Formal verification of a static analyzer based on abstract interpretation

Sandrine Blazy

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





IFIP WG 1.9/2.15, 2014-07-14

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.

```
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).
```

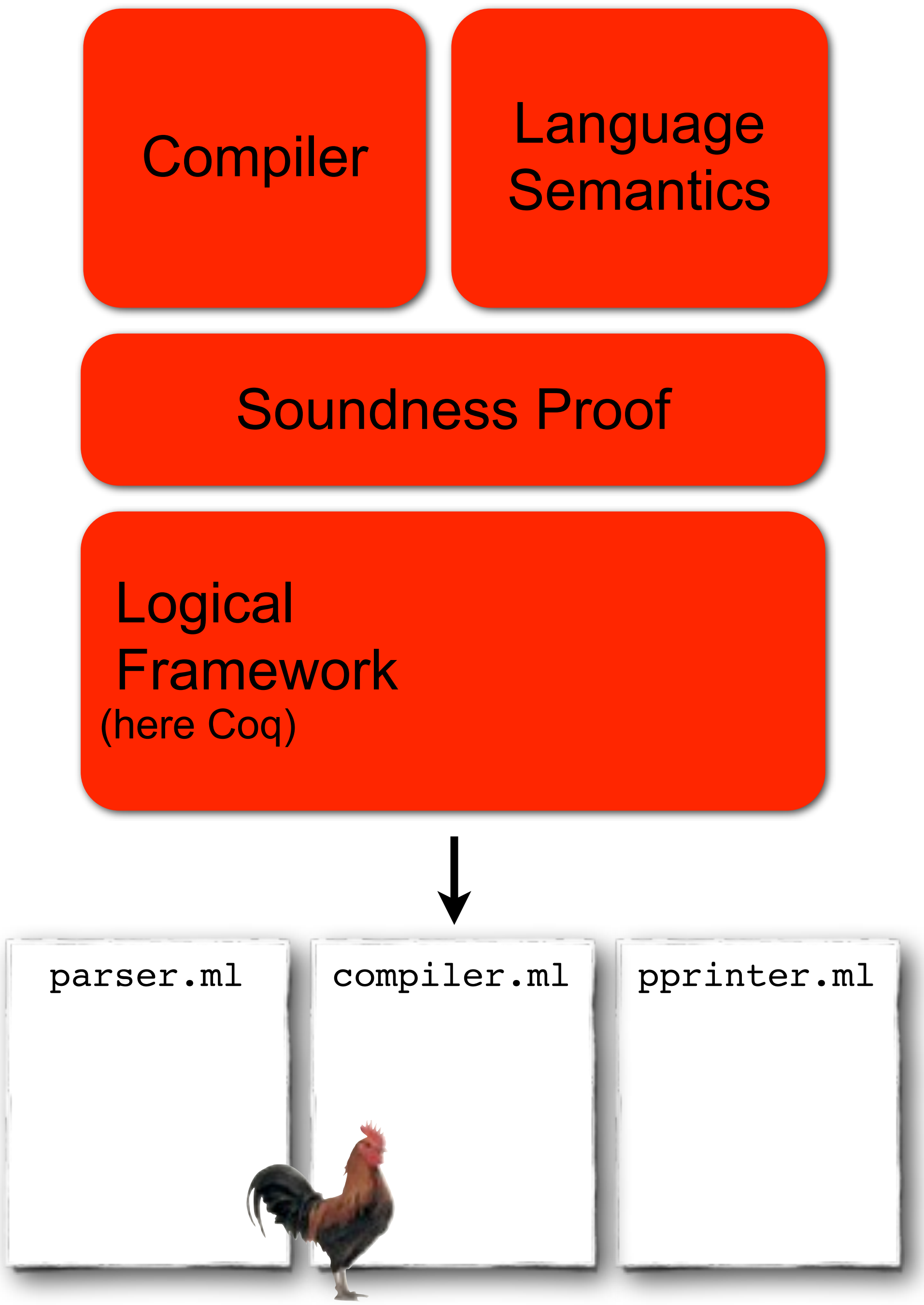
The generated assembly code can not wrong.

Proof methodology

- The compiler is written inside the purely functional Coq programming language.
- We state its soundness 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

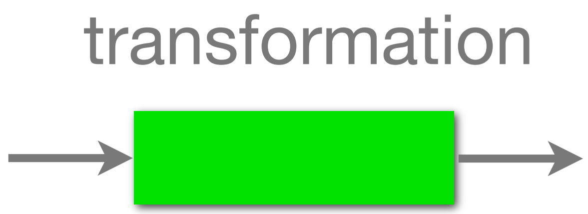
- The compiler is written inside the purely functional Coq programming language.
- We state its soundness w.r.t. a formal specification of the language semantics.
- We interactively and mechanically prove this.
- We decompose the proof in proofs for each compiler pass.
- We extract a Caml implementation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



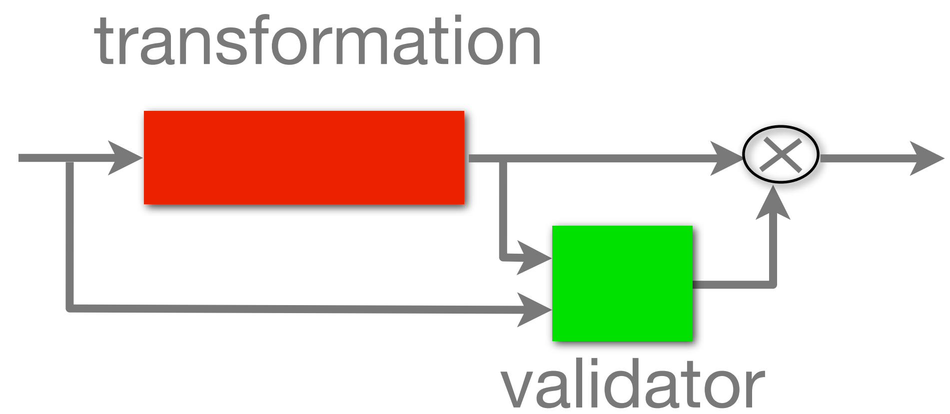
CompCert components

Verification patterns
(for each compilation pass)

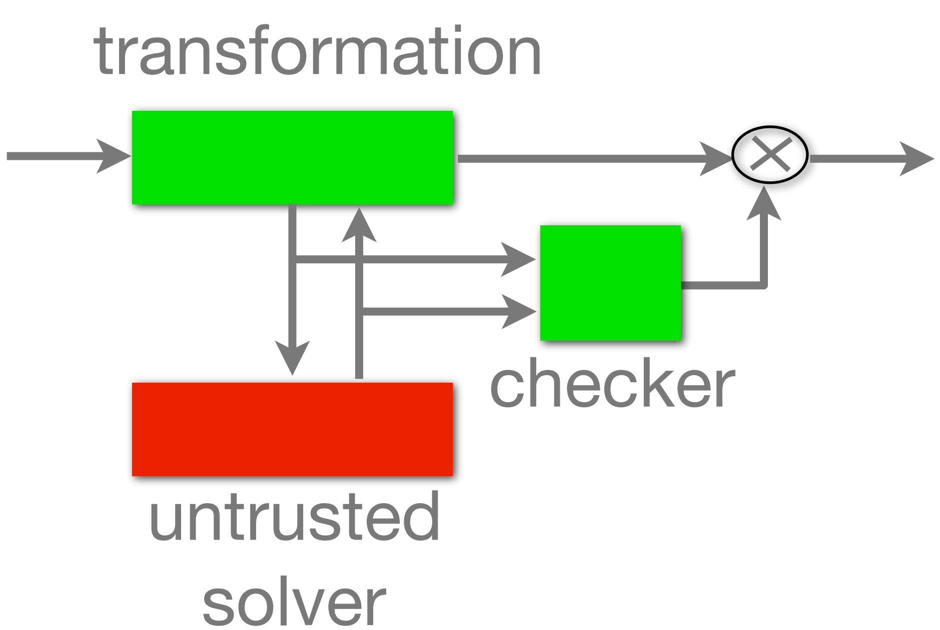
Verified transformation

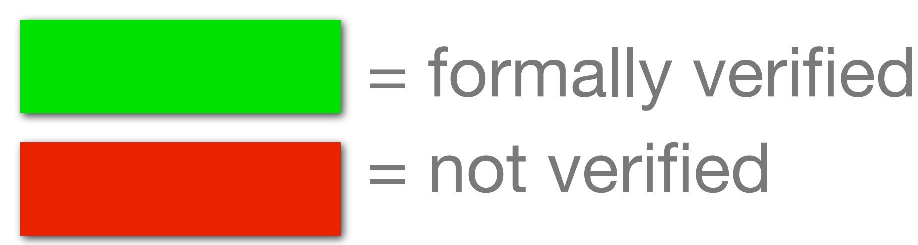


Verified translation validation



External solver with verified transformation

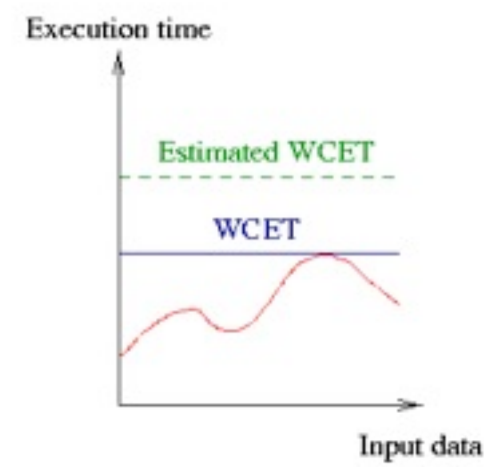




Compiling critical embedded software

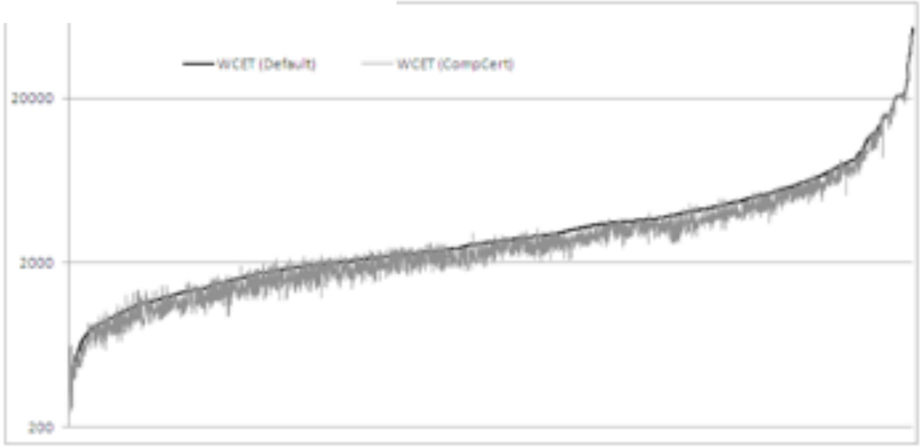
Fly-by-wire software, Airbus A380 and A400M

- FCGU (3600 files, 3.96 MB of assembly code): mostly control-command code generated from block diagrams (Scade)
- minimalistic OS



Results

- Estimated WCET for each file
- Average improvement per file: 14%
- Compiled with CompCert 2.3, May 2014

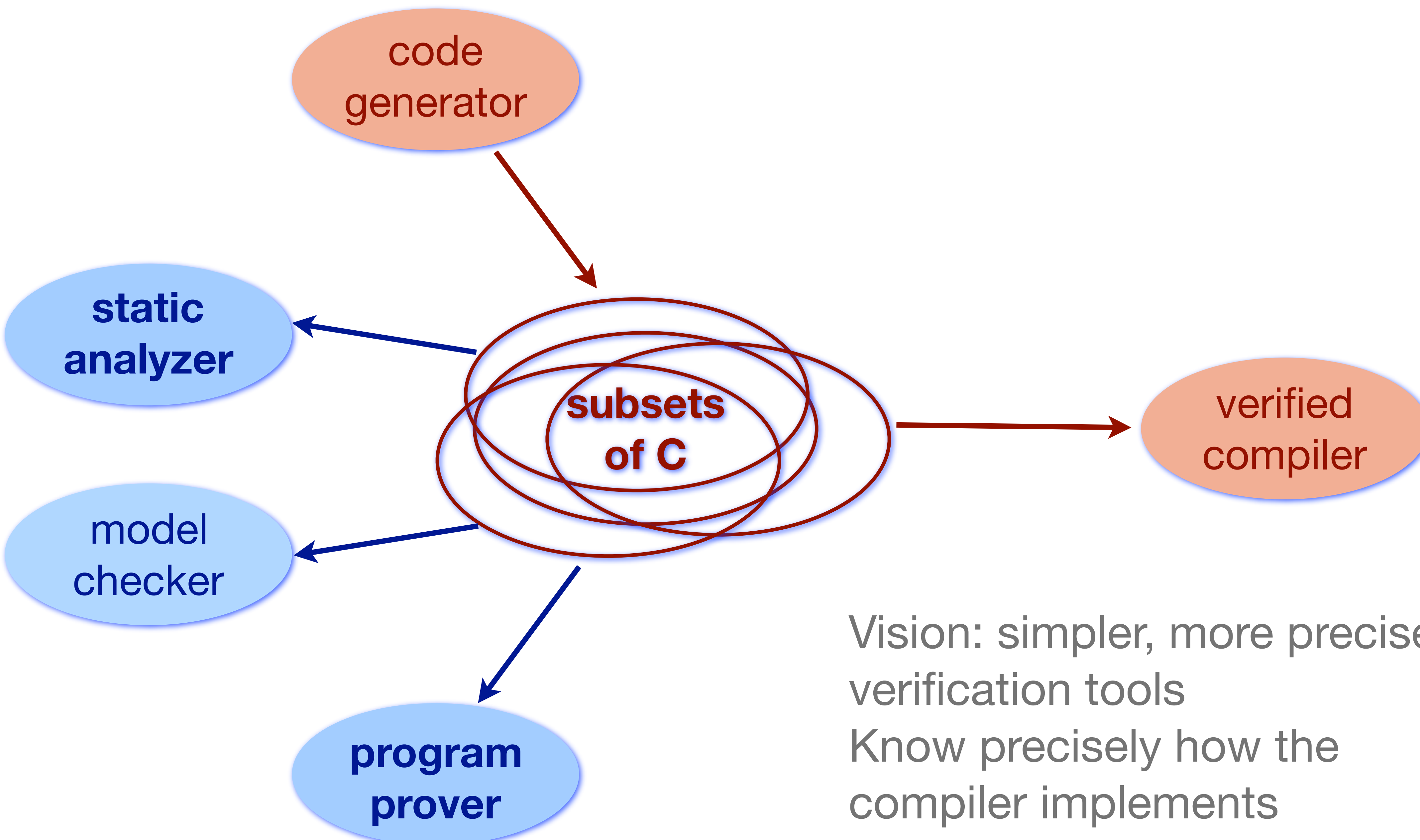


Conformance to the certification process (DO-178)

- Trade-off between traceability guarantees and efficiency of the generated code

Tools that participate in the production and verification of critical embedded software

Are these verification tools semantically sound ?



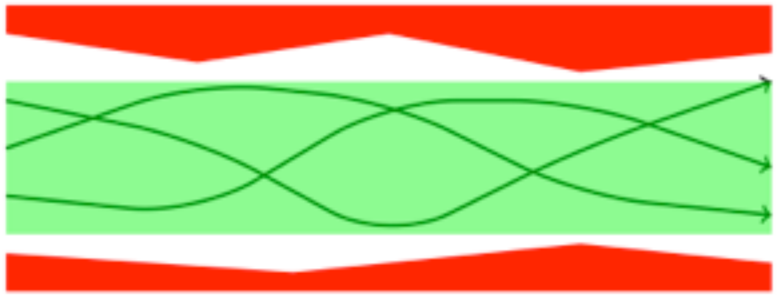
Vision: simpler, more precise verification tools
Know precisely how the compiler implements unspecified behaviors of C

This talk

- From CompCert to formally verified static analysis
- A first formally verified static analyzer
  - Architecture
  - Applications
- Quantitative jump: an improved formally verified static analyzer

From CompCert to formally verified static analysis

Static analysis



Absence of run-time errors in programs

A reference tool: the Astrée static analyzer (P.Cousot et al.)

- Based on a rock salt theory: abstract interpretation
- Programmed in Caml and highly modular
- Takes care of numerical pitfalls
  - machine integers and floating point numbers
  - both in the C semantics and in the analyzer's own computations
- Memory safety of the A380 fly-by-wire software (~5 hours of computation)

Implementations on real languages are still error-prone.

- Abstract interpretation proofs are (mainly) done on paper and without direct linkk to the actual implementation

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
- Nontrivial abstract domains, including relational domains
- Modular architecture inspired from Astrée's
- Decent alarm reporting

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

VERASCO

http://verasco.imag.fr

Building a static analyzer in ML

Modular design

```
module IntervalAbVal : ABVAL = ...

module NonRelAbEnv (AV:ABVAL) : ABENV = ...

module SimpleAbMem (AE:ABENV) : ABMEMORY = ...

module Iterator (AM:ABMEMORY) : ANALYZER = ...

module myAnalyzer = Iterator(SimpleAbMem(NonRelAbEnv(IntervalAbVal)))
```

Example of interface

```
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
```

Building a static analyzer

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_monotone : ∀ 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)
}
```

Lazy proofs

Proof by necessity

- We don't prove properties that are not strictly necessary to establish a soundness theorem.

What we don't prove

- `(ab,le,join)` enjoy a lattice structure
- gamma is a meet morphism between complete lattices (Galois connection)
- widen is a sound widening operator

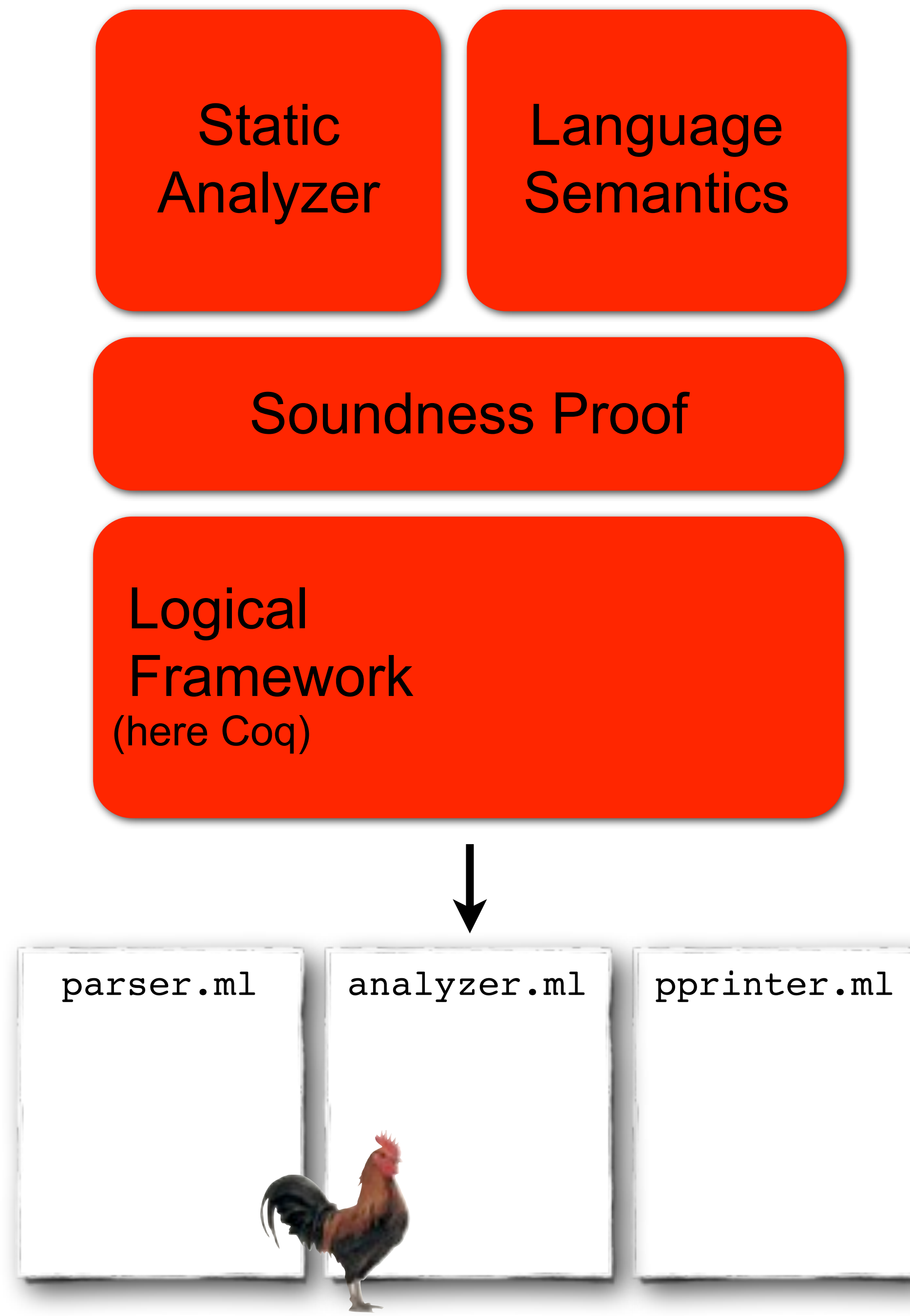
```
Class adom (ab:Type) (c:Type) := {
  le : ab → ab → bool;
  top : ab;
  join : ab → ab → ab;
  widen : ab → ab → ab;

  gamma : ab → ℘(c);

  gamma_monotone : ∀ 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

```
Definition analyzer (p: program) := ...
Theorem analyzer_is_sound :
   ∀ p, analyzer p = Success ->
        sound(p).
Proof. ...(* few months later *)...Qed.
Extraction analyzer.
```

Verifying a static analyzer

```
Definition analyzer (p: program) := ...
Theorem analyzer_is_sound :
   ∀ p, analyzer p = Success ->
        sound(p).
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

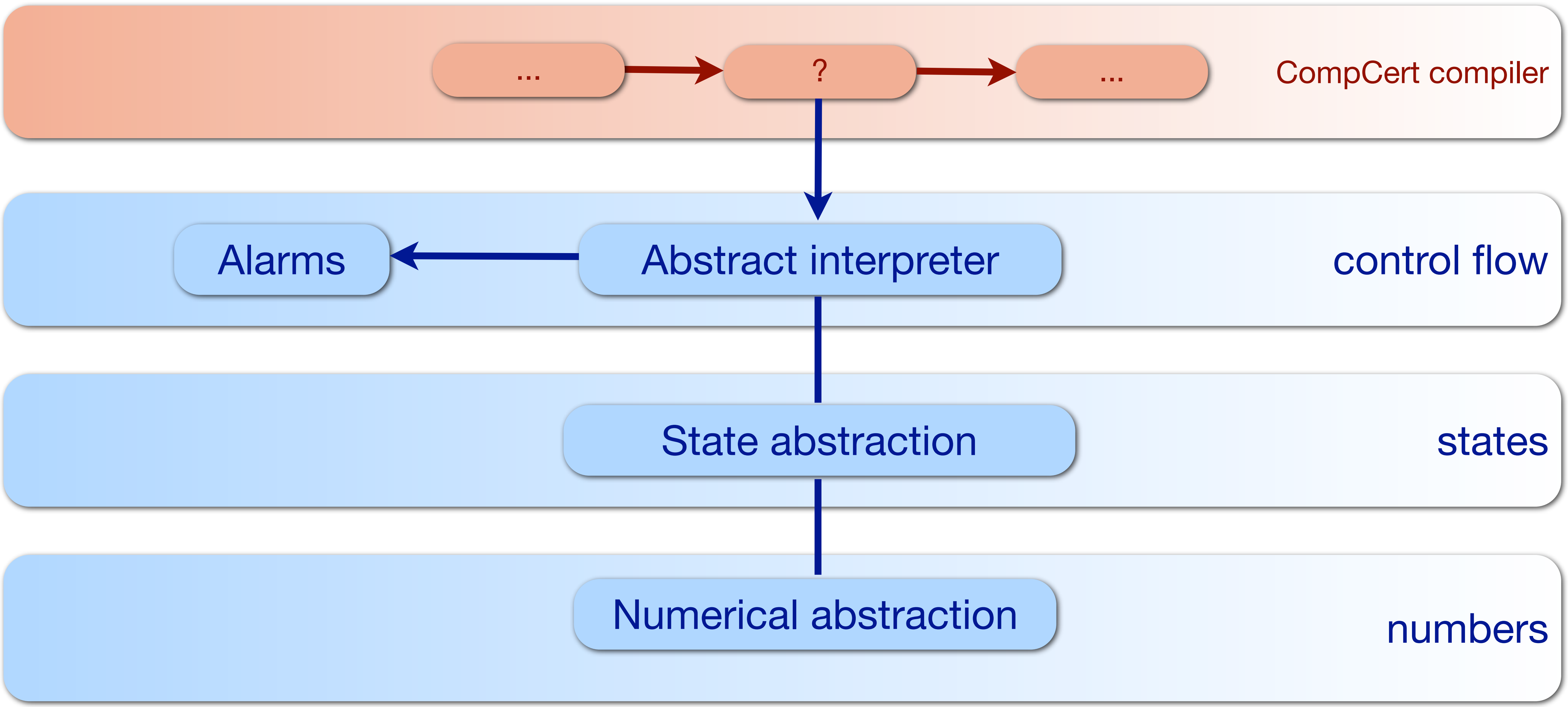
Theorem analyzer_is_correct:
forall p, static_analyzer_result p = Success →
(forall behv, exec_C_program p behv → not_wrong behv).

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).

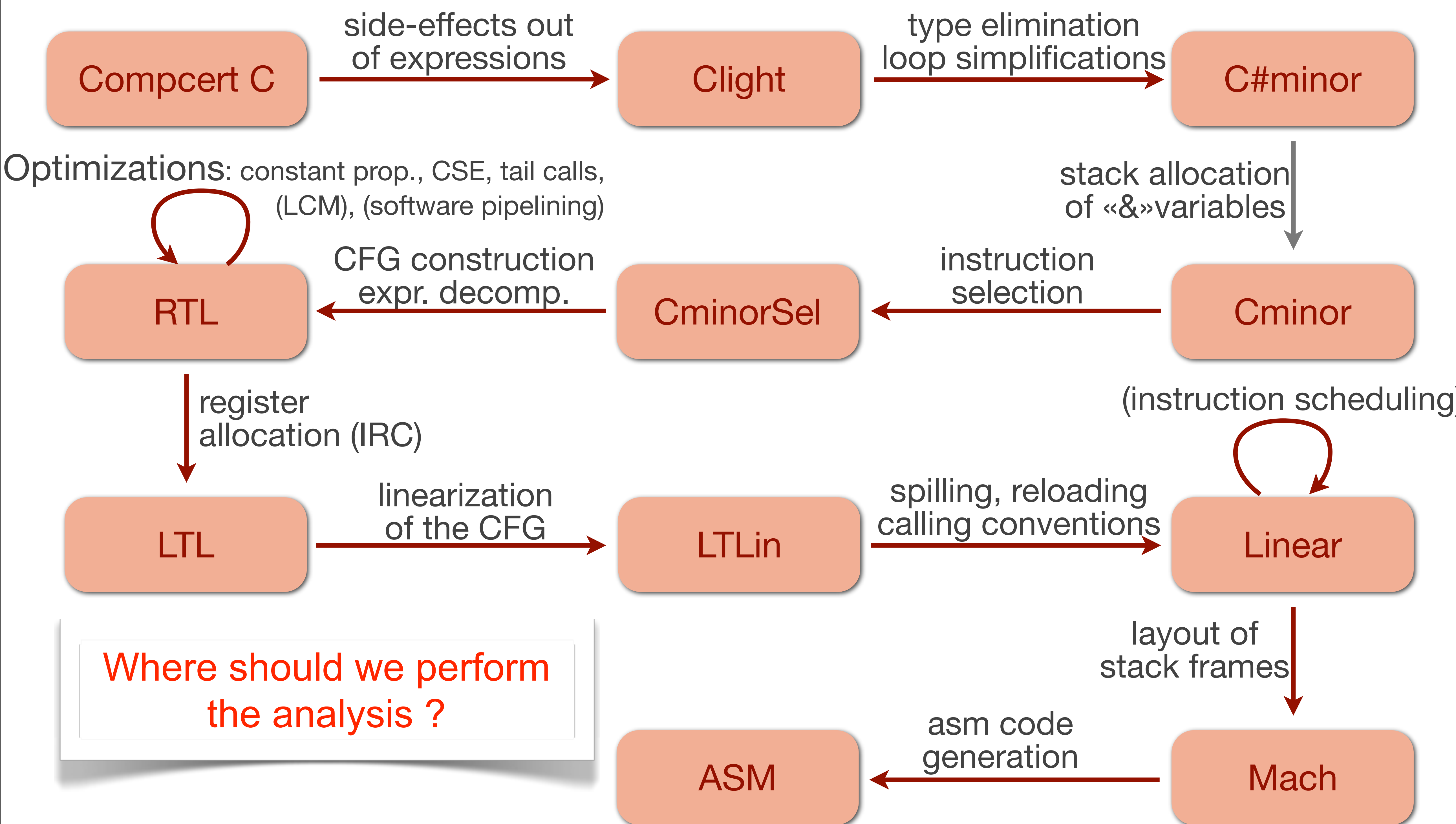
Verasco 1.0

General architecture



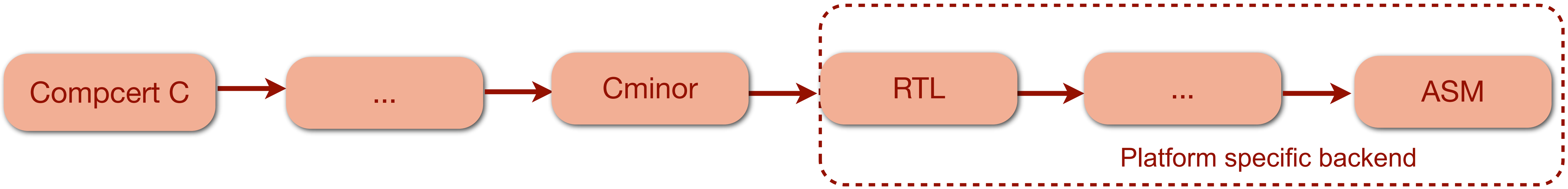
Each layer is parameterized by the underlying one.

CompCert: 1 compiler, 11 languages



Where should we perform
the analysis ?

Which CompCert representation ?



C source ?

- the place where we want to prove program safety
- but the most difficult place to start (not an IR but a source language)

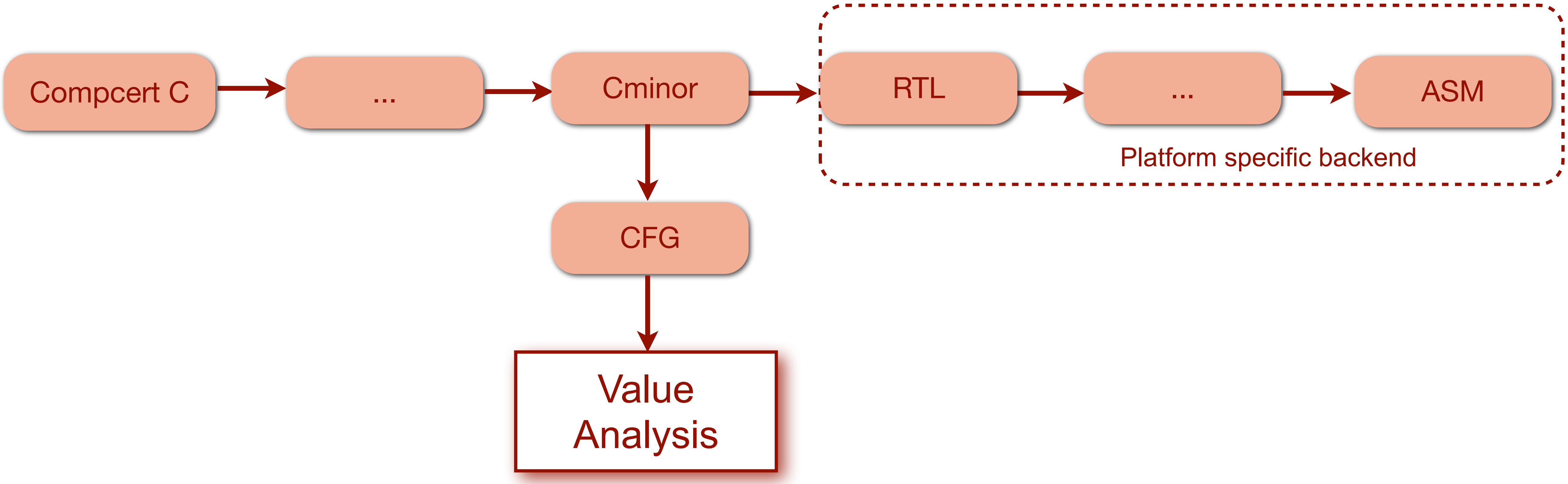
RTL ?

- the place where most CompCert optimizations take place
- but platform specific, flat expressions

Cminor ?

- the last step before platform specific semantics
- designed to welcome forthcoming extensions
- but control flow still less uniform than in RTL (nested blocks and exits)

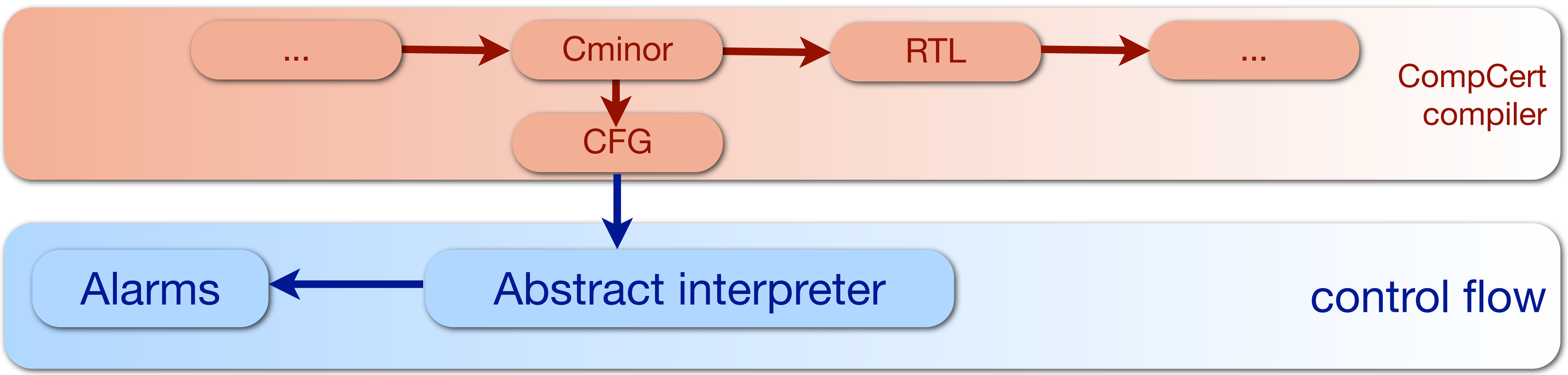
Which CompCert representation ?



A new representation: CFG

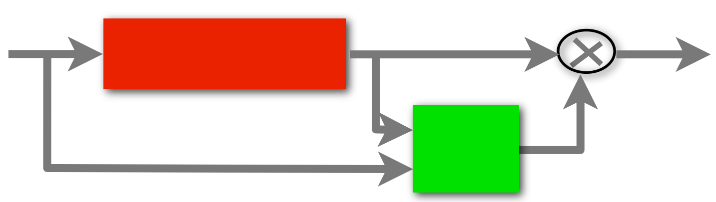
- Cminor expressions (i.e. side effect free C expressions)
- control flow graphs with explicit program points
- control flow is restricted to simple unconditional and conditional jumps
- platform independent

Modular design
The abstract interpreter



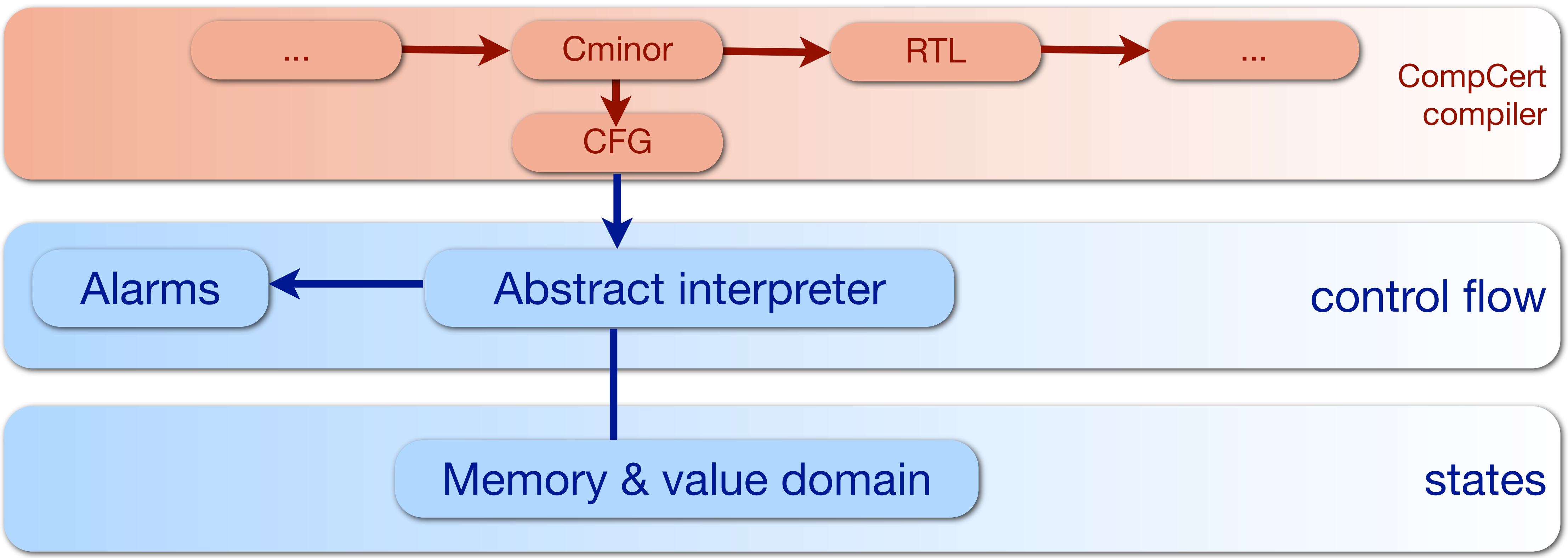
CFG program are unstructured.

- We need to build widening strategies on unstructured control flow graph !
- We let an external tool computes a post-fixpoint and check the result in Coq.



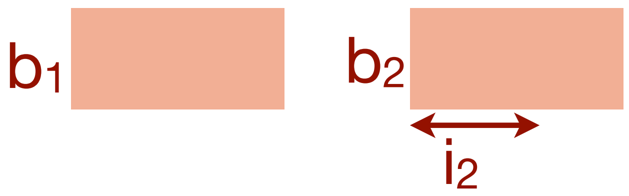
- The external tool is complex (Bourdoncle strategy + widening heuristics),
- but we don't prove anything about it as well as about all widening operators.

Modular design
The state abstract domain

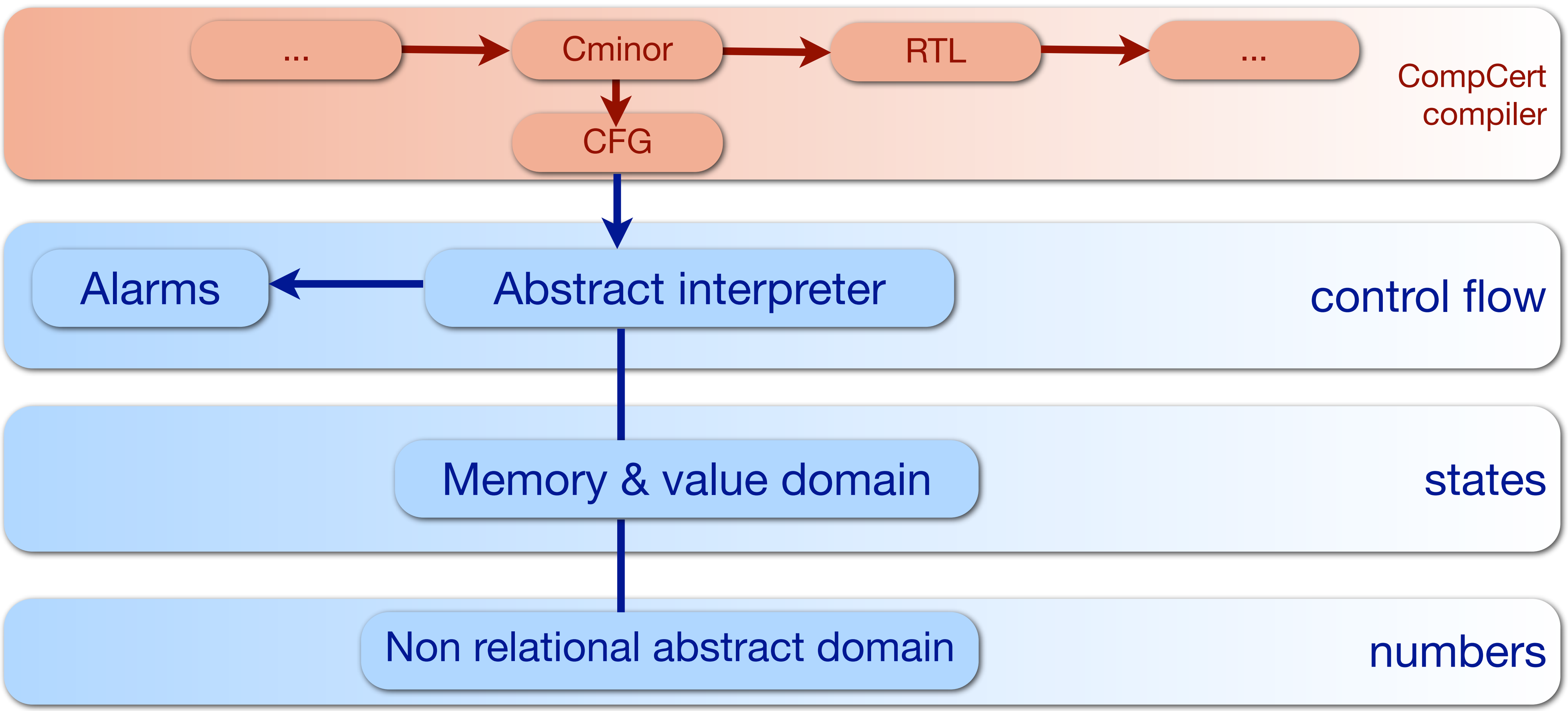


From a simple imperative semantic domain to the C memory

- The current functor tracks only the content of local variables.
- A pointer `Vptr(b,i)` is abstracted by its offset `i`.
- 1 concrete block = 1 abstract block



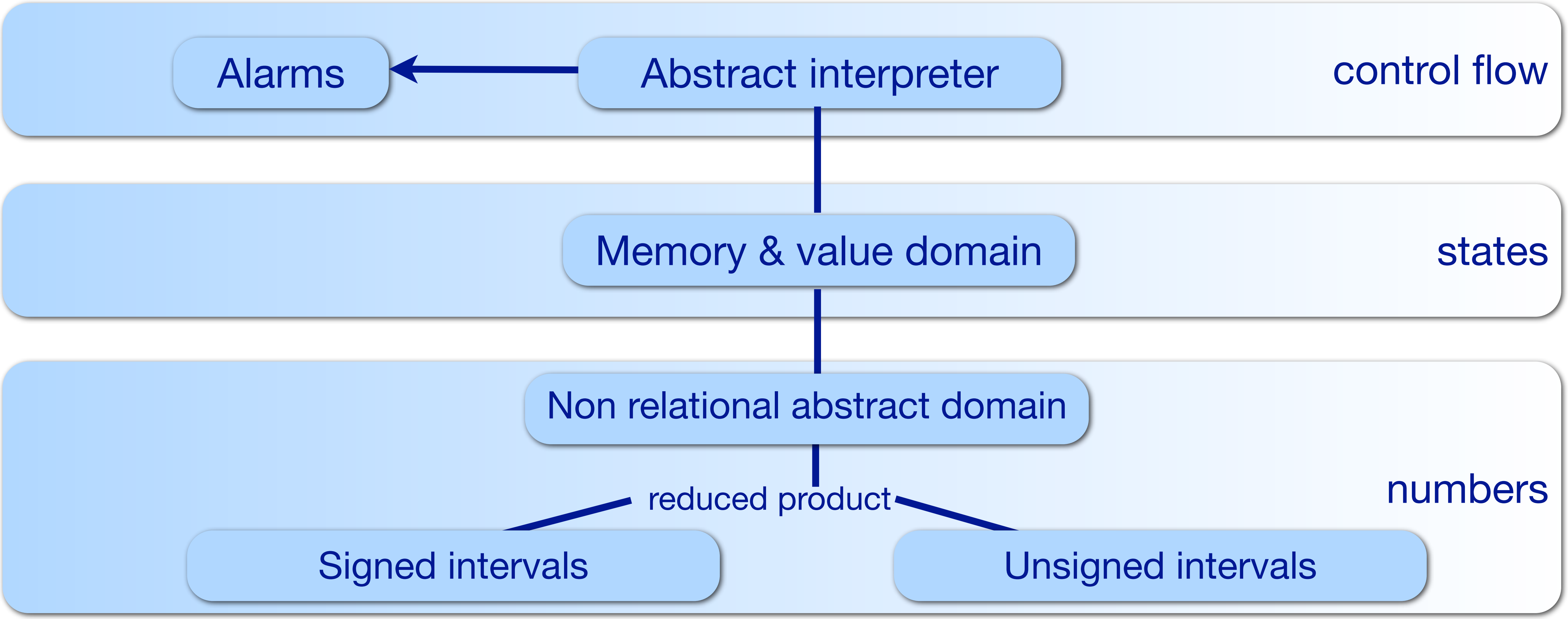
Modular design
The non relational abstract domain



From a set of values to one single value

- The set of abstract values is implemented with efficient binary tree representations.
- We use downward iterations of branch conditions.
- As much as we can, we reduce empty properties to a single abstract element.

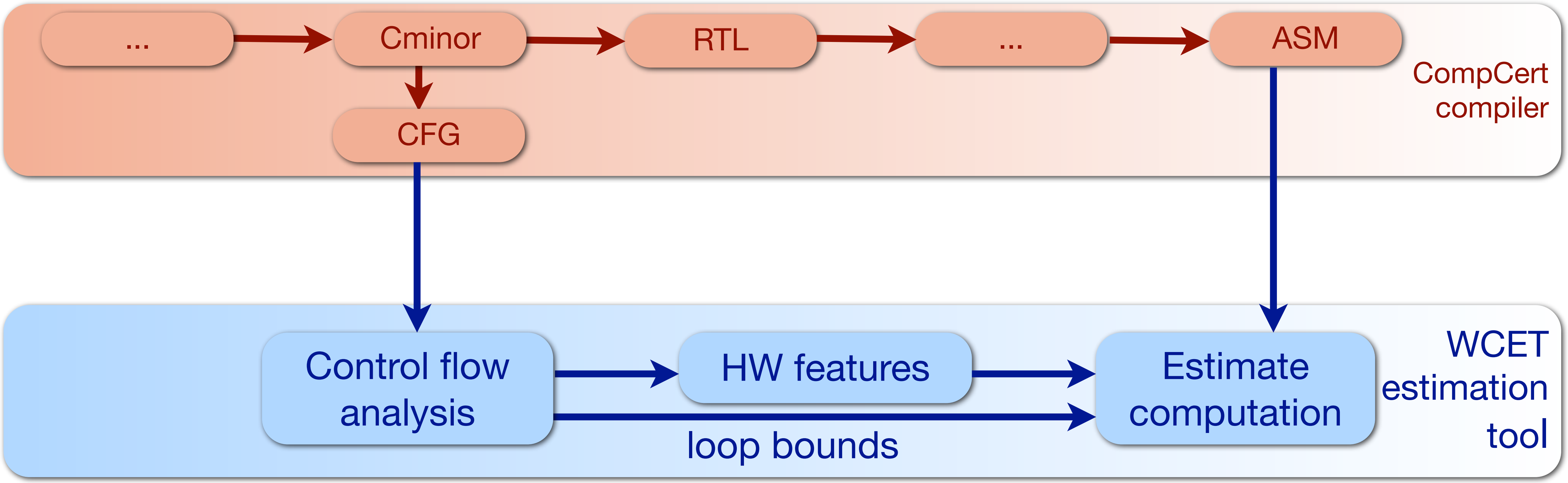
Modular design
The interval numeric abstraction



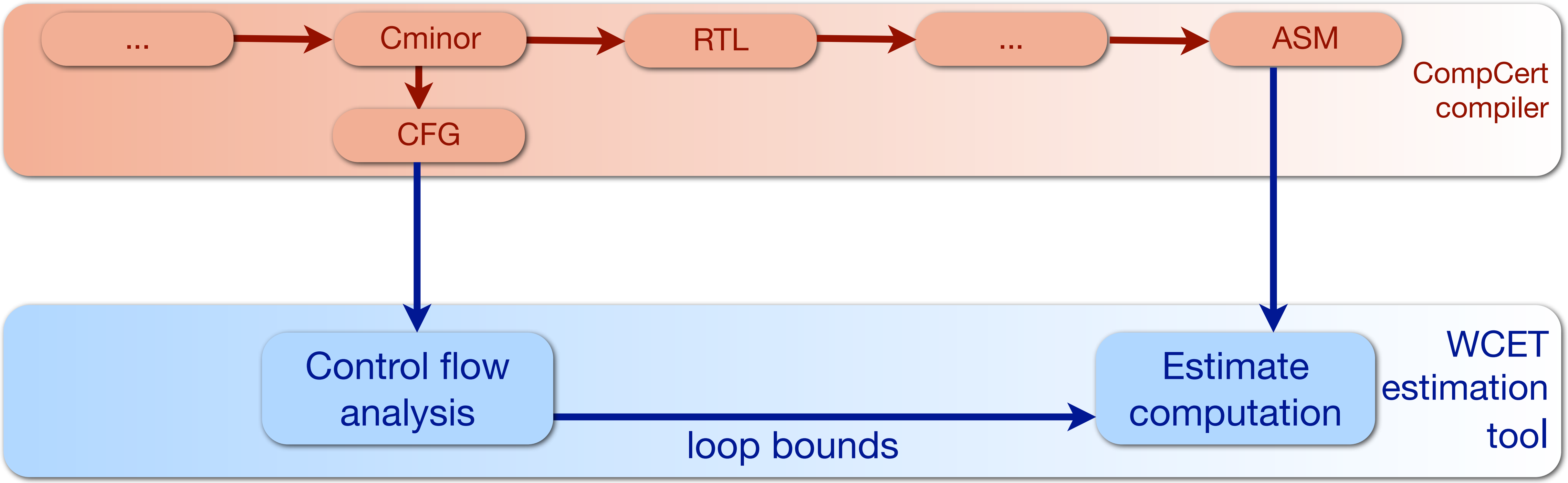
Compiler internal representation of integers

- At this level of CompCert, there are no signed or unsigned integers: only machine integers.
- An interval abstraction can represent a range for the signed interpretation of integers, or the unsigned interpretation.
- A reduced product combines both abstractions for enhanced precision.

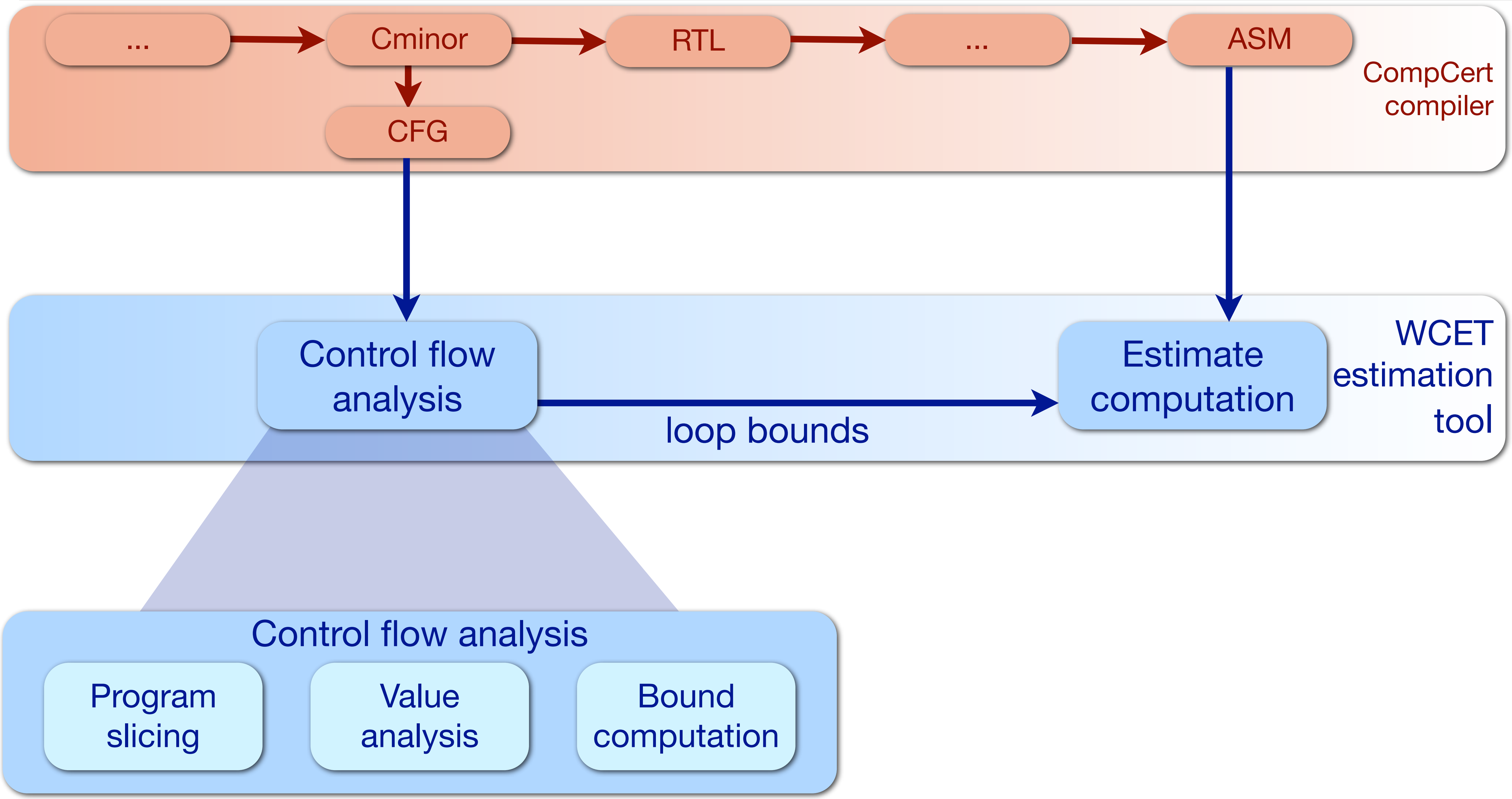
Application:
a formally verified WCET estimation tool [WCET2014]



Application:
a formally verified WCET estimation tool [WCET2014]



Application:
a formally verified WCET estimation tool [WCET2014]



A second application
Disassembling low-level self-modifying code [ITP2014]

Small assembly language inspired from x86, with indirect jumps

07000607
03000000
00000005
00000000
00000100
09000000
00000004
09000002
00000002
05000002
04000000
00000001

07000607
03000000
00000004
00000000
00000100
09000000
00000004
09000002
00000002
05000002
04000000
00000001

```
 0: cmp R6, R7
 1: gotoLE 5
 2:
 3: halt R0
 4: halt R1
 5: cst 4 → R0
 6:
 7: cst 2 → R2
 8:
 9: store R0 → *R2
10: goto 1
11:
```

Our approach

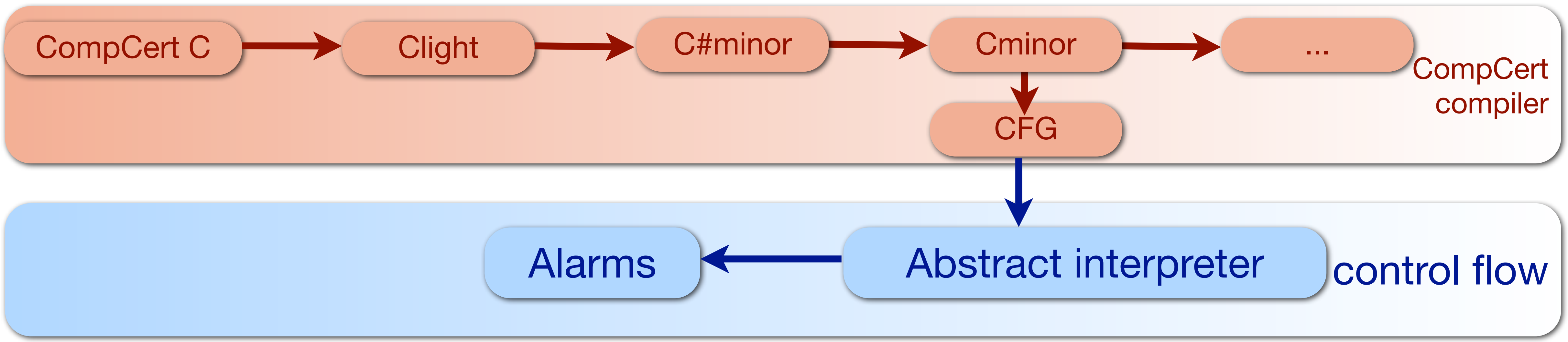
- Value analysis
  - Attach to each reachable program point an over-approximation of the state at that point
  - Analyze the content of memory and of the registers (*e.g.* check every memory write to decide if it modifies the code)
- No previous disassembling of CFG reconstruction

- New abstract domain: integer congruences (strided intervals)
  - combines interval and congruence information
  - Ex.: {1000; 2000}.4 represents {1000; 10004; 1008; . . . ; 2000}

Verasco 2.0

- modular architecture
- much more complex language
- much more sophisticated static analysis techniques

From CFG to C#minor

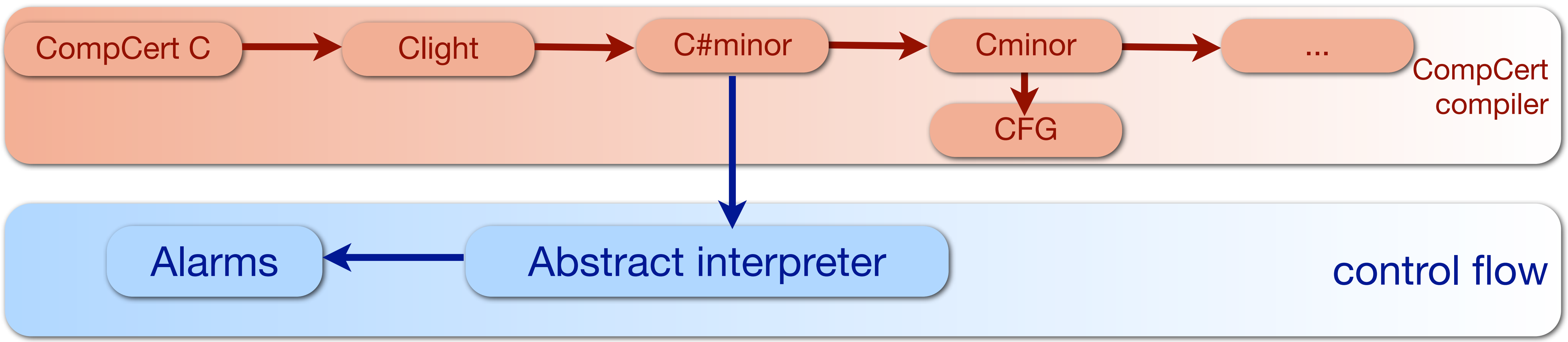


C-like language, but

- no side effects in expressions
- no overloading in C operators
- no implicit casts

C#minor is a mostly structured language (only gotos are unstructured)

From CFG to C#minor

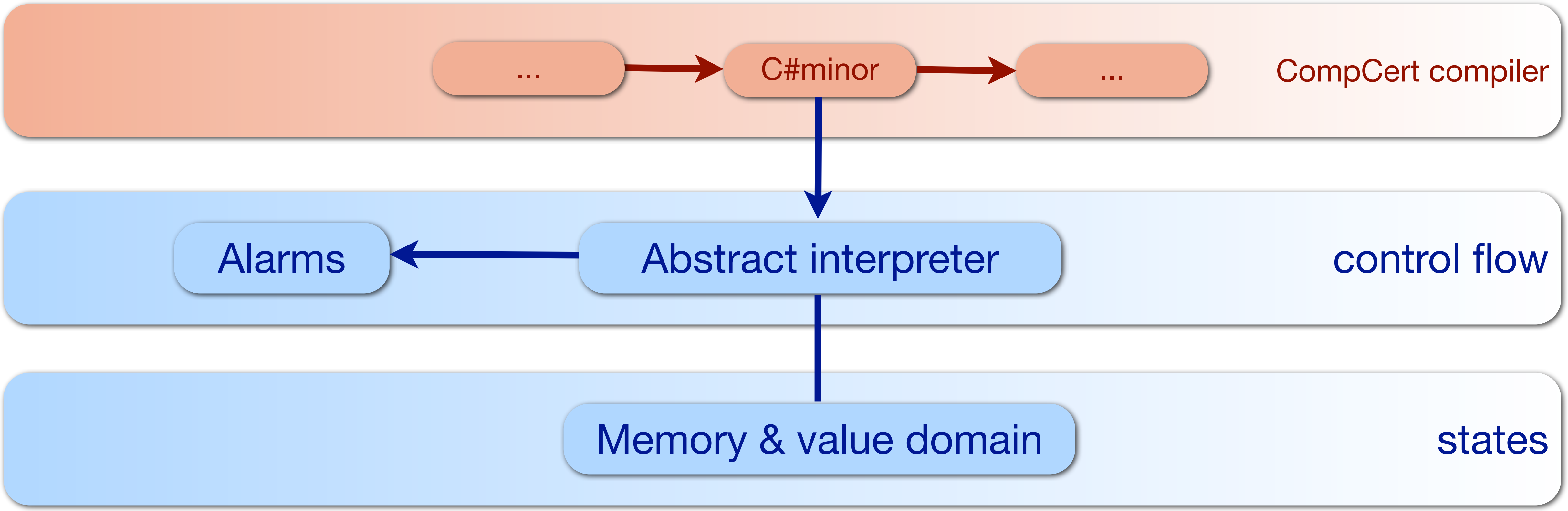


C-like language, but

- no side effects in expressions
- no overloading in C operators
- no implicit casts

C#minor is a mostly structured language (only gotos are unstructured)

Abstract interpreter



Structural approach instead of CFG approach

- obviates the need to define program points
- uses less memory than the CFG-based interpreter
- transfer functions are more involved (control can leave a stmt in many ways)

Parameterized by a relational abstract domain for execution states (environment + memory state + call stack)

Abstract interpreter (cont'd)

Loops

- post-fixpoint (pfp) computation written in Coq
- using a widening operator provided by the abstract domain
- Once a pfp is found, use of narrowing in the hope of finding a smaller pfp

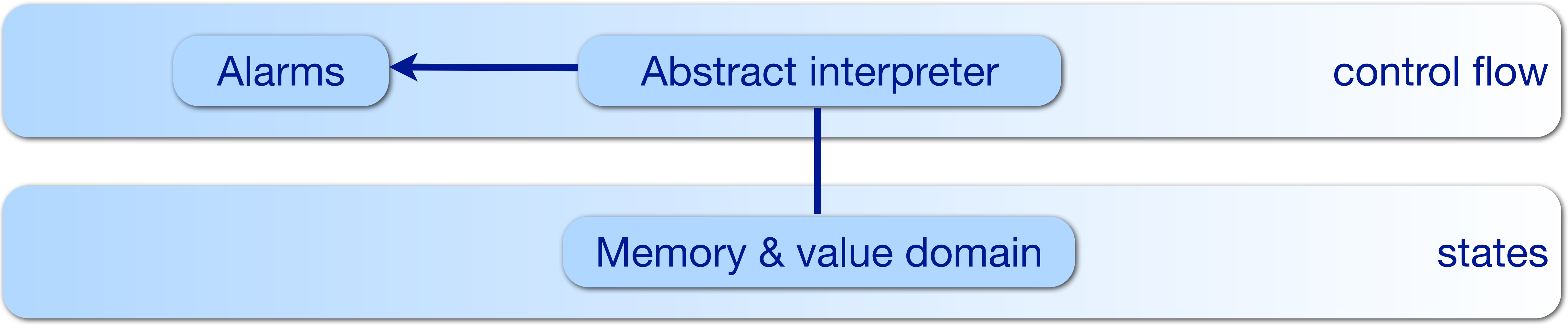
Local fixpoints for each loop + per-function fixpoint for gotos + per-program fixpoint for function calls (interprocedural analysis)

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

- logging monad collecting alarms in the log while the analysis continues

Coq soundness proof relying on axiomatics semantics and step indexing semantics

The state abstract domain



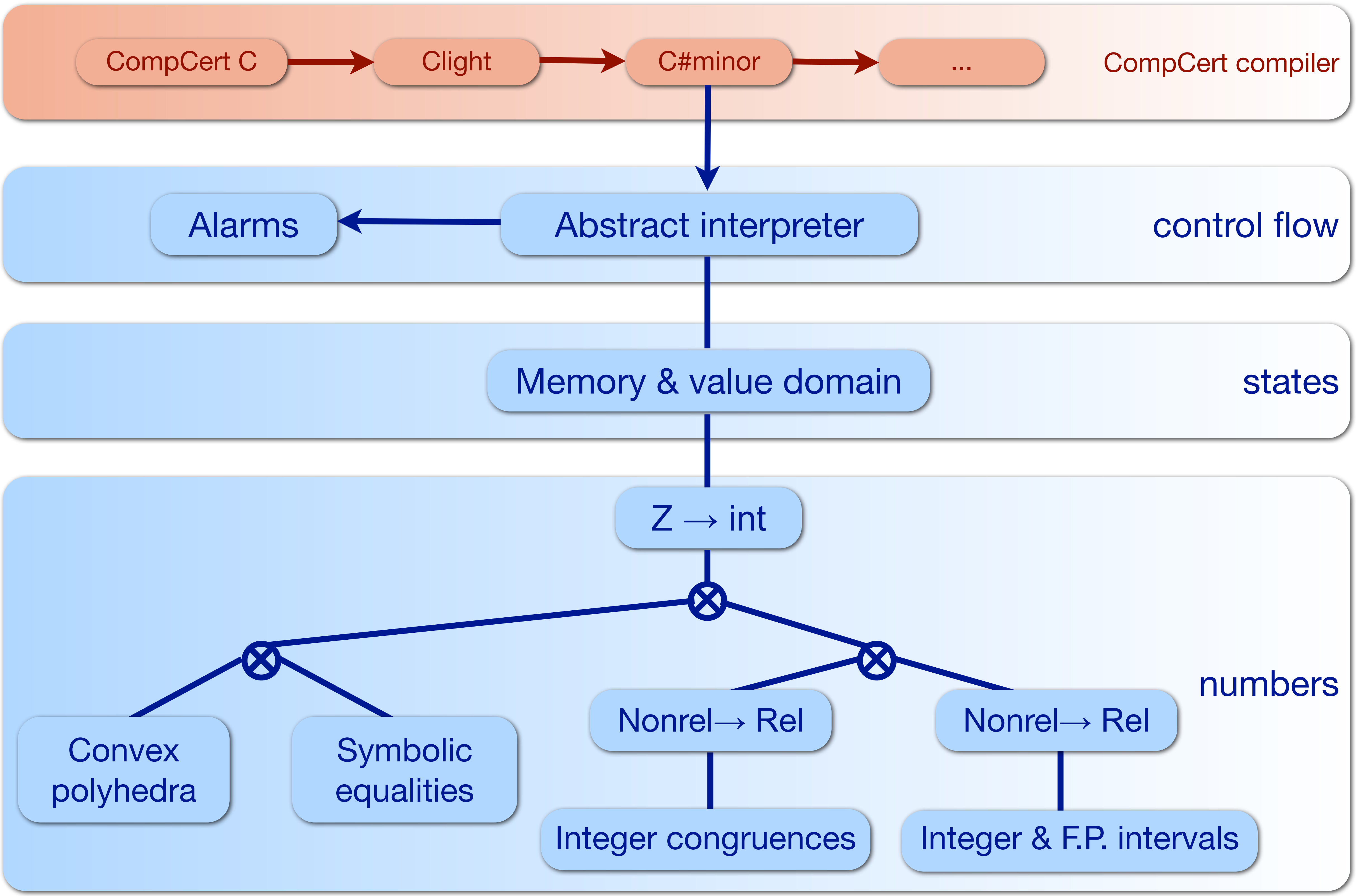
Abstract memory cell: 1 unit of storage

Abstract value: (type, points-to, num)

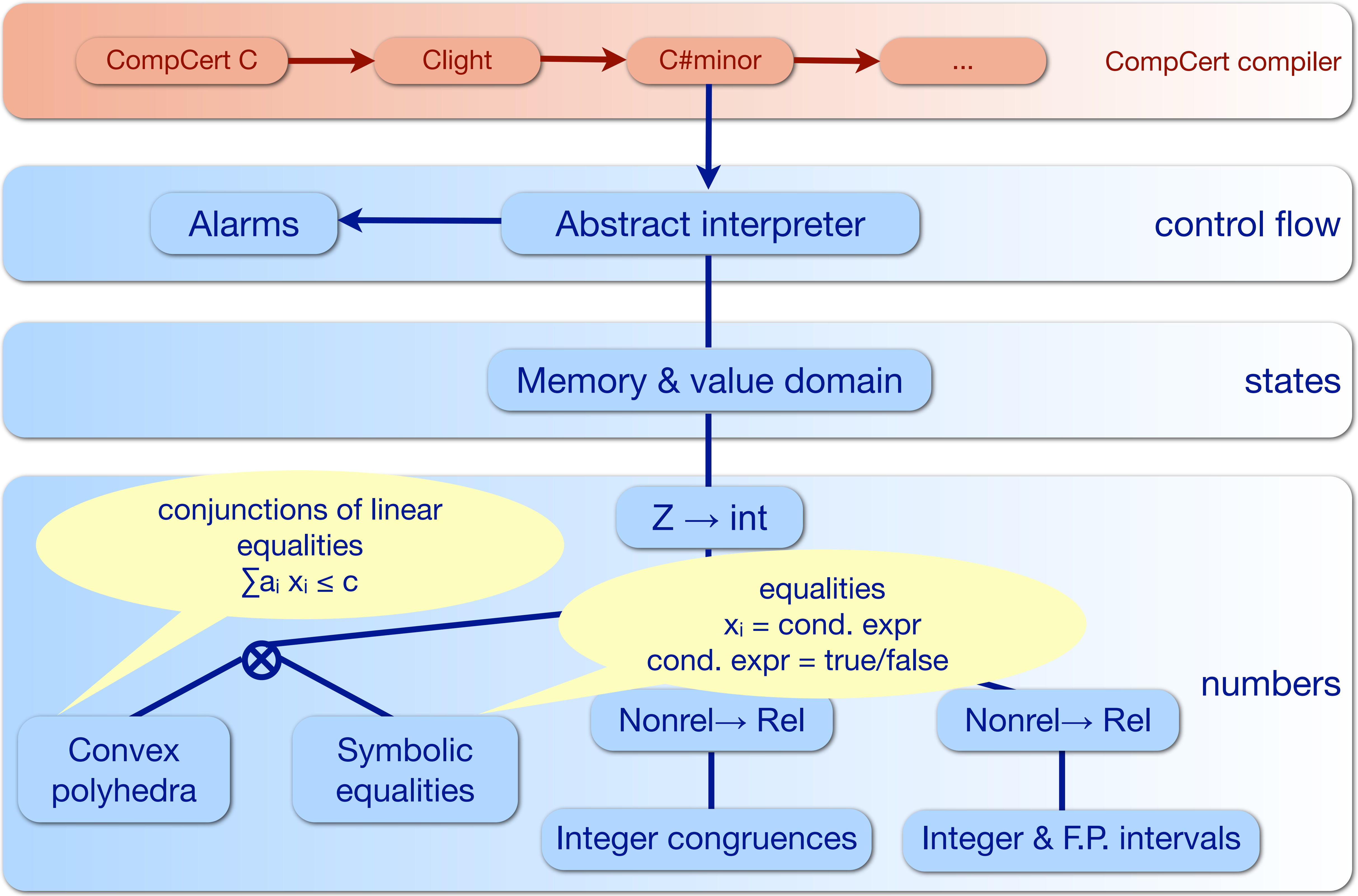
The domain is parameterized by a relational numerical domain where cells act as variables.

Block fusion and strong/weak updates
Ex.: memory store to an array cell. The analysis generates the set of cells that may be accessed. When this set is a singleton, the analysis can perform a strong update.

General architecture
Abstract numerical domains



General architecture
Abstract numerical domains



Combining abstract domains

Implementations of reduced products tend to be specific to the 2 domains being combined.

System of inter-domains communication channels inspired by that of Astrée

- Channels are used by domains when they need information from another domain.
- The information already present in a channel is enriched with information of a query.

Implementation

30 000 lines of Coq, excluding blanks and comments
+ parts reused from CompCert

Bulk of the development: abstract domains for states and for numbers (involve large case analyses and difficult proofs over integer and floating points arithmetic)

Except for the operations over polyhedra, the algorithms are implemented directly in Coq's specification language.

Experimental results

Preliminary experiments on small C programs (up to a few hundred lines)

- CompCert benchmarks
- Cryptographic routines (NaCl library)

Exercise many delicate aspects of the C language: arrays, pointer arithmetic, function pointers, floating-point arithmetic.

The analyzer can take several minutes to analyze a few hundred lines of C.

Future directions

Conclusion

Static analyzer based on abstract interpretation which establishes the absence of run-time errors in C programs (excluding recursion and dynamic allocation)

```
Theorem vanalysis_is_correct:
forall prog res tr,
vanalysis prog = (res, nil) →
program_behaves (semantics prog) (Goes_wrong tr)→
False.
```

Modular architecture supporting the extensible combination of multiple abstract domains (relational and non-relational)

Integrates with CompCert, so that the soundness of the analysis is guaranteed on the compiled code as well

Future directions

Improving the algorithmic efficiency of the static analyzer

- from Coq's integer and FP arithmetic (list of bits) to more efficient libraries
- purely functional data structures used for maps and sets

Extend the memory abstract domain to handle dynamic memory allocation

- one memory cell could stand for several concrete memory locations (*e.g.* all blocks created by `malloc` inside a loop)

Improving the precision of the analysis

- on-the-fly unrolling of certain loops (based on unverified heuristics)

New abstract domains, *e.g.* octagons (linear inequalities $\pm x \pm y \leq c$)

Questions ?

Interval abstraction:
signed or unsigned ?

Interval abstraction:
signed or unsigned ?

```
int f(void) { signed s; unsigned u;
  if (*) u = 2^31 - 1; else u = 2^31;
  if (*) s = 0; else s = -1;
  return u + s;  }
```

Interval abstraction:
signed or unsigned ?

```
int f(void) { signed s; unsigned u;
  if (*) u = 2^31 - 1; else u = 2^31;
  if (*) s = 0; else s = -1;
  return u + s;  }
```

$u \in [2^{31}-1; 2^{31}]$ (unsigned)

Interval abstraction:
signed or unsigned ?

int f(void) { signed s; unsigned u;
  if (*) u = 2^{31} - 1; else u = 2^{31};
  if (*) s = 0; else s = -1;
  return u + s;   }

$u \in [2^{31}-1; 2^{31}]$ (unsigned)

$u \in \top$ (signed)

43

Interval abstraction:
signed or unsigned ?

```
int f(void) { signed s; unsigned u;
  if (*) u = 2^31 - 1; else u = 2^31;
  if (*) s = 0; else s = -1;
  return u + s;  }
```

$u \in [2^{31}-1; 2^{31}]$ (unsigned)

$u \in \top$ (signed)

$s \in \top$ (unsigned)

Interval abstraction:

signed or unsigned ?

`int f(void) { signed s; unsigned u;`

`if (*) u =` 2^{31} `- 1; else u =` 2^{31}`;`

`if (*) s = 0; else s = -1;`

`return u + s; }`

$u \in [2^{31}-1; 2^{31}]$ (unsigned)

$u \in \top$ (signed)

$s \in \top$ (unsigned)

$s \in [-1; 0]$ (signed)