Interpréteurs abstraits mécanisés

David Pichardie

Comment vérifier un logiciel ?

Comment vérifier un logiciel ?

La complexité croissante des logiciels nécessite des technique de validation toujours plus efficaces

Comment vérifier un logiciel ?

La complexité croissante des logiciels nécessite des technique de validation toujours plus efficaces

- Vérification manuelle

  ➡ ne passe pas à l'échelle

Comment vérifier un logiciel ?

La complexité croissante des logiciels nécessite des technique de validation toujours plus efficaces

- Vérification manuelle
  - ➡ ne passe pas à l’échelle
- Recherche automatique d’erreurs
  - ➡ peut oublier des erreurs

Comment vérifier un logiciel ?

La complexité croissante des logiciels nécessite des technique de validation toujours plus efficaces

- Vérification manuelle
  - ➡ ne passe pas à l'échelle
- Recherche automatique d'erreurs
  - ➡ peut oublier des erreurs
- Vérification automatique exhaustive
  - ➡ trouve toutes les erreurs, mais peut lancer des fausses alarmes
    exemple : l'analyseur Astrée

logiciel de commande de vol
~1M lignes de code C
0 alarmes

Comment vérifier un logiciel ?

La complexité croissante des logiciels nécessite des technique de validation toujours plus efficaces

- Vérification manuelle
  - ➡ ne passe pas à l’échelle
- Recherche automatique d’erreurs
  - ➡ peut oublier des erreurs
- Vérification automatique exhaustive
  - ➡ trouve toutes les erreurs, mais peut lancer des fausses alarmes
    exemple : l’analyseur Astrée

Comment vérifier un logiciel ?

La complexité croissante des logiciels nécessite des technique de validation toujours plus efficaces

- Vérification manuelle
  - ➡ ne passe pas à l'échelle
- Recherche automatique d'erreurs
  - ➡ peut oublier des erreurs
- Vérification automatique exhaustive
  - ➡ trouve toutes les erreurs, mais peut lancer des fausses alarmes
    exemple : l'analyseur Astrée
- Vérification formelle vérifiée
  - ➡ le vérificateur est accompagnée d'une preuve de sa propre correction
  - ➡ la preuve est vérifiée dans un assistant de preuve

http://www.astree.ens.fr/

logiciel de commande de vol
~1M lignes de code C
0 alarmes

Comment vérifier le vérificateur ?

Comment vérifier le vérificateur ?

Une idée simple

Comment vérifier le vérificateur ?

Une idée simple

Programmer et prouver le vérificateur dans le même langage !

Comment vérifier le vérificateur ?

Une idée simple

Programmer et prouver le vérificateur dans le même langage !

Quel langage ?

Comment vérifier le vérificateur ?

Une idée simple

Programmer et prouver le vérificateur dans le même langage !

Quel langage ?

Coq

Coq : un animal à deux visages…

Premier visage

- un assistant de preuve qui permet de prouver de façon interactive des preuves mathématiques

Deuxième visage

- un langage de programmation fonctionnelle avec un système de type très riche

```
tri : ∀ l: list int, { l’: list int | Triée l ∧ Permutation l l’ }
```

- et un mécanisme d’extraction vers OCaml

```
tri : int list → int list
```

Méthodologie

Méthodologie

Système formel
(ici Coq)

Méthodologie

Nous programmons l'analyseur en Coq

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

Analyseur statique

Système formel
(ici Coq)

Méthodologie

Nous programmons l'analyseur en Coq

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

Nous énonçons son théorème de correction vis-à-vis de la sémantique formelle du langage analysé

```
Theorem analyser_is_sound :
  ∀ p, analyser p = Yes → Sound(p)
```

Analyseur statique

Sémantique du langage

Système formel
(ici Coq)

Méthodologie

Nous programmons l'analyseur en Coq

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

Nous énonçons son théorème de correction vis-à-vis de la sémantique formelle du langage analysé

```
Theorem analyser_is_sound :
  ∀ p, analyser p = Yes → Sound(p)
```

Nous prouvons ce théorème de manière interactive

```
Proof. ... (* few days later *) ... Qed.
```

Analyseur statique

Sémantique du langage

Preuve de correction

Système formel
(ici Coq)

Méthodologie

Nous programmons l'analyseur en Coq

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

Nous énonçons son théorème de correction vis-à-vis de la sémantique formelle du langage analysé

```
Theorem analyser_is_sound :
  ∀ p, analyser p = Yes → Sound(p)
```

Nous prouvons ce théorème de manière interactive

```
Proof. ... (* few days later *) ... Qed.
```

Nous extrayons une implémentation OCaml de l'analyseur

```
Extraction analyzer.
```

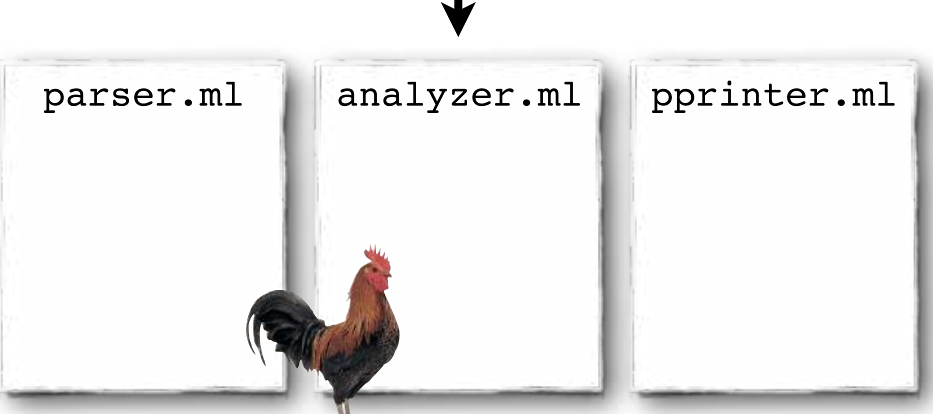
Analyseur statique

Sémantique du langage

Preuve de correction

Système formel
(ici Coq)

↓



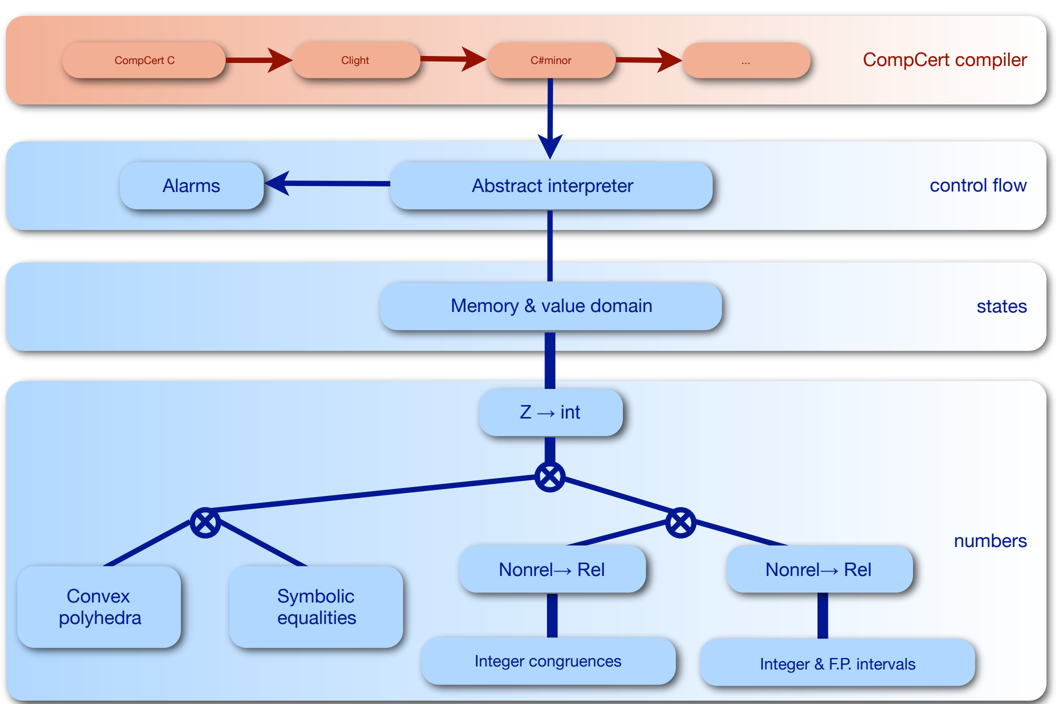
Plan de ce séminaire

Un interpréteur abstrait dénotationnel vérifié

```
Fixpoint AbSem (i:instr) (l2:pp): t -> array t :=
  match i with
    | Skip l1 => fun Pre => ⊥♯ +[l1↦Pre]♯ +[l2↦Pre]♯
    | Assign l1 x e => fun Pre => ⊥♯ +[l1↦Env]♯ +[l2↦AbEnv.assign Env x e]♯
    | Assert l1 t => fun Pre => ⊥♯ +[p↦Env]♯ +[l↦AbEnv.assume t Env]♯
    | If l1 t i1 i2 => fun Pre =>
        let C1 := AbSem i1 l2 (AbEnv.assert t Env) in
        let C2 := AbSem i2 l2 (AbEnv.assert (Not t) Env) in
           (C1 ⊔♯ C2) +[l1↦Env]♯
    | While l1 t i => fun Pre =>
       let I := approx_lfp
                  (fun X => Env ⊔♯ (get (AbSem i l1 (AbEnv.assume t X)) l1)) in
          (AbSem i l1 (AbEnv.assume t I)) +[l1↦I]♯ +[l2↦AbEnv.assume (Not t) I]♯
    | Seq i1 i2 => fun Pre =>
       let C := (AbSem i1 (first i2) Pre) in
         C ⊔♯ (AbSem i2 l2 (get C (first i2)))
  end.
```

David Cachera and David Pichardie. A certified denotational abstract interpreter. In *Proc. of International Conference on Interactive Theorem Proving (ITP-10)*, volume 6172 of *Lecture Notes in Computer Science*, pages 9--24. Springer-Verlag, 2010

Un interpréteur abstrait C vérifié : Verasco



Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In 42nd symposium Principles of Programming Languages, pages 247--259. ACM Press, 2015

Un (autre) interpréteur abstrait mécanisé

Objectifs

- formaliser un analyseur statique en suivant la méthodologie interprétation abstraite

Notre formalisation s'appuie sur les notions de

- treillis complets
- sémantique collectrice
- le slogan « correcte par construction de l'interprétation abstraite »

Mais laisse de côté des éléments méthodologies forts

- notamment le calcul symbolique des fonctions abstraites par raffinement successifs

Syntaxe du langage

```
Inductive instr :=
    Assign            (x:var) (e:expr)
  | Skip
  | Assert            (t:test)
  | If            (t:test) (b1 b2:instr)
  | While            (t:test) (b:instr)
  | Seq (i1 i2:instr).
```

Syntaxe du langage

```
Inductive instr :=
    Assign (p:pp) (x:var) (e:expr)
  | Skip (p:pp)
  | Assert (p:pp) (t:test)
  | If (p:pp) (t:test) (b1 b2:instr)
  | While (p:pp) (t:test) (b:instr)
  | Seq (i1 i2:instr).
```

on ajoute des labels pour accrocher des informations

Syntaxe du langage

```
Definition var := Word.
Definition pp := Word.
Inductive op := Add | Sub | Mult.
Inductive expr :=
    Const (n:Z)
  | Unknown
  | Var (x:var)
  | Numop (o:op) (e1 e2:expr).
Inductive comp := Eq | Lt.
Inductive test :=
  | Numcomp (c:comp) (e1 e2:expr)
  | Not (t:test)
  | And (t1 t2:test)
  | Or (t1 t2:test).
```

```
Inductive instr :=
    Assign (p:pp) (x:var) (e:expr)
  | Skip (p:pp)
  | Assert (p:pp) (t:test)
  | If (p:pp) (t:test) (b1 b2:instr)
  | While (p:pp) (t:test) (b:instr)
  | Seq (i1 i2:instr).

Record program  := Prog {
  p_instr:instr;
  p_end: pp;
  vars: list var
}.
```

Syntaxe du langage

```
Definition var := Word.
Definition pp := Word.
Inductive op := Add | Sub | Mult.
Inductive expr :=
    Const (n:Z)
  | Unknown
  | Var (x:var)
  | Numop (o:op) (e1 e2:expr).
Inductive comp := Eq | Lt.
Inductive test :=
  | Numcomp (c:comp) (e1 e2:expr)
  | Not (t:test)
  | And (t1 t2:test)
  | Or (t1 t2:test).
```

```
Inductive instr :=
    Assign (p:pp) (x:var) (e:expr)
  | Skip (p:pp)
  | Assert (p:pp) (t:test)
  | If (p:pp) (t:test) (b1 b2:instr)
  | While (p:pp) (t:test) (b:instr)
  | Seq (i1 i2:instr).

Record program := P
  p_instr:instr;
  p_end: pp;
  vars: list var
}.
```

instruction principale

Syntaxe du langage

```
Definition var := Word.
Definition pp := Word.
Inductive op := Add | Sub | Mult.
Inductive expr :=
    Const (n:Z)
  | Unknown
  | Var (x:var)
  | Numop (o:op) (e1 e2:expr).
Inductive comp := Eq | Lt.
Inductive test :=
  | Numcomp (c:comp) (e1 e2:expr)
  | Not (t:test)
  | And (t1 t2:test)
  | Or (t1 t2:test).
```

```
Inductive instr :=
    Assign (p:pp) (x:var) (e:expr)
  | Skip (p:pp)
  | Assert (p:pp) (t:test)
  | If (p:pp) (t:test) (b1 b2:instr)
  | While (p:pp) (t:test) (b:instr)
  | Seq (i1 i2:instr).
```

```
Record program := P
  p_instr:instr;
  p_end: pp;
  vars: list var
}.
```

instruction principale

dernier label

Syntaxe du langage

```
Definition var := Word.
Definition pp := Word.
Inductive op := Add | Sub | Mult.
Inductive expr :=
    Const (n:Z)
  | Unknown
  | Var (x:var)
  | Numop (o:op) (e1 e2:expr).
Inductive comp := Eq | Lt.
Inductive test :=
  | Numcomp (c:comp) (e1 e2:expr)
  | Not (t:test)
  | And (t1 t2:test)
  | Or (t1 t2:test).
```

```
Inductive instr :=
    Assign (p:pp) (x:var) (e:expr)
  | Skip (p:pp)
  | Assert (p:pp) (t:test)
  | If (p:pp) (t:test) (b1 b2:instr)
  | While (p:pp) (t:test) (b:instr)
  | Seq (i1 i2:instr).
```

```
Record program := P
  p_instr:instr;
  p_end: pp;
  vars: list var
}.
```

instruction principale

dernier label

déclarations des variables

Syntaxe du langage

entiers 32 bits

```
Definition var := Word.
Definition pp := Word.
Inductive op := Add | Sub | Mult.
Inductive expr :=
    Const (n:Z)
  | Unknown
  | Var (x:var)
  | Numop (o:op) (e1 e2:expr).
Inductive comp := Eq | Lt.
Inductive test :=
  | Numcomp (c:comp) (e1 e2:expr)
  | Not (t:test)
  | And (t1 t2:test)
  | Or (t1 t2:test).
```

```
Inductive instr :=
    Assign (p:pp) (x:var) (e:expr)
  | Skip (p:pp)
  | Assert (p:pp) (t:test)
  | If (p:pp) (t:test) (b1 b2:instr)
  | While (p:pp) (t:test) (b:instr)
  | Seq (i1 i2:instr).
```

```
Record program  := P
  p_instr:instr;
  p_end: pp;
  vars: list var
}.
```

instruction principale

dernier label

déclarations des variables

Sémantique du langage

Domaines sémantiques

```
Definition env := var -> Z.
Inductive config := | Final (ρ:env) | Inter (i:instr) (ρ:env).
```

Sémantique opérationnelle

```
Inductive sos (p:program) : (instr * env) -> config -> Prop :=
 | sos_affect : ∀ l x e n ρ1 ρ2,
     sem_expr p ρ1 e n ->
     subst ρ1 x n ρ2 ->
     In x (vars p) ->
     sos p (Assign l x e,ρ1) (Final ρ2)
  | …
```

Etats accessibles à partir d’un environnement initial

```
Inductive reachable_sos (p:program) : pp*env -> Prop := …
```

Objectifs de cette formalisation

L'analyseur calcule une abstraction de la sémantique du programme

```
Definition analyse : program -> abdom := […]
```

A chaque élément abstrait correspond une propriété sur `pp` × `env`

```
Definition γ : abdom -> 𝒫(pp * env) := […]
```

L'analyseur doit calculer une sur-approximation des états accessibles

```
Theorem analyse_correct : ∀ prog,
  reachable_sos prog ⊆ γ (analyse prog).
Proof.
  […]
Qed.
```

Feuille de route

Sémantique standard

Sémantique abstraite

Feuille de route

Sémantiques mécanisées, cinquième cours

Un art abstrait:
l'analyse statique par interprétation abstraite

Xavier Leroy
2020-01-17

Collège de France, chaire de sciences du logiciel

Sémantique standard

preuve directe

Sémantique abstraite

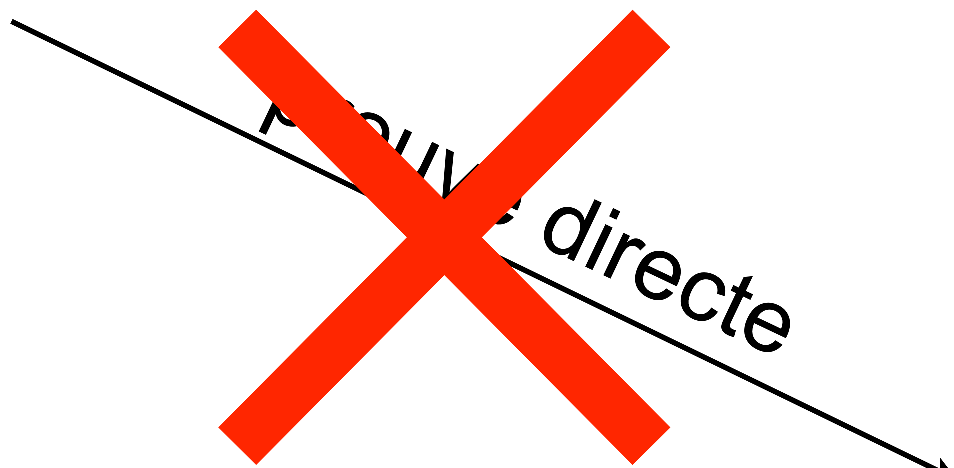
Feuille de route

Sémantiques mécanisées, cinquième cours

Un art abstrait:
l'analyse statique par interprétation abstraite

Xavier Leroy
2020-01-17
Collège de France, chaire de sciences du logiciel

Sémantique standard



Sémantique abstraite

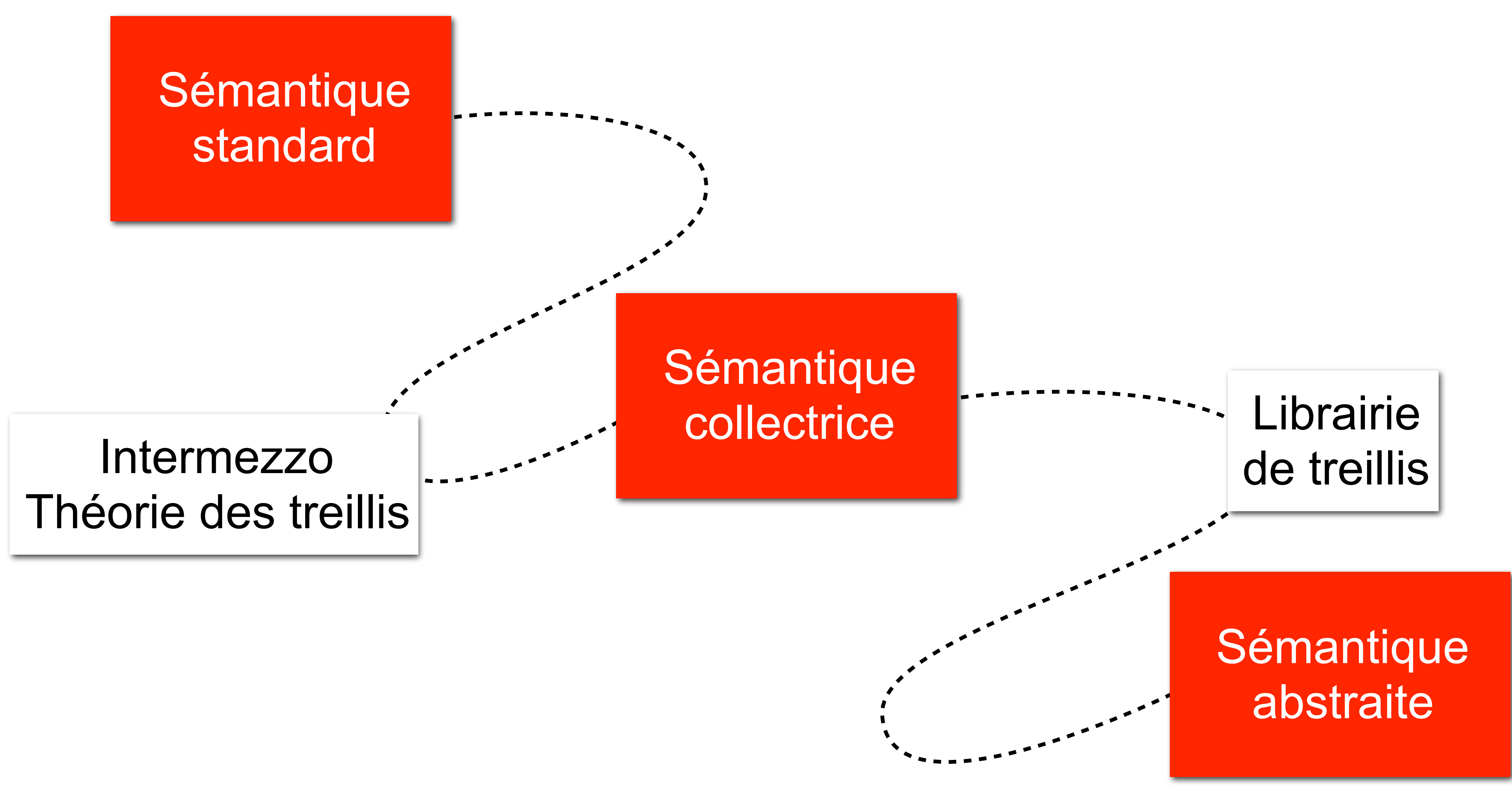
Feuille de route

Sémantique standard

Sémantique collectrice

Sémantique abstraite

Feuille de route



Quelques éléments de théorie des treillis

Nous voulons formaliser la notion de plus petit point fixe

- Treillis complets
- Théorème de Knaster-Tarski

Théorème de Knaster-Tarski

```
Definition lfp {L:Type} {CL:CompleteLattice.t L} (f:monotone L L) : L :=
  CompleteLattice.meet (PostFix f).
```

$$\bigsqcap \{x \mid f(x) \sqsubseteq x\}$$

Théorème de Knaster-Tarski

Fonctions monotones de L → L

```
Definition lfp {L:Type} {CL:CompleteLattice.t L} (f:monotone L L) : L :=
  CompleteLattice.meet (PostFix f).
```

$$\bigsqcap \{x \mid f(x) \sqsubseteq x\}$$

Théorème de Knaster-Tarski

Treillis complet sur L

Fonctions monotones de L → L

```
Definition lfp {L:Type} {CL:CompleteLattice.t L} (f:monotone L L) : L :=
  CompleteLattice.meet (PostFix f).
```

$$\bigsqcap \{x \mid f(x) \sqsubseteq x\}$$

Fonctions monotones

```
Class monotone A B {PA:Poset.t A} {PB:Poset.t B} : Type := Mono {
  mon_func : A -> B;
  mon_prop : ∀ a1 a2, a1 ⊑ a2 -> (mon_func a1) ⊑ (mon_func a2)
}.
```

Fonctions monotones

Nous utilisons des *type classes*

```
Class monotone A B {PA:Poset.t A} {PB:Poset.t B} : Type := Mono {
  mon_func : A -> B;
  mon_prop : ∀ a1 a2, a1 ⊑ a2 -> (mon_func a1) ⊑ (mon_func a2)
}.
```

Fonctions monotones

Nous utilisons des *type classes*

Une fonction monotone est un terme (Mono f π) avec π une preuve de monotonie de f

```
Class monotone A B {PA:Poset.t A} {PB:Poset.t B} : Type := Mono {
  mon_func : A -> B;
  mon_prop : ∀ a1 a2, a1 ⊑ a2 -> (mon_func a1) ⊑ (mon_func a2)
}.
```

Théorème de Knaster-Tarski

```
Definition lfp {L:Type} {CL:CompleteLattice.t L} (f:monotone L L) : L :=
  CompleteLattice.meet (PostFix f).
```

$$\bigsqcap \{x \mid f(x) \sqsubseteq x\}$$

```
Section Knaster_Tarski.

Variable L : Type.
Variable CL : CompleteLattice.t L.
Variable f : monotone L L.

Lemma lfp_fixpoint : f (lfp f) == lfp f.
Lemma lfp_least_fixpoint : ∀ x, f x == x -> lfp f ⊑ x.
Lemma lfp_postfixpoint : f (lfp f) ⊑ lfp f.
Lemma lfp_least_postfixpoint : ∀ x, f x ⊑ x -> lfp f ⊑ x.
Lemma lfp_monotone : ∀ f1 f2 : monotone L L, f1 ⊑ f2 -> lfp f1 ⊑ lfp f2.

End Knaster_Tarski.
```

Théorème de Knaster-Tarski

Argument implicite des *types classes*

```
Definition lfp {L:Type} {CL:CompleteLattice.t L} (f:monotone L L) : L :=
  CompleteLattice.meet (PostFix f).
```

$$\bigsqcap \{x \mid f(x) \sqsubseteq x\}$$

```
Section Knaster_Tarski.

Variable L : Type.
Variable CL : CompleteLattice.t L.
Variable f : monotone L L.

Lemma lfp_fixpoint : f (lfp f) == lfp f.
Lemma lfp_least_fixpoint : ∀ x, f x == x -> lfp f ⊑ x.
Lemma lfp_postfixpoint : f (lfp f) ⊑ lfp f.
Lemma lfp_least_postfixpoint : ∀ x, f x ⊑ x -> lfp f ⊑ x.
Lemma lfp_monotone : ∀ f1 f2 : monotone L L, f1 ⊑ f2 -> lfp f1 ⊑ lfp f2.

End Knaster_Tarski.
```

Théorème de Knaster-Tarski

Argument implicite des *types classes*

```
Definition lfp {L:Type} {CL:CompleteLattice.t L} (f:monotone L L) : L :=
  CompleteLattice.meet (PostFix f).
```

$$\bigsqcap \{x \mid f(x) \sqsubseteq x\}$$

```
Section Knaster_Tarski.

Variable L : Type.
Variable CL : CompleteLattice.t L.
Variable f : monotone L L.

Lemma lfp_fixpoint : f (lfp f) == lfp f.
Lemma lfp_least_fixpoint : ∀ x, f x == x -> lfp f ⊑ x.
Lemma lfp_postfixpoint : f (lfp f) ⊑ lfp f.
Lemma lfp_least_postfixpoint : ∀ x, f x ⊑ x -> lfp f ⊑ x.
Lemma lfp_monotone : ∀ f1 f2 : monotone L L, f1 ⊑ f2 -> lfp f1 ⊑ lfp f2.

End Knaster_Tarski.
```

Inutile de fournir les arguments implicites

Treillis complets

```
Module CompleteLattice.
  Class t (A:Type) : Type := Make
  { porder :> Poset.t A;
    join : subset A -> A;
    join_bound : ∀x:A, ∀f:subset A, f x -> x ⊑ join f;
    join_lub : ∀f:subset A, ∀z, (∀ x:A, f x -> x ⊑ z) -> join f ⊑ z
  }.
End CompleteLattice.
```

Treillis complets

```
Module CompleteLattice.
  Class t (A:Type) : Type := Make
  { porder :> Poset.t A;
    join : subset A -> A;
    join_bound : ∀x:A, ∀f:subset A, f x -> x ⊑ join f;
    join_lub : ∀f:subset A, ∀z, (∀ x:A, f x -> x ⊑ z) -> join f ⊑ z
  }.
End CompleteLattice.
```

héritage

Treillis complets

```
Class subset A {E:Equiv.t A} : Type := SubSet
  { carrier : A -> Prop;
    subset_comp_eq : ∀ x y:A, x==y -> carrier x -> carrier y}.

Module CompleteLattice.
  Class t (A:Type) : Type := Make
  { porder :> Poset.t A;
    join : subset A -> A;
    join_bound : ∀x:A, ∀f:subset A, f x -> x ⊑ join f;
    join_lub : ∀f:subset A, ∀z, (∀ x:A, f x -> x ⊑ z) -> join f ⊑ z
  }.
End CompleteLattice.
```

héritage

Treillis complets

```
Class subset A {E:Equiv.t A} : Type := SubSet
  { carrier : A -> Prop;
    subset_comp_eq : ∀ x y:A, x==y -> carrier x -> carrier y}.
Coercion carrier : subset >-> Funclass.

Module CompleteLattice.
  Class t (A:Type) : Type := Make
  { porder :> Poset.t A;
    join : subset A -> A;
    join_bound : ∀x:A, ∀f:subset A, f x -> x ⊑ join f;
    join_lub : ∀f:subset A, ∀z, (∀ x:A, f x -> x ⊑ z) -> join f ⊑ z
  }.
End CompleteLattice.
```

héritage

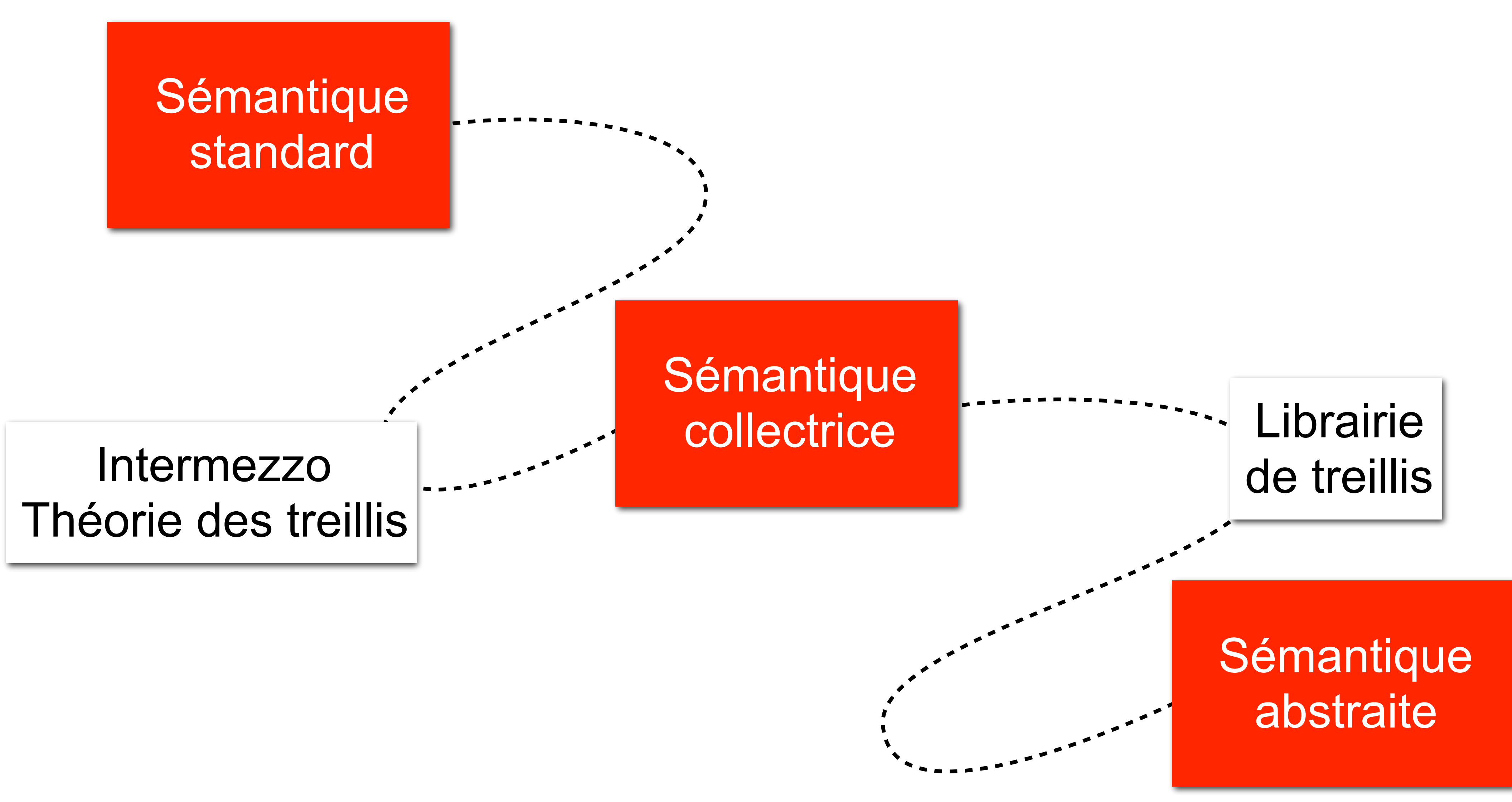
Treillis complets

```
Class subset A {E:Equiv.t A} : Type := SubSet
  { carrier : A -> Prop;
    subset_comp_eq : ∀ x y:A, x==y -> carrier x -> carrier y}.
Coercion carrier : subset >-> Funclass.

Module CompleteLattice.
  Class t (A:Type) : Type := Make
  { porder :> Poset.t A;
    join : subset A -> A;
    join_bound : ∀x:A, ∀f:subset A, f x -> x ⊑ join f;
    join_lub : ∀f:subset A, ∀z, (∀ x:A, f x -> x ⊑ z) -> join f ⊑ z
  }.
End CompleteLattice.
```

coercion

héritage

Feuille de route



Sémantique collectrice

- Un élément méthodologique fort en interprétation abstraite
- Qui ressemble à une analyse statique ...
- ... mais aussi précise qu'une sémantique
- Similaire à une sémantique dénotationelle mais sur $\mathscr{P}(S)$ au lieu de $S_\perp$.

Sémantique collectrice : exemple

```
i = 0; k = 0;

while [k < 10]^l1 {

   [i = 0]^l2;

   while [i < 9]^l3 {

      [i = i + 2]^l4
   };

   [k = k + 1]^l5
}_l6
```

k i

$l_1 \mapsto [0,10] \times ([0,10] \cap \text{Paires})$

$l_2 \mapsto [0,9] \times ([0,10] \cap \text{Paires})$

$l_3 \mapsto [0,9] \times ([0,10] \cap \text{Paires})$

$l_4 \mapsto [0,9] \times ([0,8] \cap \text{Paires})$

$l_5 \mapsto [0,9] \times ([0,10] \cap \text{Paires})$

$l_6 \mapsto \{(10,10)\}$

Sémantique collectrice

```
Program Fixpoint Collect (i:instr) (l2:pp): monotone (℘(env)) (pp->℘(env)) :=
  match i with
    | Skip l1 =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦Pre]) _
    | Assign l1 x e =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assign x e Pre]) _
    | Assert l1 t =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assume t Pre]) _
    | If l1 t i1 i2 =>
      Mono (fun Pre =>
              let C1 := Collect i1 l1 (assume t Pre) in
              let C2 := Collect i2 l1 (assume (Not t) Pre) in
                (C1 ⊔ C2)+[l1↦Pre]) _
    | While l1 t i =>
      Mono (fun Pre =>
          let F := fun X:℘(env) => Pre ⊔ Collect i l1 (assume t X) l1 in
          let I := lfp F in
                (Collect i l1 (assume t I))+[l1↦I]+[l2↦assume (Not t) I]) _
    | Seq i1 i2 =>
      Mono (fun Pre =>
              let C := Collect i1 (first i2) Pre in
                C ⊔ (Collect i2 l2 (C (first i2)))) _
  end.
```

Sémantique collectrice

```
Program Fixpoint Collect (i:instr) (l2:pp): monotone (℘(env)) (pp->℘(env)) :=
  match i with
    | Sk
      Mo
    | As
      Mo
    | Assert l1 t =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assume t Pre]) _
    | If l1 t i1 i2 =>
      Mono (fun Pre =>
                let C1 := Collect i1 l1 (assume t Pre) in
                let C2 := Collect i2 l1 (assume (Not t) Pre) in
                  (C1 ⊔ C2)+[l1↦Pre]) _
    | While l1 t i =>
      Mono (fun Pre =>
           let F := fun X:℘(env) => Pre ⊔ Collect i l1 (assume t X) l1 in
           let I := lfp F in
                  (Collect i l1 (assume t I))+[l1↦I]+[l2↦assume (Not t) I]) _
    | Seq i1 i2 =>
      Mono (fun Pre =>
                let C := Collect i1 (first i2) Pre in
                  C ⊔ (Collect i2 l2 (C (first i2)))) _
  end.
```

On collecte tous les états accessibles, en chaque points de programme, pour une précondition fixée

Sémantique collectrice

```
Program Fixpoint Collect (i:instr) (l2:pp): monotone (℘(env)) (pp->℘(env)) :=
  match i with
    | Skip l1 =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦Pre]) _
    | Assign l1 x e =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assign x e Pre]) _
    | Assert l1 t =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assume t Pre]) _
    | If l1 t i1 i2 =>
      Mono (fun Pre =>
              let C1 := Collect i1 l1 (assume t Pre) in
              let C2 := Collect i2 l1 (assume (Not t) Pre) in
                (C1 ⊔ C2)+[l1↦Pre]) _
    | While l1 t i =>
      Mono (fun Pre =>
          let F := fun X:℘(env) => Pre ⊔ Collect i l1 (assume t X) l1 in
          let I := lfp F in
                (Collect i l1 (assume t I))+[l1↦I]+[l2↦assume (Not t) I]) _
    | Seq i1 i2 =>
      Mono (fun Pre =>
              let C := Collect i1 (first i2) Pre in
                C ⊔ (Collect i2 l2 (C (first i2)))) _
  end.
```

```
Definition Esubst {A} (f:pp->℘(A)) (k:pp) (v:℘ A) : pp->℘(A) :=
  fun k' => if pp_eq k' k then (f k) ⊔ v else f k'.
Notation "f +[ x ↦ v ] " := (Esubst f x v) (at level 100).
```

```
Program Fixpoint Collect (i:instr) (l2:pp): monotone (℘(env)) (pp->℘(env)) :=
  match i with
    | Skip l1 =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦Pre]) _
    | Assign l1 x e =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assign x e Pre]) _
    | Assert l1 t =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assume t Pre]) _
    | If l1 t i1 i2 =>
      Mono (fun Pre =>
                let C1 := Collect i1 l1 (assume t Pre) in
                let C2 := Collect i2 l1 (assume (Not t) Pre) in
                  (C1 ⊔ C2)+[l1↦Pre]) _
    | While l1 t i =>
      Mono (fun Pre =>
           let F := fun X:℘(env) => Pre ⊔ Collect i l1 (assume t X) l1 in
           let I := lfp F in
                  (Collect i l1 (assume t I))+[l1↦I]+[l2↦assume (Not t) I]) _
    | Seq i1 i2 =>
      Mono (fun Pre =>
                let C := Collect i1 (first i2) Pre in
                  C ⊔ (Collect i2 l2 (C (first i2)))) _
  end.
```

Sémantique collectrice

```
Program Fixpoint Collect (i:instr) (l2:pp): monotone (℘(env)) (pp->℘(env)) :=
  match i with
    | Skip l1 =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦Pre]) _
    | Assign l1 x e =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assign x e Pre]) _
    | Assert l1 t =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assume t Pre]) _
    | If l1 t i1 i2 =>
      Mono (fun Pre =>
              let C1 := Collect i1 l1 (assume t Pre) in
              let C2 := Collect i2 l1 (assume (Not t) Pre) in
                (C1 ⊔ C2)+[l1↦Pre]) _
    | While l1 t i =>
      Mono (fun Pre =>
          let F := fun X:℘(env) => Pre ⊔ Collect i l1 (assume t X) l1 in
          let I := lfp F in
                (Collect i l1 (assume t I))+[l1↦I]+[l2↦assume (Not t) I]) _
    | Seq i1 i2 =>
      Mono (fun Pre =>
              let C := Collect i1 (first i2) Pre in
                C ⊔ (Collect i2 l2 (C (first i2)))) _
  end.
```

Sémantique collectrice

```
Program Fixpoint Collect (i:instr) (l2:pp): monotone (℘(env)) (pp->℘(env)) :=
  match i with
    | Skip l1 =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦Pre]) _
    | Assign l1 x e =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assign x e Pre]) _
    | Assert l1 t =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assume t Pre]) _
    | If l1 t i1 i2 =>
      Mono (fun Pre =>
              let C1 := Coll
              let C2 := Coll
                (C1 ⊔ C2)+[l
    | While l1 t i =>
      Mono (fun Pre =>
           let F := fun X:℘(env) => Pre ⊔ Collect i l1 (assume t X) l1 in
           let I := lfp F in
                (Collect i l1 (assume t I))+[l1↦I]+[l2↦assume (Not t) I]) _
    | Seq i1 i2 =>
      Mono (fun Pre =>
              let C := Collect i1 (first i2) Pre in
                C ⊔ (Collect i2 l2 (C (first i2)))) _
  end.
```

I == Pre ⊔ Collect i l1 (assume t I) l1

Sémantique collectrice

```
Program Fixpoint Collect (i:instr) (l2:pp): monotone (℘(env)) (pp->℘(env)) :=
  match i with
    | Skip l1 =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦Pre]) _
    | Assign l1 x e =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assign x e Pre]) _
    | Assert l1 t =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assume t Pre]) _
    | If l1 t i1 i2 =>
      Mono (fun Pre =>
              let C1 := Collect i1 l1 (assume t Pre) in
              let C2 := Collect i2 l1 (assume (Not t) Pre) in
                (C1 ⊔ C2)+[l1↦Pre]) _
    | While l1 t i =>
      Mono (fun Pre =>
          let F := fun X:℘(env) => Pre ⊔ Collect i l1 (assume t X) l1 in
          let I := lfp F in
                (Collect i l1 (assume t I))+[l1↦I]+[l2↦assume (Not t) I]) _
    | Seq i1 i2 =>
      Mono (fun Pre =>
              let C := Collect i1 (first i2) Pre in
                C ⊔ (Collect i2 l2 (C (first i2)))) _
  end.
```

Sémantique collectrice : correction

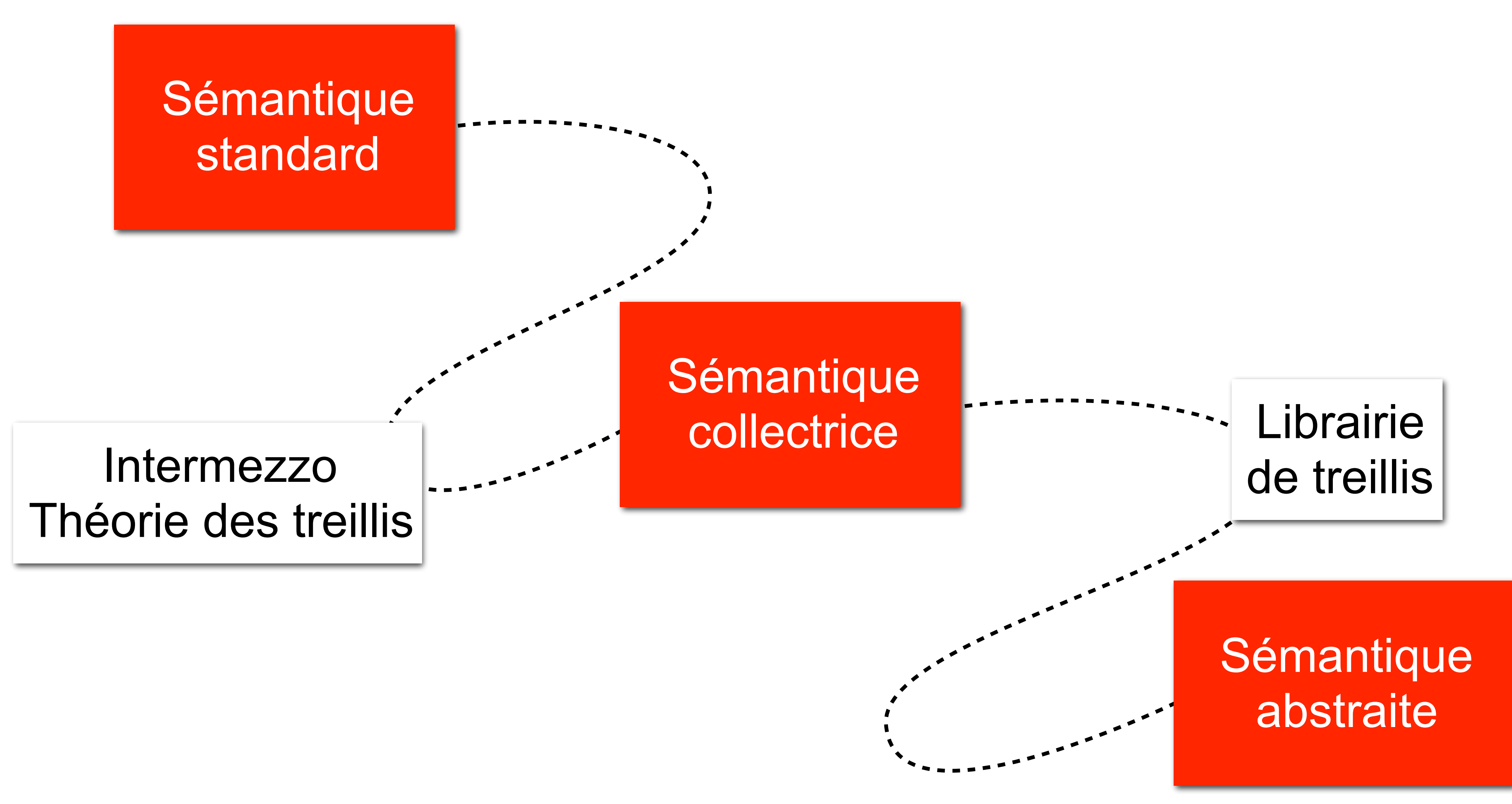
```
Definition reachable_collect (p:program) (s:pp*env) : Prop :=
  let (k,env) := s in
    Collect p (p_instr p) (p_end p) (⊤) k env.

Theorem reachable_sos_implies_reachable_collect : ∀ p s,
  reachable_sos p s -> reachable_collect p s.
```

Sémantique collectrice : correction

```
Definition reachable_collect (p:program) (s:pp*env) : Prop :=
  let (k,env) := s in
    Collect p (p_instr p) (p_end p) (⊤) k env.

Theorem reachable_sos_implies_reachable_collect : ∀ p s,
  reachable_sos p s -> reachable_collect p s.
```

Un théorème qu'on ne prend parfois pas le temps de prouver !

Feuille de route



Treillis abstrait

- On ne peut pas *extraire* la semantique collectrice
  - ➡ elle calcule sur Prop
  - ➡ ce qui explique pourquoi nous avons réussi a définir un calcul de plus petit point fixe pourtant non calculable...

- La sémantique abstraite va calculer sur un treillis $A^\sharp$ au lieu de pp->$\wp$(env)

Treillis abstrait

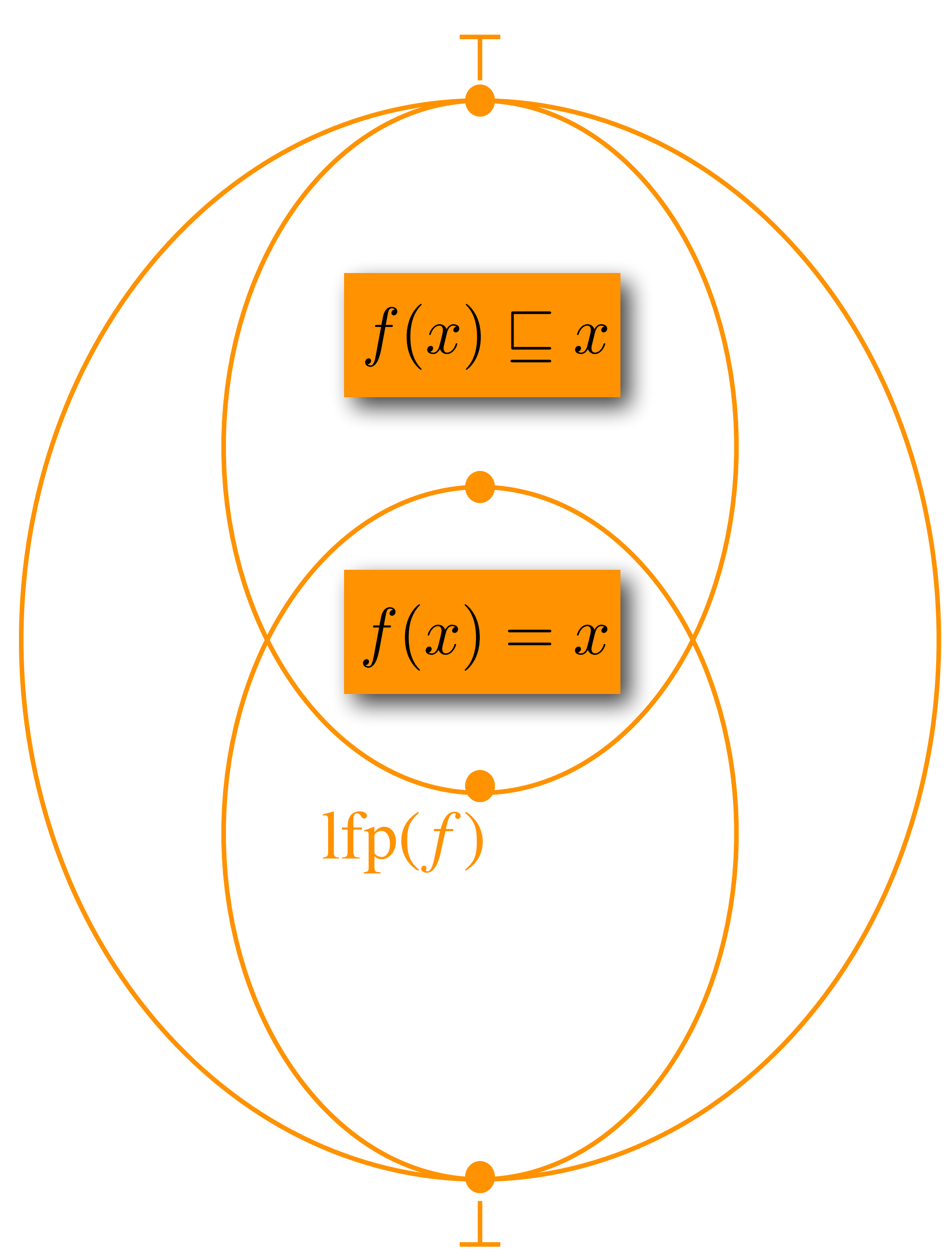
Les treillis abstraits sont formalisés avec des *type classes*

`AbLattice.t :` $\sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \bot^\sharp$ ..., et aussi élargissement/rétrécissement

Chaque treillis est équipée avec un solveur de post point fixe

```
Definition approx_lfp {t} {L:AbLattice.t t} : (t->t) -> t := [...]

Theorem approx_lfp_is_postfixpoint :
  ∀ t (L:AbLattice.t t) (f:t->t),
     f (approx_lfp f) ⊑♯ (approx_lfp f).
Proof. [...] Qed.
```

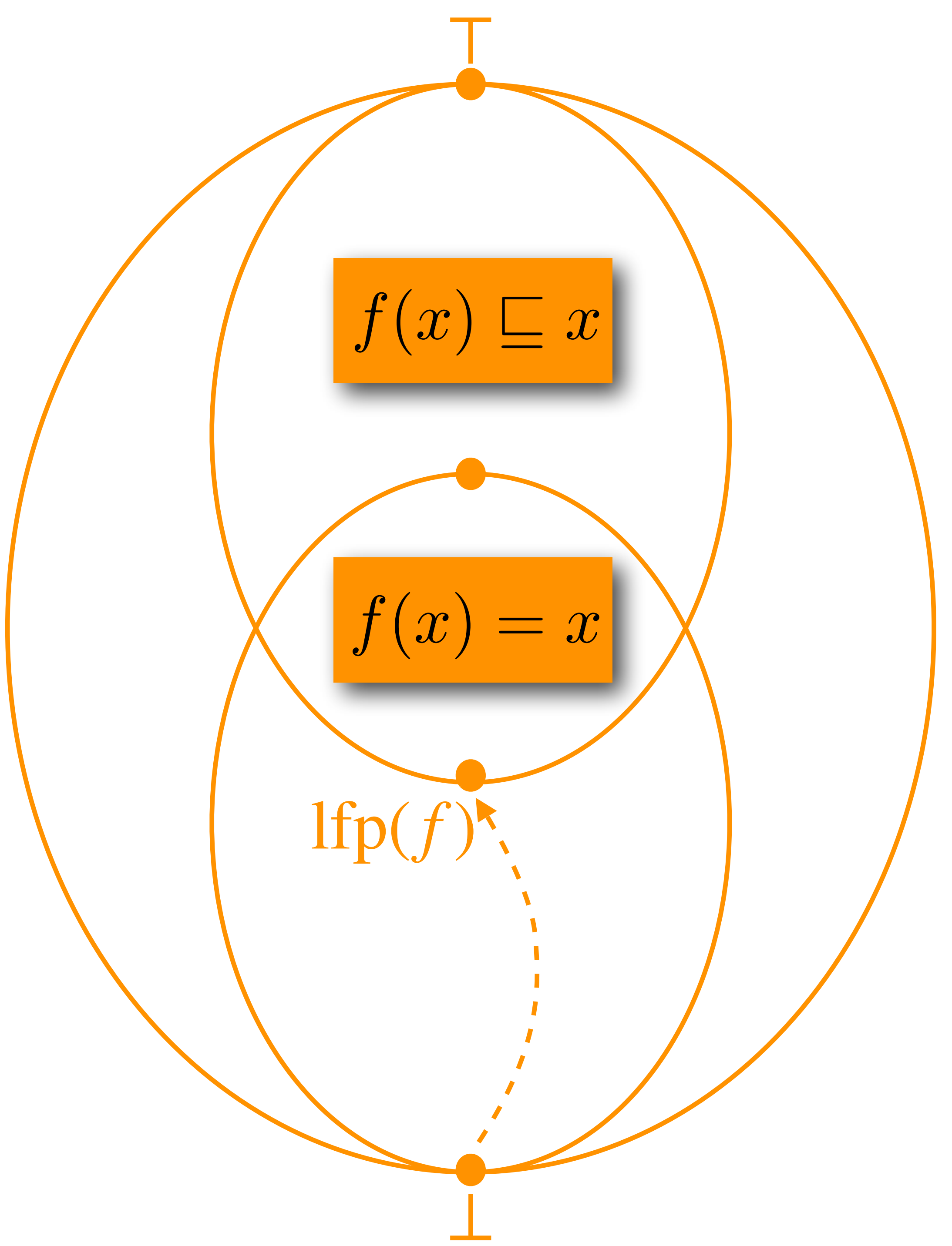
Approximation de points fixes



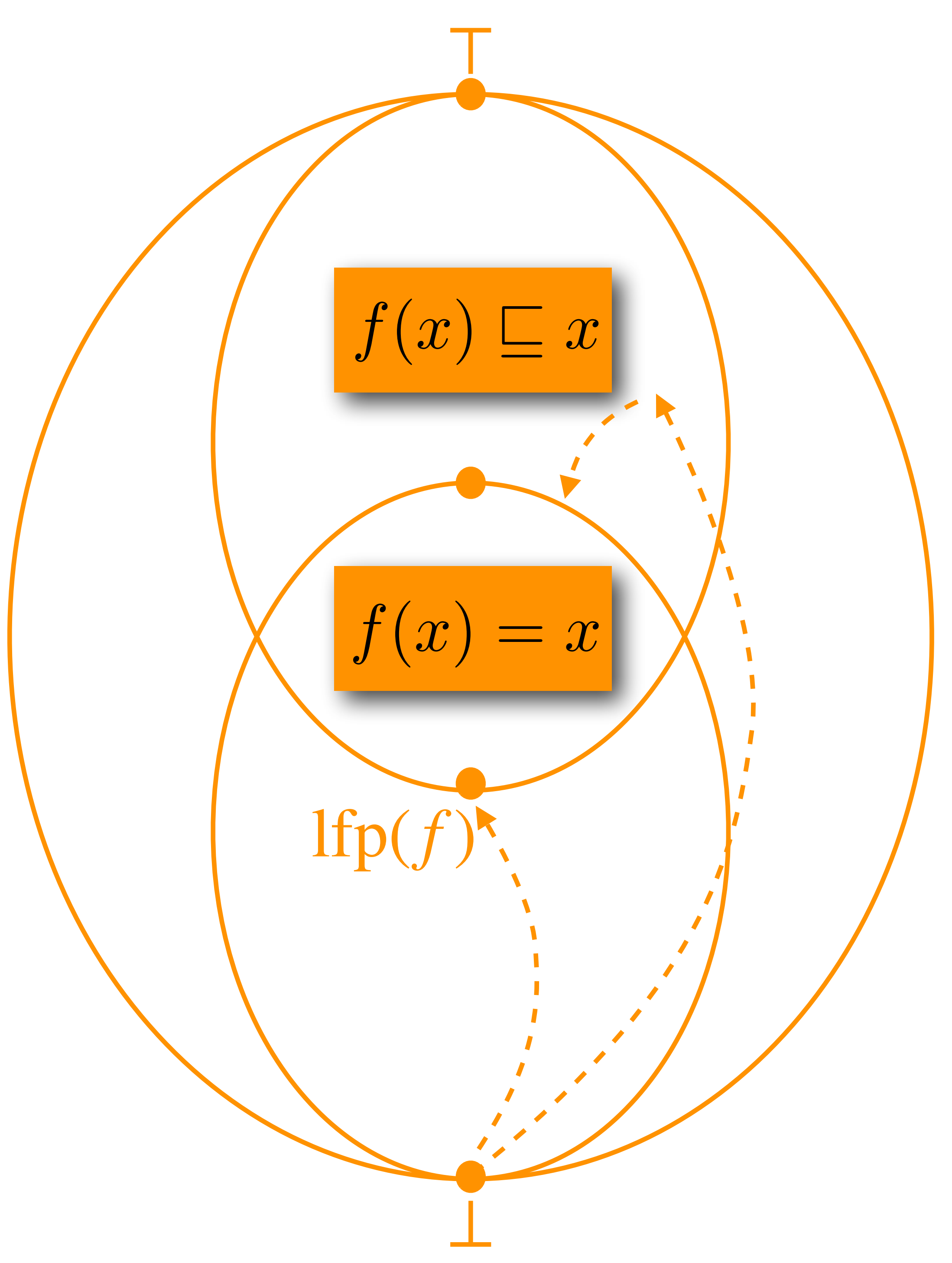
Approximation de points fixes

Théorème de point fixe de Kleene

- convergence trop lente pour les *treillis profonds*
- et parfois pas de convergence !



Approximation de points fixes



Théorème de point fixe de Kleene

- convergence trop lente pour les *treillis profonds*
- et parfois pas de convergence !

Accélération de convergence par élargissement/rétrécissement

- sur-approxime le plus petit point fixe
- nécessite une preuve de terminaison spécifique
- attention a l'ordre d'iteration dans les systèmes a plusieurs variables !

Construire des treillis abstraits

Les treillis sont construits par assemblages *modulaire* grâce à une librairie proposant

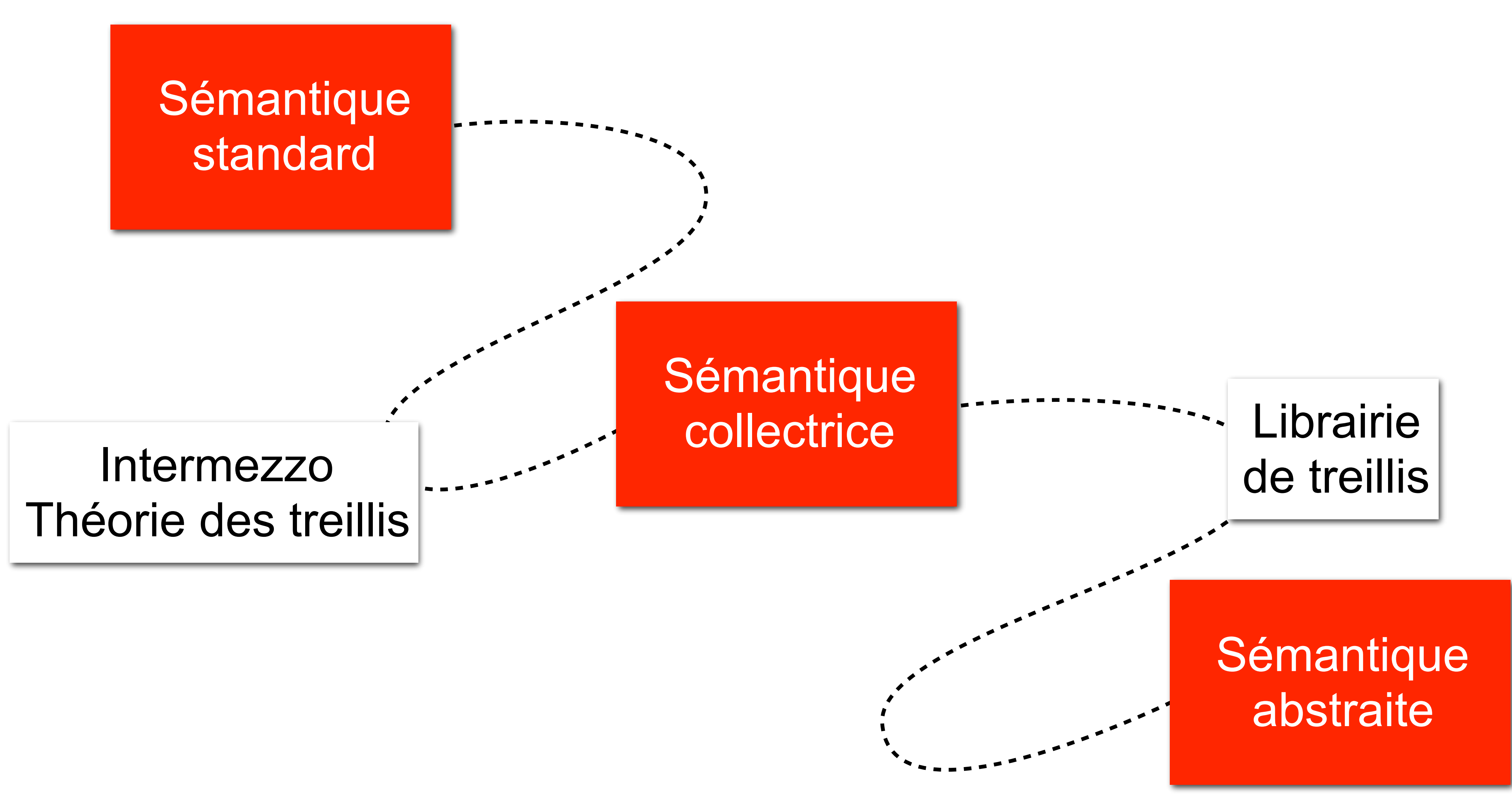
- des briques de base (intervalles, constantes, congruence,...)
- des foncteurs (tableaux, listes, produits, sommes)

Permet une construction *modulaire* des preuves de terminaison de l'analyseur

Exemple

```
Instance ArrayLattice t {L:AbLattice.t t} : AbLattice.t (array A) :=
  [...]
```

Feuille de route



Sémantique abstraite

```
Section prog.
Variable (t:Type) (L:AbLattice.t t) (prog:program) (Ab:AbEnv.t L prog).

Fixpoint AbSem (i:instr) (l2:pp): t -> array t :=
  match i with
    | Skip l1 => fun Pre => ⊥♯ +[l1↦Pre]♯ +[l2↦Pre]♯
    | Assign l1 x e => fun Pre => ⊥♯ +[l1↦Env]♯ +[l2↦AbEnv.assign Pre x e]♯
    | Assert l1 t => fun Pre => ⊥♯ +[p↦Pre]♯ +[l↦AbEnv.assume t Pre]♯
    | If l1 t i1 i2 => fun Pre =>
         let C1 := AbSem i1 l2 (AbEnv.assert t Pre) in
         let C2 := AbSem i2 l2 (AbEnv.assert (Not t) Pre) in
            (C1 ⊔♯ C2) +[l1↦Pre]♯
    | While l1 t i => fun Pre =>
        let I := approx_lfp
                   (fun X => Pre ⊔♯ (get (AbSem i l1 (AbEnv.assume t X)) l1)) in
           (AbSem i l1 (AbEnv.assume t I)) +[l1↦I]♯ +[l2↦AbEnv.assume (Not t) I]♯
    | Seq i1 i2 => fun Pre =>
      let C := (AbSem i1 (first i2) Pre) in
        C ⊔♯ (AbSem i2 l2 (get C (first i2)))
  end.
End prog.
```

Sémantique abstraite *vs* sémantique collectrice

```
Program Fixpoint Collect (i:instr) (l2:pp): monotone (℘(env)) (pp->℘(env)) :=
  match i with
    | Skip l1 =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦Pre]) _
    | Assign l1 x e =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assign x e Pre]) _
    | Assert l1 t =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assume t Pre]) _
    | If l1 t i1 i2 =>
      Mono (fun Pre =>
              let C1 := Collect i1 l1 (assume t Pre) in
              let C2 := Collect i2 l1 (assume (Not t) Pre) in
                (C1 ⊔ C2)+[l1↦Pre]) _
    | While l1 t i =>
      Mono (fun Pre =>
         let F := fun X:℘(env) => Pre ⊔ Collect i l1 (assume t X) l1 in
         let I := lfp F in
                (Collect i l1 (assume t I))+[l1↦I]+[l2↦assume (Not t) I]) _
    | Seq i1 i2 =>
      Mono (fun Pre =>
              let C := Collect i1 (first i2) Pre in
                C ⊔ (Collect i2 l2 (C (firs
  end.
```

```
Section prog.
Variable (t:Type) (L:AbLattice.t t) (prog:program) (Ab:AbEnv.t L prog).

Fixpoint AbSem (i:instr) (l2:pp): t -> array t :=
  match i with
    | Skip l1 => fun Pre => ⊥# +[l1↦Pre]# +[l2↦Pre]#
    | Assign l1 x e => fun Pre => ⊥# +[l1↦Env]# +[l2↦AbEnv.assign Pre x e]#
    | Assert l1 t => fun Pre => ⊥# +[p↦Pre]# +[l↦AbEnv.assume t Pre]#
    | If l1 t i1 i2 => fun Pre =>
        let C1 := AbSem i1 l2 (AbEnv.assert t Pre) in
        let C2 := AbSem i2 l2 (AbEnv.assert (Not t) Pre) in
          (C1 ⊔# C2) +[l1↦Pre]#
    | While l1 t i => fun Pre =>
       let I := approx_lfp
                  (fun X => Pre ⊔# (get (AbSem i l1 (AbEnv.assume t X)) l1)) in
         (AbSem i l1 (AbEnv.assume t I)) +[l1↦I]# +[l2↦AbEnv.assume (Not t) I]#
    | Seq i1 i2 => fun Pre =>
      let C := (AbSem i1 (first i2) Pre) in
        C ⊔# (AbSem i2 l2 (get C (first i2)))
  end.
End prog.
```

Sémantique abstraite *vs* sémantique collectrice

```
Program Fixpoint Collect (i:instr) (l2:pp): monotone (℘(env)) (pp->℘(env)) :=
  match i with
    | Skip l1 =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦Pre]) _
    | Assign l1 x e =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assign x e Pre]) _
    | Assert l1 t =>
      Mono (fun Pre => ⊥+[l1↦Pre]+[l2↦assume t Pre]) _
    | If l1 t i1 i2 =>
      Mono (fun Pre =>
              let C1 := Collect i1 l1 (assume t Pre) in
              let C2 := Collect i2 l1 (assume (Not t) Pre) in
                (C1 ⊔ C2)+[l1↦Pre]) _
    | While l1 t i =>
      Mono (fun Pre =>
         let F := fun X:℘(env) => Pre ⊔ Collect i l1 (assume t X) l1 in
         let I := lfp F in
                (Collect i l1 (assume t I))+[l1↦I]+[l2↦assume (Not t) I]) _
    | Seq i1 i2 =>
      Mono (fun Pre =>
              let C := Collect i1 (first i2) Pre in
                C ⊔ (Collect i2 l2 (C (firs
  end.
```

```
Section prog.
Variable (t:Type) (L:AbLattice.t t) (prog:program) (Ab:AbEnv.t L prog).

Fixpoint AbSem (i:instr) (l2:pp): t -> array t :=
  match i with
    | Skip l1 => fun Pre => ⊥♯ +[l1↦Pre]♯ +[l2↦Pre]♯
    | Assign l1 x e => fun Pre => ⊥♯ +[l1↦Env]♯ +[l2↦AbEnv.assign Pre x e]♯
    | Assert l1 t => fun Pre => ⊥♯ +[p↦Pre]♯ +[l↦AbEnv.assume t Pre]♯
    | If l1 t i1 i2 => fun Pre =>
        let C1 := AbSem i1 l2 (AbEnv.assert t Pre) in
        let C2 := AbSem i2 l2 (AbEnv.assert (Not t) Pre) in
           (C1 ⊔♯ C2) +[l1↦Pre]♯
    | While l1 t i => fun Pre =>
       let I := approx_lfp
                   (fun X => Pre ⊔♯ (get (AbSem i l1 (AbEnv.assume t X)) l1)) in
          (AbSem i l1 (AbEnv.assume t I)) +[l1↦I]♯ +[l2↦AbEnv.assume (Not t) I]♯
    | Seq i1 i2 => fun Pre =>
      let C := (AbSem i1 (first i2) Pre) in
        C ⊔♯ (AbSem i2 l2 (get C (first i2)))
  end.
End prog.
```

Sémantique abstraite : correction

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).
```

Sémantique abstraite : correction

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).
```

```
Definition reachable_collect (p:program) (s:pp*env) : Prop :=
  let (k,env) := s in
    Collect p (p_instr p) (p_end p) (⊤) k env.

Theorem reachable_sos_implies_reachable_collect : ∀ p s,
  reachable_sos p s -> reachable_collect p s.
```

définis précédemment

Sémantique abstraite : correction

```
Theorem AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (γ Env) ⊑ γ (AbSem i l_end Env).


Definition reachable_collect (p:program) (s:pp*env) : Prop :=
  let (k,env) := s in
    Collect p (p_instr p) (p_end p) (⊤) k env.

Theorem reachable_sos_implies_reachable_collect : ∀ p s,
  reachable_sos p s -> reachable_collect p s.


Definition analyse : array t :=
  AbSem prog.(p_instr) prog.(p_end) (AbEnv.top).

Theorem analyse_correct : ∀ k env,
  reachable_sos prog (k,env) -> γ (get analyse k) env.
```

définis précédemment

théorème final

Extraction

L'analyseur obtenu peut être extrait en Ocaml puis exécuté sur des programmes

```
i = 0; k = 0;
          k ∈ [0, 10]   i ∈ [0, 10]
while k < 10 {
          k ∈ [0, 9]   i ∈ [0, 10]
  i = 0;
          k ∈ [0, 9]   i ∈ [0, 10]
  while i < 9 {
          k ∈ [0, 9]   i ∈ [0, 8]
    i = i + 2
  };
          k ∈ [0, 9]   i ∈ [9, 10]
  k = k + 1
}
          k ∈ [10, 10]   i ∈ [0, 10]
```

intervalles

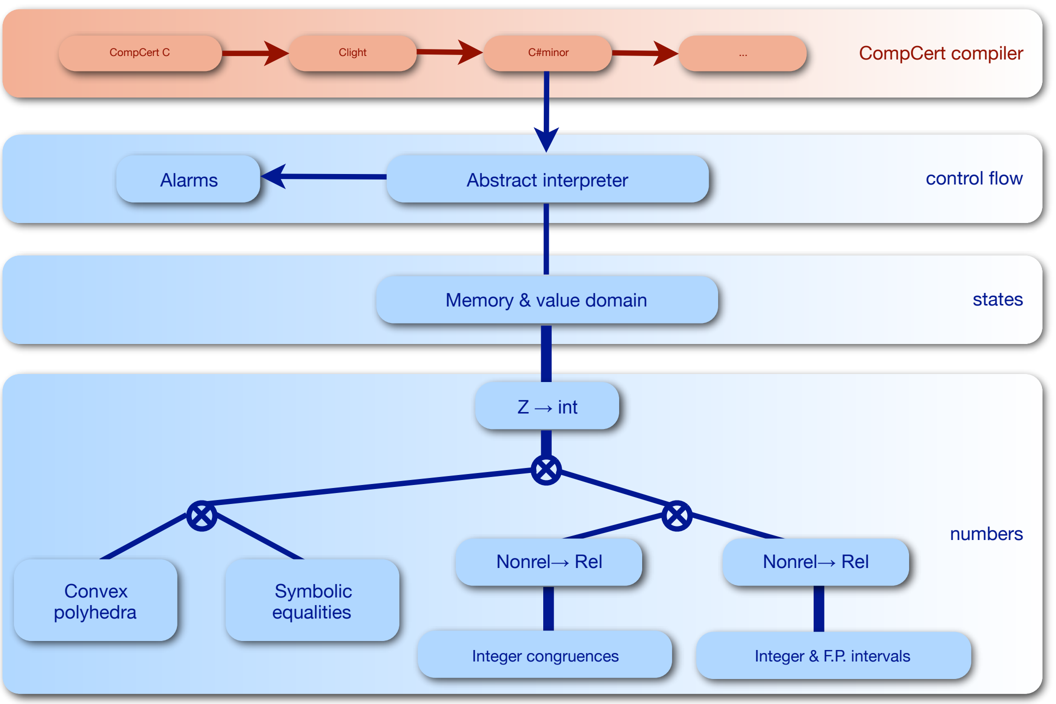
```
i = 0; k = 0;
          k ≥ 0   i ≥ 0
while k < 10 {
          k ≥ 0   i ≥ 0
  i = 0;
          k ≥ 0   i ≥ 0
  while i < 9 {
          k ≥ 0   i ≥ 0
    i = i + 2
  };
          k ≥ 0   i > 0
  k = k + 1
}
          k > 0   i ≥ 0
```

signes

```
i = 0; k = 0;
          i ≡ 0 mod 2
while k < 10 {
          i ≡ 0 mod 2
  i = 0;
          i ≡ 0 mod 2
  while i < 9 {
          i ≡ 0 mod 2
    i = i + 2
  };
          i ≡ 0 mod 2
  k = k + 1
}
          i ≡ 0 mod 2
```

congruences

Un interpréteur abstrait C vérifié : Verasco



Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In 42nd symposium Principles of Programming Languages, pages 247--259. ACM Press, 2015

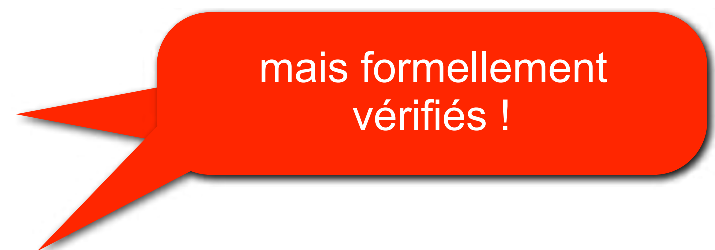
Comment appliquer cette méthodologie pour un langage réaliste ?

VERASCO

http://verasco.imag.fr

- Projet ANR Verasco : 2010-2015
  - ➡ prouver et vérifier en Coq un analyseur *à la Astrée*
  - ➡ en s'appuyant sur le compilateur vérifié CompCert
  - ➡ langage analysé : sous-ensemble CompCert du C
  - ➡ domaines abstraits avancés (relationnels)
  - ➡ architecture modulaire
  - ➡ avec une precision *décente*

- Slogan
  - ➡ si CompCert représente 1/10 de GCC...
  - ➡ ... Verasco est un 1/10 d'Astrée



Modularité

- Astrée possède une architecture très modulaire
  - ➡ programmé en OCaml
  - ➡ et son système de modules
- Verasco suit une architecture proche
  - ➡ programmé en Coq
  - ➡ et son système de *type classes*

Construire un analyseur en OCaml

- Construction modulaire

```
module IntervalAbVal : ABVAL = ...

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

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

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

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

- Exemple d'interface

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

Construire un analyseur en Coq

en OCaml

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

en Coq

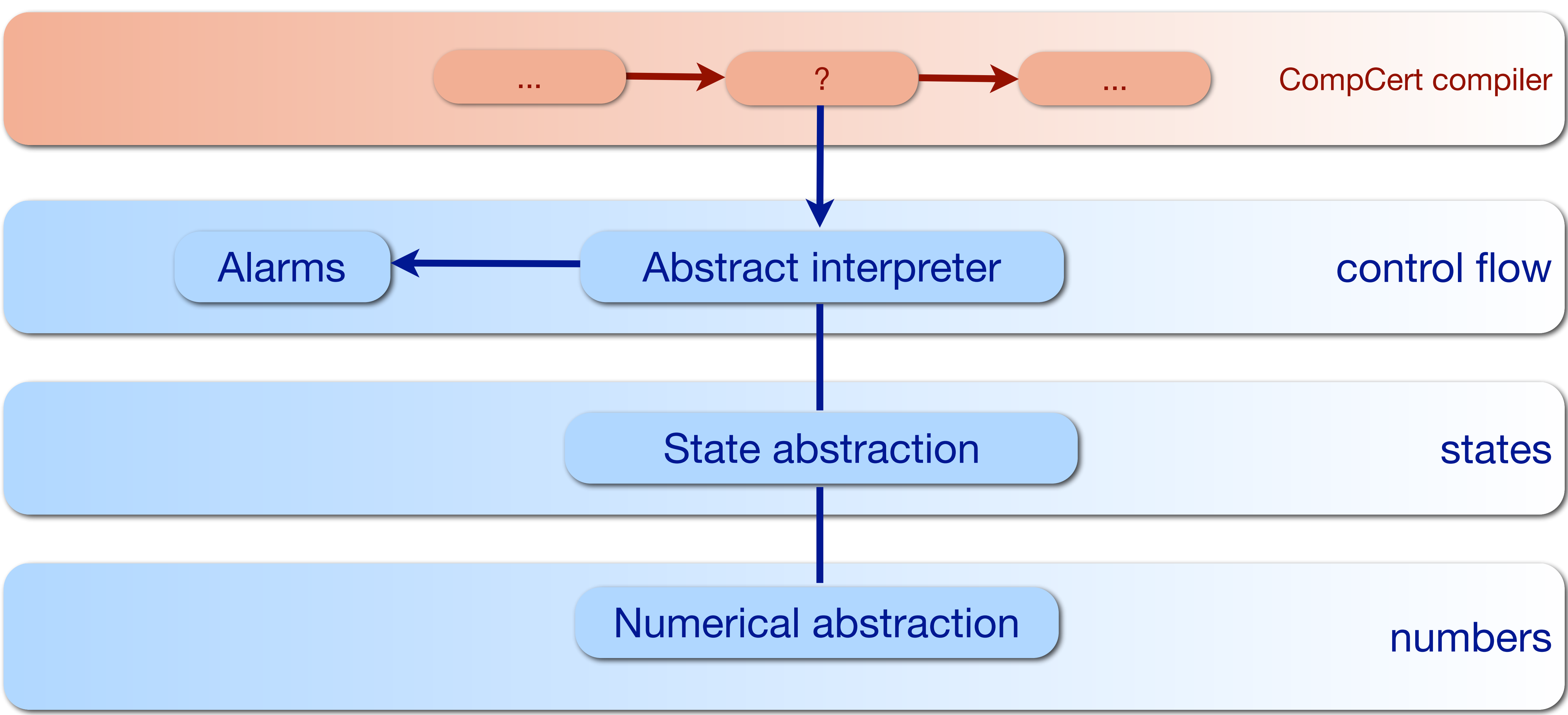
```
Class adom (ab:Type) (c:Type) := {
  le : ab → ab → bool;
  top : ab;
  join : ab → ab → ab;
  widen : ab → ab → ab;

  gamma : ab → ℘(c);

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

Juste prouver le nécessaire...

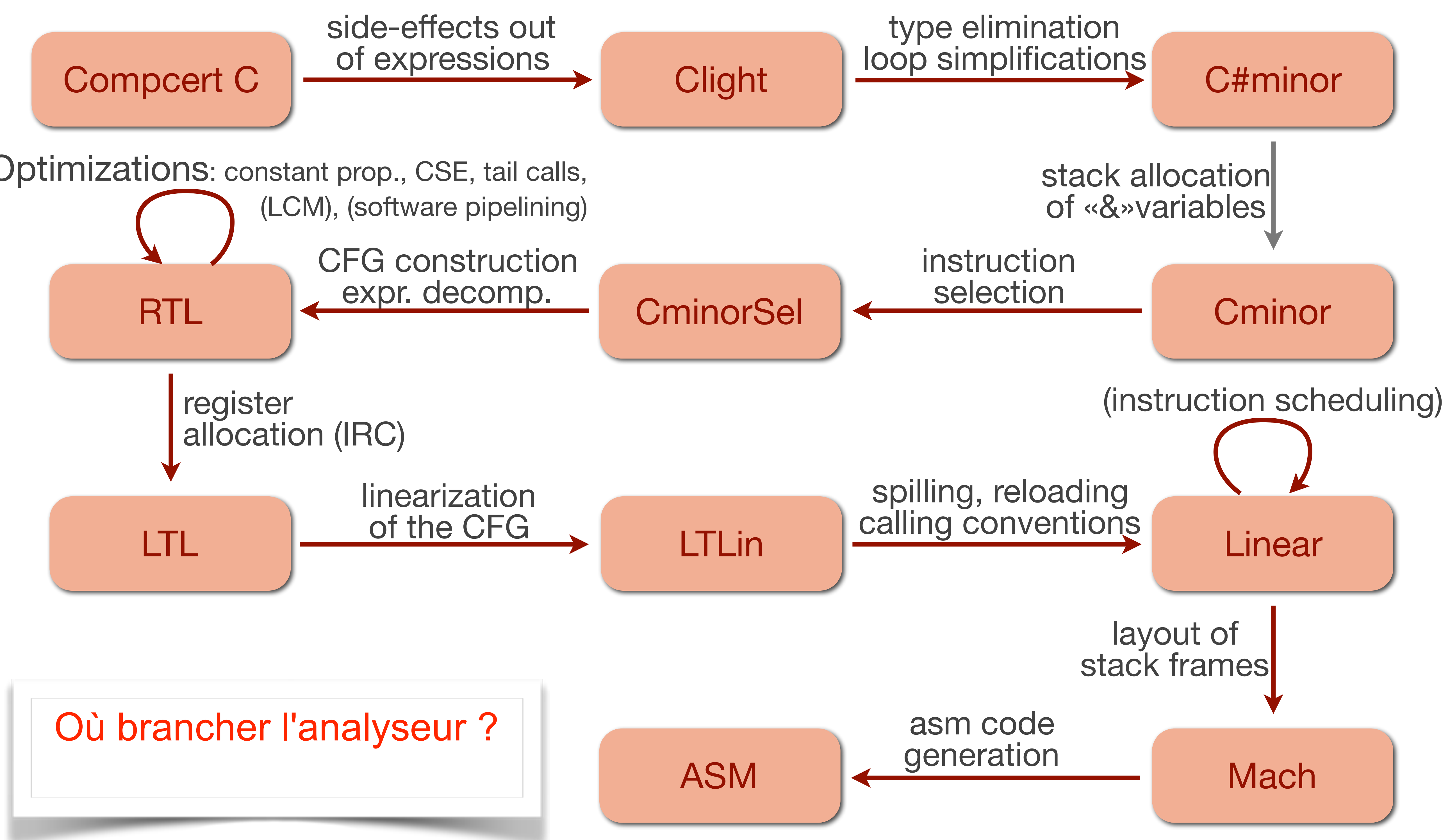
- Preuves *paresseuses*
  - nous ne prouvons que ce qui est vraiment nécessaire pour la sûreté de l'analyseur
- Nous ne prouvons pas
  - la terminaison et la correction des élargissements
  - la qualité des abstractions
    - structure de treillis
    - connexions de Galois

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

  gamma : ab → ℘(c);

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

Architecture



CompCert : 1 compilateur, 11 langages

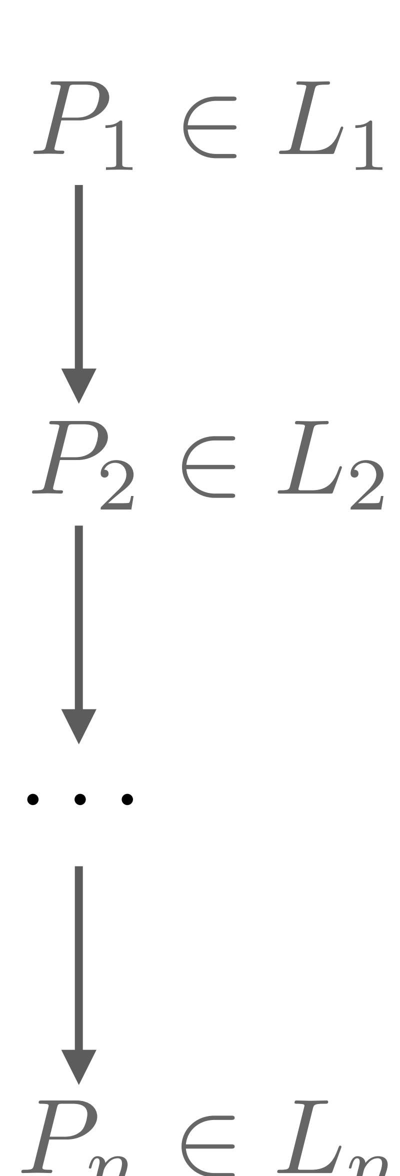


Théorèmes de preservation des comportements

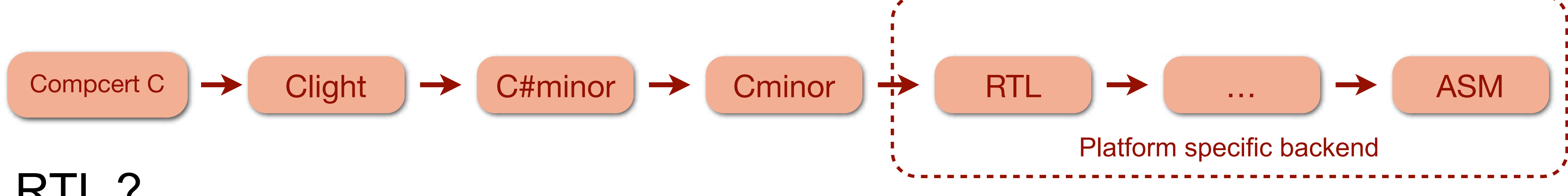
- Théorèmes

$$\forall P_i \; \forall P_{i+1}, \mathcal{C}(P_i) = P_{i+1} \implies \mathbb{B}(P_i + 1) \subseteq \mathbb{B}(P_i)$$

- Corollaires
  - si le programme cible échoue, le programme source échoue
  - un vérificateur sur P_i donne un verdict pertinent sur P_{i+1}, mais pas nécessairement sur P_{i-1}



Choix de la représentation intermédiaire



RTL ?

- la représentation des analyses statiques pour l'optimisation
- mais opérations spécifiques a l'architecture cible et expressions plates

Source C ?

- un langage pour les programmeurs, pas pour les outils

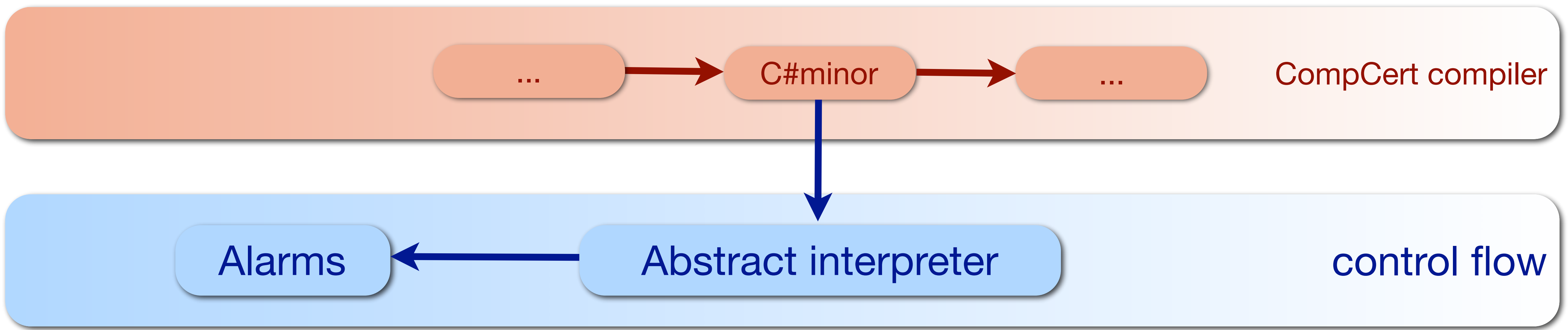
Clight ?

- syntax C sans les effets de bord dans les expressions

C#minor

- très proche de Clight, mais conçu pour les outils

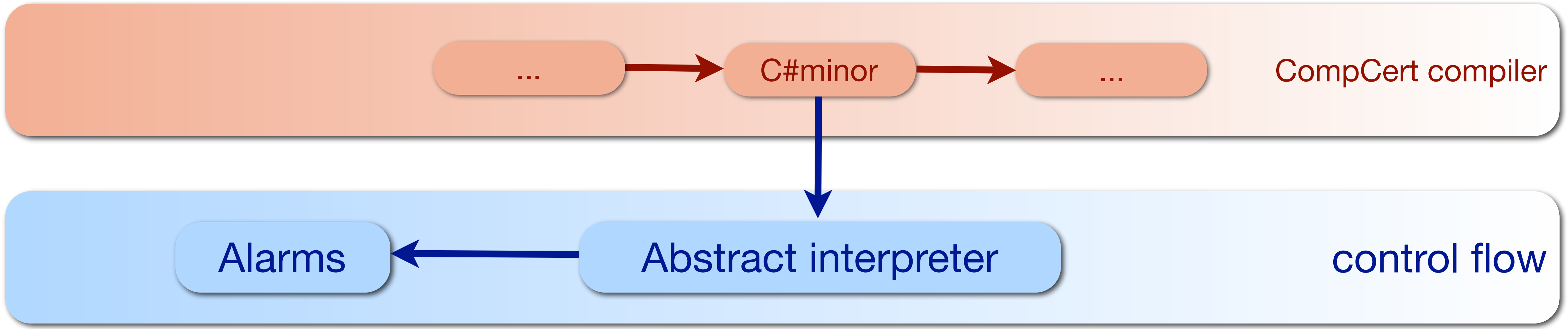
Interpréteur abstrait C#minor



C#minor

- instructions structurées
- exit *n* (pour traduire les break/continue) : saut à la fin du (*n+1)e* bloc englobant
- sauts goto
- variables
  - globales (dont l'adresse peut être manipulée, allouées statiquement)
  - locales (dont l'adresse peut être manipulée, allouées et libérées lors des appels)
  - temporaires (pas en mémoire)

Interpréteur abstrait C#minor

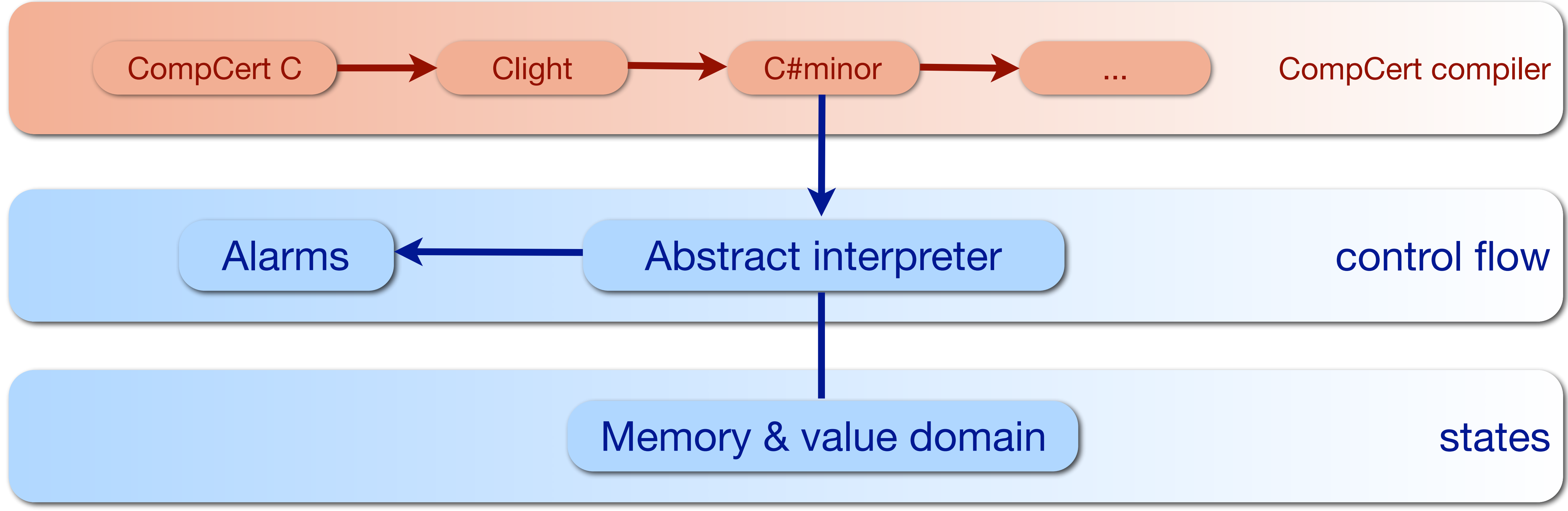


Itération directe sur la syntaxe

- évite de definer les points de programme
- moins gourmand en mémoire
- mais le flot de contrôle est complexe
- résolution locale des points fixes
- un appel de fonction nécessite un appel récursif de l'analyseur
- les gotos nécessitent une resolution de point fixe globale

Paramétré par une abstraction relationnelle des états

Le domaine abstrait mémoire



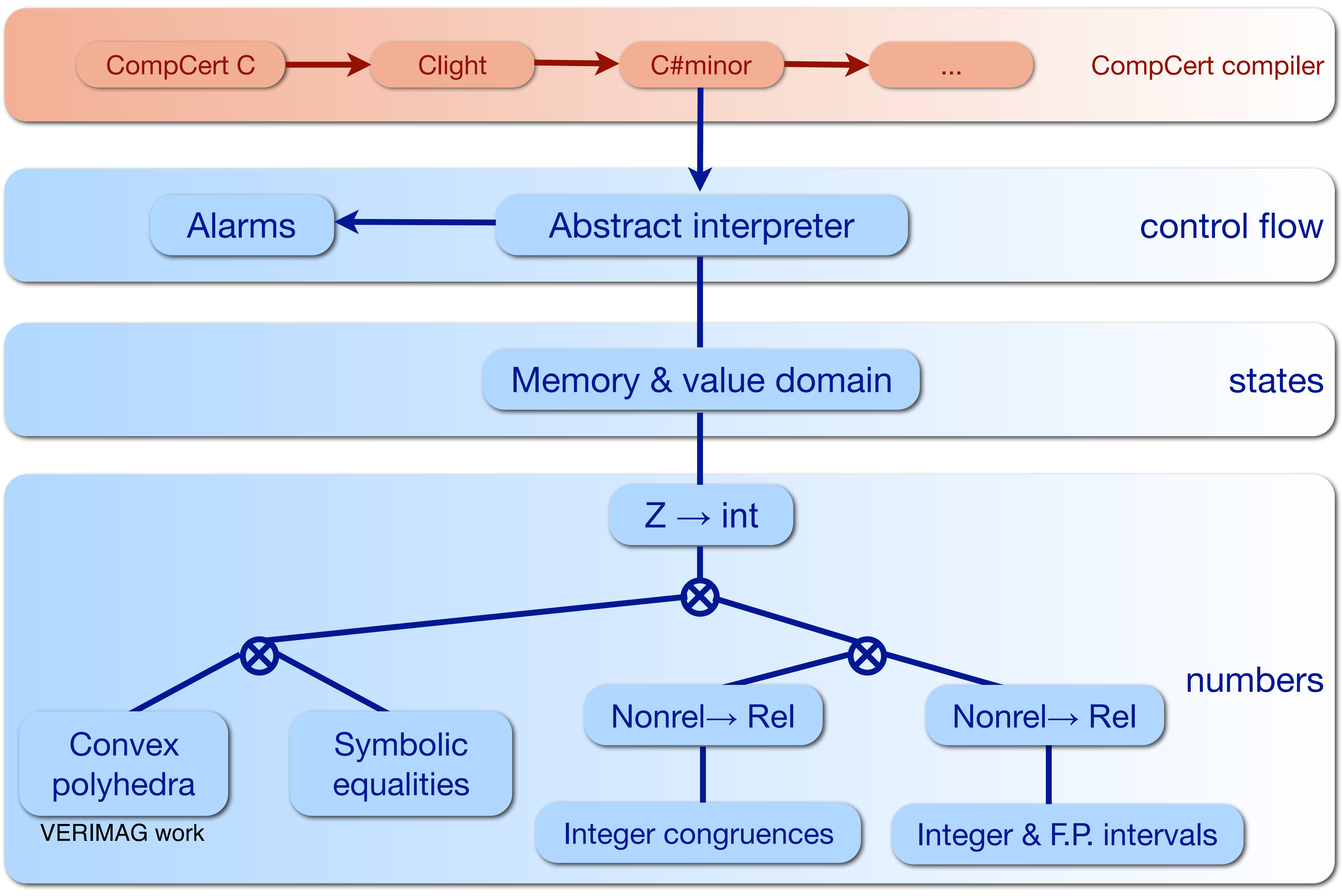
Cellule mémoire = 1 unité de stockage

c ::= temp(f,t) | local(f,x,offset,size) | global(x,offset,size)

Valeur abstraite: (type du contenu, graphe *points-to*, abstraction numérique)

Paramétré par une abstraction numérique relationnelle où les cellules jouent le rôle des variables

Domaines numériques abstraits



Résultats expérimentaux

Analyseur extrait et testé sur des petits programmes C (quelques centaines de lignes de C).

Programmes exerçant des aspects délicats du C : tableaux, arithmétique de pointeurs, pointeurs de fonctions, flottants.

L'analyseur peut parfois prendre plusieurs secondes pour analyser ces programmes.

0 alarmes $\Rightarrow$ preuve formelle automatique des bons comportements d'un programme C !

Conclusion

L'interprétation abstraite vérifiée tient ses promesses

- adaptée à une conception rigoureuse et méthodologique d'un analyseur
- pas seulement pour des analyseurs jouets
- mais il faut parfois faire le tri dans les propriétés a démontrer formellement

L'interprétation abstraite a bien plus de choses à apporter

- formaliser la qualité des abstractions
- dériver systématiquement les fonctions de transferts des analyseurs

L'interaction compilation vérifiée & analyse statique verifiée est prometteuse

- amélioration des optimisations de CompCert
- analyses de sécurité pour les programmes C