A Certified Denotational Abstract Interpreter

(Proof Pearl)

David Pichardie
INRIA Rennes

David Cachera
IRISA / ENS Cachan (Bretagne)

Static Analysis

Static Analysis

Static analysis by abstract interpretation is able

Static Analysis

Static analysis by abstract interpretation is able

- to prove automatically restricted properties
  - like absence of runtime errors

Static Analysis

Static analysis by abstract interpretation is able

- to prove automatically restricted properties
  - like absence of runtime errors
- on large programs

  - Astrée analyses ~1Mloc of a critical software

Static Analysis

Static analysis by abstract interpretation is able

- to prove automatically restricted properties
  - like absence of runtime errors
- on large programs

  - Astrée analyses ~1Mloc of a critical software

But such tools are very complex softwares

- How to establish the soundness of these implementations ?

Certified Static Analysis

Certified Static Analysis

A simple idea:

Certified Static Analysis

A simple idea:

Program and prove your analysis in the same language !

Certified Static Analysis

A simple idea:

Program and prove your analysis in the same language !

Which language ?

Certified Static Analysis

A simple idea:

Program and prove your analysis in the same language !

Which language ?

Program...

Prove...

Extract...

Execute !

This work

We study a small abstract interpreter

- following Cousot’s lecture notes
- represents an embryo of the Astrée analyser

Challenges

- be able to follow the textbook approach without remodeling the algorithms and the proofs
- first machine-checked instance of the motto
  « my abstract interpreter is correct by construction »

Language Syntax

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

Language Syntax

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

Instructions are labelled (program points)

Language Syntax

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

Record program := {
  p_stmt: stmt;
  p_end: pp;
  vars: list var
}.
```

Language Syntax

binary numbers with at most 32 bits (useful to prove termination)

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

Record program := {
  p_stmt: stmt;
  p_end: pp;
  vars: list var
}.
```

Language Syntax

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

```
Record program := {
  p_stmt: stmt;
  p_end: pp;
  vars: list var
}.
```

main statement

Language Syntax

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

```
Record program := {
  p_stmt: stmt;
  p_end: pp;
  vars: list var
}.
```

last label

Language Syntax

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

Record program := {
  p_stmt: stmt;
  p_end: pp;
  vars: list var
}.
```

variable declaration

Language Semantics

Semantic Domains

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

Language Semantics

Semantic Domains

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

Structural Operational Semantics

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ 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)
[...]
```

Language Semantics

Semantic Domains

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

Structural Operational Semantics

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ 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)
[...]
```

$$\frac{\texttt{sem\_expr}\ p\ \rho_1\ e\ n \quad \rho_2 = \rho_1[x \mapsto n] \quad x \in (\texttt{vars}\ p)}{\texttt{sos}\ p\ (\texttt{Assign}\ l\ x\ e, \rho_1)\ (\texttt{Final}\ \rho_2)}$$

Language Semantics

Semantic Domains

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

Structural Operational Semantics

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ 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)
[...]
```

$$\frac{\texttt{sem\_expr}\ p\ \rho_1\ e\ n \quad \rho_2 = \rho_1[x \mapsto n] \quad x \in (\texttt{vars}\ p)}{\texttt{sos}\ p\ (\texttt{Assign}\ l\ x\ e, \rho_1)\ (\texttt{Final}\ \rho_2)}$$

Language Semantics

Semantic Domains

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

Structural Operational Semantics

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ 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)
[...]
```

$$\frac{\texttt{sem\_expr}\ p\ \rho_1\ e\ n \qquad \rho_2 = \rho_1[x \mapsto n] \qquad x \in (\texttt{vars}\ p)}{\texttt{sos}\ p\ (\texttt{Assign}\ l\ x\ e, \rho_1)\ (\texttt{Final}\ \rho_2)}$$

Language Semantics

Semantic Domains

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

Structural Operational Semantics

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ 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)
[...]
```

$$\frac{\texttt{sem\_expr}\ p\ \rho_1\ e\ n \quad \rho_2 = \rho_1[x \mapsto n] \quad x \in (\texttt{vars}\ p)}{\texttt{sos}\ p\ (\texttt{Assign}\ l\ x\ e, \rho_1)\ (\texttt{Final}\ \rho_2)}$$

Language Semantics

Semantic Domains

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

Structural Operational Semantics

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ 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)
[...]
```

$$\frac{\texttt{sem\_expr}\ p\ \rho_1\ e\ n \quad \rho_2 = \rho_1[x \mapsto n] \quad x \in (\texttt{vars}\ p)}{\texttt{sos}\ p\ (\texttt{Assign}\ l\ x\ e, \rho_1)\ (\texttt{Final}\ \rho_2)}$$

Language Semantics

Semantic Domains

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

Structural Operational Semantics

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ 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)
[...]
```

Reachable states from any initial environment

```
Inductive reachable_sos (p:program) : pp*env → Prop
  := [...]
```

Final Objective for today

Final Objective for today

The analyzer computes an abstract representation of the program semantics

```
Definition analyse : program → abdom := [...]
```

Final Objective for today

The analyzer computes an abstract representation of the program semantics

```
Definition analyse : program → abdom := [...]
```

Each abstract element is given a concretization in $\mathcal{P}(\mathrm{pp} \times \mathrm{env})$

```
Definition γ : abdom → (pp*env → Prop) := [...]
```

Final Objective for today

The analyzer computes an abstract representation of the program semantics

```
Definition analyse : program → abdom := [...]
```

Each abstract element is given a concretization in $\mathcal{P}(\mathrm{pp} \times \mathrm{env})$

```
Definition γ : abdom → (pp*env → Prop) := [...]
```

The analyzer must compute a correct over-approximation of the reachable states

```
Theorem analyse_correct : ∀ prog:program,
  reachable_sos prog ⊆ γ (analyse prog).
```

Roadmap

Standard Semantics

Abstract Semantics

Roadmap

Standard Semantics

direct soundness proof →

Abstract Semantics

Previous works:

- Y. Bertot. *Structural abstract interpretation, a formal study in Coq*. ALFA Summer School 2008
- X. Leroy. *Mechanized semantics, with applications to program proof and compiler verification.* Marktoberdorf Summer School 2009

Roadmap

Previous works:

Y. Bertot. *Structural abstract interpretation, a formal study in Coq*. ALFA Summer School 2008

X. Leroy. *Mechanized semantics, with applications to program proof and compiler verification.* Marktoberdorf Summer School 2009

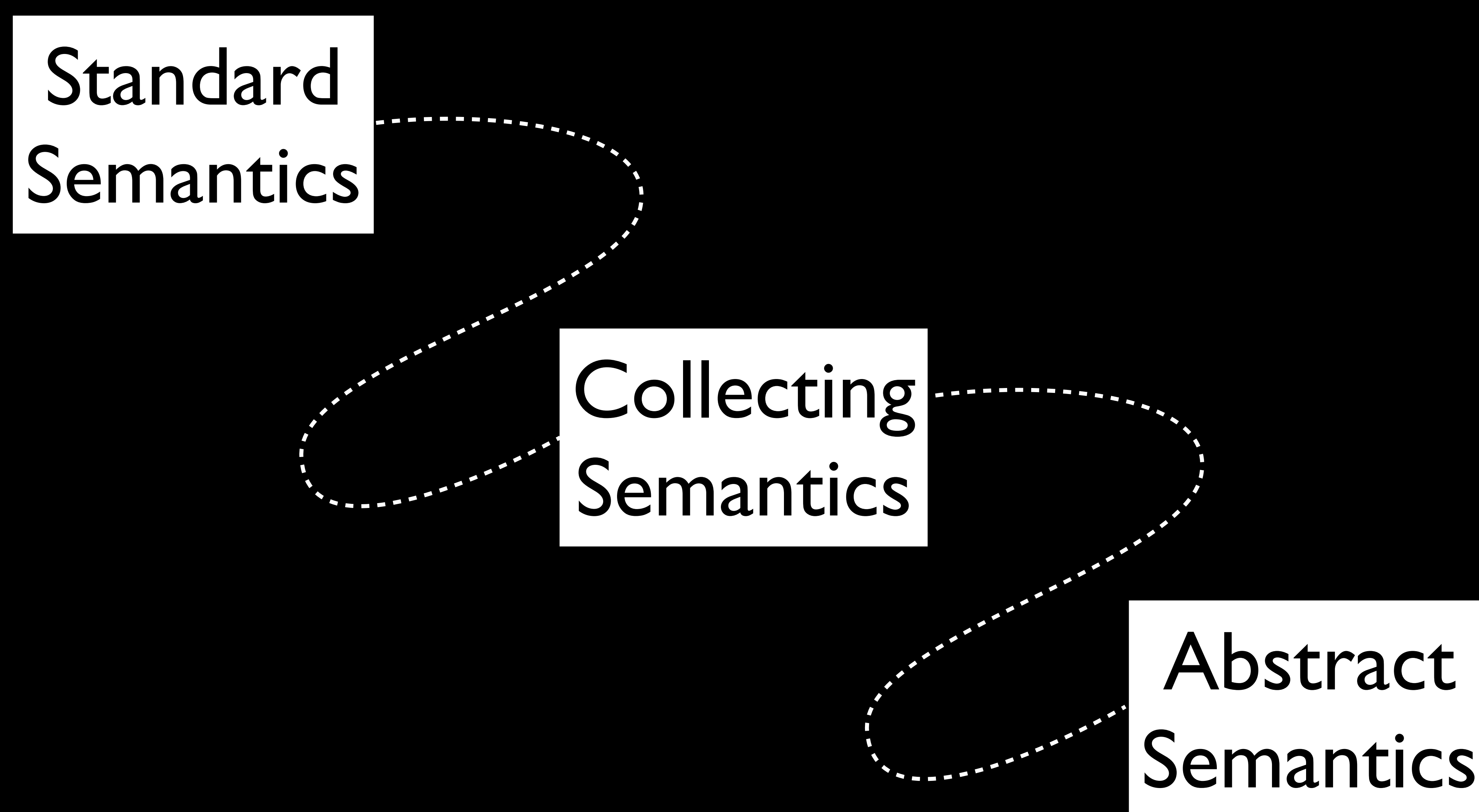
Roadmap

Standard Semantics

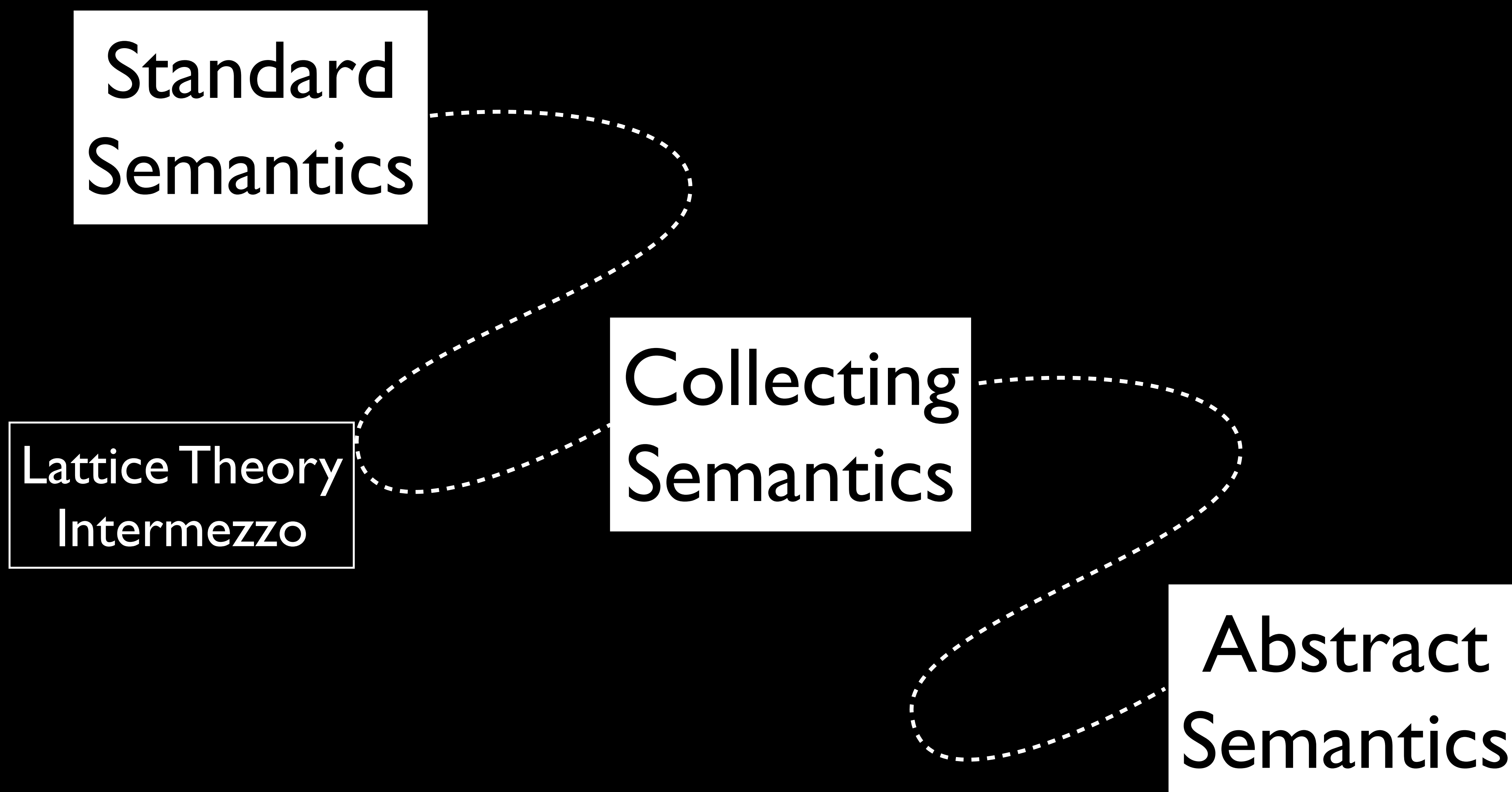
Collecting Semantics

Abstract Semantics

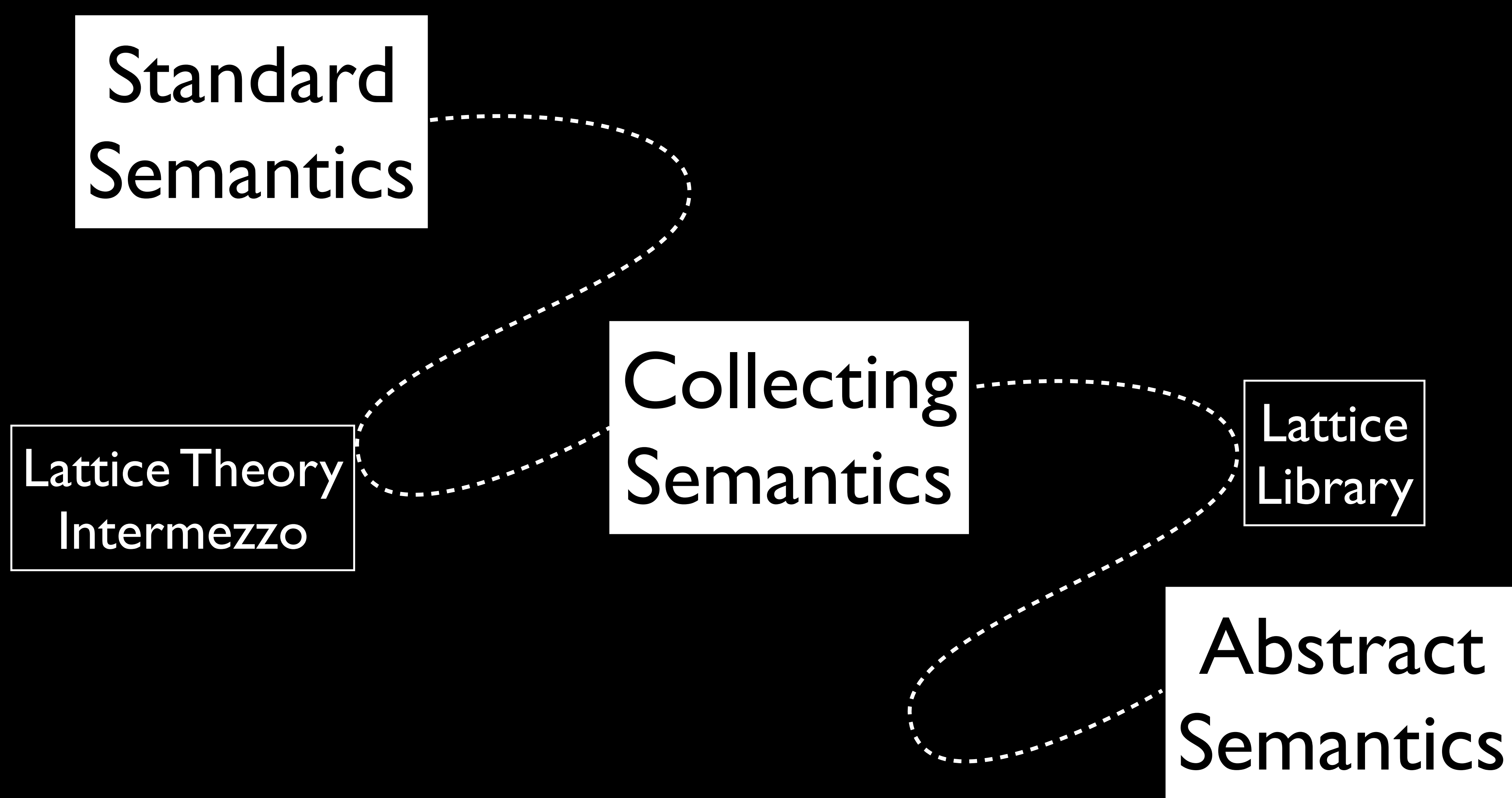
Roadmap


Standard Semantics
Collecting Semantics
Abstract Semantics

Roadmap


Standard Semantics
Lattice Theory Intermezzo
Collecting Semantics
Abstract Semantics

Roadmap



Standard Semantics
Lattice Theory Intermezzo
Collecting Semantics
Lattice Library
Abstract Semantics

Lattice Theory Intermezzo

A Few Lattice Theory

We need a least-fixpoint operator in Coq

- Formalization of complete lattices
- Proof of Knaster-Tarski theorem
- Construction of some useful complete lattices

Knaster-Tarski Theorem

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

Knaster-Tarski Theorem

Complete lattices on elements of type A

Monotone functions from L to L

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

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

Monotone functions

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

A monotone function is a term
(Mono f π)

Knaster-Tarski Theorem

Complete lattices on elements of type A

Monotone functions from L to L

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

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

Knaster-Tarski Theorem

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

Section KnasterTarski.
  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. [...]
End KnasterTarski.
```

Knaster-Tarski Theorem

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

Section KnasterTarski.
  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. [...]
End KnasterTarski.
```

Coq Type Classes = Record + Inference (super) capabilities

Knaster-Tarski Theorem

We declare this argument as implicit

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

Section KnasterTars
  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 (lf
  Lemma lfp_least_postfixpoint :                    ⊑ x. [...]
End KnasterTarski.
```

Coq Type Classes = Record + Inference (super) capabilities

The implicit argument of type `(CompleteLattice.t L)` is automatically inferred

Canonical Complete Lattices

```
Instance PowerSetCL A : CompleteLattice.t 𝒫(A) := [...]


Instance PointwiseCL A L {CompleteLattice.t L} :
                        CompleteLattice.t (A → L) := [...]
```

Canonical Complete Lattices

Notation for (A->Prop)

```
Instance PowerSetCL A : CompleteLattice.t 𝒫(A) := [...]
```

```
Instance PointwiseCL A L {CompleteLattice.t L} :
                        CompleteLattice.t (A → L) := [...]
```

Canonical Complete Lattices

Notation for (A->Prop)

```
Instance PowerSetCL A : CompleteLattice.t 𝒫(A) := [...]
```

Set inclusion ordering

```
Instance PointwiseCL A L {CompleteLattice.t L} :
                        CompleteLattice.t (A → L) := [...]
```

Canonical Complete Lattices

Notation for `(A->Prop)`

```
Instance PowerSetCL A : CompleteLattice.t 𝒫(A) := [...]
```

Set inclusion ordering

```
Instance PointwiseCL A L {CompleteLattice.t L} :
                        CompleteLattice.t (A → L) := [...]
```

Functor

Pointwise ordering

Canonical Complete Lattices

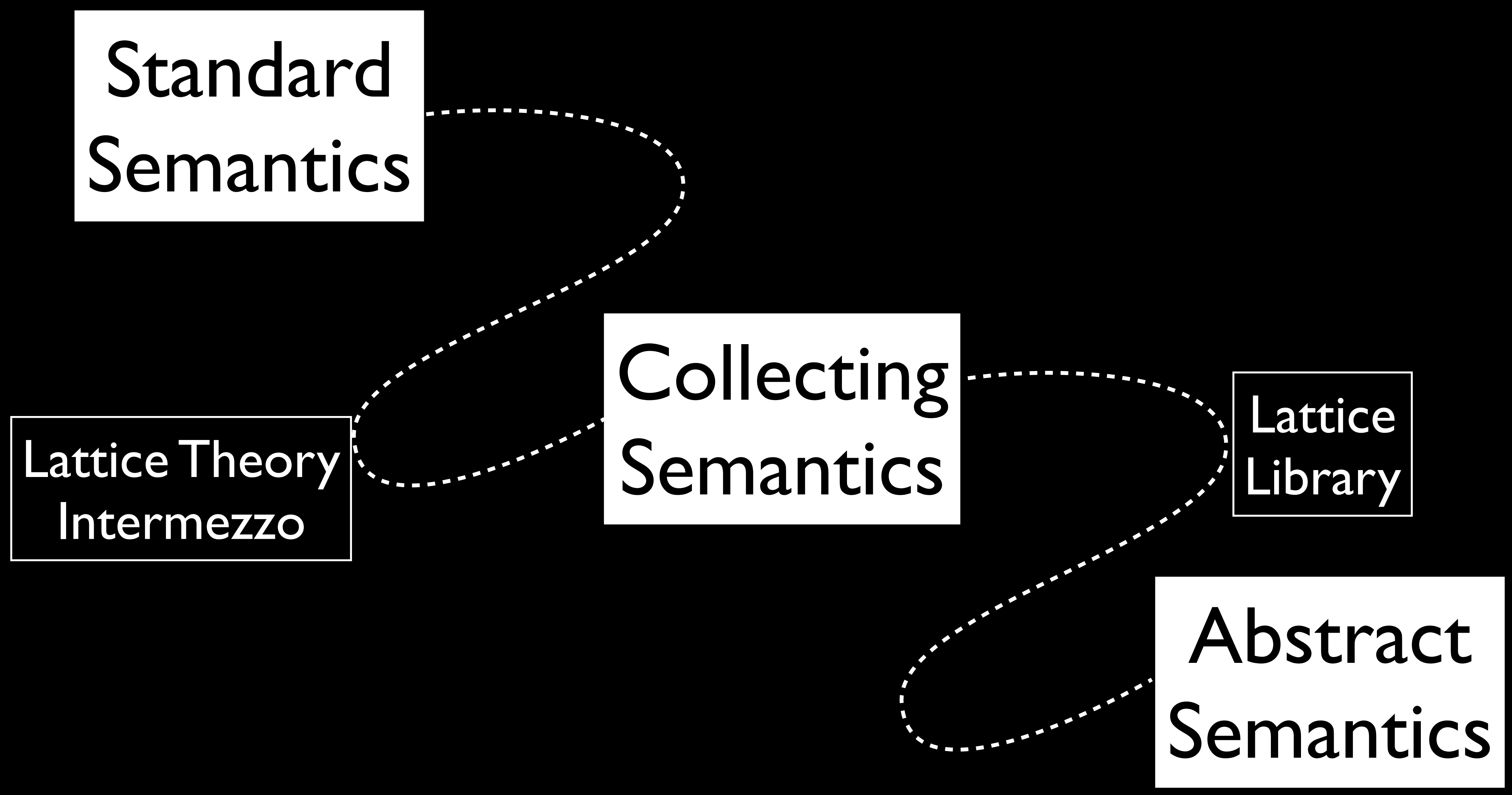
```
Instance PowerSetCL A : CompleteLattice.t 𝒫(A) := [...]

Instance PointwiseCL A L {CompleteLattice.t L} :
                        CompleteLattice.t (A → L) := [...]

Definition example (f:monotone (B → 𝒫(C)) (B → 𝒫(C))) :=
  lfp f.
```

The right complete lattice is automatically inferred

Roadmap


Standard Semantics
Lattice Theory Intermezzo
Collecting Semantics
Lattice Library
Abstract Semantics

Collecting Semantics

- An important component in the Abstract Interpretation framework
- Mimics the behavior of the static analysis (fixpoint iteration)
- But still in the concrete domain
- Similar to a denotational semantics but operates on $\wp(State)$ instead of $State_{\perp}$

Collecting Semantics: Example

```
i = 0; k = 0;

while  k < 10  {

   i = 0 ;

  while  i < 9  {

     i = i + 2
  };

   k = k + 1
}
```

Collecting Semantics: Example

```
i = 0; k = 0;

while [k < 10]^l1 {

   [i = 0]^l2;

   while [i < 9]^l3 {

      [i = i + 2]^l4
   };

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

Collecting Semantics: Example

```
i = 0; k = 0;

while [k < 10]^l1 {

   [i = 0]^l2;

   while [i < 9]^l3 {

      [i = i + 2]^l4
   };

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

$$l_1 \mapsto [0, 10] \times ([0, 10] \cap \mathrm{Even})$$

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

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

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

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

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

Collecting Semantics

precondition

`Collect (i:stmt) (l:pp) :` $\mathcal{P}$`(env)` $\rightarrow$ `(pp` $\rightarrow \mathcal{P}$`(env))`

label after i

invariants on each reachable states during execution of i

Collecting Semantics

$$\texttt{Collect (i:stmt) (l:pp) : monotone (}\mathcal{P}\texttt{(env)) (pp} \rightarrow \mathcal{P}\texttt{(env))}$$

We generate only
monotone operators

Collecting Semantics

`Collect (i:stmt) (l:pp) : monotone` $(\mathcal{P}$`(env))` `(pp` $\rightarrow \mathcal{P}$`(env))`

Final instanciation:

`Collect p.(p_stmt) p.(p_end)` $\top$ `: (pp` $\rightarrow \mathcal{P}$`(env))`

invariants on each reachable states

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                             monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e =>
    Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

  | While p t i =>
    [...]




  | [...]
end.
```

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                           monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e =>
    Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

  | While p t i =>
    [...]

  | [...]
end.
```

map substitution + union

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                          monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e =>
    Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

  | While p t i =>
    [...]

  | [...]
end.
```

map substitution + union

Strongest post-condition assignment transformer

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                              monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
   | Assign p x e =>
     Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

   | While p t i =>
     Mono (fun Env =>
          let I:𝒫(env) := lfp      ?                        in
            (Collect i p (assert t I))
                 +[p ↦ I] +[l ↦ assert (Not t) I]) _

   | [...]
end.
```

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                          monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  |
  | While p t i =>
    Mono (fun Env =>
         let I:𝒫(env) := lfp      ?                            in
           (Collect i p (assert t I))
                +[p ↦ I] +[l ↦ assert (Not t) I]) _

  | [...]
end.
```

Fixpoint equation: `I == Env ⊔ (Collect i p (assert t I) p)`

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                            monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  |
  | While p t i =>
    Mono (fun Env =>
          let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
            (Collect i p (assert t I))
                 +[p ↦ I] +[l ↦ assert (Not t) I]) _

  | [...]
end.
```

Fixpoint equation: `I == Env ⊔ (Collect i p (assert t I) p)`

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                            monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e =>
    Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

  | While p t i =>
    Mono (fun Env =>
          let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
            (Collect i p (assert t I))
                 +[p ↦ I] +[l ↦ assert (Not t) I]) _

  | [...]
end.
```

must be monotone

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                            monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Assign p x e =>
    Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

  | While p t i =>
    Mono (fun Env =>
          let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
            (Collect i p (assert t I))
                  +[p ↦ I] +[l ↦ assert (Not t) I]) _

  | [...]
end.
```

proof obligation

must be monotone

proof obligation

Collecting Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp):
                          monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
   | Assign p x e =>
     Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _

   | While p t i =>
     Mono (fun Env =>
           let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
             (Collect i p (assert t I))
                  +[p ↦ I] +[l ↦ assert (Not t) I]) _

   | [...]
end.
```

proof obligation

must be monotone

proof obligation

Proof obligations are generated by the `Program` mechanism and then automatically discharged by a custom tactic for monotonicity proofs

Collecting Semantics

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

Theorem reachable_sos_implies_reachable_collect :
  ∀ p, reachable_sos p ⊆ reachable_collect p.
```

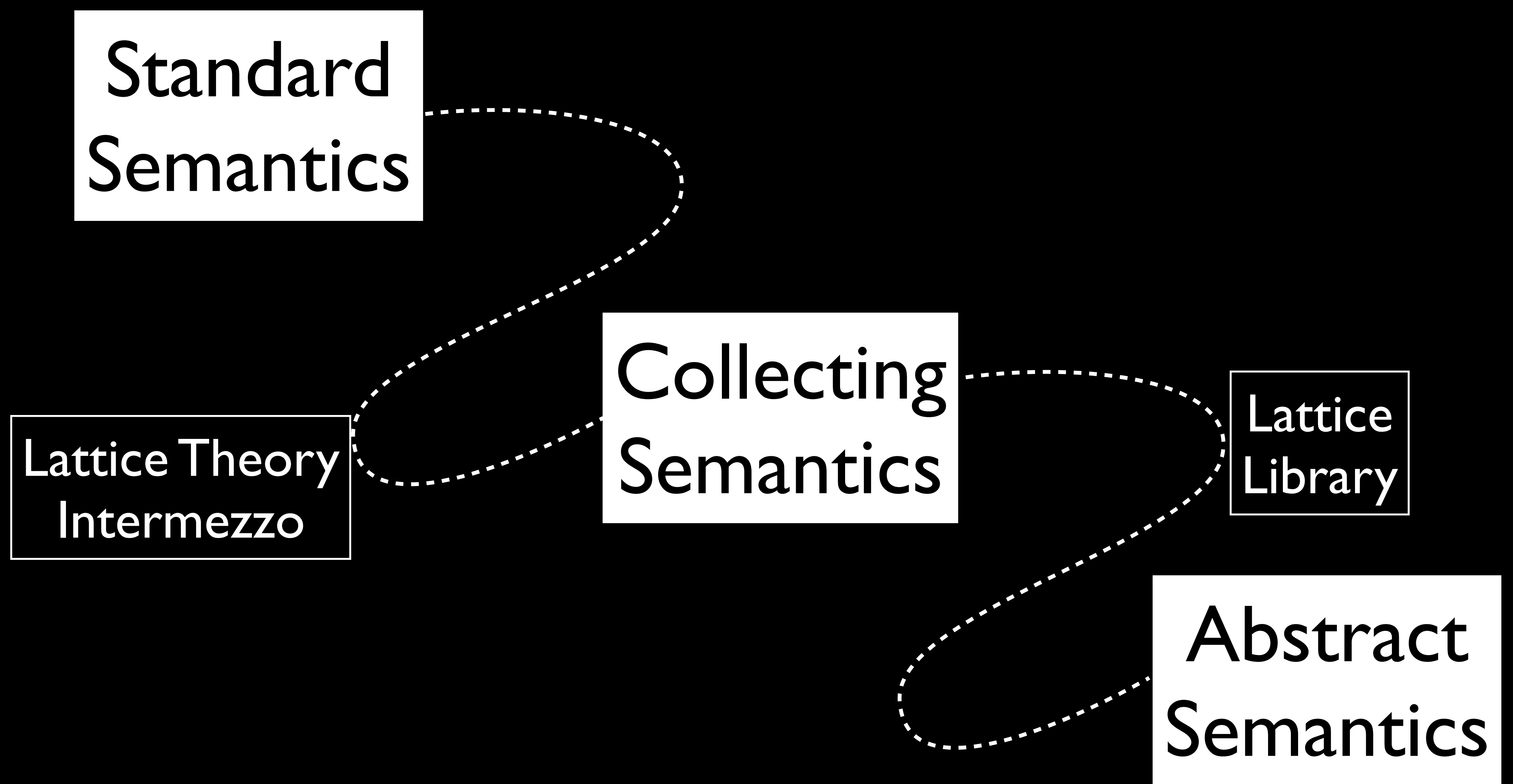
Collecting Semantics

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

Theorem reachable_sos_implies_reachable_collect :
  ∀ p, reachable_sos p ⊆ reachable_collect p.
```

This is the most difficult proof of this work. It is sometimes just skipped in the AI literature because people *start* from a collecting semantics.

Roadmap


Standard Semantics
Lattice Theory Intermezzo
Collecting Semantics
Lattice Library
Abstract Semantics

Abstract Lattices

- Nothing can be extracted from the collecting semantics
  - it operates on Prop
  - that's why we were able to *program* the *not-so-constructive* lfp operator in Coq
- The abstract semantics will not computes on $\texttt{(pp} \to \mathcal{P}\texttt{(env))}$ but on an *abstract lattice* $\texttt{A}^\sharp$

Abstract Lattice

Abstract lattices are formalized with type classes

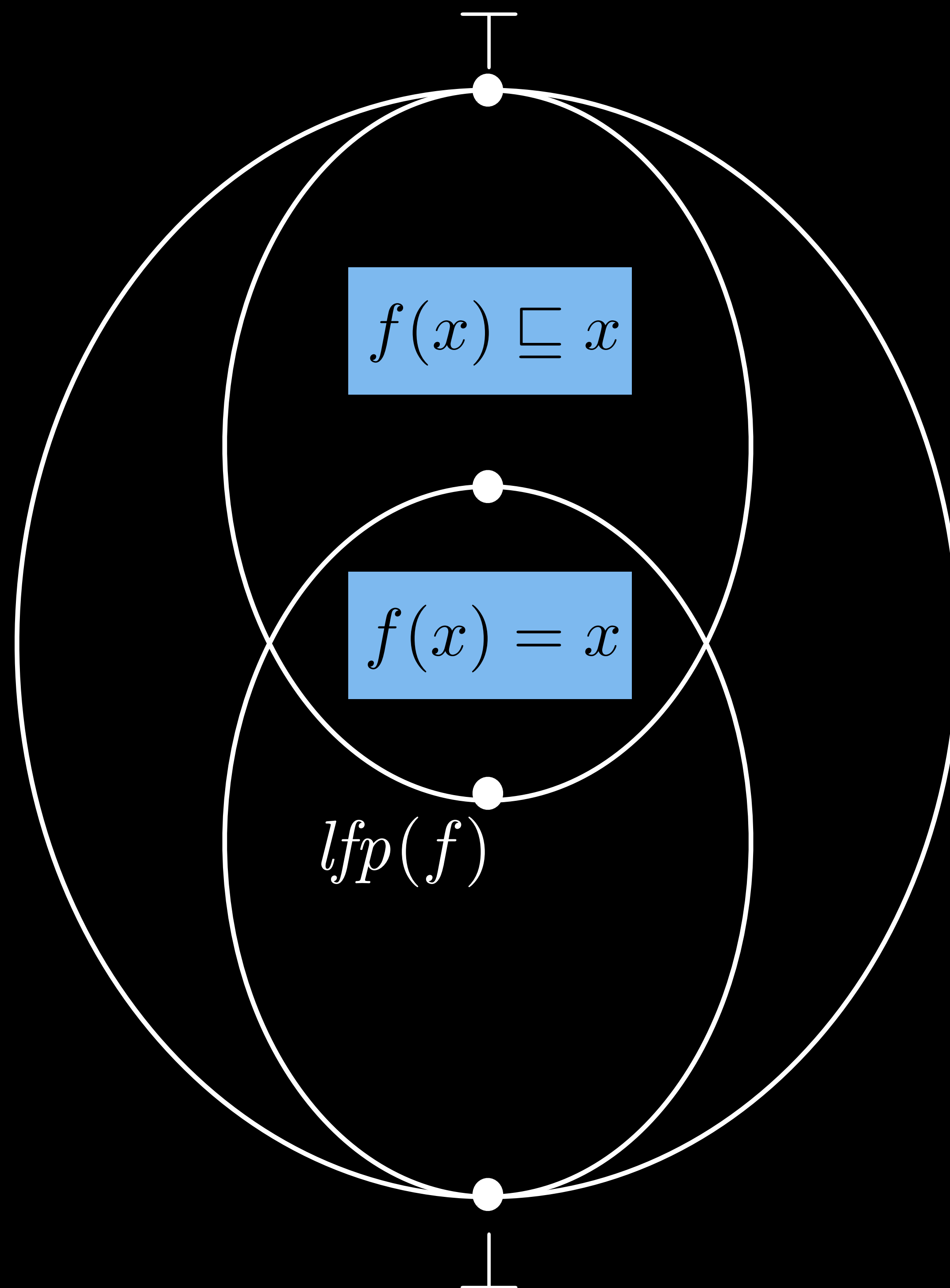
AbLattice.t : $\sqsubseteq^\sharp, \sqcap^\sharp, \sqcup^\sharp, \bot^\sharp$ + widening/narrowing

Each abstract lattice is equipped with a post-fixpoint solver

```
Definition approx_lfp :
    ∀ `{AbLattice.t t}, (t → t) → t := [...]

Lemma approx_lfp_is_postfixpoint :
    ∀ `{AbLattice.t t} (f:t → t),
         f (approx_lfp f) ⊑♯ (approx_lfp f).
```

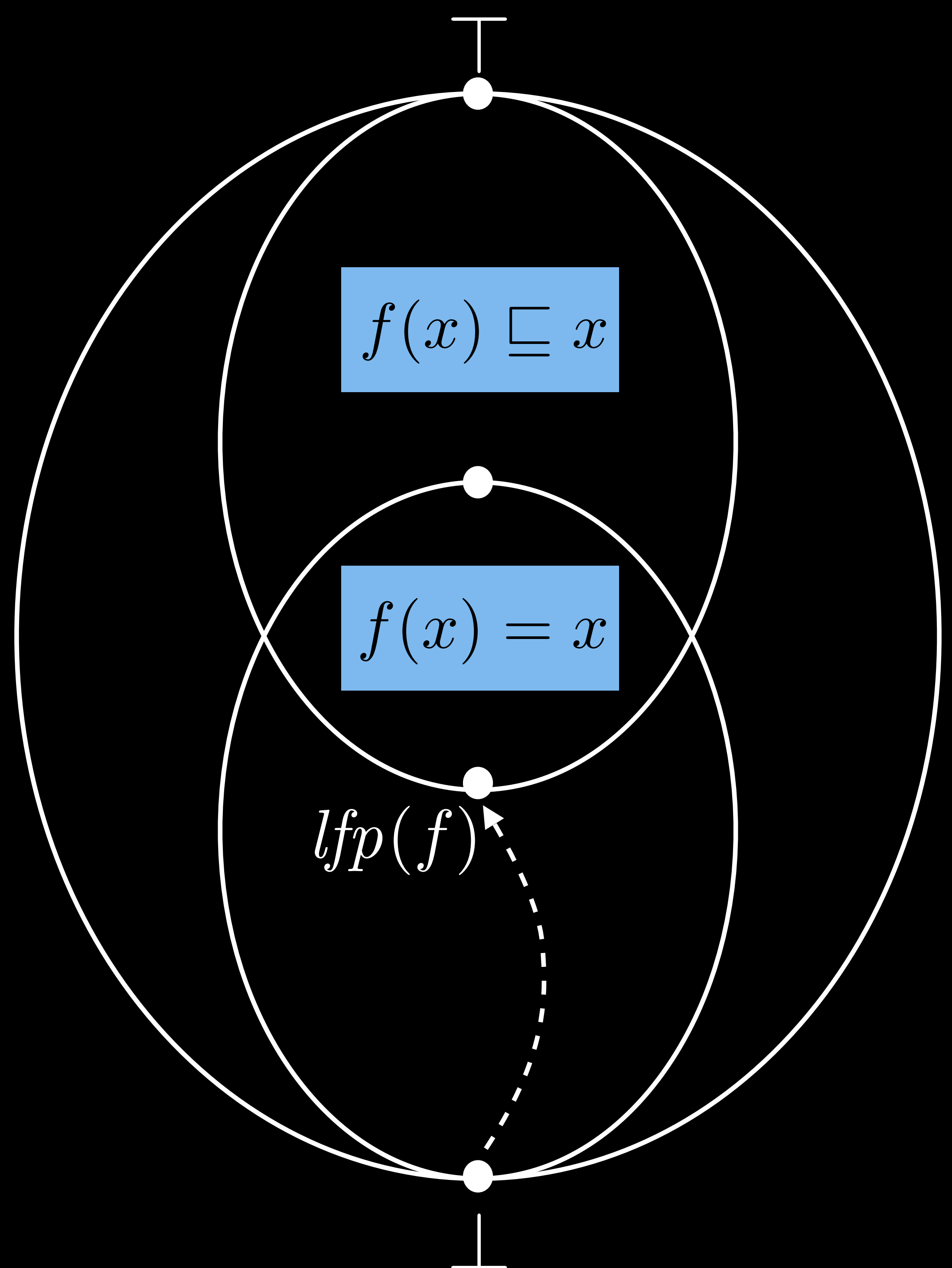
Fixpoint approximation with widening/narrowing



Fixpoint approximation with widening/narrowing

Standard Kleene fixed-point theorem

- too slow for big lattices (or just infinite)



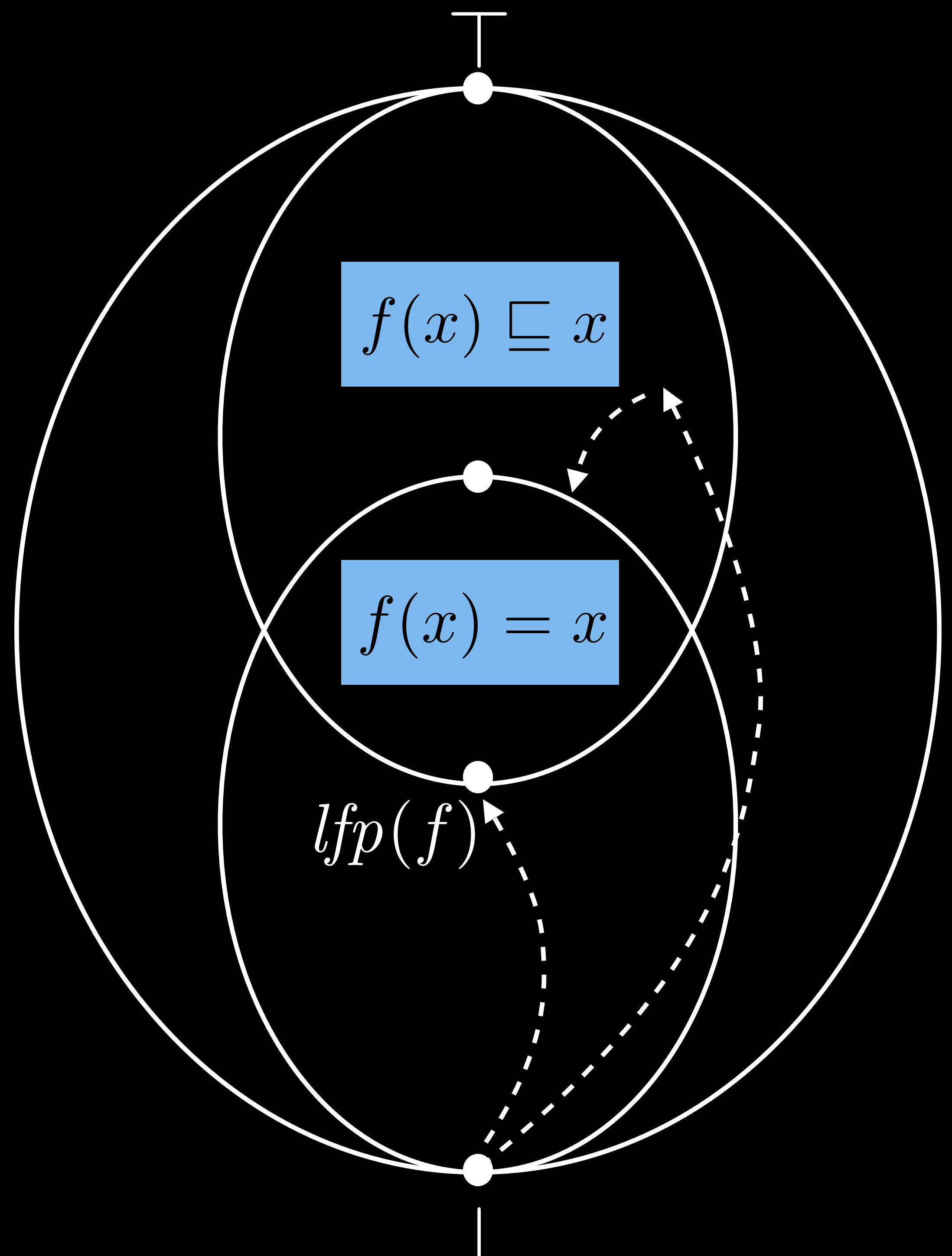
Fixpoint approximation with widening/narrowing

Standard Kleene fixed-point theorem

- too slow for big lattices (or just infinite)

Fixpoint approximation by widening/ narrowing

- over-approximates the lfp.
- requires different termination proofs than ascending chain condition
- on fixpoint equations, iteration order matters a lot !



Abstract Lattice

A library[1] is provided to build complex lattice objects with various functors for products, sums, lists and arrays.

Examples:

```
Instance ProdLattice
  t1 t2 {L1:AbLattice.t t1} {L2:AbLattice.t t2}:
  AbLattice.t (t1*t2) := [...]


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

---

[1] Adapted from our previous work: *Building certified static analysers by modular construction of well-founded lattices*. FICS'08

Abstract Lattice

A library[1] is provided to build complex lattice objects with various functors for products, sums, lists and arrays.

Examples:

```
Instance ProdLattice
  t1 t2 {L1:AbLattice.t t1} {L2:AbLattice.t t2}:
  AbLattice.t (t1*t2) := [...]
```

Contains a difficult termination proof !

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

---

[1] Adapted from our previous work: *Building certified static analysers by modular construction of well-founded lattices*. FICS'08

Abstract Lattice

A library[1] is provided to build complex lattice objects with various functors for products, sums, lists and arrays.

Examples:

```
Instance ProdLattice
  t1 t2 {L1:AbLattice.t t1} {L2:AbLattice.t t2}:
  AbLattice.t (t1*t2) := [...]
```

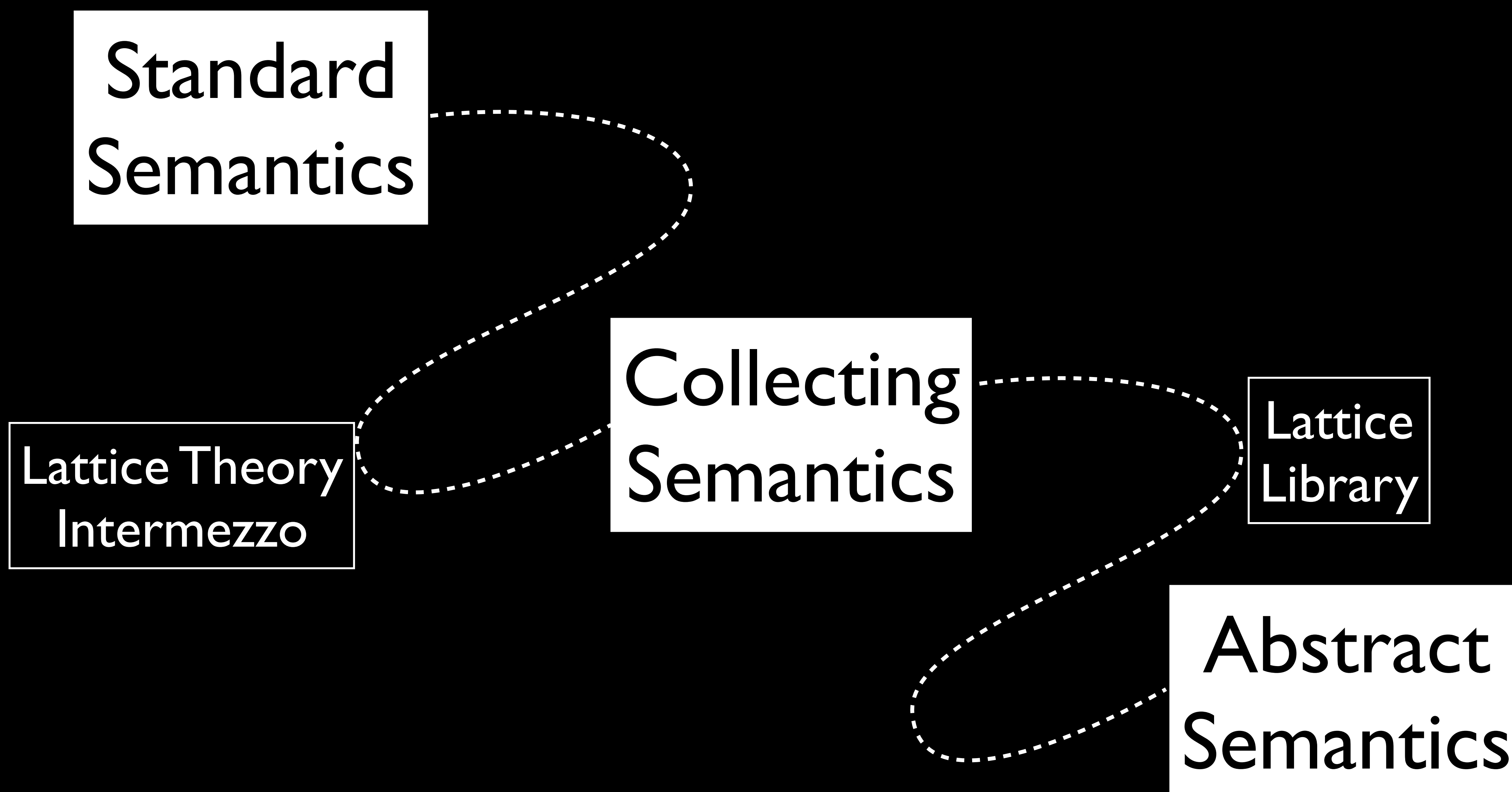
Contains a difficult termination proof !

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

Functional maps

[1] Adapted from our previous work: *Building certified static analysers by modular construction of well-founded lattices*. FICS'08

Roadmap


Standard Semantics
Lattice Theory Intermezzo
Collecting Semantics
Lattice Library
Abstract Semantics

Abstract Algebra

The analyzer is parameterized wrt. to an environment abstraction.

The development provides several non-relational instantiations.

Abstract Algebra

The analyzer is parameterized wrt. to an environment abstraction.

The development provides several non-relational instantiations.

```
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]
```

interval

Abstract Algebra

The analyzer is parameterized wrt. to an environment abstraction.

The development provides several non-relational instantiations.

```
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]
```

interval

```
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
```

sign

Abstract Algebra

The analyzer is parameterized wrt. to an environment abstraction.

The development provides several non-relational instantiations.

```
i = 0; k = 0;
```
$k \in [0, 10] \quad i \in [0, 10]$
```
while k < 10 {
```
$k \in [0, 9] \quad i \in [0, 10]$
```
  i = 0;
```
$k \in [0, 9] \quad i \in [0, 10]$
```
  while i < 9 {
```
$k \in [0, 9] \quad i \in [0, 8]$
```
    i = i + 2
  };
```
$k \in [0, 9] \quad i \in [9, 10]$
```
  k = k + 1
}
```
$k \in [10, 10] \quad i \in [0, 10]$

interval

```
i = 0; k = 0;
```
$k \geq 0 \quad i \geq 0$
```
while k < 10 {
```
$k \geq 0 \quad i \geq 0$
```
  i = 0;
```
$k \geq 0 \quad i \geq 0$
```
  while i < 9 {
```
$k \geq 0 \quad i \geq 0$
```
    i = i + 2
  };
```
$k \geq 0 \quad i > 0$
```
  k = k + 1
}
```
$k > 0 \quad i \geq 0$

sign

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

congruence

Abstract Semantics

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

Fixpoint AbSem (i:instr) (l:pp) : t → array t :=
match i with
| Assign p x e =>
  fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assign Env x e]♯
| While p t i => fun Env =>
  let I := approx_lfp
            (fun X => Env ⊔♯
                      (get (AbSem i p (Ab.assert t X)) p)) in
  (AbSem i p (Ab.assert t I))
    +[p ↦ I]♯+[l ↦ Ab.assert (Not t) I]♯
| [...]
end.
```

Abstract Semantics

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

Fixpoint AbSem (i:instr) (l [...] t :=
match i with
| Assign p x e =>
  fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assign Env x e]♯
| While p t i => fun Env =>
  let I := approx_lfp
             (fun X => Env ⊔♯
                        (get (AbSem i p (Ab.assert t X)) p)) in
  (AbSem i p (Ab.assert t I))
    +[p ↦ I]♯+[l ↦ Ab.assert (Not t) I]♯
| [...]
end.
```

Abstract counterpart of
concrete operations

Abstract Semantics

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

Fixpoint AbSem (i:instr) (l                          :=
match i with
| Assign p x e =>
  fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assign Env x e]♯
| While p t i => fun Env =>
  let I := approx_lfp
              (fun X => Env ⊔♯
                          (get (AbSem i p (Ab.assert t X)) p)) in
   (AbSem i p (Ab.assert t I))
    +[p ↦ I]♯+[l ↦ Ab.assert (Not t) I]♯
| [...]
end.
```

Abstract counterpart of concrete operations

Fixpoint approximation instead of least fixpoint computation

Connecting Concrete and Abstract Semantics

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

Connecting Concrete and Abstract Semantics

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

Soundness proof between abstract and collecting semantics

Type Classes to the rescue

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

Need 4 minutes
after

Type Classes to the rescue

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

canonical order on pp → $\mathcal{P}$(env)

Need 4 minutes after

Type Classes to the rescue

concretization on pp $\to \mathcal{P}$(env)

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

canonical order on pp $\to \mathcal{P}$(env)

Need 4 minutes after

Type Classes to the rescue

concretization on `pp` $\to \mathcal{P}$`(env)`

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

concretization on $\mathcal{P}$`(env)`

canonical order on `pp` $\to \mathcal{P}$`(env)`

Need 4 minutes after

Type Classes to the rescue

concretization on $\texttt{pp} \rightarrow \mathcal{P}(\texttt{env})$

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

concretization on $\mathcal{P}(\texttt{env})$

canonical order on $\texttt{pp} \rightarrow \mathcal{P}(\texttt{env})$

Without Type Classes

Need 4 minutes after

```
Theorem AbSem_correct : ∀ i l_end Env,
  (PointwisePoset (PowerSetPoset env)).(Poset.o
  (Collect prog i l_end (AbEnv.(AbEnv.gamma) Env))
  (FuncLattice.Gamma AbEnv.(AbEnv.gamma) (AbSem i l_end Env)).
```

Type Classes to the rescue

concretization on **pp** $\rightarrow \mathcal{P}$**(env)**

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

concretization on $\mathcal{P}$**(env)**

canonical order on **pp** $\rightarrow \mathcal{P}$**(env)**

Without Type Classes

Need 4 minutes after

canonical order on **pp** $\rightarrow \mathcal{P}$**(env)**

```
Theorem AbSem_correct : ∀ i l_end Env,
  (PointwisePoset (PowerSetPoset env)).(Poset.o
  (Collect prog i l_end (AbEnv.(AbEnv.gamma) Env))
  (FuncLattice.Gamma AbEnv.(AbEnv.gamma) (AbSem i l_end Env)).
```

Type Classes to the rescue

concretization on pp $\rightarrow \mathcal{P}$(env)

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

concretization on $\mathcal{P}$(env)

canonical order on pp $\rightarrow \mathcal{P}$(env)

Without Type Classes

Need 4 minutes after

canonical order on pp $\rightarrow \mathcal{P}$(env)

```
Theorem AbSem_correct : ∀ i l_end Env,
  (PointwisePoset (PowerSetPoset env)).(Poset.o
  (Collect prog i l_end (AbEnv.(AbEnv.gamma) Env))
  (FuncLattice.Gamma AbEnv.(AbEnv.gamma) (AbSem i l_end Env)).
```

concretization on pp $\rightarrow \mathcal{P}$(env)

Type Classes to the rescue

concretization on pp $\rightarrow \mathcal{P}$(env)

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

concretization on $\mathcal{P}$(env)

canonical order on pp $\rightarrow \mathcal{P}$(env)

Without Type Classes

Need 4 minutes after

canonical order on pp $\rightarrow \mathcal{P}$(env)

```
Theorem AbSem_correct : ∀ i l_end Env,
  (PointwisePoset (PowerSetPoset env)).(Poset.o
  (Collect prog i l_end (AbEnv.(AbEnv.gamma) Env))
  (FuncLattice.Gamma AbEnv.(AbEnv.gamma) (AbSem i l_end Env)).
```

concretization on pp $\rightarrow \mathcal{P}$(env)

concretization on $\mathcal{P}$(env)

Connecting Concrete and Abstract Semantics

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

Connecting Concrete and Abstract Semantics

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

The proof is easy because the two semantics are very similar

Abstract Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp): monotone (𝒫(env)) (pp → 𝒫(env)) :=
  match i with
    | Assign p x e =>
      Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
    | While p t i =>
      Mono (fun Env =>
              let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
              (Collect i p (assert t I)) +[p ↦ I] +[l ↦ assert (Not t) I]) _
    [...]
  end.
```

The proof is easy because the two semantics are very similar

```
Fixpoint AbSem (i:instr) (l:pp) : t → array t :=
  match i with
  | Assign p x e =>
    fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assign Env x e]♯
  | While p t i => fun Env =>
    let I := approx_lfp
                (fun X => Env ⊔♯ (get (AbSem i p (Ab.assert t X)) p)) in
      (AbSem i p (Ab.assert t I)) +[p ↦ I]♯+[l ↦ Ab.assert (Not t) I]♯
  [...]
  end.
```

Abstract Semantics

```
Program Fixpoint Collect (i:stmt) (l:pp): monotone (𝒫(env)) (pp → 𝒫(env)) :=
  match i with
    | Assign p x e =>
      Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
    | While p t i =>
      Mono (fun Env =>
              let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
              (Collect i p (assert t I)) +[p ↦ I] +[l ↦ assert (Not t) I]) _
    [...]
  end.
```

First Coq instance of the slogan

My abstract interpreter is correct by construction

```
Fixpoint AbSem (i:instr) (l:pp) : t → array t :=
  match i with
  | Assign p x e =>
    fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assign Env x e]♯
  | While p t i => fun Env =>
    let I := approx_lfp
               (fun X => Env ⊔♯ (get (AbSem i p (Ab.assert t X)) p)) in
      (AbSem i p (Ab.assert t I)) +[p ↦ I]♯+[l ↦ Ab.assert (Not t) I]♯
  [...]
  end.
```

Final Theorem

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

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

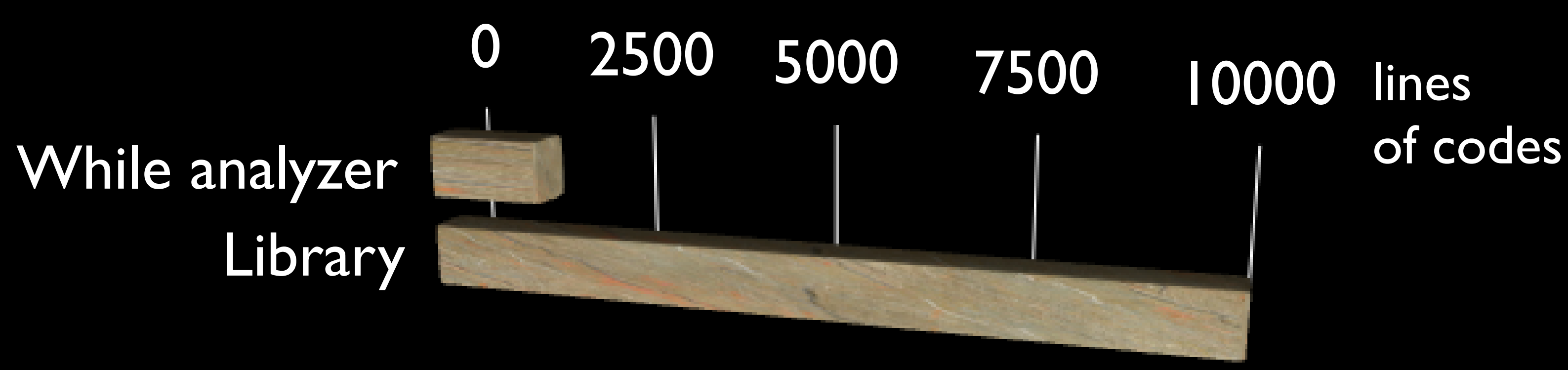
The function `analyse` can be extracted to real OCaml code

You can type-check, extract and run the analyser yourself !
http://www.irisa.fr/celtique/pichardie/ext/itp10/

Conclusions

The first mechanized proof of an abstract interpreter based on a collecting semantics

- requires lattice theory components
- provides a reusable library

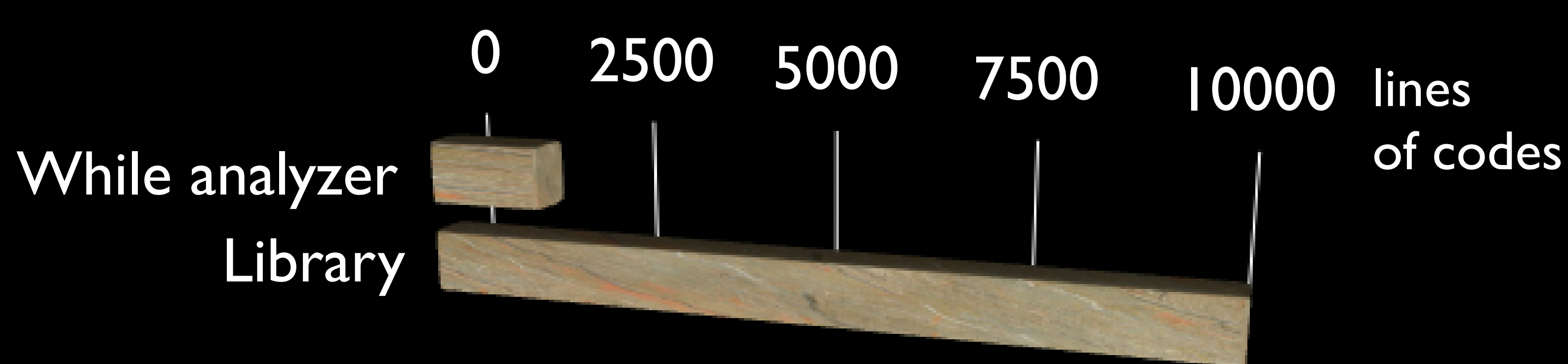


- the proof is more methodic and elegant than previous attempts

Conclusions

The first mechanized proof of an abstract interpreter based on a collecting semantics

- requires lattice theory components
- provides a reusable library



- the proof is more methodic and elegant than previous attempts

Well, of course this is matter of taste...

Perspectives

A first (small) step towards a certified Astrée-*like* analyser

- Ongoing project: scaling such an analyser to a C language
  - on top of the Compcert semantics
  - for a restricted C (no recursion, restricted use of pointers)

Abstraction Interpretation methodology

- would be nice to use more deeply the Galois connexion framework
- we prove soundness and termination: what about precision ?

Thanks !