Chapter 7

A reflection-based proof tactic for lattices in Coq

Daniel W. H. James and Ralf Hinze[1]
Category: Research

7.1 INTRODUCTION

Coq is a proof assistant featuring a tactic-based interactive theorem prover. The latest incarnation comes with over 150 tactics that assist the user in developing a formal proof. These tactics range from the simple and mundane to the 'all-powerful'. Some examples from the latter category are the `omega` tactic, which solves a goal in Presburger arithmetic, and the `ring` and `field` tactics, which solve identities modulo associativity and commutativity in ring and field structures.

This paper presents a new proof tactic that decides equalities and inequalities between terms over lattices. It uses a decision procedure that is a variation on Whitman's algorithm [16] and is implemented using a technique known as *proof by reflection*. We will paint the essence of the approach in broad strokes and discuss the use of certified functional programs to aid the automation of formal reasoning.

This paper makes three contributions. Firstly, it serves as an introduction to using the proof by reflection approach in Coq. This utilizes the Ltac language and two recent extensions to the Coq system: type classes and the PROGRAM extension. Secondly, it gives a certified implementation of Whitman's algorithm in Coq, with proofs of correctness and termination. Thirdly, the final product of this work is a usable proof tactic that can be applied to proof goals involving equalities and inequalities in lattice theory.

The rest of the paper is organized as follows: Section 2 will give a short introduction to Coq, highlighting the concept of a *proof term*, and will refresh the

[1]Computing Laboratory, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK; `{daniel.james, ralf.hinze}@comlab.ox.ac.uk`

necessary details of lattice theory. We will give a motivation for the problem in Section 3 and a Coq-based introduction to free lattices, along with the decision procedure, in Section 4. Section 5 will describe the technique of proof by reflection and Section 6 will discuss related and future work. A familiarity with typed functional programming is recommended.

7.2 BACKGROUND

7.2.1 Coq

Coq is based on the Calculus of Inductive Constructions (CIC), a type theory which in turn is an extension of the Calculus of Constructions (CC) with (co-)inductive types. Barendregt's λ-cube details the richness of this polymorphic, dependent type system [1]. As a type-theory-based proof assistant it follows the propositions-as-types, proofs-as-programs interpretation attributed to Curry and Howard, which brings the user into a realm in which the lines between *programming* and *proving* become blurred. The underlying typed λ-calculus allows the definition of functions as in any other functional programming language, yet when viewed as a constructive higher-order logic, the same language can be used to write specifications and construct proofs.

Constructing a proof for a given formula is, of course, undecidable in general. One of the faces of Coq is a tactic-based, interactive theorem prover. A theorem is proven in Coq through the interactive application of tactics on a proof goal. The product of this interactive session is a proof term that is alleged to prove the given formula. Due to the Curry-Howard isomorphism, this is the same as saying that the term is alleged to inhabit the type specified. Thus when in the role of 'proof-checker', Coq is simply type-checking the proof term, a procedure that *is* decidable.

The idea of producing a proof term is an important one. It enables Coq to satisfy what is known as the *de Bruijn criterion* [2], where the trust in the reliability of the system must extend only to a small *kernel* and not to the system as a whole. For us to trust a proof it is not necessary to make sure that every tactic used to construct it is bug free, we need only trust Coq's kernel type-checker. One could even see a proof term as a certificate that can be externally verified.

7.2.2 Lattices

A lattice is a set A equipped with two binary operators, meet $\sqcap$ and join $\sqcup$, where the following identities hold for all elements a, b, c of A,

$$\begin{array}{llll} a \sqcup b = b \sqcup a & a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c & a \sqcup (a \sqcap b) = a & a \sqcup a = a \\ a \sqcap b = b \sqcap a & a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c & a \sqcap (a \sqcup b) = a & a \sqcap a = a \end{array}$$

namely the commutative, associative, absorptive and idempotent laws. The second row is simply the dual of the first row. This presentation defines a lattice as an algebraic structure, $(A, \sqcap, \sqcup)$.

There is more than one design route we can take to encode this mathematical definition into the Coq system. We have chosen to use a new feature of Coq, called 'type classes' [15]. Type classes in Coq are a lot like type classes in Haskell. The *Monad* type class is well known in Haskell. It consists of the monadic operators *return* and *bind*, but the monad laws are provided only as part of the documentation, in the hope that programmers abide by them. Coq allows us to do better than that. For lattices, we wish to declare the operators *meet* and *join*, but we can also specify the laws that they must satisfy.[2]

```
Class Lattice (A : Set) := {
  meet : A → A → A;
  join   : A → A → A;

  meet_commutative : ∀ a b, meet a b = meet b a;
  meet_associative   : ∀ a b c, meet (meet a b) c = meet a (meet b c);
  meet_absorptive    : ∀ a b, meet a (join a b) = a;
  meet_idempotent    : ∀ a, meet a a = a;

  join_commutative   : ∀ a b, join a b = join b a;
  join_associative     : ∀ a b c, join (join a b) c = join a (join b c);
  join_absorptive      : ∀ a b, join a (meet a b) = a
  join_idempotent      : ∀ a, join a a = a;
}.
Infix "⊓" := meet.
Infix "⊔" := join.
```

Our type class Lattice is parameterized by a set A, and we have declared some infix notation for *meet* and *join*. The scope of these notations extends over the remainder of this paper.

We can impose an ordering on elements of the lattice with the following definition:

$$a \leq b \iff a = a \sqcap b \iff b = a \sqcup b.$$

We could also give an order-theoretic presentation of lattices by starting with a partially-ordered set, such that for any two elements, the *meet* and the *join* are the infimum and supremum, respectively. Let's continue by first giving a definition for what it means to be a partial order: a binary relation that is reflexive, antisymmetric and transitive.

```
Class Order (A : Set) := {
  le : A → A → Prop;
  reflexive : ∀ a, le a a;
  antisymmetric : ∀ a b, le a b ∧ le b a → a = b;
  transitive : ∀ a b c, le a b ∧ le b c → le a c
}.
Infix "≤" := le.
```

[2]Using Leibniz equality is sufficient for expository purposes, but Setoid equality would be more general.

Again, we have given some convenient notation for the ordering relation. These two type classes can be combined into a single consistent type class that represents the concept of a lattice-ordered set: something that is both a lattice and a partial order, and the two are consistent according to the definition given above.

```
Class LOSet (A : Set) := {
  order :> Order A;
  lattice :> Lattice A;
  consistent : ∀ a b, a ≤ b ↔ a = a ⊓ b
}.
```

The notation :> is Coq's version of subclassing for type classes: a LOSet is also an Order and a Lattice. For many of the definitions in the rest of this paper, we will need to use meet, join or le, so we will need to parameterize each definition by a lattice-ordered set. We can do this implicitly by using the `Context` command. This introduces both a set A and an instance l of the type class into scope. Again, we will assume that this scope extends over the remainder of this paper.

```
Context `{l : LOSet A}.
```

For any elements a,b,x of the lattice A, we can give the following universal property for *meet*,

```
Theorem meet_is_glb : ∀ a b : A,
  ∀ x, x ≤ a ∧ x ≤ b ↔ x ≤ a ⊓ b.
```

Read from right to left, this says that every x below a *meet* is a lower bound and, conversely, the *meet* is the greatest of every lower bound. By trivially instantiating x to be $a \sqcap b$, we derive the inequalities $a \sqcap b \leq a$ and $a \sqcap b \leq b$. The duals of these properties hold for *join*,

```
Theorem join_is_lub : ∀ a b : A,
  ∀ x, a ≤ x ∧ b ≤ x ↔ a ⊔ b ≤ x.
```

and $a \leq a \sqcup b$ and $b \leq a \sqcup b$. The theorems meet_is_glb and join_is_lub, whose proofs proceed by straightforward induction, are two out of a collection of theorems and lemmas that have been developed in the process of implementing our lattice proof tactic. Their purpose is ultimately for the major correctness proofs; however, this is not the limit of their usefulness.

7.3 SOLVING LATTICE (IN)EQUALITIES

Our aim is an automatic procedure, but we will begin with an example to discuss how proofs proceed on paper as well as in Coq. We use the median property of a lattice as our starting example,

$$(a \sqcap b) \sqcup (b \sqcap c) \sqcup (c \sqcap a) \leq (a \sqcup b) \sqcap (b \sqcup c) \sqcap (c \sqcup a).$$

In words, the *join* of the *meets* is at most the *meet* of the *joins*. We can give an elegant equational proof of this property by repeatedly using the universal properties of *meet* and *join* (meet_is_glb and join_is_lub):

$$
\begin{array}{ll}
 & (a \sqcap b) \sqcup (b \sqcap c) \sqcup (c \sqcap a) \leq (a \sqcup b) \sqcap (b \sqcup c) \sqcap (c \sqcup a) \\
\Leftrightarrow & \quad \{ \text{ universal property of } \sqcup \ \} \\
 & (a \sqcap b \leq rhs) \wedge (b \sqcap c \leq rhs) \wedge (c \sqcap a \leq rhs) \\
\Leftrightarrow & \quad \{ \text{ universal property of } \sqcap \ \} \\
 & (a \sqcap b \leq a \sqcup b) \wedge (a \sqcap b \leq b \sqcup c) \wedge (a \sqcap b \leq c \sqcup a) \wedge \ldots
\end{array}
$$

This is a conjunction of nine inequalities, where each *meet* on the left hand side is compared to each *join* on the right-hand side, each of which can be proved by transitivity and the inequalities derived above. For example, the second conjunct follows from $a \sqcap b \leq b \leq b \sqcup c$. The other conjuncts follow similarly.

This is what we might call a *human consumable* proof. It is certainly not in a form that can be immediately verified with a system such as Coq. The first two steps of the proof above use the universal properties of *meet* and *join*. To repeat these two steps in Coq we must use the equivalences in the theorems meet_is_glb and join_is_lub as right-to-left rewrite rules. The following is a Coq formalization of the median inequality:

```
Lemma median_inequality : ∀ x y z : A,
  (x ⊓ y) ⊔ (y ⊓ z) ⊔ (z ⊓ x) ≤ (x ⊔ y) ⊓ (y ⊔ z) ⊓ (z ⊔ x).
Proof.
  intros x y z.
  set (lhs := (x ⊓ y) ⊔ (y ⊓ z) ⊔ (z ⊓ x)).
  repeat (rewrite ← (meet_is_glb lhs)).
  unfold lhs.
  repeat (rewrite ← (join_is_lub (x ⊔ y))).
  repeat (rewrite ← (join_is_lub (y ⊔ z))).
  repeat (rewrite ← (join_is_lub (z ⊔ x))).
  intuition
    (apply (transitive (b := x)); split; auto with order; fail) ||
    (apply (transitive (b := y)); split; auto with order; fail) ||
    (apply (transitive (b := z)); split; auto with order; fail).
Qed.
```

This formal proof follows exactly the same structure as the paper-and-pencil proof; however, without the first as a guide, it would be very difficult to get an intuition without interactively stepping through the proof. This is, unfortunately, a general comment one can make about proof developments in Coq. This proof is short and has achieved its length by using several of Coq's automation features: the `repeat` tactical and the tactics `intuition` and `auto`. The `auto` tactic is invoked with hints from the order namespace, which has been populated with lemmas from our library. We have not presented this proof in the hope that the reader will comprehend it, but rather to give a sense of the sort of proofs we will be automating. Before moving on, there is one final comment to be made about proofs involving rewriting. In this proof, a total of eight rewrites are needed, and the story is similar for other lattice equalities and inequalities. More recent versions of Coq have increased the control over the application of the `rewrite` tactic; however, `rewrite` is fundamentally fragile to change. Even something as simple as changing the order or parentheses is likely to cause an application of the `rewrite` tactic to break. There are also technical reasons why a substantial use of rewriting is undesirable in Coq (see Section 6), but for now our simple motivation is to provide as much automation to users as possible. Therefore, our goal is to produce a tactic that can turn the proof of the median equality into a one-liner.

```
Lemma median_inequality : ∀ x y z : A,
  (x ⊓ y) ⊔ (y ⊓ z) ⊔ (z ⊓ x) ≤ (x ⊔ y) ⊓ (y ⊔ z) ⊓ (z ⊔ x).
Proof.
  solve_lattice_inequality.
Qed.
```

For the purposes of a running example, we will use the following smaller inequality, which requires all but idempotence to prove:

$$\begin{array}{ll} & a \sqcap b \leq a \sqcup b \\ \Leftrightarrow & \quad \{ \text{ induced ordering } \} \\ & a \sqcap b = (a \sqcap b) \sqcap (a \sqcup b) \\ \Leftrightarrow & \quad \{ \text{ associativity and commutativity } \} \\ & a \sqcap b = a \sqcap (b \sqcap (b \sqcup a)) \\ \Leftrightarrow & \quad \{ \text{ absorption } \} \\ & a \sqcap b = a \sqcap b \\ \Leftrightarrow & \quad \{ \text{ reflexivity } \} \\ & True \end{array}$$

The corresponding Coq proof is as follows:

```
Lemma running_example : ∀ a b : A,
  a ⊓ b ≤ a ⊔ b.
Proof.
  intros a b.
  rewrite consistent.
  rewrite meet_associative.
  rewrite join_commutative.
  rewrite meet_absorptive.
  reflexivity.
Qed.
```

7.4 A DECISION PROCEDURE FOR LATTICES

We will now present a step-by-step development of a decision procedure for (in-)equalities on lattices, introducing the concept of a *free lattice* along the way.

We eschew a mathematical presentation in favour of a concrete one in Coq.

We will begin by defining *lattice terms*. Variables are terms, and given two terms, t_1 and t_2, we can construct terms representing the *meet* and the *join*. The set of terms can be defined by the following inductive type in Coq:

```
Inductive Term : Set :=
  | Var : nat → Term
  | Meet : Term → Term → Term
  | Join : Term → Term → Term.
```

The type **Term** is of sort Set, and it consists of three data constructors. The constructors Meet and Join both take two terms and construct a term. With no loss of generality, variables are simply identified by natural numbers.

As an illustration of simple recursive functions on terms, each term has a length (or *rank*) and a depth (or *complexity*), which are defined by the following functions:

```
Fixpoint length (t : Term) : nat :=
  match t with
  | Var n ⇒ 1
  | Meet t1 t2 ⇒
    1 + length t1 + length t2
  | Join t1 t2 ⇒
    1 + length t1 + length t2
  end.
```

```
Fixpoint depth (t : Term) : nat :=
  match t with
  | Var n ⇒ 0
  | Meet t1 t2 ⇒
    1 + max (depth t1) (depth t2)
  | Join t1 t2 ⇒
    1 + max (depth t1) (depth t2)
  end.
```

There are a number of important points to note here. There is no general recursion in Coq, so recursive functions defined with the `Fixpoint` command must use structural recursion. For length and depth, each recursive call must operate on a subterm of the input term. Furthermore, functions must be total and terminating, and pattern-matching must be complete.

Using the expressive power of dependent types, we can encode a subterm relation on terms as an inductive datatype. The inductive type **Subterm** is *indexed* by two terms and is a proposition (an element of sort `Prop`).

```
Inductive Subterm : Term → Term → Prop :=
  | Meet1 : ∀ t1 t2, Subterm t1 (Meet t1 t2)
  | Meet2 : ∀ t1 t2, Subterm t2 (Meet t1 t2)
  | Join1 : ∀ t1 t2, Subterm t1 (Join t1 t2)
  | Join2 : ∀ t1 t2, Subterm t2 (Join t1 t2).
```

In words, the first data constructor, Meet_1, says for all terms t_1 and t_2, t_1 is a subterm of (Meet t_1 t_2). Note that this relation is irreflexive (a strict partial-order) and induces a well-founded relation. We will make use of this relation later to prove the termination of functions on terms that do not follow a simple structural recursion scheme.

Let t be a term and A be a lattice-ordered set, then we can evaluate t in the lattice by providing an environment (env : **nat** $\rightarrow A$) that maps the free variables to elements of the lattice.

```
Fixpoint eval (t : Term) (env : nat → A) {struct t} : A :=
  match t with
  | Var n ⇒ env n
  | Meet t1 t2 ⇒ ⟦t1⟧env ⊓ ⟦t2⟧env
  | Join t1 t2 ⇒ ⟦t1⟧env ⊔ ⟦t2⟧env
  end where "⟦ t ⟧env" := (eval t env).
```

The eval function is within the scope of the earlier `Context` command, so is parameterized by an instance of the *LOSet* type class. In the function body, the binary operators $\sqcap$ and $\sqcup$ are the *meet* and *join* of the parameterized, lattice-ordered set. The annotation `{struct` t`}` indicates to Coq that eval will structurally recurse over the term argument. The function eval is a fold over terms, with Meet mapped to *meet*, Join mapped to *join* and variables mapped to lattice elements using the provided environment.

A lattice has an ordering via the induced ordering given in Section 2.2; let's specify what it means for *lattice terms* (of type **Term**) to be ordered. Using eval, we can define a *semantic* preorder (quasiorder) on terms. Let s and t be terms, then s is semantically less than or equal to t if, and only if, for all lattices and for all environments the evaluation of s is less than or equal to the evaluation of t. An equivalence relation can be defined similarly:

```
Definition Leq (s t : Term) : Prop := ∀ env, ⟦s⟧env ≤ ⟦t⟧env.
Definition Equiv (s t : Term) : Prop := ∀ env, ⟦s⟧env = ⟦t⟧env.
```

Note that there is only one universal quantifier in each of these definitions, yet above we said that we are quantifying over all lattices and all environments. This is because of the implicit quantification introduced by the `Context` command. When restricted to variables, both the preorder and the equivalence relation are exactly the equality relation. The relation clearly cannot be more permissive, for exactly the reason that we are quantifying over all lattices and all environments. The relation Equiv is also a congruence relation over *meet* and *join*.

The Equiv relation allows us to express the mathematical concept of a *free lattice*. Let T be the set of lattice terms built from the set of variable indices N (where $N \subseteq \mathbb{N}$), then the free lattice is the quotient set of T by equivalence relation Equiv. Informally, the free lattice is an object in which all the lattice laws hold, but nothing more. If we can prove a proposition about a free lattice, then it holds for all lattices: a powerful statement indeed.

The relations Leq and Equiv very succinctly specify ordering and equivalence on lattice terms; however, they go nowhere in informing us how actually to compute the ordering or equivalence of two concrete terms. A free lattice is really only a useful structure, to us at least, if we have a decision procedure for the ordering relation Leq (Equiv can be computed from Leq). The *word problem* for free lattices is the problem of finding such a procedure.

Let's begin by exploring simple cases of this problem. As noted before, variables of lattice terms are only comparable by equality. If we have two terms s and

t, and s is ($\mathsf{Var}\ i$) and t is ($\mathsf{Var}\ j$), then ($\mathsf{Leq}\ s\ t$) holds iff $i = j$. Now suppose that s is ($\mathsf{Join}\ a\ b$) and t is any term, then by the universal property of *join* (join_is_lub), ($\mathsf{Leq}\ s\ t$) holds iff ($\mathsf{Leq}\ a\ t$) and ($\mathsf{Leq}\ b\ t$) holds. Similarly, suppose that s is any term and t is ($\mathsf{Meet}\ a\ b$); then, by the universal property of meet (meet_is_glb), ($\mathsf{Leq}\ s\ t$) holds iff ($\mathsf{Leq}\ s\ a$) and ($\mathsf{Leq}\ s\ b$) hold. The tricky case is when s is (Meet $a\ b$) and t is ($\mathsf{Join}\ c\ d$), where a *meet* is below a *join*. Whitman has given a very influential solution to this problem [16].

Whitman's solution

In the case of a *meet* below a *join*, Whitman gave the following solution, and this is known as 'Whitman's condition':

> If $s = \mathsf{Meet}\ s_1\ s_2$ and $t = \mathsf{Join}\ t_1\ t_2$ and $\mathsf{Leq}\ s\ t$, then either $\mathsf{Leq}\ s_i\ t$ for some i, or $\mathsf{Leq}\ s\ t_j$ for some j.

We refer the reader to Freese et al. [7] for the proof and a full presentation on free lattices.

The complete solution is, of course, a computable preorder $\lesssim$ on lattice terms. Using Whitman's condition, along with equality on variables and the universal properties of *meet* and *join*, the following enumerates all six cases for the computable preorder $\lesssim$:

1. If $s = \mathit{Var}\ i$ and $t = \mathit{Var}\ j$, then $s \lesssim t$ holds iff $i = j$.
2. If $s = \mathit{Join}\ s_1\ s_2$, then $s \lesssim t$ holds iff $s_1 \lesssim t$ and $s_2 \lesssim t$.
3. If $t = \mathit{Meet}\ t_1\ t_2$, then $s \lesssim t$ holds iff $s \lesssim t_1$ and $s \lesssim t_2$.
4. If $s = \mathit{Var}\ i$ and $t = \mathit{Join}\ t_1\ t_2$, then $s \lesssim t$ holds iff $s \lesssim t_1$ or $s \lesssim t_2$.
5. If $s = \mathit{Meet}\ s_1\ s_2$ and $t = \mathit{Var}\ i$, then $s \lesssim t$ holds iff $s_1 \lesssim t$ or $s_2 \lesssim t$.
6. If $s = \mathit{Meet}\ s_1\ s_2$ and $t = \mathit{Join}\ t_1\ t_2$, then $s \lesssim t$ holds iff $s_1 \lesssim t$ or $s_2 \lesssim t$ or $s \lesssim t_1$ or $s \lesssim t_2$.

Cases (4) and (5) have been introduced as special cases of Whitman's condition, when a variable is below a *join* and above a *meet*. Whitman's solution gives a preorder $\lesssim$ on lattice terms, with which we can construct the equivalence relation in the standard way: $s \sim t \iff s \lesssim t \wedge t \lesssim s$.

Now we have a decision procedure in hand, we can revisit our running example: $a \sqcap b \leq a \sqcup b$. Before we proceed, we must *reflect* the inequality from a problem in a lattice, to a problem on lattice terms: $\mathit{Meet}\ a\ b \lesssim \mathit{Join}\ a\ b$. Originally, a and b were elements of a lattice, but in an abuse of notation they are now variables for lattice terms. We let $a = \mathsf{Var}\ 0$ and $b = \mathsf{Var}\ 1$. To begin with, the sixth case applies (Whitman's condition) and we recurse with $a \lesssim \mathit{Join}\ a\ b$. Then the forth case applies and we recurse with $a \lesssim a$, and finally the first case applies and we succeed by equality.

Figure 7.1 lists an implementation in Coq that very closely matches the mathematical presentation of Whitman's solution. There are a number of points of interest with respect to this implementation. We will start by focusing on the type

```
Program Fixpoint leq (p : Term × Term) {wf R}
  : { b : bool | Is_true b → Leq (fst p) (snd p) } :=
  match p with
  | (s, t) ⇒
    match (s, t) with
    | (Var i, Var j) ⇒ nat_eq_bool i j
    | (Join s1 s2, t) ⇒ s1 ≲ t ∧ s2 ≲ t
    | (s, Meet t1 t2) ⇒ s ≲ t1 ∧ s ≲ t2
    | (Var m, Join t1 t2) ⇒ s ≲ t1 ∨ s ≲ t2
    | (Meet s1 s2, Var n) ⇒ s1 ≲ t ∨ s2 ≲ t
    | (Meet s1 t2, Join t1 t2) ⇒ s1 ≲ t ∨ s2 ≲ t ∨ s ≲ t1 ∨ s ≲ t2
    end
  end where "s ≲ t" := (leq (s, t)).
Next Obligation.
  ⋮
```

FIGURE 7.1. leq: A variant of Whitman's algorithm in Coq

of leq. The function takes a pair of terms as input (**Term** × **Term**) and returns a Boolean value, but not *just* a Boolean value. The return type of leq is a subset type, which has the general form $\{ x : A \mid P\, x \}$. In words, this is the subset of elements of type A that satisfy the predicate P. The predicate used in the return type of leq states that the truth of the Boolean value implies the truth of the semantic preorder on the input terms. This is exactly the correctness property of leq, that it is sound with respect to Leq. Subset types are another example of dependent types: they are a dependent pair consisting of a value and a property that depends on the value, so an inhabitant of a subset type is a value and a proof object that depends on that value.

The expressive power of dependent types has allowed us to give a strong specification of a function in the function's type itself. However, there is a hitch. We have added propositions into our types, so as a result we will have proofs in our programs. We do not want proofs polluting our computations, and this brings us on to the next point of interest in the implementation of leq. At the start of Figure 7.1, we use the `Program` command to invoke the PROGRAM extension [13, 14]. Without this extension, we would not be able to write the body of the function using Coq's functional language, which we used earlier to define length, depth and eval, because we would have to manipulate proof terms. Instead, Coq would have forced us to drop into the theorem prover and develop the body of the function interactively using proof tactics. Writing complex programs procedurally using tactics, even with automation, is infeasible.

The PROGRAM extension allows us to separate proofs from programs. It uses the RUSSELL programming language, which is more permissive than Coq: only the algorithmic content is required as the body of a strongly specified function, and the user can use non-structural recursion. The RUSSELL type-checker is responsible for type-checking this term, and subsequently the term is elaborated into CIC, if indeed the term can been seen as a valid Coq term. It is only at this point that the proof content of the function is considered and the necessary *proof obligations* for termination and correctness are generated. At each point in the program structure where a proof would normally be required, the RUSSELL type-checker accepts the absence of the proof and defers its necessity. Figure 7.1 shows only the algorithmic content of leq. The proofs of proof obligations are written below, but have been omitted, so proofs are quite literally below the program rather than in the program. Our definition of leq generates over thirty proof obligations. This sounds like a lot, but many of the proofs follow a common pattern, and with this insight we have minimized the length of the proof development by using custom tactics and automation.

One of the restrictions that the PROGRAM extension loosens is the requirement for structural recursion. Instead, the extension allows one to use a measure function or a well-founded relation to express termination. This is one of the features of the PROGRAM extension that we make use of, as Whitman's solution, or more precisely Whitman's condition, is not structurally recursive. It has four recursive calls, two of which reduce the first term, with the other two reducing the second term. While termination seems intuitively clear, Coq would not accept this definition if we were to use the `struct` annotation as we did in the definition of eval. Instead, leq uses the annotation {`wf` R} in its function signature, which indicates that we are using the well-founded relation R to express termination. The relation R is the relational symmetric product of the **Subterm** relation, so it expresses a well-founded ordering on pairs of terms.

`Definition` R := symprod **Term Term Subterm Subterm**.

Many of the proof obligations that the PROGRAM extension will generate will ask for proofs that recursive calls of leq respect the relation R, as well as a proof that R is a well-founded relation and thus that **Subterm** is a well-founded relation. The latter can be proved by induction on **Subterm**.

There is scope for a number of optimizations in the implementation of leq. Firstly, in Figure 7.1 the $\wedge$ and $\vee$ operators represent the *lazy* Boolean operators *and* and *or*, respectively. The *short circuit* behaviour provides significant speedups by eliminating needless computation. A second optimization is to specialize the behaviour of the fourth and fifth cases. If the variable on one side of the inequality is not present on the other, it is not possible for a recursive call to return true. Two auxiliary specializations of leq are needed, one for a variable on the left-hand side of the inequality and one for the right-hand side. For brevity, we have omitted the presentation of this.

7.5 PROOF BY REFLECTION

Suppose that we have a proof goal of the form $a \leq b$, where a and b are arbitrary expressions in a lattice-ordered set (they have type A, where A is an instance of the

LOSet type class). To decide this theorem, we must reflect it into our inductive data-type for lattice terms. Using reflection, we generate two lattice terms, s and t, and an environment *env*, such that

$$[\![s]\!]_{env} \equiv_\beta a \quad \text{and} \quad [\![t]\!]_{env} \equiv_\beta b. \tag{7.1}$$

This is to say, that when evaluated in the extracted environment, terms s and t are β-convertible with a and b.

This crucial initial step of reflection is implemented in Coq's tactic language Ltac [6]. This is an untyped, domain-specific metalanguage, the crucial feature of which is pattern-matching on arbitrary Coq terms. The first Ltac function that we need is one to compute the environment from expressions a and b. The environment comprises maximal subexpressions that are not *meets* or *joins*, and these will form the variables. For brevity, we omit this function and instead present the following tactic that performs the actual reflection:

```
Ltac reflect env m j t :=
  match t with
  | (m ?X1 ?X2) ⇒ let r1 := reflect env m j X1
                  with r2 := reflect env m j X2 in
                   constr:(Meet r1 r2)
  | (j ?X1 ?X2) ⇒ let r1 := reflect env m j X1
                  with r2 := reflect env m j X2 in
                   constr:(Join r1 r2)
  | ?X1 ⇒ let n := inv_lookup env X1 in constr:(Var n)
  end.
```

The *reflect* tactic takes as arguments the environment *env*, the operators *meet* and *join* (m and j) of the lattice-ordered set and an expression t in that lattice. The tactic recursively builds up a term from the structure of the lattice expression, mapping the parameterized *meet* and *join* operators to the data constructors of **Term**. When a subexpression is not a meet or a join, it performs an inverse lookup on the environment to retrieve an index for the Var data constructor.

The next step is to change the proof goal with the `change` tactic, which lets us replace the current goal with another term, provided the replacement is well-formed and convertible with the goal. To proceed, we apply the `change` tactic with: $[\![t]\!]_{env} \leq [\![u]\!]_{env}$, which is convertible by equations (7.1).

We are now at the second crucial point in the proof by reflection technique, where the correctness of *leq* with respect to *Leq* comes into play. The proof of correctness is wrapped up in the definition of *leq*; however, we can easily extract it by proving the following lemma:

`Lemma` leq_correct : $\forall\, t\, u$: **Term**, Is_true ‘$(t \lesssim u) \to \forall\, env,\ [\![t]\!]_{env} \leq [\![u]\!]_{env}$.

The function Is_true interprets Booleans as propositions and the notation back-quote, (‘), extracts the Boolean value from the subset value returned by leq. Applying this correctness lemma to our goal gives us a new goal of Is_true ‘$(t \lesssim u)$. To complete the proof we need do nothing more than reduce it to a value. The

$$\begin{tikzcd}
 & \text{Term} \arrow[dl, bend right, "\text{compute}"'] \arrow[dr, bend left, "\text{eval}"'] & \\
\text{bool} \arrow[rr, "\text{correct}"'] & & \text{LOSet} \arrow[ul, bend right, "\text{reflect}"']
\end{tikzcd}$$

FIGURE 7.2. Overview of proof by reflection

speediest way to do this in Coq is with the `vm_compute` tactic, which evaluates the goal using an optimized, call-by-value evaluation, bytecode-based virtual machine [8]. The end result is either the proposition True or the proposition False.

This entire process can be wrapped up in a single Ltac tactic. We define an identical tactic for lattice equalities by using a second correctness lemma for the Equiv equivalence on lattice terms. Figure 7.2 gives a diagrammatic summary of the proof by reflection technique and the relationships between the constituent types, functions and lemmas.

7.6 DISCUSSION

Richard Weyhrauch, the author of the FOL proof-checker, summarized the essence of proof by reflection: ‘to change theorem proving in the theory into evaluation in the meta-theory’ [5]. Harrison, in his survey on and critique of the use of reflection, cuts to the heart of the motivation when he says that ‘computational reflection principles do not extend the power of the logic, but may make decisions in it more efficient’ [10]. The use of reflection for solving lattice (in)equalities has bought us a lot. Instead of manual rewrites, we have an entirely automatic tactic. Even if there were a rewrite-based decision procedure, the reflection-based procedure would have one major advantage. Each time rewriting is used in Coq, the type-checker has to verify the sequence of rewrite steps, whereas the correctness of leq has been be proved and type-checked once and for all. We have replaced explicit rewriting steps with implicit reductions.

7.6.1 Related work

Boutin was the first to use reflection to build decision procedures in Coq, when he gave the initial implementation of the `ring` tactic [5]. Boutin’s reflection step, which he called *syntactification*, was implemented in Coq’s implementation language O’Caml. While this does not damage reliability, it means that a tactic is necessarily tied into the source tree of Coq itself. At the same time, Barthe et al. introduced their idea of the ‘two-level approach’, which formalized the idea of distinguishing between syntax and semantics; however, they dismissed the idea of using reduction/normalization due to efficiency concerns [3]. This work later inspired the `quote` tactic, which performs function inversion and allows one to use the two-level approach without diving into the Coq source tree. Oostdijk and Geuvers presented a proof by reflection tactic for primitive recursive arithmetic; however, efficiency concerns remained and they had no automatic way of performing the reflection step [12].

In all of this work, efficiency was a primary concern. At the time there were many proposals for exporting some of the specialist tasks to computer algebra systems and other external programs. Even Boutin was proposing to extract the decision procedure from Coq to O'Caml [5]. The issue of reliability was the fundamental stumbling block for these ideas. Once we step outside of the well-understood ecosystem of Coq, we lose many of the important safety guarantees.

There have been two very important advancements for Coq, with respect to implementing proof tactics using proof by reflection. The first of these is the introduction of the tactic language Ltac [6]. This remedies the major shortcoming that Oostdijk and Geuvers found in their work, that the reflection step can finally be implemented by system users not system designers. The second improvement is the development of a compiler and a bytecode-based virtual machine for strong reduction in Coq. This provides significant speed-ups over the original interpreter. With these two issues addressed, all the components of proof by reflection can be achieved without needing to leave the Coq ecosystem. Taking advantage of these features, Bertot and Castéran introduced proof by reflection through the example of identities modulo associativity and commutativity in their book [4], Narboux presented a decision procedure for geometry [11], and Grégoire and Mahboubi revisited the `ring` tactic [9].

The more recent PROGRAM extension may be the final key to the puzzle. Its use in the development of our implementation of Whitman's algorithm allowed us to strike a favourable balance between the intertwining of programs and proofs. Our type included a specification of the function's correctness, yet we were not forced to give up the use of Coq's functional programming language to implement the algorithm. The program and the proof are separate, but side by side. Alternatively, we could have used a simple type for leq and developed the correctness proof in its entirety in a separate lemma, however, the PROGRAM extension divided a single proof into 30 bite-sized chunks. This gave us the confidence to tinker with the algorithmic part of the definition with the reasonable expectation that at most only a few proof obligations would change, rather than an entire proof breaking. If one were to attempt to implement an especially complex decision procedure, this would appear to be an invaluable property to have. We conjecture that not only would the correctness proof be more manageable, but the barrier to attempting optimizations would be lowered.

7.6.2 Future work

Unfortunately, Whitman's solution has an exponential running time in the worst case. This is easy to see; cases (2) to (5) have two recursive calls, and Whitman's condition in case (6) has four. In practice, leq runs very quickly on most inputs; however, it is straightforward to construct an input that will force the worst case behaviour. For example:

$$\begin{gathered} x_3 \sqcap x_4 \sqcap (y_3 \sqcup (x_2 \sqcap x_3 \sqcap (y_2 \sqcup (x_1 \sqcap x_2 \sqcap (y_1 \sqcup (x_0 \sqcap x_1)))))) \\ \leq \\ y_3 \sqcup y_4 \sqcup (x_3 \sqcap (y_2 \sqcup y_3 \sqcup (x_2 \sqcap (y_1 \sqcup y_2 \sqcup (x_1 \sqcap (y_0 \sqcup y_1)))))) \end{gathered}$$

This input takes three seconds to falsify (AMD Athlon 2.7GHz), whereas a similarly sized input will usually take fractions of a second.

This is a computational problem that exhibits the two classic properties of *overlapping subproblems* and *optimal substructure*. Possible future work is to turn our current implementation of leq into one that uses dynamic programming to memoize the recursive calls. However, this is not a trivial task. Coq's programming language is purely functional; there are no arrays and no dynamic hash tables, so any data structure that we use for memoization must be purely functional, and operations on that data structure must all be proved terminating.

7.7 CONCLUSION

We have presented a new tactic for the Coq proof system that solves equalities and inequalities in lattices, using an approach known as 'proof by reflection'. The engine of this tactic is our implementation of Whitman's solution to the word problem for free lattices. We have given a strong specification of our implementation and have proved it correct and terminating with respect to this specification. All of the constituent parts of the tactic have been developed within Coq, and there are four key features of Coq that we have used. Firstly, type classes, which have allowed us to succinctly encode the concept of a lattice-ordered set and bind together the operators and relations with the laws they must satisfy. Secondly, the Ltac language has provided the means to interface with users of the proof assistant and write the crucial *reflection* step that transforms a proof goal into our internal representation. Thirdly, Coq's bytecode-based virtual machine has made it tolerable to use the proof by reflection approach with a decision procedure that has an exponential worst case running time. Finally, the PROGRAM extension has allowed us to separate proofs from programs, so that we can implement the computational content of our decision procedure using Coq's functional language and provide the necessary proofs subsequently in a segmented fashion.

REFERENCES

[1] H. Barendregt. Introduction to Generalized Type Systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[2] H. Barendregt and E. Barendsen. Autarkic Computations in Formal Proofs. *Journal of Automated Reasoning*, 28(3):321–336, 1997.

[3] G. Barthe, M. Ruys, and H. Barendregt. A Two-Level Approach towards Lean Proof-Checking. In *Types for Proofs and Programs*, volume 1158 of *LNCS*, pages 16–35. Springer, 1996.

[4] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[5] S. Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In *Theoretical Aspects of Computer Software*, volume 1281 of *LNCS*, pages 515–529. Springer, 1997.

[6] D. Delahaye. A Tactic Language for the System Coq. In *Logic for Programming and Automated Reasoning*, volume 1955 of *LNCS*, pages 85–95. Springer, 2000.

[7] R. S. Freese, J. Ježek, and J. B. Nation. *Free Lattices*. American Mathematical Society, 1995.

[8] B. Grégoire and X. Leroy. A Compiled Implementation of Strong Reduction. In *International Conference on Functional Programming*, pages 235–246. ACM, 2002.

[9] B. Grégoire and A. Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In *Theorem Proving in Higher Order Logics*, volume 3603 of *LNCS*, pages 98–113. Springer, 2005.

[10] J. Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI Cambridge, 1995.

[11] J. Narboux. A Decision Procedure for Geometry in Coq. In *Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 225–240. Springer, 2004.

[12] M. Oostdijk and H. Geuvers. Proof by Computation in the `Coq` System. *Theoretical Computer Science*, 272(1-2):293–314, 2002.

[13] M. Sozeau. Subset Coercions in Coq. In *Types for Proofs and Programs*, volume 4502 of *LNCS*, pages 237–252. Springer, 2007.

[14] M. Sozeau. *Un Environnement pour la Programmation avec Types Dépendants*. PhD thesis, Université de Paris XI, 2008.

[15] M. Sozeau and N. Oury. First-Class Type Classes. In *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, pages 278–293. Springer, 2008.

[16] P. M. Whitman. Free Lattices. *Annals of Mathematics*, 42(1):325–330, 1941.