Approximation Fixpoint Theory in Coq
with an Application to Logic Programming

Bart Bogaerts[1][0000−0003−3460−4251] [⋆] and Luís Cruz-Filipe[2][0000−0002−7866−7484]

[1] Vrije Universiteit Brussel (VUB), Dept. Computer Science
Pleinlaan 2, 1050 Brussels, Belgium
bart.bogaerts@vub.be

[2] University of Southern Denmark, Dept. Mathematics and Computer Science
Campusvej 55, 5230 Odense, Denmark
lcfilipe@gmail.com

Abstract. Approximation Fixpoint Theory (AFT) is an abstract framework based on lattice theory that unifies semantics of different non-monotonic logic. AFT has revealed itself to be applicable in a variety of new domains within knowledge representation. In this work, we present a formalisation of the key constructions and results of AFT in the Coq theorem prover, together with a case study illustrating its application to propositional logic programming.

1 Introduction

Approximation Fixpoint Theory (AFT) is an abstract lattice-theoretic framework originally designed to unify semantics of non-monotonic logics [12]. Its first applications were on unifying all major semantics of logic programming [34], autoepistemic logic (AEL) [27], and default logic (DL) [29], thereby resolving a long-standing issue about the relationship between AEL and DL [13,14,22]. AFT builds on Tarski's fixpoint theory of monotone operators on a complete lattice [32], but crucially moves from the original lattice $\langle L, \leq\rangle$ to the bilattice[3] $\langle L^2, \leq, \leq_p\rangle$, where $\leq$ is just the pointwise extension of the order on L and $\leq_p$ is the precision order defined by $(x, y) \leq_p (u, v)$ if $x \leq u$ and $v \leq y$. Intuitively, a pair $(x, y) \in L^2$ approximates elements in the interval $[x, y] = \{z \in L \mid x \leq z \leq y\}$.

AFT generalizes Tarski's study of monotone operators to non-monotone operators using the notion of *approximator*: a monotone operator $A\colon L^2 \to L^2$ *approximates* a (possibly non-monotone) $O : L \to L$ if $A(x, x) = (O(x), O(x))$ for all $x \in L$. This simple notion allows us to define a variety of different types of fixpoints of interest; in particular:

– A *partial supported fixpoint* of A is a fixpoint of A.

[⋆] This work was partially supported by Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G0B2221N).

[3] While a bilattice is usually defined as an arbitrary set with two compatible orders, AFT is only concerned with bilattices that are in fact *square* lattices; i.e., the underlying set is of the form L^2.

– The *Kripke-Kleene fixpoint* of A is the least fixpoint of A, denoted $\mathrm{lfp}(A)$.
– A *partial stable fixpoint* of A is a pair (x, y) such that $x = \mathrm{lfp}(A(\cdot, y)_1)$ and $y = \mathrm{lfp}(A(x, \cdot)_2)$, where $A(\cdot, y)_1$ denotes the function $L \to L\colon z \mapsto A(z, y)_1$, and analogously for $A(x, \cdot)_2$.
– The *well-founded fixpoint* of A is the least precise partial stable fixpoint of A.

When applying this to logic programming, Denecker and his coauthors [12] found that Fitting's four-valued immediate consequence operator Ψ_P [16] is an approximator of Van Emden & Kowalski's [34] immediate consequence operator T_P and that all major semantics of logic programming correspond to the fixpoints defined above. The same kind of situation is observed in other fields, such as AEL and DL. Crucially, all that is required to apply AFT to a formalism and obtain several semantics is to define an appropriate approximating operator $L^2 \to L^2$ on this bilattice; often there is an obvious choice for such an approximating operator. Once this is done, the algebraic theory of AFT directly defines different types of fixpoints that correspond to different types of semantics of the application domain. This immediately give insight into how this domain is positioned with respect to other fields (semantically), but also internally within the domain shines light on how different semantics are related.

The last decade has added new application domains to AFT, such as abstract argumentation [31], extensions of logic programming [1,11,24,28], extensions of autoepistemic logic [36], active integrity constraints [5], and constraint languages for the semantic web [8]. The original theory of AFT has also been extended significantly with new types of fixpoints [9,10], and results on *stratification*, [6,37], *predicate introduction* [38], *strong equivalence* [33], and non-deterministic operators [20]. All of these results were developed in the highly general setting of lattice theory, making them directly applicable to all application domains, and such ensuring that researchers do not "reinvent the wheel".

Given the success and wide range of applicability of AFT, it sounded natural to formalise this theory using a suitable theorem prover. We chose Coq [3] for this purpose, as this is the theorem prover we are most familiar with. This option poses some challenges: presentations of AFT routinely use classical reasoning principles and most proofs are by transfinite induction. Our option was to develop constructive alternatives to standard proofs and explicitly include some hypotheses that are classically trivially true where needed. We believe this effort to be meaningful, since an important motivations for studying fixpoints in computer science is their computability.

A natural question that arises is whether there are constructive models of our axioms, and this motivated including some examples in the formalisation. We also point out that most practical applications of AFT (in particular, our larger example described in Section 4) use the subset lattice of a finite base set with decidable equality, where most of the working assumptions hold.

2 Ordinals and lattices

AFT relies heavily on definitions of ordinals and lattices. We formalised these as the first step in our development, choosing to use only results from Coq's standard library on lists and natural numbers. The primary reason for this choice was that we did not initially find any existing formalisations that were easy to reuse for our purposes. It was only later in the process that we discovered other libraries close to our development – see Section 5 for a discussion.

Ordinals. We define a type `Ordinal` of (unbounded) ordinals as a record type containing a support `Type`, an equivalence relation `eq` (defined equality), a distinguished element `zero`, a successor function `succ` and a strict total order `lt` that is well-founded and compatible with `eq`.[4]

```
Record Ordinal : Type := { T :> Type;
                           eq : T → T → Prop;
                           zero : T;
                           succ : T → T;
                           lt : relation T;
                           ...  }
```

We require `succ x` to be the least element strictly greater than `x`, and that we can decide for each element whether it is of the form `succ x` for some `x`, or not; the elements for which this does not hold are called limits. This case analysis, together with well-foundedness of `lt`, allows us to do proofs about ordinals by transfinite induction, as long as the property being proved is stable under equality (this is due to working with a defined equality).

```
Lemma transfinite_induction (P:O → Prop) :
  (∀ x y, eq x y → P x → P y) →
  (∀ x, P x → P (succ x)) →
  (∀ x, limit x → (∀ y, lt y x → P y) → P x)
  → ∀ x, P x.
```

Intuitively, the elements of `O:Ordinal` are the ordinals smaller than `O`. As a sanity check, we construct some typical examples, including the first infinite ordinal ω (with support `nat`) and the type of all polynomials in ω (with support `list nat`). We do not deal with arithmetic on ordinals, since this is immaterial for our development.

Lattices. (Complete) lattices are similarly defined as a record type consisting of a carrier type `C` with a defined equality (an equivalence relation compatible with the order), a partial order, and an operator `lub` computing least upper bounds.

[4] Throughout this presentation we write ... for additional arguments that are left out for conciseness; we leave some arguments implicit when they can be inferred from the context by a human (even if not by Coq); we omit universally quantified variables at the top of lemmas; and we ignore namespace clashes that force us to include module names in the Coq source.

Requiring least upper bounds to be computable restricts the kind of lattices that we can define; still we show that we capture e.g. powerset lattices (which appear in all applications of AFT so far).

```
Record Lattice : Type := { C :> Type;
                           eq : relation C;
                           leq : relation C;
                           lub : (C → Prop) → C;
                           ...  }
```

Bottom and joins are defined using `lub`, while the strict order `less` is defined as `fun x y ⇒ leq x y ∧ ~eq x y`. We prove the usual properties of all these functions, including uniqueness of `lub` modulo `eq`. Reversing the order in a lattice yields a dual lattice, and we use a reflection technique to port results about `leq` and `lub` to `geq` and `glb`.

A standard example of a complete lattice, ubiquous in AFT, is the powerset lattice. We formalise this construction as an operator `PowerSet: Type → Lattice`, such that `PowerSet T` has carrier `T → Prop`.

Given a lattice $\langle L, \leq \rangle$, the corresponding precision lattice is the lattice $\langle L^2, \leq_p \rangle$ where $(x, y) \leq_p (x', y')$ iff $x \leq x'$ and $y' \leq y$. We formalise this construction as an operator `BiLattice: Lattice → Lattice`.

```
Definition L2 : Type := L*L.
Definition L2eq := fun x y ⇒ eq (fst x) (fst y) ∧ eq (snd x) (snd y).
Definition leqp := fun x y ⇒ leq (fst x) (fst y) ∧ leq (snd y) (snd x).
Definition L2lub := fun (S:L2 → Prop) ⇒ (lub (fun x ⇒ ∃ y, eq x (fst y) ∧ S y),
                                         glb (fun x ⇒ ∃ y, eq x (snd y) ∧ S y)).
Definition BiLattice := Build_Lattice L2 L2eq leqp L2lub ... .
```

We now prove our first set of results about fixpoints, starting with the Knaster–Tarski theorem. Given a monotonic function `f` on a lattice `L`, we define `lfp f` as the `glb` of all its prefixpoints, and show that this is the smallest fixpoint of `f`.

```
Definition prefp : (f:L → L) → L → Prop := fun x ⇒ leq (f x) x.
Definition lfp (f:L → L) := glb (prefp f).

Lemma fp_lfp : monotonic f → fixpoint f (lfp f).
Lemma lfp_lfp : fixpoint f x → leq (lfp f) x.
```

Next, we define `chain f` as a predicate over `L` that holds for those elements of `L` that are reachable from `bot L` by repeated application of `f` and `lub`s. We prove that, if `f` is monotonic, `lub (chain f)` also satisfies `chain`, and that it coincides with `lfp f`. All these proofs mostly follow the standard pen-and-paper argumentation.

```
Inductive chain (f:L → L) : L → Prop :=
| base x : eq x (bot L) → chain f x
| succ x y : chain y → eq x (f y) → chain f x
| lim x (S:L → Prop) : (∀ y, S y → chain f y) → eq x (lub S) → chain f x.

Lemma lub_chain_lfp : monotonic f → eq (lfp f) (lub (chain f)).
```

Finally, given a function `f` that respects equality, we define iteration of a function `f` an ordinal number of times using well-founded recursion. This definition is not very informative, and we show that the resulting definition coincides with what is expected, and only generates elements in `chain f`.

```
Definition iterate_fun (f: L → L) {O:Ordinal} :=
  fun (x:O) (F:(∀ (y:O), lt y x → C L)) ⇒
    match (succ_or_limit x) with
    | inleft (existT _ y H) ⇒ f (F y (succ_lt' _ _ _ H))
    | _ ⇒ (lub (fun z ⇒ ∃ y (H:lt y x), eq z (F y H)))
    end.

Definition iterate {O:Ordinal} (t:T O) :=
  Fix (lt_wf O) _ iterate_fun t.

Lemma iterate_succ : eq (iterate f (succ x)) (f (iterate f x)).
Lemma iterate_limit : limit x →
  eq (iterate f x) (lub (fun z ⇒ ∃ y, lt y x ∧ eq z (iterate f y))).
Lemma iterate_zero : eq (iterate f zero) (bot L).

Lemma iterate_chain : chain f (iterate f t).
```

O-inductions. In this section, we fix a lattice `L`, an ordinal `o` and an operator `O:L → L`. We write `eqL` for the equality on `L` and `eqO` for the equality on `o`.

AFT provides an alternative characterisation of lfps using what are called *O*-inductions, which relax the definition of chain by allowing the sequence to grow "slower". In the application of AFT to logic programming this boils down to not necessarily applying all applicable rules at every step.

```
Definition O_refinement (x y:L) := leq x y ∧ leq y (join x (O x)).
Definition O_induction (i:o → L) := (∀ y, O_refinement (i y) (i (succ y)))
  ∧ (∀ y, limit y → eqL (i y) (lub (fun l ⇒ ∃ z, lt z y ∧ eqL l (i z))))
  ∧ ∀ x y, eqO x y → eqL (i x) (i y).
```

An `O_induction` that cannot be refined further is called `terminal`. We formalise this notion by explicitly identifying an ordinal that returns its last element. If `O` is monotonic, then all `O_induction`s converge to its least fixpoint.

```
Definition terminal (i:o → L) (o':o) := ∀ y, O_refinement (i o') y → eqL y (i o').
Lemma terminal_O_induction_limit : terminal i o' → eqL (i o') (lfp O).
```

One of the classic results in AFT states that every monotonic operator has an `O_induction` that is `terminal`. The proof uses the fact that any chain can be injected in some "large enough" ordinal.

```
Definition large_enough := ∀ f, increasing f →
  ∃ (i:O → L), O_induction f i ∧ ∀ y, chain f y → ∃ o, eqL y (i o).

Lemma large_enough_terminal : large_enough → ∀ f, monotonic f →
  ∃ i o, O_induction f i ∧ terminal f i o.
```

We also prove a weaker version of this result, where the size of the ordinal is allowed to depend on `f`.

```
Lemma large_enough_terminal' : monotonic f → O_induction f O i →
  (∀ y, chain f y → ∃ o, eqL y (i o)) → ∃ o, terminal L f O i o.
```

We discuss the existence of such ordinals separately. In particular, we prove that the hypotheses of the last lemma hold for any monotonic function on any lattice assuming (i) that the chain is a total order and (ii) a restricted form of Markov's principle.

```
Hypothesis Hf : ∀ x y:L,  chain f x → chain f y → leq x y ∨ ~leq x y.

Hypothesis forall_exists: ∀ (S:L → Prop),
  (~∀ x, S x → leq x y) → ∃ x, S x ∧ less y x.

Lemma chain_has_large_enough : monotonic f →
  ∃ O i,  O_induction f O i ∧ (∀ y, chain f y → ∃ o, eqL y (i o)).
```

The construction of the ordinal is a bit involved, and due to space restrictions we omit it here.

3 Approximators and fixpoints

The bulk of AFT is built on the notion of approximator of an operator O: a (precision-)monotonic function on the billatice that coincides with O on exact values. Throughout this section we assume the lattice `L` to be fixed, as well as `O:L → L` and `A:BiLattice L → BiLattice L`.

```
Definition approximator O A := monotonic A ∧ ∀ x, eqL (A (x,x)) (O x,O x).
```

Two types of fixpoints used in AFT are defined directly from the approximator: supported fixpoints are simply fixpoints of A; the Kripke-Kleene fixpoint is the least fixpoint of A.

```
Definition supported_fp (x:BiLattice L) := fixpoint A x.
Definition Kripke_Kleene_fp := lfp A.
```

Stable and well-founded fixpoints are defined using the stable revision operator `stable_revision`, which maps (x, y) to $(\mathrm{lfp}\, A(\cdot, y)_1, \mathrm{lfp}\, A(x, \cdot)_2)$. The two components of this operator are defined separately using two operators `partial_A_1` and `partial_A_2`. Reasoning about the monotonicity of these allows us to show that `stable_revision` is monotonic, after which we can define the above-mentioned fixpoints and prove their usual relationships.

```
Definition stable_fp (x:BiLattice L) := fixpoint (stable_revision A) x.
Definition wf_fp := lfp (stable_revision A).

Lemma stable_fp_fp_A : monotonic A → ∀ x, stable_fp A x → supported_fp A x.
Lemma wf_fp_stable : monotonic A → stable_fp A (wf_fp A).
Lemma wf_fp_fp : monotonic A → supported_fp A (wf_fp A).
Lemma Kripke_Kleene_wf_fp : monotonic A → leq (Kripke_Kleene_fp A) (wf_fp A).
```

An element (x,y) of `BiLattice L` is consistent (`L_consistent`) if `leq x y`, i.e., if the set of elements `z` it approximates is non-empty, and an operator is consistent (`A_consistent`) if it maps consistent elements to consistent elements. In particular, all approximators are `A_consistent`. If `A_consistent A`, then all values in an `O_induction` over `A` are `L_consistent`; furthermore, if `A` is symmetric (meaning $A(x,y)_1 = A(y,x)_2$ for all x and y), then the stable revision operator is consistent as well: `A_consistent (stable_revision A)`.

To prove additional results on preservation of consistency we needed to assume the existence of a large enough ordinal.

```
Hypothesis Ho : large_enough o (BiLattice L).
Lemma consistent_lfp : monotonic A → A_consistent A → L_consistent (lfp A).
Lemma consistent_Kripke_Kleene_fp : L_consistent (Kripke_Kleene_fp A).
Lemma wf_consistent : A_symmetric A → L_consistent (wf_fp A).
```

These lemmas also follow from the fact that every element in `chain A` is `L_consistent`; however, we were not able to prove this result without assuming existence of a large-enough ordinal, either.

Lastly, we show that well-founded fixpoints can be computed via *well-founded inductions* (`wf_induction`), first defined by Denecker and Vennekens [15]. These are transfinite sequences over `BiLattice L` that generalise O-inductions: a refinement either updates (x, y) by following the approximator (to something at most as precise as $A(x, y)$), or it decreases the second component, intuitively by removing elements that could only ever be derived by means of ungrounded (self-supporting) reasoning.

```
Inductive A_refinement (x y : BiLattice L) : Prop :=
| A_application : leq x y → leq y (join x (A x)) → A_refinement x y
| A_unfounded : eqL (fst x) (fst y) → leq (snd y) (snd x) →
                    leq (snd (A y)) (snd y) → A_refinement x y.
```

The definition of `wf_induction` is similar to the definition of `O_induction`, using `A_refinement` instead of `O_refinement`. We can similarly define a notion of `wf_terminal`, and use it to relate each `wf_induction` with fixpoints of `A`.

```
Lemma wf_induction_fp : wf_terminal i o' → fixpoint A (i o').
Lemma wf_terminal_stable_fp : wf_terminal i o' → stable_fp A (i o').
Lemma wf_terminal_wf_fp : wf_terminal i o' → eqL (i o') (wf_fp A).
```

The last two results are proved in two steps. First, we use transfinite induction to show that any `wf_induction` always stays below those fixpoints. Secondly, we define an auxiliary notion

```
Definition prudent a := ∀ x, leq (fst (A (x,snd a))) x → leq (fst a) x.
```

Intuitively, this tells us that `fst a` is derived for a "good" reason. Indeed, `a` is *not* `prudent` in case there is some `x` smaller than `fst a` that is a prefixpoint of $A(\cdot, a_2)$, meaning that the only way to derive `a` would be starting from something larger than `x` in the first place: `fst a` could not be derived from the ground up.

We then show that all elements of a `wf_induction` are `prudent`, which follows from the definition of `prudent`.

This concludes the first part of the formalisation. After fine-tuning the definitions of ordinals and lattices and finishing the respective formalisations, the development of AFT described in this section was then relatively straightforward. The main challenges were (a) identifying the results that depend on the possibility of building a large-enough ordinal and (b) developing direct arguments for several results where the classical proof is by contradiction. The latter turned out sometimes to be a tricky exercise, but always possible.

4 An example: propositional logic programming

We now formalise a complete example, to show that our theory is applicable. We focus on propositional logic programming with negation, and show that the standard, classical, semantics correspond to fixpoints in AFT.

Throughout this section we assume a fixed non-empty set `symb:Set` of propositional symbols with decidable equality. The type of `literal`s is defined as an inductive type with two constructors `pos,neg:symb → literal`, and `rule` as a `list literal` (the body of the rule) paired with a single `symb` (the head of the rule). We add the standard notation `h :- b` for `(b,h): rule`. Finally, a `program` is a list of `rule`s. We define predicates `pos_L`, `pos_R` and `pos_P` for positive literals, rules and programs – those where all `literal`s are built using the constructor `pos`. We also define the list `symbs_in_P P` of all symbols in `P:program`; this list is defined straightforwardly by going through the program and appending any symbols found, with no efforts to remove duplicates.

The next step is defining the standard semantics of logic programming. The type `interpretation` is simply defined as `symb → Prop`; this is also the carrier type of the lattice `PowerSet symb`, on which we work later on. Interpretations map literals, rules and programs to propositions in the natural way, and `I: interpretation` is a `model` of `P: program` if `I` maps `P` to a true proposition. We also define `supported_model`: an interpretation `I` where all true propositional symbols are supported by some rule whose body is also true in `I` – the function `intlL` generalizes an interpretation to a list of literals.

```
Definition supported_model (I:interpretation) (P:program) : Prop :=
  model P ∧ ∀ s, I s → ∃ b, In (s :- b) P ∧ int_lL I b.
```

A `least_model` is a model that is smaller (wrt the order in `PowerSet symb`) than all other models, and a `minimal_model` is one that is not strictly larger than any other model.

The semantics of programs with negation is classically defined using the notion of reduct [17]. Computing the reduct of a program requires being able to decide whether an interpretation satisfies the negative literals in the body of a rule; instead, we define what it means for a program `Pr` to be a reduct of another program `P`, and show that there is at most one program with this property (up to reordering and duplication of rules). Furthermore, this program is always positive.

```
Definition reduct_R (r:rule) := (head r :- pos_atoms (body r)).
Definition reduct (I:interpretation) (P Pr:program) : Prop :=
  ∀ r, In r Pr ↔ ∃ r', In r' P ∧ all_negs_true r' I ∧ r = reduct_R r'.

Lemma reduct_unique : reduct I P Pr → reduct I P Pr' → ∀ r, In r Pr → In r Pr'.
Lemma reduct_pos : reduct I P P' → pos_P P'.
```

We can now define stable models, van Emden and Kowalski's immediate consequence operator, and Fitting's consequence operator. The latter works on `BiLattice (PowerSet symb)`, viewing the pair `(I,J)` as the knowledge that everything in `I` holds and nothing outside `J` holds.

```
Definition stable (I:interpretation) (P:program) :=
  ∃ Pr, reduct I P Pr ∧ minimal_model I Pr.

Definition consequence (P:program) : PowerSet symb → PowerSet symb :=
  fun I s ⇒ ∃ r, In r P ∧ s = head r ∧ int_lL I (body r).

Definition comb_int_L (I J:interpretation) (l:literal) :=
  match l with pos s ⇒ I s | neg s ⇒ ~J s end.

Definition comb_int_P (P:program) (I J:interpretation) : interpretation :=
  fun s ⇒ ∃ r, In r P ∧ s = head r ∧ ∀ l, In l (body r) → comb_int_L I J l.

Definition Fitting_cons (P:program) :
  BiLattice (PowerSet symb) → BiLattice (PowerSet symb) :=
  fun X ⇒ match X with (I1,I2) ⇒ (comb_int_P P I1 I2,comb_int_P P I2 I1) end.
```

From these definitions we can show the classical results that all models of a program are prefixpoints of the associated immediate consequence operator, and all supported models are fixpoints. The converse implications require some classical reasoning.

```
Lemma model_prefp : model I P → prefp (immediate_cons P) I.
Lemma supp_model_fp : supported_model I P → fixpoint (immediate_cons P) I.

Hypothesis I_classical : ∀ I symb, I symb ∨ ~I symb.

Lemma prefp_model : prefp (immediate_cons P) I → model I P.
Lemma fp_supp_model : fixpoint (immediate_cons P) I → supported_model I P.
```

With this assumption we can also show that positive programs have a unique minimal model, which coincides with `lfp (immediate_cons P)`.

```
Lemma pos_P_minimal_lfp :
  pos_P P → minimal_model I P → eqL (lfp (immediate_cons P)) I.
```

We then move to the Fitting consequence operator. We show that it is `monotonic` and that it approximates the immediate consequence operator. Furthermore, stable models of a program are stable fixpoints of the Fitting consequence operator; the converse holds provided that reducts always exist.

```
Lemma stable_model_stable_fp_Fitting :
  stable I P → stable_fp (Fitting_cons P) (I,I).

Hypothesis P_has_reduct : ∀ I, ∃ P', reduct I P P'.

Lemma stable_fp_Fitting_stable_model :
  stable_fp (Fitting_cons P) (I,I) → stable I P.
```

Next, we turn our attention to the well-founded semantics of logic programs. We formalise the construction of well-founded models described in [35], which we now describe.

Well-founded models are built using a notion of partial interpretation – two disjoint[5] lists of propositional symbols, those known to be true, and those known to be false. We define this a record type. Partial interpretations are ordered by knowledge.

```
Record partial_int := { ppos : list symb;
                        pneg  :  list symb;
                        pcons : disjoint ppos pneg }.

Definition p_le (I J: partial_int) : Prop :=
  incl (ppos I)(ppos J) ∧ incl (pneg I) (pneg J).
```

Satisfaction of literals simply reduces to checking whether the underlying propositional symbol is included in the corresponding list. This notion generalises to rules and programs in the natural way. Satisfaction is decidable, since it is based on finite lists over a decidable type. Falsification (stronger than the negation of satisfaction) is defined similarly, but checking whether the symbol underlying a positive (respectively, negative) literal is in the negative (resp. positive) list of symbols in the partial interpretation.

Well-founded models are built as fixpoints of an operator that works distinctly on the positive and negative parts of a partial interpretation. For the positive part, the authors use an operator T that naturally generalises the immediate consequence operator. (We prepend the authors' initials to the name of the Coq counterparts to these operators.)

```
Fixpoint GRS_T (P: program) (I: partial_int) : list symb :=
  match P with
  | nil ⇒ nil
  | (b,h) :: P' ⇒ if satisfy_lL_dec I b then (h :: GRS_T P' I) else (GRS_T P' I)
  end.
```

`GRS_T` simply iterates through a program and collects the heads of the rules whose bodies are satisfied in `I`. We show that it behaves as expected.

```
Lemma GRS_T_char : In s (GRS_T P I) → ∃ b, In (b,s) P ∧ satisfy_lL I b.
Lemma GRS_T_char' : In (b,s) P → satisfy_lL I b → In s (GRS_T P I).
```

For the negative part, the authors use the notion of unfounded set wrt a partial interpretation `I` – a set of atoms that can never be proven by extending

[5] Disjointness of this lists is called *consistency* in the original reference.

I – and define the greatest unfounded set wrt I as the union of all these sets. The latter definition is not directly translatable to Coq; instead, we construct this set directly and prove that it has the desired property.

```
Definition unfounded_R (l: list symb) (I: partial_int) (s: symb) (r: rule) :=
  match r with (b,h) ⇒ s = h → falsify_lL I b ∨ ∃ s', In (pos s') b ∧ In s' l end.

Definition unfounded_P (l: list symb) (I: partial_int) (s: symb) :=
  ∀ r, In r P → unfounded_R l I s r.

Definition unfounded (l: list symb) (I: partial_int) : Prop :=
  ∀ s, In s l → In s (symbs_in_P P) ∧ unfounded_P l I s.
```

To construct the greatest unfounded set wrt I, we first define a function `remove_not_unfounded` that removes all elements from a list `l:list symb` that are unfounded wrt I. We then define another function `remove_n_times` that iterates the previous function, and prove that this reaches a fixpoint when given `length l` as an argument. Finally, `gus I` is defined by computing this fixpoint from the list of symbols in P. We show that this is indeed the greatest unfounded set wrt I according to [35].

```
Lemma gus_unfounded : unfounded (gus I) I.
Lemma gus_greatest: unfounded l I →
  (∀ s, In s l → In s (symbs_in_P P)) → ∀ s, In s l → In s (gus I).
```

In the second lemma, the extra condition is needed to account for the fact that our type `symb` may include symbols not in the Herbrand base of P.

The operator U maps each interpretation to its greatest unfounded set.

```
Definition GRS_U : partial_int → list symb := fun I ⇒ gus I.
```

Combining T and U yields an operator W whose iterations from the empty partial interpretation are always partial interpretations (i.e., consistent – cf. Lemma 3.4 in [35]). In general, though, W does not preserve consistency, and the proof of the lemma cited uses some properties that hold specifically for iterates of W. We call partial interpretations with this properties *rich partial interpretations*.

```
Record rich_pint : Set :=
  { rpint :> partial_int;
    rgen  : ∀ s, In s (ppos rpint) → ∃ b, In (b,s) P ∧ satisfy_lL rpint b;
    runf  : ∀ s, In s (pneg rpint) →
              ∃ l, (∀ s, In s l → In s (symbs_in_P P))
              ∧ In s l ∧ unfounded l rpint;
    rprop : ∀ J l p, rpint ≤ p J → In p (ppos rpint) → unfounded l J → ~In p l}.
```

For `I:rich_pint`, property `rgen` imposes that all elements in `ppos I` are supported by some rule that is true in I; dually, `runf` ensures that all elements in `pneg I` must be in some unfounded set wrt I. The last property states that no element of `ppos I` can ever become unfounded, in the sense that it can not be in any unfounded set wrt a partial interpretation J extending I.

We show that, for `I:rich_pint`, the lists `GRS_T I` and `GRS_U I` are disjoint and satisfy the properties `rgen`, `runf` and `rprop`. This allows us to define `GRS_W_rich` as an operator over `rich_pints`, and iterating it from the empty partial interpretation we obtain `GRS_W_aux : nat → rich_pint` (the partial interpretation W_n in [35]). Finally, we define `GRS_W : nat → partial_int` by forgetting the extra structure in `GRS_W_aux n`.

The next step is showing that `GRS_W` reaches a fixpoint (the well-founded model of the program). We employ a counting trick: we show that the number of decided symbols in `GRS_W n` (i.e. the number of symbols appearing in either `ppos (GRS_W n)` or `pneg (GRS_W n)`) increases unless `GRS_W n` and `GRS_w (S n)` are equal as partial interpretations.

```
Lemma GRS_W_conv_1 : ∀ (I:rich_pint), ~I =p preW I →
  ∃ s, decided s (preW I) ∧ ~decided s I.

Lemma GRS_W_conv_2 : ~GRS_W n =p GRS_W (S n) →
  S n ≤ size_as_set (ppos (GRS_W (S n)) ++ pneg (GRS_W (S n))).

Lemma GRS_W_fp_char :
  GRS_W (length (symbs_in_P P)) =p GRS_W (length (symbs_in_P P) + 1).

Definition wf_model : partial_int := GRS_W (length (symbs_in_P P)).
```

The last step is showing that `wf_model P` and `wf_fp (Fitting_cons P)` coincide. The hard part of the proof is relating the second component of the stable revision operator built from `Fitting_cons P` with the notion of greatest unfounded set.

```
Lemma gus_Fitting : In s (symbs_in_P P) → In s (gus I) ↔
    ~snd (stable_revision (Fitting_cons P) (p_int_to_pair_int I)) s.
```

This lemma relies heavily on the fact that `I` is built from two lists, and therefore we can always decide whether `I s` or `~I s` holds; this implies that the fixpoint constructions inside the computation of `stable_revision` converge in a (computable) number of iterations, and that the result is also decidable in the same sense. From this lemma, we prove the final result in our formalisation. The function `p_int_to_pair_int` is the natural map from `partial_ints` to `BiLattice (PowerSet symb)`.

```
Lemma wf_model_wf_fp :
  eqL (p_int_to_pair_int wf_model) (wf_fp (Fitting_cons P)).
```

5 Related work

To the best of our knowledge, this is the first time that AFT has been formalised using a theorem prover. The most closely related works that we are aware of deal are formalisations of ordinal theory and transfinite induction, or of results in lattice theory.

Several authors have formalised transfinite induction in Coq [2,19,26,30]. Due to the difficulty of developing such a formalisation satisfactorily in a constructive setting, several of them [26,30] are based on classical set theory.

Barras [2] discusses formalising a model of the Calculus of Inductive Constructions in Coq. His work uses an impredicative definition of ordinals as the least collection of transitive sets such that any set of members of the collection also belongs to the collection, and proves the Knaster-Tarski theorem.

Grimm [19] formalises three different types of ordinals in Coq, including arithmetic operations on them and a notion of order. To be comparable, ordinals need to be reduced to a normal form. The formalisation also includes principles for reasoning using transfinite induction, and the author proves a correspondence to van Neumann ordinals.

Due to the authors' different purposes, all these works end up being difficult to use in our setting, which motivated us to include a novel definition of ordinals in our current contribution.

Other formalisations of Tarki's fixed point theorem are included in the work of Grall [18] and in the CoLoR library [4,25]. However, these works do not deal with ordinals, and even though their approach to lattice theory is similar to ours the overlap is very reduced.

Ordinals have also been formalised in Agda using Homotopy Type Theory (HoTT), where extensional equality occurs naturally. In particular, it has been shown [21] that, in HoTT, the constructive definition of ordinals as a hereditarily transitive set coincides with their definition as a type with a transitive, wellfounded and extensional order relation.

One may question whether formalising AFT requires using ordinals and transfinite induction at all, as the latter can often be replaced by well-founded induction [23]. We chose not to investigate this alternative path, as we were trying to follow existing references on AFT as closely as possible. Furthermore, several concepts in AFT are defined directly using ordinals (e.g. well-founded inductions), so bypassing ordinals would require major changes to the theory.

6 Conclusions

We described a formalisation of approximation fixpoint theory in the theorem prover Coq, together with an example of how it can be applied to propositional logic programming. In our work, we tried to work constructively as much as possible, in the sense that we developed alternative, direct, proofs instead of the standard proofs by contradiction or case analysis that are found in the literature. Where such an approach was not possible, we opted for explicitly identifying our classical assumptions (e.g., decidability of equality on some types). We believe there is value in this exercise, as it increases our understanding of how much AFT depends on classical reasoning.

As an example, we formalised the classical theory of propositional logic programming, including the immediate consequence operator, Fitting's consequence operator and the construction of well-founded models as fixpoints of an operator based on unfounded sets, and showed that these can be characterised through corresponding concepts in AFT. Again, we assume some principles of classical reasoning in these proofs, namely decidability of equality on propositional symbols.

We included a few examples throughout our work to show that there we can actually construct objects of the types we define constructively – in particular, we can construct ordinals such as ω or any polynomial on ω. Our example on propositional logic programming also illustrates that, for the kind of finite structures that are often used in practice, our development applies without resorting to full-blown classical reasoning.

Our development consists of 6850 lines of Coq code divided into 6 files, containing 166 definitions and 426 lemmas. The source code was developed using Coq version 8.18, and is available at `https://zenodo.org/records/10709614` [7].

References

1. Antic, C., Eiter, T., Fink, M.: Hex semantics via approximation fixpoint theory. In: Cabalar, P., Son, T.C. (eds.) Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings. LNCS, vol. 8148, pp. 102–115. Springer (2013), `http://dx.doi.org/10.1007/978-3-642-40564-8_11`
2. Barras, B.: Sets in Coq, Coq in Sets. J. Formaliz. Reason. **3**(1), 29–48 (2010)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)
4. Blanqui, F., Koprowski, A.: Color: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. Math. Struct. Comput. Sci. **21**(4), 827–859 (2011)
5. Bogaerts, B., Cruz-Filipe, L.: Fixpoint semantics for active integrity constraints. Artif. Intell. **255**, 43–70 (2018), `https://doi.org/10.1016/j.artint.2017.11.003`
6. Bogaerts, B., Cruz-Filipe, L.: Stratification in approximation fixpoint theory and its application to active integrity constraints. ACM Trans. Comput. Log. **22**(1), 6:1–6:19 (2021), `https://doi.org/10.1145/3430750`
7. Bogaerts, B., Cruz-Filipe, L.: A formalisation of approximation fixpoint theory in Coq (Feb 2024). `https://doi.org/10.5281/zenodo.10709614`, `https://doi.org/10.5281/zenodo.10709614`
8. Bogaerts, B., Jakubowski, M.: Fixpoint semantics for recursive SHACL. In: Formisano, A., Liu, Y.A., Bogaerts, B., Brik, A., Dahl, V., Dodaro, C., Fodor, P., Pozzato, G.L., Vennekens, J., Zhou, N. (eds.) Proceedings 37th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2021, Porto (virtual event), 20-27th September 2021. EPTCS, vol. 345, pp. 41–47 (2021), `https://doi.org/10.4204/EPTCS.345.14`
9. Bogaerts, B., Vennekens, J., Denecker, M.: Grounded fixpoints and their applications in knowledge representation. Artif. Intell. **224**, 51–71 (2015), `http://dx.doi.org/10.1016/j.artint.2015.03.006`
10. Bogaerts, B., Vennekens, J., Denecker, M.: Safe inductions and their applications in knowledge representation. Artificial Intelligence **259**, 167 – 185 (2018), `http://www.sciencedirect.com/science/article/pii/S000437021830122X`

11. Charalambidis, A., Rondogiannis, P., Symeonidou, I.: Approximation fixpoint theory and the well-founded semantics of higher-order logic programs. Theory Pract. Log. Program. **18**(3-4), 421–437 (2018), `https://doi.org/10.1017/S1471068418000108`
12. Denecker, M., Marek, V., Truszczyński, M.: Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, The Springer International Series in Engineering and Computer Science, vol. 597, pp. 127–144. Springer US (2000), `http://dx.doi.org/10.1007/978-1-4615-1567-8_6`
13. Denecker, M., Marek, V., Truszczyński, M.: Uniform semantic treatment of default and autoepistemic logics. Artif. Intell. **143**(1), 79–122 (2003), `http://dx.doi.org/10.1016/S0004-3702(02)00293-X`
14. Denecker, M., Marek, V., Truszczyński, M.: Reiter's default logic is a logic of autoepistemic reasoning and a good one, too. In: Brewka, G., Marek, V., Truszczyński, M. (eds.) Nonmonotonic Reasoning – Essays Celebrating Its 30th Anniversary, pp. 111–144. College Publications (2011), `http://arxiv.org/abs/1108.3278`
15. Denecker, M., Vennekens, J.: Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In: Baral, C., Brewka, G., Schlipf, J.S. (eds.) LPNMR. Lecture Notes in Computer Science, vol. 4483, pp. 84–96. Springer (2007), `http://dx.doi.org/10.1007/978-3-540-72200-7_9`
16. Fitting, M.: Fixpoint semantics for logic programming — A survey. Theoretical Computer Science **278**(1-2), 25–51 (2002), `http://dx.doi.org/10.1016/S0304-3975(00)00330-3`
17. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K.A. (eds.) ICLP/SLP. pp. 1070–1080. MIT Press (1988), `http://citeseer.ist.psu.edu/viewdoc/summary?nodoi=10.1.1.24.6050`
18. Grall, H.: Proving fixed points. Tech. Rep. hal-00507775, HAL archives ouvertes (2010)
19. Grimm, J.: Implementation of three types of ordinals in Coq. Tech. Rep. RR-8407, INRIA (2013)
20. Heyninck, J., Bogaerts, B.: Non-deterministic approximation operators: ultimate operators, semi-equilibrium semantics and aggregates (full version). CoRR **abs/2305.10846** (2023), `https://doi.org/10.48550/arXiv.2305.10846`
21. de Jong, T., Kraus, N., Forsberg, F.N., Xu, C.: Set-theoretic and type-theoretic ordinals coincide. In: LICS. pp. 1–13 (2023). `https://doi.org/10.1109/LICS56636.2023.10175762`
22. Konolige, K.: On the relation between default and autoepistemic logic. Artif. Intell. **35**(3), 343–382 (1988), `http://dx.doi.org/10.1016/0004-3702(88)90021-5`
23. Kuratowski, C.: Une méthode d'élimination des nombres transfinis des raisonnements mathématiques. Fundamenta Mathematicae **3**(1), 76–108 (1922), `http://eudml.org/doc/213282`
24. Liu, F., Bi, Y., Chowdhury, M.S., You, J., Feng, Z.: Flexible approximators for approximating fixpoint theory. In: Khoury, R., Drummond, C. (eds.) Advances in Artificial Intelligence - 29th Canadian Conference on Artificial Intelligence, Canadian AI 2016, Victoria, BC, Canada, May 31 - June 3, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9673, pp. 224–236. Springer (2016), `http://dx.doi.org/10.1007/978-3-319-34111-8_28`
25. (maintainer), F.B.: `https://github.com/fblanqui/color/blob/master/Util/Relation/Tarski.v`, accessed: 2021-06-02

26. (maintainer), P.C.: `https://github.com/coq-community/hydra-battles/tree/master/theories/ordinals`, accessed: 2021-06-02
27. Moore, R.C.: Semantical considerations on nonmonotonic logic. Artif. Intell. **25**(1), 75–94 (1985), `http://dx.doi.org/10.1016/0004-3702(85)90042-6`
28. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and stable semantics of logic programs with aggregates. TPLP **7**(3), 301–353 (2007), `http://dx.doi.org/10.1017/S1471068406002973`
29. Reiter, R.: A logic for default reasoning. Artif. Intell. **13**(1-2), 81–132 (1980), `http://dx.doi.org/10.1016/0004-3702(80)90014-4`
30. Simpson, C.: Set-theoretical mathematics in Coq. CoRR **abs/math/0402336** (2004)
31. Strass, H.: Approximating operators and semantics for abstract dialectical frameworks. Artif. Intell. **205**, 39–70 (2013), `http://dx.doi.org/10.1016/j.artint.2013.09.004`
32. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics (1955)
33. Truszczyński, M.: Strong and uniform equivalence of nonmonotonic theories - an algebraic approach. Ann. Math. Artif. Intell. **48**(3-4), 245–265 (2006), `http://dx.doi.org/10.1007/s10472-007-9049-2`
34. van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. J. ACM **23**(4), 733–742 (1976), `http://dx.doi.org/10.1145/321978.321991`
35. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM **38**(3), 620–650 (1991), `http://dx.doi.org/10.1145/116825.116838`
36. Van Hertum, P., Cramer, M., Bogaerts, B., Denecker, M.: Distributed autoepistemic logic and its application to access control. In: Kambhampati, S. (ed.) Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016. pp. 1286–1292. IJCAI/AAAI Press (2016), `http://www.ijcai.org/Abstract/16/186`
37. Vennekens, J., Gilis, D., Denecker, M.: Splitting an operator: Algebraic modularity results for logics with fixpoint semantics. ACM Trans. Comput. Log. **7**(4), 765–797 (2006), `http://dx.doi.org/10.1145/1182613.1189735`
38. Vennekens, J., Mariën, M., Wittocx, J., Denecker, M.: Predicate introduction for logics with a fixpoint semantics. Parts I and II. Fundamenta Informaticae **79**(1-2), 187–227 (2007)