 Electronic Notes in Theoretical Computer Science 212 (2008) 225–239 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

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 demonstrates the ability of the recent Coq module system in manipulating algebraic structures and extracting efficient Ocaml implementations from them. The second contribution of this work is a generic result, based on the con- structive notion of accessibility predicate, about preservation of accessibility properties when combining relations.

*Keywords:* Proof assistant, Constructive proofs, Static analysis.

# Introduction

Static program analyses rely on fixpoint computations on lattice structures to solve data flows equations. The basic algorithms are relatively simple, but lattice struc- tures can be complex when dealing with realistic programming languages. Termi- nation 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 [[4](#_bookmark37)]. 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.

1571-0661 © 2008 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2008.04.064

We first present in Section [2](#_bookmark1) the module signature that models the kind of lattice we want to build. In Section [3](#_bookmark6) we motivate this library with a challenging example of lattice to be built in Coq. Sections [4](#_bookmark9) and [5](#_bookmark28) then present various lattice functors proposed in the library. Section [4](#_bookmark9) discusses binary functors, in particular the product functor. Section [5](#_bookmark28) deals with a functor of functions with various possible implementations. Conclusions are given in Section [6](#_bookmark32).

We expect the reader to be familiar with the ML module system. The whole

Coq development is available on-line [1](#_bookmark0) .

Related work

This paper is a descendent of the work of Jones [[10](#_bookmark43)] 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 [[4](#_bookmark37)]. 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 re- ported before. Other existing works only deal with ascending chain condition [[11](#_bookmark44),[2](#_bookmark34),[5](#_bookmark38),[3](#_bookmark35)]. 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](#_bookmark28)) which allows to con- struct 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 Min´e [[13](#_bookmark46)] as regards [[16](#_bookmark49)]). Few detailed constructive proofs about accessibility properties have been pub- lished. The reference in this field is the work of Paulson [[15](#_bookmark48)] 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 [4.6](#_bookmark15) is new.

1 <http://www.irisa.fr/lande/pichardie/lattice/>

# Module signatures for lattices

This work is based on two algebraic structures: *partially ordered sets* (posets) and *lattices* (see [[8](#_bookmark41)] for standard definitions). To be precise we consider a more general notion that posets because the posets (*A,* ≡*,* ±) we consider in this paper are in fact composed of a set *A* and a pre-order ±. ≡ is the associated equivalence relation.

In Coq, the corresponding definitions are given as module signatures (see Fig- ure [1](#_bookmark4)). 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 contain 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 relations 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, otherwise if it is false, they are not.

The Lattice signature includes all elements of the Poset signature with the com- mand **Include** Poset [2](#_bookmark2) . 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.

We will need a further property to be able to compute over-approximation of fixpoints in such structures. In our previous work [[4](#_bookmark37)] we considered the ascending chain condition but in this work we are interested in more general criterion: the existence of *a widening operator*.

The standard fixpoint iteration *`a 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 Sec- tion [3](#_bookmark6)). The solution proposed by Cousot and Cousot [[7](#_bookmark40)] is a fixpoint approximation by a post fixpoint. Such a post fixpoint is computed with an algorithm of the form *x*0 = ⊥, and ∀*n, xn*+1 = *xn*o*f* (*xn*) with o a binary operator on *A* which ”extrap- olates” its two arguments. The computed sequence should be increasing (property ensured if o satisfies ∀*x, y* ∈ *A, x* ± *x*o*y*) and should over-approximate the classical iteration: *fn*(⊥) ± *xn* (property ensured if o satisfies ∀*x, y* ∈ *A, y* ± *x*o*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 ± *x*1 ± ··· ± *xn* ± ··· *,* the chain *y*0 = *x*0*, yn*+1 = *yn*o*xn*+1 eventually reaches a rank *k* with *yk* ≡ *yk*+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](#_bookmark3) relation [[1](#_bookmark36)].

Definition 2.1 (Accessibility) Given a relation ≺ on a set *A*, the set Acc≺ of

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.

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

**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

*accessibles* elements with respect to ≺ are inductively defined as

∀*y* ∈ *A, y* ≺ *x* ⇒ *y* ∈ Acc≺

*x* ∈ Acc≺

Definition 2.2 (Noetherian relation) A relation ≺ on a set *A* is *noetherian* if all elements in *A* is accessible with respect to ≺.

Intuitively, an element is accessible with respect to a relation ≺ if it is not the starting point of any infinite ≺-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) ≺o (*x*2*, y*2) iff *x*2 ± *x*1 ∧ *y*1 ≡ *y*2o*x*1 ∧ *y*1 /≡ *y*2. Then, the following equivalence holds

there exists a chain *x*0 ± ··· ± *xn*+1 ± · · · with *y*0 = *x*0*,* and ∀*n, yn*+1 = *yn*o*xn* satisfying ∀*k, yk* /≡ *yk*+1

there exists a sequence ((*xk, yk*))*k*∈N

⇐⇒ satisfying *x*0 = *y*0

and ∀*k,* (*xk*+1*, yk*+1) ≺o (*xk, yk*)

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

∀*x* ∈ *A,* (*x, x*) ∈ Acc≺o

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 Fig- ure [2](#_bookmark5). The properties widen\_eq1 and widen\_eq2 ensure that o respects the equivalence

≡ taken on *A*. The definition of the signature LatticeWiden (lattice with a widening operator) is expressed in a similar way.

Figure [3](#_bookmark7) 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 fixpoints. It is expressed in the type of the operator as follow: given a function f, if f is monotone then the function returns 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

# A challenging example

When formalizing analyses for realistic programming language, the underlying lat- tice 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 [[7](#_bookmark40)].

Val = *℘*(ClassName) + Interval

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

*St* = *L* × *H*

with *L* and *H* some function domains of the form:

*L* = Context → ((Val)*٨* × (Var → Val)) *H* = ClassName → (FieldName → Val)

Var, ClassName, FieldName, MethodName and ProgPoint represent here the (fi- nite) 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 com- posed of list of couples in MethodName × ProgPoint. These lists have at most *k* elements and represent the last *k* call sites.

Context = (MethodName × ProgPoint)∗≤*k*

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

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](#_bookmark8). For Val we use a functor SumLiftLatticeWiden that builds the disjoint sum of two lattices. We build *H* with the function functor presented in Section [5](#_bookmark28). The MapLatticeWiden functor allows to build function with a complex codomain (here Context). Its utilisation (corresponding to line 8 to 11) will be explained in Section [5](#_bookmark28). The final lattice is built with the product functor ProdLatticeWiden presented in the next section.

1

**Module** Val := SumLiftLatticeWiden(IntervalLattice)(FiniteSetLatticeWiden).

**Module** H := ArrayBinLatticeWiden(ArrayBinLatticeWiden(Val)).

**Module** LocalVar := ArrayBinLatticeWiden Val.

**Module** Stack := ListLiftLatticeWiden Val.

**Module** N5. **Definition** val : nat := 5. **End** N5.

**Module** Context := ListFiniteSet(N5)(ProdFiniteSet(WordFiniteSet)(WordFiniteSet)).

**Module** Map := FMapList.Make Context.

**Module** L := MapLatticeWiden(Context)(Map)(ProdLatticeWiden(Stack)(LocalVar)).

**Module** GlobalState := ProdLatticeWiden(L)(H).

2

3

4

5

6

7

8

9

10

11

12

13

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 that are specific to lattice do not require any technical details.

# 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.

* 1. *Poset product*

Lemma 4.1 *(Poset product) Given two posets* (*A,* ≡*A,* ±*A*) *and* (*B,* ≡*B,* ±*B*)*, the triplet* (*A* × *B,* ≡*A*×*B,* ±*A*×*B*) *deﬁned by* ≡*A*×*B*= {((*a*1*, b*1)*,* (*a*2*, b*2)) | *a*1 ≡*A a*2 ∧ *b*1 ≡*B b*2} *and* ±*A*×*B*= {((*a*1*, b*1)*,* (*a*2*, b*2)) | *a*1 ±*A a*2 ∧ *b*1 ±*B b*2} *is a poset, called* poset product*.*

In Coq, Lemma [4.1](#_bookmark10) corresponds to a functor which takes two modules of sig- nature Poset and returns a module respecting the Poset signature for the product structure.

* 1. *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)

**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.

**with Definition** widen := **fun** (x y : (P1.t \* P2.t)) ⇒

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

The construction of the poset-with-widening product functor is given in Figure [5](#_bookmark11). 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 [[4](#_bookmark37)] 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,* ≡*A,* ±*A*) *and* (*B,* ≡*B,* ±*B*)*, two binary operators* o*A and* o*B on A and B, if* ∀*a* ∈ *A,* (*a, a*) ∈ Acc≺o*A and* ∀*b* ∈ *B,* (*b, b*) ∈ Acc≺o*B then the operator* o*A*×*B deﬁned by*

(*a*1*, b*1) o*A*×*B* (*a*2*, b*2)= (*a*1o*A a*2*, b*1o*B b*2)

*satisﬁes* ∀*c* ∈ *A* × *B,* (*c, c*) ∈ Acc≺o*A*×*B .*

This result in standard when proved in classical logic [[7](#_bookmark40)]. In constructive logic, it has never 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 ≺o*A*×*B* as a lexicographic product between ≺o*A* and

≺o*B* . We then have to prove a result of the form

∀*a* ∈ Acc¢ *,* ∀*b* ∈ Acc¢ *,* (*a, b*) ∈ Acc lex

*A B* ¢

*A*×*B*

with D*A* playing the role of ≺o*A* and D*B* the one of ≺o*B* . However if Dlex

*A*×*B*

denotes

the standard lexicographic product of the two relations, the result is generally false:

Lemma 4.3 *Given two relations* D*A and* D*B on sets A and B, if a* ∈ Acc¢*A and*

*b* ∈ Acc¢*B , if there exist b*' ∈ *B such that b*' /∈ Acc¢*B and a*' ∈ *A such that a*'D*Aa*

*then* (*a, b*) /∈ Acc *lex .*

¢

*A*×*B*

Proof. Let us suppose that (*a, b*) ∈ Acc lex

¢

*A*×*B*

and show that *b*' ∈ Acc¢*B* then holds.

Because (*a*'*, b*')Dlex

*A*×*B*

(*a, b*), we have (*a*'*, b*') ∈ Acc lex

*A*×*B*

¢

. But it implies *b*' ∈ Acc¢*B* .

Indeed, if we consider the function *f* : *B* → (*A* × *B*) defined by *f* (*x*)= (*a*'*, x*), we

have

∀*b*1*, b*2 ∈ *B, b*1D*Bb*2 ⇒ *f* (*b*1)Dlex

*A*×*B*

*f* (*b*2)

so we can apply the Lemma [A.3](#_bookmark50) 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*1D*Aa*2 in the definition of Dlex

*A*×*B*

is hence too weak. We

have to make restrictions on *b*1 and *b*2. To this purpose, we introduce a relation (*B*

and propose a new product of the form

(*a*1*, b*1)Dlex(*a*2*, b*2) ⇐⇒ (*a*1D*Aa*2 and *b*1(*Bb*2) or (*a*1 = *a*2 and *b*1D*Bb*2)

adding a constraint between D*B* and (*B* to prevent having any *b*' as previously: if *b*2(*Bb*1 and *b*1 ∈ Acc¢*B* then *b*2 should stay in Acc¢*B* . We will take a simpler sufficient condition (requiring no accessibility):

∀*b*1*, b*2*, b*3 ∈ *B, b*1D*Bb*2 and *b*2(*Bb*3 implies *b*1D+*b*3

We can even propose a symmetric definition and encompass the case of ±*A*×*B* (where *a*1 = *a*2 was replaced by *a*1 ≡*A a*2) by introducing a relation (*A* satisfying a similar property than (*B*.

Definition 4.4 (Extended lexicographic product) Given two pairs of relations D*A* and (*A* on a set *A*, D*B* and (*B* on *B*. The *extended lexicographic product* is the relation D(lex(¢*A,*¢*B,*{*A,*{*B* ) defined on *A* × *B* by

(*a*1*, b*1)D(lex(¢*A,*¢*B ,*{*A,*{*B* )(*a*2*, b*2) ⇐⇒

(*a*1D*Aa*2 and *b*1(*Bb*2) or (*a*1(*Aa*2 and *b*1D*Bb*2)

with the following conditions

∀*a*1*, a*2*, a*3 ∈ *A, a*1D*Aa*2 and *a*2(*Aa*3 implies *a*1D+*a*3 (1)

*A*

∀*b*1*, b*2*, b*3 ∈ *B, b*1D*Bb*2 and *b*2(*Bb*3 implies *b*1D+*b*3 (2)

*B*

When the context will allow us to do it without ambiguity, we will use D( to denote this relation.

Example 4.5 The standard lexicographic product is a special case of D(.

(*a*1*, b*1) N*A*×*B* (*a*2*, b*2) ⇐⇒ (*a*1 N*A a*2 and *b*1 ± *b*2) or (*a*1 ≡*A a*2 and *b*1 N *b*2) and we have

∀*a*1*, a*2*, a*3 ∈ *A, a*1 N*A a*2 and *a*2 ≡*A a*3 implies *a*1 N*A a*3

and

*B*

∀*b*1*, b*2*, b*3 ∈ *B, b*1 N*B b*2 and *b*2 ±*B b*3 implies *b*1 N+ *b*3

Then N*A*×*B* = D(lex(+*A,*+*B,*≡*A,*±*B*).

Theorem 4.6 *If* D*A,* D*B,* (*A and* (*B satisfy the hypotheses of the previous deﬁ- nition, then for all a* ∈ Acc¢*A and b* ∈ Acc¢*B ,* (*a, b*) ∈ Acc¢{*.*

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

∀*a* ∈ *A,* P(*a*) := ”∀*b* ∈ Acc¢*B ,* (*a, b*) ∈ Acc¢{”

But the generated induction principle will be too weak because only usable for an element *b* ∈ Acc¢*B* .

We hence take a stronger goal by proving

∀*a* ∈ Acc

+ *,* ∀*b*1 ∈ Acc¢ *,* ∀*b*2 ∈ *B, b*2(∗ *b*1*,* ⇒ (*a, b*2) ∈ Acc¢{ (3)

¢*A B B*

with (∗ the reflexive transitive closure of (*B*. This result is sufficient to es-

*B*

tablish our theorem because (∗

*B*

is reflexive and because Acc¢*A* = Acc

+ (using

*A*

¢

Lemma [A.4](#_bookmark51)). To prove ([3](#_bookmark16)), we use a noetherian induction on *a* and D+. We

*A*

consider an element *a*1 ∈ Acc¢+ such that

*A*

∀*a*2 ∈ *A, a*2D+*a*1 ⇒

*A*

∀*b*1 ∈ Acc¢*B ,* ∀*b*2 ∈ *B, b*2(∗ *b*1*,* ⇒ (*a*2*, b*2) ∈ Acc¢{

*B*

(4)

and try to prove ∀*b*1 ∈ Acc¢*B ,* ∀*b*2 ∈ *B, b*2(∗ *b*1*,* ⇒ (*a*1*, b*2) ∈ Acc¢. Once time again a direct proof by induction on *b*1 will be unsuccessful without reinforcing first the current goal. We prove instead:

*B*

∀*b*1 ∈ Acc¢+ *,*

*B*

∀*b*2 ∈ *B, b*2(∗ *b*1*,* ⇒

*B*

∀*a*3 ∈ *A, a*3(∗ *a*1 ⇒ (*a*3*, b*2) ∈ Acc¢{

*A*

(5)

This is a sufficient result because Acc¢*B* = Acc

+ and (∗

*B*

¢

*A*

is reflexive. We prove

([5](#_bookmark18)) by noetherian induction on *b*1. Hence we take an element *b*1 ∈ Acc¢+ such that

*B*

∀*b*1 ∈ Acc + *,* ∀*b*3 ∈ *B, b*3D+*b*1 ⇒

¢*B B*

∀*b*2 ∈ *B, b*2(∗ *b*3*,* ⇒

*B*

(6)

∀*a*3 ∈ *A, a*3(∗ *a*1 ⇒ (*a*3*, b*2) ∈ Acc¢{

*A*

and we look for a proof of

∀*b*2 ∈ *B, b*2(∗ *b*1*,* ⇒ ∀*a*3 ∈ *A, a*3(∗ *a*1 ⇒ (*a*3*, b*2) ∈ Acc¢{

*B A*

Let us suppose

*b*1 ∈ Acc¢+

*B*

(7)

*b*2(∗ *b*1 (8)

*B*

*a*3(∗ *a*1 (9)

*A*

and prove (*a*3*, b*2) ∈ Acc¢{. We have to consider a predecessor (*a*4*, b*4)

(*a*4*, b*4)D((*a*3*, b*2) (10)

and we have to prove (*a*4*, b*4) ∈ Acc¢{. Using the definition of D(, two cases results from hypothesis ([10](#_bookmark23)).

* Case 1 :

*a*4D*Aa*3 (11)

*b*4(*Bb*2 (12)

We can use the induction hypothesis ([4](#_bookmark17)). Three conditions must be satisfied

* *a*4D*Aa*1 : true using hypotheses ([1](#_bookmark13)), ([9](#_bookmark22)) and ([11](#_bookmark24)).
* *b*1 ∈ Acc¢*B* : true using ([7](#_bookmark20)) and the general result Acc¢+ = Acc¢*B*

*B*

* *b*4(∗ *b*1 : true because of ([12](#_bookmark25)) and ([8](#_bookmark21))

*B*

* + Case 2 :

*a*4(*Aa*3 (13)

*b*4D*Bb*2 (14)

This time, we can use the induction hypothesis ([6](#_bookmark19)). Three conditions must be satisfied

* *b*4D+*b*1 : true because of the hypotheses ([2](#_bookmark14)), ([8](#_bookmark21)) and ([14](#_bookmark26))

*A*

* *b*4(∗ *b*4 : true by reflexivity of (∗

*B B*

* *a*4(∗ *a*1 : true using hypotheses ([9](#_bookmark22)) and ([13](#_bookmark27))

*A*

Hence the main result is established.

To prove Lemma [4.2](#_bookmark12), we only have to use a *measure function* (see Lemma [A.3](#_bookmark50)) *f* : (*A* × *B*) × (*A* × *B*) → (*A* × *A*) × (*B* × *B*) defined by *f* ((*a*1*, b*1) *,* (*a*2*, b*2)) = ((*a*1*, a*2) *,* (*b*1*, b*2)) and considering the relation ≺o*A*×*B* on (*A* × *B*) × (*A* × *B*) and

D(lex(≺o*A ,*≺o*B ,* o*A ,*Go*B* ) on (*A* × *A*) × (*B* × *B*) where o

*A*

and “o*B*

are defined by

(*x*1*, y*1) o*A* (*x*2*, y*2) ⇐⇒ *x*2 ±*A x*1 ∧ *y*1 ≡ *y*2 and (*x*1*, y*1) “o*B* (*x*2*, y*2) ⇐⇒

*x*2 ±*B x*1 ∧ *y*1 ≡*B y*2o*B x*1. It is not difficult to verify that hypotheses of Theo- rem [4.6](#_bookmark15) are fulfilled and then conclude.

# 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”.

* 1. *Function implementation*

The notion of function implementation is given in Figure [6](#_bookmark30). This signature handles

* + - a module [4](#_bookmark29) A with a signature FiniteSet (associated with the function domain). The FiniteSet signature is given in Figure [7](#_bookmark31). It represents set in bijection with parts 0*,* cardinal − 1) of 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.

4 Modules can handles modules. The corresponding signature element is then introduced by

**Declare Module**.

**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

* 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 imple- mentations (eq\_dec and order\_dec).
* 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

* 1. *A widening operator on functions*

Now for any function implementation we build a poset with a standard widening operator. For functions in *A* → *B* this operator is defined as

∀*f*1*, f*2 ∈ *A* → *B,* ∀*a* ∈ *A,* (*f*1o*f*2)(*a*)= *f*1(*a*)o*Af*2(*a*)

The proof of the termination criterion relies on the Theorem [4.6](#_bookmark15) and the finiteness of the codomain.

* 1. *Two efficient implementations*

We propose two function implementations in our library. The first is a specific im- plementation for functions whose domain is a bounded binary integer (each integer denotes a position in a tree [[14](#_bookmark47)]). This kind of efficient implementation is heavily used in Leroy’s certified compiler [[12](#_bookmark45)]. 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 [[9](#_bookmark42)] [5](#_bookmark33) . Maps can by built on any finite set. Finite sets can be constructed with the previously enumer- ated functors.

To conclude this section, we finish by commenting the example presented in Figure [4](#_bookmark8). 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.

# Conclusion

We have presented a framework for programming fixpoint computations on lat- tice structures in a dependently typed functional language. In order to construct complex lattices, we propose a library of Coq module functors. We focused our ex- planations 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 [[7](#_bookmark40)]. 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 [4.6](#_bookmark15). 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

5 Balanced trees are a keystone of the industrial-task Astre static analyser [[6](#_bookmark39)].

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 *Proc. of the 7th International Conference on Fundamental Approaches to Software Engineering,* *(FASE’04)*, volume 2984 of *LNCS*, pages 99–113. Springer, 2004.
3. Yves Bertot, Benjamin Gr´egoire, and Xavier Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Proc. of the International Workshop on Types for Proofs* *and Programs (TYPES’04)*, volume 3839 of *LNCS*, pages 66–81. Springer, 2004.
4. 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.
5. Solange Coupet-Grimal and William Delobel. A uniform and certified approach for two static analyses. In *Proc. of the International Workshop on Types for Proofs and Programs (TYPES’04)*, volume 3839 of *LNCS*, pages 115–137. Springer, 2004.
6. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min´e, D. Monniaux, and X. Rival. The ASTRE´E Analyser. In *Proc. of the 14th European Symposium on Programming Languages and Systems (ESOP’05)*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.
7. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings* *of the 6th Symposium on Principles of Programming Languages (POPL’79)*, pages 269–282. ACM Press, New York, 1979.
8. B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
9. Jean-Christophe Filliˆatre and Pierre Letouzey. Functors for Proofs and Programs. In *Proc. of the 13th European Symposium on Programming Languages and Systems (ESOP’04)*, number 2986 in LNCS, pages 370–384. Springer-Verlag, 2004.
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 *Proc. of 33th ACM Symposium on Principles of Programming Languages (POPL’06)*, pages 42–54. ACM Press, 2006.
13. Antoine Min´e. 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. Ran Shaham, Elliot K. Kolodner, and Shmuel Sagiv. Automatic removal of array memory leaks in java. In *Proc. of the 9th International Conference on Compiler Construction (CC’00)*, LNCS, pages 50–66. Springer, 2000.

# A General results about accessibility predicats

The following results are proved by Paulson in [[15](#_bookmark48)] and belong to the Coq standard library.

Lemma A.1 *For all relations R*1*, R*2 *on a set A, if R*1 ⊆ *R*2 *then* Acc*R*2 ⊆ Acc*R*1 *.*

Lemma A.2 *For all relation RB on a set B, for all function f* : *A* → *B with A a set, the relation RA deﬁned on A by RA* = {(*a*1*, a*2) | (*f* (*a*1)*,f* (*a*2)) ∈ *RB*} *satisﬁes*

∀*a* ∈ *A, f* (*a*) ∈ Acc*RB* ⇒ *a* ∈ Acc*RA.*

These two lemmas can be summarised in the following result

Lemma A.3 *For all relations RA on a set A, RB on a set B, for all function f* : *A* → *B satisfying* ∀(*a*1*, a*2) ∈ *RA,* (*f* (*a*1)*,f* (*a*2)) ∈ *RB, we have* ∀*a* ∈ *A, f* (*a*) ∈ Acc*RB* ⇒ *a* ∈ Acc*RA.*

*f* is often called a *measure function*.

Lemma A.4 *For all relation R on a set A,* Acc*R* = Acc*R*+ *with R*+ *the transitive closure of R.*