Formal verification of a C static analyzer

Sandrine Blazy

joint work with J.-H. Jourdan, V. Laporte, X. Leroy, D. Pichardie




IFIP WG 2.11, 2015-01-20

Background: verifying a compiler

Compiler + proof that the compiler does not introduce bugs

CompCert, a moderately optimizing C compiler usable for critical embedded software

Using the Coq proof assistant, we prove the following semantic preservation property:

> For all source programs S and compiler-generated code C,
> if the compiler generates machine code C from source S,
> without reporting a compilation error,
> then «C behaves like S».

Compiler written from scratch, along with its proof; not trying to prove an existing compiler

CompCert main correctness theorem

If the source program can not go wrong, then the behavior of the generated assembly code is exactly one of the behaviors of the source program.

Behavior = termination / divergence / undefined («going wrong»)
+ trace of I/O operations performed

```
Theorem transf_c_program_is_refinement:
forall p tp, transf_c_program p = OK tp →
(forall behv, exec_C_program p behv → not_wrong behv) →
(forall behv,  exec_Asm_program tp behv → exec_C_program p behv).
```

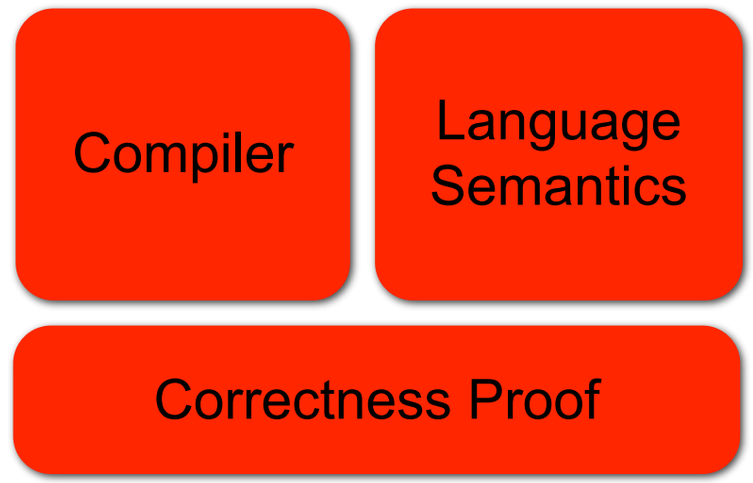
The generated assembly code can not wrong.

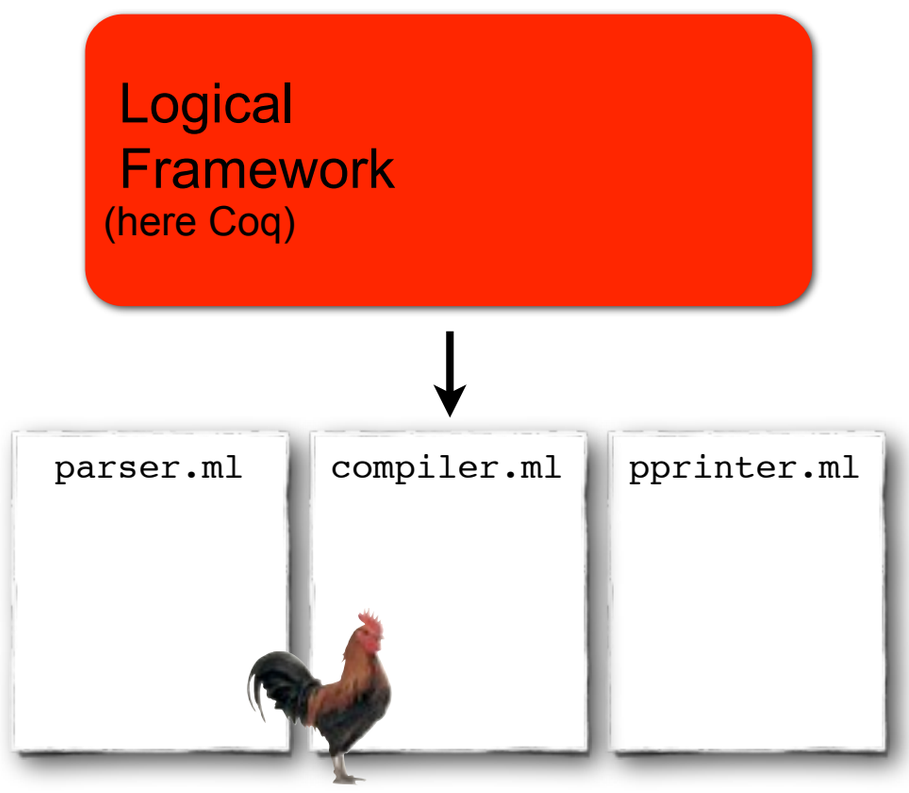
Proof methodology

- The compiler is written inside the purely functional Coq programming language.
- We state its correctness w.r.t. a formal specification of the language semantics.
- We interactively and mechanically prove this.
- We decompose the proof in proofs for each compiler pass.
- We extract a Caml implementation of the compiler.

Proof methodology

- The compiler is written inside the purely functional Coq programming language.
- We state its correctness w.r.t. a formal specification of the language semantics.
- We interactively and mechanically prove this.
- We decompose the proof in proofs for each compiler pass.
- We extract a Caml implementation of the compiler.





From CompCert to formally verified static analysis

The Verasco project

INRIA Celtique, Gallium, Abstraction, Toccata + VERIMAG + Airbus

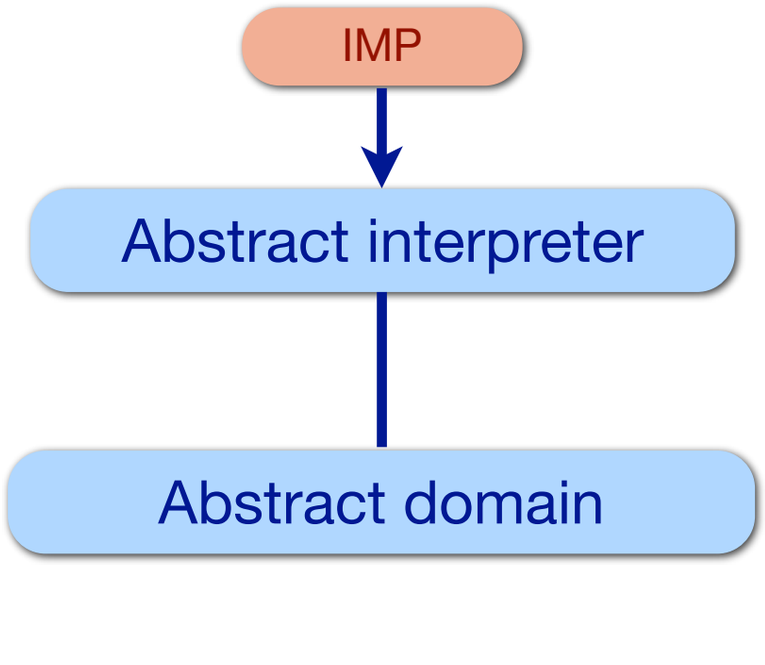
---

Goal: develop and verify in Coq a realistic static analyzer by abstract interpretation

- Language analyzed: the CompCert subset of C
  - no dynamic allocation, no recursion
- Based on abstract interpretation
  - Modular architecture inspired from Astrée's
- Proves the absence of undefined behaviors
  - memory safety
  - no arithmetic exceptions

Slogan: if « CompCert ≈ 1/10th of GCC but formally verified»,
likewise «Verasco ≈1/10th of Astrée but formally verified»

Static analysis formalized in textbooks



- IMP toy language
- No pointer
- Unbounded integers

Example of interface

in ML

```
module type ABDOM = sig
  type ab
  val le : ab → ab → bool
  val top : ab
  val join : ab → ab → ab
  val widen : ab → ab → ab
end
```

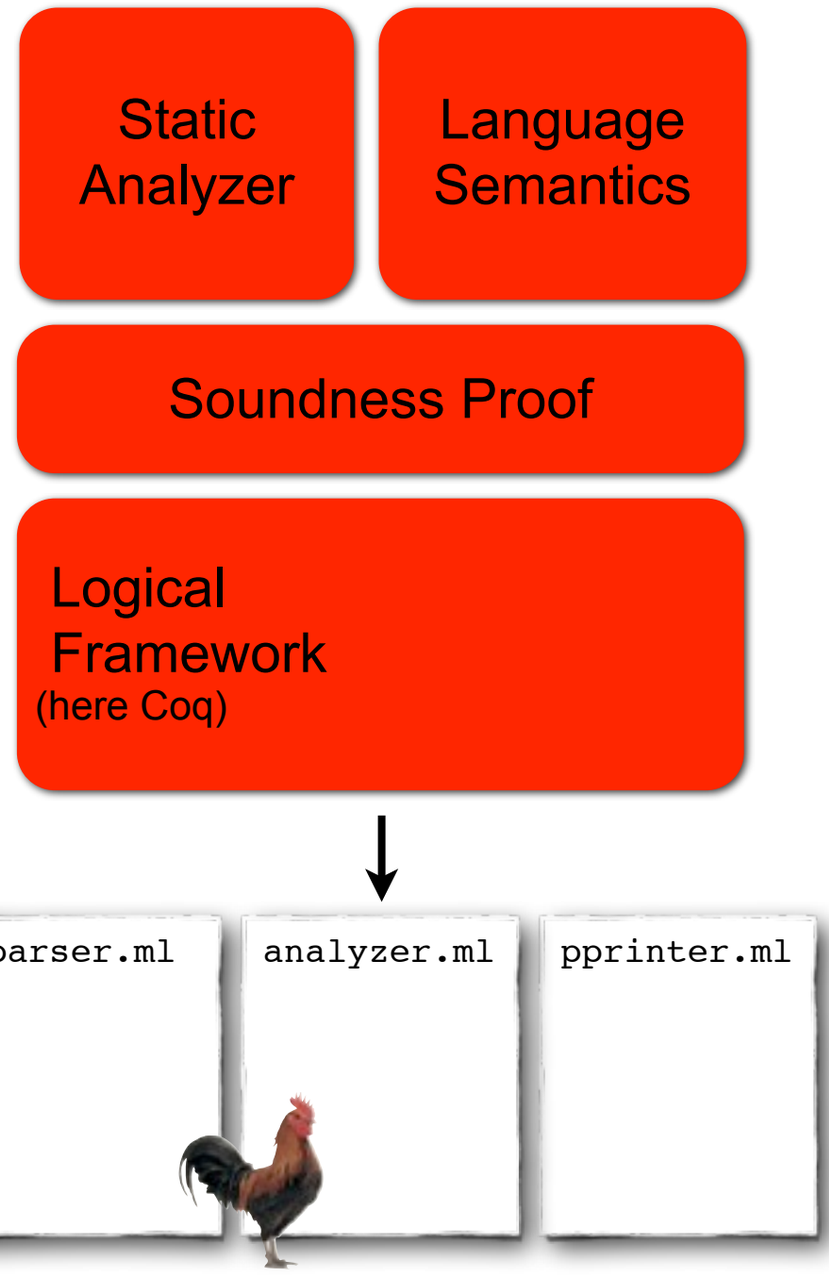
in Coq

```
Class adom (ab:Type) (c:Type) := {
  le : ab → ab → bool;
  top : ab;
  join : ab → ab → ab;
  widen : ab → ab → ab;

  gamma : ab → ℘(c);

  gamma_monotonic : ∀ a1 a2,
      le a1 a2 = true  ⟹
      gamma a1 ⊆ gamma a2;
  gamma_top : ∀ x,
      x ∈ gamma top;
  join_sound : ∀ x y,
      gamma x ∪ gamma y
                ⊆ gamma (join x y)
}
```

Verifying a static analyzer

written in monadic style so that
alarms can be reported during analysis

```
Definition analyzer (p: program) := ...

Theorem analyzer_is_sound :
 ∀ p, analyzer p = Success ->
 (∀ behv, exec_C_program p behv → not_wrong behv).
Proof. ...(* few months later *)...Qed.
Extraction analyzer.
```



A holistic effect with compiler verification

Compiler

Theorem transf_c_program_is_refinement:
forall p tp, transf_c_program p = OK tp →
(forall behv, exec_C_program p behv → not_wrong behv) →
(forall behv, exec_Asm_program tp behv → exec_C_program p behv).

Static analyzer

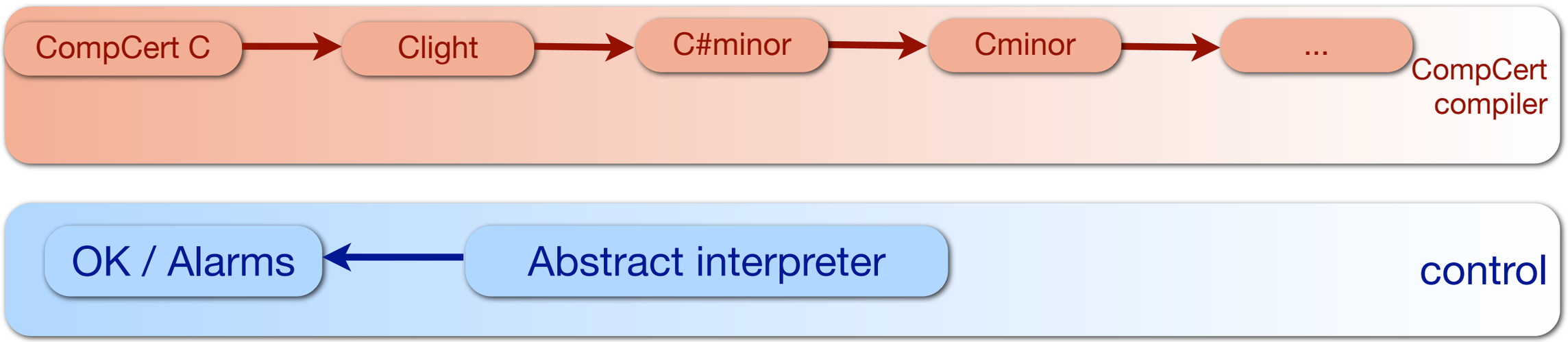
Theorem analyzer_is_correct:
forall p, static_analyzer_result p = Success →
(forall behv, exec_C_program p behv → not_wrong behv).

Stronger correctness result

Theorem transf_c_program_is_refinement:
forall p tp, transf_c_program p = OK tp →
(forall behv, exec_Asm_program tp behv → exec_C_program p behv).

The Verasco static analyzer

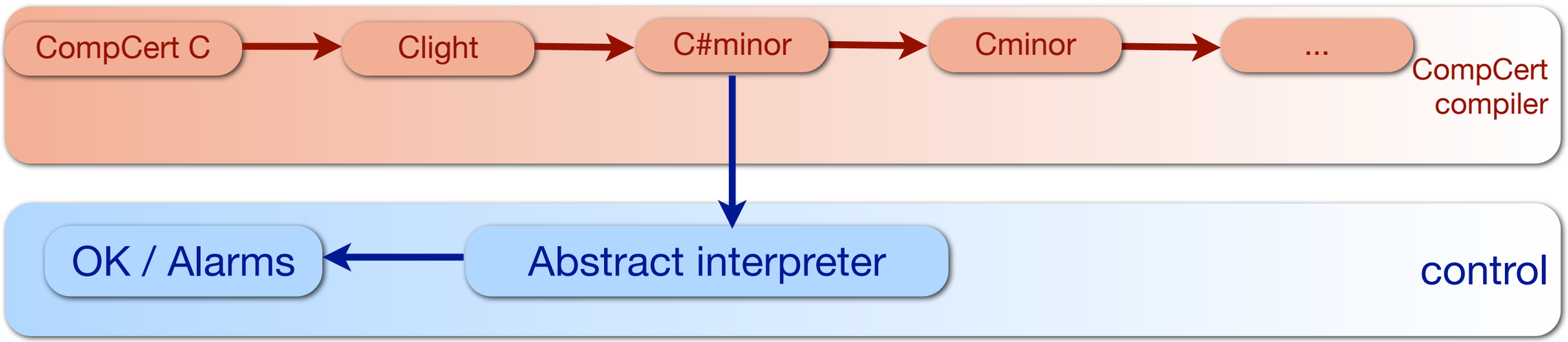
Integration with CompCert



Reuses the CompCert front-end until C#minor (simpler language than C99)

Guarantees provided by the analyzer provably extends to the assembly code

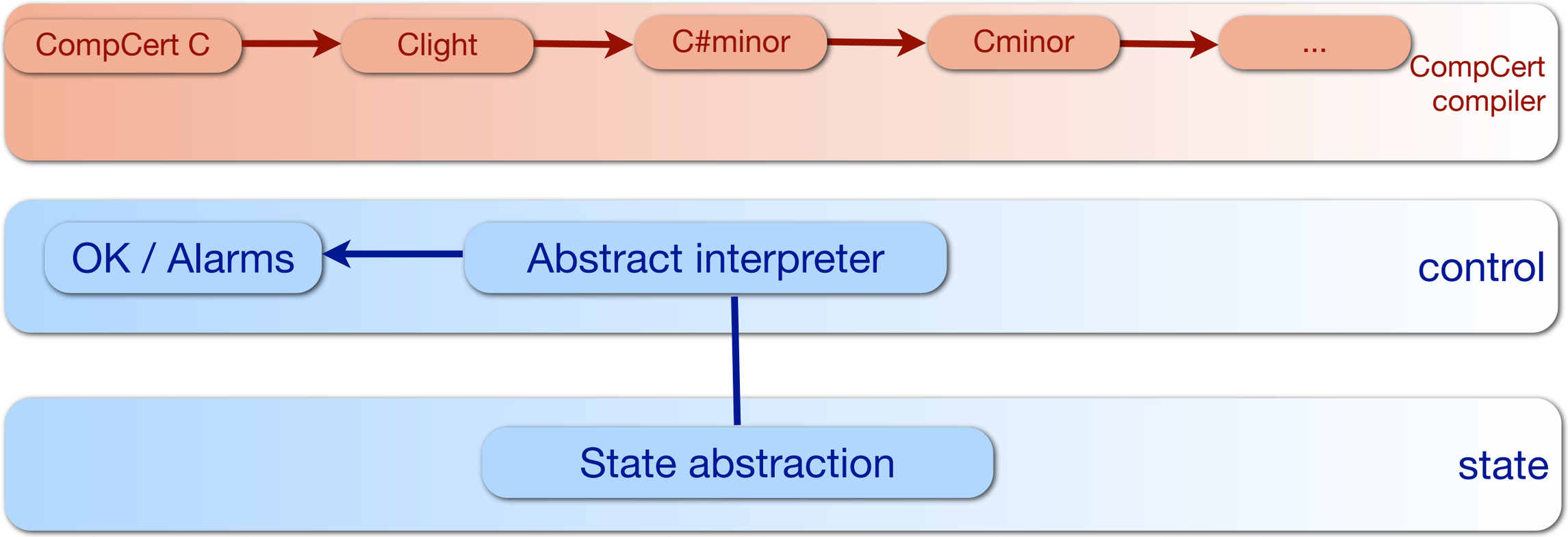
Integration with CompCert



Reuses the CompCert front-end until C#minor (simpler language than C99)

Guarantees provided by the analyzer provably extends to the assembly code

Abstract interpreter



Abstract interpreter parameterized by a state abstract domain

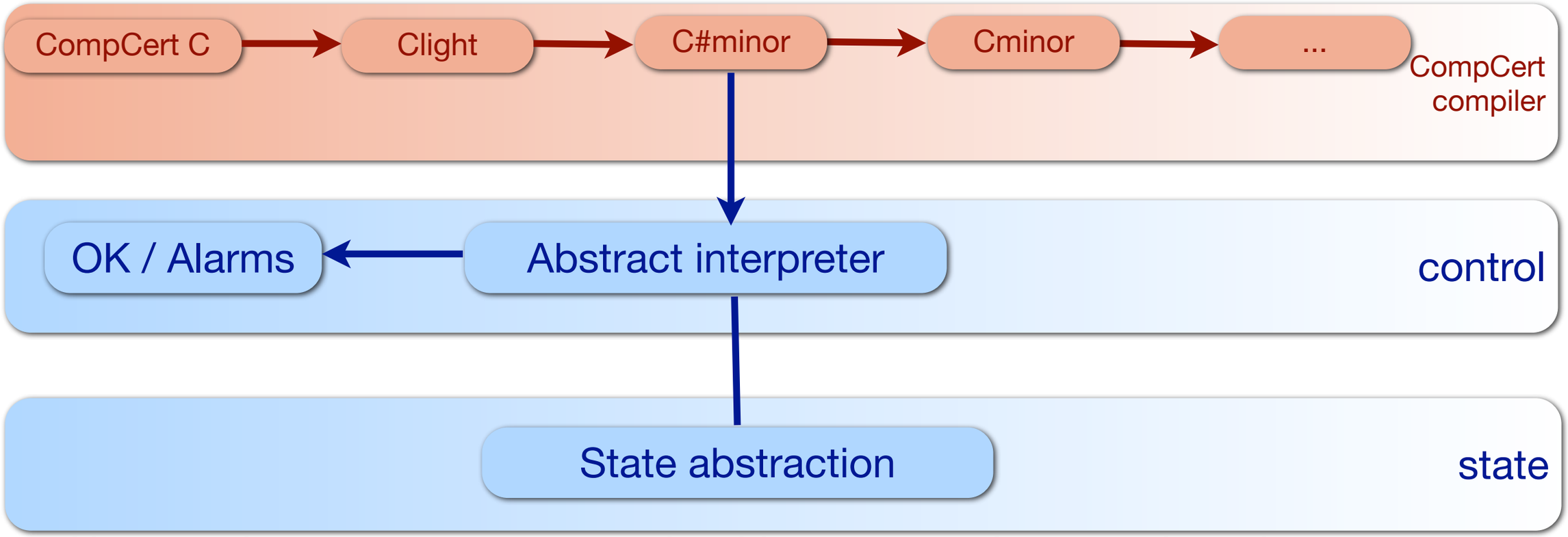
Structural recursion on syntax tree

- unfolding functions at call sites

Fixpoint iteration using widening and narrowing

- one fixpoint iteration per loop
- gotos: one fixpoint iteration per function

Abstract interpreter



Abstract interpreter parameterized by a state abstract domain

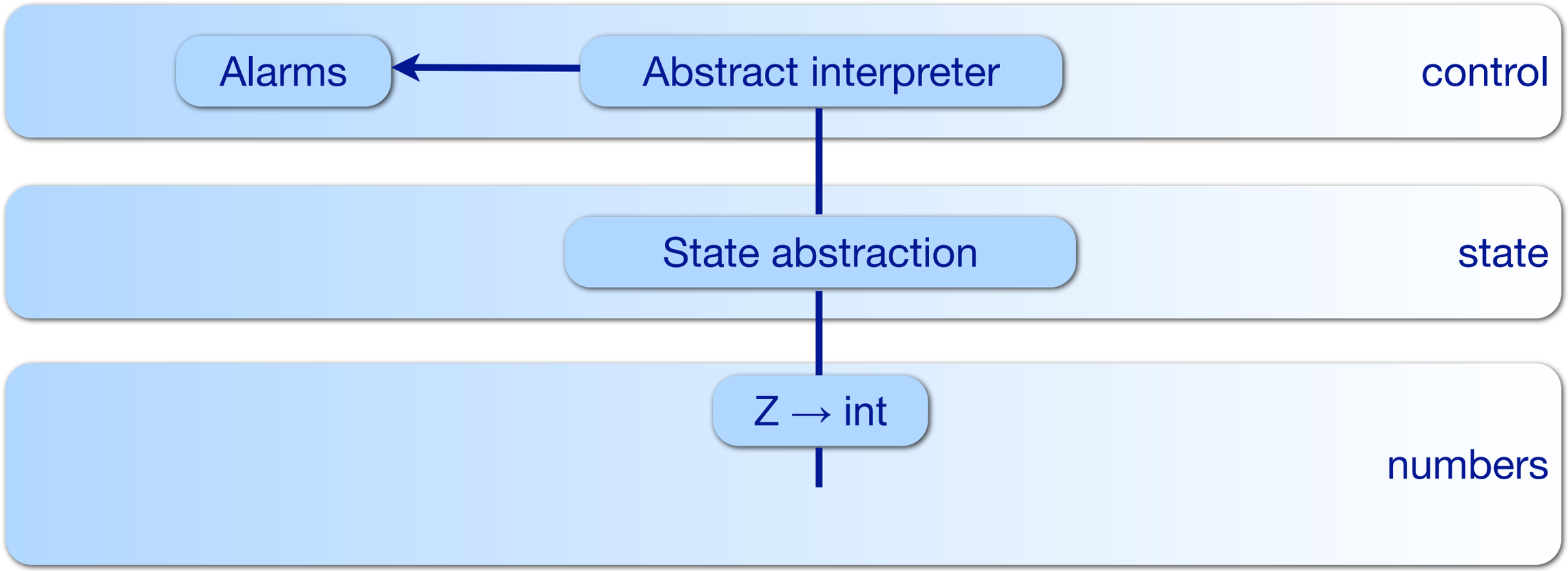
Structural recursion on syntax tree

- unfolding functions at call sites

Fixpoint iteration using widening and narrowing

- one fixpoint iteration per loop
- gotos: one fixpoint iteration per function

State abstract domain

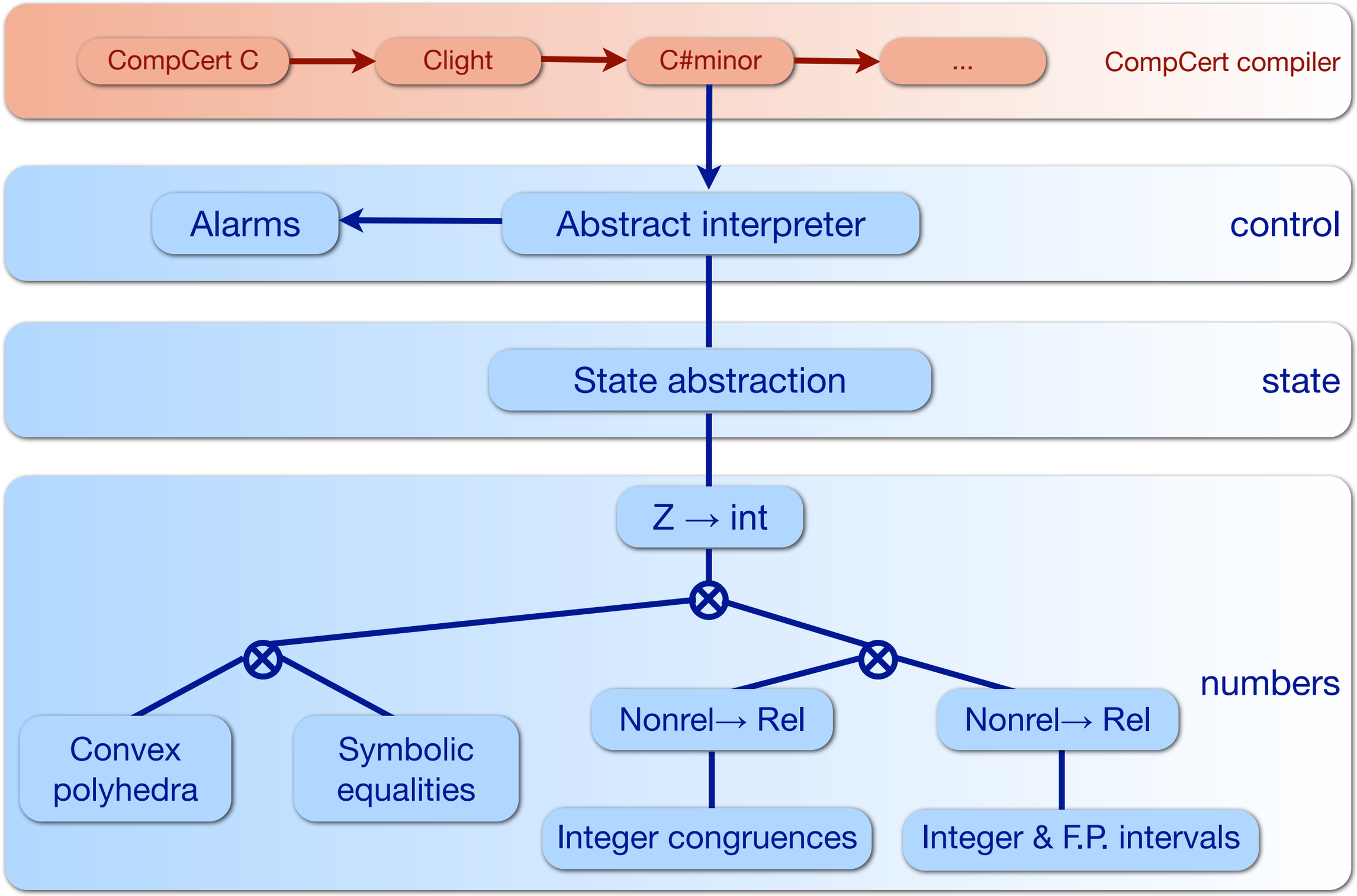


Parameterized by a numerical abstract domain (concretizes to numerical environments)

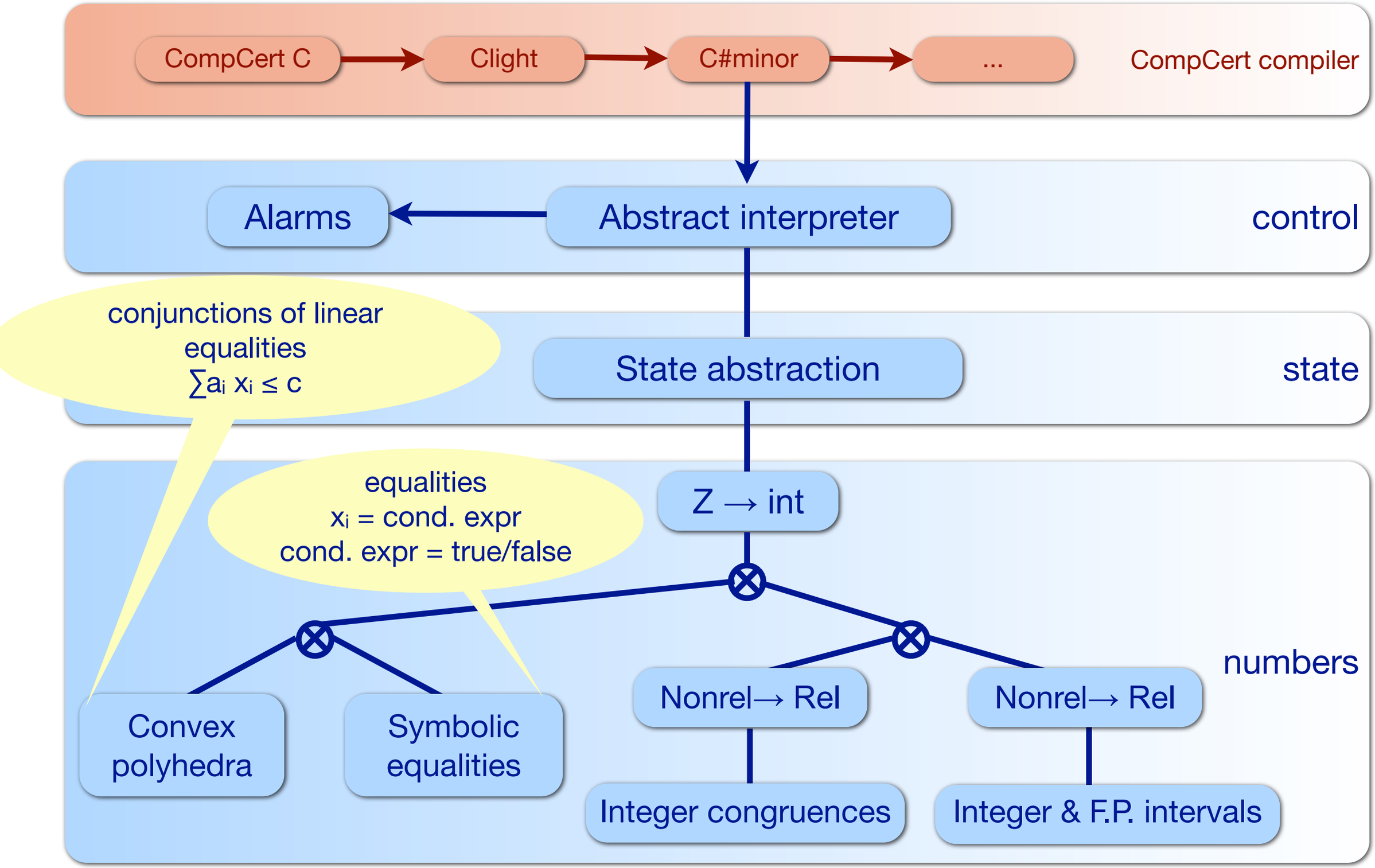
Solves pointer references (points-to domain)

Checks type and memory safety (types domain, permission domain)

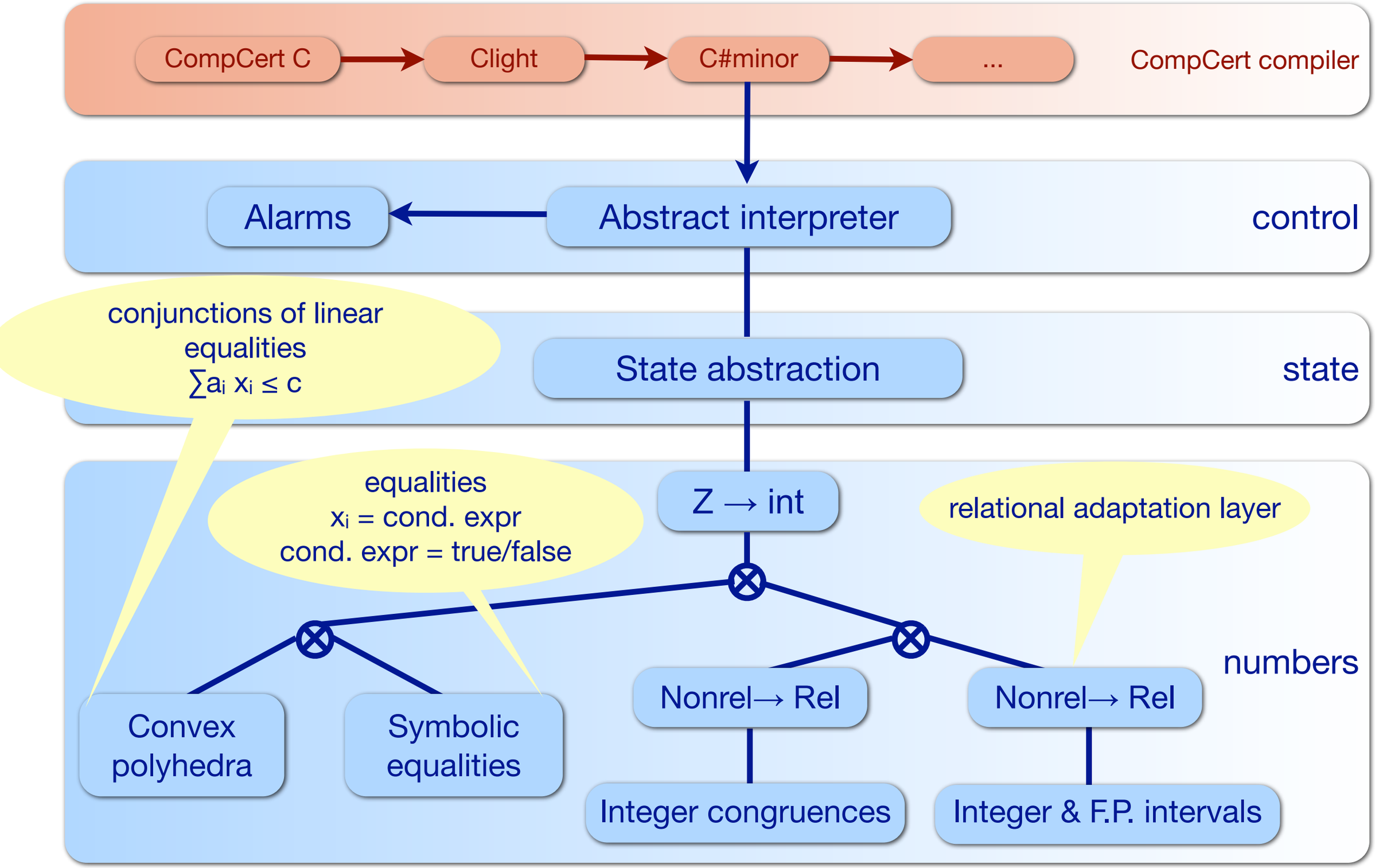
Several numerical domains



Several numerical domains



Several numerical domains

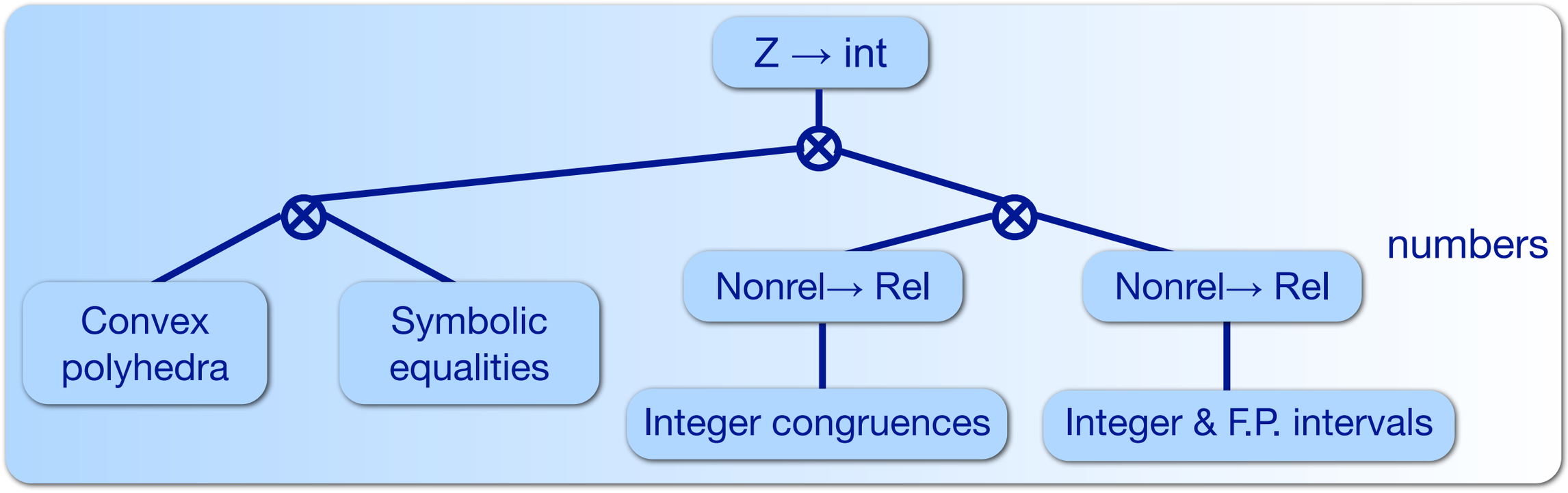


Combination of numerical domains

No full reduced product

More flexible channel-based communication system

- Domains can query each others
- Inspired by Astrée



First example

```
const double sqrt_tab[128] = { ... };

int main() {
  double x = any_double();

  verasco_assume(0.3 < x && x < 0.7);

  double y = sqrt_tab[ (int)(x*128.) ] ;

  verasco_assert(0.54 < y && y < 0.84);
  return 0;
}
```

First example

state and numerical domains cooperation

integer and float intervals

```
const double sqrt_tab[128] = { ... };

int main() {
  double x = any_double();
  verasco_assume(0.3 < x && x < 0.7);
  double y = sqrt_tab[ (int)(x*128.) ] ;
  verasco_assert(0.54 < y && y < 0.84);
  return 0;
}
```

precise bounds on results

Second example

```
int main() {
  int v0 = 0, v1 = 1;
  int* tab[6] = {&v0, &v1, &v0, &v1, &v0, &v1};

  for(int i = -5; i < 6; i++) {
    int in_range_odd = 0 <= 2*i+1 && 2*i+1 < 6;
    int some_other_computation = i*i+32;
    if( in_range_odd )
       verasco_assert( *(tab[ 2*i+1 ]) == 1);
  }
  return 0;
}
```

Second example

```
int main() {
  int v0 = 0, v1 = 1;

  int* tab[6] = {&v0, &v1, &v0, &v1, &v0, &v1};


  for(int i = -5; i < 6; i++) {
    int in_range_odd = 0 <= 2*i+1 && 2*i+1 < 6;

    int some_other_computation = i*i+32;
    if( in_range_odd )
       verasco_assert( *(tab[ 2*i+1 ]) == 1);
  }

  return 0;
}
```

symbolic propagation of conditions

complex memory access

use of congruence information

Conclusion

Proving correct a realistic static analyzer based on abstract interpretation seems feasible

- realistic, complicated language (C99)
- complex combination of abstract domains

Plenty of possible extensions

- Numerical abstract domains
  (e.g. octagons - linear inequalities ± x ± y ≤ c, linear filters)
- Better handling of control-flow (e.g. recursion)
- Memory abstract domains
  (e.g. C union, dynamic allocation, shape analysis)

http://compcert.inria.fr/verasco/

Questions ?

Handling bounded machine integers

Parameterized by an abstract domain for Z

Wraparound when overflow

Checks numerical errors: division by zero, undefined overflows