n = 10;

int[] t = new int[n];

// We omit array
//      initialisation

for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (t[j+1] < t[j])
     { int tmp = t[j];
       t[j] = t[j+1];
       t[j+1] = tmp;}
```

Program example: Bubble sort

```
0 Ipush  10
1 Store  n
```

```
int n = 10;

int[] t = new int[n];

// We omit array
//      initialisation

for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (t[j+1] < t[j])
     { int tmp = t[j];
       t[j] = t[j+1];
       t[j+1] = tmp;}
```

Program example: Bubble sort

```
0 Ipush  10
1 Store  n

2 Load   n
3 Newarray
4 Store  t
```

```
int n = 10;

int[] t = new int[n];

// We omit array
//      initialisation

for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (t[j+1] < t[j])
     { int tmp = t[j];
       t[j] = t[j+1];
       t[j+1] = tmp;}
```

Program example: Bubble sort

```
 0 Ipush  10
 1 Store  n

 2 Load   n
 3 Newarray
 4 Store  t

 5 Ipush 0
 6 Store i
 7 Load i
 8 Load n
 9 Ipush 1
10 Binop Sub
11 If Ge 58
```

```
int n = 10;

int[] t = new int[n];

// We omit array
//      initialisation

for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (t[j+1] < t[j])
     { int tmp = t[j];
       t[j] = t[j+1];
       t[j+1] = tmp;}
```

Program example: Bubble sort

```
 0 Ipush  10
 1 Store  n

 2 Load   n
 3 Newarray
 4 Store  t

 5 Ipush 0
 6 Store i
 7 Load i
 8 Load n
 9 Ipush 1
10 Binop Sub
11 If Ge 58

12 Ipush 0
13 Store j
14 Load j
15 Load n
16 Ipush 1
17 Binop Sub
18 Load i
19 Binop Sub
```

```
int n = 10;

int[] t = new int[n];

// We omit array
//      initialisation

for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (t[j+1] < t[j])
     { int tmp = t[j];
       t[j] = t[j+1];
       t[j+1] = tmp;}
```

Program example: Bubble sort

```
 0 Ipush  10
 1 Store  n

 2 Load   n
 3 Newarray
 4 Store  t

 5 Ipush 0
 6 Store i
 7 Load i
 8 Load n
 9 Ipush 1
10 Binop Sub
11 If Ge 58

12 Ipush 0
13 Store j
14 Load j
15 Load n
16 Ipush 1
17 Binop Sub
18 Load i
19 Binop Sub
20 If Ge 53
21 Load t
22 Load j
23 Ipush 1
24 Binop Add
25 Arrayload
26 Load t
27 Load j
28 Arrayload
29 If Ge 48
```

```
int n = 10;

int[] t = new int[n];

// We omit array
//      initialisation

for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (t[j+1] < t[j])
     { int tmp = t[j];
       t[j] = t[j+1];
       t[j+1] = tmp;}
```

Program example: Bubble sort

```
 0 Ipush  10      20 If Ge 53
 1 Store  n       21 Load t
                  22 Load j
 2 Load   n       23 Ipush 1
 3 Newarray       24 Binop Add
 4 Store  t       25 Arrayload
                  26 Load t
 5 Ipush 0        27 Load j
 6 Store i        28 Arrayload
 7 Load i         29 If Ge 48
 8 Load n
 9 Ipush 1        30 Load t
10 Binop Sub      31 Load j
11 If Ge 58       32 Arrayload
                  33 Store tmp
12 Ipush 0        34 Load t
13 Store j        35 Load j
14 Load j         36 Load t
15 Load n         37 Load j
16 Ipush 1        38 Ipush 1
17 Binop Sub      39 Binop Add
18 Load i         40 Arrayload
19 Binop Sub      41 Arraystore
```

```
int n = 10;

int[] t = new int[n];

// We omit array
//      initialisation

for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (t[j+1] < t[j])
     { int tmp = t[j];
       t[j] = t[j+1];
       t[j+1] = tmp;}
```

Program example: Bubble sort

```
 0 Ipush  10
 1 Store  n
 2 Load   n
 3 Newarray
 4 Store  t
 5 Ipush 0
 6 Store i
 7 Load i
 8 Load n
 9 Ipush 1
10 Binop Sub
11 If Ge 58
12 Ipush 0
13 Store j
14 Load j
15 Load n
16 Ipush 1
17 Binop Sub
18 Load i
19 Binop Sub
20 If Ge 53
21 Load t
22 Load j
23 Ipush 1
24 Binop Add
25 Arrayload
26 Load t
27 Load j
28 Arrayload
29 If Ge 48
30 Load t
31 Load j
32 Arrayload
33 Store tmp
34 Load t
35 Load j
36 Load t
37 Load j
38 Ipush 1
39 Binop Add
40 Arrayload
41 Arraystore
42 Load t
43 Load j
44 Ipush 1
45 Binop Add
46 Load tmp
47 Arraystore
```

```
int n = 10;

int[] t = new int[n];

// We omit array
//      initialisation

for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (t[j+1] < t[j])
     { int tmp = t[j];
       t[j] = t[j+1];
       t[j+1] = tmp;}
```

Program example: Bubble sort

```
 0 Ipush  10
 1 Store  n

 2 Load   n
 3 Newarray
 4 Store  t

 5 Ipush 0
 6 Store i
 7 Load i
 8 Load n
 9 Ipush 1
10 Binop Sub
11 If Ge 58

12 Ipush 0
13 Store j
14 Load j
15 Load n
16 Ipush 1
17 Binop Sub
18 Load i
19 Binop Sub
20 If Ge 53
21 Load t
22 Load j
23 Ipush 1
24 Binop Add
25 Arrayload
26 Load t
27 Load j
28 Arrayload
29 If Ge 48

30 Load t
31 Load j
32 Arrayload
33 Store tmp
34 Load t
35 Load j
36 Load t
37 Load j
38 Ipush 1
39 Binop Add
40 Arrayload
41 Arraystore
42 Load t
43 Load j
44 Ipush 1
45 Binop Add
46 Load tmp
47 Arraystore

48 Load j
49 Ipush 1
50 Binop Add
51 Store j
52 Goto 14
```

```
int n = 10;

int[] t = new int[n];

// We omit array
//      initialisation

for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (t[j+1] < t[j])
     { int tmp = t[j];
       t[j] = t[j+1];
       t[j+1] = tmp;}
```

Program example: Bubble sort

```
 0 Ipush  10
 1 Store  n
 2 Load   n
 3 Newarray
 4 Store  t
 5 Ipush 0
 6 Store i
 7 Load i
 8 Load n
 9 Ipush 1
10 Binop Sub
11 If Ge 58
12 Ipush 0
13 Store j
14 Load j
15 Load n
16 Ipush 1
17 Binop Sub
18 Load i
19 Binop Sub
20 If Ge 53
21 Load t
22 Load j
23 Ipush 1
24 Binop Add
25 Arrayload
26 Load t
27 Load j
28 Arrayload
29 If Ge 48
30 Load t
31 Load j
32 Arrayload
33 Store tmp
34 Load t
35 Load j
36 Load t
37 Load j
38 Ipush 1
39 Binop Add
40 Arrayload
41 Arraystore
42 Load t
43 Load j
44 Ipush 1
45 Binop Add
46 Load tmp
47 Arraystore
48 Load j
49 Ipush 1
50 Binop Add
51 Store j
52 Goto 14
53 Load i
54 Ipush 1
55 Binop Add
56 Store i
57 Goto 7
58
```

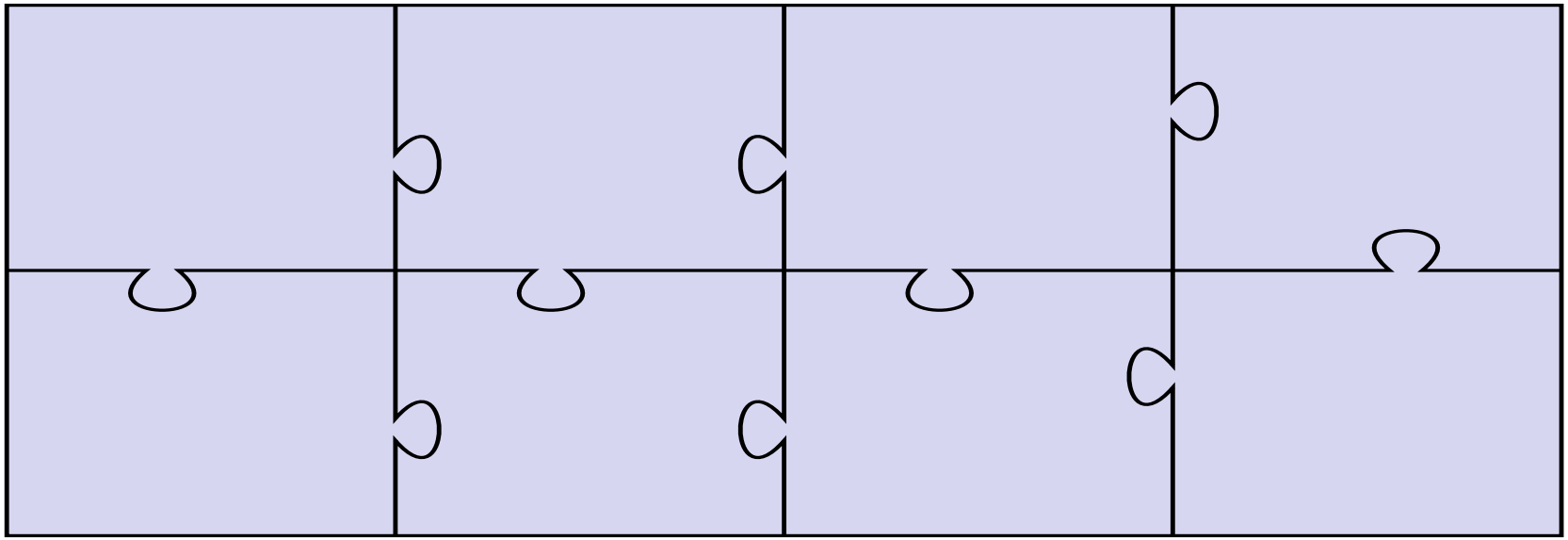
```
int n = 10;

int[] t = new int[n];

// We omit array
//      initialisation

for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (t[j+1] < t[j])
     { int tmp = t[j];
       t[j] = t[j+1];
       t[j+1] = tmp;}
```

Our objectif for today

```
int n = 10;
int[] t = new int[n];
for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (t[j+1] < t[j])
     { int tmp = t[j]; t[j] = t[j+1]; t[j+1] = tmp;}
```

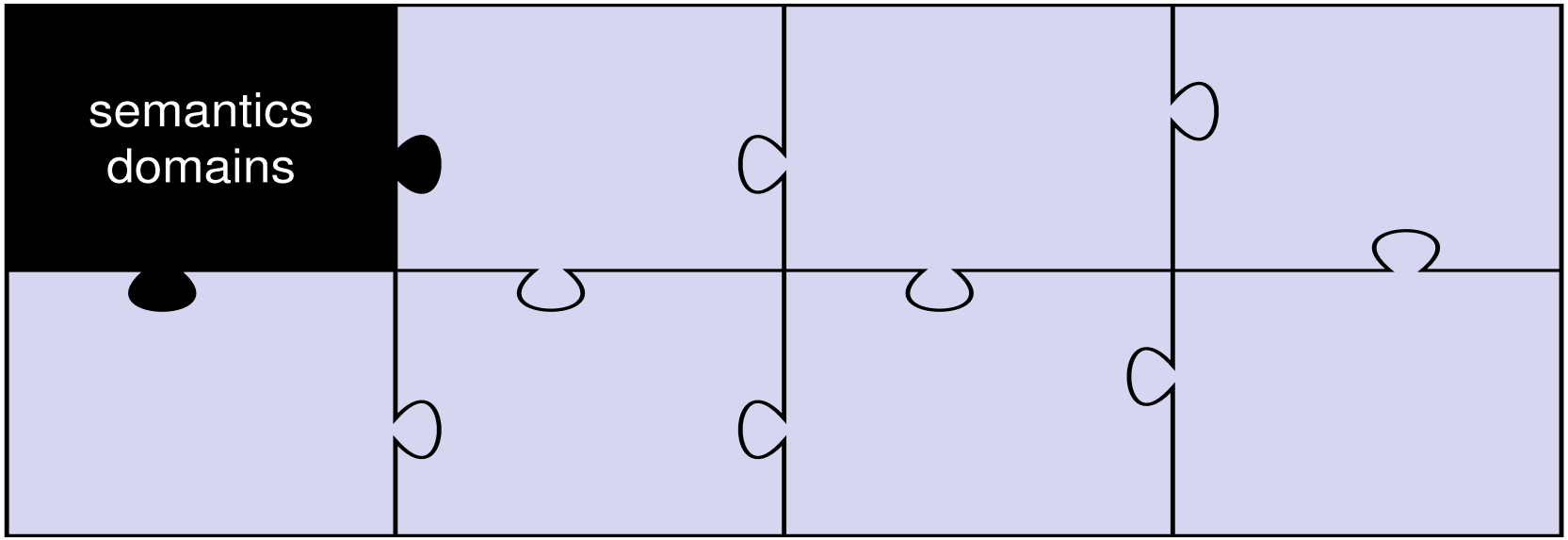
- Build a certified static analysis which proves that all array accesses are in the right bounds.

Building a certified static analyser



- A puzzle with 8 pieces,
- Each piece interacts with its neighbors

Building a certified static analyser



We have to define formally the runtime structures that are manipulated by a program during execution.

Semantic domains

```
Inductive val : Set :=
| Num (i:integer)
| Ref (l:location).

Inductive var_val : Set := Undef | Def (v:val).
Definition locvar : Set := var → var_val.

Definition opstack : Set := list val.

Record array : Set := {
  array_length : integer;
  array_values : ∀ i:integer, (0 <= i < array_length) → integer
}.

Inductive heap_val : Set := No | Array (a:array).
Definition heap := location → heap_val.

Inductive state : Set :=
| St (i:pc) (s:opstack) (l:locvar) (h:heap)
| Error.
```

Dependent type

```
Record array : Set := {
  array_length : integer;
  array_values : ∀ i:integer, (0 <= i < array_length) → integer
}.
```

We use here Coq dependent types.

- `array` is a record where the type of the second field depends on the value of the first field.

A function `f` of type

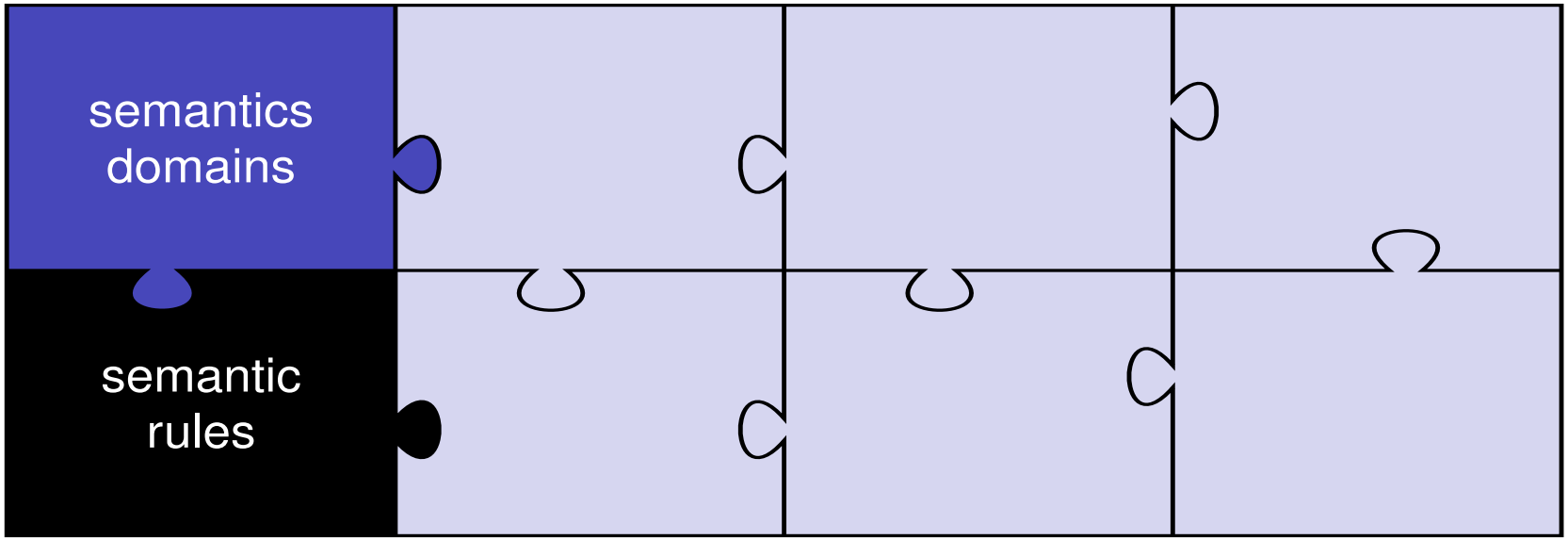
```
∀ i:integer, (0 <= i < array_length) → integer
```

takes as argument

- an integer (let's call it `i`)
- a *proof* that `0 <= i < array_length` holds

and returns an integer.

Building a certified static analyser



We have to formally explain how a program manipulates the elements of the semantic domain during an execution.

- Operational semantics: transition relation $\rightarrow_p$ between states
- This is naturally done in Coq with inductive definitions

Semantic rules

In LaTeX

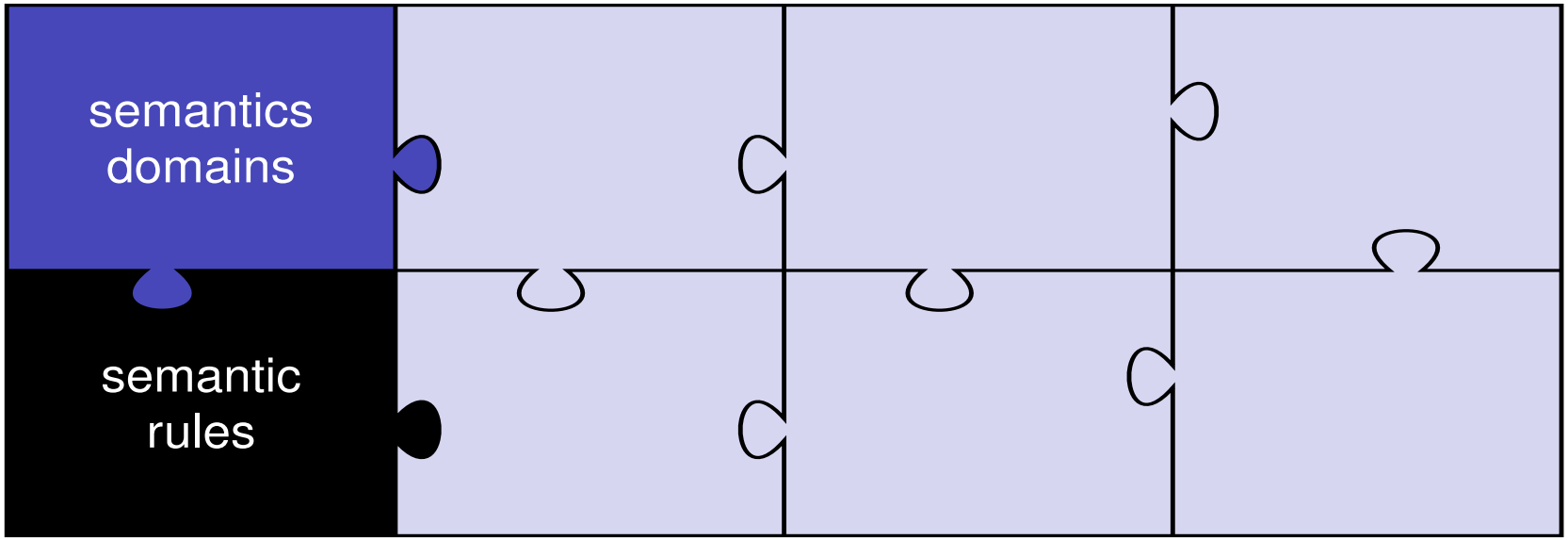
$$\frac{\mathsf{instrAt}_p(pc) = \texttt{nop}}{\langle\!\langle pc, s, l, h \rangle\!\rangle \rightarrow_p \langle\!\langle pc{+}1, s, l, h \rangle\!\rangle} \qquad \frac{\mathsf{instrAt}_p(pc) = \texttt{push}\ n}{\langle\!\langle pc, s, l, h \rangle\!\rangle \rightarrow_p \langle\!\langle pc{+}1, n :: s, l, h \rangle\!\rangle}$$

$$\frac{\mathsf{instrAt}_p(pc) = \texttt{load}\ x \qquad x \in \mathrm{dom}(l)}{\langle\!\langle pc, s, l, h \rangle\!\rangle \rightarrow_p \langle\!\langle pc{+}1, l(x) :: s, l, h \rangle\!\rangle}$$

In Coq

```
Inductive step (p:program) : state → state → Prop :=
 | step_nop : ∀ pc h s l,
     instr_at p pc = Some Nop →
     (St pc s l h) -[p]→ (St (next pc) s l h)
 | step_push : ∀ pc h s l n,
     instr_at p pc = Some (Push n) →
     (St pc s l h) -[p]→ (St (next pc) (Num n :: s) l h)
 | step_load : ∀ pc h s l x v,
     instr_at p pc = Some (Load x) →
     l x = Def v →
     (St pc s l h) -[p]→ (St (next pc) (v::s) l h)
 | ...
where "s1 -[ p ]→ s2" := (step p s1 s2).
```

Building a certified static analyser



We can then define the set of reachable states during the execution of a program.

$$[\![P]\!] = \{s \mid \langle\langle 0, \varepsilon, \emptyset, \emptyset \rangle\rangle \rightarrow_p^* s\}$$

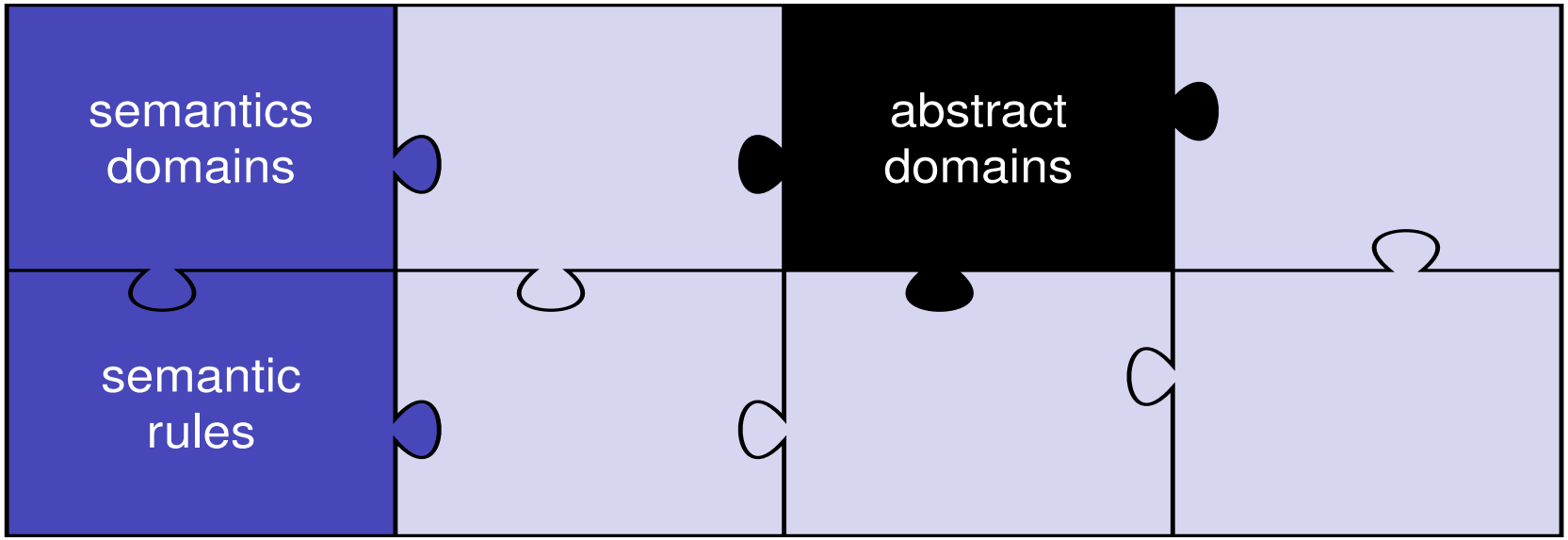
Reachable states

$$[\![P]\!] = \{s \mid \langle\langle 0, \varepsilon, \emptyset, \emptyset \rangle\rangle \rightarrow_p^* s\}$$

```
Inductive ReachableStates (p:program) : state → Prop :=
 | reach_init :
   ReachableStates p (St Word_1 nil (fun _ ⇒ Undef) (fun _ ⇒ No))
 | reach_next : ∀ st1 st2,
     ReachableStates p st1 →
     st1 -[p]→ st2 →
     ReachableStates p st2.
```

We want to compute a sound over-approximation of $[\![P]\!]$.

Building a certified static analyser



- Each semantic sub-domain has its abstract counterpart
- An abstract domain is a lattice $\left(\mathcal{D}^\sharp, =, \sqsubseteq, \bot, \sqcup, \sqcap\right)$ + a property for ensuring termination of fixpoint iteration.

Abstract Interpretation Philosophy

An abstract interpreter executes program on properties instead of simple values.

The want to compute a property at each program point of a program. Then the abstract domain will be of the form

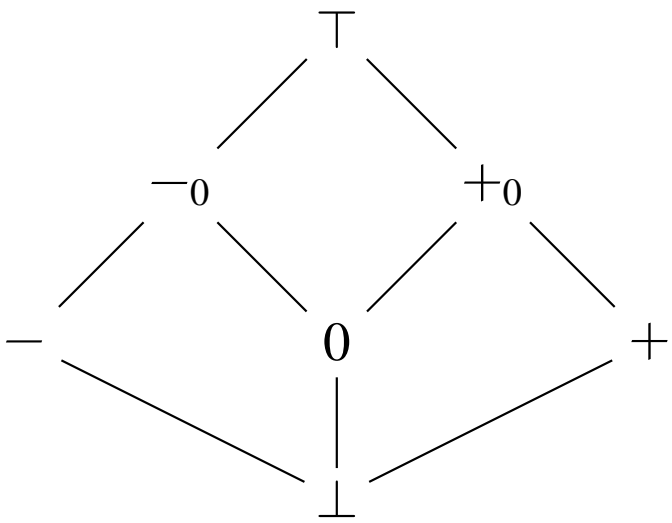
$$A^\sharp = \mathbf{pc} \to \mathrm{Mem}^\sharp$$

We will abstract arrays by their lengths.

- Array are dynamically created but their length is immutable.

We need a numerical abstraction $Num^\sharp$ (example: signs).

An abstraction by signs



$\bot$	represents the property	$\emptyset$
$-$	represents the property	$\{ z \mid z < 0 \}$
0	represents the property	$\{0\}$
$+$	represents the property	$\{ z \mid z > 0 \}$
$-_0$	represents the property	$\{ z \mid z \leqslant 0 \}$
$+_0$	represents the property	$\{ z \mid z \geqslant 0 \}$
$\top$	represents the property	$\mathbb{Z}$

A first attempt with `if (i+1 < n) ...`

```
0 Load i

1 Ipush 1

2 Binop Add

3 Load  n

4 If Ge 27
```

This is not precise enough. Why ?

A first attempt with `if (i+1 < n) ...`

$\square$ $\quad i : +_0; \quad n : \top;$

```
0 Load i

1 Ipush 1

2 Binop Add

3 Load  n

4 If Ge 27
```

This is not precise enough. Why ?

A first attempt with `if (i+1 < n) ...`

$[]$ $\quad i : +_0; \quad n : \top;$

```
0 Load i
```

$[+_0]$ $\quad i : +_0; \quad n : \top;$

```
1 Ipush 1

2 Binop Add

3 Load  n

4 If Ge 27
```

This is not precise enough. Why ?

A first attempt with `if (i+1 < n) ...`

```
          []          i : +0;  n : ⊤;
0 Load i
          [+0]        i : +0;  n : ⊤;
1 Ipush 1
          [+0; +]     i : +0;  n : ⊤;
2 Binop Add

3 Load  n

4 If Ge 27
```

This is not precise enough. Why ?

A first attempt with `if (i+1 < n) ...`

```
          []        i : +0;   n : ⊤;
0 Load i
          [+0]      i : +0;   n : ⊤;
1 Ipush 1
          [+0; +]   i : +0;   n : ⊤;
2 Binop Add
          [+]       i : +0;   n : ⊤;
3 Load  n


4 If Ge 27
```

This is not precise enough. Why ?

A first attempt with `if (i+1 < n) ...`

```
                []        i : +0;   n : ⊤;
0 Load i
              [+0]        i : +0;   n : ⊤;
1 Ipush 1
              [+0; +]     i : +0;   n : ⊤;
2 Binop Add
              [+]         i : +0;   n : ⊤;
3 Load  n
              [+; ⊤]      i : +0;   n : ⊤;
4 If Ge 27
```

This is not precise enough. Why ?

A first attempt with `if (i+1 < n) ...`

```
          []        i : +0;  n : ⊤;
0 Load i
          [+0]      i : +0;  n : ⊤;
1 Ipush 1
          [+0; +]   i : +0;  n : ⊤;
2 Binop Add
          [+]       i : +0;  n : ⊤;
3 Load  n
          [+; ⊤]    i : +0;  n : ⊤;
4 If Ge 27
          []        i : +0;  n : ⊤;
```

This is not precise enough. Why ?

A first attempt with `if (i+1 < n) ...`

```
          []        i : +0;   n : ⊤;
0 Load i
          [+0]      i : +0;   n : ⊤;
1 Ipush 1
          [+0; +]   i : +0;   n : ⊤;
2 Binop Add
          [+]       i : +0;   n : ⊤;
3 Load  n
          [+; ⊤]    i : +0;   n : ⊤;
4 If Ge 27
          []        i : +0;   n : ⊤; ← +
```

This is not precise enough. Why ?

Recovering relation between op. stack and local vars

We will abstract operand stacks by stacks of symbolic expressions.

Each symbolic expression represents a relation between the corresponding value of the operand stack and the local variables.

The language of expressions is defined by the following inductive type.

```
Inductive expr :=
  Const (n:integer) | Var (x:var) | Bin (bin:binop) (e1 e2:expr).
```

Symbolic op. stack

$[]$ $\quad i : +_0; \quad n : \top;$

```
0 Load i
```
$[+_0]$ $\quad i : +_0; \quad n : \top;$

```
1 Ipush 1
```
$[+_0; +]$ $\quad i : +_0; \quad n : \top;$

```
2 Binop Add
```
$[+]$ $\quad i : +_0; \quad n : \top;$

```
3 Load  n
```
$[+; \top]$ $\quad i : +_0; \quad n : \top;$

```
4 If Ge 27
```
$[]$ $\quad i : +_0; \quad n : \top;$

```
0 Load i
```

```
1 Ipush 1
```

```
2 Binop Add
```

```
3 Load  n
```

```
4 If Ge 27
```

Symbolic op. stack

```
                []        i : +0;  n : ⊤;

0 Load i
                [+0]      i : +0;  n : ⊤;

1 Ipush 1
                [+0; +]   i : +0;  n : ⊤;

2 Binop Add
                [+]       i : +0;  n : ⊤;

3 Load  n
                [+; ⊤]    i : +0;  n : ⊤;

4 If Ge 27
                []        i : +0;  n : ⊤;
```

```
                []        i : +0;  n : ⊤;

0 Load i

1 Ipush 1

2 Binop Add

3 Load  n

4 If Ge 27
```

Symbolic op. stack

```
                []        i : +0;  n : ⊤;
0 Load i
              [+0]        i : +0;  n : ⊤;
1 Ipush 1
              [+0; +]     i : +0;  n : ⊤;
2 Binop Add
               [+]        i : +0;  n : ⊤;
3 Load  n
              [+; ⊤]      i : +0;  n : ⊤;
4 If Ge 27
                []        i : +0;  n : ⊤;
```

```
                []        i : +0;  n : ⊤;
0 Load i
               [i]        i : +0;  n : ⊤;
1 Ipush 1
2 Binop Add
3 Load  n
4 If Ge 27
```

Symbolic op. stack

```
                []        i : +0;  n : ⊤;
0 Load i
              [+0]        i : +0;  n : ⊤;
1 Ipush 1
           [+0; +]        i : +0;  n : ⊤;
2 Binop Add
              [+]         i : +0;  n : ⊤;
3 Load  n
            [+; ⊤]        i : +0;  n : ⊤;
4 If Ge 27
                []        i : +0;  n : ⊤;
```

```
                []        i : +0;  n : ⊤;
0 Load i
               [i]        i : +0;  n : ⊤;
1 Ipush 1
            [i; 1]        i : +0;  n : ⊤;
2 Binop Add
3 Load  n
4 If Ge 27
```

Symbolic op. stack

```
                []        i : +0;  n : ⊤;
0 Load i
              [+0]        i : +0;  n : ⊤;
1 Ipush 1
           [+0; +]        i : +0;  n : ⊤;
2 Binop Add
              [+]         i : +0;  n : ⊤;
3 Load  n
            [+; ⊤]        i : +0;  n : ⊤;
4 If Ge 27
                []        i : +0;  n : ⊤;
```

```
                []        i : +0;  n : ⊤;
0 Load i
               [i]        i : +0;  n : ⊤;
1 Ipush 1
            [i; 1]        i : +0;  n : ⊤;
2 Binop Add
             [i+1]        i : +0;  n : ⊤;
3 Load  n
4 If Ge 27
```

Symbolic op. stack

$[]$ $\quad i : +_0; \quad n : \top;$

```
0 Load i
```

$[+_0]$ $\quad i : +_0; \quad n : \top;$

```
1 Ipush 1
```

$[+_0; +]$ $\quad i : +_0; \quad n : \top;$

```
2 Binop Add
```

$[+]$ $\quad i : +_0; \quad n : \top;$

```
3 Load  n
```

$[+; \top]$ $\quad i : +_0; \quad n : \top;$

```
4 If Ge 27
```

$[]$ $\quad i : +_0; \quad n : \top;$

$[]$ $\quad i : +_0; \quad n : \top;$

```
0 Load i
```

$[i]$ $\quad i : +_0; \quad n : \top;$

```
1 Ipush 1
```

$[i; 1]$ $\quad i : +_0; \quad n : \top;$

```
2 Binop Add
```

$[i+1]$ $\quad i : +_0; \quad n : \top;$

```
3 Load  n
```

$[i+1; n]$ $\quad i : +_0; \quad n : \top;$

```
4 If Ge 27
```

Symbolic op. stack

$[]$ $\quad i : +_0;\quad n : \top;$

```
0 Load i
```

$[+_0]$ $\quad i : +_0;\quad n : \top;$

```
1 Ipush 1
```

$[+_0; +]$ $\quad i : +_0;\quad n : \top;$

```
2 Binop Add
```

$[+]$ $\quad i : +_0;\quad n : \top;$

```
3 Load  n
```

$[+; \top]$ $\quad i : +_0;\quad n : \top;$

```
4 If Ge 27
```

$[]$ $\quad i : +_0;\quad n : \top;$

$[]$ $\quad i : +_0;\quad n : \top;$

```
0 Load i
```

$[i]$ $\quad i : +_0;\quad n : \top;$

```
1 Ipush 1
```

$[i; 1]$ $\quad i : +_0;\quad n : \top;$

```
2 Binop Add
```

$[i+1]$ $\quad i : +_0;\quad n : \top;$

```
3 Load  n
```

$[i+1; n]$ $\quad i : +_0;\quad n : \top;$

```
4 If Ge 27
```

$[]$ $\quad i : +_0;\quad n : +;$

Symbolic op. stack

$[]$ $\quad i : +_0;\quad n : \top;$

```
0 Load i
```

$[+_0]$ $\quad i : +_0;\quad n : \top;$

```
1 Ipush 1
```

$[+_0; +]$ $\quad i : +_0;\quad n : \top;$

```
2 Binop Add
```

$[+]$ $\quad i : +_0;\quad n : \top;$

```
3 Load  n
```

$[+; \top]$ $\quad i : +_0;\quad n : \top;$

```
4 If Ge 27
```

$[]$ $\quad i : +_0;\quad n : \top; \leftarrow +$

$[]$ $\quad i : +_0;\quad n : \top;$

```
0 Load i
```

$[i]$ $\quad i : +_0;\quad n : \top;$

```
1 Ipush 1
```

$[i; 1]$ $\quad i : +_0;\quad n : \top;$

```
2 Binop Add
```

$[i+1]$ $\quad i : +_0;\quad n : \top;$

```
3 Load  n
```

$[i+1; n]$ $\quad i : +_0;\quad n : \top;$

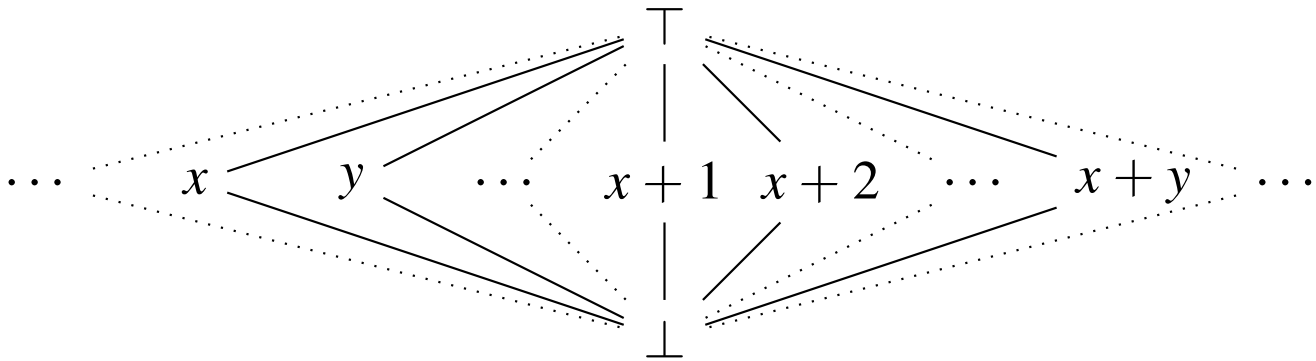
```
4 If Ge 27
```

$[]$ $\quad i : +_0;\quad n : +;$

Our abstract domain

$$\begin{aligned} Mem^\sharp &= list(\mathbf{Expr}^\top_\bot) \times (\mathbf{var} \to Num^\sharp) \\ A^\sharp &= \mathbf{pc} \to \mathsf{Mem}^\sharp \end{aligned}$$

$\mathbf{Expr}^\top_\bot$ is a flat lattice



Building lattices in Coq

We propose a technique based on the Coq module system (inspired by the ML module system)

- Lattice requirements are collected in a module contract

Lattice contract

```
Module Type Lattice.
```

```
End Lattice.
```

Lattice contract

```
Module Type Lattice.
  Parameter t : Set.

End Lattice.
```

Lattice contract

```
Module Type Lattice.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter eq_prop : ...
              (* eq (=) is a computable equivalence relation *)
```

```
End Lattice.
```

Lattice contract

```
Module Type Lattice.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter eq_prop : ...
              (* eq (=) is a computable equivalence relation *)
  Parameter order : t → t → Prop.
  Parameter order_prop : ...
              (* order (⊑) is a computable order relation *)
```

```
End Lattice.
```

Lattice contract

```
Module Type Lattice.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter eq_prop : ...
              (* eq (=) is a computable equivalence relation *)
  Parameter order : t → t → Prop.
  Parameter order_prop : ...
              (* order (⊑) is a computable order relation *)
  Parameter join : t → t → t.
  Parameter join_prop : ...
              (* join (⊔) is a binary least upper bound *)
```

```
End Lattice.
```

Lattice contract

```
Module Type Lattice.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter eq_prop : ...
              (* eq (=) is a computable equivalence relation *)
  Parameter order : t → t → Prop.
  Parameter order_prop : ...
              (* order (⊑) is a computable order relation *)
  Parameter join : t → t → t.
  Parameter join_prop : ...
              (* join (⊔) is a binary least upper bound *)
  Parameter meet : t → t → t.
  Parameter meet_prop : ...
              (* meet (⊓) is a binary greatest lower bound *)



End Lattice.
```

Lattice contract

```
Module Type Lattice.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter eq_prop : ...
              (* eq (=) is a computable equivalence relation *)
  Parameter order : t → t → Prop.
  Parameter order_prop : ...
              (* order (⊑) is a computable order relation *)
  Parameter join : t → t → t.
  Parameter join_prop : ...
              (* join (⊔) is a binary least upper bound *)
  Parameter meet : t → t → t.
  Parameter meet_prop : ...
              (* meet (⊓) is a binary greatest lower bound *)
  Parameter bottom : t.
              (* bottom element to start iteration *)
  Parameter bottom_is_bottom : ∀ x : t, order bottom x.

End Lattice.
```

Lattice contract

```
Module Type Lattice.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter eq_prop : ...
              (* eq (=) is a computable equivalence relation *)
  Parameter order : t → t → Prop.
  Parameter order_prop : ...
              (* order (⊑) is a computable order relation *)
  Parameter join : t → t → t.
  Parameter join_prop : ...
              (* join (⊔) is a binary least upper bound *)
  Parameter meet : t → t → t.
  Parameter meet_prop : ...
              (* meet (⊓) is a binary greatest lower bound *)
  Parameter bottom : t.
              (* bottom element to start iteration *)
  Parameter bottom_is_bottom : ∀ x : t, order bottom x.
  Parameter termination_property : ...
End Lattice.
```

Building lattices in Coq

We propose a technique based on the Coq module system (inspired by the ML module system)

- Lattice requirements are collected in a module contract

- Various functors are proposed in order to build lattices by composition of others

Lattice functors

- Base lattices: signs, congruences, constants, intervals, finite set (implemented with trees)
- Functors : product, disjoint and lifted sums, lists, functional arrays (implemented with trees)
  - Keys must be in finite numbers: we use the type `word` that represents binary numbers with at most 32 bits.

For each functor the most challenging proofs deals with the preservation of the termination criterion.

Building lattices in Coq

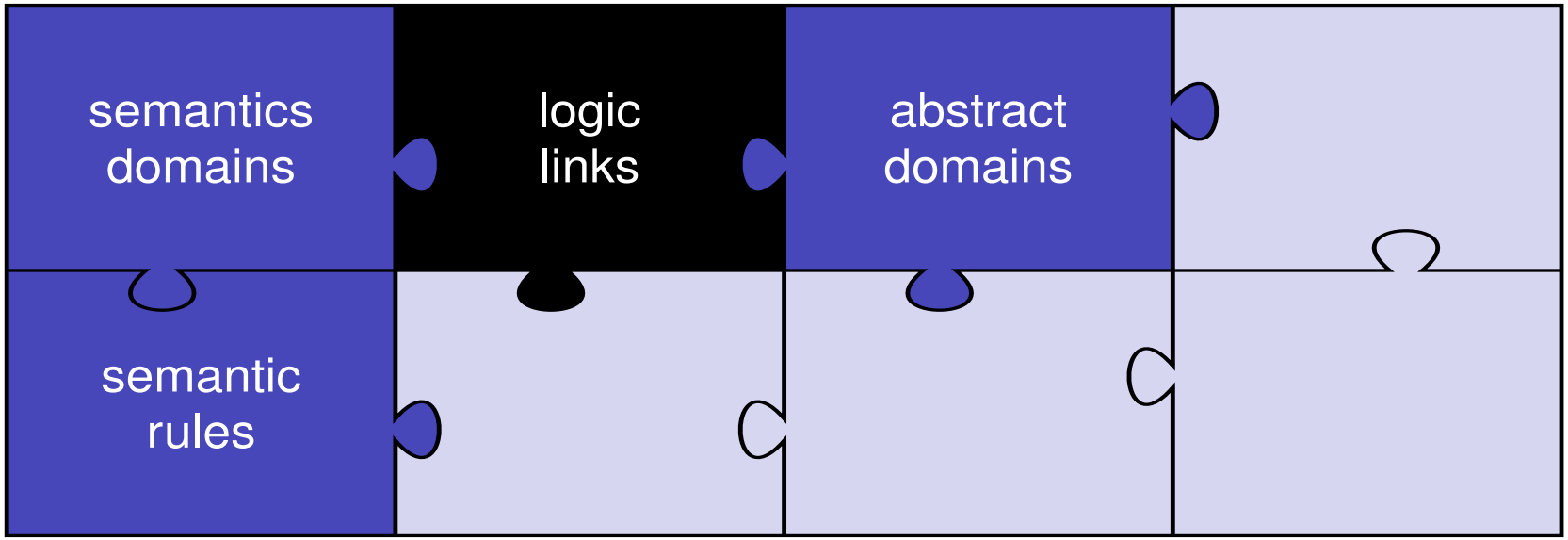
We propose a technique based on the Coq module system (inspired by the ML module system)

- Lattice requirements are collected in a module contract
- Various functors are proposed in order to build lattices by composition of others

Example:

```
Module ExprLat := FlatLattice Expr.
Module LocvarLat := ArrayBinLattice SignLat.
Module OpstackLat := ListLattice ExprLat.
Module MemLat := ProdLattice StackLat LocalVarLat.
Module GlobalLat := ArrayBinLattice MemLat.
```

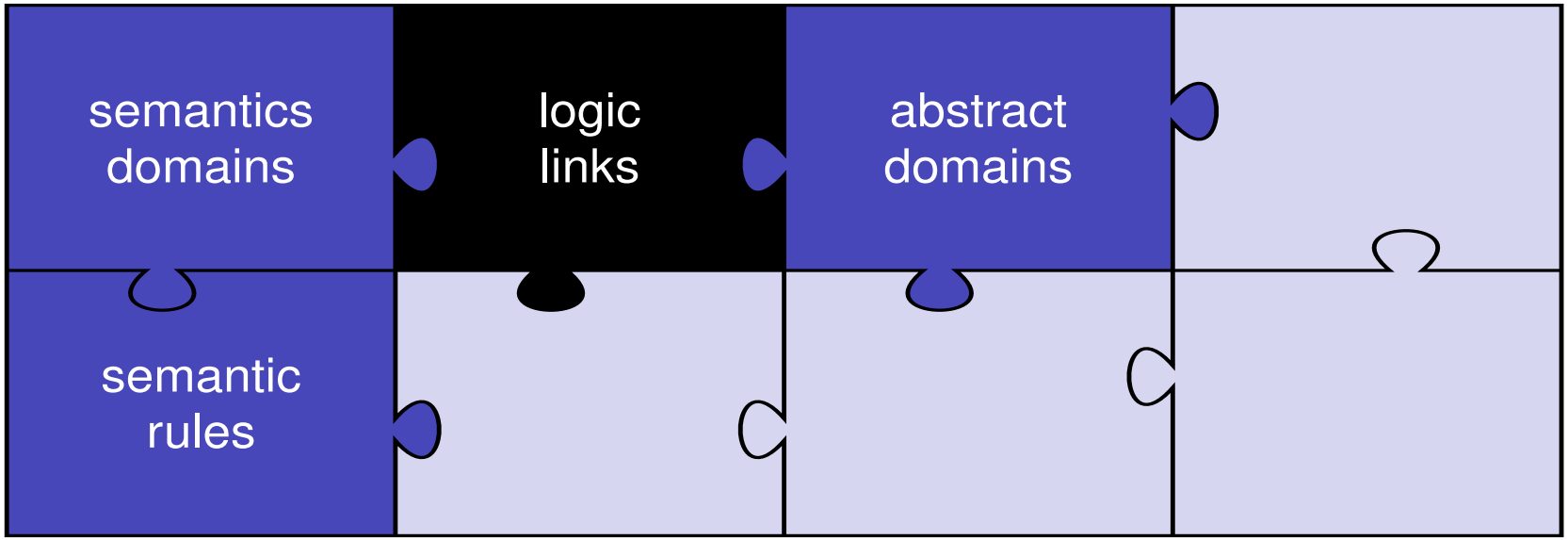
Building a certified static analyser



- Each abstract value represents a property on concrete values
- This correspondence is formalised by a monotone concretisation function

$$\gamma : \left(\mathcal{D}^\sharp, \sqsubseteq\right) \longrightarrow_m \left(\wp(\mathcal{D}), \subseteq\right)$$

Building a certified static analyser

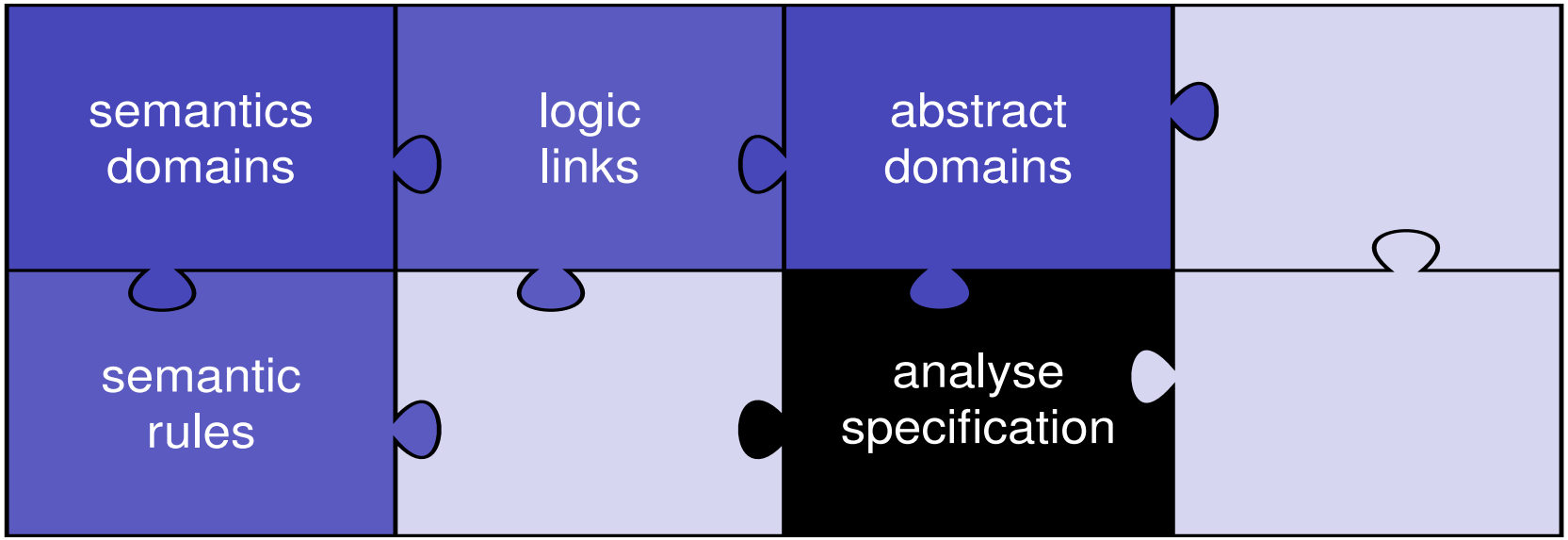


- Each abstract value represents a property on concrete values
- This correspondence is formalised by a monotone concretisation function

$$\gamma : \left(\mathcal{D}^\sharp, \sqsubseteq\right) \longrightarrow_m \left(\wp(\mathcal{D}), \subseteq\right)$$

$x \subseteq \gamma(x^\sharp)$ means "$x^\sharp$ is a correct approximation of x"

Building a certified static analyser



- the analysis is specified as a solution of a post fixpoint problem

$$F_P^\sharp(X) \sqsubseteq^\sharp X$$

- after partitioning $X \in \mathbf{pc} \to \text{Mem}^\sharp$: constraint system

$$\begin{cases} f_1^\sharp(X[1],\ldots,X[n]) & \sqsubseteq^\sharp & X[i_1] \\ \ldots & & \\ f_n^\sharp(X[1],\ldots,X[n]) & \sqsubseteq^\sharp & X[i_n] \end{cases}$$

Constraint representation

```
Record cstr : Set := C {
  source : pc;
  target : pc;
  constraint : MemLattice.t → MemLattice.t
}.
```

A triplet (i, j, F) represents the constraint

$$F(X(i)) \sqsubseteq X(j)$$

We generate one or two constraints for each program point:

```
Definition gen_one_constraint (i:pc) (ins:instruction) : list cstr :=
  match ins with
    | Nop ⇒ C i (next i) (fun x ⇒ x) ::nil
    ...
```

Required operators on the numeric abstraction (1/4)

- $\text{const}^\sharp \in \text{Num} \to \text{Num}^\sharp$ computes an approximations of constants

$$\forall n \in \mathbb{Z},\ \{n\} \subseteq \gamma_{\text{Num}}(\text{const}^\sharp(n))$$

- $\top_{\text{Num}} \in \text{Num}^\sharp$ approximates any numeric value

$$\mathbb{Z} \subseteq \gamma_{\text{Num}}(\top_{\text{Num}})$$

- $[\![o]\!]\uparrow^\sharp_{\text{op}} \in \text{Num}^\sharp \times \text{Num}^\sharp \to \text{Num}^\sharp$ is a correct approximation of the arithmetic operators $o \in \{+, -, \times\}$

$$\forall n_1^\sharp, n_2^\sharp \in \text{Num}^\sharp,$$
$$\{\ n_1\ [\![o]\!]\uparrow_{\text{op}}\ n_2 \mid n_1 \in \gamma_{\text{Num}}(n_1^\sharp),\ n_2 \in \gamma_{\text{Num}}(n_2^\sharp)\ \} \subseteq \gamma_{\text{Num}}([\![o]\!]\uparrow^\sharp_{\text{op}}\ (n_1^\sharp, n_2^\sharp))$$

Required operators on the numeric abstraction (2/4)

$$[\![e_1\ c\ e_2]\!]^\sharp_{\text{test}}(\rho^\sharp) \quad = \quad \left([\![e_1]\!]\downarrow^\sharp_{\text{expr}}\ (\rho^\sharp, n_1^\sharp) \sqcap^\sharp_{\text{Env}} [\![e_2]\!]\downarrow^\sharp_{\text{expr}}\ (\rho^\sharp, n_2^\sharp)\right)$$
$$\text{with } (n_1^\sharp, n_2^\sharp) = [\![c]\!]\downarrow^\sharp_{\text{comp}}\ \left([\![e_1]\!]\uparrow^\sharp_{\text{expr}}\ (\rho^\sharp), [\![e_2]\!]\uparrow^\sharp_{\text{expr}}\ (\rho^\sharp)\right)$$

- $[\![c]\!]\downarrow^\sharp_{\text{comp}} \in \text{Num}^\sharp \times \text{Num}^\sharp \to \text{Num}^\sharp \times \text{Num}^\sharp$ computes a refinement of two numeric abstract values, knowing that they verifies condition c
- $[\![e]\!]\downarrow^\sharp_{\text{expr}} \in \text{Env}^\sharp \times \text{Num}^\sharp \to \text{Env}^\sharp$: $[\![e]\!]\downarrow^\sharp_{\text{expr}}\ (\rho^\sharp, n^\sharp)$ computes a refinement of the abstract environment $\rho^\sharp$, knowing that the expression e evaluates into a value that is approximated by $n^\sharp$ in this environment.

Required operators on the numeric abstraction (3/4)

$$\left\{ (n_1, n_2) \mid n_1 \in \gamma_{\mathsf{Num}}(n_1^\sharp),\ n_2 \in \gamma_{\mathsf{Num}}(n_2^\sharp),\ n_1 [\![c]\!]\uparrow_{\mathrm{comp}} n_2 \right\}$$
$$\subseteq \gamma_{\mathsf{Num}}(m_1^\sharp) \times \gamma_{\mathsf{Num}}(m_2^\sharp)$$
$$\text{with } (m_1^\sharp, m_2^\sharp) = [\![c]\!]\downarrow^\sharp_{\mathrm{comp}} (n_1^\sharp, n_2^\sharp)$$

$$\begin{array}{rcl}
[\![n]\!]\downarrow^\sharp_{\mathrm{expr}} (\rho^\sharp, n^\sharp) & = & \begin{cases} \bot_{\mathsf{Env}} & \text{if } \mathrm{const}^\sharp(n) \sqcap^\sharp_{\mathsf{Num}} n^\sharp = \bot_{\mathsf{Num}} \\ \rho^\sharp & \text{otherwise} \end{cases} \\
[\![?]\!]\downarrow^\sharp_{\mathrm{expr}} (\rho^\sharp, n^\sharp) & = & \rho^\sharp \\
[\![x]\!]\downarrow^\sharp_{\mathrm{expr}} (\rho^\sharp, n^\sharp) & = & (\rho^\sharp[x \mapsto \rho^\sharp(x) \sqcap^\sharp_{\mathsf{Num}} n^\sharp]) \\
[\![-e]\!]\downarrow^\sharp_{\mathrm{expr}} (\rho^\sharp, n^\sharp) & = & [\![e]\!]\downarrow^\sharp_{\mathrm{expr}} (\rho^\sharp, [\![-]\!]\uparrow^\sharp_{\mathrm{op}} (n^\sharp)) \\
[\![e_1\ o\ e_2]\!]\downarrow^\sharp_{\mathrm{expr}} (\rho^\sharp, n^\sharp) & = & \left([\![e_1]\!]\downarrow^\sharp_{\mathrm{expr}} (\rho^\sharp, n_1^\sharp) \sqcap^\sharp_{\mathsf{Env}} [\![e_2]\!]\downarrow^\sharp_{\mathrm{expr}} (\rho^\sharp, n_2^\sharp) \right) \\
\multicolumn{3}{r}{\text{with } (n_1^\sharp, n_2^\sharp) = [\![o]\!]\downarrow^\sharp_{\mathrm{op}} (n^\sharp, [\![e_1]\!]\uparrow^\sharp_{\mathrm{expr}} (\rho^\sharp), [\![e_2]\!]\uparrow^\sharp_{\mathrm{expr}} (\rho^\sharp))}
\end{array}$$

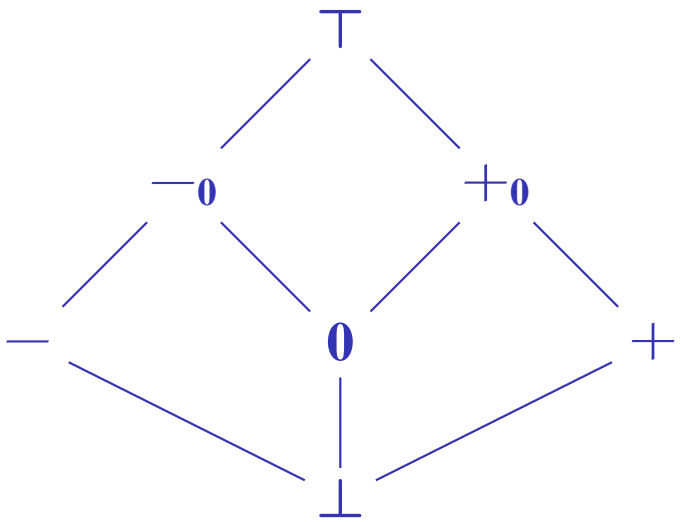
Required operators on the numeric abstraction (4/4)

$$[\![o]\!]\downarrow^\sharp_{\text{op}} \in \text{Num}^\sharp \times \text{Num}^\sharp \times \text{Num}^\sharp \to \text{Num}^\sharp \times \text{Num}^\sharp$$

$[\![o]\!]\downarrow^\sharp_{\text{op}}\ (n^\sharp, n_1^\sharp, n_2^\sharp)$ computes a refinement of two numeric values $n_1^\sharp$ and $n_2^\sharp$ knowing that the result of the binary operation o is approximated by $n^\sharp$ on their concretisations.

$$\forall n^\sharp, n_1^\sharp, n_2^\sharp \in \text{Num}^\sharp,$$
$$\left\{ (n_1, n_2) \mid n_1 \in \gamma_{\text{Num}}(n_1^\sharp),\ n_2 \in \gamma_{\text{Num}}(n_2^\sharp),\ \left(n_1 [\![o]\!]\uparrow_{\text{op}} n_2\right) \in \gamma_{\text{Num}}(n^\sharp) \right\}$$
$$\subseteq \gamma_{\text{Num}}(m_1^\sharp) \times \gamma_{\text{Num}}(m_2^\sharp)$$
$$\text{with } (m_1^\sharp, m_2^\sharp) = [\![o]\!]\downarrow^\sharp_{\text{op}}\ (n^\sharp, n_1^\sharp, n_2^\sharp)$$

Sign abstraction



$$
\begin{aligned}
\gamma_{\mathrm{Num}}(\bot) &= \emptyset \\
\gamma_{\mathrm{Num}}(-) &= \{\, z \mid z < 0 \,\} \\
\gamma_{\mathrm{Num}}(0) &= \{0\} \\
\gamma_{\mathrm{Num}}(+) &= \{\, z \mid z > 0 \,\} \\
\gamma_{\mathrm{Num}}(-_0) &= \{\, z \mid z \leq 0 \,\} \\
\gamma_{\mathrm{Num}}(+_0) &= \{\, z \mid z \geq 0 \,\} \\
\gamma_{\mathrm{Num}}(\top) &= \mathbb{Z}
\end{aligned}
$$

Exercise : give the optimal abstract operators for the numeric abstraction by signs.

$$\text{const}^\sharp(n) = \begin{cases} \end{cases}$$

	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$-^\sharp$							

$+^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$-^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$\times^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$$\text{const}^\sharp(n) = \begin{cases} + & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ - & \text{if } n < 0 \end{cases}$$

	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$-^\sharp$							

$+^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$-^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$\times^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$$\text{const}^\sharp(n) = \begin{cases} + & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ - & \text{if } n < 0 \end{cases}$$

	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$-^\sharp$	$\bot$	$+$	$-$	0	$+_0$	$-_0$	$\top$

$+^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$-^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$\times^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$$\text{const}^\sharp(n) = \begin{cases} + & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ - & \text{if } n < 0 \end{cases}$$

	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$-^\sharp$	$\bot$	$+$	$-$	0	$+_0$	$-_0$	$\top$

$+^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$-$	$\bot$	$-$	$\top$	$-$	$-$	$\top$	$\top$
$+$	$\bot$	$\top$	$+$	$+$	$\top$	$+$	$\top$
0	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$-_0$	$\bot$	$-$	$\top$	$-_0$	$-_0$	$\top$	$\top$
$+_0$	$\bot$	$\top$	$+$	$+_0$	$\top$	$+_0$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

$-^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$\times^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$$\text{const}^\sharp(n) = \begin{cases} + & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ - & \text{if } n < 0 \end{cases}$$

	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$-^\sharp$	$\bot$	$+$	$-$	0	$+_0$	$-_0$	$\top$

$+^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$-$	$\bot$	$-$	$\top$	$-$	$-$	$\top$	$\top$
$+$	$\bot$	$\top$	$+$	$+$	$\top$	$+$	$\top$
0	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$-_0$	$\bot$	$-$	$\top$	$-_0$	$-_0$	$\top$	$\top$
$+_0$	$\bot$	$\top$	$+$	$+_0$	$\top$	$+_0$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

$-^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$-$	$\bot$	$\top$	$-$	$-$	$\top$	$-$	$\top$
$+$	$\bot$	$+$	$\top$	$+$	$+$	$\top$	$\top$
0	$\bot$	$+$	$-$	0	$+_0$	$-_0$	$\top$
$-_0$	$\bot$	$\top$	$-$	$-_0$	$\top$	$-_0$	$\top$
$+_0$	$\bot$	$+$	$\top$	$+_0$	$+_0$	$\top$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

$\times^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$$\text{const}^\sharp(n) = \begin{cases} + & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ - & \text{if } n < 0 \end{cases}$$

	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$-^\sharp$	$\bot$	$+$	$-$	0	$+_0$	$-_0$	$\top$

$+^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$-$	$\bot$	$-$	$\top$	$-$	$-$	$\top$	$\top$
$+$	$\bot$	$\top$	$+$	$+$	$\top$	$+$	$\top$
0	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$-_0$	$\bot$	$-$	$\top$	$-_0$	$-_0$	$\top$	$\top$
$+_0$	$\bot$	$\top$	$+$	$+_0$	$\top$	$+_0$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

$-^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$-$	$\bot$	$\top$	$-$	$-$	$\top$	$-$	$\top$
$+$	$\bot$	$+$	$\top$	$+$	$+$	$\top$	$\top$
0	$\bot$	$+$	$-$	0	$+_0$	$-_0$	$\top$
$-_0$	$\bot$	$\top$	$-$	$-_0$	$\top$	$-_0$	$\top$
$+_0$	$\bot$	$+$	$\top$	$+_0$	$+_0$	$\top$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

$\times^\sharp$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$-$	$\bot$	$+$	$+$	0	$+_0$	$-_0$	$\top$
$+$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
0	$\bot$	0	0	0	0	0	0
$-_0$	$\bot$	$+_0$	$-_0$	0	$+_0$	$-_0$	$\top$
$+_0$	$\bot$	$-_0$	$+_0$	0	$-_0$	$+_0$	$\top$
$\top$	$\bot$	$\top$	$\top$	0	$\top$	$\top$	$\top$

$$[\![=]\!]^\sharp\downarrow (x^\sharp, y^\sharp) =$$

$[\![\neq]\!]^\sharp\downarrow$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$$[\![=]\!]^\sharp\downarrow (x^\sharp, y^\sharp) = (x^\sharp \sqcap^\sharp y^\sharp, x^\sharp \sqcap^\sharp y^\sharp)$$

$[\![\neq]\!]^\sharp\downarrow$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$$[\![=]\!]^\sharp\downarrow (x^\sharp, y^\sharp) = (x^\sharp \sqcap^\sharp y^\sharp, x^\sharp \sqcap^\sharp y^\sharp)$$

$[\![\neq]\!]^\sharp\downarrow$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$
$-$	$(\bot,\bot)$	$(-,-)$	$(-,+)$	$(-,0)$	$(-,-_0)$	$(-,+_0)$	$(-,\top)$
$+$	$(\bot,\bot)$	$(+,-)$	$(+,+)$	$(+,0)$	$(+,-_0)$	$(+,+_0)$	$(+,\top)$
0	$(\bot,\bot)$	$(0,-)$	$(0,+)$	$(\bot,\bot)$	$(0,-)$	$(0,+)$	$(0,\top)$
$-_0$	$(\bot,\bot)$	$(-_0,-)$	$(-_0,+)$	$(-,0)$	$(-_0,-_0)$	$(-_0,+_0)$	$(-_0,\top)$
$+_0$	$(\bot,\bot)$	$(+_0,-)$	$(+_0,+)$	$(+,0)$	$(+_0,-_0)$	$(+_0,+_0)$	$(+_0,\top)$
$\top$	$(\bot,\bot)$	$(\top,-)$	$(\top,+)$	$(\top,0)$	$(\top,-_0)$	$(\top,+_0)$	$(\top,\top)$

$[\![<]\!]^\sharp_\downarrow$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$[\![\leqslant]\!]^\sharp_\downarrow$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$[\![<]\!]^\sharp_\downarrow$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$
$-$	$(\bot,\bot)$	$(-,-)$	$(-,+)$	$(-,0)$	$(-,-_0)$	$(-,+_0)$	$(-,\top)$
$+$	$(\bot,\bot)$	$(\bot,\bot)$	$(+,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(+,+)$	$(+,+)$
0	$(\bot,\bot)$	$(\bot,\bot)$	$(0,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(0,+)$	$(0,+)$
$-_0$	$(\bot,\bot)$	$(-_0,-)$	$(-_0,+)$	$(-_0,0)$	$(-_0,-_0)$	$(-_0,+_0)$	$(-_0,\top)$
$+_0$	$(\bot,\bot)$	$(\bot,\bot)$	$(+_0,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(+_0,+_0)$	$(+_0,+)$
$\top$	$(\bot,\bot)$	$(-,-)$	$(\top,+)$	$(-,0)$	$(-,-_0)$	$(\top,+_0)$	$(\top,\top)$

$[\![\leqslant]\!]^\sharp_\downarrow$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$[\![<]\!]^\sharp_\downarrow$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$
$-$	$(\bot,\bot)$	$(-,-)$	$(-,+)$	$(-,0)$	$(-,-_0)$	$(-,+_0)$	$(-,\top)$
$+$	$(\bot,\bot)$	$(\bot,\bot)$	$(+,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(+,+)$	$(+,+)$
0	$(\bot,\bot)$	$(\bot,\bot)$	$(0,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(0,+)$	$(0,+)$
$-_0$	$(\bot,\bot)$	$(-_0,-)$	$(-_0,+)$	$(-_0,0)$	$(-_0,-_0)$	$(-_0,+_0)$	$(-_0,\top)$
$+_0$	$(\bot,\bot)$	$(\bot,\bot)$	$(+_0,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(+_0,+_0)$	$(+_0,+)$
$\top$	$(\bot,\bot)$	$(-,-)$	$(\top,+)$	$(-,0)$	$(-,-_0)$	$(\top,+_0)$	$(\top,\top)$

$[\![\leqslant]\!]^\sharp_\downarrow$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$
$-$	$(\bot,\bot)$	$(-,-)$	$(-,+)$	$(-,0)$	$(-,-_0)$	$(-,+_0)$	$(-,\top)$
$+$	$(\bot,\bot)$	$(\bot,\bot)$	$(+,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(+,+)$	$(+,+)$
0	$(\bot,\bot)$	$(\bot,\bot)$	$(0,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(0,+_0)$	$(0,+_0)$
$-_0$	$(\bot,\bot)$	$(-_0,-)$	$(-_0,+)$	$(-_0,0)$	$(-_0,-_0)$	$(-_0,+_0)$	$(-_0,\top)$
$+_0$	$(\bot,\bot)$	$(\bot,\bot)$	$(+_0,+)$	$(0,0)$	$(0,0)$	$(+_0,+_0)$	$(+_0,+_0)$
$\top$	$(\bot,\bot)$	$(-,-)$	$(\top,+)$	$(-_0,0)$	$(-_0,-_0)$	$(\top,+_0)$	$(\top,\top)$

$[\![+]\!]^\sharp\downarrow(+,\cdot,\cdot)$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$\ldots$

$[\![\times]\!]^\sharp\downarrow(0,\cdot,\cdot)$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$[\![+]\!]^\sharp\downarrow\ (+,\cdot,\cdot)$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$
$-$	$(\bot,\bot)$	$(\bot,\bot)$	$(-,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(-,+)$	$(-,+)$
$+$	$(\bot,\bot)$	$(+,-)$	$(+,+)$	$(+,0)$	$(+,-_0)$	$(+,+_0)$	$(+,\top)$
0	$(\bot,\bot)$	$(\bot,\bot)$	$(0,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(0,+)$	$(0,+)$
$-_0$	$(\bot,\bot)$	$(\bot,\bot)$	$(-_0,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(-_0,+)$	$(-_0,+)$
$+_0$	$(\bot,\bot)$	$(+,-)$	$(+_0,+)$	$(+,0)$	$(+,-_0)$	$(+_0,+_0)$	$(+_0,\top)$
$\top$	$(\bot,\bot)$	$(+,-)$	$(\top,+)$	$(+,0)$	$(+,-_0)$	$(\top,+_0)$	$(\top,\top)$

$\ldots$

$[\![\times]\!]^\sharp\downarrow\ (0,\cdot,\cdot)$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$							
$-$							
$+$							
0							
$-_0$							
$+_0$							
$\top$							

$[\![+]\!]^\sharp\downarrow(+,\cdot,\cdot)$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$
$-$	$(\bot,\bot)$	$(\bot,\bot)$	$(-,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(-,+)$	$(-,+)$
$+$	$(\bot,\bot)$	$(+,-)$	$(+,+)$	$(+,0)$	$(+,-_0)$	$(+,+_0)$	$(+,\top)$
0	$(\bot,\bot)$	$(\bot,\bot)$	$(0,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(0,+)$	$(0,+)$
$-_0$	$(\bot,\bot)$	$(\bot,\bot)$	$(-_0,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(-_0,+)$	$(-_0,+)$
$+_0$	$(\bot,\bot)$	$(+,-)$	$(+_0,+)$	$(+,0)$	$(+,-_0)$	$(+_0,+_0)$	$(+_0,\top)$
$\top$	$(\bot,\bot)$	$(+,-)$	$(\top,+)$	$(+,0)$	$(+,-_0)$	$(\top,+_0)$	$(\top,\top)$

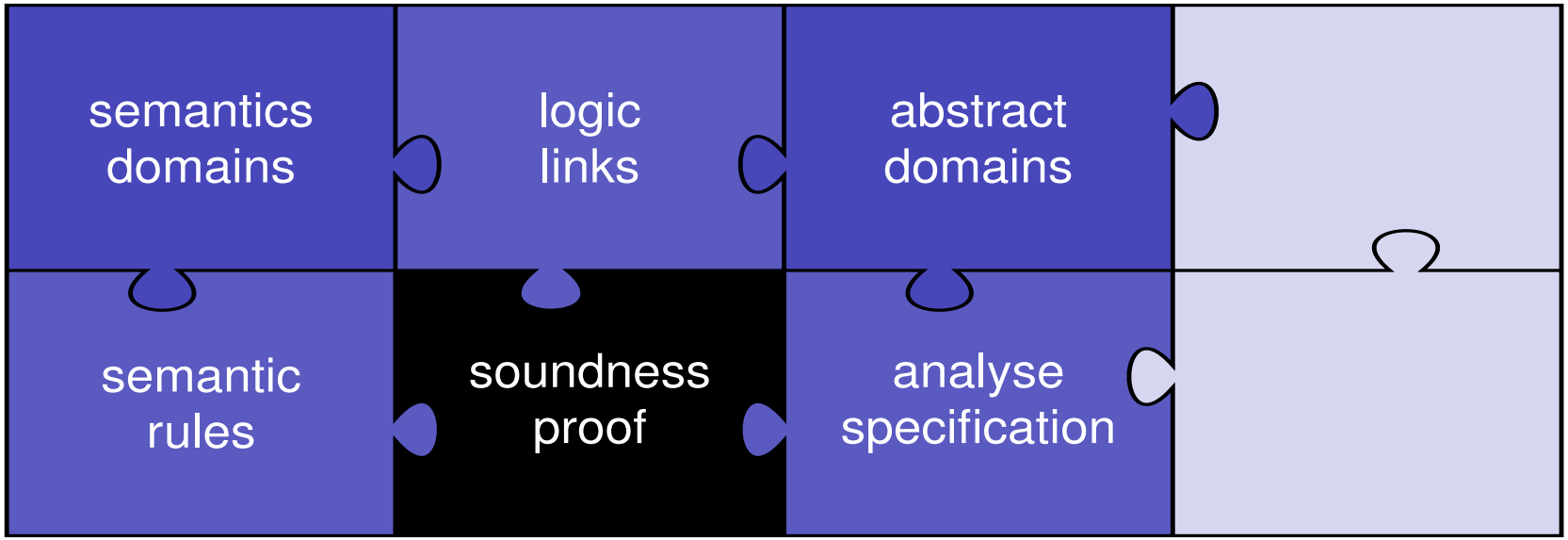
...

$[\![\times]\!]^\sharp\downarrow(0,\cdot,\cdot)$	$\bot$	$-$	$+$	0	$-_0$	$+_0$	$\top$
$\bot$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$
$-$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(-,0)$	$(-,0)$	$(-,0)$	$(-,0)$
$+$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(+,0)$	$(+,0)$	$(+,0)$	$(+,0)$
0	$(\bot,\bot)$	$(0,)$	$(0,+)$	$(0,0)$	$(0,-_0)$	$(0,+_0)$	$(0,\top)$
$-_0$	$(\bot,\bot)$	$(0,-)$	$(0,+)$	$(-_0,0)$	$(-_0,-_0)$	$(-_0,+_0)$	$(-_0,\top)$
$+_0$	$(\bot,\bot)$	$(0,-)$	$(0,+)$	$(+_0,0)$	$(+_0,-_0)$	$(+_0,+_0)$	$(+_0,\top)$
$\top$	$(\bot,\bot)$	$(0,-)$	$(0,+)$	$(\top,0)$	$(\top,0)$	$(\top,0)$	$(\top,\top)$

Final specification

```
Definition cstr2prop (s:GlobalLattice.t) (c:cstr) : Prop :=
  MemLattice.order
    (c.(constraint) (GlobalLattice.get s c.(source)))
    (GlobalLattice.get s c.(target)).


Definition Spec (p:program) (s:GlobalLattice.t) : Prop :=
  MemLattice.order (Base nil,LocalVarLattice.bottom) (GlobalLattice.get s
  ∀ pc instr,
    instr_at p pc = Some instr →
    ∀ c, In c (gen_one_constraint pc instr) → cstr2prop s c.
```

Building a certified static analyser



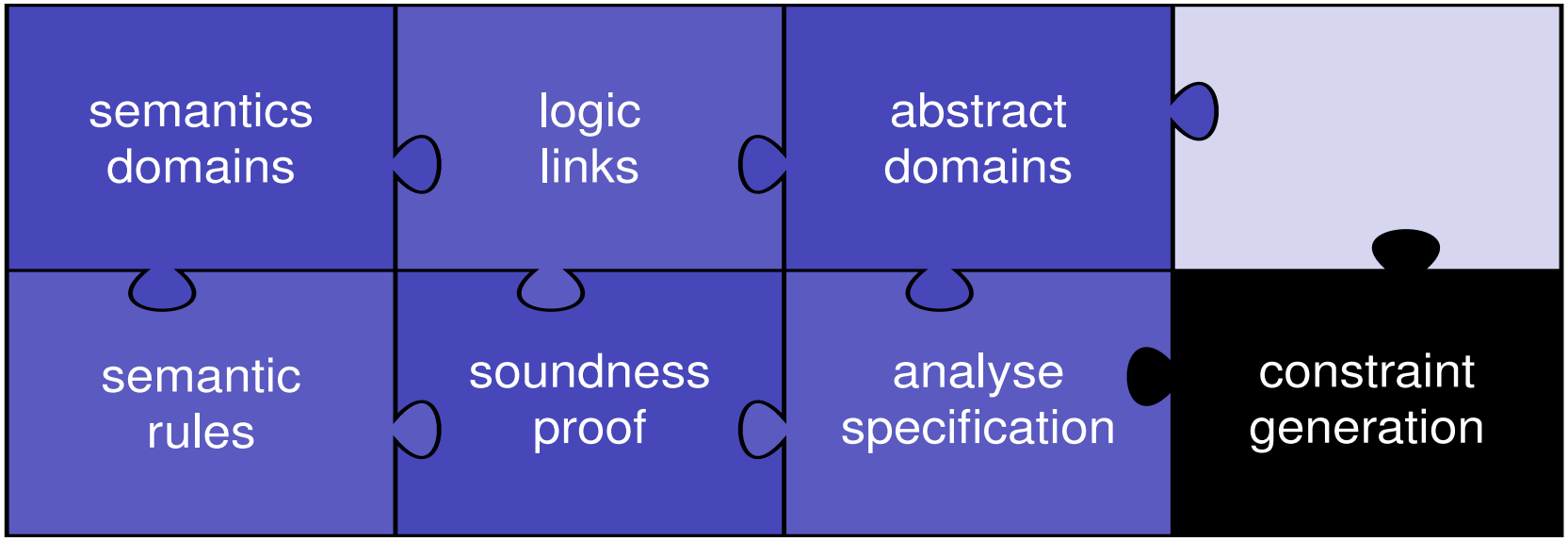
$$\forall P,\ \forall s^\sharp,\quad F^\sharp_P(s^\sharp) \sqsubseteq^\sharp s^\sharp \ \Rightarrow\ [\![P]\!] \subseteq \gamma(s^\sharp)$$

- easy proof, but tedious
- one proof by instruction: a long work for real langages

The main technical lemma

```
Lemma step_correct : ∀ p F i ins h l s s2,
  instr_at p i = Some ins →
  (∀ c, In c (gen_one_constraint i ins) → cstr2prop F c) →
  (St i s l h) -[p]→ s2 →
  wf_state (St i s l h) →
  gamma p F (St i s l h) →
  gamma p F s2.
```

Building a certified static analyser

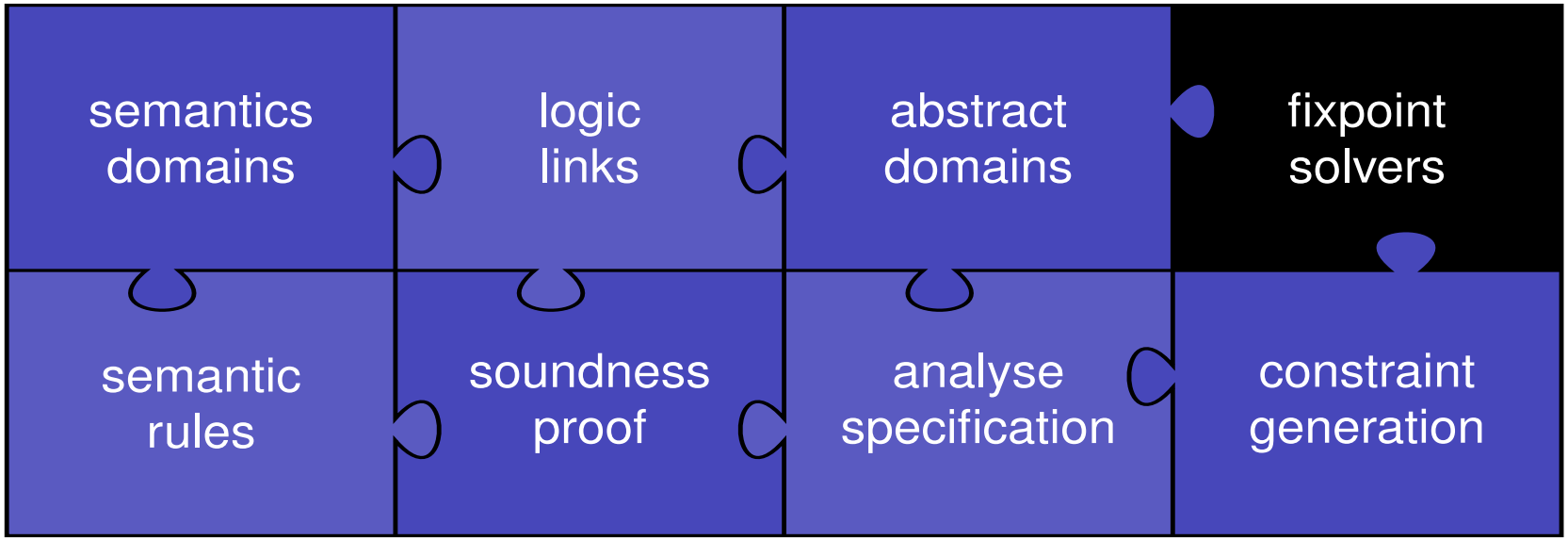


- collects all constraints in a program
- generic tool

Correctness of the constraint generation

```
Fixpoint collect_all_cstr (f:pc→instruction→list cstr)
  (l:list (pc*instruction)) : list cstr :=
  match l with
    | nil ⇒ nil
    | (i,ins)::q ⇒ (f i ins)++(collect_all_cstr f q)
  end.

Lemma In_collect_all_cstr : ∀ f l pc instr c,
  instr_at l pc = Some instr →
  In c (f pc instr) →
  In c (collect_all_cstr f l).

...

Lemma Func_correct : ∀ p s,
  GlobalLattice.order (Func p s) s → Spec p s.
```

Building a certified static analyser



$$\forall P,\ \exists s^\sharp,\quad F_P^\sharp(s^\sharp) \sqsubseteq s^\sharp$$

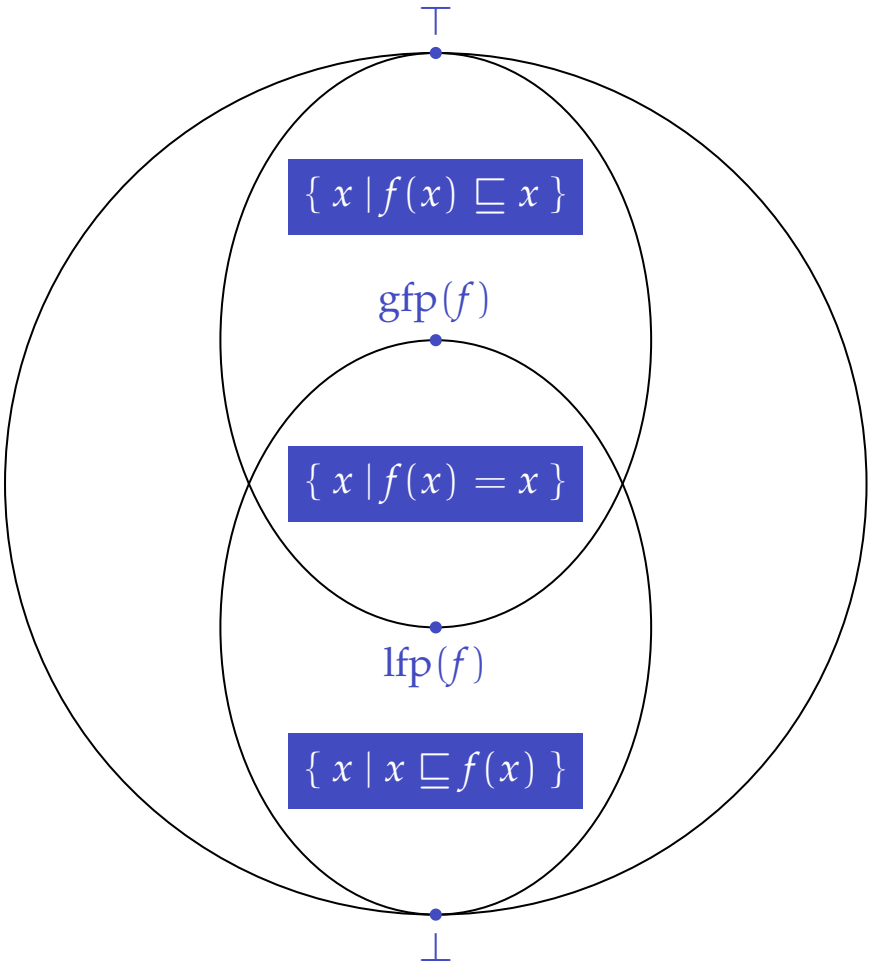
Two techniques of iterative computation

- traditional least (post)-fixpoint computation

$$\bot \longrightarrow F_P^\sharp(\bot) \longrightarrow {F_P^\sharp}^2(\bot) \longrightarrow \cdots \mathrm{lfp}(F_P^\sharp)$$

- post-fixpoint computation by widening/narrowing

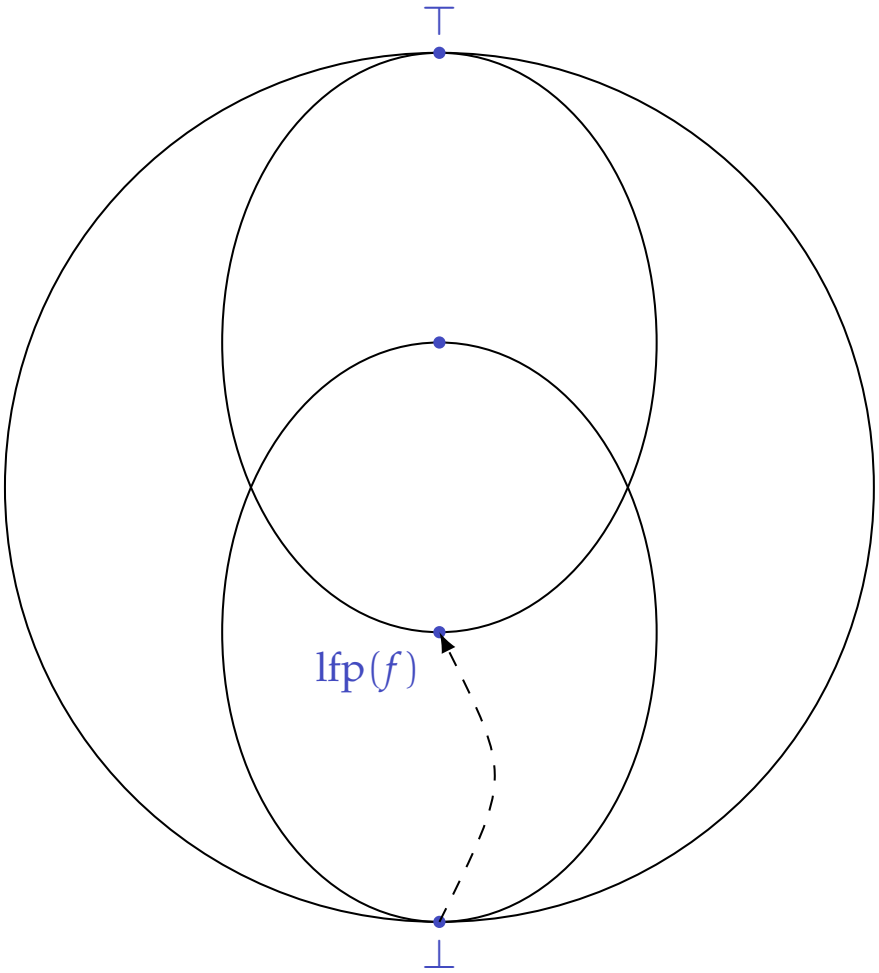
Knaster-Tarski theorem



Knaster-Tarski theorem states that in any complete lattice, there exists a smallest fixpoint for any monotone function.

- ensures existence,
- dosn't say how to compute it.

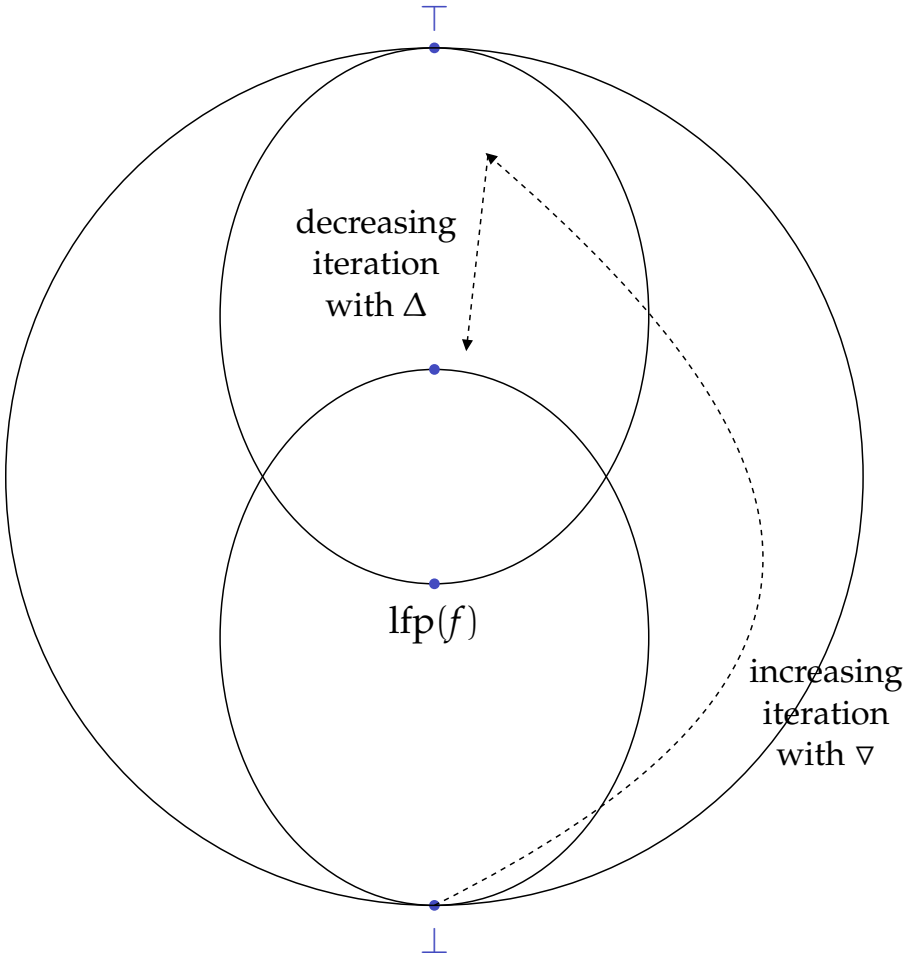
Kleen theorem



Kleen theorem says that, if the function is continuous, iterating the function from the smallest element of the lattice indeed reaches the smallest fixpoint.

- may require an infinite number of iteration...
- always terminates if there is no infinite ascending chain (*ascending chain condition*).

Widening/Narrowing



Any over-approximation of $\mathrm{lfp}\, f$ is sound.

- First an ascending iteration reaches the post-fixpoint zone,
- Then a descending iteration refines the results

Abstraction by intervals

$$\text{Int} \stackrel{\text{def}}{=} \{\ [a,b] \mid a,b \in \overline{\mathbb{Z}},\ a \leqslant b\ \} \cup \{\bot\} \quad \text{with } \overline{\mathbb{Z}} = \mathbb{Z} \cup \{-\infty, +\infty\}$$

Lattice :

$$\frac{I \in \text{Int}}{\bot \sqsubseteq_{\text{Int}} I} \qquad \frac{c \leqslant a \quad b \leqslant d \quad a,b,c,d \in \overline{\mathbb{Z}}}{[a,b] \sqsubseteq_{\text{Int}} [c,d]}$$

$$\begin{aligned} I \sqcup_{\text{Int}} \bot &\stackrel{\text{def}}{=} = I,\ \forall I \in \text{Int} \\ \bot \sqcup_{\text{Int}} I &\stackrel{\text{def}}{=} = I,\ \forall I \in \text{Int} \\ [a,b] \sqcup_{\text{Int}} [c,d] &\stackrel{\text{def}}{=} [\min(a,c), \max(b,d)] \end{aligned}$$

$$\begin{aligned} I \sqcap_{\text{Int}} \bot &\stackrel{\text{def}}{=} \bot,\ \forall I \in \text{Int} \\ \bot \sqcap_{\text{Int}} I &\stackrel{\text{def}}{=} \bot,\ \forall I \in \text{Int} \\ [a,b] \sqcap_{\text{Int}} [c,d] &\stackrel{\text{def}}{=} \rho_{\text{Int}}([\max(a,c), \min(b,d)]) \end{aligned}$$

with $\rho_{\text{Int}} \in (\overline{\mathbb{Z}} \times \overline{\mathbb{Z}}) \to \text{Int}$ defined by

$$\rho_{\text{Int}}(a,b) = \begin{cases} [a,b] & \text{if } a \leqslant b, \\ \bot & \text{otherwise} \end{cases}$$

$$\begin{array}{lcl} \bot_{\text{Int}} & \stackrel{\text{def}}{=} & \bot \\ \top_{\text{Int}} & \stackrel{\text{def}}{=} & [-\infty, +\infty] \end{array}$$

$$\begin{array}{rcl} \gamma_{\text{Int}}(\bot) & \stackrel{\text{def}}{=} & \emptyset \\ \gamma_{\text{Int}}([a,b]) & \stackrel{\text{def}}{=} & \{\, z \in \mathbb{Z} \mid a \leqslant z \text{ et } z \leqslant b \,\} \end{array}$$

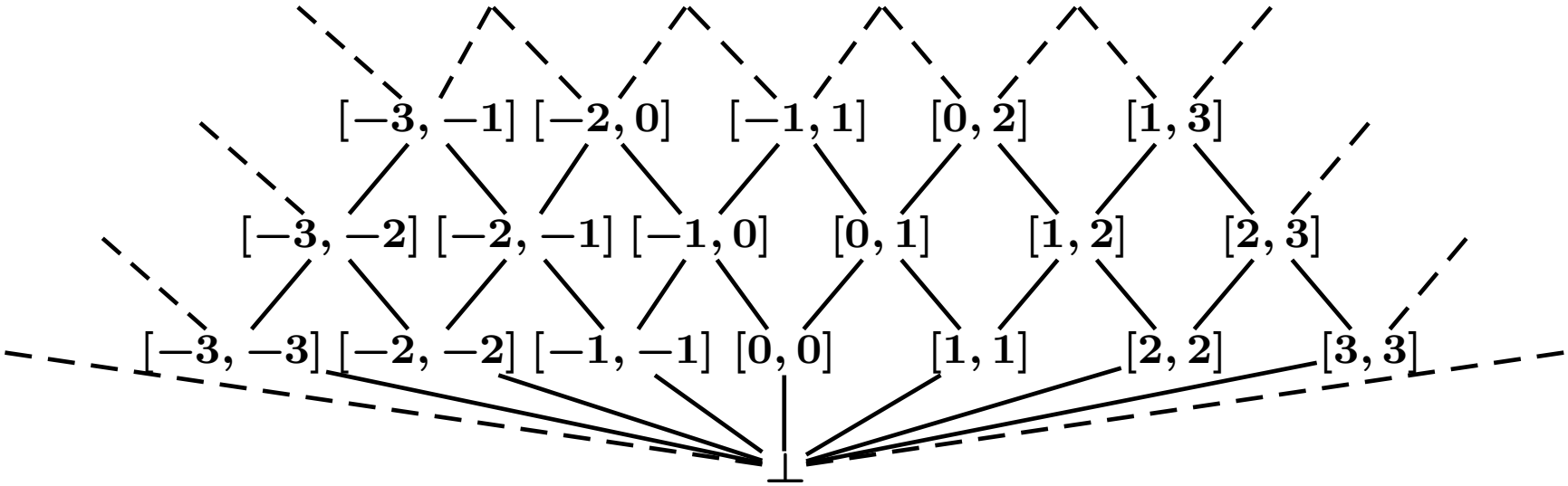
All the other operators are *stricts*: they return $\bot$ if one of their arguments is $\bot$.

$$
\begin{aligned}
[\![+]\!]^{\sharp}_{\text{op}}\ ([a,b],[c,d]) &= [a+c, b+d] \\
[\![-]\!]^{\sharp}_{\text{op}}\ ([a,b],[c,d]) &= [a-d, b-c] \\
[\![\times]\!]^{\sharp}_{\text{op}}\ ([a,b],[c,d]) &= [\min(ac,ad,bc,bd), \max(ac,ad,bc,bd)]
\end{aligned}
$$

$$
\begin{aligned}
[\![+]\!]^{\sharp}_{\text{op}}\ ([a,b],[c,d],[e,f]) &= (\rho(\max(c, a-f), \min(d, b-e)), \\
&\qquad \rho(\max(e, a-d), \min(f, b-c))) \\
[\![-]\!]^{\sharp}_{\text{op}}\ ([a,b],[c,d],[e,f]) &= (\rho(\max(c, a+e), \min(d, b+f)), \\
&\qquad \rho(\max(e, c-b), \min(f, d-a))) \\
[\![*]\!]^{\sharp}_{\text{op}}\ ([a,b],[c,d],[e,f]) &= ([c,d],[e,f])
\end{aligned}
$$

$$
\begin{aligned}
[\![=]\!]^{\sharp}_{\text{comp}}\ ([a,b],[c,d]) &= ([a,b] \sqcap_{\text{Int}} [c,d], [a,b] \sqcap_{\text{Int}} [c,d]) \\
[\![<]\!]^{\sharp}_{\text{comp}}\ ([a,b],[c,d]) &= ([a,b] \sqcap_{\text{Int}} [-\infty, d-1], [a+1, +\infty] \sqcap_{\text{Int}} [c,d]) \\
[\![\leqslant]\!]^{\sharp}_{\text{comp}}\ ([a,b],[c,d]) &= ([a,b] \sqcap_{\text{Int}} [-\infty, d], [a, +\infty] \sqcap_{\text{Int}} [c,d]) \\
[\![\neq]\!]^{\sharp}_{\text{comp}}\ ([a,b],[c,d]) &= ? \textit{ exercice...} \\
\text{const}(n)^{\sharp} &= [n,n]
\end{aligned}
$$

Convergence problem



Such a lattice does not satisfy the ascending chain condition.

Example of infinite increasing chain :

$$\bot \sqsubset [0,0] \sqsubset [0,1] \sqsubset \cdots \sqsubset [0,n] \sqsubset \cdots$$

Solution : dynamic approximation

- we extrapolate the limit thanks to a widening operator ∇

Widening

Idea: the standard iteration is of the form

$$x^0 = \bot, x^{n+1} = F(x^n) \quad = x^n \sqcup F(x^n)$$

We will replace it by something of the form

$$y^0 = \bot, y^{n+1} = y^n \nabla F(y^n)$$

such that

(i) (y^n) is increasing,

(ii) $x^n \sqsubseteq y^n$, for all n,

(iii) and (y^n) stabilizes after a finite number of steps.

But we also want a ∇ operator that is independent of F.

Widening : definition

A widening is an operator $\nabla : L \times L \to L$ such that

- $\forall x, x' \in L, x \sqcup x' \sqsubseteq x \nabla x'$ (implies (i) & (ii))
- If $x^0 \sqsubseteq x^1 \sqsubseteq \ldots$ is an increasing chain, then the increasing chain $y^0 = x^0, y^{n+1} = y^n \nabla x^{n+1}$ stabilizes after a finite number of steps (implies (iii)).

Usage: we replace $x^0 = \bot, x^{n+1} = F(x^n)$
by $y^0 = \bot, y^{n+1} = y^n \nabla F(y^n)$

Widening: theorem

Theorem

Let L a complete lattice, $F : L \to L$ a monotone function and $\nabla : L \times L \to L$ a widening operator. The chain $y^0 = \bot, y^{n+1} = y^n \nabla F(y^n)$ stabilizes after a finite number of steps towards a post-fixpoint y of F.

Corollary: $\mathrm{lfp}(F) \sqsubseteq y$.

Example: widening on intervals

Idea: as soon as a bound is not stable, we extrapolate it by $+\infty$ (or $-\infty$).
After such an extrapolation, the bound can't move any more.

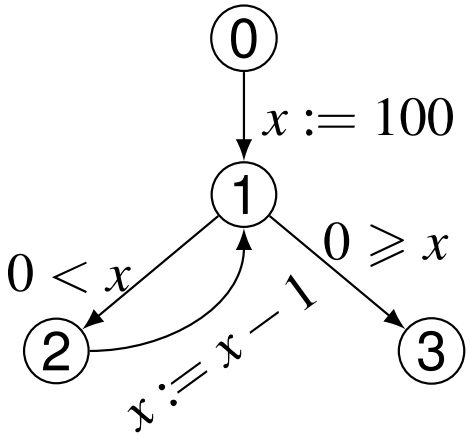
Definition:

$$
\begin{array}{lcl}
[a,b]\nabla_{\text{Int}}[a',b'] & = & [\quad \text{if } a' < a \text{ then } -\infty \text{ otherwise } a, \\
 & & \quad\ \text{if } b' > b \text{ then } +\infty \text{ otherwise } b\,] \\
\bot\nabla_{\text{Int}}[a',b'] & = & [a',b'] \\
I\ \nabla_{\text{Int}}\ \bot & = & I
\end{array}
$$

Examples:
$[-3,4]\nabla_{\text{Int}}[-3,2] = [-3,4]$
$[-3,4]\nabla_{\text{Int}}[-3,5] = [-3,+\infty]$

Example

```
x := 100;

while 0 < x {

   x := x − 1;
}
```



$$X_1 = [100, 100] \sqcup_{\text{Int}} \left(X_2 -^{\sharp} [1, 1]\right)$$

$$X_2 = [1, +\infty] \sqcap_{\text{Int}} X_1$$

$$X_3 = [-\infty, 0] \sqcap_{\text{Int}} X_1$$

Example : without widening

$$
\begin{aligned}
X_1 &= [100, 100] \sqcup_{\text{Int}} \left(X_2 -^\sharp [1, 1]\right) \\
X_2 &= [1, +\infty] \sqcap_{\text{Int}} X_1 \\
X_3 &= [-\infty, 0] \sqcap_{\text{Int}} X_1
\end{aligned}
$$

Iteration strategy : $1 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow \cdots$

$$
\begin{aligned}
X_1^0 &= \bot \quad & X_1^{n+1} &= [100, 100] \sqcup_{\text{Int}} \left(X_2^n -^\sharp [1, 1]\right) \\
X_2^0 &= \bot \quad & X_2^{n+1} &= [1, +\infty] \sqcap_{\text{Int}} X_1^{n+1} \\
X_3^0 &= \bot \quad & X_3^{n+1} &= [-\infty, 0] \sqcap_{\text{Int}} X_1^{n+1}
\end{aligned}
$$

X_1	$\bot$	$[100, 100]$	$[99, 100]$	$[98, 100]$	$[97, 100]$	$\cdots$	$[1, 100]$	$[0, 100]$
X_2	$\bot$	$[100, 100]$	$[99, 100]$	$[98, 100]$	$[97, 100]$	$\cdots$	$[1, 100]$	$[1, 100]$
X_3	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\cdots$	$\bot$	$[0, 0]$

Example : with widening at each nodes of the cfg

$$
\begin{aligned}
X_1 &= [100,100] \sqcup_{\text{Int}} \left(X_2 -^\sharp [1,1]\right) \\
X_2 &= [1,+\infty] \sqcap_{\text{Int}} X_1 \\
X_3 &= [-\infty,0] \sqcap_{\text{Int}} X_1
\end{aligned}
$$

Iteration strategy : $1 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow \cdots$

$$
\begin{array}{ll}
X_1^0 = \bot & X_1^{n+1} = X_1^n \nabla_{\text{Int}} \left([100,100] \sqcup_{\text{Int}} \left(X_2^n -^\sharp [1,1]\right)\right) \\
X_2^0 = \bot & X_2^{n+1} = X_2^n \nabla_{\text{Int}} \left([1,+\infty] \sqcap_{\text{Int}} X_1^{n+1}\right) \\
X_3^0 = \bot & X_3^{n+1} = X_3^n \nabla_{\text{Int}} \left([-\infty,0] \sqcap_{\text{Int}} X_1^{n+1}\right)
\end{array}
$$

X_1	$\bot$	$[100,100]$	$[-\infty,100]$
X_2	$\bot$	$[100,100]$	$[-\infty,100]$
X_3	$\bot$	$\bot$	$[-\infty,0]$

Improving fixpoint approximation

Idea: iterating a little more may help...

Theorem

Let $(A, \sqsubseteq, \sqcup, \sqcap)$ *a complete lattice,* f *a monotone operator on* A *and* a *a post-fixpoint of* f*. The chain* $(x_n)_n$ *defined by* $\begin{cases} x_0 & = & a \\ x_{k+1} & = & f(x_k) \end{cases}$ *admits for limit* $(\bigsqcup\{x_n\})$ *the greatest fixpoint of* f *lower then* a *(noted* $\mathrm{gfp}_a(f)$*). In particular,* $\mathrm{lfp}(f) \sqsubseteq \bigsqcup\{x_n\}$*. Each intermediate step is a correct approximation:*

$$\forall k,\ \mathrm{lfp}(f) \sqsubseteq \mathrm{gfp}_a(f) \sqsubseteq x_k \sqsubseteq a$$

Narrowing : definition

A *narrowing* is an operator $\Delta : L \times L \to L$ such that

- $\forall x, x' \in L, x \sqcap x' \sqsubseteq x \Delta x' \sqsubseteq x$
- If $x^0 \sqsupseteq x^1 \sqsupseteq \ldots$ is a decreasing chain, then the increasing chain $y^0 = x^0, y^{n+1} = y^n \Delta x^{n+1}$ stabilizes after a finite number of steps.

Narrowing: decreasing iteration

Theorem

If Δ is a narrowing operator on a poset $(A, \sqsubseteq)$, if f is a monotone operator on A and a is a post-fixpoint of f then the chain $(x_n)_n$ defined by

$$\begin{cases} x_0 & = & a \\ x_{k+1} & = & x_k \Delta f(x_k) \end{cases}$$

stabilizes after a finite number of steps on a post-fixpoint of f lower than a.

Narrowing on intervals

$$
\begin{array}{rcl}
[a,b]\Delta_{\text{Int}}[c,d] & = & [\text{if } a = -\infty \text{ then } c \text{ else } a \ ; \ \text{ if } b = +\infty \text{ then } d \text{ else } b] \\
I \ \Delta_{\text{Int}} \ \bot & = & \bot \\
\bot \ \Delta_{\text{Int}} \ I & = & \bot
\end{array}
$$

Intuition : we only improve infinite bounds.

In practice : a few standard iterations already improve a lot the result that has been obtained after widening...

- Assignments by constants and conditional guards make the decreasing iterations efficient: they *filter* the (too big) approximations computed by the widening

Example : with narrowing at each nodes of the cfg

$$
\begin{aligned}
X_1 &= [100,100] \sqcup_{\text{Int}} \left(X_2 -^\sharp [1,1]\right)\\
X_2 &= [1,+\infty] \sqcap_{\text{Int}} X_1\\
X_3 &= [-\infty,0] \sqcap_{\text{Int}} X_1
\end{aligned}
$$

Iteration strategy : $1 \to 2 \to 3 \to 1 \to 2 \to \cdots$

$$
\begin{array}{ll}
X_1^0 = [-\infty,100] & X_1^{n+1} = X_1^n \Delta_{\text{Int}} \left([100,100] \sqcup_{\text{Int}} \left(X_2^n -^\sharp [1,1]\right)\right)\\
X_2^0 = [-\infty,100] & X_2^{n+1} = X_2^n \Delta_{\text{Int}} \left([1,+\infty] \sqcap_{\text{Int}} X_1^{n+1}\right)\\
X_3^0 = [-\infty,0] & X_3^{n+1} = X_3^n \Delta_{\text{Int}} \left([-\infty,0] \sqcap_{\text{Int}} X_1^{n+1}\right)
\end{array}
$$

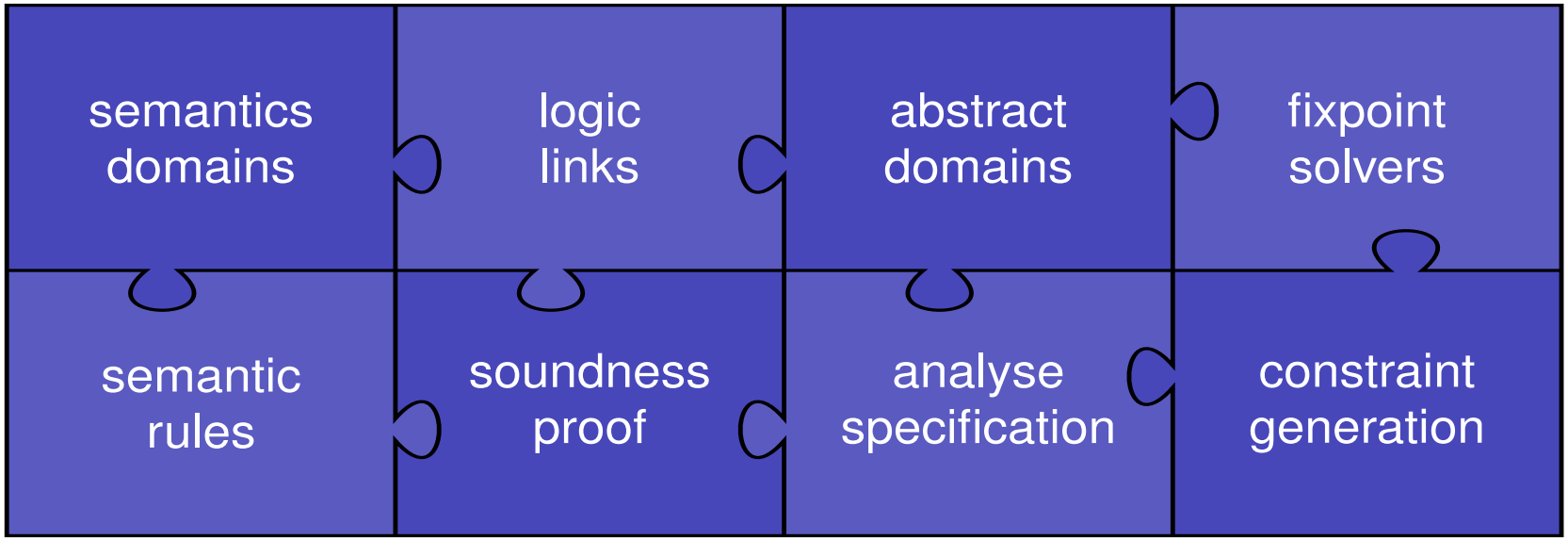
X_1	$[-\infty,100]$	$[-\infty,100]$	$[0,100]$
X_2	$[-\infty,100]$	$[1,100]$	$[1,100]$
X_3	$[-\infty,0]$	$[-\infty,0]$	$[0,0]$

A generic fixpoint solver

It takes the form of a Coq functors.

```
Module PostFixPointSolver (L:Lattice).
  Definition pfp (f : L.t → L.t) :  { x:L.t | L.order (f x) x }
    (* ... omitted ... *)
End PostFixPointSolver.
```

Building a certified static analyser



Final result

$$\left.\begin{array}{l}\forall P,\ \forall s^\sharp,\ F^\sharp_P(s^\sharp) \sqsubseteq s^\sharp \Rightarrow [\![P]\!] \subseteq \gamma(s^\sharp) \\ \forall P,\ \exists s^\sharp,\ F^\sharp_P(s^\sharp) \sqsubseteq s^\sharp\end{array}\right\} \quad \forall P,\ \exists s^\sharp,\ [\![P]\!] \subseteq \gamma(s^\sharp)$$

In Coq : `analyse : ∀ p:program, { s:abstate | sem(P) ⊆ gamma(P,s) }`

In Caml : `analyse : program → abstate`

Conclusions

We conclude our development by defining a verifier

- that runs the analyzer
- and then makes abstract checks on it.

```
Definition verifier (p:program) :
  { b:bool | b = true → ∀ st, ReachableStates p st → Safe st}.
Proof.
  intros p.
  destruct (analyzer p) as [F h].
  exist (check p F).
  exact (check_true_implies_PreSafe p F h).
Defined.
```

Conclusions

We have demonstrated how to construct a non-trivial, provably correct abstract interpreter inside Coq.

There are 3 kinds of bugs in static analysers:

- soundness
- termination
- precision

We have adresses the two firsts. Adressing the last one would require a formalisation of the Galois connexion framework.

`http://www.irisa.fr/celtique/pichardie/fosad09/`