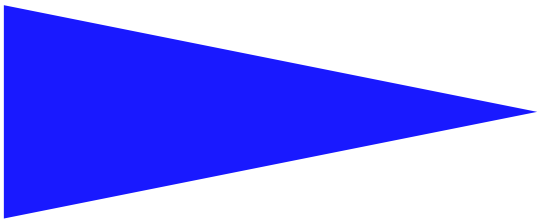
PUBLICATION
INTERNE
N°



CONSTRUCTIVE CONSTRUCTION OF LATTICES FOR WELL-FOUNDED FIXPOINT ITERATION

DAVID PICHARDIE

ISSN 1166-8687

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Constructive construction of lattices for well-founded fixpoint iteration

David Pichardie*

Systèmes symboliques
Projets Lande

Publication interne n — Mai 2005 — 23Future workssection*.86 pages

Abstract: We present a framework for performing fixpoint calculations on lattice structures. A library of Coq module functors is proposed for constructing complex lattices using efficient data structures. The lattice signature contains a well-foundedness proof obligation which ensures termination of generic fixpoint iteration algorithms. With this library, complex well-foundedness proofs can hence be constructed in a functorial fashion. This paper contains two distinct contributions. We first demonstrate the ability of the recent Coq module system in manipulating algebraic structures and extracting efficient Ocaml implementation from them. The second contribution is a generic result, based on the constructive notion of accessibility predicate, about preservation of accessibility properties when combining relations.

Key-words: Formal proof, modules, lattices

(Résumé : tsvp)

* david.pichardie@irisa.fr

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Construction constructive de treillis pour l'itération bien fondée

Résumé : Nous présentons un cadre pour réaliser des calculs de point fixe sur des structures de treillis. Une librairie de foncteurs de module Coq est proposée pour construire des treillis complexes, basées sur des structures de données efficaces. La signature de treillis contient une obligation de preuve de bonne fondaison qui assure la terminaison d'algorithmes génériques de calcul de point fixe. Avec cette librairie, des preuves complexes de terminaison peuvent ainsi être construites, de manière fonctorielle. Ce travail propose deux contributions distinctes. Tout d'abord, il démontre l'aptitude du récent système de module de Coq à manipuler des structures algébriques et à en extraire des structures de donnée efficaces. La seconde contribution est un résultat général, basé sur la notion constructive de prédicat d'accessibilité, à propos de la préservation de la propriété d'accessibilité lors de la combinaison de relations.

Mots clés : Preuve formelle, modules, treillis

1 Introduction

Static program analyses rely on fixpoint computation on lattice structures to solve data flows equations. The basic algorithms are relatively simple but lattice structures can be complex when dealing with realistic programming languages. Termination of these computations relies on specific properties of the lattice structures, like for example the condition that all ascending chain are eventually stationary. In this work we are interested in increasing confidence in static analysers by programming them in the proof-as-programming paradigm: from a machine-checked correctness proof of an analysis, we extract a certified analyser. We use the Coq [1] proof assistant with its extraction mechanism [10] to extract Ocaml [2] programs from constructive proofs. Achievements have already been reported [4, 5] about this approach. They are based on a lattice library which allows the construction of complex lattices in a modular fashion. Large termination proof are in the same way constructed. This paper presents this library based on the new Coq module system [6] and give details about the constructive proofs done for its construction.

Two distinct termination criteria are formalised, each of them associated with a fixpoint computation algorithm. The first algorithm computes the least fixpoint of a monotone operator on a lattices verifying the termination criterion of *the ascending chain condition*. The second computes a post fixpoint of a monotone operator in lattices using a *widening operator*. We present in Section 2 the two module signatures associated with each kind of lattice. The termination criteria used in our formalisation are based on the constructive notion of *accessibility* whose general results are called in Section 3. Sections 4 and 5 then present various lattice functors proposed in the library. Section 4 is devoted to binary functors, giving in particular details about the product functor. Section 5 deals with a functor of functions with various possible implementations. Conclusions are given in Section 7.

The whole Coq development is available on-line [12].

Related works The concept of lattice library was already proposed in the previous works of the author [4, 5], but few details were given about its construction. Neither Coq module nor widening operators were used in these works.

This paper is a descendant of the work of Jones [9] were a modular construction of finite lattices were proposed in the Haskell programming language using type class. Our lattice signatures are not restricted to the type of functions but also provide their specifications. It is a consequence of the expressiveness gap existing between Haskell an Coq type system.

The Coq module system is still young and few proof developments use it. Most example concern the modular construction of data structures, like the implementation of finite sets by Filliâtre and Letouzey [8] and the dictionary example proposed by Bertot and Casteran in their book [3]. Examples of algebraic structure manipulations are currently more difficult to find.

Few detailed constructive proofs about accessibility properties have been published. The reference in this field is the work of Paulson [11] where general rules to preserve accessibility

properties are given. Many proofs done in the present work relies on these rules, but the notion of widening operator has required to extend some of these results.

2 Module signatures

This work is based on two algebraic structures: *the partially ordered set* (poset) and *the lattice*.

Definition 1. ***(Partially ordered sets)*** *A* partially ordered set (poset) *is a 3-upplet* $(A, \equiv, \sqsubseteq)$ *with*

- A *a set,*
- $\equiv$ *a computable equivalence relation on* A*,*
- $\sqsubseteq$ *a computable order relation (relative to the equivalence* $\equiv$*).*

Definition 2. ***(Lattice)*** *A* lattice *is a 6-upplet* $(A, \equiv, \sqsubseteq, \sqcup, \sqcap, \bot)$ *with*

- $(A, \equiv, \sqsubseteq)$ *a poset,*
- $\sqcup$ *the least binary upper bound (with respect to* $\sqsubseteq$*),*
- $\sqcap$ *the greatest binary lower bound (with respect to* $\sqsubseteq$*),*
- $\bot$ *the least element (with respect to* $\sqsubseteq$*)*

We take here a few distance compare to the classical definitions: the role of $\equiv$ is usually played by the standard equality and the existence of a least element is not traditionally necessary in a lattice.

In the framework of static analysis formalisation, A is the domain of abstract values representing properties of the concrete domain. $\sqsubseteq$ gives the relative precision of abstract values, $\sqcup$ et $\sqcap$ gives disjonction and conjonction of properties at the abstract level and $\bot$ is used as initial value for the iterative fixpoint computation.

In Coq, the previous définition is given as a module signature. The Figure 1 gives this signature. The poset signature is similar (signature `Poset`), it only declares the definition relative to `t`, `eq` and `order`. A first consequence of these signature definitions is that the statement "all lattice is a poset" is free in Coq: all module verifying the `Lattice` signature, verifies the `Poset` signature too.

We will need further properties to be able to compute fixpoint on such structures. In this work, we are interested in two properties: *the ascending chain condition* and the existence of *a widening operator*.

```
Module Type Lattice.

  Parameter t : Set.

  Parameter eq : t → t → Prop.
  Parameter eq_refl : ∀ x : t, eq x x.
  Parameter eq_sym : ∀ x y : t, eq x y → eq y x.
  Parameter eq_trans : ∀ x y z : t, eq x y → eq y z → eq x z.
  Parameter eq_dec : ∀ x y : t, {eq x y}+{¬ eq x y}.

  Parameter order : t → t → Prop.
  Parameter order_refl : ∀ x y : t, eq x y → order x y.
  Parameter order_antisym : ∀ x y : t, order x y → order y x → eq x y.
  Parameter order_trans : ∀ x y z : t, order x y → order y z → order x z.
  Parameter order_dec : ∀ x y : t, {order x y}+{¬ order x y}.

  Parameter join : t → t → t.
  Parameter join_bound1 : ∀ x y : t, order x (join x y).
  Parameter join_bound2 : ∀ x y : t, order y (join x y).
  Parameter join_least_upper_bound :
    ∀ x y z : t, order x z → order y z → order (join x y) z.

  Parameter meet : t → t → t.
  Parameter meet_bound1 : ∀ x y : t, order (meet x y) x.
  Parameter meet_bound2 : ∀ x y : t, order (meet x y) y.
  Parameter meet_greatest_lower_bound :
    ∀ x y z : t, order z x → order z y → order z (meet x y).

  Parameter bottom : t.
  Parameter bottom_is_bottom : ∀ x : t, order bottom x.

End Lattice.
```

Figure 1: The `Lattice` signature

2.1 Least fixpoint computation

Definition 3. *(**Classical ascending chain condition**) A poset $(A, \equiv, \sqsubseteq)$ verifies* the ascending chain condition *if all increasing sequence of elements in A is eventually stationary.*

When this condition is verified, the least fixpoint of all monotone operators can be computed.

Lemma 1. *If $f \in A \to A$ is monotone operator on a set A with a lattice structure $(A, \equiv, \sqsubseteq, \sqcup, \sqcap, \bot)$ verifying the ascending chain condition, then the sequence*

$$x_0 = \bot, x_1 = f(x_0), \ldots, x_n = f(x_{n-1}), \ldots$$

is eventually stationary and stabilises on the least fixpoint of f.

In order to implement this algorithm in Coq, we will take an ascending chain condition definition better adapted to constructive proofs. This definition will be based on the notion of *accessibility* and *noetherian* relation.

Definition 4. ***(Accessibility)*** *Given a relation $\prec$ on a set A, the set* $\mathrm{Acc}_\prec$ *of* accessibles *elements with respect to $\prec$ is inductively defined as*

$$\frac{\forall y \in A,\ y \prec x \ \Rightarrow\ y \in \mathrm{Acc}_\prec}{x \in \mathrm{Acc}_\prec}$$

Definition 5. ***(Noetherian relation)*** *A relation $\prec$ on a set A is* noetherian *if all elements in A are accessibles with respect to $\prec$.*

Intuitively, an element is accessible with respect to a relation $\prec$ if it is not the starting point of any infinite $\prec$-decreasing chain. A trivial example of accessible element is an element without predecessor.

Definition 6. ***(Constructive ascending chain condition)*** *A poset $(A, \equiv, \sqsubseteq)$ verifies* the ascending chain condition *if the associated strict inverse order $\sqsupset$ is noetherian.*

In classical logic, the two ascending chain condition criteria are equivalent. The direction "Classical $\Rightarrow$ Constructive" requires the use of choice axiom. In intuitionist logic, there is no more equivalence. Let us take the example of the boolean poset $(\mathbb{B}, =, \sqsubseteq)$ whose Hasse diagram is given in Figure 2(a) ($\mathbb{B} = \{\mathfrak{t}, \mathfrak{f}\}$).

(a) Bi-valuated poset (b) Example of product poset

Theorem 1. *It does not exists any constructive proof of the statement "the poset $(\mathbb{B}, =, \sqsubseteq_{\mathbb{B}})$ verifies the classical ascending chain condition".*

Proof. If such a proof exists, there exists an algorithm which computes, for all monotone function $f \in \mathbb{N} \to \mathbb{B}$ an integer k verifying $\forall n, n \geq k \Rightarrow f(k) = f(n)$. But given a Turing machine M, the function $f_M \in \mathbb{N} \to \mathbb{B}$ which for all integer n returns $\mathfrak{t}$ if M is halted after n steps on the empty input and $\mathfrak{f}$ otherwise, is a monotone function. If we compute an integer k verifying $\forall n, n \geq k \Rightarrow f_M(k) = f_M(n)$, then the value $f_M(k)$ will allow to solve the halting problem of M for the empty imput. □

Theorem 2. *The exists a constructive proof of the statement "the poset* $(\mathbb{B}, =, \sqsubseteq_{\mathbb{B}})$ *verifies the constructive ascending chain condition".*

Proof. $\sqsupset_{\mathbb{B}}$ is reduced to $\{(\mathfrak{t}, \mathfrak{f})\}$. $\mathfrak{t} \in \mathrm{Acc}_{\sqsupset_{\mathbb{B}}}$ because $\mathfrak{t}$ does not possess predecessor with respect to $\sqsupset_{\mathbb{B}}$. $\mathfrak{f} \in \mathrm{Acc}_{\sqsupset_{\mathbb{B}}}$ because its only predecessor is $\mathfrak{t}$ which is an accessible element. □

From now we will only refer to the constructive notion of ascending chain condition. In Coq, the concept of accessibility belongs to the standard library. The `well_founded` definition is the noetherian definition taken in this paper. The notion of poset verifying the ascending chain condition can hence be represented by the module signature given in Figure 2. The signature `LatticeWf` is only the union of the signatures `Lattice` and `PosetWf`. The Figure 3 gives the traduction of the Lemma 1 in Coq.

```
Module Type PosetWf.

  Parameter t : Set.

  Parameter eq : t → t → Prop.
  Parameter eq_refl : ∀ x : t, eq x x.
  Parameter eq_sym : ∀ x y : t, eq x y → eq y x.
  Parameter eq_trans : ∀ x y z : t, eq x y → eq y z → eq x z.
  Parameter eq_dec : ∀ x y : t, {eq x y}+{¬ eq x y}.

  Parameter order : t → t → Prop.
  Parameter order_refl : ∀ x y : t, eq x y → order x y.
  Parameter order_antisym : ∀ x y : t, order x y → order y x → eq x y.
  Parameter order_trans : ∀ x y z : t, order x y → order y z → order x z.
  Parameter order_dec : ∀ x y : t, {order x y}+{¬ order x y}.

  Parameter ascending_chain_condition :
     well_founded (fun x y ⇒ ¬ eq y x ∧ order y x).

End PosetWf.
```

Figure 2: Le `PosetWf` signature

2.2 Post fixpoint computation

The previous algorithm can require an important number of iteration before convergence. Moreover, some lattices useful in static analysis do not respect the ascending chain condition (like the lattice of interval). The solution proposed by Cousot and Cousot [7] is a fixpoint approximation by a post fixpoint. Such a post fixpoint is computed with an algorithm of the form $x_0 = \bot, x_{n+1} = x_n \triangledown f(x_n)$ with $\triangledown$ a binary operator on A which "extrapolates" its two arguments. The computed sequence should be increasing (property ensured if $\triangledown$ verifies $\forall x, y \in A,\ x \sqsubseteq x \triangledown y$) and should over-approximate the classical iteration: $f^n(\bot) \sqsubseteq$

```
Module FixPoint (L:LatticeWf).

  Import L.

  Definition lfp (f:t→t) : monotone f →
      { x:t | eq x (f x) ∧ ∀ y : t, eq y (f y) → order x y } := ...

End FixPoint.
```

Figure 3: Least fixpoint computation

x_n (property ensured if $\triangledown$ verifies $\forall x, y \in A,\ y \sqsubseteq x \triangledown y$). A last condition ensures the computation convergence : after a finite number of steps, we must reach a post fixpoint. The criterion proposed in the literature is generally " for all increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$, the chain $y_0 = x_0, y_{n+1} = y_n \triangledown x_{n+1}$ eventually reachs a rank k with $y_k \equiv y_{k+1}$".

In order to express this criterion with the accessibility notion, we have to find a relation whose infinite chain will be prohibited. Such a relation is defined by $(x_1, y_1) \prec_\triangledown (x_2, y_2)$ iff $x_2 \sqsubseteq x_1 \ \wedge y_1 \equiv y_2 \triangledown x_1 \ \wedge y_1 \not\equiv y_2$. Indeed the following equivalence holds

$$\left(\begin{array}{l}\text{it exists a chain } x_0 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots \\ \text{with } y_0 = x_0, y_{n+1} = y_n \triangledown x_n \\ \text{verifying} \forall k,\ y_k \not\equiv y_{k+1}\end{array}\right) \iff \left(\begin{array}{l}\text{it exists a sequence } ((x_k, y_k))_{k\in\mathbb{N}} \\ \text{verifying } x_0 = y_0 \\ \text{and } \forall k,\ (x_{k+1}, y_{k+1}) \prec_\triangledown (x_k, y_k)\end{array}\right)$$

The classical criterion found in the literature can hence be formulated under the form

$$\forall x \in A,\ (x, x) \in \mathrm{Acc}_{\prec_\triangledown}$$

It is hence a criterion with a different presentation than the previous. We don't require all element to be accessible, only those of the form (x, x) because they are potentially starting points for iteration with widening.

Finally, these properties are collected in the `PosetWiden` interface given in Figure 4. The properties `widen_eq1` and `widen_eq2` ensure that $\triangledown$ respects the equivalence $\equiv$ taken on A. The signature `LatticeWiden` is expressed as the union of the signatures `Lattice` and `PosetWiden`. The Figure 5 gives the construction of the associated generic post fixpoint solver.

2.3 A signature hierarchy

We hence have defined five signatures. The links between these signatures are schematized in Figure 6. A rising arrow represents the inclusion of a signature in an other. The dashed arrow expresses that all lattices verifying the ascending chain condition can be seen as lattices with valid widening operator (take $\triangledown = \sqcup$). This inclusion is not syntactic: in Coq, it must be demonstrated by a functor `LatticeWiden_Of_LatticeWf`oking like

```
Module Type PosetWiden.

  Parameter t : Set.

  Parameter eq : t → t → Prop.
  Parameter eq_refl : ∀ x : t, eq x x.
  Parameter eq_sym : ∀ x y : t, eq x y → eq y x.
  Parameter eq_trans : ∀ x y z : t, eq x y → eq y z → eq x z.
  Parameter eq_dec : ∀ x y : t, {eq x y}+{¬ eq x y}.

  Parameter order : t → t → Prop.
  Parameter order_refl : ∀ x y : t, eq x y → order x y.
  Parameter order_antisym : ∀ x y : t, order x y → order y x → eq x y.
  Parameter order_trans : ∀ x y z : t, order x y → order y z → order x z.
  Parameter order_dec : ∀ x y : t, {order x y}+{¬ order x y}.

  Parameter widen : t → t → t.

  Parameter widen_bound1 : ∀ x y : t, order x (widen x y).
  Parameter widen_bound2 : ∀ x y : t, order y (widen x y).
  Parameter widen_eq1 : ∀ x y z : t, eq x y → eq (widen x z) (widen y z).
  Parameter widen_eq2 : ∀ x y z : t, eq x y → eq (widen z x) (widen z y).

  Definition widen_rel : (t×t) → (t×t) → Prop := fun x y ⇒
    order (fst y) (fst x) ∧
    eq (snd x) (widen (snd y) (fst x)) ∧
    ¬ eq (snd y) (snd x).

  Parameter widen_acc_property : ∀ x : t, Acc widen_rel (x,x).

End PosetWiden.
```

Figure 4: The `PosetWiden` signatures

```
Module PostFixPoint (L:LatticeWiden).

  Import L.

  Definition pfp_widen (f:t→t) : monotone f → { x:t | order (f x) x } := ...

End PostFixPoint.
```

Figure 5: Post fixpoint computation

```
Module LatticeWiden_Of_LatticeWf (L:WfLattice) : LatticeWiden ...
```

The final goal of this work is to construct big modules verifying the signatures `LatticeWf` or `LatticeWiden`. Because these signatures are only union of simpler ones, it is sufficient to construct module of types `Lattice`, `PosetWf` and `PosetWiden`. That is why we will mainly talk about those in the rest of the document.

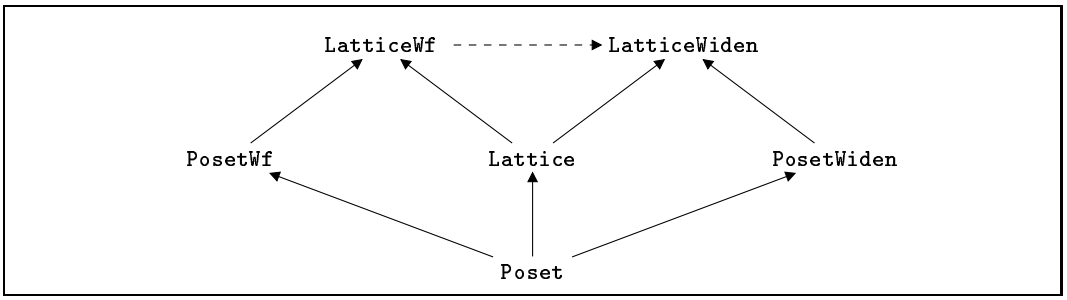


Figure 6: Module signature hierarchy

3 General results about accessibility predicats

The first result about accessibility is its associated induction principle.

Lemma 2. ***(Noetherian induction)*** *Given a property $\mathcal{P}$ on a set A and $\prec$ a relation on A, if $\forall x \in \mathrm{Acc}_\prec,\ (\forall y \in A,\ y \prec x \ \Rightarrow\ \mathcal{P}(y)) \ \Rightarrow\ \mathcal{P}(x)$ then $\forall x \in \mathrm{Acc}_\prec,\ \mathcal{P}(x)$.*

The following results are proved by Paulson in [11] and belong to the Coq standard library.

Lemma 3. *For all relations R_1, R_2 on a set A, if $R_1 \subseteq R_2$ then $\mathrm{Acc}_{R_2} \subseteq \mathrm{Acc}_{R_1}$.*

Lemma 4. *For all relation R_B on a set B, for all function $f : A \to B$ with A a set, the relation R_A defined on A by $R_A = \{(a_1, a_2) \mid (f(a_1), f(a_2)) \in R_B\}$ verifies $\forall a \in A,\ f(a) \in \mathrm{Acc}_{R_B} \ \Rightarrow\ a \in \mathrm{Acc}_{R_A}$.*

These two lemmas can be summarised in the following result

Lemma 5. *For all relations R_A on a set A, R_B on a set B, for all function $f : A \to B$ verifying $\forall (a_1, a_2) \in R_A, (f(a_1), f(a_2)) \in R_B$, we have $\forall a \in A,\ f(a) \in \mathrm{Acc}_{R_B} \ \Rightarrow\ a \in \mathrm{Acc}_{R_A}$.*

Lemma 6. *For all relation R on a set A, $\mathrm{Acc}_R = \mathrm{Acc}_{R^+}$ with R^+ the transitive closure of R.*

Lemma 7. ***(Lexicographic product)*** *For all R_A, R_B on a set A (respectively on B), if R_A and R_B are noetherian, then the* lexicographic product *R defined by*

$$R^{lex} = \{((a_1, b_1), (a_2, b_2)) \mid (a_1, a_2) \in R_A \ \vee \ (a_1 = a_2 \ \wedge (b_1, b_2) \in R_B)\}$$

is noetherian too.

This result can by generalised replacing $a_1 = a_2$ by $a_1 \equiv a_2$ with $\equiv$ an equivalence relation right-compatible with R_A ($\forall a_1, a_2, a_3 \in A,\ (a_1, a_2) \in R_A \wedge a_2 \equiv a_3 \Rightarrow (a_1, a_3) \in R$).

4 Lattice functors

Module functors are high-level functions which take modules as argument and return module. We propose three basic binary functors in our library: the product, the disjoint sum and the lifted sum. For space reason, we will restrict our explanation in this article to the product.

4.1 Poset product

Lemma 8. ***(Poset product)*** *Given two posets $(A, \equiv_A, \sqsubseteq_A)$ and $(B, \equiv_B, \sqsubseteq_B)$, the triplet $(A \times B, \equiv_{A\times B}, \sqsubseteq_{A\times B})$ defined by $\equiv_{A\times B} = \{((a_1, b_1), (a_2, b_2)) \mid a_1 \equiv_A a_2 \ \wedge \ b_1 \equiv_B b_2\}$ and $\sqsubseteq_{A\times B} = \{((a_1, b_1), (a_2, b_2)) \mid a_1 \sqsubseteq_A a_2 \ \wedge \ b_1 \sqsubseteq_B b_2\}$ is a poset, called* poset product.

The Figure 2(b) gives the example of the product of the lattice of Figure 2(a) by itself. In Coq, the Lemma 8 corresponds to a the functor `ProdPoset` definition, given in Figure 7.

```
Module ProdPoset (P1:Poset) (P2:Poset) : Poset
  with Definition t := P1.t × P2.t
  with Definition eq := fun (x y : P1.t × P2.t) ⇒
    P1.eq (fst x) (fst y) ∧ P2.eq (snd x) (snd y)
  with Definition order := fun (x y : P1.t × P2.t) ⇒
    P1.order (fst x) (fst y) ∧ P2.order (snd x) (snd y).

  ...

End ProdPoset.
```

Figure 7: The poset product functor

4.2 The noetherian poset product

The construction of the noetherian poset product functor is given in Figure 8. The interactive definition of this functor is made in three steps. We first give the functor signature with its base type `t`, its equivalence relation `eq` and its order relation `order` using the `with`

```
Module ProdPosetWf (P1:PosetWf) (P2:PosetWf) : PosetWf
  with Definition t := P1.t × P2.t
  with Definition eq := fun (x y : P1.t × P2.t) ⇒
    P1.eq (fst x) (fst y) ∧ P2.eq (snd x) (snd y)
  with Definition order := fun (x y : P1.t × P2.t) ⇒
    P1.order (fst x) (fst y) ∧ P2.order (snd x) (snd y).

  Module Pos := ProdPoset P1 P2.
  Definition t := Pos.t.
  Definition eq := Pos.eq.
  Definition eq_refl := Pos.eq_refl.
  Definition eq_sym := Pos.eq_sym.
  Definition eq_trans := Pos.eq_trans.
  Definition eq_dec := Pos.eq_dec.
  Definition order := Pos.order.
  Definition order_refl := Pos.order_refl.
  Definition order_antisym := Pos.order_antisym.
  Definition order_trans := Pos.order_trans.
  Definition order_dec := Pos.order_dec.

  Lemma ascending_chain_condition :
   well_founded (fun x y ⇒ ¬ eq y x ∧ order y x).
  Proof.
    ...
  Qed.

End ProdPosetWf.
```

Figure 8: The noetherian poset product functor

notation. In a second time, we construct the definitions dealing with the poset part using the poset product functor `ProdPoset`. This part is equivalent to the notation `include ProdPoset(P1)(P2)` available in Ocaml module system[1]. We have to remark that in the expression `ProdPoset(P1)(P2)`, modules `P1` and `P2` are used as module of type `Poset`. The signature inclusion of `Poset` into `PosetWf` allows this use. The last step concerns the new part of this functor: the proof that $\sqsupset_{A\times B}$ is noetherian. It is proved using the results given in Section 3. We just have to note that $\sqsupset_{A\times B}$ is a sub-relation of the lexicographic product between $\sqsupset_A$ and $\sqsupset_B$ (using $\equiv_A$ as equivalence relation on A). We hence can conclude using Lemmas 3 and 7.

[1] But currently not available in Coq.

4.3 Widening product

Contrary to the previous proof, terminaison proofs about widening product no more relies on well-known results. The key lemma to be established is

Lemma 9. *Given two posets* $(A, \equiv_A, \sqsubseteq_A)$ *and* $(B, \equiv_B, \sqsubseteq_B)$*, two binary operators* $\triangledown_A$ *and* $\triangledown_B$ *on* A *and* B*, if* $\forall a \in A, (a,a) \in \mathrm{Acc}_{\prec_{\triangledown_A}}$ *and* $\forall b \in B, (b,b) \in \mathrm{Acc}_{\prec_{\triangledown_B}}$ *then the operator* $\triangledown_{A\times B}$ *defined by*

$$(a_1, b_1)\ \triangledown_{A\times B}\ (a_2, b_2) = (a_1 \triangledown_A a_2, b_1 \triangledown_B b_2)$$

verifies $\forall c \in A \times B, (c,c) \in \mathrm{Acc}_{\prec_{\triangledown_{A\times B}}}$.

The notation $\prec_\triangledown$ was given in page 8.

This result requires a technical proof to be directly established (because by example, it relies on pairs of pair). We can make a more general proof and express the current problem as a particular case. The idea consists in express $\prec_{\triangledown_{A\times B}}$ as a lexicographic product between $\prec_{\triangledown_A}$ and $\prec_{\triangledown_B}$. We then have to prove a result looking like

$$\forall a \in \mathrm{Acc}_{\lhd_A},\ \forall b \in \mathrm{Acc}_{\lhd_B},\ (a,b) \in \mathrm{Acc}_{\lhd^{\mathrm{lex}}_{A\times B}}$$

with $\lhd_A$ playing the role of $\prec \triangledown_A$ and $\lhd_B$ the one of $\prec \triangledown_B$. With the standard lexicographic product definition, the result is generally false.

Lemma 10. *Given two relations* $\lhd_A$ *and* $\lhd_B$ *on sets* A *and* B*, if* $a \in \mathrm{Acc}_{\lhd_A}$ *and* $b \in \mathrm{Acc}_{\lhd_B}$*, if there exists* $b' \in B$ *such that* $b' \notin \mathrm{Acc}_{\lhd_B}$ *and* $a' \in A$ *such that* $a' \lhd_A a$ *then* $(a,b) \notin \mathrm{Acc}_{\lhd^{lex}_{A\times B}}$.

Proof. Let us suppose that $(a,b) \in \mathrm{Acc}_{\lhd^{\mathrm{lex}}_{A\times B}}$ and show that $b' \in \mathrm{Acc}_{\lhd_B}$ then holds. Because $(a',b') \lhd^{\mathrm{lex}}_{A\times B} (a,b)$, we have $(a',b') \in \mathrm{Acc}_{\lhd^{\mathrm{lex}}_{A\times B}}$. But it implies $b' \in \mathrm{Acc}_{\lhd_B}$. Indeed, if we consider the function $f : B \to (A \times B)$ defined by $f(x) = (a', x)$, we have

$$\forall b_1, b_2 \in B,\ b_1 \lhd_B b_2 \ \Rightarrow\ f(b_1) \lhd^{\mathrm{lex}}_{A\times B} f(b_2)$$

so we can apply the Lemma 5 with $f(b') = (a', b')$ to conclude. □

The problem here is that we can take any element b' to obtain a predecessor (a', b') of (a, b). The case $a_1 \lhd_A a_2$ in the definition of $\lhd^{\mathrm{lex}}_{A\times B}$ is hence too weak. We have to make restriction on b_1 and b_2. We use for it a relation $\blacktriangleleft_B$ and propose a new product looking like

$$(a_1, b_1) \lhd^{\mathrm{lex}} (a_2, b_2) \iff \begin{array}{c} (a_1 \lhd_A a_2 \wedge b_1 \blacktriangleleft_B b_2) \\ \vee \\ (a_1 = a_2\ \wedge b_1 \lhd_B b_2) \end{array}$$

adding a constraint between $\lhd_B$ and $\blacktriangleleft_B$ to prevent to have any b' as previously: if $b_2 \blacktriangleleft_B b_1$ and $b_1 \in \mathrm{Acc}_{\lhd_B}$ then b_2 should stay in $\mathrm{Acc}_{\lhd_B}$. We will take a simpler sufficient condition (without accessibility notion inside):

$$\forall b_1, b_2, b_3 \in B, b_1 \lhd_B b_2 \wedge b_2 \blacktriangleleft_B b_3 \ \Rightarrow\ b_1 \lhd^+ b_3$$

We can even propose a symmetric definition and encompass the case of $\sqsubseteq_{A\times B}$ (where $a_1 = a_2$ was replaced by $a_1 \equiv_A a_2$) by introducing a relation $\blacktriangleleft_A$ verifying a similar property than $\blacktriangleleft_B$.

Definition 7. ***(Extended lexicographic product)*** *Given two pairs of relations $\lhd_A$ et $\blacktriangleleft_A$ on a set A, $\lhd_B$ and $\blacktriangleleft_B$ on B. The* extended lexicographic product *is the relation $\lessdot\!\!\blacktriangleleft^{lex(\lhd_A,\lhd_B,\blacktriangleleft_A,\blacktriangleleft_B)}$ defined on $A \times B$ by*

$$(a_1,b_1)\lessdot\!\!\blacktriangleleft^{lex(\lhd_A,\lhd_B,\blacktriangleleft_A,\blacktriangleleft_B)}(a_2,b_2) \iff \begin{array}{c}(a_1\lhd_A a_2 \ \wedge\ b_1\blacktriangleleft_B b_2)\\ \vee \\ (a_1\blacktriangleleft_A a_2 \ \wedge\ b_1\lhd_B b_2)\end{array}$$

with the following conditions

$$\forall a_1,a_2,a_3 \in A,\ a_1\lhd_A a_2 \ \wedge\ a_2\blacktriangleleft_A a_3 \ \Rightarrow\ a_1\lhd_A^+ a_3 \tag{1}$$

$$\forall b_1,b_2,b_3 \in B,\ b_1\lhd_B b_2 \ \wedge\ b_2\blacktriangleleft_B b_3 \ \Rightarrow\ b_1\lhd_B^+ b_3 \tag{2}$$

When the context will allow us to do it without ambiguity, we will note $\lessdot\!\!\blacktriangleleft$ this relation.

Example 1.

$$(a_1,b_1)\sqsupset_{A\times B}(a_2,b_2) \iff \begin{array}{c}(a_1 \sqsupset_A a_2 \wedge b_1 \sqsupseteq b_2)\\ \vee \\ (a_1 \equiv_A a_2 \wedge b_1 \sqsupset b_2)\end{array}$$

and we have

$$\forall a_1,a_2,a_3 \in A,\ a_1 \sqsupset_A a_2 \ \wedge\ a_2 \equiv_A a_3 \ \Rightarrow\ a_1 \sqsupset_A a_3$$

and

$$\forall b_1,b_2,b_3 \in B,\ b_1 \sqsupset_B b_2 \ \wedge\ b_2 \sqsupseteq_B b_3 \ \Rightarrow\ b_1 \sqsupset_B^+ b_3$$

Then $\sqsupset_{A\times B} = \lessdot\!\!\blacktriangleleft^{lex(\sqsupset_A,\sqsupset_B,\equiv_A,\sqsupseteq_B)}$.

Theorem 3. *If $\lhd_A$, $\lhd_B$, $\blacktriangleleft_A$ and $\blacktriangleleft_B$ verify the hypotheses of the previous definition, then for all $a \in \mathrm{Acc}_{\lhd_A}$ and $b \in \mathrm{Acc}_{\lhd_B}$, $(a,b) \in \mathrm{Acc}_{\lessdot\!\!\blacktriangleleft}$.*

Proof. The form of this statement encourages to use a noetherian induction on the property

$$\forall a \in A,\ \mathcal{P}(a) := \text{"}\forall b \in \mathrm{Acc}_{\lhd_B},\ (a,b) \in \mathrm{Acc}_{\lessdot\!\!\blacktriangleleft}\text{"}$$

But the generated induction principle will be too weak because only usable for an element $b \in \mathrm{Acc}_{\lhd_B}$.

We hence take a stronger goal by proving

$$\forall a \in \mathrm{Acc}_{\lhd_A^+},\ \forall b_1 \in \mathrm{Acc}_{\lhd_B},\ \forall b_2 \in B,\ b_2\blacktriangleleft_B^* b_1,\ \Rightarrow\ (a,b_2) \in \mathrm{Acc}_{\lessdot\!\!\blacktriangleleft} \tag{3}$$

with $\blacktriangleleft_B^*$ the reflexive transitive closure of $\blacktriangleleft_B$. This result is sufficient to establish our theorem because $\blacktriangleleft_B^*$ is reflexive and because $\mathrm{Acc}_{\lhd_A} = \mathrm{Acc}_{\lhd_A^+}$ (using Lemma 6). To prove

(3), we use a noetherian induction on a and $\lhd_A^+$. We consider an element $a_1 \in \mathrm{Acc}_{\lhd_A^+}$ such that

$$\begin{array}{l} \forall a_2 \in A,\ a_2 \lhd_A^+ a_1\ \Rightarrow \\ \quad \forall b_1 \in \mathrm{Acc}_{\lhd_B},\ \forall b_2 \in B,\ b_2 \blacktriangleleft_B^* b_1,\ \Rightarrow\ (a_2, b_2) \in \mathrm{Acc}_{\lll} \end{array} \tag{4}$$

and search to prove $\forall b_1 \in \mathrm{Acc}_{\lhd_B},\ \forall b_2 \in B,\ b_2 \blacktriangleleft_B^* b_1,\ \Rightarrow\ (a_1, b_2) \in \mathrm{Acc}_{\lhd}$ Once time again a direct proof by induction on b_1 will be useless without first enforcing the current goal. We prove instead:

$$\begin{array}{l} \forall b_1 \in \mathrm{Acc}_{\lhd_B^+}, \\ \quad \forall b_2 \in B,\ b_2 \blacktriangleleft_B^* b_1,\ \Rightarrow \\ \quad \forall a_3 \in A,\ a_3 \blacktriangleleft_A^* a_1\ \Rightarrow\ (a_3, b_2) \in \mathrm{Acc}_{\lll} \end{array} \tag{5}$$

It is a sufficient result because $\mathrm{Acc}_{\lhd_B} = \mathrm{Acc}_{\lhd_B^+}$ and $\blacktriangleleft_A^*$ is reflexive. We prove (5) by noetherian induction on b_1. Hence we take an element $b_1 \in \mathrm{Acc}_{\lhd_B^+}$ such that

$$\begin{array}{l} \forall b_1 \in \mathrm{Acc}_{\lhd_B^+},\ \forall b_3 \in B,\ b_3 \lhd_B^+ b_1\ \Rightarrow \\ \quad \forall b_2 \in B,\ b_2 \blacktriangleleft_B^* b_3,\ \Rightarrow \\ \quad \forall a_3 \in A,\ a_3 \blacktriangleleft_A^* a_1\ \Rightarrow\ (a_3, b_2) \in \mathrm{Acc}_{\lll} \end{array} \tag{6}$$

and we search now a proof for

$$\forall b_2 \in B,\ b_2 \blacktriangleleft_B^* b_1,\ \Rightarrow\ \forall a_3 \in A,\ a_3 \blacktriangleleft_A^* a_1\ \Rightarrow\ (a_3, b_2) \in \mathrm{Acc}_{\lll}$$

Let us suppose

$$b_1 \in \mathrm{Acc}_{\lhd_B^+} \tag{7}$$

$$b_2 \blacktriangleleft_B^* b_1 \tag{8}$$

$$a_3 \blacktriangleleft_A^* a_1 \tag{9}$$

and prove $(a_3, b_2) \in \mathrm{Acc}_{\lll}$. We have to consider a predecessor (a_4, b_4)

$$(a_4, b_4) \lll (a_3, b_2) \tag{10}$$

and we have to prove $(a_4, b_4) \in \mathrm{Acc}_{\lll}$. Using the definition of $\lll$, two cases results from hypothesis (10).

- Case 1 :

$$a_4 \lhd_A a_3 \tag{11}$$

$$b_4 \blacktriangleleft_B b_2 \tag{12}$$

We can use the induction hypothesis (4). Three conditions must be verified

- $a_4 \lhd_A a_1$: true using hypotheses (1), (9) and (11).
- $b_1 \in \mathrm{Acc}_{\lhd_B}$: true using (7) and the general result $\mathrm{Acc}_{\lhd_B^+} = \mathrm{Acc}_{\lhd_B}$
- $b_4 \blacktriangleleft_B^* b_1$: true because of (12) and (8)

- Case 2 :

$$a_4 \blacktriangleleft_A a_3 \tag{13}$$
$$b_4 \lhd_B b_2 \tag{14}$$

This time, we can use the induction hypothesis (6). Three conditions must be verified

- $b_4 \lhd_A^+ b_1$: true because of the hypotheses (2), (8) and (14)
- $b_4 \blacktriangleleft_B^* b_4$: true by reflexivity of $\blacktriangleleft_B^*$
- $a_4 \blacktriangleleft_A^* a_1$: true using hypotheses (9) and (13)

Hence is established the main result. □

To prove the Lemma 9, we only have to use the Lemma 5 taking $f : (A \times B) \times (A \times B) \to (A \times A) \times (B \times B)$ defined by $f\left((a_1, b_1), (a_2, b_2)\right) = \left((a_1, a_2), (b_1, b_2)\right)$ and considering the relation $\prec_{\nabla_{A\times B}}$ on $(A \times B) \times (A \times B)$ and $\blacktriangleleft^{\mathrm{lex}(\prec_{\nabla_A}, \prec_{\nabla_B}, \preccurlyeq_{\nabla_A}, \preccurlyeq_{\nabla_B})}$ on $(A \times A) \times (B \times B)$ where $\preccurlyeq_{\nabla_A}$ and $\preccurlyeq_{\nabla_B}$ are defined by $(x_1, y_1) \preccurlyeq_{\nabla_A} (x_2, y_2) \iff x_2 \sqsubseteq_A x_1 \wedge y_1 \equiv y_2$ and $(x_1, y_1) \preccurlyeq_{\nabla_B} (x_2, y_2) \iff x_2 \sqsubseteq_B x_1 \wedge y_1 \equiv_B y_2 \nabla_B x_1$. It is not difficult to verify that hypotheses of Theorem 3 are fulfilled and then conclude.

4.4 Lattice product

The construction of the lattice product functor is simpler than the previous because it does not require termination proof. The header of the corresponding functor is given in Figure 9.

5 Function lattice

An other important functor concerns functions because underlying data structures are complex and as a consequence termination proof are difficult. We will describe now the functor of noetherian poset associated. The widening functor follows a similar approach and the lattice functor is easy to construct. The noetherian poset functor is based on the following mathematic result.

Lemma 11. ***(Function noetherian poset)*** *For all finite set A and all noetherian poset $(B, \equiv_B, \sqsubseteq_B)$, the triplet $(A \to B, \equiv_{A\to B}, \sqsubseteq_{A\to B})$ is a noetherian poset, with $f_1 \equiv_{A\to B} f_2 \iff \forall a, f_1(a) \equiv_B f_2(a)$ and $f_1 \sqsubseteq_{A\to B} f_2 \iff \forall a, f_1(a) \sqsubseteq_B f_2(a)$.*

Take care that A must be finite, otherwise the result is false.

This result is not satisfying for our works because we want to be able to reason on efficient function implementations. We will then prove the previous result "for all function implementation".

```
Module ProdLattice (P1:Lattice) (P2:Lattice) : Lattice
  with Definition t := P1.t × P2.t
  with Definition eq := fun (x y : P1.t × P2.t) ⇒
    P1.eq (fst x) (fst y) ∧ P2.eq (snd x) (snd y)
  with Definition order := fun (x y : P1.t × P2.t) ⇒
    P1.order (fst x) (fst y) ∧ P2.order (snd x) (snd y)
  with Definition join := fun (x y : P1.t × P2.t) ⇒
    (P1.join (fst x) (fst y), P2.join (snd x) (snd y))
  with Definition meet := fun (x y : P1.t × P2.t) ⇒
    (P1.meet (fst x) (fst y), P2.meet (snd x) (snd y))
  with Definition bottom := (P1.bottom, P2.bottom).

  ...

End ProdLattice.
```

Figure 9: Lattice product functor

```
Module Type Func_FiniteSet_PosetWf.

  Declare Module A : FiniteSet.
  Declare Module B : PosetWf.

  Parameter t : Set.

  Parameter get : t → A.t → B.t.

  Definition eq : t → t → Prop := fun f1 f2 ⇒
    ∀ a1 a2 : A.t, A.eq a1 a2 → B.eq (get f1 a1) (get f2 a2).
  Parameter eq_refl : ∀ x : t, eq x x.
  Parameter eq_dec : ∀ x y : t, {eq x y}+{¬ eq x y}.

  Definition order : t → t → Prop := fun f1 f2 ⇒
    ∀ a1 a2 : A.t, A.eq a1 a2 → B.order (get f1 a1) (get f2 a2).
  Parameter order_dec : ∀ x y : t, {order x y}+{¬ order x y}.

End Func_FiniteSet_PosetWf.
```

Figure 10: Function implementation signature

5.1 Function implementation

The notion of function implementation is given in Figure 10. This signature handles

- a module `A` with a signature `FiniteSet` (associated with the function domain). The `FiniteSet` signature is given in Figure 11. It represents set in bijection with parts

$[\![0, \texttt{cardinal}-1]\!]$ of $\mathbb{Z}$. Our library propose finite set functors (product, list of bounded length) and a base finite set module (binary number on 32 bits).

- a poset module `B` associated with codomain.
- an abstract type `t` used to represent functions.
- a function `get` where `(get F a)` gives the image of `a:A.t` for the function associated with the element `F:t`.
- fixed equivalence (`eq`) and order (`order`) relation definitions with their test implementations (`eq_dec` and `order_dec`).
- the property `eq_refl` ensures that `get` respects the equivalence relation `A.eq` taken on `A.t`.

```
Module Type FiniteSet.

  Parameter t : Set.

  Parameter eq : t → t → Prop.
  Parameter eq_refl : ∀ x : t, eq x x.
  Parameter eq_sym : ∀ x y : t, eq x y → eq y x.
  Parameter eq_trans : ∀ x y z : t, eq x y → eq y z → eq x z.
  Parameter eq_dec : ∀ x y : t, {eq x y}+{¬ eq x y}.

  Parameter cardinal : Z.
  Parameter cardinal_positive : cardinal > 0.

  Parameter inject : t → Z.
  Parameter nat2t : Z → t.
  Parameter inject_bounded : ∀ x : t, 0 ≤ (inject x) < cardinal.
  Parameter inject_nat2t : ∀ n : Z, 0 ≤ n < cardinal → inject (nat2t n) = n.
  Parameter inject_injective : ∀ x y : t, inject x = inject y → eq x y.
  Parameter inject_comp_eq : ∀ x y : t, eq x y → inject x = inject y.

End FiniteSet.
```

Figure 11: Finite Set signature

5.2 A noetherian poset of functions

A functor `FuncPosetWf` allows to construct noetherian poset structures on top of any function implementation (see Figure 12). The termination proof is done using a function f between `F.t` and $\mathbb{N} \to$ `B.t`.

```
f := fun (x : F.t) => fun (n : nat) => get x (F.A.nat2t (Z_of_nat n))
```

The noetherian relation used on $\mathbb{N} \to$ `B.t` is a generalised lexicographic product: given a set A and a noetherian relation $\prec$ on it, the relation $\prec_n$ defined on $\mathbb{N} \to A$ by

$$f_1 \prec_n f_2 \iff \exists k,\ k < n \ \wedge\ (\forall i, i < n \Rightarrow f_1(i) = f_2(i)) \ \wedge\ f_1(k) \prec f_2(k)$$

is noetherian for all integer n.

```
Module FuncPosetWf (F:Func_FiniteSet_PosetWf) : PosetWf
  with Definition t := F.t
  with Definition eq := fun (f1 f2 : F.t) ⇒
       ∀ a1 a2 : F.A.t, F.A.eq a1 a2 → F.B.eq (get f1 a1) (get f2 a2)
  with Definition order := fun (f1 f2 : F.t) ⇒
       ∀ a1 a2 : F.A.t, F.A.eq a1 a2 → F.B.order (get f1 a1) (get f2 a2).

  ...

End FuncPosetWf.
```

Figure 12: Functor to construct noetherian poset of functions

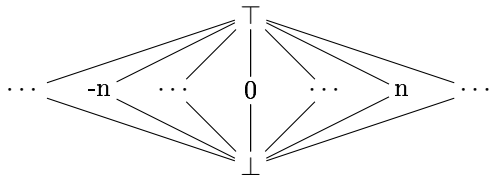
5.3 Two efficient implementations

We propose two kinds of function implementation in our library. The first is a specific implementation for functions whose domain is bounded binary integer (each integer denotes a position in a tree). The second allows to take any finite set (possibly constructed by some finite set functors) as domain. It is based on an abstract implementation of Ocaml maps. We currently propose a sorted list implementation and plan an implementation with balanced tree, both based on the previous formalisation done in [8].

6 An example of complex lattice construction

When formalising static analyses for realistic languages, required lattices can be complex. We give now the example of a control flow analysis for bytecode Java. The lattice shape follows the concrete semantic domain.

Values manipulated by the virtual machine are numerics or references. We first abstract numerics with the constant lattice whose Hasse diagram is:



A functor in the library allows to construct flat lattices from sets of value. A set is only a type with a computable equivalence relation (signature SetSign). A functor `SetSign_of_SetSimplSign` allow to construct `SetSign` module using only the standard equality as equivalence relation.

```
Module ZSet <: SetSimplSign.
  Definition t := Z.
  Definition eq_dec := Z_eq_dec.
End ZSet.

Module ZSetSign := SetSign_of_SetSimplSign ZSet.
Module Num <: LatticeWf := FlatLattice ZSetSign.
```

Concerning references, we use an abstraction by set of class names. A special module in the library proposes such a construction (class names are represented by binary integers on 32 bits), using tree representation.

```
Module Ref <: LatticeWf := FiniteSetLatticeWf.
```

We then take the disjoint sum of the two lattices to obtain the value abstract domain.

```
Module Val <: LatticeWf := SumLiftLatticeWf Num Ref.
```

Local variables are abstracted by functions from variable name to abstract value. Variable names are represented by binary integers on 32 bits, hence we use a tree representation with position coded by binary integer.

```
Module LocalVar <: LatticeWf :=  ArrayBinLatticeWf Val.
```

Operand stacks are abstracted by list of abstract values.

```
Module OperandStack <: LatticeWf :=  ListLiftLatticeWf Val.
```

An abstract heap is a function from (ClassName $\times$ FieldName) to abstract values. Using a curryfied view, we compose twice the `ArrayBinLatticeWf` functor.

```
Module Heap' <: LatticeWf :=  ArrayBinLatticeWf Val.
Module Heap <: LatticeWf :=  ArrayBinLatticeWf Heap'.
```

A state is then a triplet composed by an operand stack, an array of local variables and a heap.

```
Module State' <: LatticeWf := ProdLatticeWf LocalVar Heap.
Module State <: LatticeWf := ProdLatticeWf OperandStack State'.
```

To perform a context sensitive analysis, we abstract the call stack by a bounded list of pair in (MethodName × ProgPoint).

```
Module Context' <: FiniteSet := ProdFiniteSet WordFiniteSet WordFiniteSet.

Module N5 <:  NAT.
  Definition val : nat := (5)%nat.
End N5.

Module Context <: FiniteSet := ListFiniteSet N5 Context'.
```

The global state is finally a function from context to abstract state. We use a sorted list implementation for maps.

```
Module ContextOT <: FiniteSetOT := FiniteSetOT_Of_FiniteSet Context.

Module Map := FMapList.Make ContextOT.OT.

Module GlobalState <: LatticeWf := MapLatticeWf ContextOT Map State.
```

7 Conclusion

We have presented a framework for performing fixpoint calculations on lattice structures. In order to construct complex lattices, we propose a library of Coq module functors. We have particularly focus our explanations on the product and the function functor but other functors are available in the Coq development. In Section 6 we have given an example of a complex lattice construction.

The first contribution is to give a nice example about the use of the Coq module system. This example mixes algebraic structures and efficient data structures, which seems to be a perfect playground for the module system: algebraic structures defined in Coq become data structures when extracted in Ocaml. The actual module system is however limited, prohibiting first-class modules. An alternative is to used Coq records like

```
Record Poset : Type := { t : Set; eq : t -> t -> Prop; ... }
```

but contrary to modules they don't have nice counterparts in the Ocaml type system (no type definition in records). To construct our lattice library it has sometimes been necessary to first manipulate the "record version" of the lattice structures and then convert them into modules. Records were hence used as intermediary tools.

The second contribution concerns constructive proofs about termination properties. The termination criterion used with widening operators has required extensions of previous

known results about accessibility predicates. We hope the proofs presented here could give good example for people who want to do similar constructions in theorem provers for type theory.

Future works A first extension would be to use narrowing operators [7] to enhance the post fixpoint computation done by widenings. Similar construction than those currently proposed should be necessary.

An other extension concerns the construction of base lattices, those which are used to instantiate lattice functors and construct bigger lattices. We think some automation could be proposed to quickly construct finite lattices with their correctness proofs starting from a text description of their Hasse diagram.

References

[1] The Coq Proof Assistant. http://coq.inria.fr/.

[2] The Objective Caml language. http://caml.inria.fr/.

[3] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[4] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a Data Flow Analyser in Constructive Logic. In *Proc. of 13th European Symposium on Programming (ESOP'04)*, number 2986 in Lecture Notes in Computer Science, pages 385–400. Springer-Verlag, 2004.

[5] David Cachera, Thomas Jensen, David Pichardie, and Gerardo Schneider. Certified Memory Usage Analysis. In *Proc. of 13th International Symposium on Formal Methods (FM'05)*, Lecture Notes in Computer Science. Springer-Verlag, 2005. to appear.

[6] Jacek Chrząszcz. Implementation of modules in the Coq system. In *Proc. of 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, volume 2758 of *LNCS*, pages 270–286. Springer-Verlag, 2003.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of 4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, New York, 1977.

[8] Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In *Proc. of 13th European Symposium on Programming (ESOP'04)*, number 2986 in Lecture Notes in Computer Science, pages 370–384. Springer-Verlag, 2004.

[9] Mark P. Jones. Computing with lattices: An application of type classes. *Journal of Functional Programming*, 2(4):475–503, October 1992.

[10] Pierre Letouzey. A New Extraction for Coq. In *Types for Proofs and Programs, Second International Workshop, (TYPES'02)*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[11] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2(4):325–355, December 1986.

[12] David Pichardie. Coq sources of the development. http://www.irisa.fr/lande/pichardie/CarmelCoq/.