Building certified static analysers by modular construction of well-founded lattices

David Pichardie

Building certified static analysers by modular construction of well-founded lattices

David Pichardie

INRIA Rennes - Bretagne Atlantique, France

Abstract

This paper presents fixpoint calculations on lattice structures as example of highly modular programming in a dependently typed functional language. We propose a library of Coq module functors 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 implementations 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.

Keywords: Proof assistant, Constructive proofs, Static analysis.

1 Introduction

Static program analyses rely on fixpoint computations 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, as for example the condition that all ascending chains are eventually stationary. In this work, we aim at increasing confidence in static analysers by using the proof-as-programs paradigm: from a machine-checked correctness proof of an analysis, we extract a certified analyser. We use the extraction mechanism of the Coq proof assistant to extract Ocaml programs from constructive proofs. In earlier work, we presented a lattice library which allows the construction of complex lattices in a modular fashion [3]. It was shown how this library was used to construct large termination proofs based on the ascending chain condition. This paper presents a new version of this library, based now on the more general termination criteria of widening.

We first present in Section 2 the module signature that models the kind of lattice we want to build. In Section 3 we motivate this library with a challenging example of lattice to be built in Coq. Sections 4 and 5 then present various lattice functors proposed in the library. Section 4 discusses binary functors, in particular the product functor. Section 5 deals with a functor of functions with various possible implementations. Conclusions are given in Section 6.

We expect the reader to be familiar with the ML module system. The whole Coq development is available on-line [1]. The theorems in this paper are given without proofs but they can be found in a companion report [16].

Related work

This paper is a descendent of the work of Jones [10] where a modular construction of finite lattices was proposed in the Haskell programming language using type classes. Our lattice signatures are not restricted to ML function types but they are also equipped with a specification. This is a consequence of the expressiveness gap existing between the Haskell and the Coq type systems.

In earlier work, we already introduced the lattice library [3]. However, we mainly discussed the semantic proofs required for *certified analyses*. Only ascending chain conditions proofs were studied and few details were given about their constructions. The current paper proposes several improvements:

- Mechanical proofs about fixpoint iteration using widenings has never been reported before. Other existing works only deal with ascending chain condition [11,2,4,9]. Widening operators require more complex termination proofs.
- In particular, new theoretical results about accessibility predicates are necessary to handle product of widening operators in the constructive logic of Coq.
- We propose a modular notion of functions (see Section 5) which allows to construct termination proofs without relying on the actual implementation chosen. Previous proofs were specific to one implementation, and as a consequence it was very difficult to adapt them to new function implementations.

The technical contribution of this paper deals with the modular construction of large proof terminations in a proof assistant. Proving termination of static analysers is sometimes considered as useless because we only need to check the result of the analyser, if it terminates. Nevertheless, bugs concerning termination of fixpoint iteration are difficult to debug: when do you stop the analyser ? Because of their non-monotonous nature, widening operators break human intuition and sometimes leads to invalid termination proofs (as noticed by Antoine Miné [13] as regards [17]).

Few detailed constructive proofs about accessibility properties have been published. The reference in this field is the work of Paulson [15] where general rules to preserve accessibility properties are given. Many of our proofs depend on these rules, however the notion of widening operator required further extensions. As far as we know the result proved in Theorem **??** is new.

2 Module signatures for lattices

This work is based on two algebraic structures: *partially ordered sets* (posets) and *lattices* (see [7] for standard definitions). To be precise we consider a more general

[1] http://www.irisa.fr/lande/pichardie/lattice/main.html

notion that posets because the posets $(A, \equiv, \sqsubseteq)$ we consider in this paper are in fact composed of a set A and a pre-order $\sqsubseteq$. $\equiv$ is the associated equivalence relation.

In Coq, the corresponding definitions are given as module signatures (see Figure 1). The `Poset` signature reads as follow: a module of type (or signature) `Poset` must at least contain a type `t` (to model elements in the posets) and two relations `eq` and `order`. It must also contains proofs that `eq` (resp `order`) is an equivalence relation (resp. a partial order). These required proofs are represented with the keyword **`Axiom`**. At last the two relation `eq` and `order` must come with a computable test function `eq_dec` and `order_dec`. The type of the operator `eq_dec` is a dependent type that expresses the following: for any `x` and `y` of type `t`, the function must return a boolean such that, if the boolean is true , `x` and `y` are equivalent, if it is false, they are not.

The `Lattice` signature includes all elements of the `Poset` signature with the command **`Include`** `Poset`[2]. A first consequence of these signature definitions is that the statement "every lattice is a poset" is free in Coq: a module satisfying the `Lattice` signature, satisfies the `Poset` signature too.

```
Module Type Poset.
  Parameter t : Set.

  Parameter eq : t → t → Prop.
  Axiom eq_refl : ∀ x : t, eq x x.
  Axiom eq_sym : ∀ x y : t, eq x y → eq y x.
  Axiom 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.
  Axiom order_refl : ∀ x y : t, eq x y → order x y.
  Axiom order_antisym : ∀ x y : t, order x y → order y x → eq x y.
  Axiom 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}.
End Poset.

Module Type Lattice.
  Include Poset.

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

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

  Parameter bottom : t.
  Axiom bottom_is_bottom : ∀ x : t, order bottom x.
End Lattice.
```

Fig. 1. The lattice signature

We will need a further property to be able to compute over-approximation of fixpoints in such structures. In our previous work [3] we considered the ascending chain condition but in this work we are interested in more general criterion: the existence of *a widening operator*.

[2] This command is not currently available in the Coq system. It should be replaced by the complete list of elements found in the module.

The standard fixpoint iteration *à la* Kleene may require an important number of iterations before convergence. Moreover, some lattices used in static analysis do not respect the ascending chain condition (like the lattice of intervals used in Section 3). The solution proposed by Cousot and Cousot [6] is a fixpoint approximation by a post fixpoint. Such a post fixpoint is computed with an algorithm of the form $x_0 = \bot$, and $\forall n, x_{n+1} = x_n \nabla f(x_n)$ with ∇ a binary operator on A which "extrapolates" its two arguments. The computed sequence should be increasing (property ensured if ∇ satisfies $\forall x, y \in A,\ x \sqsubseteq x \nabla y$) and should over-approximate the classical iteration: $f^n(\bot) \sqsubseteq x_n$ (property ensured if ∇ satisfies $\forall x, y \in A,\ y \sqsubseteq x \nabla y$). 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 chains $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$, the chain $y_0 = x_0, y_{n+1} = y_n \nabla x_{n+1}$ eventually reaches a rank k with $y_k \equiv y_{k+1}$".

In order to implement this algorithm in Coq, we will work with a definition which is better adapted to constructive proofs. This definition will be based on the notion of *accessibility* and of *noetherian* [3] relation [1].

Definition 2.1 (**Accessibility**)
Given a relation $\prec$ on a set A, the set $\mathrm{Acc}_\prec$ of *accessibles* elements with respect to $\prec$ are inductively defined as

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

Definition 2.2 (**Noetherian relation**) A relation $\prec$ on a set A is *noetherian* if all elements in A is accessible 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.

In order to express this widening criterion with the accessibility notion, we need to define a relation where infinite chains will be prohibited. Such a relation is defined by $(x_1, y_1) \prec_\nabla (x_2, y_2) \quad \text{iff} \quad x_2 \sqsubseteq x_1 \ \wedge y_1 \equiv y_2 \nabla x_1 \ \wedge y_1 \not\equiv y_2$. Then, the following equivalence holds

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

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

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

Note that we do not require all elements to be accessible, only those of the form (x, x) because they are potential starting points for iteration with widening.

Finally, these properties are collected in the `PosetWiden` interface given in Figure 2. The properties `widen_eq1` and `widen_eq2` ensure that ∇ respects the equivalence $\equiv$ taken on A. The definition of the signature `LatticeWiden` (lattice with a widening operator) is expressed in a similar way.

[3] The Coq library uses the inappropriate name of well-founded relation.

Figure 3 gives the construction of the associated generic post fixpoint solver. This module is a functor that takes in argument a module of type `LatticeWiden` and build an operator `pfp_widen` that computes post fixpoint. It is expressed in the type of the operator as follow: given a function `f`, if `f` is monotone then the function return an element `x` in the lattice that is a post fixpoint.

```
Module Type PosetWiden.
  Include Poset.

  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.
```

Fig. 2. The module signature for poset with a widening operator

```
Module PostFixPoint (L:LatticeWiden).
  Definition monotone f := ∀ x y, L.order x y → L.order (f x) (f y).
  Definition pfp_widen f : monotone f → { x:L.t | L.order (f x) x } :=
    (* ... omitted ... *)
End PostFixPoint.
```

Fig. 3. Postfixpoint computation

3 A challenging example

When formalizing analyses for realistic programming language, the underlying lattice may be complex, even for analyses of middle precision. We give here an example of such lattice in order to motivate and illustrate our lattice library.

The aim of this lattice is to abstract the memory of a Java virtual machine with a context-sensitive interval abstraction for numerical values and context-sensitive class abstraction for references. Because in Java, values are numerics or references it is natural to abstract them with a sum of lattice, here the sum of the set of class name and the interval domain [6].

$$\text{Value}^\sharp = \mathcal{P}(\text{ClassName}) + \text{Interval}$$

The global structure of the lattice is then of the form:

$$\text{State}^\sharp = \text{Local}^\sharp \times \text{Heap}^\sharp$$

with L and H some function domains of the form:

$$\text{Local}^\sharp = \text{Context} \to \left(\left(\text{Value}^\sharp\right)^\star \times \left(\text{Var} \to \text{Value}^\sharp\right)\right)$$

$$\text{Heap}^\sharp = \text{ClassName} \rightarrow \left(\text{FieldName} \rightarrow \text{Value}^\sharp\right)$$

Var, ClassName, FieldName, MethodName and ProgPoint represent here the (finite) sets of variable name, class name, field name, method name and program points. All this set are encoded with integers on 32 bits. The set Context is composed of list of couples in MethodName $\times$ ProgPoint. These lists have at most k elements and represent the last k call sites.

$$\text{Context} = (\text{MethodName} \times \text{ProgPoint})^{*\leq k}$$

L is the flow-sensitive local abstraction of operand stack and local variables. H is the flow-insensitive abstraction of the heap.

$$\begin{aligned}
\text{Value}^\sharp &= \mathcal{P}(\text{ClassName}) + \text{Interval} \\
\text{Context} &= (\text{MethodName} \times \text{ProgPoint})^{*\leq k} \\
\text{Local}^\sharp &= \text{Context} \rightarrow \left(\left(\text{Value}^\sharp\right)^\star \times \left(\text{Var} \rightarrow \text{Value}^\sharp\right)\right) \\
\text{Heap}^\sharp &= \text{ClassName} \rightarrow \left(\text{FieldName} \rightarrow \text{Value}^\sharp\right) \\
\text{State}^\sharp &= \text{Local}^\sharp \times \text{Heap}^\sharp
\end{aligned}$$

The global domain St admit a lattice structure with a widening operator. Thanks to our lattice library it can be simply built by composition of functors. The construction is presented in Figure 4. For Value$^\sharp$ we use a functor `SumLiftLatticeWiden` that builds the disjoint sum of two lattices. We build H with the function functor presented in Section 5. The `MapLatticeWiden` functor allows to build function with a complex codomain (here `Contexte`). Its utilisation (corresponding to line 8 to 11) will be explained in Section 5. The final lattice is built with the product functor `ProdLatticeWiden` presented in the next section.

```
1  Module Val := SumLiftLatticeWiden(IntervalLattice)(FiniteSetLatticeWiden).
2
3  Module H := ArrayBinLatticeWiden(ArrayBinLatticeWiden(Val)).
4
5  Module LocalVar := ArrayBinLatticeWiden Val.
6  Module Stack := ListLiftLatticeWiden Val.
7
8  Module N5. Definition val : nat := 5. End N5.
9  Module Context := ListFiniteSet(N5)(ProdFiniteSet(WordFiniteSet)(WordFiniteSet)).
10 Module Map := FMapList.Make Context.
11 Module L := MapLatticeWiden(Context)(Map)(ProdLatticeWiden(Stack)(LocalVar)).
12
13 Module GlobalState := ProdLatticeWiden(L)(H).
```

Fig. 4. Construction of the global lattice in Coq

We will now present some of the functors introduced during this example. We will generally focus on the poset part (with widening) of the modules because the operators specific to lattice do not require any technical details.

4 Lattice functors

We propose three basic binary functors in our library: the product, the disjoint sum and the lifted sum. Due to lack of space, we will restrict our explanations in this paper to the product.

4.1 Poset product

Lemma 4.1 ***(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*.*

In Coq, Lemma 4.1 corresponds to a functor which takes two modules of signature `Poset` and returns a module respecting the `Poset` signature for the product structure.

4.2 The poset-with-widening product

```
Module ProdPosetWiden (P1:PosetWiden) (P2:PosetWiden) : PosetWiden
  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 widen := fun (x y : (P1.t * P2.t)) ⇒
    match (x,y) with
      ((x1,x2),(y1,y2)) ⇒ (P1.widen x1 y1, P2.widen x2 y2)
    end.

  Include  ProdPoset(P1)(P2).

  Definition widen (x y : t) :=
    match (x,y) with
      ((x1,x2),(y1,y2)) ⇒ (P1.widen x1 y1, P2.widen x2 y2)
    end.
  ...

  Lemma widen_acc_property : ∀ x:t, Acc widen_rel (x,x).
  Proof. ... Qed.

End ProdPosetWiden.
```

Fig. 5. The poset-with-widening product functor

The construction of the poset-with-widening product functor is given in Figure 5. 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`, its order relation `order` and the considered widening operator using the **`with`** notation. In a second step, we construct the definitions dealing with the poset part using the poset product functor `ProdPoset`. Note that in the expression `ProdPoset(P1)(P2)`, modules `P1` and `P2` are used as module of type `Poset`. The signature inclusion of `Poset` into `PosetWiden` allows this use without requiring any proof of coercion. This is a convenient feature when manipulating nested algebraic structures.

The last step concerns the new part of this functor: the proof that the widening operator satisfies its termination criterion. In our previous work [3] the termination criterion for the product of noetherian poset (*i.e.* that satisfy the ascending chain condition) was proved using a classical result about lexicographic products, but it is not possible for widening operators. Indeed, the key lemma to be established is:

Lemma 4.2 *Given two posets* $(A, \equiv_A, \sqsubseteq_A)$ *and* $(B, \equiv_B, \sqsubseteq_B)$*, two binary operators* ∇_A *and* ∇_B *on* A *and* B*, if* $\forall a \in A, (a, a) \in \mathrm{Acc}_{\prec_{\nabla_A}}$ *and* $\forall b \in B, (b, b) \in \mathrm{Acc}_{\prec_{\nabla_B}}$

then the operator $\nabla_{A\times B}$ *defined by*

$$(a_1, b_1) \nabla_{A\times B} (a_2, b_2) = (a_1 \nabla_A a_2, b_1 \nabla_B b_2)$$

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

Theorem Given two lattices $(A, \sqsubseteq_A, \sqcup_A, \sqcap_A, \nabla_A)$ and $(B, \sqsubseteq_B, \sqcup_B, \sqcap_B, \nabla_B)$ the operator $\nabla_{A\times B}$ defined by

$$(a_1, b_1) \nabla_{A\times B} (a_2, b_2) = (a_1 \nabla_A a_2, b_1 \nabla_B b_2)$$

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

This result in standard when proved in classical logic [6]. In constructive logic, it has not been proved before (as far as we know). It requires a technical proof to be directly established (because by example, it relies on pairs of pairs). We can make a more general proof and express the current problem as a particular case. The idea consists in expressing $\prec_{\nabla_{A\times B}}$ as a lexicographic product between $\prec_{\nabla_A}$ and $\prec_{\nabla_B}$. We then have to prove a result of the form

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

with $\lhd_A$ playing the role of $\prec_{\nabla_A}$ and $\lhd_B$ the one of $\prec_{\nabla_B}$. However if $\lhd^{\mathrm{lex}}_{A\times B}$ denotes the standard lexicographic product of the two relations, the result is generally false:

Lemma 4.3 *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 exist* $b' \in B$ *such that* $b' \notin \mathrm{Acc}_{\lhd_B}$ *and* $a' \in A$ *such that* $a' \lhd_A a$ *then* $(a, b) \notin \mathrm{Acc}_{\lhd^{lex}_{A\times B}}$.

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 restrictions on b_1 and b_2. To this purpose, we introduce a relation $\blacktriangleleft_B$ and propose a new product of the form

$$(a_1, b_1) \lhd^{\mathrm{lex}} (a_2, b_2) \iff (a_1 \lhd_A a_2 \text{ and } b_1 \blacktriangleleft_B b_2) \text{ or } (a_1 = a_2 \text{ and } b_1 \lhd_B b_2)$$

adding a constraint between $\lhd_B$ and $\blacktriangleleft_B$ to prevent having 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 (requiring no accessibility):

$$\forall b_1, b_2, b_3 \in B, b_1 \lhd_B b_2 \text{ and } b_2 \blacktriangleleft_B b_3 \text{ implies } b_1 \lhd^+ b_3$$

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 $\blacktriangleleft_A$ satisfying a similar property than $\blacktriangleleft_B$.

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

$$(a_1,b_1)\lhd\!\!\blacktriangleleft^{\mathrm{lex}(\lhd_A,\lhd_B,\blacktriangleleft_A,\blacktriangleleft_B)}(a_2,b_2) \iff$$
$$(a_1 \lhd_A a_2 \ \text{ and } \ b_1 \blacktriangleleft_B b_2) \ \text{ or } \ (a_1 \blacktriangleleft_A a_2 \ \text{ and } \ b_1 \lhd_B b_2)$$

with the following conditions

$$\forall a_1,a_2,a_3 \in A,\ a_1 \lhd_A a_2 \ \text{ and } \ a_2 \blacktriangleleft_A a_3 \text{ implies } a_1 \lhd_A^+ a_3 \tag{1}$$
$$\forall b_1,b_2,b_3 \in B,\ b_1 \lhd_B b_2 \ \text{ and } \ b_2 \blacktriangleleft_B b_3 \text{ implies } b_1 \lhd_B^+ b_3 \tag{2}$$

Theorem If $\lhd_A$, $\lhd_B$, $\blacktriangleleft_A$ and $\blacktriangleleft_B$ satisfy the hypotheses of the previous definition, then for all $a \in \mathrm{Acc}_{\lhd_A}$ and $b \in \mathrm{Acc}_{\lhd_B}$, $(a,b) \in \mathrm{Acc}_{\lhd\!\!\blacktriangleleft^{\mathrm{lex}(\lhd_A,\lhd_B,\blacktriangleleft_A,\blacktriangleleft_B)}}$.

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

Example 4.4 The standard lexicographic product is a special case of $\lhd\!\!\blacktriangleleft$.

$$(a_1,b_1) \sqsupset_{A\times B} (a_2,b_2) \iff (a_1 \sqsupset_A a_2 \text{ and } b_1 \sqsupseteq b_2) \ \text{ or } \ (a_1 \equiv_A a_2 \text{ and } b_1 \sqsupset b_2)$$

and we have

$$\forall a_1,a_2,a_3 \in A,\ a_1 \sqsupset_A a_2 \ \text{ and } \ a_2 \equiv_A a_3 \text{ implies } a_1 \sqsupset_A a_3$$

and

$$\forall b_1,b_2,b_3 \in B,\ b_1 \sqsupset_B b_2 \ \text{ and } \ b_2 \sqsupseteq_B b_3 \text{ implies } b_1 \sqsupset_B^+ b_3$$

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

To prove Lemma 4.2, we only have to use a *measure function* $f : (A\times B)\times(A\times B) \to (A\times A)\times(B\times B)$ 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 $\lhd\!\!\blacktriangleleft^{\mathrm{lex}\left(\prec_{\nabla_A},\prec_{\nabla_B},\preccurlyeq_{\nabla_A},\preccurlyeq_{\nabla_B}\right)}$ 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 satisfy that hypotheses of Theorem **??** are fulfilled and then conclude.

5 Lattices of functions

Another important functor concerns functions. Static analyses make heavy use of functions during their computations. Efficiency of the underlying data structures is hence crucial. However proof of termination properties on complex data structures can be hard. This section proposes an abstract notion of function implementation for which we prove termination properties. These proof can then be used for several efficient implementations. We now describe the functor which builds a poset with widening for functions.

First, we remark that implementing functions with the native functions of the chosen functional programming language is not a reasonable solution. It is better to use encoding as association lists, balanced trees, ... We will then prove the termination criterion of widening "for all function implementations".

5.1 Function implementation

```
Module Type Func_FiniteSet_PosetWiden.
  Declare Module A : FiniteSet.
  Declare Module B : PosetWiden.

  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).
  Axiom eq_refl : ∀ x : t, eq x x.
  Axiom 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_PosetWiden.
```

Fig. 6. Function implementation signature

The notion of function implementation is given in Figure 6. This signature handles

- a module[4] `A` with a signature `FiniteSet` (associated with the function domain). The `FiniteSet` signature is given in Figure 7. It represents set in bijection with parts $[\![0, \texttt{cardinal} - 1]\!]$ of $\mathbb{Z}$. Our library proposes 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`).

[4] Modules can handles modules. The corresponding signature element is then introduced by **`Declare Module`**.

- the property `eq_refl` ensures that `get` is compatible with the equivalence relation `A.eq` taken on `A.t`.

```
Module Type FiniteSet.
  Parameter t : Set.

  Parameter eq, eq_dec [...]
  Axiom eq_refl, eq_sym, eq_trans [...]

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

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

Fig. 7. FiniteSet signature

5.2 A widening operator on functions

Now for any function implementations we build a poset with a standard widening operator. For functions in $A \to B$ this operator is defined as

$$\forall f_1, f_2 \in A \to B, \forall a \in A,\ (f_1 \nabla f_2)(a) = f_1(a) \nabla_A f_2(a)$$

The proof of the termination criterion relies on the Theorem **??** and the finiteness of the codomain.

5.3 Two efficient implementations

We propose two function implementations in our library. The first is a specific implementation for functions whose domain is a bounded binary integer (each integer denotes a position in a tree [14]). This kind of efficient implementation is heavily used in Leroy's certified compiler [12]. The second implementation is based on an abstract implementation of Ocaml maps. We have adapted the Ocaml signature to Coq and proven that any map fulfils the `Func_FiniteSet_PosetWiden` signature. We currently propose a sorted list implementation and plan an implementation with balanced tree, both based on the previous formalisation done in [8] [5] . Maps can by built on any finite set. Finite sets can be constructed with the previously enumerated functors.

To conclude this section, we finish by commenting the example presented in Figure 4. `Context` is a module of type `FiniteSet` that is built with the functor `ListFiniteSet`. `N5` is a module that encapsulate the natural number 5. We hence bound our lists with at most 5 elements. `MapLatticeWiden` is a functor that take as argument a finite set (here `Context`), a map implementation (here `Map` built with sorted list) and a lattice with a widening operator. It builds the expected lattice and its widening operator.

[5] Balanced trees are a keystone of the industrial-task Astre static analyser [5].

6 Conclusion

We have presented a framework for programming fixpoint computations on lattice structures in a dependently typed functional language. In order to construct complex lattices, we propose a library of Coq module functors. We focused our explanations on the product and the function functor, but other functors are available in our Coq development.

The main contribution of this work deals with constructive proofs of termination properties. The termination criteria used with widening operators has required extensions of previousy known results about accessibility predicates. Termination proofs are often very difficult to do in a proof assistant. This library shows the benefit of modular reasoning to handle such complex proofs. By composing the various functors that we propose, it is now possible to easily construct termination proofs for deep structures with efficient extracted data structures in Ocaml.

We have more recently extend our library to handle narrowing operators [6]. Again the technical difficulty relies in the functor product. It is interesting to notice that termination criterion of the narrowing operator is proved with the Theorem **??**. It confirms that this theoretical result was a cornerstone for our work.

We imagine two extension for our library. The first one concerns the construction of base lattices, those which are used to instantiate lattice functors and construct bigger lattices. Some automation could be proposed to quickly construct finite lattices with their correctness proofs starting from a text description of their Hasse diagram. The second one concerns Galois connexion that could be constructed in the same modular way.

References

[1] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–. North-Holland Publishing Company, 1977.

[2] Gilles Barthe and Guillaume Dufay. A tool-assisted framework for certified bytecode verification. In *FASE'04*, LNCS, pages 99–113, 2004.

[3] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a Data Flow Analyser in Constructive Logic. *Theoretical Computer Science*, 342(1):56–78, September 2005.

[4] Solange Coupet-Grimal and William Delobel. A uniform and certified approach for two static analyses. In *TYPES*, 2004. To appear.

[5] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *ESOP'05*, volume 3444 of *LNCS*, pages 21–30, 2005.

[6] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL'79*, pages 269–282. ACM Press, New York, 1979.

[7] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[8] Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In *ESOP'04*, number 2986 in LNCS, pages 370–384. Springer-Verlag, 2004.

[9] Benjamin Grgoire, Yves Bertot, and Xavier Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *TYPES*, 2004. To appear.

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

[11] Gerwin Klein and Tobias Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298(3):583–626, 2002.

[12] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *ACM POPL'06*. ACM Press, 2006. To appear.

[13] Antoine Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[14] Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *Proc. of the ACM SIGPLAN Workshop on ML*, pages 77–86, 1998.

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

[16] David Pichardie. Building certified static analysers by modular construction of well-founded lattices. Technical Report ????, INRIA, 2008. Companion report, `http://www.irisa.fr/lande/pichardie/lattice/report.pdf`.

[17] Ran Shaham, Elliot K. Kolodner, and Shmuel Sagiv. Automatic removal of array memory leaks in java. In *CC'00*, LNCS, pages 50–66, 2000.