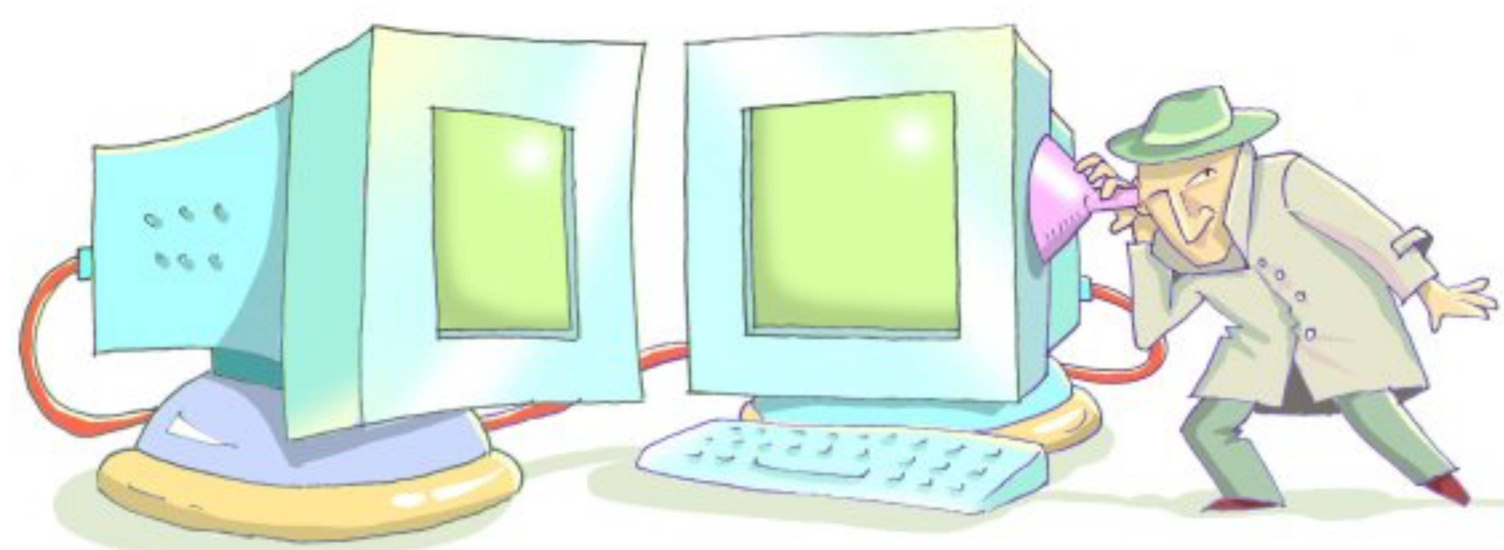
Cryptographic Constant-Time Verification of C programs by Abstract Interpretation

David Pichardie

Cache timing attacks

- Common side-channel: Cache timing attacks
- Exploit the latency between cache hits and misses
- Attackers can recover cryptographic keys
    - Tromer et al (2010), Gullasch et al (2011) show efficient attacks on AES implementations
- Based on the use of look-up tables
    - Access to memory addresses that depend on the key



Constant-time programs

Characterization

- Constant-time programs do not:
  - branch on secrets
  - perform memory accesses that depend on secrets
- There are constant-time implementations of many cryptographic algorithms: AES, DES, RSA, etc

Constant-time programs

Example

Constant-time programs

Example

```
boolean testPIN(int code[]) {
  for (int i=0; i<N; i++) {
    if (code[i] != secret[i]) return false;
  }
  return true;
}
```

Constant-time programs

Example

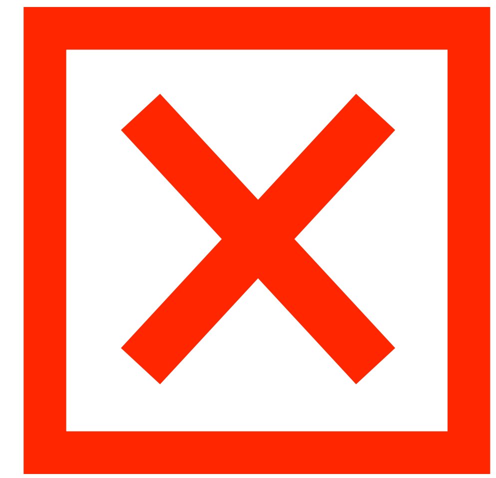
```
boolean testPIN(int code[]) {
  for (int i=0; i<N; i++) {
    if (code[i] != secret[i]) return false;
  }
  return true;
}
```

Not constant-time

Constant-time programs

Example

```
boolean testPIN(int code[]) {
  for (int i=0; i<N; i++) {
    if (code[i] != secret[i]) return false;
  }
  return true;
}
```



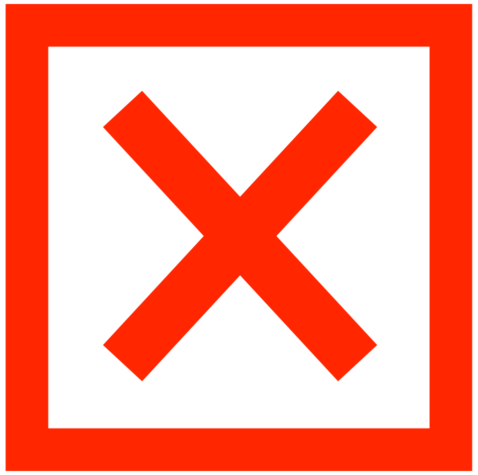
Not constant-time

```
boolean testPIN(int code[]) {
  int diff = 0;
  for (int i=0; i<N; i++) {
    diff = diff | (code[i] ^ secret[i]);
  }
  return (diff == 0);
}
```

Constant-time programs

Example

```
boolean testPIN(int code[]) {
  for (int i=0; i<N; i++) {
    if (code[i] != secret[i]) return false;
  }
  return true;
}
```



Not constant-time

```
boolean testPIN(int code[]) {
  int diff = 0;
  for (int i=0; i<N; i++) {
    diff = diff | (code[i] ^ secret[i]);
  }
  return (diff == 0);
}
```

Constant-time

This lecture

1. Presentation of recent works on the topic
2. Introductory course on abstract interpretation
3. Back to recent research works and conclusion

Verification of constant-time programs

Challenges

- Provide a mechanism to formally check that a program is constant-time
    - static tainting analysis for implementations of cryptographic algorithms
- At low level implementation (C, assembly), advanced static analysis is required
    - secrets depends on data, data depends on control flow, control flow depends on data…
- A high level of reliability is required
    - semantic justifications, Coq mechanizations…
- Attackers exploit executable code, not source code
    - we need guaranties at the assembly level using a compiler toolchain

Background: verifying a compiler

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

= compiler + proof that the compiler does not introduce bugs

Using the Coq proof assistant, X. Leroy proves 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».

Background: verifying a compiler

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

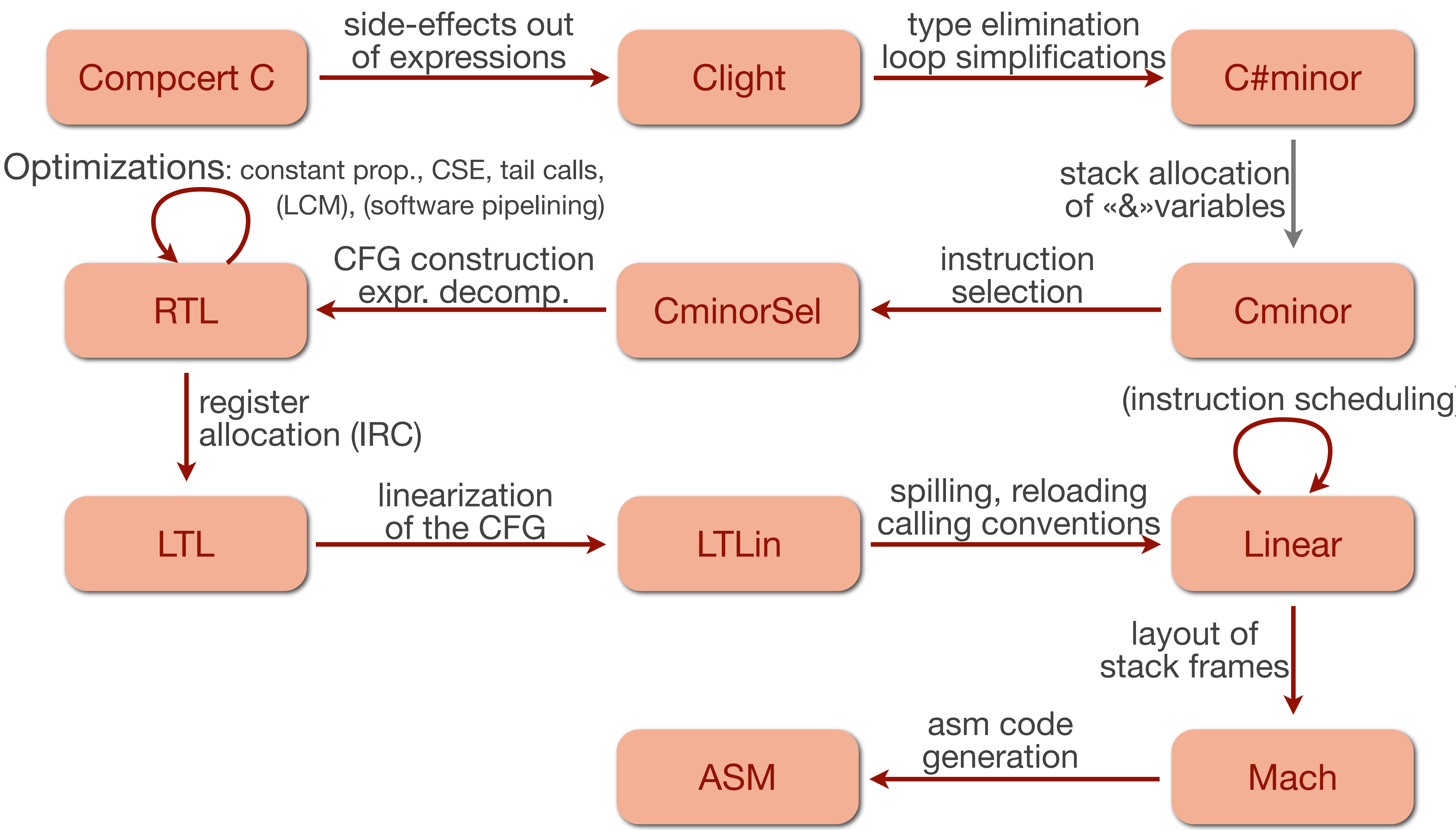
= compiler + proof that the compiler does not introduce bugs

Using the Coq proof assistant, X. Leroy proves the following semantic preservation property:

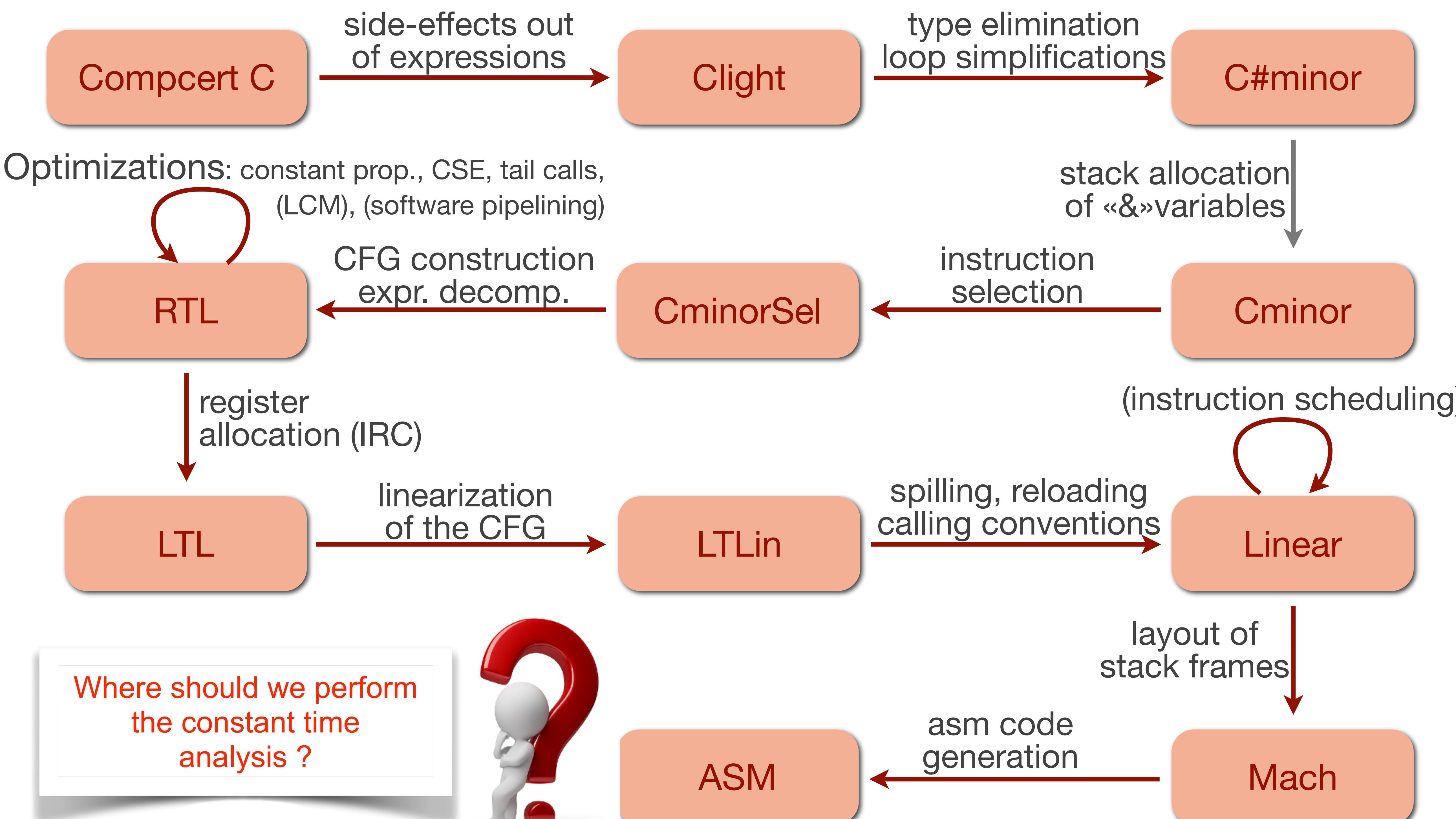
> For all source programs S and compiler-generated code C,
> if the compiler generates machine code C from source S,
> without reporting a compilation error,
> then «C behaves like S».

does not deal with the constant-time security property !

CompCert: 1 compiler, 11 languages



CompCert: 1 compiler, 11 languages



Where should we perform the constant time analysis ?

Our approach

1. Analyse the program at source level

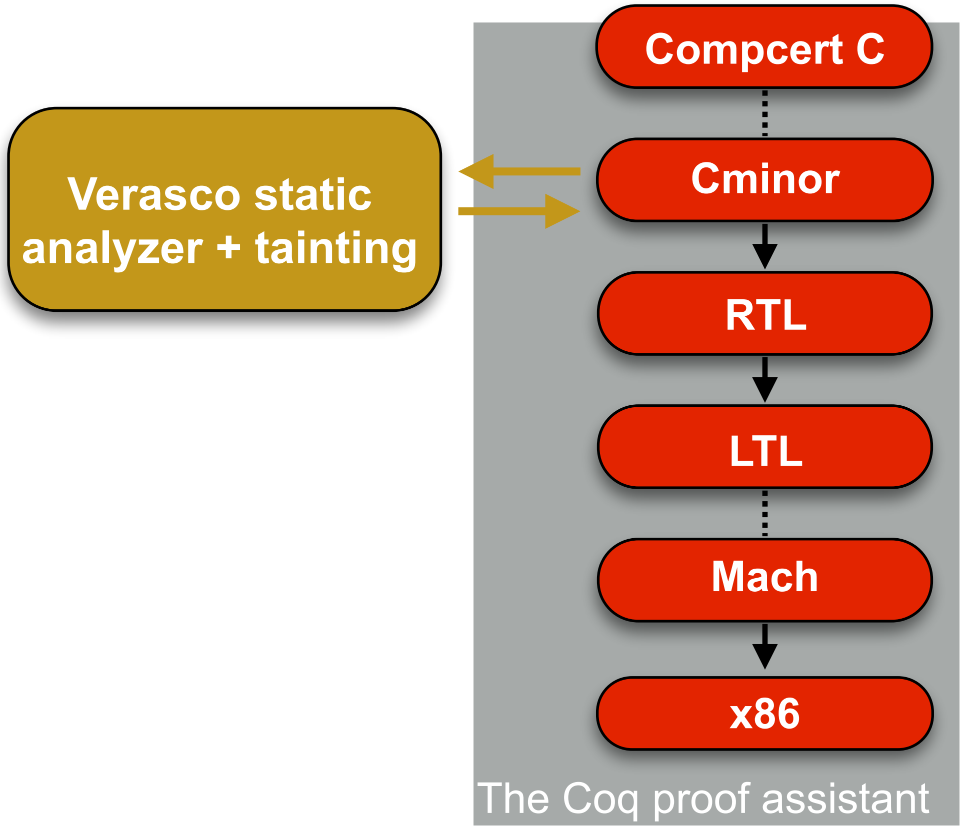
Sandrine Blazy, David Pichardie, Alix Trieu.
Verifying Constant-Time Implementations by Abstract Interpretation.
ESORICS 2017.

2. Make the compiler preserve the property

G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, A. Trieu.
Formal verification of a constant-time preserving C compiler.
POPL 2020.

Constant-time analysis at source level

Sandrine Blazy, David Pichardie, Alix Trieu.
Verifying Constant-Time Implementations by Abstract Interpretation.
ESORICS 2017.



We perform static analysis at (almost) C level

- Based on previous work with a value analyser, Verasco
- We mix Verasco memory tracking with fine-grained tainting
- Main difficulty : alias analysis taking into account pointer arithmetic

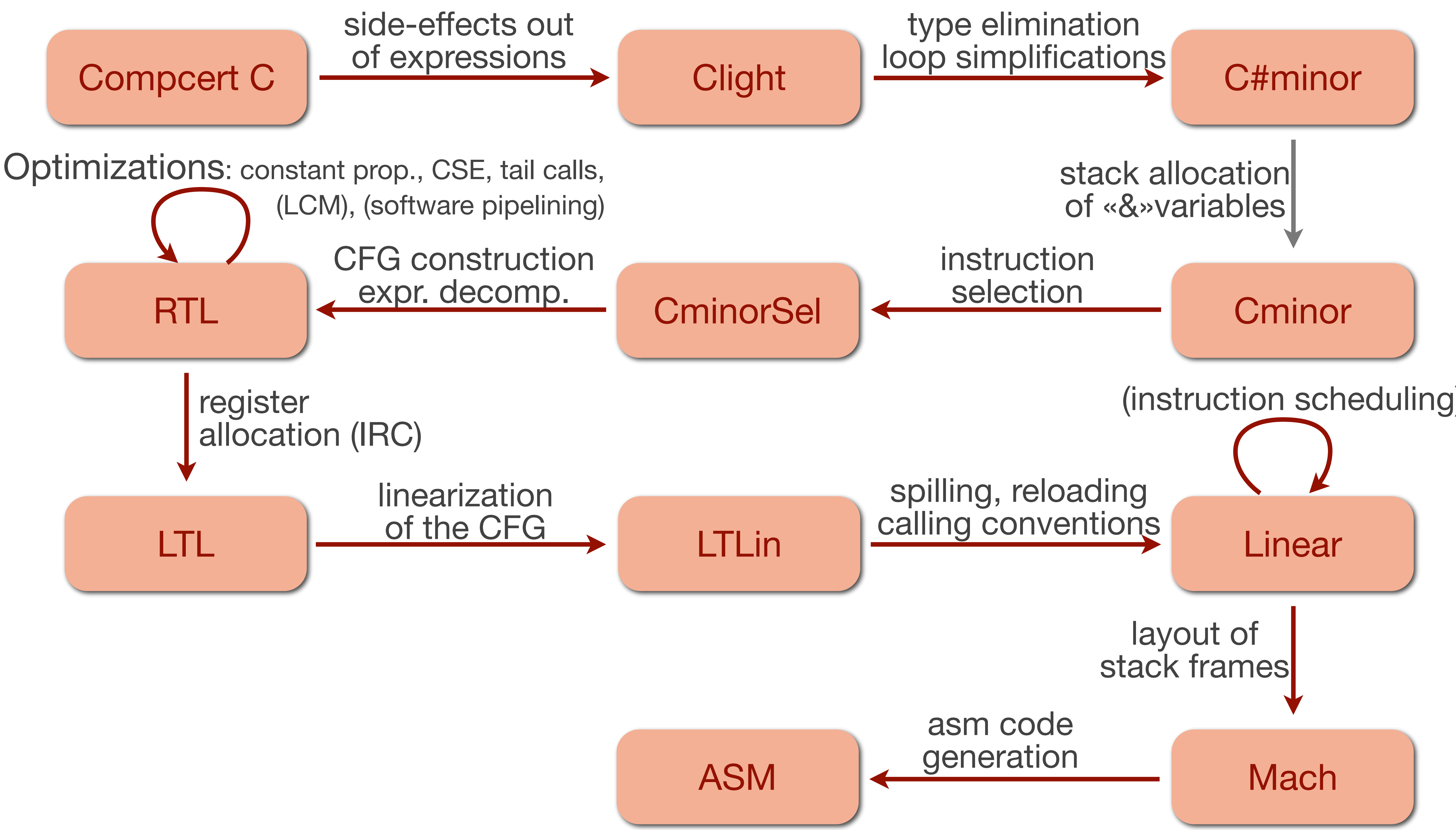
Cf next part on Abstract Interpretation

Preserving the property through compilation

G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, A. Trieu.
Formal verification of a constant-time preserving C compiler.
POPL 2020.

- Makes precise what secure compilation means for cryptographic constant-time
- Provides a machine checked-proof that a mildly modified version of the CompCert compiler preserves cryptographic constant-time
- Explains how to turn a pre-exisiting formally-verified compiler into a formally-verified secure compiler
- Provides a proof toolkit for proving security preservation with simulation diagrams

CompCert: 1 compiler, 11 languages



CompCert preservation proof methodology

- Each langage is given an **operational semantics** $s \xrightarrow{t} s'$ that models a small step transition from a state s to a state s' by emitting a trace of external events t.

- From this stems a notion of **program behavior** (event trace) for complete (possibly infinite) executions.

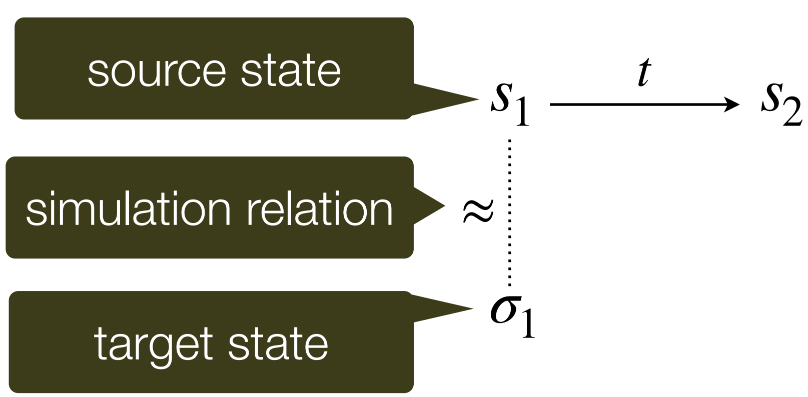
- Behavior preservation is proved via backward and forward simulation, but thanks to langage determinism, **forward simulation** is enough.

CompCert preservation proof methodology

- Each langage is given an **operational semantics** $s \xrightarrow{t} s'$ that models a small step transition from a state s to a state s' by emitting a trace of external events t.

- From this stems a notion of **program behavior** (event trace) for complete (possibly infinite) executions.

- Behavior preservation is proved via backward and forward simulation, but thanks to langage determinism, **forward simulation** is enough.

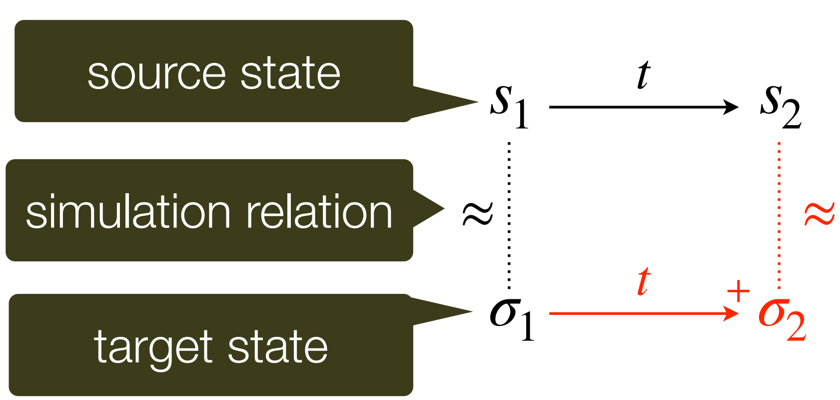


CompCert preservation proof methodology

- Each langage is given an **operational semantics** $s \xrightarrow{t} s'$ that models a small step transition from a state s to a state s' by emitting a trace of external events t.

- From this stems a notion of **program behavior** (event trace) for complete (possibly infinite) executions.

- Behavior preservation is proved via backward and forward simulation, but thanks to langage determinism, **forward simulation** is enough.

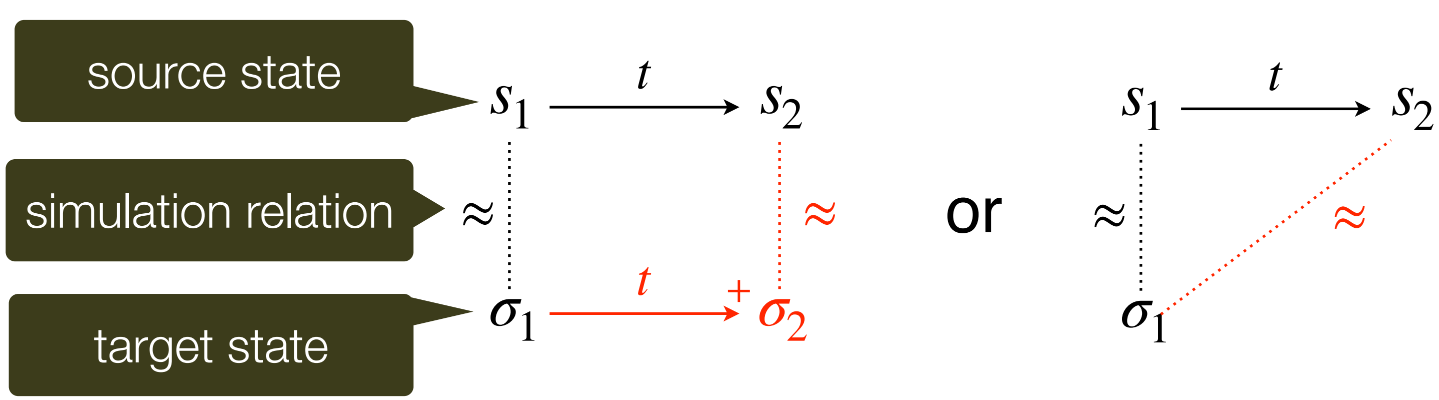


CompCert preservation proof methodology

- Each langage is given an **operational semantics** $s \xrightarrow{t} s'$ that models a small step transition from a state s to a state s' by emitting a trace of external events t.

- From this stems a notion of **program behavior** (event trace) for complete (possibly infinite) executions.

- Behavior preservation is proved via backward and forward simulation, but thanks to langage determinism, **forward simulation** is enough.



with $t = \varepsilon$
and $m(s_2) < m(s_1)$

well founded measure

CompCert:17 preservations proofs

Compiler pass	Explanation on the pass
Cshmgen	Type elaboration, simplification of control
Cminorgen	Stack allocation
Selection	Recognition of operators and addr. modes
RTLgen	Generation of CFG and 3-address code
Tailcall	Tailcall recognition
Inlining	Function inlining
Renumber	Renumbering CFG nodes
ConstProp	Constant propagation
CSE	Common subexpression elimination
Deadcode	Redundancy elimination
Allocation	Register allocation
Tunneling	Branch tunneling
Linearize	Linearization of CFG
CleanupLabels	Removal of unreferenced labels
Debugvar	Synthesis of debugging information
Stacking	Laying out stack frames
Asmgen	Emission of assembly code

Cryptographic constant-time property: defining leakages

- We enrich the CompCert traces of events with **leakages** of two types
  - either the truth value of a condition,
  - or a pointer representing the address of
    - either a memory access (i.e., a load or a store)
    - or a called function
- Using **event erasure**, from $s \xrightarrow{t} s'$ we can extract
  - the compile-only judgment $s \xrightarrow{t}_{\text{comp}} s'$
  - the leak-only judgment $s \xrightarrow{t}_{\text{leak}} s'$
- **Program leakage** is defined as the behavior of the $\rightarrow_{\text{leak}}$ semantics

Cryptographic constant-time property: preservation

- We note $\varphi(s, s')$ the fact that two initial states s and s' share the same values for public inputs, but may differ on the values of secret inputs

- A program is **constant-time secure w.r.t. φ** if for two initial states s and s' such that $\varphi(s, s')$ holds, then both leak-only executions starting from s and s' observe the same leakage

$$\begin{array}{l} s \xrightarrow{\quad t \quad}{}^{*}_{\text{leak}} \\ \varphi \,\big| \\ s' \xrightarrow{\quad t' \quad}{}^{*}_{\text{leak}} \end{array} \quad \text{implies} \quad t = t'$$

Cryptographic constant-time property: preservation

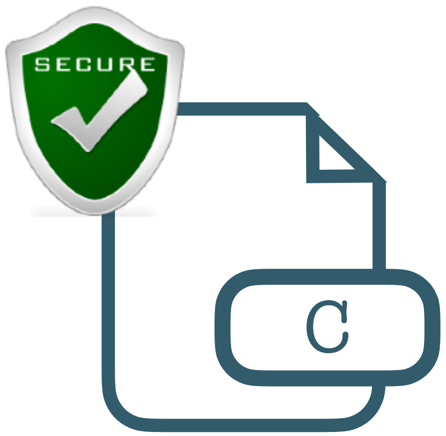
- We note $\varphi(s, s')$ the fact that two initial states s and s' share the same values for public inputs, but may differ on the values of secret inputs

- A program is **constant-time secure w.r.t. φ** if for two initial states s and s' such that $\varphi(s, s')$ holds, then both leak-only executions starting from s and s' observe the same leakage

Main Theorem (Constant-Time security preservation): Let P be a safe Clight source program that is compiled into an x86 assembly program P'. If P is constant-time w.r.t. φ, then so is P'.

Take-away message

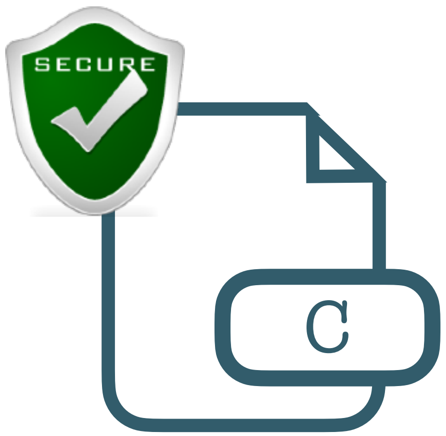
Take-away message



Static analysis

- Abstract Interpretation can secure program art source level

Take-away message

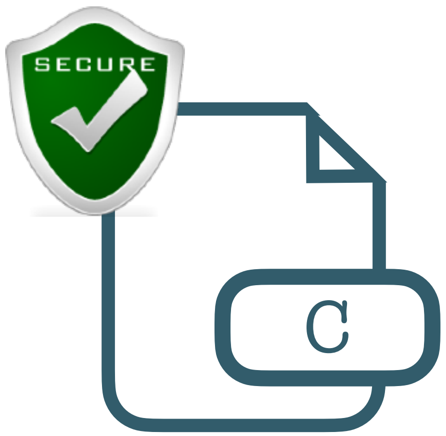


Static analysis

- Abstract Interpretation can secure program art source level

- But we must make sure the compiler will preserve the security policy

Take-away message



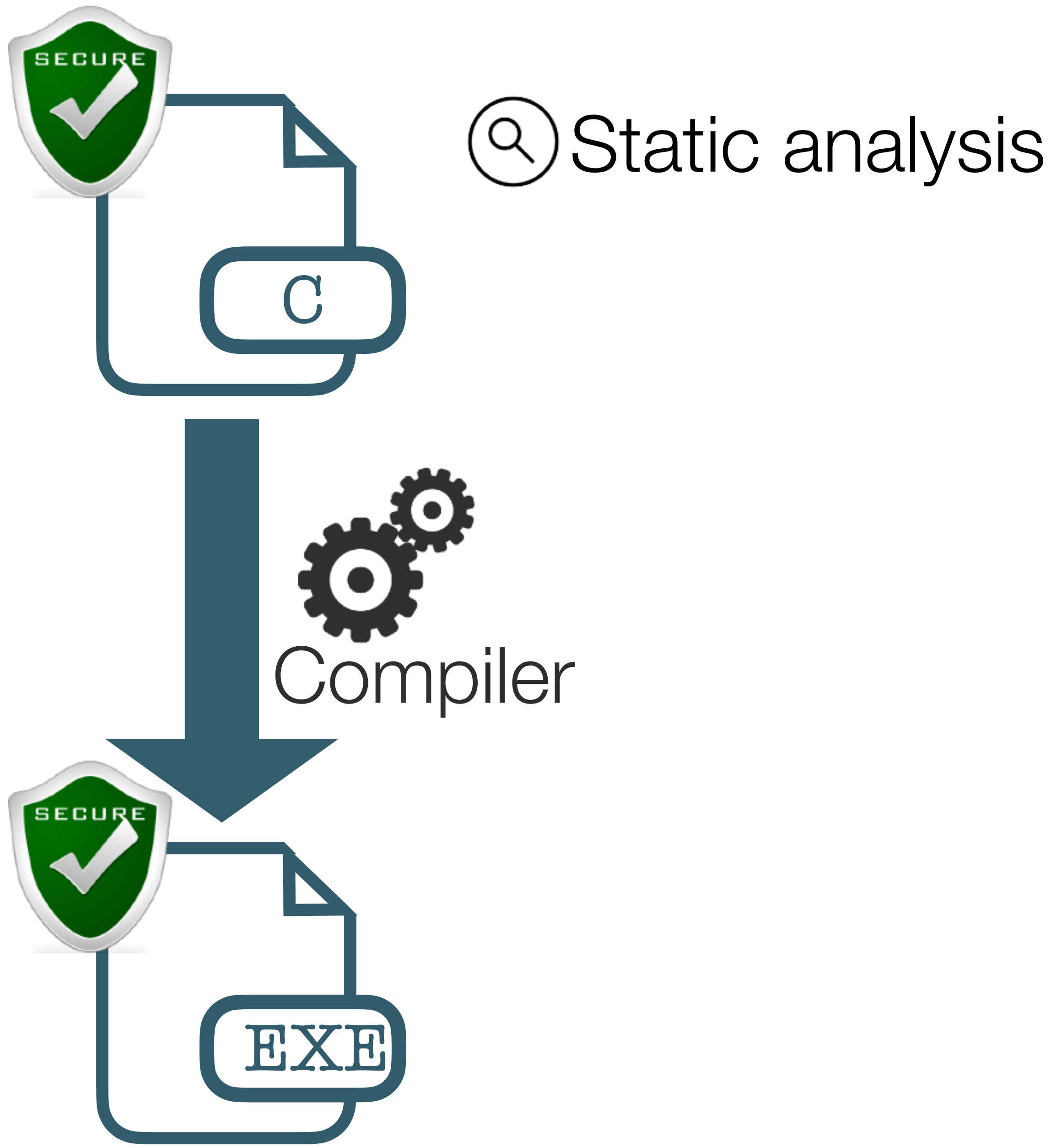
Static analysis

- Abstract Interpretation can secure program art source level

- But we must make sure the compiler will preserve the security policy

Take-away message

- Abstract Interpretation can secure program art source level
- But we must make sure the compiler will preserve the security policy
- Let's start the Abstract Interpretation lecture!



Abstract Interpretation (an introduction)

Static program analysis

The goals of static program analysis

- to prove properties about the run-time behaviour of a program
- in a fully automatic way
- without actually executing this program

Applications

- code optimisation
- error detection (array out of bound access, null pointers)
- proof support (invariant extraction)

Abstract Interpretation

[Cousot&Cousot 75, 76, 77, 79, 80, 81, 82, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 00, 01, 02, 03, 04, 05, 06, 07,08,09,10,11,. . .] [1]

Patrick Cousot

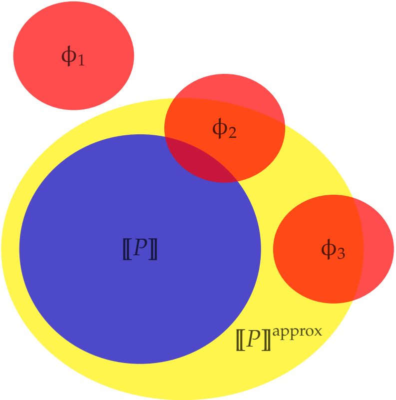
Radhia Cousot

A theory which unifies a large variety of static analysis

- formalises the approximated analyse of programs
- allows to compare relative precision of analyses
- facilitates the conception of sophisticated analyses

1. See `http://www.di.ens.fr/~cousot/`

Static analysis computes approximations [2]



- P is safe w.r.t. ϕ_1 and the analyser proves it

$$[\![P]\!] \cap \phi_1 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_1 = \emptyset$$

- P is unsafe w.r.t. ϕ_2 and the analyser warns about it

$$[\![P]\!] \cap \phi_2 \neq \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_2 \neq \emptyset$$

- but P is safe w.r.t. ϕ_3 and the analyser can't prove it (this is called a *false alarm*)

$$[\![P]\!] \cap \phi_3 = \emptyset \qquad [\![P]\!]^{\text{approx}} \cap \phi_3 \neq \emptyset$$

$[\![P]\!]$:	concrete semantics (e.g. set of reachable states)	(not computable)
ϕ_1, ϕ_2, ϕ_3 :	erroneous/dangerous set of states	(computable)
$[\![P]\!]^{\text{approx}}$:	analyser result (here over-approximation)	(computable)

2. see `https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x}, \mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {                    }
while (x<6) {
  if (?) {
      {                    }
    y = y+2;
      {                    }
  };
      {                                }
  x = x+1;
      {                                }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {(0,0)                }
while (x<6) {
  if (?) {
      {                     }
    y = y+2;
      {                     }
  };
      {                                  }
  x = x+1;
      {                                  }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x}, \mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {(0,0)                }
while (x<6) {
  if (?) {
      {(0,0)                }
    y = y+2;
      {                     }
  };
      {                                }
  x = x+1;
      {                                }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {(0,0)              }
while (x<6) {
  if (?) {
      {(0,0)              }
    y = y+2;
      {(0,2)              }
  };
      {                             }
  x = x+1;
      {                             }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
      {(0,0)                }
while (x<6) {
  if (?) {
      {(0,0)                }
    y = y+2;
      {(0,2)                }
  };
      {(0,0),(0,2)                        }
  x = x+1;
      {                                   }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
        {(0,0)                }
while (x<6) {
  if (?) {
        {(0,0)                }
    y = y+2;
        {(0,2)                }
  };
        {(0,0),(0,2)                }
  x = x+1;
        {(1,0),(1,2)                }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
        {(0,0),(1,0),(1,2)     }
while (x<6) {
  if (?) {
        {(0,0)                 }
    y = y+2;
        {(0,2)                 }
  };
        {(0,0),(0,2)                 }
  x = x+1;
        {(1,0),(1,2)                 }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
        {(0,0), (1,0), (1,2)     }
while (x<6) {
  if (?) {
        {(0,0), (1,0), (1,2)     }
    y = y+2;
        {(0,2)                   }
  };
        {(0,0), (0,2)                        }
  x = x+1;
        {(1,0), (1,2)                        }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x}, \mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
        {(0,0),(1,0),(1,2)    }
while (x<6) {
  if (?) {
        {(0,0),(1,0),(1,2)    }
    y = y+2;
        {(0,2),(1,2),(1,4)    }
  };
        {(0,0),(0,2)                    }
  x = x+1;
        {(1,0),(1,2)                    }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
        {(0,0),(1,0),(1,2)    }
while (x<6) {
  if (?) {
        {(0,0),(1,0),(1,2)    }
    y = y+2;
        {(0,2),(1,2),(1,4)    }
  };
        {(0,0),(0,2),(1,0),(1,2),(1,4)    }
  x = x+1;
        {(1,0),(1,2)                      }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x},\mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
        {(0,0),(1,0),(1,2)     }
while (x<6) {
  if (?) {
        {(0,0),(1,0),(1,2)     }
    y = y+2;
        {(0,2),(1,2),(1,4)     }
  };
        {(0,0),(0,2),(1,0),(1,2),(1,4)     }
  x = x+1;
        {(1,0),(1,2),(2,0),(2,2),(2,4)     }
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Collecting semantics

- A state property is a subset in $\mathcal{P}(\mathbb{Z}^2)$ of $(\mathtt{x}, \mathtt{y})$ values.
- When a point is reached for a second time we make an union with the previous property.
- We "execute" the program until stability
  - It may take an infinite number of steps...
  - But the limit always exists (explained later)

```
x = 0; y = 0;
        {(0,0),(1,0),(1,2),...}
while (x<6) {
  if (?) {
        {(0,0),(1,0),(1,2),...}
    y = y+2;
        {(0,2),(1,2),(1,4),...}
  };
        {(0,0),(0,2),(1,0),(1,2),(1,4),...}
  x = x+1;
        {(1,0),(1,2),(2,0),(2,2),(2,4),...}
}
        {(6,0),(6,2),(6,4),(6,6),...}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{rcl}
P & ::= & x\,C\,0 \wedge y\,C\,0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
```

$\mathtt{x} = 0 \wedge \mathtt{y} = 0$

```
while (x<6) {
  if (?) {

    y = y+2;

  };

  x = x+1;

}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$\begin{array}{rcl} P & ::= & x\,C\,0 \;\wedge\; y\,C\,0 \\ C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant \end{array}$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
      x = 0 ∧ y = 0
while (x<6) {
  if (?) {
      x = 0 ∧ y = 0
    y = y+2;


  };


  x = x+1;


}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{rcl}
P & ::= & x\,C\,0 \wedge y\,C\,0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
      x = 0 ∧ y = 0
while (x<6) {
  if (?) {
      x = 0 ∧ y = 0
    y = y+2;
      x = 0 ∧ y > 0  over-approximation!
  };

  x = x+1;

}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$\begin{array}{lcl} P & ::= & x\,C\,0 \wedge y\,C\,0 \\ C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant \end{array}$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x = 0 ∧ y = 0
while (x<6) {
  if (?) {
        x = 0 ∧ y = 0
    y = y+2;
        x = 0 ∧ y > 0
  };
        x = 0 ∧ y ⩾ 0
  x = x+1;


}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{rcl}
P & ::= & x\, C\, 0 \wedge y\, C\, 0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x = 0 ∧ y = 0
while (x<6) {
  if (?) {
        x = 0 ∧ y = 0
    y = y+2;
        x = 0 ∧ y > 0
  };
        x = 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0  over-approximation!
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{lcl}
P & ::= & x\,C\,0 \wedge y\,C\,0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x ⩾ 0 ∧ y ⩾ 0
while (x<6) {
  if (?) {
        x = 0 ∧ y = 0
    y = y+2;
        x = 0 ∧ y > 0
  };
        x = 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{rcl}
P & ::= & x\,C\,0 \wedge y\,C\,0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x ⩾ 0 ∧ y ⩾ 0
while (x<6) {
  if (?) {
        x ⩾ 0 ∧ y ⩾ 0
    y = y+2;
        x = 0 ∧ y > 0
  };
        x = 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{lcl}
P & ::= & x \, C \, 0 \wedge y \, C \, 0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x ⩾ 0 ∧ y ⩾ 0
while (x<6) {
  if (?) {
        x ⩾ 0 ∧ y ⩾ 0
    y = y+2;
        x ⩾ 0 ∧ y > 0
  };
        x = 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{lcl}
P & ::= & x \; C \; 0 \; \wedge \; y \; C \; 0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x ⩾ 0 ∧ y ⩾ 0
while (x<6) {
  if (?) {
        x ⩾ 0 ∧ y ⩾ 0
    y = y+2;
        x ⩾ 0 ∧ y > 0
  };
        x ⩾ 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{rcl}
P & ::= & x\, C\, 0 \wedge y\, C\, 0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x ⩾ 0 ∧ y ⩾ 0
while (x<6) {
  if (?) {
        x ⩾ 0 ∧ y ⩾ 0
    y = y+2;
        x ⩾ 0 ∧ y > 0
  };
        x ⩾ 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0
}
```

A flavor of abstract interpretation

Abstract interpretation executes programs on state properties instead of states.

Approximation

- The set of manipulated properties may be restricted to ensure computability of the semantics.
  Example : sign of variables

$$
\begin{array}{rcl}
P & ::= & x \, C \, 0 \wedge y \, C \, 0 \\
C & ::= & < \mid \leqslant \mid = \mid > \mid \geqslant
\end{array}
$$

- To stay in the domain of selected properties, we over-approximate the concrete properties.

```
x = 0; y = 0;
        x ⩾ 0 ∧ y ⩾ 0
while (x<6) {
  if (?) {
        x ⩾ 0 ∧ y ⩾ 0
    y = y+2;
        x ⩾ 0 ∧ y > 0
  };
        x ⩾ 0 ∧ y ⩾ 0
  x = x+1;
        x > 0 ∧ y ⩾ 0
}
        x ⩾ 0 ∧ y ⩾ 0
```

An other example : the interval analysis

For each point k and each numeric variable x, we infer an interval in which x must belong to.

Example : insertion sort, array access verification

```
assume(T.length=100); i=1;
```

$\{i \in [1, 100]\}$

```
while (i<T.length) {
```

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

```
    p = T[i]; j = i-1;
```

$\{i \in [1, 99],\ j \in [-1, 98]\}$

```
    while (0<=j and T[j]>p) {
```

$\{i \in [1, 99],\ j \in [0, 98]\}$

```
        T[j]=T[j+1]; j = j-1;
```

$\{i \in [1, 99],\ j \in [-1, 97]\}$

```
    };
```

$\{i \in [1, 99],\ j \in [-1, 98]\}$

```
    T[j+1]=p; i = i+1;
```

$\{i \in [2, 100],\ j = [-1, 98]\}$

```
};
```

$\{i = 100\}$

An other example : the polyhedral analysis

For each point k and we infer invariant linear equality and inequality relationships among variables.

Example : insertion sort, array access verification

```
assume(T.length>=1); i=1;
                                        {1 ⩽ i ⩽ T.length}
while i<T.length {
                                        {1 ⩽ i ⩽ T.length − 1}
    p = T[i]; j = i-1;
                                        {1 ⩽ i ⩽ T.length − 1 ∧ −1 ⩽ j ⩽ i − 1}
    while 0<=j and T[j]>p {
                                        {1 ⩽ i ⩽ T.length − 1 ∧ 0 ⩽ j ⩽ i − 1}
        T[j]=T[j+1]; j = j-1;
                                        {1 ⩽ i ⩽ T.length − 1 ∧ −1 ⩽ j ⩽ i − 2}
    };
                                        {1 ⩽ i ⩽ T.length − 1 ∧ −1 ⩽ j ⩽ i − 1}
    T[j+1]=p; i = i+1;
                                        {2 ⩽ i ⩽ T.length + 1 ∧ −1 ⩽ j ⩽ i − 2}
};
                                        {i = T.length}
```

This lecture

1. Introduction
2. Intermediate representation : syntax and semantics
3. Collecting semantics
4. Just put some $^\sharp$...
5. Building a generic abstract interpreter
6. Numeric abstraction by intervals
7. Widening/Narrowing
8. Polyhedral abstract interpretation
9. Readings

Outline

A flowchart representation of program

The standard model of program in static analysis is *control flow graph*.
The graph model used here :

- the nodes are program point $k \in \mathbb{P}$,
- the edges are labeled with *basic instructions*

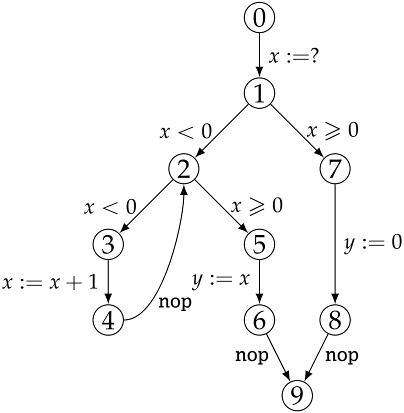
$$\begin{array}{rlll} \mathit{Instr} ::= & x \ \texttt{:=} \ \mathit{Exp} & & \textit{assignment} \\ | & \texttt{nop} & & \\ | & \texttt{assume}\ \mathit{Test} & & \textit{execution continues only if} \\ & & & \textit{the test successes} \end{array}$$

  (*Exp* and *Test* to be defined in the next slide)
- formally a cfg is a couple $(k_{\mathrm{init}}, S)$with
  - $k_{\mathrm{init}} \in \mathbb{P}$: the entry point,
  - $S \subseteq \mathbb{P} \times \mathit{Instr} \times \mathbb{P}$ the set of edges.

Remark : data-flow analyses are generally based on other versions of control flow graph (nodes are put in instructions).

Example

```
x = read_input()
if x<0 {
  while (x<0) x++
  y = x
} else {
  y = 0
}
```



Expression and test language for today

In OCaml syntax

We will restrict our study to a simple numeric subset of Java expressions

```
type binop =
  | Add | Sub | Mult

type expr =
  | Const of int
  | Var of var
  | Binop of binop * expr * expr

type comp = Eq | Neq | Le | Lt

type test =
  | Cond of expr * comp * expr        (* e1 cmp e2 *)
  | And of test * test                (* t1 && t2 *)
  | Or of test * test                 (* t1 || t2 *)


type instr =
  | Nop
  | Forget of var                     (* x := ? *)
  | Assign of var * expr              (* x := e *)
  | Assume of test                    (* assume t *)
```

Semantics

Semantic domains

$$\begin{array}{rcl} Env & \stackrel{\text{def}}{=} & \mathbb{V} \to \mathbb{Z} \\ State & \stackrel{\text{def}}{=} & \mathbb{P} \times Env \end{array}$$

Semantics of expressions (standard then omitted)

$$\mathcal{A}[\![e]\!]\,\rho \in \mathbb{Z}, \quad e \in Exp,\ \rho \in Env$$

Semantics of tests (standard then omitted)

$$\mathcal{B}[\![t]\!]\,\rho \in \mathbb{B}, \quad t \in Test,\ \rho \in Env$$

Small-step semantics of cfg

We first define the semantics of instructions : $\xrightarrow{i} \subseteq Env \times Env$

$$\frac{v \in \mathbb{Z}}{\rho \xrightarrow{x := ?} \rho[x \mapsto v]} \qquad \frac{}{\rho \xrightarrow{x := a} \rho[x \mapsto \mathcal{A}[\![a]\!]\rho]} \qquad \frac{\mathcal{B}[\![t]\!]\rho = \mathbf{tt}}{\rho \xrightarrow{t} \rho}$$

Then a small-step relation $\rightarrow_{cfg} \subseteq State \times State$ for a $cfg = (k_{\text{init}}, S)$

$$\frac{(k_1, i, k_2) \in S \qquad \rho_1 \xrightarrow{i} \rho_2}{(k_1, \rho_1) \rightarrow_{cfg} (k_2, \rho_2)}$$

Reachable states for control flow graphs

$$[\![cfg]\!] = \{\ (k, \rho) \mid \exists \rho_0 \in Env,\ (k_{\text{init}}, \rho_0) \rightarrow^*_{cfg} (k, \rho)\ \}$$

where $cfg = (k_{\text{init}}, S)$

Starting from an other semantics ?

Remark : for the purpose of the talk, we directly start with a *cfg*-semantics. We could have started from a more conventionnal operational semantics. See

- Patrick Cousot, *MIT Course 16.399 : Abstract Interpretation*, `http://www.mit.edu/~cousot/`
- David Cachera and David Pichardie. *A certified denotational abstract interpreter*. In Proc. of ITP-10, 2010.

Outline

Collecting Semantics

We will consider a collecting semantics that give us the set of reachable states $[\![p]\!]^{\text{col}}_k$ at each program points k.

$$\forall k \in \mathbb{P},\ [\![p]\!]^{\text{col}}_k = \{\ \rho \mid (k, \rho) \in [\![p]\!]\ \}$$

Theorem

$[\![p]\!]^{col}$ may be characterized as the least fixpoint of the following equation system.

$$\forall k \in labels(p),\ X_k = X_k^{init} \cup \bigcup_{(k',i,k)\in p} [\![i]\!]\,(X_{k'})$$

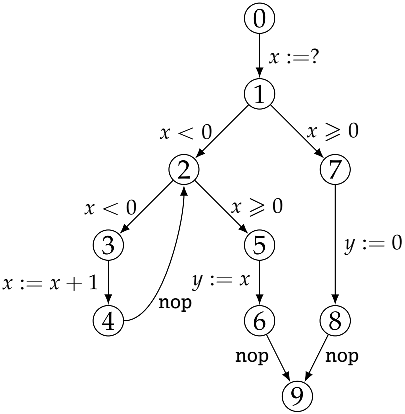
with $X_k^{\text{init}} = \begin{cases} Env & \text{if } k = k_{\text{init}} \\ \emptyset & \text{otherwise} \end{cases}$

and

$$\forall i \in Instr,\ \forall X \subseteq Env,\ [\![i]\!]\,(X) = \left\{\ \rho_2 \mid \exists \rho_1 \in X,\ \rho_1 \xrightarrow{i} \rho_2\ \right\} = \text{post}\left[\xrightarrow{i}\right](X)$$

Example

For the following program, $[\![P]\!]^{\text{col}}$ is the least solution of the following equation system :



$$
\begin{aligned}
X_0 &= Env \\
X_1 &= [\![x :=?]\!]\,(X_0) \\
X_2 &= [\![x < 0]\!]\,(X_1) \cup X_4 \\
X_3 &= [\![x < 0]\!]\,(X_2) \\
X_4 &= [\![x := x + 1]\!]\,(X_3) \\
X_5 &= [\![x \geqslant 0]\!]\,(X_2) \\
X_6 &= [\![y := x]\!]\,(X_5) \\
X_7 &= [\![x \geqslant 0]\!]\,(X_1) \\
X_8 &= [\![y := 0]\!]\,(X_7) \\
X_9 &= X_6 \cup X_8
\end{aligned}
$$

Fixpoint Lattice Theory

Theorem (Knaster-Tarski)

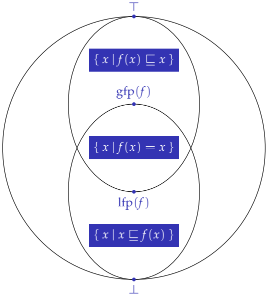
In a complete lattice $(A, \sqsubseteq, \bigsqcup)$*, for all monotone functions* $f \in A \to A$*, the least fixpoint* $\mathrm{lfp}(f)$ *of* f *exists and is* $\bigsqcap\{x \in A \mid f(x) \sqsubseteq x\}$.

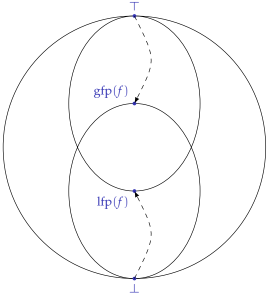
Theorem (Kleene fixpoint theorem)

In a complete lattice $(A, \sqsubseteq, \bigsqcup)$*, for all continuous function* $f \in A \to A$*, the least fixpoint* $\mathrm{lfp}(f)$ *of* f *is equal to* $\bigsqcup\{ f^n(\bot) \mid n \in \mathbb{N} \}$.

Theorem

Let $(A, \sqsubseteq)$ *a poset that verifies the ascending chain condition and* f *a monotone function. The sequence* $\bot, f(\bot), \ldots, f^n(\bot), \ldots$ *eventually stabilises. Its limit is the least fixpoint of* f.





Collecting semantics and exact analysis

The $(X_k)_{i=1..N}$ are hence specified as the least solution of a fixpoint equation system

$$X_k = F_k(X_1, X_2, \ldots, X_N) \ \ , \ \ k \in labels(p)$$

or, equivalently $\vec{X} = \vec{F}(\vec{X})$.

Exact analysis :

- Thanks to Knaster-Tarski, the least solution exists (complete lattice, F_k are monotone functions),
- Kleen fixpoint theorem (F_k are continuous functions) says it is the limit of

$$X_k^0 = \emptyset \ \ , \ \ X_k^{n+1} = F_k(X_1^n, X_2^n, \ldots, X_N^n)$$

Uncomputable problem :

- Representing the X_k may be hard (infinite sets)
- The limit may not be reachable in a finite number of steps

Approximate analysis

Exact analysis :
Least solution of $X = F(X)$ in the complete lattice $(\mathcal{P}(Env)^N, \subseteq, \cup, \cap)$
or limit of $X^0 = \bot, X^{n+1} = F(X^n)$

Approximate analysis :

- Static approximation : we replace the concrete lattice $(\mathcal{P}(Env), \subseteq, \cup, \cap)$ by an abstract lattice $(L^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp)$
  - whose elements can be (efficiently) represented in computers,
  - in which we know how to compute $\sqcup^\sharp, \sqcap^\sharp, \sqsubseteq^\sharp, \ldots$

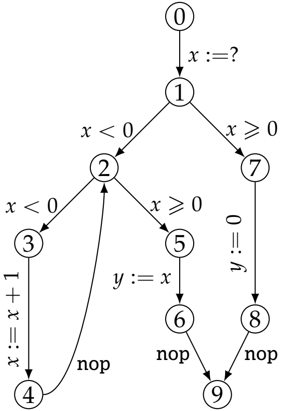
  and we "transpose" the equation $X = F(X)$ of $\mathcal{P}(Env)^N$ into $(L^\sharp)^N$.
- Dynamic approximation : when $L^\sharp$ does not verifies the ascending chain condition, the iterative computation may not terminate in a finite number of steps (or sometimes too slowly). In this case, we can only approximate the limit (see widening/narrowing).

Outline

Just put some $^\sharp$...

From $\mathcal{P}(Env)$ to $Env^\sharp$

control flow graph	collecting semantics	abstract semantics



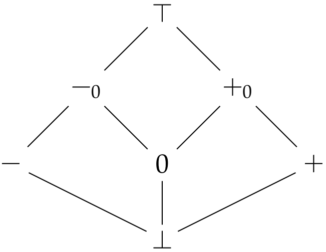
$$
\begin{array}{lcl}
X_0 & = & Env \\
X_1 & = & [\![x :=?]\!]\,(X_0) \\
X_2 & = & [\![x < 0]\!]\,(X_1) \cup X_4 \\
X_3 & = & [\![x < 0]\!]\,(X_2) \\
X_4 & = & [\![x := x + 1]\!]\,(X_3) \\
X_5 & = & [\![x \geqslant 0]\!]\,(X_2) \\
X_6 & = & [\![y := x]\!]\,(X_5) \\
X_7 & = & [\![x \geqslant 0]\!]\,(X_1) \\
X_8 & = & [\![y := 0]\!]\,(X_7) \\
X_9 & = & X_6 \cup X_8
\end{array}
$$

$$
\begin{array}{lcl}
X_0^\sharp & = & \top_{Env}^\sharp \\
X_1^\sharp & = & [\![x :=?]\!]^\sharp\,(X_0^\sharp) \\
X_2^\sharp & = & [\![x < 0]\!]^\sharp\,(X_1^\sharp) \sqcup^\sharp X_4^\sharp \\
X_3^\sharp & = & [\![x < 0]\!]^\sharp\,(X_2^\sharp) \\
X_4^\sharp & = & [\![x := x + 1]\!]^\sharp\,(X_3^\sharp) \\
X_5^\sharp & = & [\![x \geqslant 0]\!]^\sharp\,(X_2^\sharp) \\
X_6^\sharp & = & [\![y := x]\!]^\sharp\,(X_5^\sharp) \\
X_7^\sharp & = & [\![x \geqslant 0]\!]^\sharp\,(X_1^\sharp) \\
X_8^\sharp & = & [\![y := 0]\!]^\sharp\,(X_7^\sharp) \\
X_9^\sharp & = & X_6^\sharp \sqcup^\sharp X_8^\sharp
\end{array}
$$

Abstract semantics : the ingredients

- A lattice structure $(Env^\sharp, \sqsubseteq^\sharp_{Env}, \sqcup^\sharp_{Env}, \sqcap^\sharp_{Env}, \perp^\sharp_{Env}, \top^\sharp_{Env})$
  - $\sqsubseteq^\sharp_{Env}$ is an approximation of $\subseteq$
  - $\sqcup^\sharp_{Env}$ is an approximation of $\cup$
  - $\sqcap^\sharp_{Env}$ is an approximation of $\cap$
  - $\perp^\sharp_{Env}$ is an approximation of $\emptyset$
  - $\top^\sharp_{Env}$ is an approximation of Env
- For all $x \in \mathbb{V}$,
  $[\![x :=?]\!]^\sharp \in Env^\sharp \to Env^\sharp$ an approximation of $[\![x :=?]\!]$
- For all $x \in \mathbb{V}$, $e \in Exp$,
  $[\![x := e]\!]^\sharp \in Env^\sharp \to Env^\sharp$ an approximation of $[\![x := e]\!]$
- For all $t \in Test$,
  $[\![t]\!]^\sharp \in Env^\sharp \to Env^\sharp$ an approximation of $[\![t]\!]$
- A concretisation $\gamma \in Env^\sharp \to \mathcal{P}(Env)$ that explains which property $\gamma(x^\sharp) \in \mathcal{P}(Env)$ is represented by each abstract element $x^\sharp \in Env^\sharp$.

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	$\mathbb{Z}$

$\boldsymbol{Env}^\sharp \stackrel{\text{def}}{=} \mathbb{V} \to \boldsymbol{Sign}$: a sign is associated to each variable.

An abstraction by signs : example

$$
\begin{array}{rcl}
X_0^\sharp & = & \top_{Env}^\sharp \\
X_1^\sharp & = & [\![x := ?]\!]^\sharp\,(X_0^\sharp) \\
X_2^\sharp & = & [\![x < 0]\!]^\sharp\,(X_1^\sharp) \sqcup^\sharp X_4^\sharp \\
X_3^\sharp & = & [\![x < 0]\!]^\sharp\,(X_2^\sharp) \\
X_4^\sharp & = & [\![x := x + 1]\!]^\sharp\,(X_3^\sharp) \\
X_5^\sharp & = & [\![x \geqslant 0]\!]^\sharp\,(X_2^\sharp) \\
X_6^\sharp & = & [\![y := x]\!]^\sharp\,(X_5^\sharp) \\
X_7^\sharp & = & [\![x \geqslant 0]\!]^\sharp\,(X_1^\sharp) \\
X_8^\sharp & = & [\![y := 0]\!]^\sharp\,(X_7^\sharp) \\
X_9^\sharp & = & X_6^\sharp \sqcup^\sharp X_8^\sharp
\end{array}
\quad \xrightarrow[\text{simplifies into}]{\text{which}} \quad
\begin{array}{rcl}
X_0^\sharp & = & [x : \top;\ y : \top] \\
X_1^\sharp & = & X_0^\sharp[x \mapsto \top] \\
X_2^\sharp & = & X_1^\sharp[x \mapsto -] \sqcup^\sharp X_4^\sharp \\
X_3^\sharp & = & X_2^\sharp[x \mapsto -] \\
X_4^\sharp & = & X_3^\sharp[x \mapsto \mathrm{succ}^\sharp(X_3^\sharp(x))] \\
X_5^\sharp & = & X_2^\sharp[x \mapsto +_0] \\
X_6^\sharp & = & X_5^\sharp[y \mapsto X_5^\sharp(x)] \\
X_7^\sharp & = & X_1^\sharp[x \mapsto +_0] \\
X_8^\sharp & = & X_7^\sharp[y \mapsto 0] \\
X_9^\sharp & = & X_6^\sharp \sqcup^\sharp X_8^\sharp
\end{array}
$$

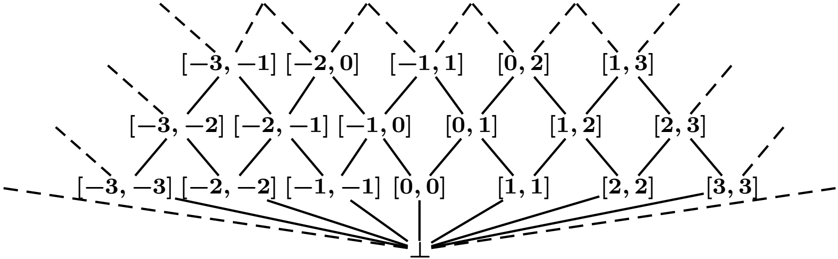
with

$$
\begin{array}{rcl}
\mathrm{succ}^\sharp(\bot) & = & \bot \\
\mathrm{succ}^\sharp(-) & = & -_0 \\
\mathrm{succ}^\sharp(0) & = & \mathrm{succ}^\sharp(+) = \mathrm{succ}^\sharp(+_0) = + \\
\mathrm{succ}^\sharp(-_0) & = & \mathrm{succ}^\sharp(\top) = \top
\end{array}
$$

Abstraction by intervals

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

with $\overline{\mathbb{Z}} = \mathbb{Z} \cup \{-\infty, +\infty\}$.
$\bot$ represents $\emptyset$ and $[a,b]$ the property $\{z \mid a \leqslant z \leqslant b\}$.



$\boldsymbol{Env}^\sharp \stackrel{\text{def}}{=} \mathbb{V} \to \boldsymbol{Int}$: an interval is associated to each variable.

Abstraction by intervals : example

$$
\begin{array}{lcl}
X_0^\sharp & = & \top_{Env}^\sharp \\
X_1^\sharp & = & [\![x := ?]\!]^\sharp\,(X_0^\sharp) \\
X_2^\sharp & = & [\![x < 0]\!]^\sharp\,(X_1^\sharp) \sqcup^\sharp X_4^\sharp \\
X_3^\sharp & = & [\![x < 0]\!]^\sharp\,(X_2^\sharp) \\
X_4^\sharp & = & [\![x := x+1]\!]^\sharp\,(X_3^\sharp) \\
X_5^\sharp & = & [\![x \geqslant 0]\!]^\sharp\,(X_2^\sharp) \\
X_6^\sharp & = & [\![y := x]\!]^\sharp\,(X_5^\sharp) \\
X_7^\sharp & = & [\![x \geqslant 0]\!]^\sharp\,(X_1^\sharp) \\
X_8^\sharp & = & [\![y := 0]\!]^\sharp\,(X_7^\sharp) \\
X_9^\sharp & = & X_6^\sharp \sqcup^\sharp X_8^\sharp
\end{array}
$$

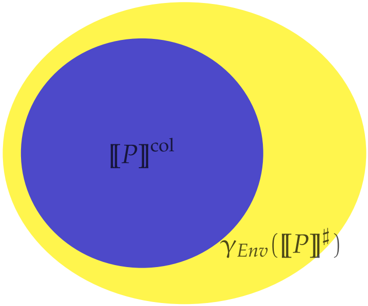
$$
\begin{array}{lcl}
X_0^\sharp & = & [x : [-\infty,+\infty];\ y : [-\infty,+\infty]] \\
X_1^\sharp & = & X_0^\sharp[x \mapsto [-\infty,+\infty]] \\
X_2^\sharp & = & X_1^\sharp[x \mapsto X_1^\sharp(x) \sqcap^\sharp [-\infty,-1]] \sqcup^\sharp X_4^\sharp \\
X_3^\sharp & = & X_2^\sharp[x \mapsto X_2^\sharp(x) \sqcap^\sharp [-\infty,-1]] \\
X_4^\sharp & = & X_3^\sharp[x \mapsto \mathrm{succ}^\sharp(X_3^\sharp(x))] \\
X_5^\sharp & = & X_2^\sharp[x \mapsto X_2^\sharp(x) \sqcap^\sharp [0,+\infty]] \\
X_6^\sharp & = & X_5^\sharp[y \mapsto X_5^\sharp(x)] \\
X_7^\sharp & = & X_1^\sharp[x \mapsto X_1^\sharp(x) \sqcap^\sharp [0,+\infty]] \\
X_8^\sharp & = & X_7^\sharp[y \mapsto [0,0]] \\
X_9^\sharp & = & X_6^\sharp \sqcup^\sharp X_8^\sharp
\end{array}
$$

with

$$
\begin{array}{rcl}
\mathrm{succ}^\sharp(\bot) & = & \bot \\
\mathrm{succ}^\sharp([a,b]) & = & [a+1, b+1]
\end{array}
$$

Outline

Soundness criterion



Given an environment concretisation function $\gamma_{Env} \in Env^\sharp \to \mathcal{P}(Env)$, we want to compute an abstract semantics $[\![P]\!]^\sharp \in \mathbb{P} \to Env^\sharp$ that is a conservative approximation of $[\![P]\!]^{\text{col}}$.

$$\forall k \in \mathbb{P},\ [\![P]\!]^{\text{col}}(k) \subseteq \gamma([\![P]\!]^\sharp(k))$$

This leads to a sound over-approximation of $[\![P]\!]$ since $[\![P]\!]$ and $[\![P]\!]^{\text{col}}$ are equivalents.

$$[\![P]\!] = \{\ (k, \rho) \mid \rho \in [\![p]\!]^{\text{col}}(k)\ \}$$

Function approximation

When some computations in the concrete world are uncomputable or too costly, the abstract world can be used to execute a simplified version of these computations.

- the abstract computation must always give a conservative answer w.r.t. the concrete computation

Let $f \in \mathcal{A} \to \mathcal{A}$ in the concrete world and $f^\sharp \in \mathcal{A}^\sharp \to \mathcal{A}^\sharp$ which correctly approximates each concrete computation.

$$
\begin{array}{ccc}
\mathcal{A} & \xrightarrow{f} & \mathcal{A} \\
\uparrow\gamma & & \uparrow\gamma \\
\mathcal{A}^\sharp & \xrightarrow{f^\sharp} & \mathcal{A}^\sharp
\end{array}
$$

Correctness criterion : $f \circ \gamma \sqsubseteq \gamma \circ f^\sharp$

Fixpoint transfert

Theorem

Given a monotone concretisation between two complete lattices $\left(\mathcal{A}^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp, \bigsqcap^\sharp\right) \to (\mathcal{A}, \sqsubseteq, \bigsqcup, \bigsqcap)$*, a function* $f^\sharp \in \mathcal{A}^\sharp \to \mathcal{A}^\sharp$ *and a monotone function* $f \in \mathcal{A} \to \mathcal{A}$ *which verify* $f \circ \gamma \sqsubseteq \gamma \circ f^\sharp$*, we have*

$$\mathrm{lfp}(f) \sqsubseteq \gamma(\mathrm{lfp}(f^\sharp))$$

It means it is generally sound to mimic fixpoint computation in the abstract.

Environment abstraction : sufficient elements

Thanks to the previous theorem, it is sufficient to design an abstraction domain $Env^\sharp$ with a correct approximation $[\![i]\!]^\sharp$ of $[\![i]\!]$ for all instructions i.

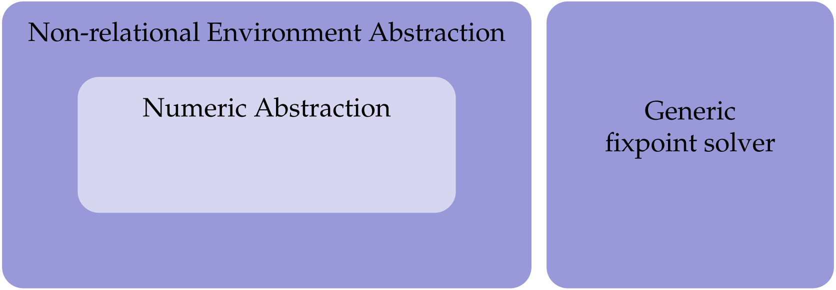
$$\forall \rho^\sharp \in Env^\sharp,\ [\![i]\!]\left(\gamma_{Env}(\rho^\sharp)\right) \subseteq \gamma_{Env}\left([\![i]\!]^\sharp\left(\rho^\sharp\right)\right)$$

And $[\![P]\!]^\sharp$ is defined as the least fixpoint of the system :

$$\forall k \in labels(P),\ X_k^\sharp = {X^\sharp}_k^{\text{init}} \sqcup^\sharp \bigsqcup\nolimits^\sharp_{(k',i,k)\in P} [\![i]\!]^\sharp\left({X^\sharp}_{k'}\right)$$

with ${X^\sharp}_k^{\text{init}} = \begin{cases} \top_{Env} & \text{if } k = k_{\text{init}} \\ \emptyset & \text{otherwise} \end{cases}$

A Generic Abstract Interpreter



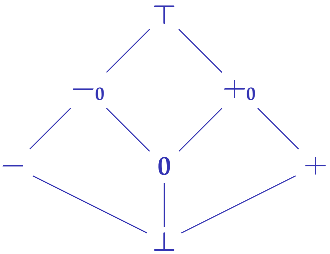
Non-relational environment abstraction

We start with the description of a non-relational abstraction : each variable is abstracted independently.

$$
\begin{array}{rcl}
Env^\sharp & \stackrel{\text{def}}{=} & \mathbb{V} \rightarrow \mathrm{Num}^\sharp \\
\forall \rho_1^\sharp, \rho_2^\sharp \in Env^\sharp,\ \rho_1^\sharp \sqsubseteq_{Env}^\sharp \rho_2^\sharp & \stackrel{\text{def}}{=} & \forall x \in \mathbb{V},\ \rho_1^\sharp(x) \sqsubseteq_{\mathrm{Num}}^\sharp \rho_2^\sharp(x) \\
\forall \rho^\sharp \in Env^\sharp,\ \gamma_{Env}(\rho^\sharp) & \stackrel{\text{def}}{=} & \{\ \rho \mid \forall x \in \mathbb{V},\ \rho(x) \in \gamma_{\mathrm{Num}}(\rho^\sharp(x))\ \}
\end{array}
$$

See the end of the lecture for a relational abstraction.

Sign abstraction



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

We will use this abstract domain as runnign example but you should keep in mind this is just an example among other numerical abstract domains.

Construction of $[\![x := ?]\!]^\sharp$

$$[\![x := ?]\!]^\sharp\,(\rho^\sharp) = \rho^\sharp[x \mapsto \top_{\text{Num}}]\,,\ \forall \rho^\sharp \in Env^\sharp$$

with $\top_{\text{Num}} \in \text{Num}^\sharp$ such that $\mathbb{Z} \subseteq \gamma_{\text{Num}}(\top_{\text{Num}})$.

Construction of $[\![x := e]\!]^\sharp$

$$[\![x := e]\!]^\sharp\,(\rho^\sharp) = \rho^\sharp\Big[x \mapsto \mathcal{A}[\![e]\!]^\sharp\,(\rho^\sharp)\Big]\,,\ \forall \rho^\sharp \in Env^\sharp$$

with

$$\forall e \in \mathrm{Expr},\ \mathcal{A}[\![e]\!]^\sharp \in Env^\sharp \to \mathrm{Num}^\sharp$$

a (forward) abstract evaluation of expressions

$$\begin{array}{rcl}
\mathcal{A}[\![n]\!]^\sharp\,(\rho^\sharp) & = & \mathbf{const}^\sharp(n) \\
\mathcal{A}[\![x]\!]^\sharp\,(\rho^\sharp) & = & \rho^\sharp(x) \\
\mathcal{A}[\![e_1\,o\,e_2]\!]^\sharp\,(\rho^\sharp) & = & o^\sharp\left(\mathcal{A}[\![e_1]\!]^\sharp\,(\rho^\sharp), \mathcal{A}[\![e_2]\!]^\sharp\,(\rho^\sharp)\right)
\end{array}$$

Required operators on the numeric abstraction

- $\text{const}^\sharp \in \text{Num} \to \text{Num}^\sharp$ computes an approximation 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

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

- $o^\sharp \in \text{Num}^\sharp \times \text{Num}^\sharp \to \text{Num}^\sharp$ is a correct approximation of the arithmetic operators $o \in \{+, -, \times\}$

$$\begin{array}{l} \forall n_1^\sharp, n_2^\sharp \in \text{Num}^\sharp, \\ \qquad \{\, n_1 \ o\ n_2 \mid n_1 \in \gamma_{\text{Num}}(n_1^\sharp),\ n_2 \in \gamma_{\text{Num}}(n_2^\sharp) \,\} \subseteq \gamma_{\text{Num}}(o^\sharp(n_1^\sharp, n_2^\sharp)) \end{array}$$

Example : sign abstract domain

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

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

Example : sign abstract domain

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

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

Example : sign abstract domain

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

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

Example : sign abstract domain

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

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

Example : sign abstract domain

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

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

Construction of $[\![t]\!]^\sharp$

More difficult, because ideally such a refinement should be possible...

$$[x \mapsto +;\ y \mapsto -_0] \xrightarrow{[\![(0-y)-x>0]\!]^\sharp} [x \mapsto +;\ y \mapsto -]$$

Construction of $[\![t]\!]^\sharp$

$$[\![e_1 \; c \; e_2]\!]^\sharp\,(\rho^\sharp) \quad = \quad \Big([\![e_1]\!]{\downarrow}^\sharp_{\text{expr}}\,(\rho^\sharp, n_1^\sharp) \sqcap^\sharp_{Env} [\![e_2]\!]{\downarrow}^\sharp_{\text{expr}}\,(\rho^\sharp, n_2^\sharp)\Big)$$
$$\text{with } (n_1^\sharp, n_2^\sharp) = [\![c]\!]{\downarrow}^\sharp_{\text{comp}}\,\Big(\mathcal{A}[\![e_1]\!]^\sharp\,(\rho^\sharp), \mathcal{A}[\![e_2]\!]^\sharp\,(\rho^\sharp)\Big)$$

- $[\![c]\!]{\downarrow}^\sharp_{\text{comp}} \in \text{Num}^\sharp \times \text{Num}^\sharp \to \text{Num}^\sharp \times \text{Num}^\sharp$ computes a refinement of two numeric abstract values, knowing that they verify condition c
- $[\![e]\!]{\downarrow}^\sharp_{\text{expr}} \in Env^\sharp \times \text{Num}^\sharp \to Env^\sharp$: $[\![e]\!]{\downarrow}^\sharp_{\text{expr}}\,(\rho^\sharp, n^\sharp)$ 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.

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

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

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

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

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

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

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

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

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

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

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

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

Required operators on the numeric abstraction

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

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

Required operators on the numeric abstraction

$$[\![o]\!]{\downarrow}^{\sharp}_{\mathrm{op}} \in \mathrm{Num}^{\sharp} \times \mathrm{Num}^{\sharp} \times \mathrm{Num}^{\sharp} \rightarrow \mathrm{Num}^{\sharp} \times \mathrm{Num}^{\sharp}$$

$[\![o]\!]{\downarrow}^{\sharp}_{\mathrm{op}}\ (n^{\sharp}, n_1^{\sharp}, n_2^{\sharp})$ 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.

$$\begin{array}{l}
\forall n^{\sharp}, n_1^{\sharp}, n_2^{\sharp} \in \mathrm{Num}^{\sharp}, \\
\quad \left\{ (n_1, n_2) \mid n_1 \in \gamma_{\mathrm{Num}}(n_1^{\sharp}),\ n_2 \in \gamma_{\mathrm{Num}}(n_2^{\sharp}),\ (n_1\ o\ n_2) \in \gamma_{\mathrm{Num}}(n^{\sharp}) \right\} \\
\qquad \subseteq \gamma_{\mathrm{Num}}(m_1^{\sharp}) \times \gamma_{\mathrm{Num}}(m_2^{\sharp}) \\
\qquad\qquad\qquad\qquad \text{with } (m_1^{\sharp}, m_2^{\sharp}) = [\![o]\!]{\downarrow}^{\sharp}_{\mathrm{op}}\ (n^{\sharp}, n_1^{\sharp}, n_2^{\sharp})
\end{array}$$

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

...

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

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

...

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

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

...

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

Ocaml code...

```
module type NumAbstraction =
 sig
  module L : Lattice

  val backTest : comp -> L.t -> L.t -> L.t * L.t

  val semOp : op -> L.t -> L.t -> L.t

  val back_semOp : op -> L.t -> L.t -> L.t -> L.t * L.t

  val const : int -> L.t

  val top : L.t

  val to_string : string -> L.t -> string
 end

module EnvNotRelational = functor (AN:NumAbstraction) ->
 (struct ... end : EnvAbstraction)
```

Outline

Abstraction by intervals

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

Lattice :

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

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

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

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

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

$$\begin{aligned} \perp_{\text{Int}} &\stackrel{\text{def}}{=} \perp \\ \top_{\text{Int}} &\stackrel{\text{def}}{=} [-\infty, +\infty] \end{aligned}$$

$$\begin{aligned} \gamma_{\text{Int}}(\perp) &\stackrel{\text{def}}{=} \emptyset \\ \gamma_{\text{Int}}([a, b]) &\stackrel{\text{def}}{=} \{\, z \in \mathbb{Z} \mid a \leqslant z \text{ and } z \leqslant b \,\} \end{aligned}$$

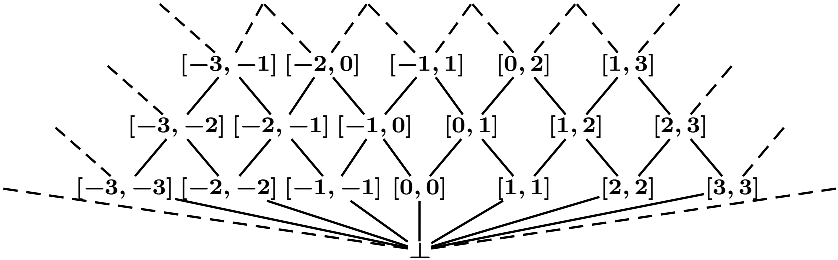
All the other operators are *stricts* : they return $\perp$ if one of their arguments is $\perp$.

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

$$
\begin{array}{rcl}
[\![+]\!]{\downarrow}^\sharp_{\mathrm{op}}([a,b],[c,d],[e,f]) & = & (\rho(\max(c,a-f),\min(d,b-e)), \\
 & & \quad \rho(\max(e,a-d),\min(f,b-c))) \\
[\![-]\!]{\downarrow}^\sharp_{\mathrm{op}}([a,b],[c,d],[e,f]) & = & (\rho(\max(c,a+e),\min(d,b+f)), \\
 & & \rho(\max(e,c-b),\min(f,d-a))) \\
[\![*]\!]{\downarrow}^\sharp_{\mathrm{op}}([a,b],[c,d],[e,f]) & = & ([c,d],[e,f])
\end{array}
$$

$$
\begin{array}{rcl}
[\![=]\!]{\downarrow}^\sharp_{\mathrm{comp}}([a,b],[c,d]) & = & ([a,b] \sqcap_{\mathrm{Int}} [c,d], [a,b] \sqcap_{\mathrm{Int}} [c,d]) \\
[\![<]\!]{\downarrow}^\sharp_{\mathrm{comp}}([a,b],[c,d]) & = & ([a,b] \sqcap_{\mathrm{Int}} [-\infty, d-1], [a+1,+\infty] \sqcap_{\mathrm{Int}} [c,d]) \\
[\![\leqslant]\!]{\downarrow}^\sharp_{\mathrm{comp}}([a,b],[c,d]) & = & ([a,b] \sqcap_{\mathrm{Int}} [-\infty, d], [a,+\infty] \sqcap_{\mathrm{Int}} [c,d]) \\
[\![\neq]\!]{\downarrow}^\sharp_{\mathrm{comp}}([a,b],[c,d]) & = & ?\ \textit{exercise...} \\
\mathrm{const}(n)^\sharp & = & [n,n]
\end{array}
$$

Convergence problem



Such a lattice does not satisfy the ascending chain condition.

Example of infinite increasing chain :

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

Solution : dynamic approximation

- we extrapolate the limit thanks to a widening operator ∇

$$\perp \sqsubset [0,0] \sqsubset [0,1] \sqsubset [0,2] \sqsubset [0,+\infty] = [0,2]\nabla[0,3]$$

Outline

Fixpoint approximation

Lemma

Let $(A, \sqsubseteq, \sqcup, \sqcap)$ *a complete lattice and* f *a monotone operator on* A*. If* a *is a post-fixpoint of* f *(i.e.* $f(a) \sqsubseteq a$*), then* $\mathrm{lfp}(f) \sqsubseteq a$*.*

We may want to compute an over-approximation of $\mathrm{lfp}(f)$ in the following cases :

- The lattice does not satisfies the ascending chain condition, the iteration $\perp, f(\perp), \ldots, f^n(\perp), \ldots$ may never terminates.
- The ascending chain condition is satisfied but the iteration chain is too long to allow an efficient computation.
- Id the underlying lattice is not complete, the limits of the ascending iterations do not necessarily belongs to the abstraction domain.

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

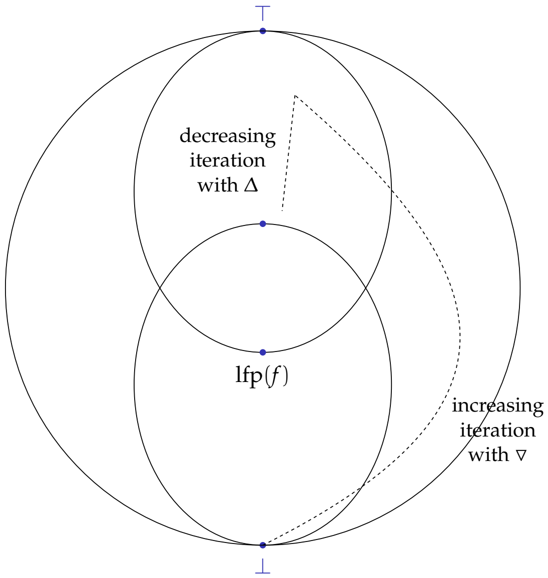
Widening : theorem

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

Scheme


⊤
decreasing
iteration
with Δ
lfp(f)
increasing
iteration
with ∇
⊥

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 :

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

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

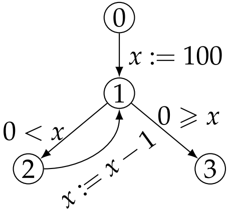
Example

```
x := 100;

while 0 < x {

  x := x − 1;
}
```

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

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

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



Example : without widening

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

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

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

X_1	$\bot$	$\cdots$
X_2	$\bot$	$\cdots$
X_3	$\bot$	$\cdots$

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

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

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

Example : with widening at each nodes of the cfg

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

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

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

X_1	$\bot$
X_2	$\bot$
X_3	$\bot$

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

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

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

Improving fixpoint approximation

Idea : iteration 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 than a (written $\mathrm{gfp}_a(f)$). 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

- $\forall x, x' \in L, x' \sqsubseteq x \Delta x' \sqsubseteq x$
- If $x^0 \sqsupseteq x^1 \sqsupseteq \dots$ 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

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

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

Example : with narrowing at each nodes of the cfg

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

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

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

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

The particular case of an equation system

Consider a system

$$\begin{cases} x_1 & = & f_1(x_1, \ldots, x_n) \\ & \vdots & \\ x_n & = & f_n(x_1, \ldots, x_n) \end{cases}$$

with $f_1, \ldots, f_n$ monotones.

Standard iteration :

$$\begin{array}{rcl} x_1^{i+1} & = & f_1(x_1^i, \ldots, x_n^i) \\ x_2^{i+1} & = & f_2(x_1^i, \ldots, x_n^i) \\ & \vdots & \\ x_n^{i+1} & = & f_n(x_1^i, \ldots, x_n^i) \end{array}$$

Standard iteration with widening :

$$\begin{array}{rcl} x_1^{i+1} & = & x_1^i \nabla f_1(x_1^i, \ldots, x_n^i) \\ x_2^{i+1} & = & x_2^i \nabla f_2(x_1^i, \ldots, x_n^i) \\ & \vdots & \\ x_n^{i+1} & = & x_n^i \nabla f_n(x_1^i, \ldots, x_n^i) \end{array}$$

The particular case of an equation system

$$\left\{\begin{array}{rcl} x_1 & = & f_1(x_1,\ldots,x_n) \\ & \vdots & \\ x_n & = & f_n(x_1,\ldots,x_n) \end{array}\right.$$

It is sufficient (and generally more precise) to use $\triangledown$ for a selection of index W such that each dependence cycle in the system goes through at least one point in W.

$$\forall k = 1..n,\ x_k^{i+1} = \begin{array}{ll} x_k^i \triangledown f_k(x_1^i,\ldots,x_n^i) & \text{if } k \in W \\ f_k(x_1^i,\ldots,x_n^i) & \text{otherwise} \end{array}$$

Chaotic iteration : at each step, we use only one equation, without forgetting one for ever.

Contrary, to what happen in a standard dataflow framework (with monotone functions and ascending chain condition), the iteration strategy may affect a lot the precision of the result. See F. Bourdoncle, *Efficient Chaotic Iteration Strategies with Widenings*, 1993.

Direct iteration on program syntax tree

Is is also possible to directly iterate over the program syntax tree.

$$[\![\texttt{while}\ t\ \texttt{do}\ s]\!]\,(m_0^\sharp) = [\![\neg t]\!]\left(\mathrm{lfp}\left(\lambda m^\sharp.\ m_0^\sharp \sqcup [\![t]\!] \circ [\![s]\!]\,(m^\sharp)\right)\right)$$

or

$$[\![\texttt{while}\ t\ \texttt{do}\ s]\!]\,(m_0^\sharp) = [\![\neg t]\!]\left(\mathrm{lfp}_\triangledown^\sharp\left(\lambda m^\sharp.\ m_0^\sharp \sqcup [\![t]\!] \circ [\![s]\!]\,(m^\sharp)\right)\right)$$

See A. Miné, *Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation*. Found. Trends Program. Lang. 4(3-4) : 120-372 (2017).

Outline

Polyhedral abstract interpretation

Automatic discovery of linear restraints among variables of a program.
P. Cousot and N. Halbwachs. POPL'78.

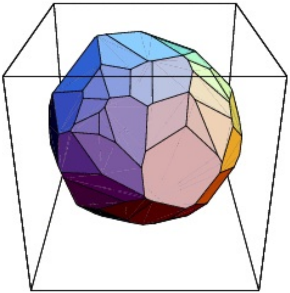
Patrick Cousot

Nicolas Halbwachs

Polyhedral analysis seeks to discover invariant linear equality and inequality relationships among the variables of an imperative program.

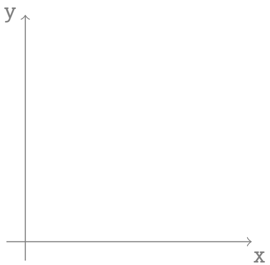
Convex polyhedra

A convex polyhedron can be defined algebraically as the set of solutions of a system of linear inequalities.
Geometrically, it can be defined as a finite intersection of half-spaces.



Polyhedral analysis

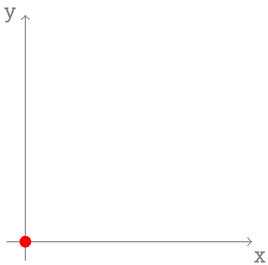
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;


while (x<6) {
  if (?) {

    y = y+2;

  };


  x = x+1;

}
```

Polyhedral analysis

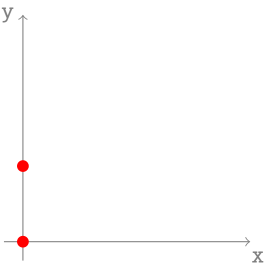
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;
        {x = 0 ∧ y = 0}

while (x<6) {
  if (?) {
        {x = 0 ∧ y = 0}
    y = y+2;

  };


  x = x+1;

}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.

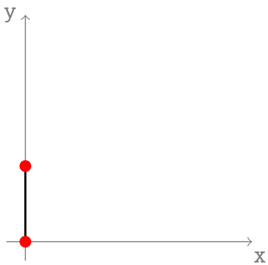


At junction points, we over-approximates union by a convex union.

```
x = 0; y = 0;
        {x = 0 ∧ y = 0}

while (x<6) {
  if (?) {
        {x = 0 ∧ y = 0}
    y = y+2;
        {x = 0 ∧ y = 2}
  };
        {x = 0 ∧ y = 0} ⊎ {x = 0 ∧ y = 2}

  x = x+1;

}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



At junction points, we over-approximates union by a convex union.

```
x = 0; y = 0;
```

$\{x = 0 \wedge y = 0\}$

```
while (x<6) {
  if (?) {
```

$\{x = 0 \wedge y = 0\}$

```
    y = y+2;
```

$\{x = 0 \wedge y = 2\}$

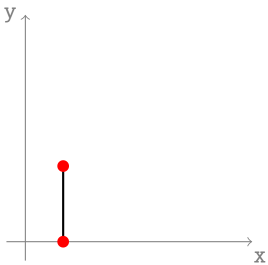
```
  };
```

$\{x = 0 \wedge 0 \leqslant y \leqslant 2\}$

```
  x = x+1;

}
```

Polyhedral analysis

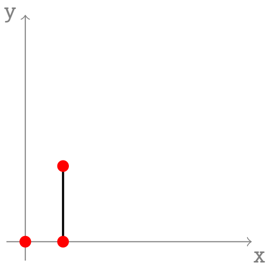
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;
        {x = 0 ∧ y = 0}

while (x<6) {
  if (?) {
        {x = 0 ∧ y = 0}
    y = y+2;
        {x = 0 ∧ y = 2}
  };
        {x = 0 ∧ 0 ⩽ y ⩽ 2}

  x = x+1;
        {x = 1 ∧ 0 ⩽ y ⩽ 2}
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;
```

$\{x = 0 \wedge y = 0\} \uplus \{x = 1 \wedge 0 \leqslant y \leqslant 2\}$

```
while (x<6) {
  if (?) {
```

$\{x = 0 \wedge y = 0\}$

```
    y = y+2;
```

$\{x = 0 \wedge y = 2\}$

```
  };
```

$\{x = 0 \wedge 0 \leqslant y \leqslant 2\}$

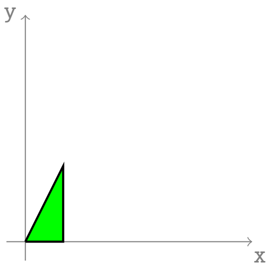
```
  x = x+1;
```

$\{x = 1 \wedge 0 \leqslant y \leqslant 2\}$

```
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;
```

$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$

```
while (x<6) {
  if (?) {
```

$\{x = 0 \wedge y = 0\}$

```
    y = y+2;
```

$\{x = 0 \wedge y = 2\}$

```
  };
```

$\{x = 0 \wedge 0 \leqslant y \leqslant 2\}$

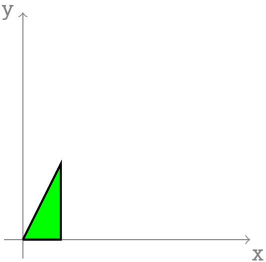
```
  x = x+1;
```

$\{x = 1 \wedge 0 \leqslant y \leqslant 2\}$

```
}
```

Polyhedral analysis

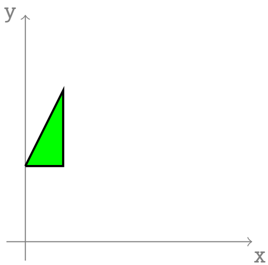
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;
      {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}

while (x<6) {
  if (?) {
      {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}
    y = y+2;
      {x = 0 ∧ y = 2}
  };
      {x = 0 ∧ 0 ⩽ y ⩽ 2}

  x = x+1;
      {x = 1 ∧ 0 ⩽ y ⩽ 2}
}
```

Polyhedral analysis

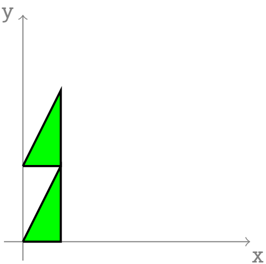
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}

while (x<6) {
  if (?) {
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}
    y = y+2;
        {x ⩽ 1 ∧ 2 ⩽ y ⩽ 2x + 2}
  };
        {x = 0 ∧ 0 ⩽ y ⩽ 2}

  x = x+1;
        {x = 1 ∧ 0 ⩽ y ⩽ 2}
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;
```
$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$

```
while (x<6) {
  if (?) {
```
$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$

```
    y = y+2;
```
$\{x \leqslant 1 \wedge 2 \leqslant y \leqslant 2x + 2\}$

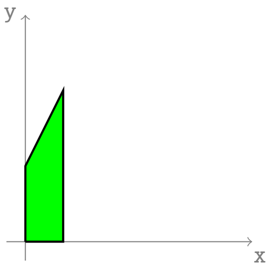
```
  };
```
$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$
$\uplus\{x \leqslant 1 \wedge 2 \leqslant y \leqslant 2x + 2\}$

```
  x = x+1;
```
$\{x = 1 \wedge 0 \leqslant y \leqslant 2\}$

```
}
```

Polyhedral analysis

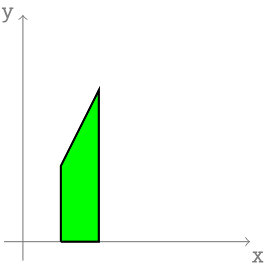
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}

while (x<6) {
  if (?) {
        {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}
    y = y+2;
        {x ⩽ 1 ∧ 2 ⩽ y ⩽ 2x + 2}
  };
        {0 ⩽ x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x + 2}

  x = x+1;
        {x = 1 ∧ 0 ⩽ y ⩽ 2}
}
```

Polyhedral analysis

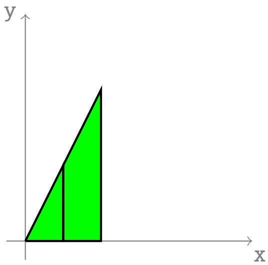
State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;
      {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}

while (x<6) {
  if (?) {
      {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}
    y = y+2;
      {x ⩽ 1 ∧ 2 ⩽ y ⩽ 2x + 2}
  };
      {0 ⩽ x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x + 2}

  x = x+1;
      {1 ⩽ x ⩽ 2 ∧ 0 ⩽ y ⩽ 2x}
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.




```
x = 0; y = 0;
```

$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$

$\nabla \{x \leqslant 2 \wedge 0 \leqslant y \leqslant 2x\}$

```
while (x<6) {
  if (?) {
```

$\{x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x\}$

```
    y = y+2;
```

$\{x \leqslant 1 \wedge 2 \leqslant y \leqslant 2x + 2\}$

```
  };
```

$\{0 \leqslant x \leqslant 1 \wedge 0 \leqslant y \leqslant 2x + 2\}$

```
  x = x+1;
```

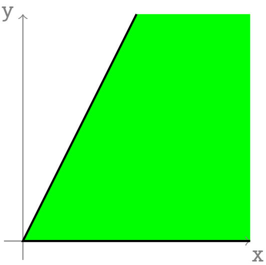
$\{1 \leqslant x \leqslant 2 \wedge 0 \leqslant y \leqslant 2x\}$

```
}
```

At loop headers, we use heuristics (widening) to ensure finite convergence.

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.



At loop headers, we use heuristics (widening) to ensure finite convergence.

```
x = 0; y = 0;
      {0 ⩽ y ⩽ 2x}

while (x<6) {
  if (?) {
      {x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x}
    y = y+2;
      {x ⩽ 1 ∧ 2 ⩽ y ⩽ 2x + 2}
  };
      {0 ⩽ x ⩽ 1 ∧ 0 ⩽ y ⩽ 2x + 2}

  x = x+1;
      {1 ⩽ x ⩽ 2 ∧ 0 ⩽ y ⩽ 2x}
}
```

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.

By propagation we obtain a post-fixpoint

```
x = 0; y = 0;
```

$\{0 \leqslant y \leqslant 2x\}$

```
while (x<6) {
  if (?) {
```

$\{0 \leqslant y \leqslant 2x \ \wedge \ x \leqslant 5\}$

```
    y = y+2;
```

$\{2 \leqslant y \leqslant 2x + 2 \ \wedge \ x \leqslant 5\}$

```
  };
```

$\{0 \leqslant y \leqslant 2x + 2 \ \wedge \ 0 \leqslant x \leqslant 5\}$

```
  x = x+1;
```

$\{0 \leqslant y \leqslant 2x \ \wedge \ 1 \leqslant x \leqslant 6\}$

```
}
```

$\{0 \leqslant y \leqslant 2x \ \wedge \ 6 \leqslant x\}$

Polyhedral analysis

State properties are over-approximated by convex polyhedra in $\mathbb{Q}^2$.

By propagation we obtain a post-fixpoint which is enhanced by downward iteration.

```
x = 0; y = 0;
```

$\{0 \leqslant y \leqslant 2x \wedge x \leqslant 6\}$

```
while (x<6) {
  if (?) {
```

$\{0 \leqslant y \leqslant 2x \wedge x \leqslant 5\}$

```
    y = y+2;
```

$\{2 \leqslant y \leqslant 2x + 2 \wedge x \leqslant 5\}$

```
  };
```

$\{0 \leqslant y \leqslant 2x + 2 \wedge 0 \leqslant x \leqslant 5\}$

```
  x = x+1;
```

$\{0 \leqslant y \leqslant 2x \wedge 1 \leqslant x \leqslant 6\}$

```
}
```

$\{0 \leqslant y \leqslant 2x \wedge 6 = x\}$

Polyhedral analysis

A more complex example.

The analysis accepts to replace some constants by parameters.

```
x = 0; y = A;
```

$\{A \leqslant y \leqslant 2x + A \wedge x \leqslant N\}$

```
while (x<N) {
  if (?) {
```

$\{A \leqslant y \leqslant 2x + A \ \wedge \ x \leqslant N - 1\}$

```
    y = y+2;
```

$\{A + 2 \leqslant y \leqslant 2x + A + 2 \ \wedge \ x \leqslant N - 1\}$

```
  };
```

$\{A \leqslant y \leqslant 2x + A + 2 \ \wedge \ 0 \leqslant x \leqslant N - 1\}$

```
  x = x+1;
```

$\{A \leqslant y \leqslant 2x + A \ \wedge \ 1 \leqslant x \leqslant N\}$

```
}
```

$\{A \leqslant y \leqslant 2x + A \ \wedge \ N = x\}$

The four polyhedra operations

- $\uplus \in \mathbb{P}_n \times \mathbb{P}_n \to \mathbb{P}_n$: convex union
  - over-approximates the concrete union at junction points
- $\cap \in \mathbb{P}_n \times \mathbb{P}_n \to \mathbb{P}_n$: intersection
  - over-approximates the concrete intersection after a conditional intruction
- $[\![\mathtt{x} := e]\!] \in \mathbb{P}_n \to \mathbb{P}_n$: affine transformation
  - over-approximates the assignment of a variable by a linear expression
- $\nabla \in \mathbb{P}_n \times \mathbb{P}_n \to \mathbb{P}_n$: widening
  - ensures (and accelerates) convergence of (post-)fixpoint iteration
  - includes heuristics to infer loop invariants

```
x = 0; y = 0;
```

$$P_0 = [\![\mathtt{y} := \mathtt{0}]\!]\,[\![\mathtt{x} := \mathtt{0}]\!]\,(\mathbb{Q}^2)\ \nabla\ P_4$$

```
while (x<6) {
  if (?) {
```

$$P_1 = P_0 \cap \{x < 6\}$$

```
    y = y+2;
```

$$P_2 = [\![\mathtt{y} := \mathtt{y} + \mathtt{2}]\!]\,(P_1)$$

```
  };
```

$$P_3 = P_1 \uplus P_2$$

```
  x = x+1;
```

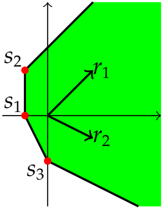
$$P_4 = [\![\mathtt{x} := \mathtt{x} + \mathtt{1}]\!]\,(P_3)$$

```
}
```

$$P_5 = P_0 \cap \{x \geqslant 6\}$$

Library for manipulating polyhedra

- Parma Polyhedra Library[3] (PPL), NewPolka : complex C/C++ libraries
- They rely on the Double Description Method
  - polyhedra are managed using two representations in parallel



  - by set of inequalities

$$P = \left\{ (x,y) \in \mathbb{Q}^2 \;\middle|\; \begin{array}{l} x \geqslant -1 \\ x - y \geqslant -3 \\ 2x + y \geqslant -2 \\ x + 2y \geqslant -4 \end{array} \right\}$$

  - by set of generators

$$P = \left\{ \lambda_1 s_1 + \lambda_2 s_2 + \lambda_3 s_3 + \mu_1 r_1 + \mu_2 r_2 \in \mathbb{Q}^2 \;\middle|\; \begin{array}{l} \lambda_1, \lambda_2, \lambda_3, \mu_1, \mu_2 \in \mathbb{R}^+ \\ \lambda_1 + \lambda_2 + \lambda_3 = 1 \end{array} \right\}$$

  - operations efficiency strongly depends on the chosen representations, so they keep both

3. Previous tutorial on polyhedra partially comes from `http://www.cs.unipr.it/ppl/`

Outline

References (1)

A few articles

- a short formal introduction

  P. Cousot and R. Cousot. Basic Concepts of Abstract Interpretation. `http://www.di.ens.fr/~cousot/COUSOTpapers/WCC04.shtml`

- technical but very complete (the logic programming part is optional) :

  P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. `http://www.di.ens.fr/~cousot/COUSOTpapers/JLP92.shtml`

- application of abstract interpretation theory to verify airbus flight commands

  P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. `http://www.di.ens.fr/~cousot/COUSOTpapers/ESOP05.shtml`

References (2)

On the web :

- informal presentation of AI with nice pictures
  http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html
- a short abstract of various works around AI
  http://www.di.ens.fr/~cousot/AI/
- very complete lecture notes
  http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/

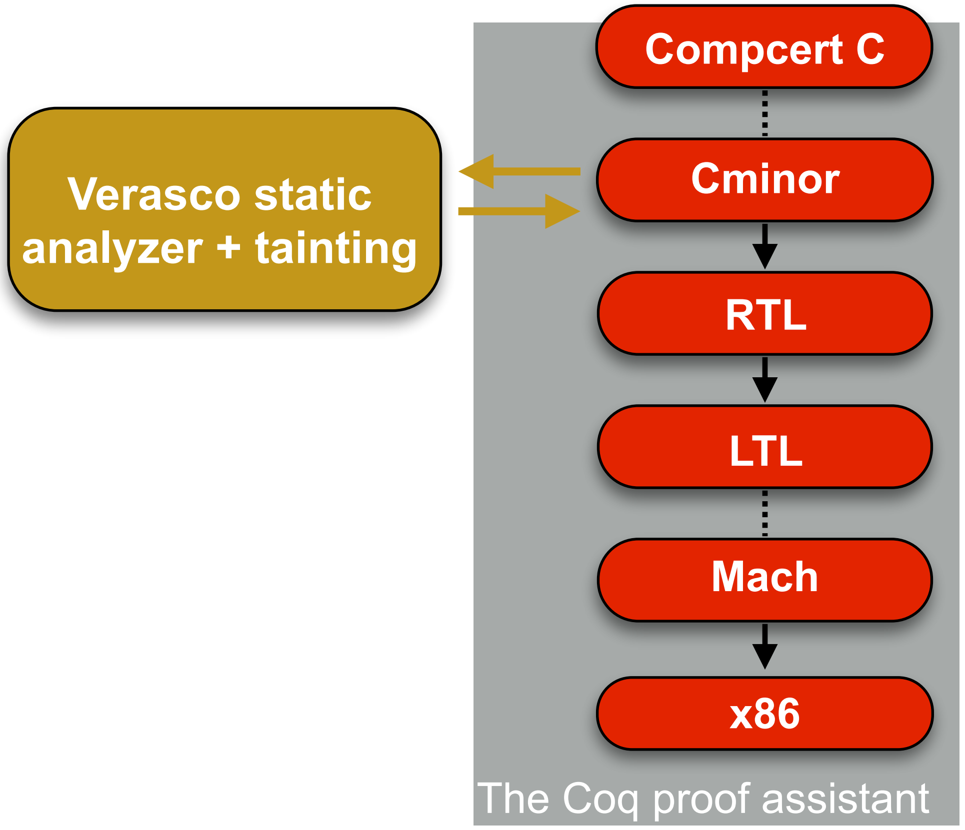
Back to constant-time static analysis

Constant-time analysis at source level

Sandrine Blazy, David Pichardie, Alix Trieu.
Verifying Constant-Time Implementations by Abstract Interpretation.
ESORICS 2017.

We perform static analysis at (almost) C level

- Based on previous work with a value analyser, Verasco
- We mix Verasco memory tracking with fine-grained tainting
- Main difficulty : alias analysis taking into account pointer arithmetic



The Verasco project

INRIA Celtique, Gallium, Antique, Toccata + VERIMAG + Airbus
ANR 2012-2016

http://verasco.imag.fr

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

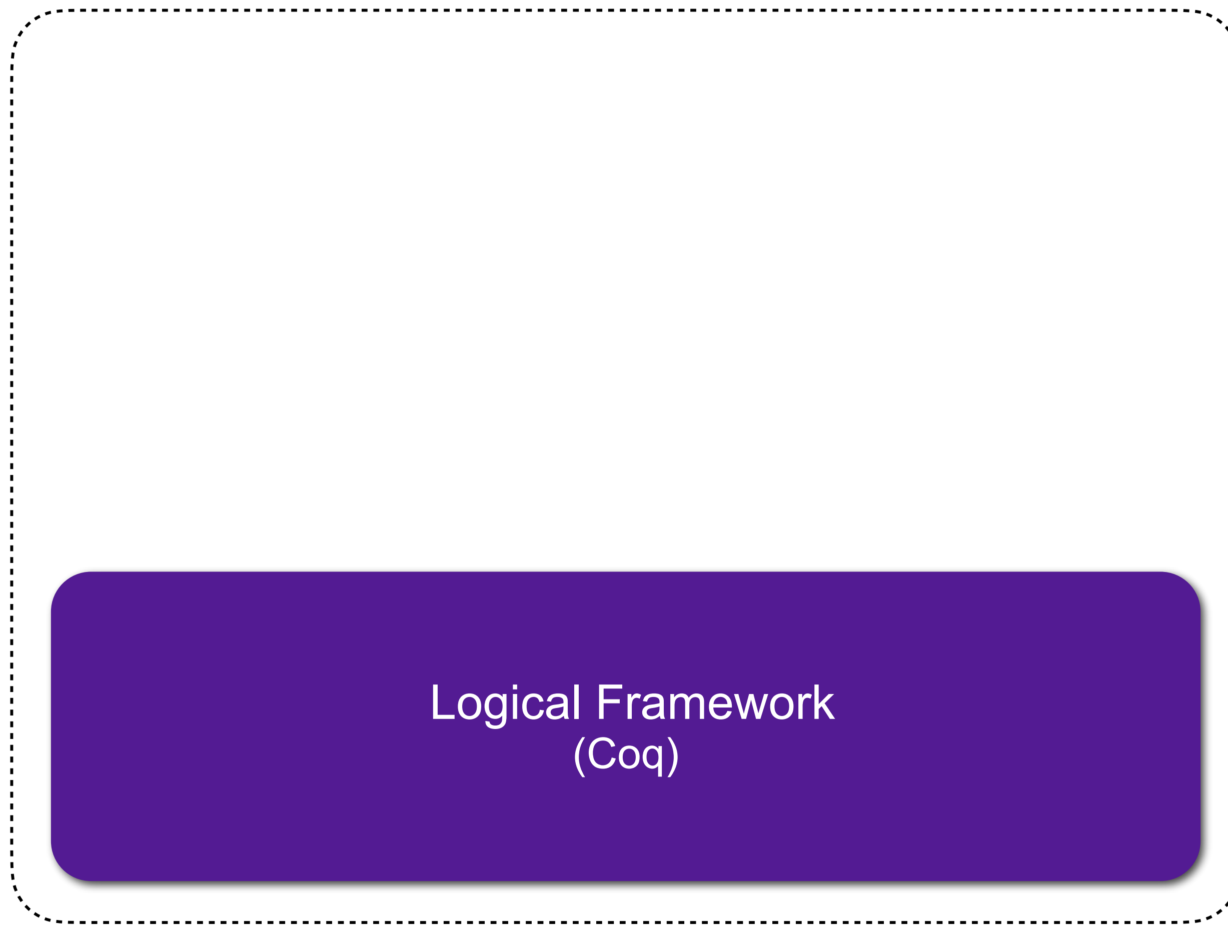
- Language analyzed: the CompCert subset of C
- Nontrivial abstract domains, including relational domains
- Modular architecture inspired from Astrée's
- To prove the absence of undefined behaviors in C source programs

Slogan:

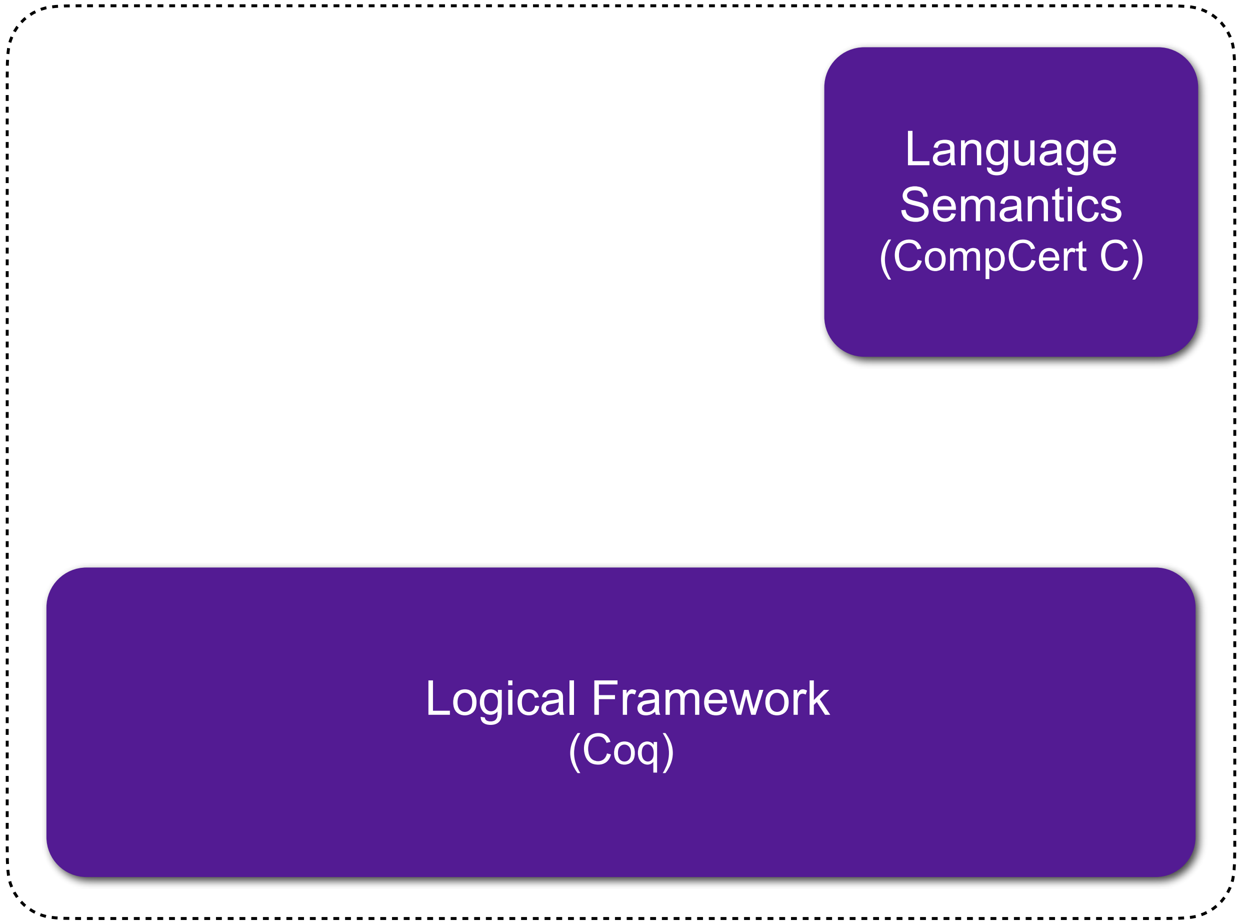
- if « CompCert ≈ 1/10th of GCC but formally verified »,
- likewise « Verasco ≈1/10th of Astrée but formally verified »

Verified Static Analysis

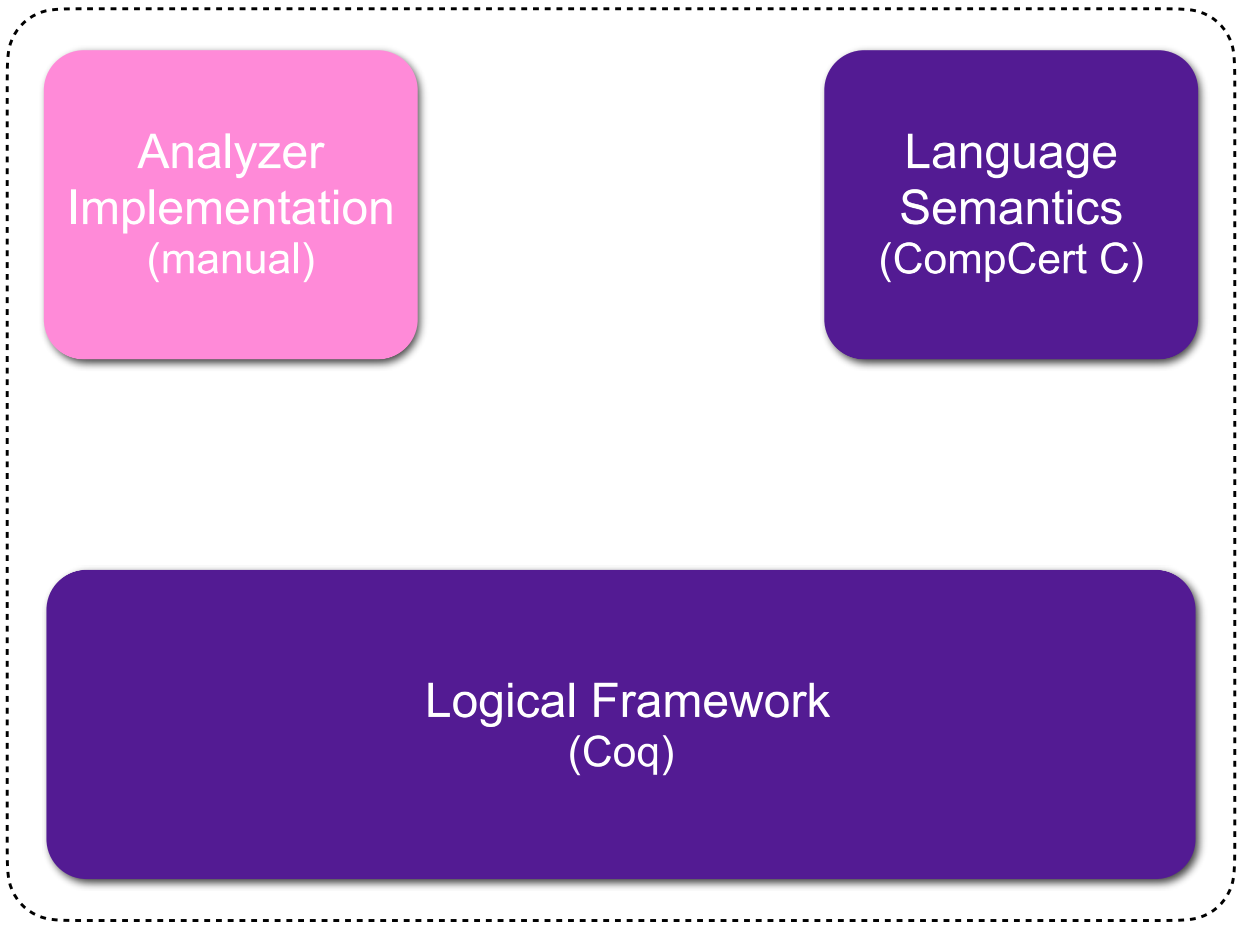
Verified Static Analysis



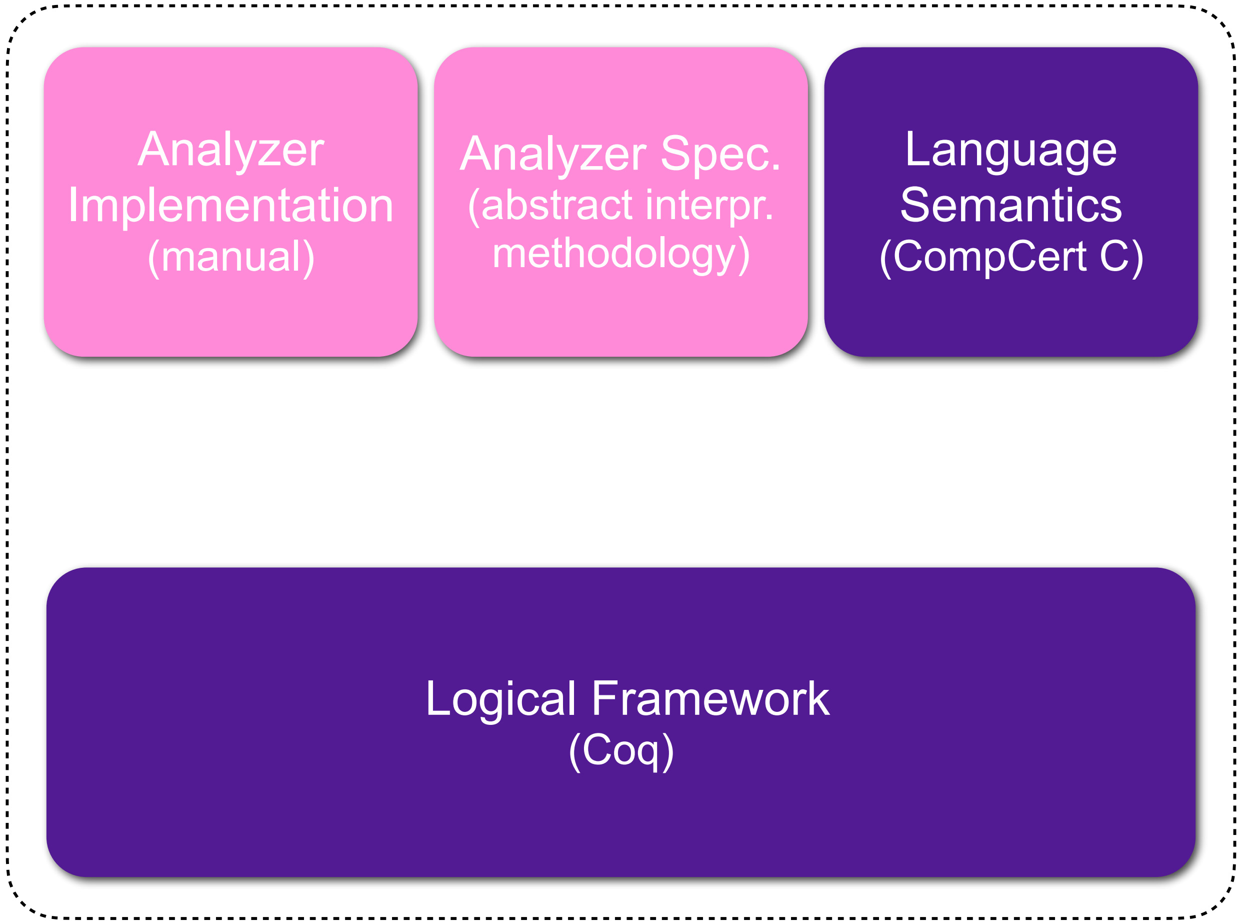
Verified Static Analysis



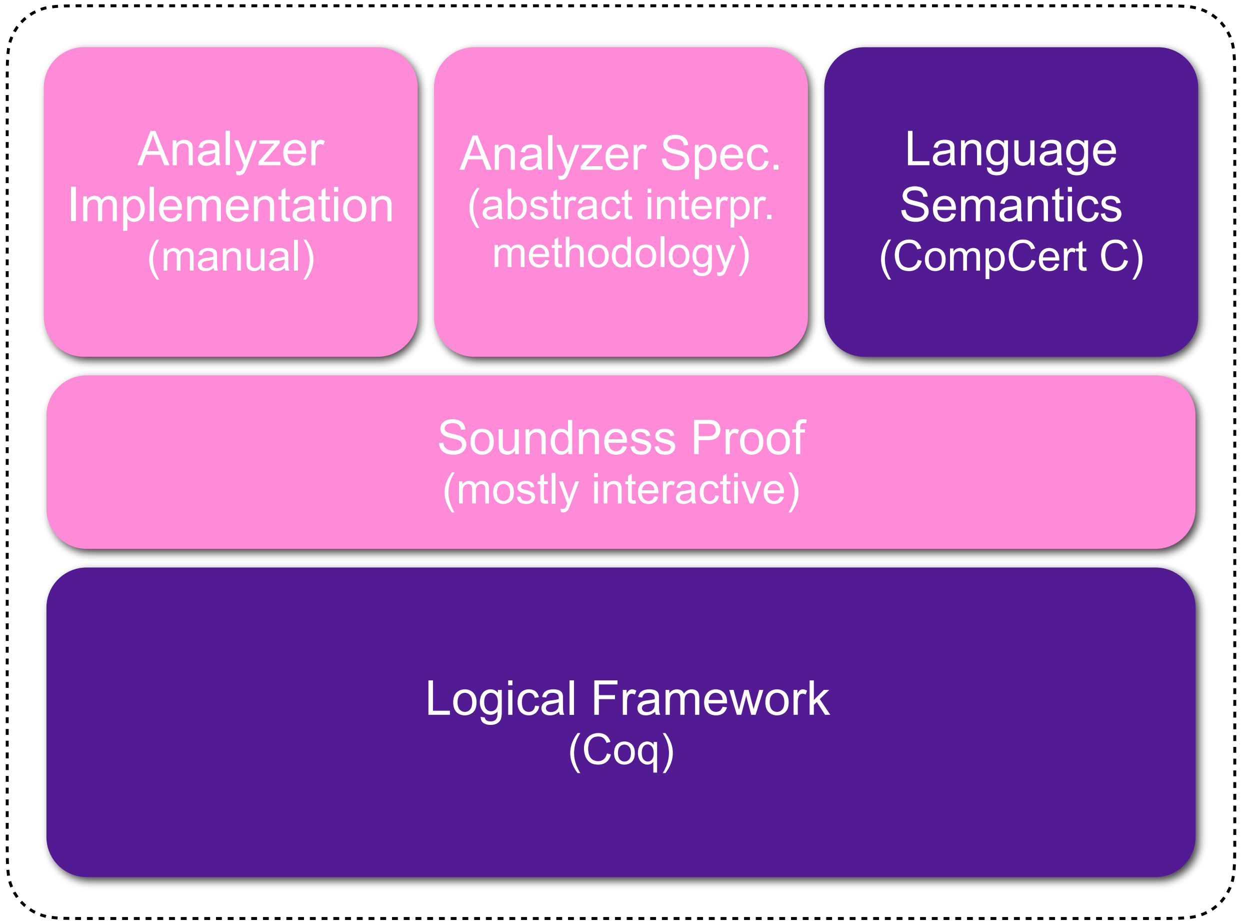
Verified Static Analysis



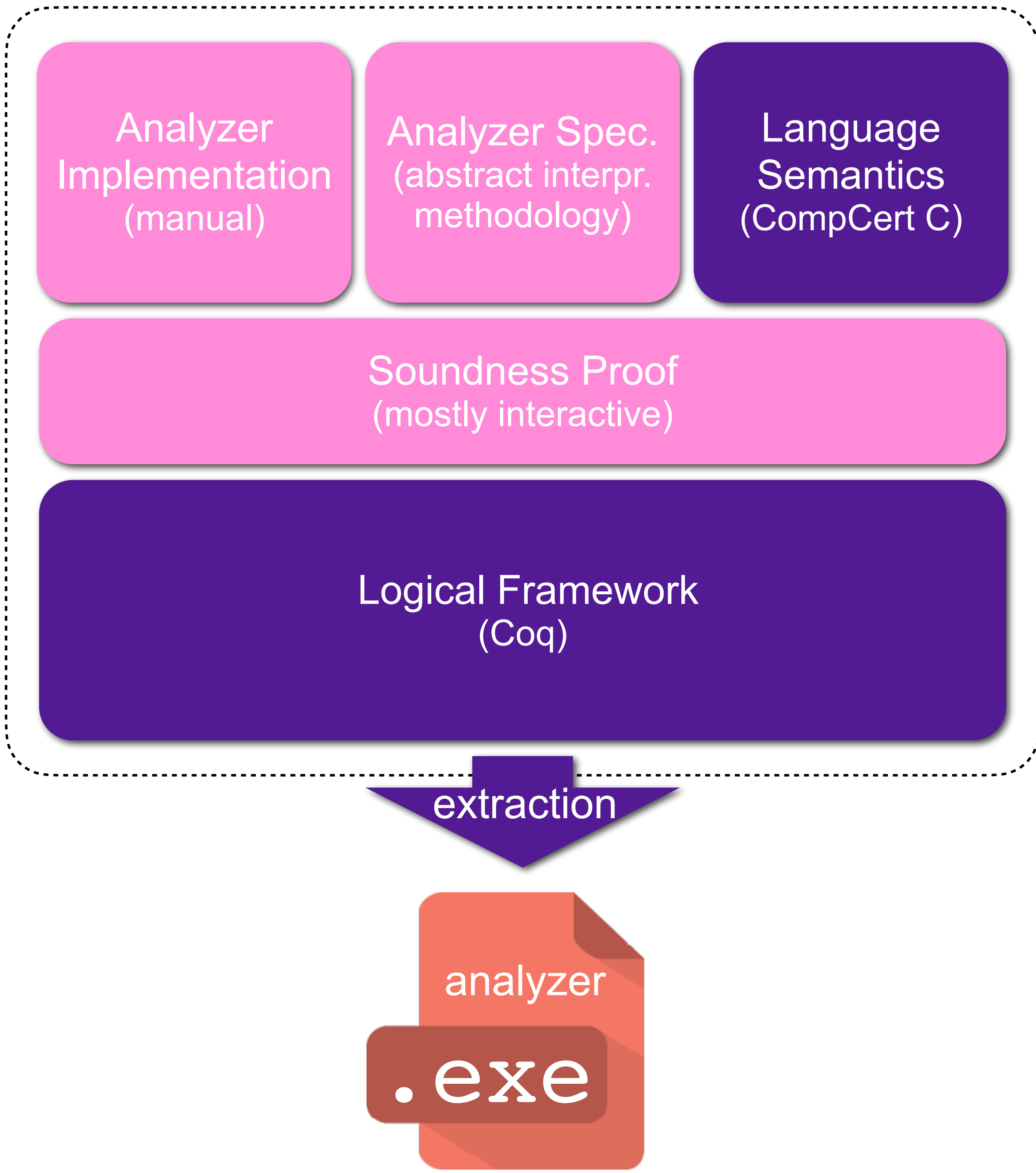
Verified Static Analysis



Verified Static Analysis



Verified Static Analysis

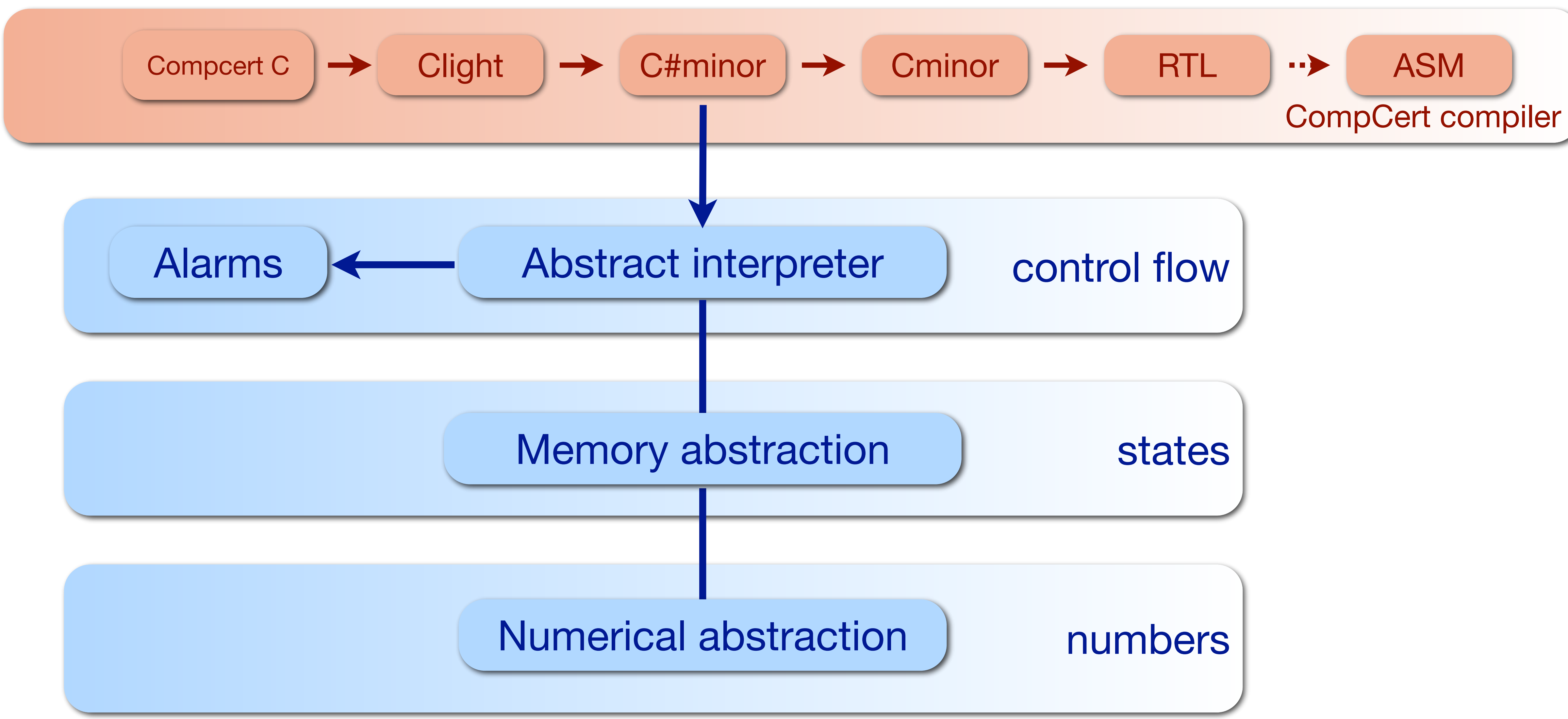


Verasco

A Formally-Verified C Static Analyzer

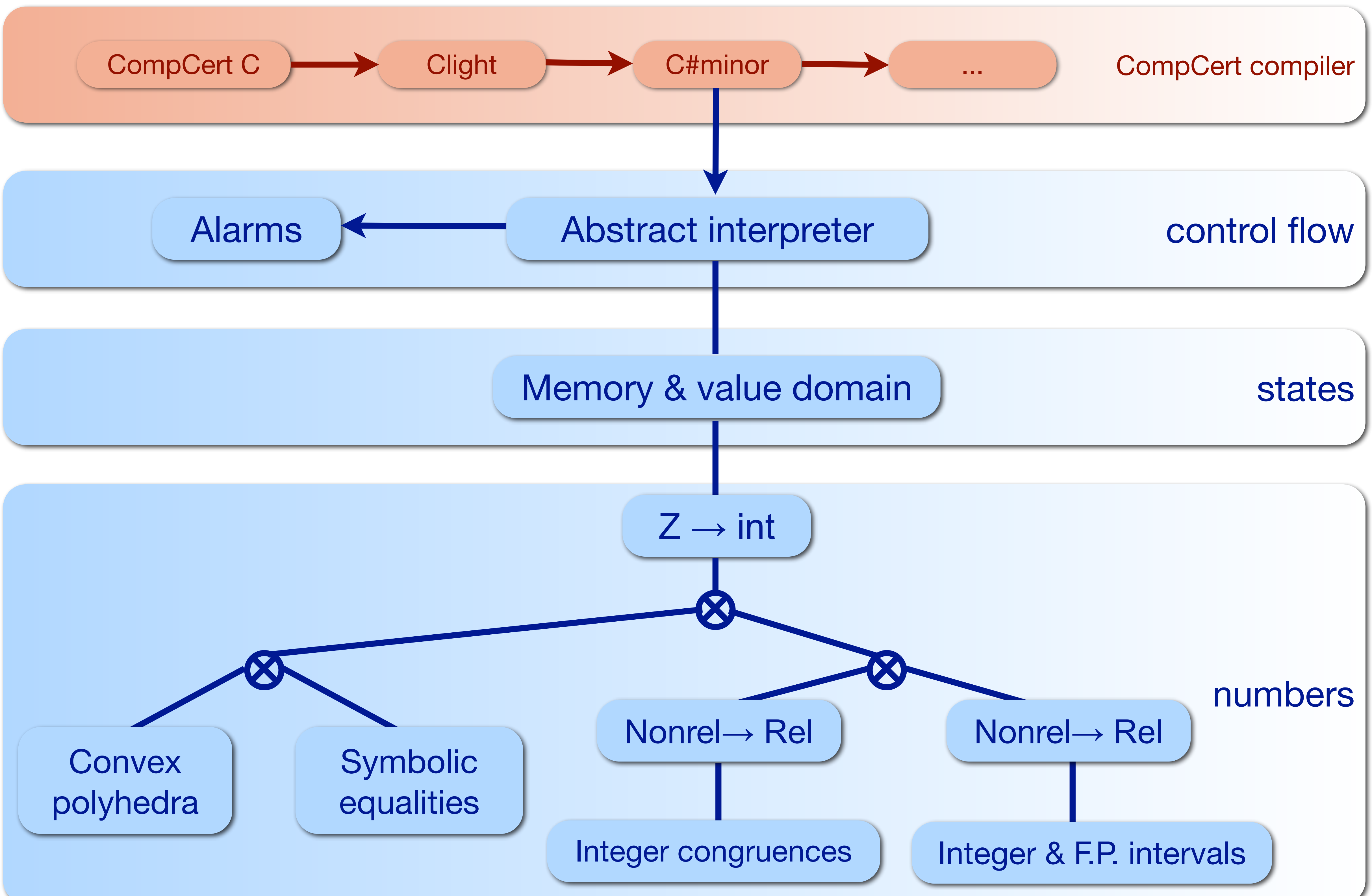
JH. Jourdan, V. Laporte, S. Blazy, X. Leroy, D. Pichardie.
A Formally-Verified C Static Analyzer.
POPL 2015.

S. Blazy, V. Laporte, D. Pichardie.
An Abstract Memory Functor for Verified C Static Analyzers.
ICFP 2016.



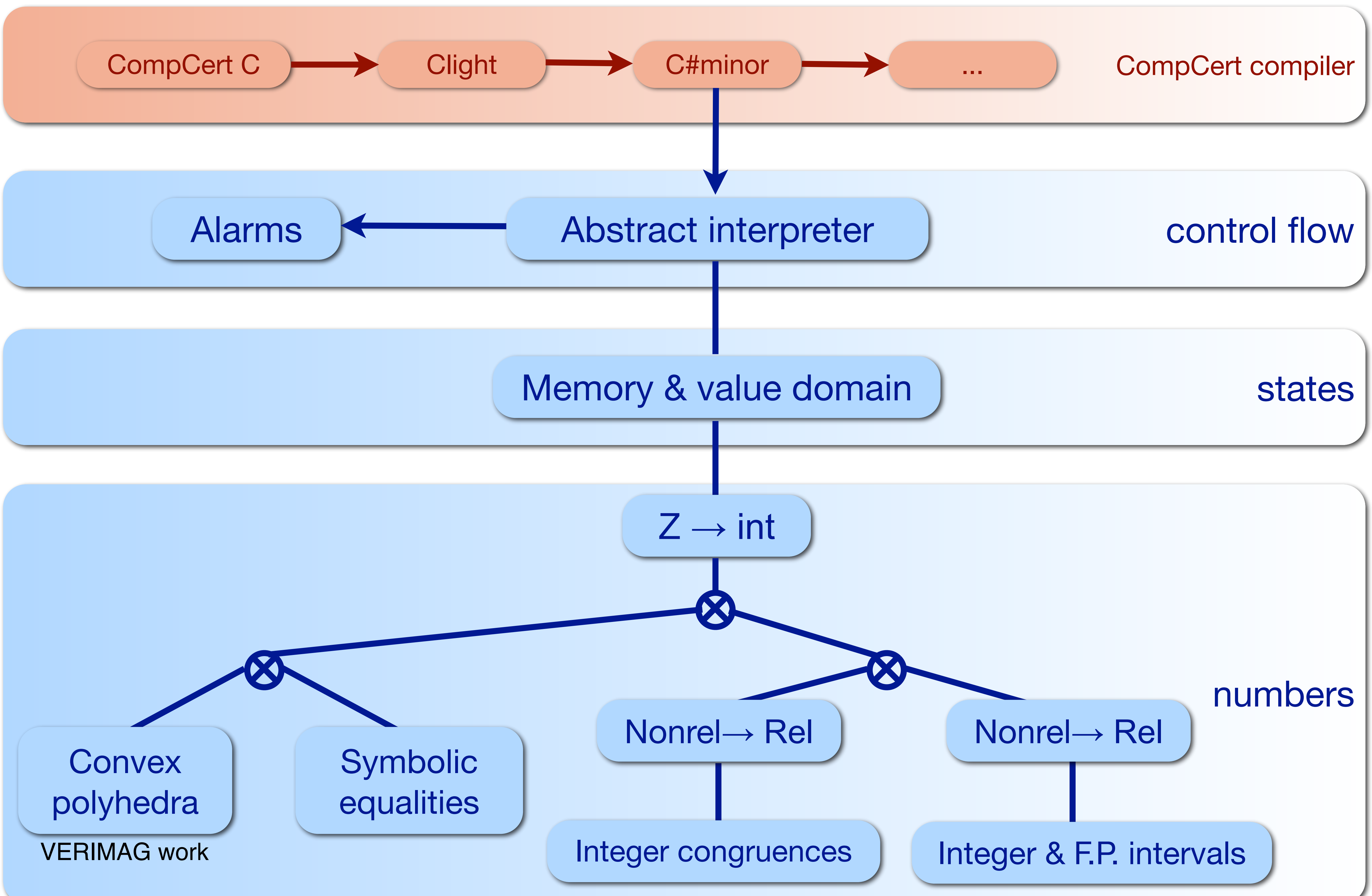
Verasco

Abstract numerical domains



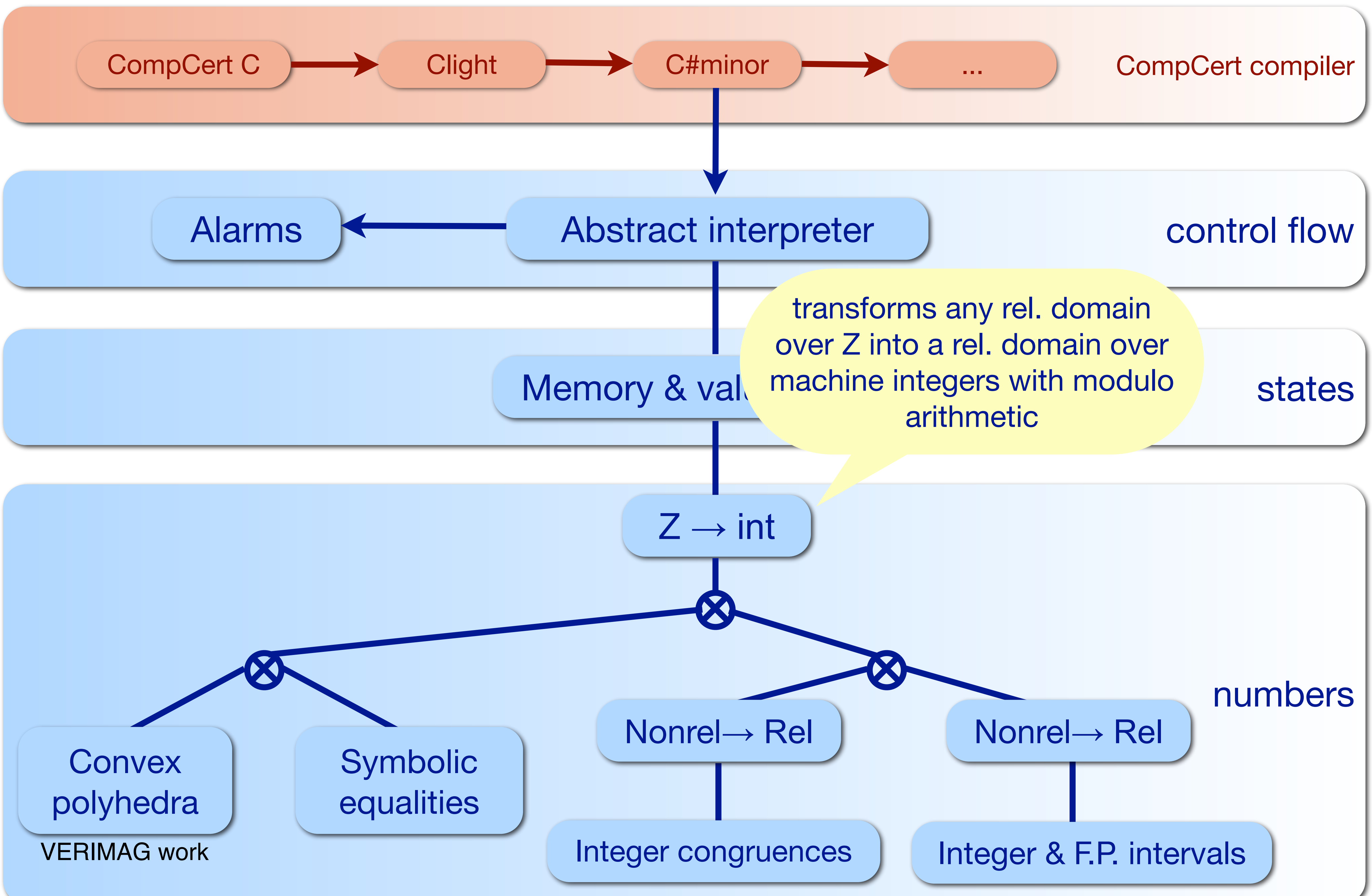
Verasco

Abstract numerical domains



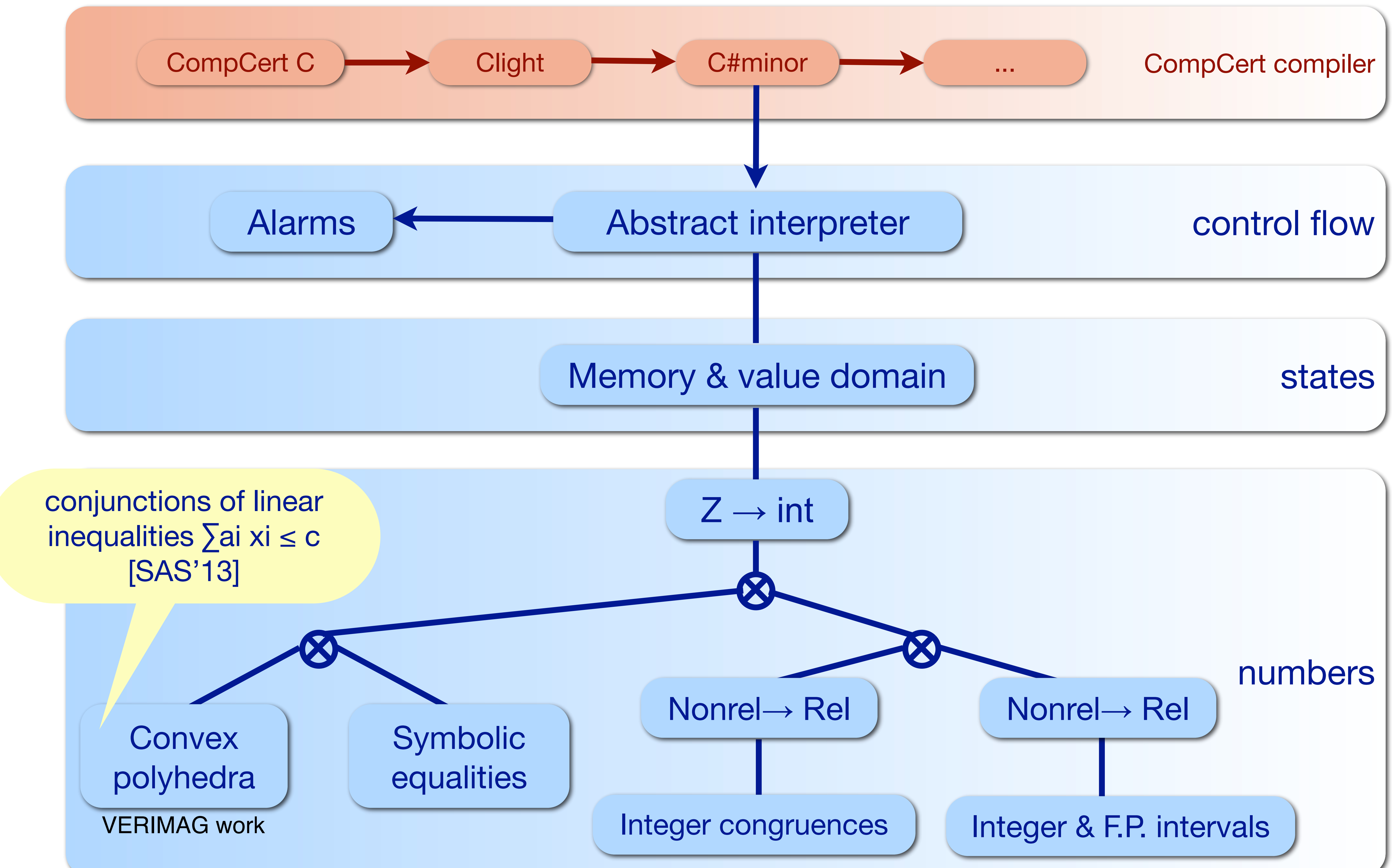
Verasco

Abstract numerical domains



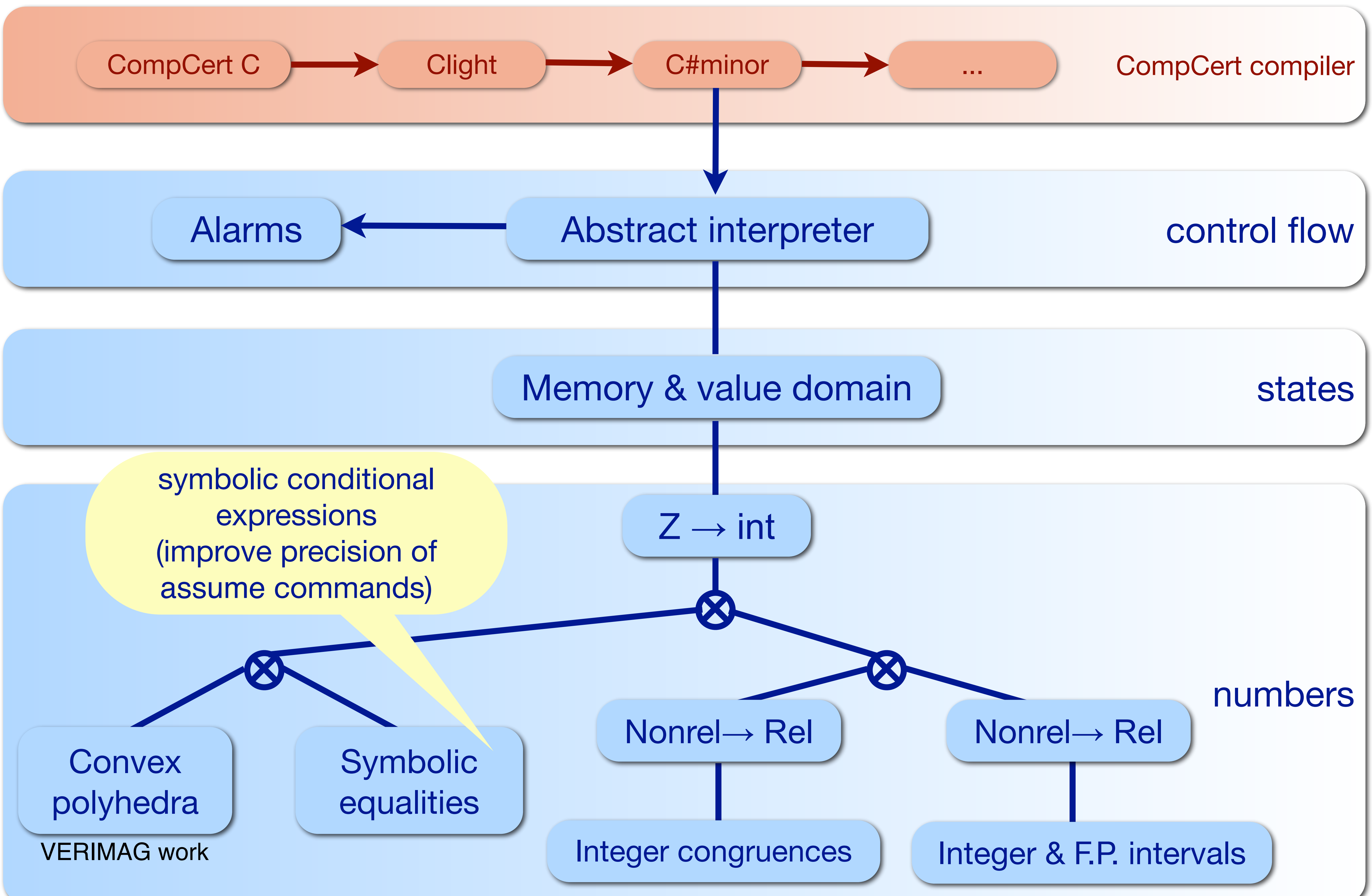
Verasco

Abstract numerical domains



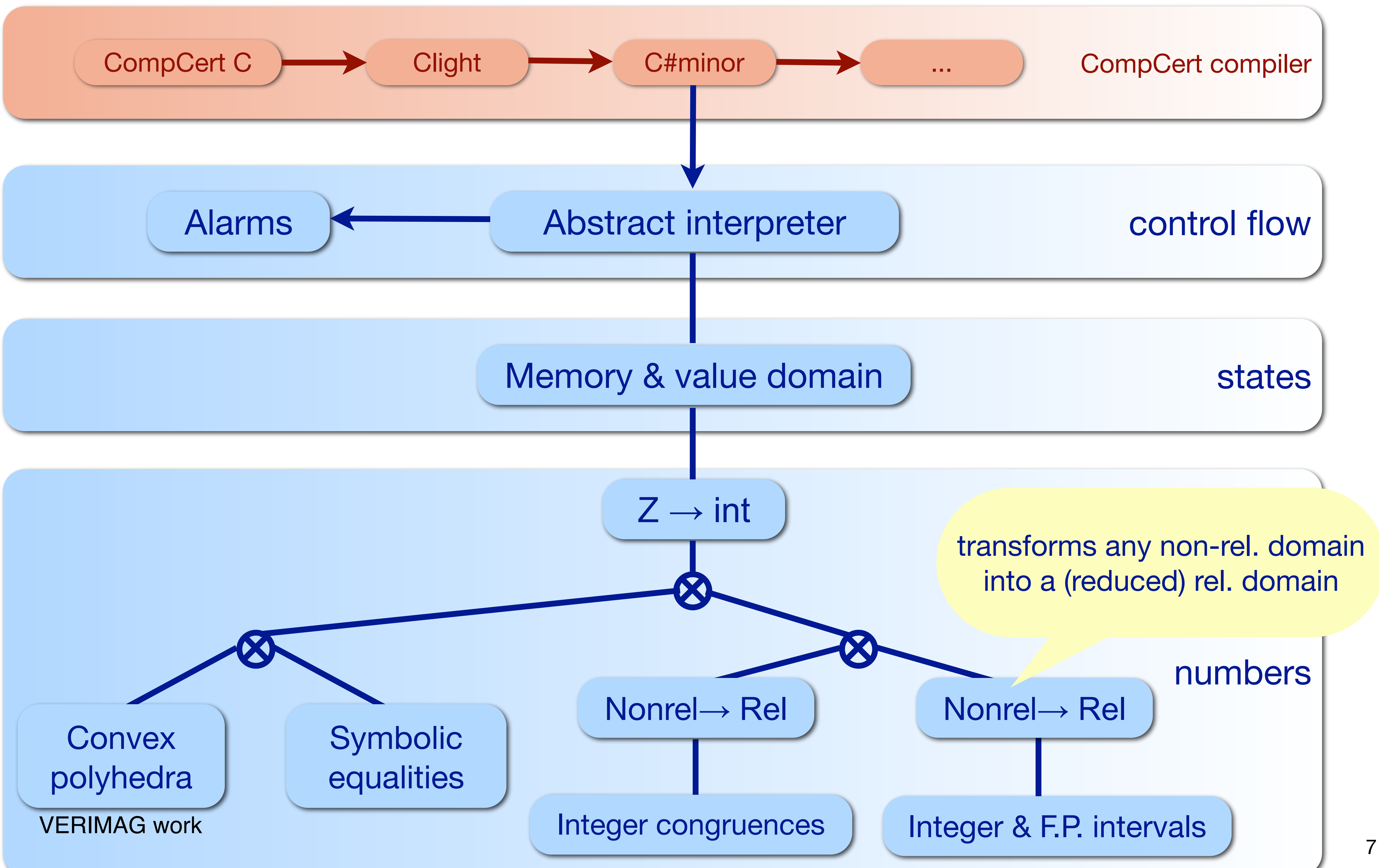
Verasco

Abstract numerical domains



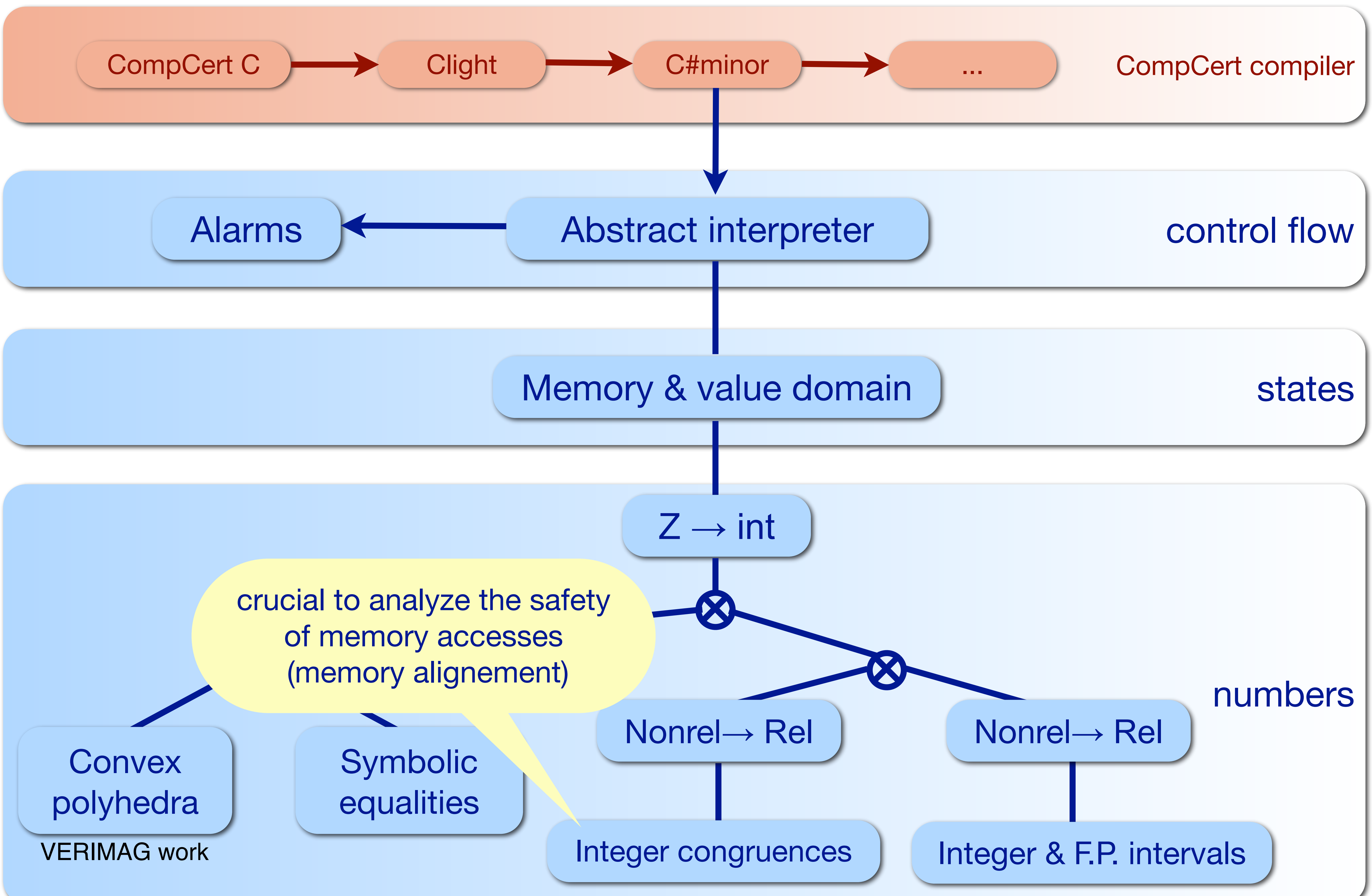
Verasco

Abstract numerical domains



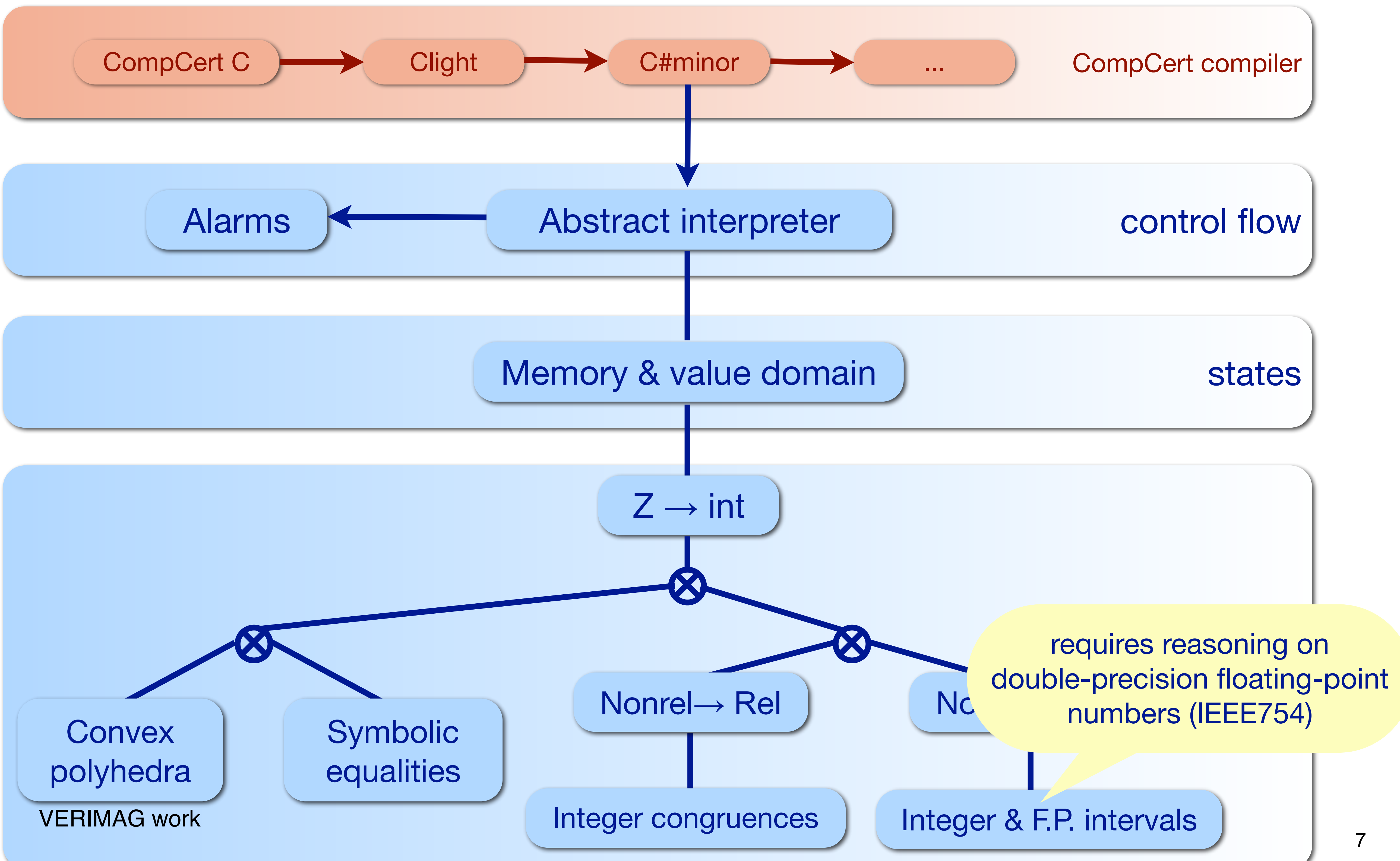
Verasco

Abstract numerical domains



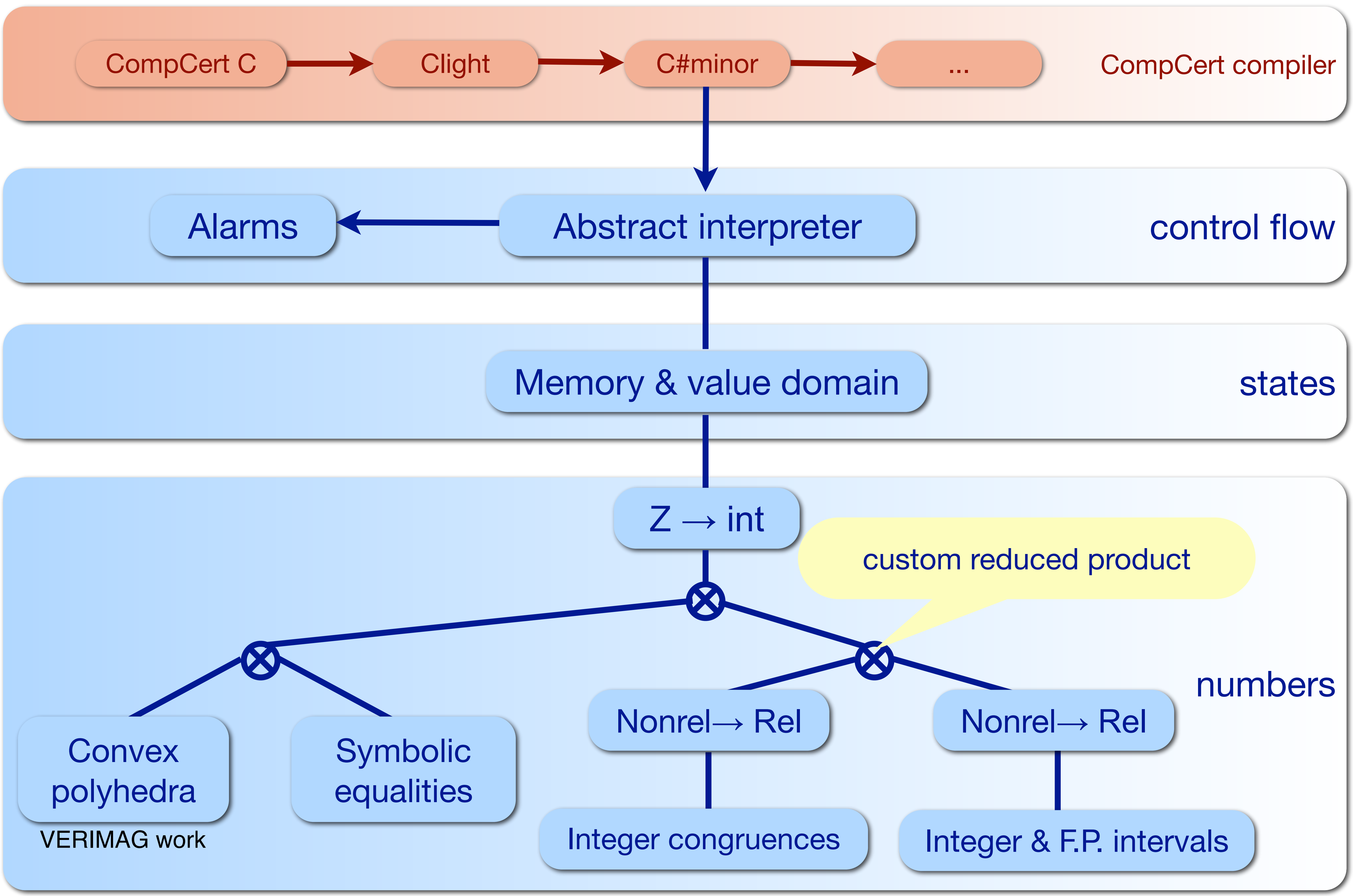
Verasco

Abstract numerical domains



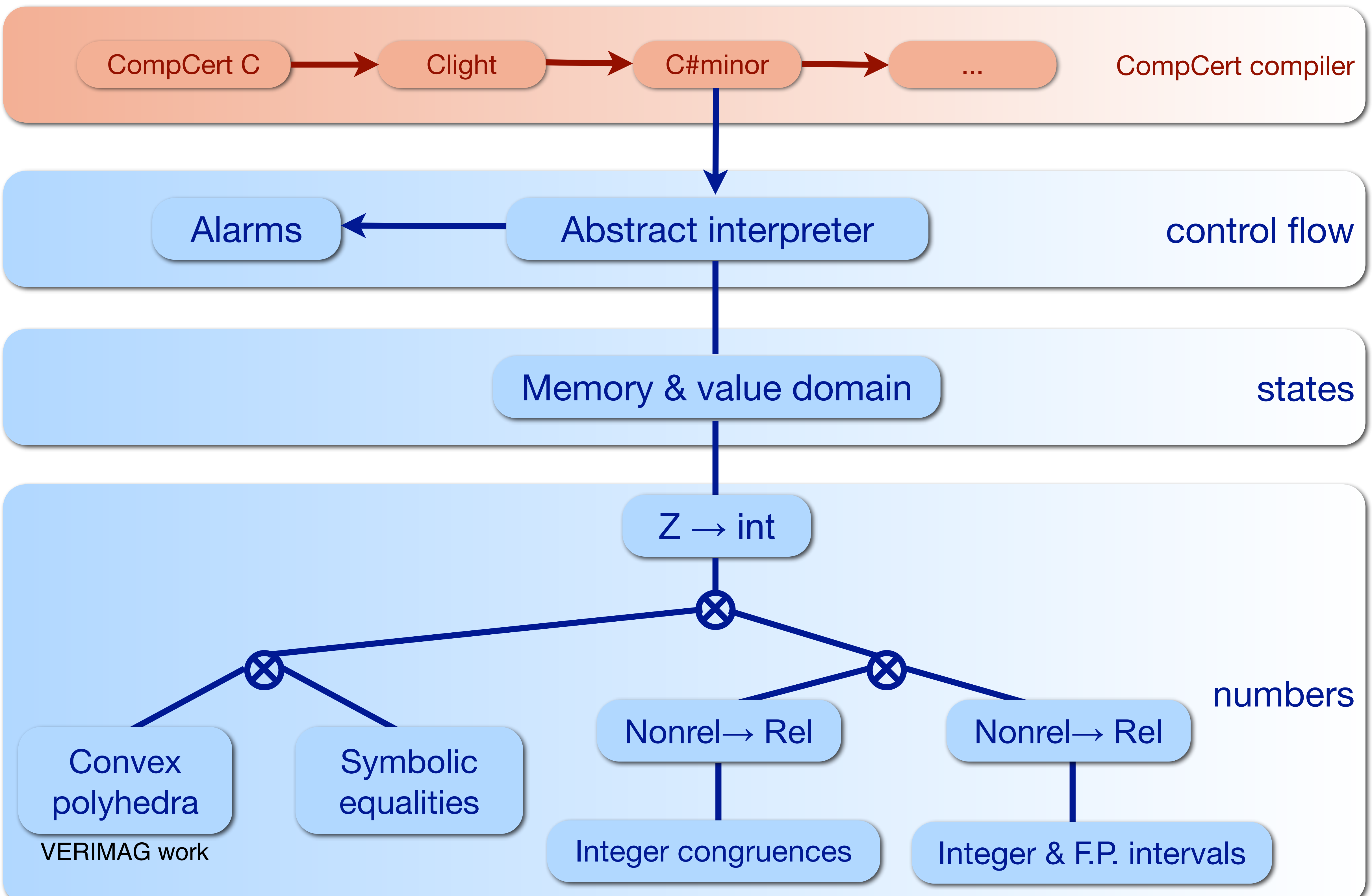
Verasco

Abstract numerical domains



Verasco

Abstract numerical domains



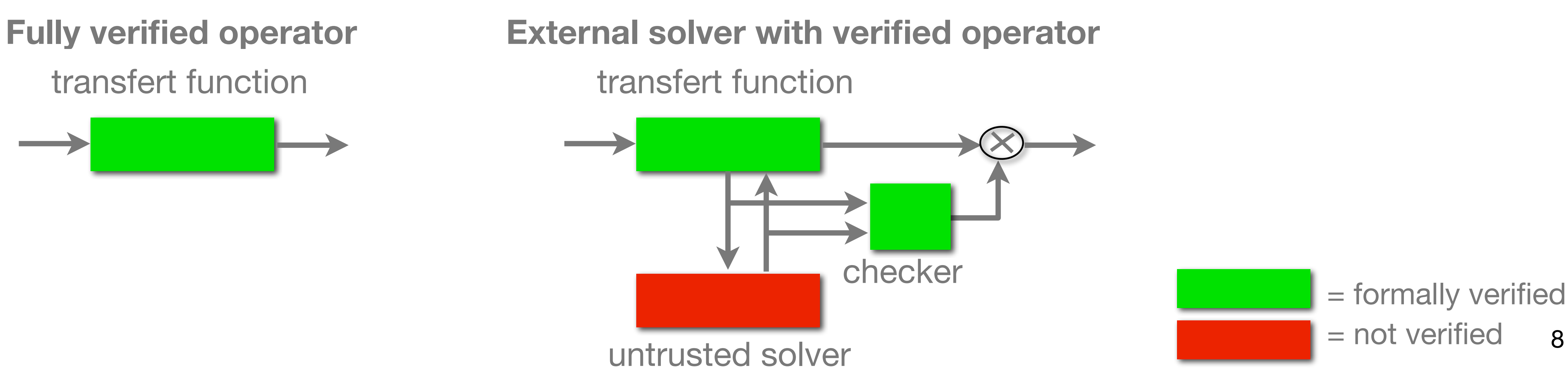
Verasco
Implementation

34 000 lines of Coq, excluding blanks and comments

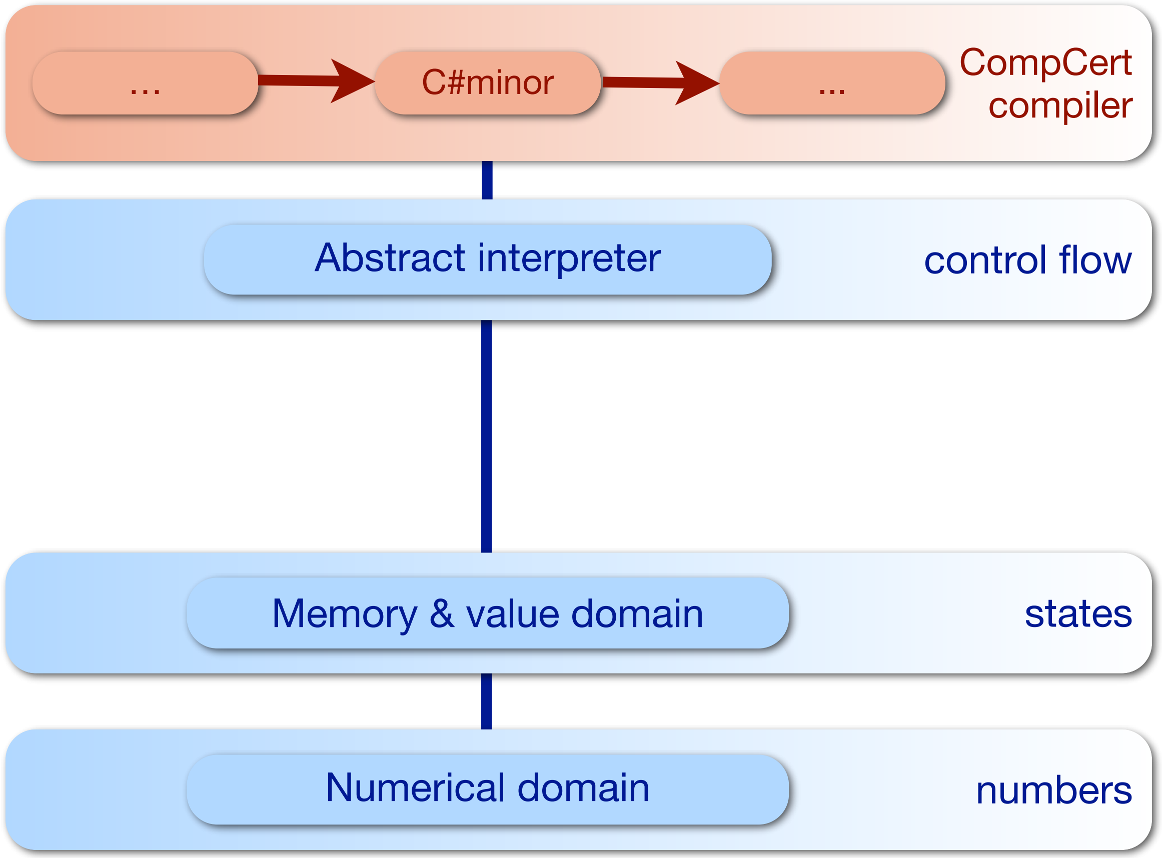
- half proof, half code & specs
- plus parts reused from CompCert

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.

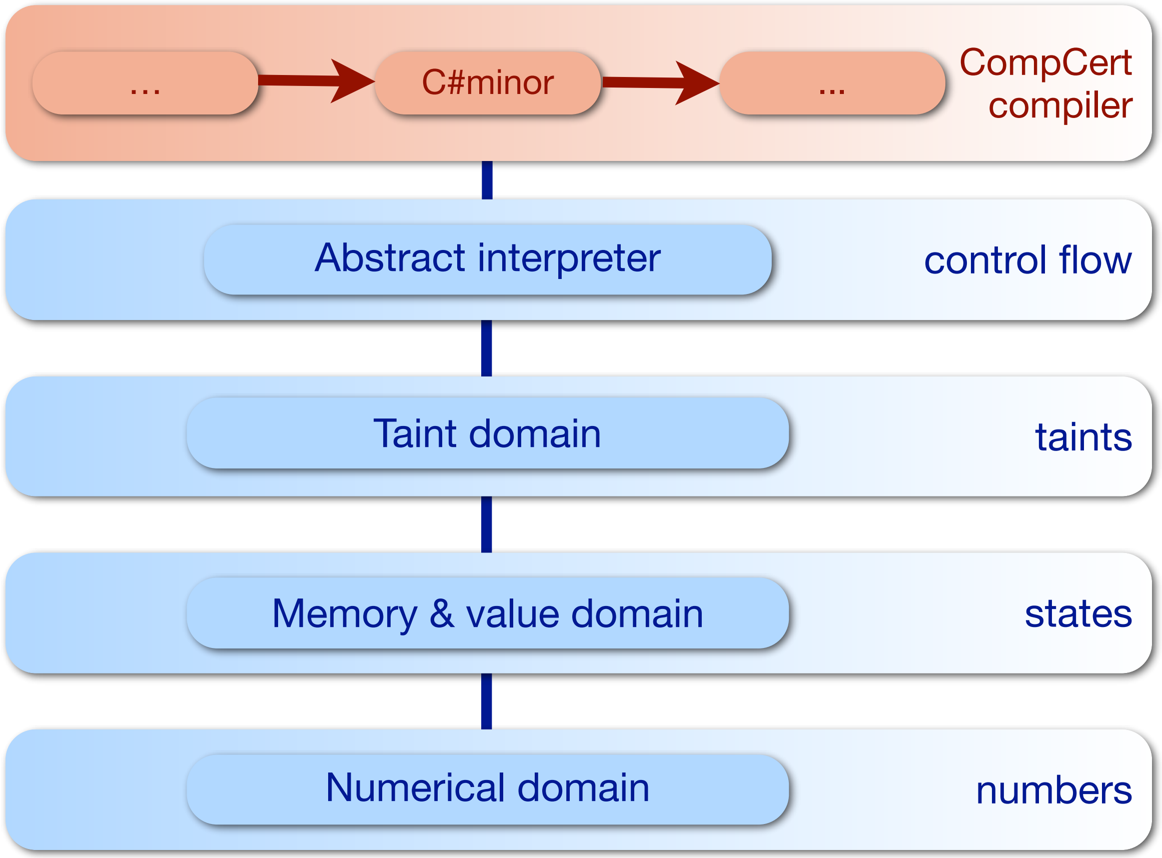


Constant-time analysis at source level



Constant-time analysis at source level

We design an *abstract functor*

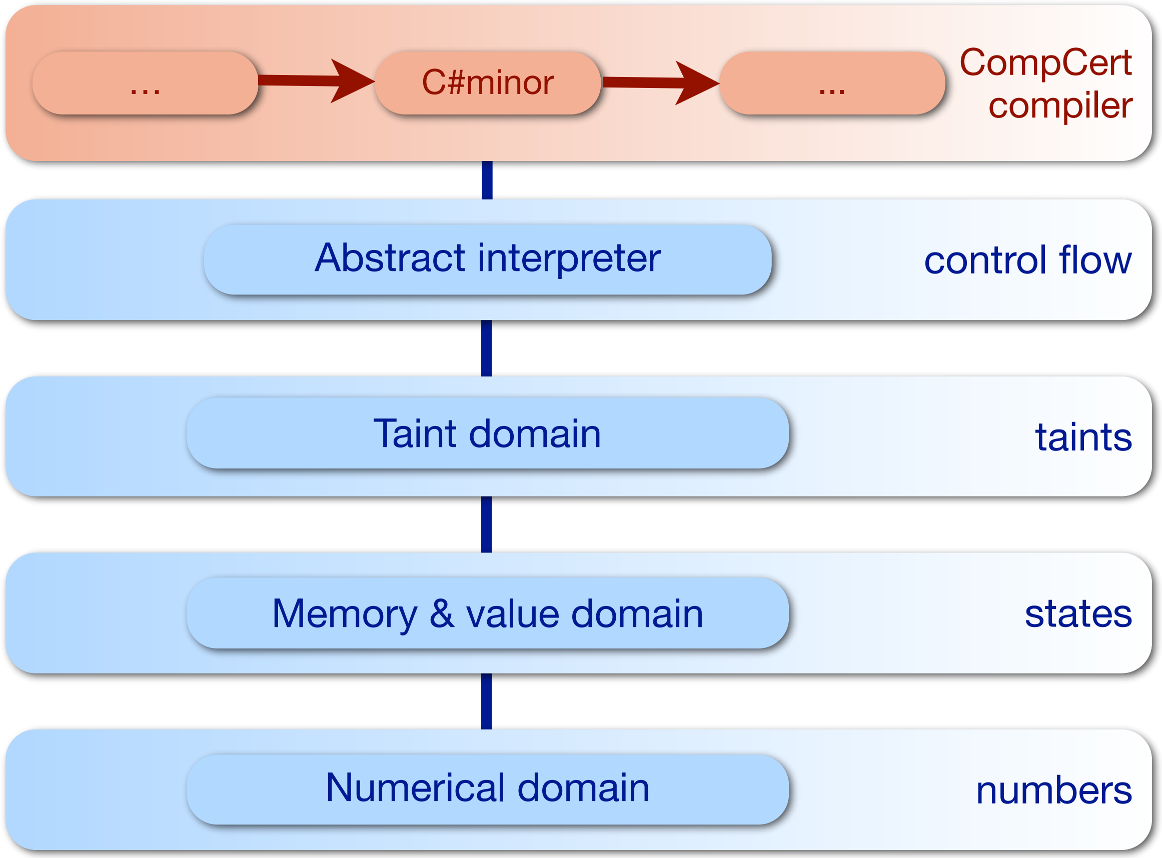


Constant-time analysis at source level

We design an *abstract functor*

- takes as input an abstract memory domain

$$
\begin{aligned}
[\![e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\
[\![x \to e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![*e_1 \to e_2]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![x \to *e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{assert}(e)^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{concretize}^\sharp &: \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{L})
\end{aligned}
$$



Constant-time analysis at source level

We design an *abstract functor*

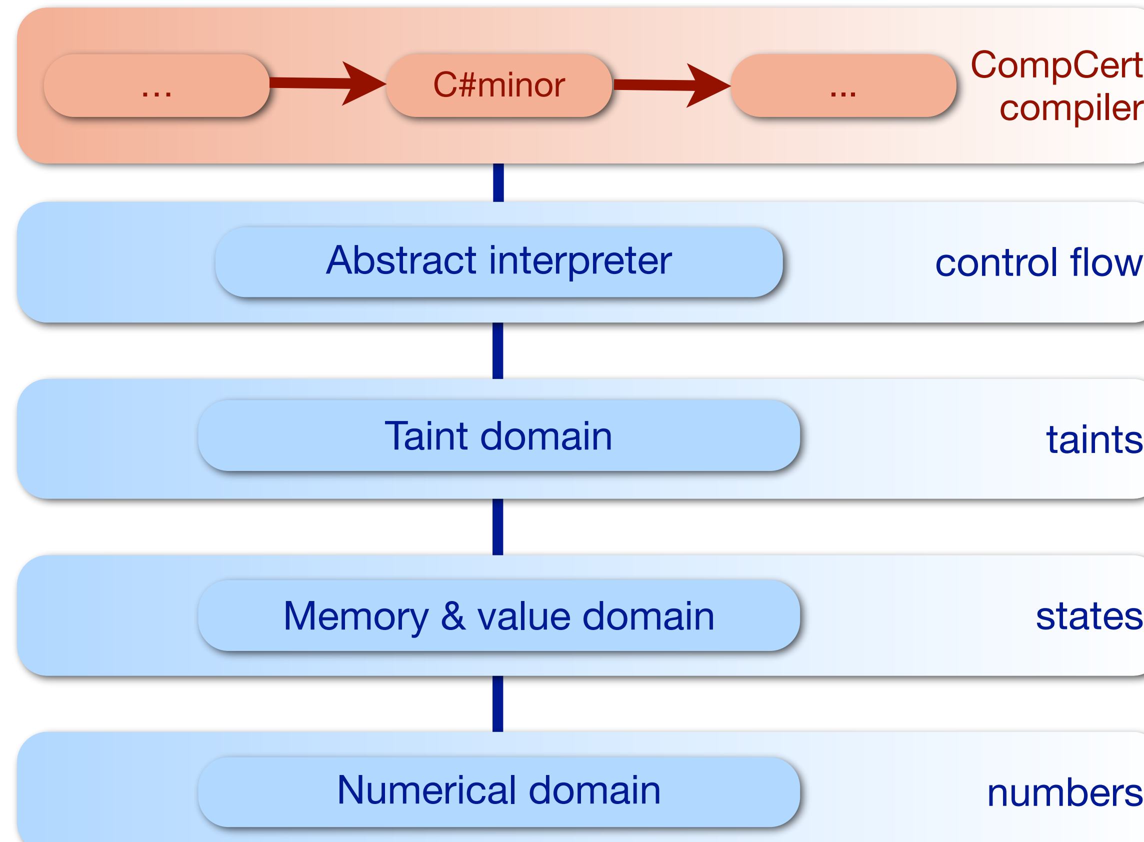
- takes as input an abstract memory domain

abstract memory

abstract value

$$
\begin{array}{rl}
[\![e]\!]^\sharp : & \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\
[\![x \to e]\!]^\sharp : & \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![*e_1 \to e_2]\!]^\sharp : & \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![x \to *e]\!]^\sharp : & \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{assert}(e)^\sharp : & \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{concretize}^\sharp : & \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{L})
\end{array}
$$

set of concrete memory locations



Constant-time analysis at source level

We design an *abstract functor*

- takes as input an abstract memory domain

abstract memory

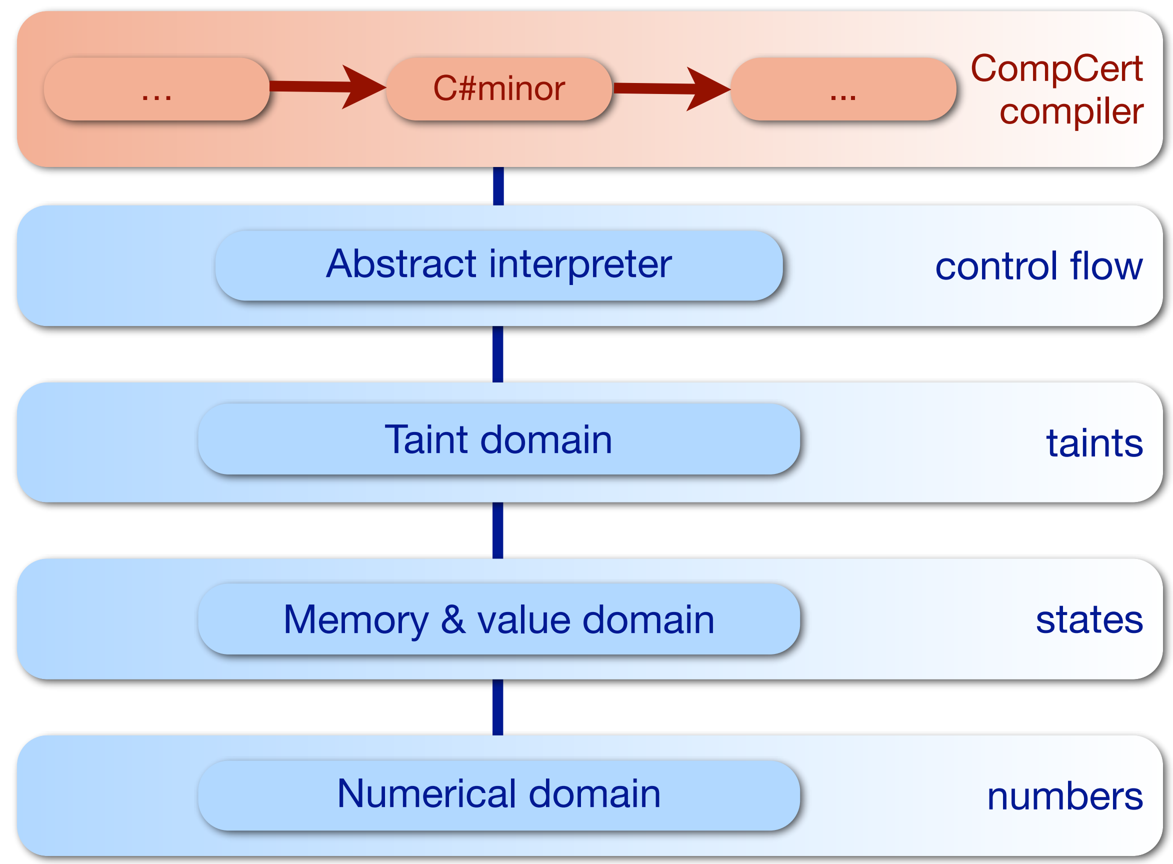
abstract value

$$
\begin{aligned}
\llbracket e \rrbracket^\sharp &: \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\
\llbracket x \to e \rrbracket^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\llbracket *e_1 \to e_2 \rrbracket^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\llbracket x \to *e \rrbracket^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{assert}(e)^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{concretize}^\sharp &: \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{L})
\end{aligned}
$$

set of concrete memory locations

- returns an abstract domain that taints every memory cells

$$
\begin{aligned}
\mathcal{T}\llbracket e \rrbracket^\sharp &: \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{V}^\sharp_{\mathsf{taint}} \\
\mathcal{T}\llbracket x \to e \rrbracket^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}} \\
\mathcal{T}\llbracket *e_1 \to e_2 \rrbracket^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}} \\
\mathcal{T}\llbracket x \to *e \rrbracket^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}}
\end{aligned}
$$



Constant-time analysis at source level

We design an *abstract functor*

- takes as input an abstract memory domain

abstract memory

abstract value

$$
\begin{aligned}
[\![e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\
[\![x \to e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![*e_1 \to e_2]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![x \to *e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{assert}(e)^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{concretize}^\sharp &: \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{L})
\end{aligned}
$$

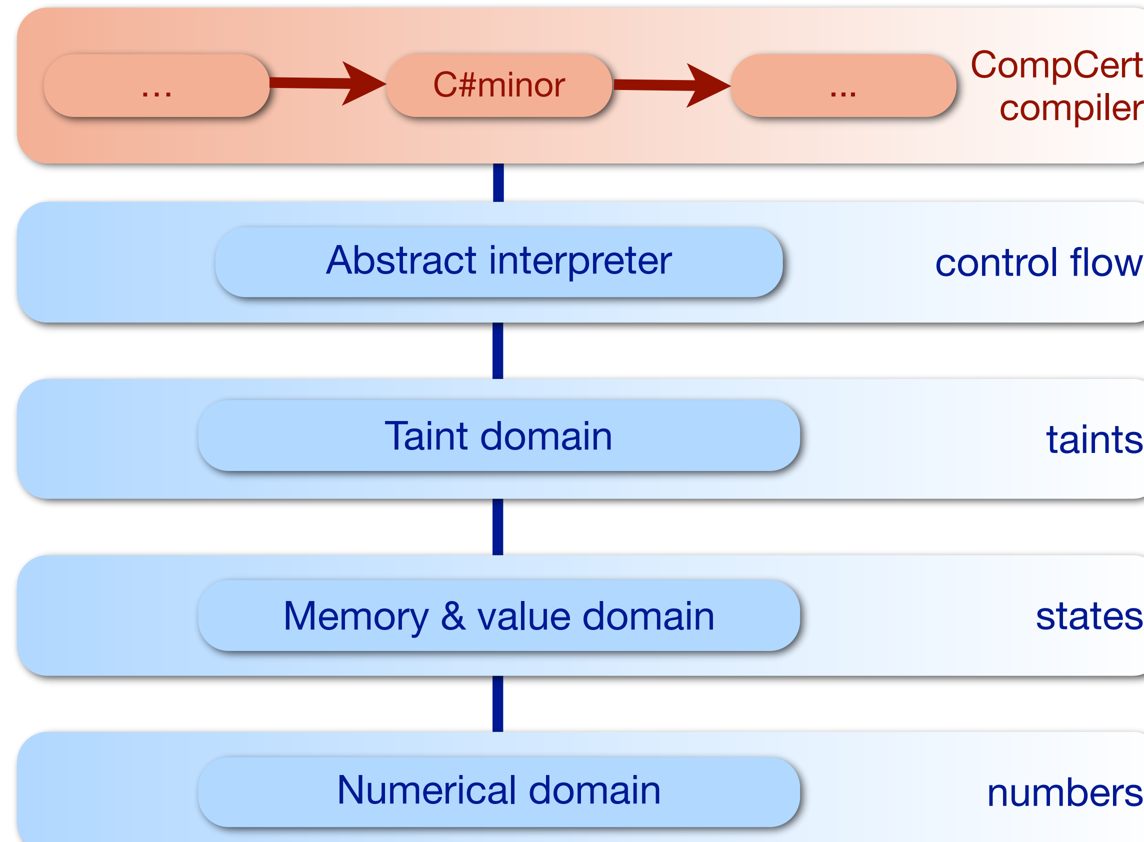
set of concrete memory locations

- returns an abstract domain that taints every memory cells

value taints {MustBeLow, MayBeHigh}

$$
\begin{aligned}
\mathcal{T}[\![e]\!]^\sharp &: \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{V}^\sharp_{\mathsf{taint}} \\
\mathcal{T}[\![x \to e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}} \\
\mathcal{T}[\![*e_1 \to e_2]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}} \\
\mathcal{T}[\![x \to *e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}}
\end{aligned}
$$

tainting of each memory cell



Constant-time analysis at source level

We design an *abstract functor*

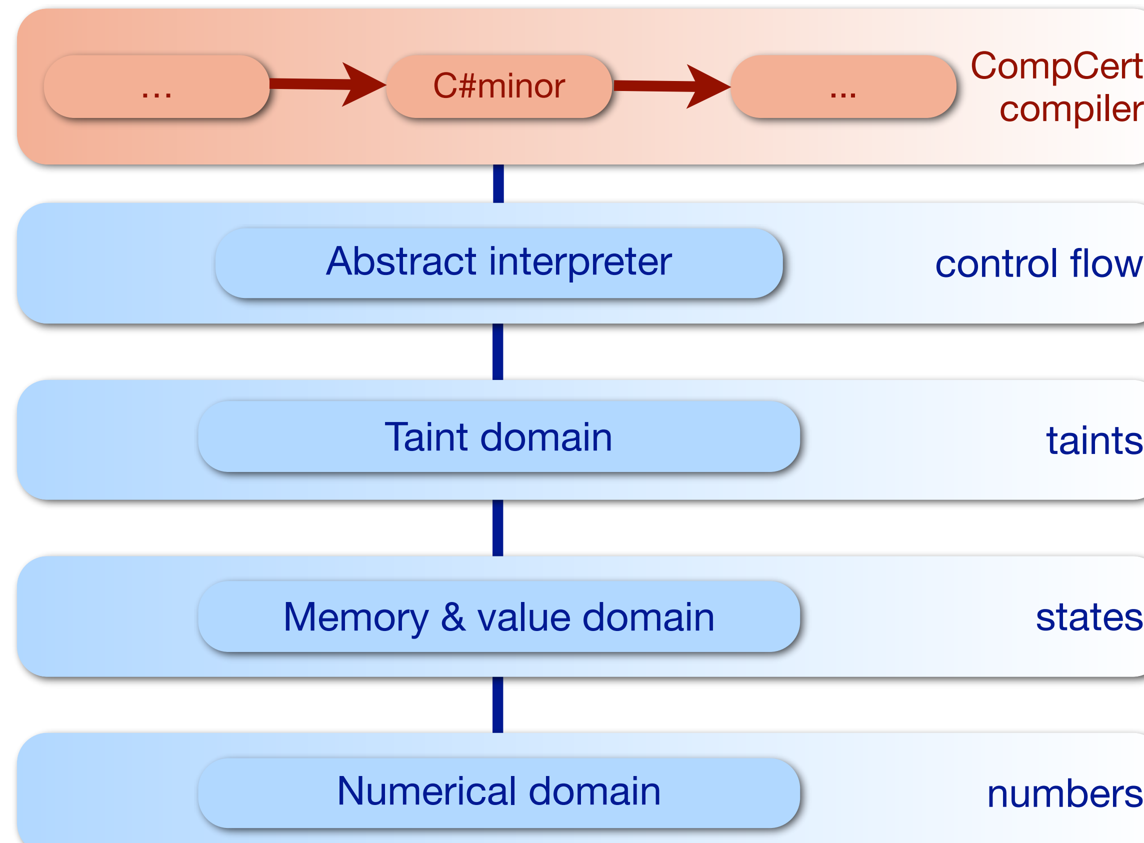
- takes as input an abstract memory domain

abstract memory

abstract value

$$\begin{aligned} [\![e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\ [\![x \to e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\ [\![*e_1 \to e_2]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\ [\![x \to *e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\ \text{assert}(e)^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\ \text{concretize}^\sharp &: \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{L}) \end{aligned}$$

set of concrete memory locations



- returns an abstract domain that taints every memory cells

value taints {MustBeLow, MayBeHigh}

$$\begin{aligned} \mathcal{T}[\![e]\!]^\sharp &: \mathbb{M}^\sharp_{\text{taint}} \to \mathbb{V}^\sharp_{\text{taint}} \\ \mathcal{T}[\![x \to e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\text{taint}} \to \mathbb{M}^\sharp_{\text{taint}} \\ \mathcal{T}[\![*e_1 \to e_2]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\text{taint}} \to \mathbb{M}^\sharp_{\text{taint}} \\ \mathcal{T}[\![x \to *e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\text{taint}} \to \mathbb{M}^\sharp_{\text{taint}} \end{aligned}$$

tainting of each memory cell

Example:

$$\mathcal{T}[\![*e_1 \to e_2]\!]^\sharp(m^\sharp, t^\sharp) = t^\sharp[l \mapsto \mathcal{T}[\![e_2]\!]^\sharp]_{\forall l \in \text{concretize}^\sharp \circ [\![e_1]\!]^\sharp(m^\sharp)}$$

Experiments at source level (ESORICS'17)

Example	Size	Loc	Time
`aes`	1171	1399	41.39
`curve25519-donna`	1210	608	586.20
`des`	229	436	2.28
`rlwe_sample`	145	1142	30.76
`salsa20`	341	652	0.04
`sha3`	531	251	57.62
`snow`	871	460	3.37
`tea`	121	109	3.47
`nacl_chacha20`	384	307	0.34
`nacl_sha256`	368	287	0.04
`nacl_sha512`	437	314	1.02
`mbedtls_sha1`	544	354	0.19
`mbedtls_sha256`	346	346	0.38
`nbedtls_sha512`	310	399	0.26
`mee-cbc`	1959	939	933.37

Experiments at source level (ESORICS'17)

Example	Size	Loc	Time
aes	1171	1399	41.39
curve25519-donna	1210	608	586.20
des	229	436	2.28
rlwe_sample	145	1142	30.76
salsa20	341	652	0.04
sha3	531	251	57.62
snow	871	460	3.37
tea	121	109	3.47
nacl_chacha20	384	307	0.34
nacl_sha256	368	287	0.04
nacl_sha512	437	314	1.02
mbedtls_sha1	544	354	0.19
mbedtls_sha256	346	346	0.38
nbedtls_sha512	310	399	0.26
mee-cbc	1959	939	933.37

Same benchmarks than Almeida et al.

J.B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir and M.Emmi. *Verifying Constant-Time Implementations.* USENIX Security Symposium 2016.

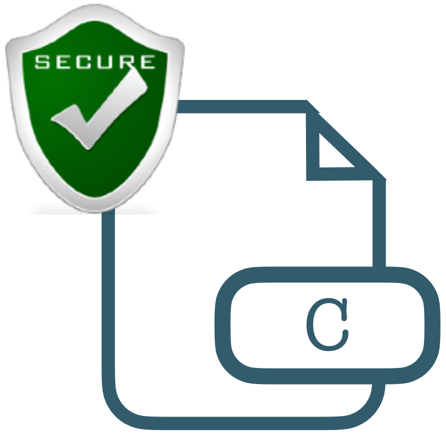
Not handled by Almeida et al. because LLVM alias analysis limitations

Conclusion

Conclusion

this lecture focused on Crypto-Constant-Time security property

- We can build secure programming abstractions at source level (C-like) using Abstract Interpretation

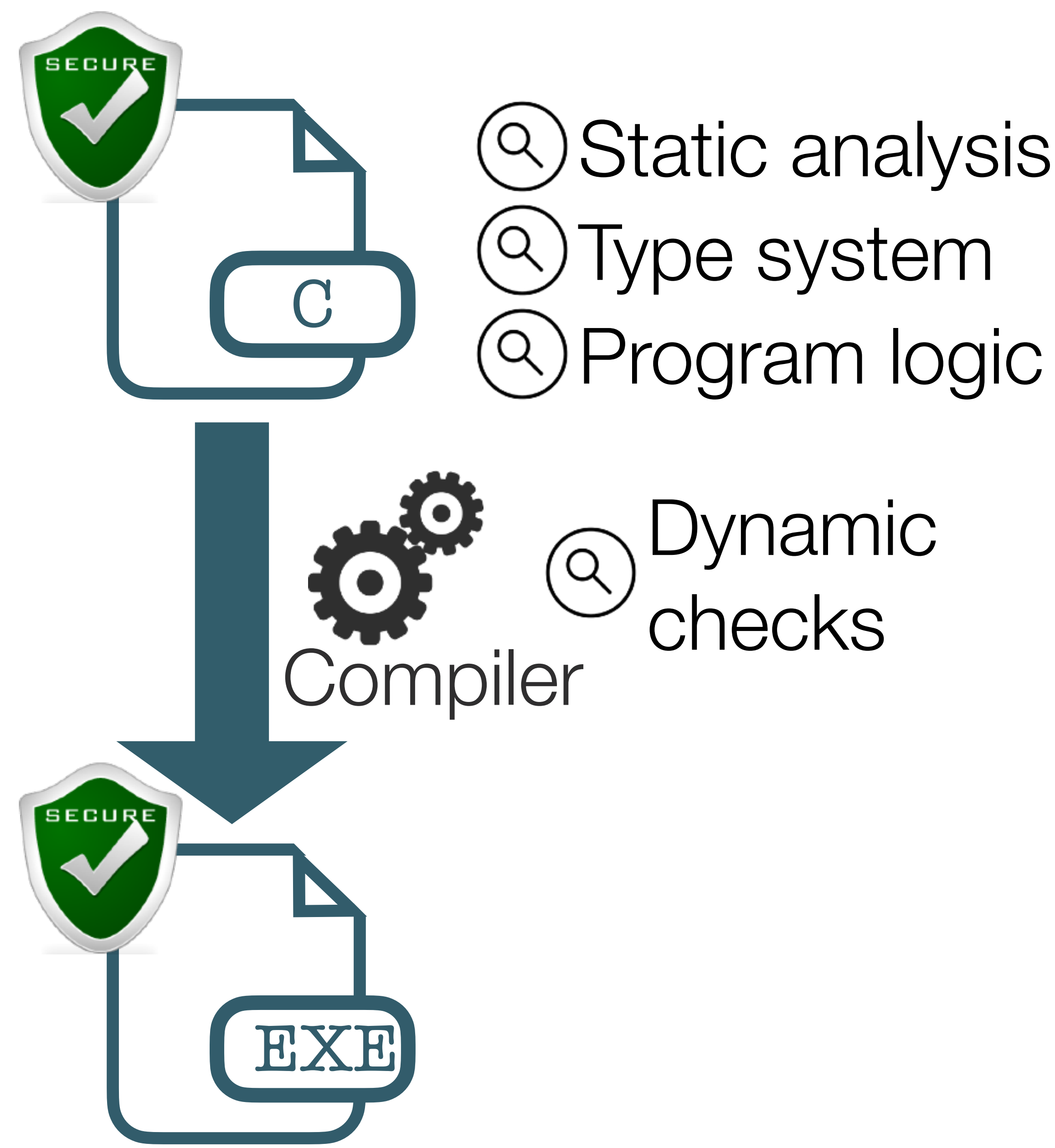


Static analysis
Type system
Program logic

Conclusion

this lecture focused on Crypto-Constant-Time security property

- We can build secure programming abstractions at source level (C-like) using Abstract Interpretation
- We make sure the compiler will generate executables that are as secure



Conclusion

this lecture focused on Crypto-Constant-Time security property

- We can build secure programming abstractions at source level (C-like) using Abstract Interpretation
- We make sure the compiler will generate executables that are as secure
- We reduce as much as possible the TCB (Trusted Computing Base) with formal proofs