# A Modular Lattice Library to Develop Certified Terminating Static Analyses

David Pichardie

IRISA / ENS Cachan, Rennes, France


# Motivations

Goal : develop certified static analyses

- using the Coq proof assistant
- extracting a certified Ocaml program
- for real analyses on real programming languages

# Motivations

Goal : develop certified static analyses

- using the Coq proof assistant
- extracting a certified Ocaml program
- for real analyses on real programming languages

Bottleneck : workforce cost

- we need a generic framework
- reusable for many analyses

# Design of a static analysis

Two parts

# Design of a static analysis

Two parts

1. specification of an abstract semantics
   - as a least fixed point of a monotone operator $F$, on a poset structure
   - should be proved correct with respect to the concrete semantics

# Design of a static analysis

Two parts

1. specification of an abstract semantics
   - as a least fixed point of a monotone operator $F$, on a poset structure
   - should be proved correct with respect to the concrete semantics
2. solving tool
   - computation of the least fixed point
   - by a generic fixed point iteration

$$\bot \to F(\bot) \to F^2(\bot) \to \cdots \to F^n(\bot) \to \cdots \to \text{lfp}(F)$$

should terminate !

# Design of a static analysis

Two parts

1. specification of an abstract semantics
   - as a least fixed point of a monotone operator $F$, on a poset structure
   - should be proved correct with respect to the concrete semantics
2. solving tool **This Talk**
   - computation of the least fixed point
   - by a generic fixed point iteration

$$\bot \to F(\bot) \to F^2(\bot) \to \cdots \to F^n(\bot) \to \cdots \to \mathrm{lfp}(F)$$

should terminate !

# The Lattice contract

```
Module Type Lattice.

End Lattice.
```

# The Lattice contract

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









End Lattice.
```

# The Lattice contract

```
Module Type Lattice.
   Parameter t : Set.
   Parameter eq : t → t → Prop.
   Parameter eq_prop : ...
              (* eq is a computable equivalence relation *)
```

```
End Lattice.
```

# The Lattice contract

```
Module Type Lattice.
   Parameter t : Set.
   Parameter eq : t → t → Prop.
   Parameter eq_prop : ...
              (* eq is a computable equivalence relation *)
   Parameter order : t → t → Prop.
   Parameter order_prop : ...
              (* order is a computable order relation *)
```

```
End Lattice.
```

# The Lattice contract

```
Module Type Lattice.
   Parameter t : Set.
   Parameter eq : t → t → Prop.
   Parameter eq_prop : ...
              (* eq is a computable equivalence relation *)
   Parameter order : t → t → Prop.
   Parameter order_prop : ...
              (* order is a computable order relation *)
   Parameter join : t → t → t.
   Parameter join_prop : ...
              (* join is a binary least upper bound *)
              (* (useful to specify the analysis) *)

End Lattice.
```

# The Lattice contract

```
Module Type Lattice.
   Parameter t : Set.
   Parameter eq : t → t → Prop.
   Parameter eq_prop : ...
              (* eq is a computable equivalence relation *)
   Parameter order : t → t → Prop.
   Parameter order_prop : ...
              (* order is a computable order relation *)
   Parameter join : t → t → t.
   Parameter join_prop : ...
              (* join is a binary least upper bound *)
              (* (useful to specify the analysis) *)
   Parameter bottom : t.
              (* bottom element to start iteration *)
   Parameter bottom_is_bottom : ∀ x : t, order bottom x.


End Lattice.
```

# The Lattice contract

```
Module Type Lattice.
   Parameter t : Set.
   Parameter eq : t → t → Prop.
   Parameter eq_prop : ...
              (* eq is a computable equivalence relation *)
   Parameter order : t → t → Prop.
   Parameter order_prop : ...
              (* order is a computable order relation *)
   Parameter join : t → t → t.
   Parameter join_prop : ...
              (* join is a binary least upper bound *)
              (* (useful to specify the analysis) *)
   Parameter bottom : t.
              (* bottom element to start iteration *)
   Parameter bottom_is_bottom : ∀ x : t, order bottom x.
   Parameter termination_property : ...
              (* to ensure iteration termination *)

End Lattice.
```

# Two termination criteria

# Two termination criteria

**Ascending Chain condition** (`LatticeWf` contract)

- classical definition : “there is no infinite strictly increasing chain”

$$x_0 \sqsubset x_1 \sqsubset x_2 \sqsubset \ldots \sqsubset x_n \sqsubset \ldots$$

- constructive definition : $\sqsupset$ is well-founded

$$\forall x : t,\ \mathrm{Acc}\ \sqsupset\ x$$

# Two termination criteria

**Ascending Chain condition** (`LatticeWf` contract)

- classical definition : "there is no infinite strictly increasing chain"

$$x_0 \sqsubset x_1 \sqsubset x_2 \sqsubset \ldots \sqsubset x_n \sqsubset \ldots$$

- constructive definition : $\sqsupset$ is well-founded

$$\forall x : t,\ \mathrm{Acc}\ \sqsupset\ x$$

**Widening operator** (`LatticeWidening` contract)

- modification of the solving algorithm

$$x_0 = \bot, \ldots, x_{n+1} = x_n \triangledown F(x_n), \ldots$$

- classical definition : for all increasing chains $x_0 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq \ldots$, the chain $y_0 = x_0, \ldots, y_{n+1} = y_n \triangledown x_n, \ldots$ is not strictly increasing
- constructive definition : $\forall x : t,\ \mathrm{Acc}\ \prec_\triangledown\ (x, x)$ with

$$(x_1, y_1) \prec_\triangledown (x_2, y_2) \text{ iff } x_2 \sqsubseteq x_1\ \wedge\ y_1 = y_2 \triangledown x_1\ \wedge\ y_1 \neq y_2$$

# WANTED

$$L = R \times S \times L \times H$$

with

$$\begin{aligned} R &= \text{MethodName} \to \text{Context} \to \text{Val}^\sharp \\ S &= (\text{MethodName} \times \text{ProgPoint}) \to \text{Context} \to \text{List}\left(\text{Val}^\sharp\right) \\ L &= (\text{MethodName} \times \text{ProgPoint}) \to \text{Context} \to \left(\text{Var} \to \text{Val}^\sharp\right) \\ H &= (\text{ClassName} \times \text{FieldName}) \to \text{Val}^\sharp \end{aligned}$$

$\text{Val}^\sharp = \wp(\text{ClassName}) + \text{Num}^\sharp \quad \text{Context} = (\text{MethodName} \times \text{ProgPoint})^{*\leq k}$

and $\text{Num}^\sharp$ which

- either respects `LatticeWf` (constant lattice)
  - we must prove that $L$ respects `LatticeWf` !
- or respects `LatticeWiden` (interval lattice)
  - we must prove that $L$ respects `LatticeWiden` !

# Proposed solution : module functors

**Module functors** : functions that take modules as arguments and produce modules as results

**Various functors**

- product, sum, list
  Example of property proved:
  $$\left.\begin{array}{l}\forall a : A,\ \mathrm{Acc}\ \prec_{\nabla_A} (a,a) \\ \forall b : B,\ \mathrm{Acc}\ \prec_{\nabla_B} (b,b)\end{array}\right) \Longrightarrow \forall c : A \times B,\ \mathrm{Acc}\ \prec_{\nabla_{A\times B}} (c,c)$$
- function on finite sets
  - implemented with functional map or dictionaries on binary keys
  - finite sets are constructed by functor compositions too

**See the poster for more details !**

# Conclusions

- For many analyses, with an appropriate combination of the proposed functors, no termination proofs are needed
- A nice illustration of module functors, more involved (wrt. modularity) than classical examples (set, dictionaries)
- A nice example mixing mathematical structures and efficient extracted data structures

Interested ?

Let’s discuss together !

Offering a postdoc position for October 2005 ?

Let’s discuss together !