JFP **32**, e13, 27 pages, 2022.

doi:10.1017/S0956796822000107

TUTORIAL

Fold–unfold lemmas for reasoning about recursive programs using the Coq proof assistant‡

OLIVIER DANVY

Yale-NUS College and School of Computing, National University of Singapore, Singapore, Singapore
(*e-mail:* danvy@acm.org)

Abstract

Fold–unfold lemmas complement the rewrite tactic in the Coq Proof Assistant to reason about recursive functions, be they defined locally or globally. Each of the structural cases gives rise to a fold–unfold lemma that equates a call to this function in that case with the corresponding case branch. As such, they are "boilerplate" and can be generated mechanically, though stating them by hand is a learning experience for a beginner, to say nothing about explaining them. Their proof is generic. Their use is precise (e.g., in terms with multiple calls) and they scale seamlessly (e.g., to continuation-passing style and to various patterns of recursion), be the reasoning equational or relational. In the author's experience, they prove effective in the classroom, considering the clarity of discourse in the subsequent term reports and oral exams, and beyond the classroom, considering their subsequent use when continuing to work with the Coq Proof Assistant. Fold–unfold lemmas also provide a measure of understanding as well as of control about what is cut short when one uses a shortcut, i.e., an automated simplification tactic. Since Version 8.0, the functional-induction plugin provides them for functions that are defined globally, i.e., recursive equations, and so does the Equations plugin now, both for global and for local declarations, a precious help for advanced users.

1 Introduction and motivation

Not so many concepts in mathematics are defined recursively, but triviality is one of them: in the jargon of mathematicians, something is "trivial" when one understands it immediately or when one can prove it trivially. And a measure of success, when undertaking a mathematical journey, is the number of things that have become trivial by the end of this journey. Other measures of success include the ability to reuse the results and the methods that were acquired in the course of this journey, and the ability to explain these new methods, these new results, and these new trivialities, as per Boileau's French alexandrines "what one understands well is told with clarity; and the words to say it arise easily" (1815).

‡The online version of this article has been updated since original publication. A notice detailing the changes has also been published at: https://doi.org/10.1017/S0956796823000011

Rota once singled out, among mathematicians, the "problem solvers" and the "theorizers" (1996). Problem solvers are after the goal, the more dazzling the better: the point of getting to a fountain is to drink, the sooner the better. Theorizers (or "theory builders" Gowers, 2000) concur with the Little Prince about getting to the fountain; they see value in the means, the more enlightening the better: the point of getting to the fountain is not just to make a splash. Rota pointed out that most mathematicians are a mixture of theorizer and problem solver.

Computer scientists too oscillate between solving problems and developing tools to solve problems (or to develop other tools). As a result, our middle path is meandering. Take a statically typed programming language, for example. Do we care about why our program has a type? No of course, we are too busy forging ahead in the safety that evaluation cannot go wrong. But if the type inferencer rejects our program, oh, the pain: we are forced to understand the error message and therefore either or all of the type system, the type inferencer, and its implementation, when all we care about is for the type inferencer to accept our program so that we can forge ahead. In this context, the middle path is to know enough about the type system to understand the common error messages of the type inferencer, an extensible middle ground where we know enough theory to solve problems.

The situation is dual for proving a proposition in the Coq Proof Assistant Bertot & Castéran (2004): instead of using a type inferencer to infer a type for a given program, we need to infer a program (i.e., write a proof) for a given type (i.e., a proposition). The Coq Proof Assistant offers a fertile ground for problem solvers and for theorizers and has made it possible for students to prove more theorems in a few weeks or months than in their entire lifetime so far. On this fertile ground, new notions of triviality have emerged, such as the common succession of proof steps and proof strategies, and these mechanical steps and strategies have been automated. Today, the culture surrounding the Coq Proof Assistant is by and large that of advanced problem solvers: proving the goal justifies any automated means. For example, "Software Foundations" (2012) uses the `simpl` tactic early in the game. This approach is effective when the means are trivial, but triviality is an evolving concept: at the beginning, nothing is trivial. For another example, in his textbook (2013), Chlipala systematically uses the `crush` tactic upfront, consistently with the philosophy of having "short and automated proofs [as] the starting point." The discourse then becomes less about the proofs and more about their automation, turning users into power-users.

For newcomers who simply want to become users, this current culture is challenging: for sure, using shortcuts without understanding what these automated simplification tactics cut short does enable one to prove many propositions. However, when one becomes stuck, it can be with no cognitive standing and no traction because these many proof trees now form a forest that is hiding the tree of knowledge, so to speak. For a less extreme but no less telling example, it also can be a sobering exercise to try to revisit textbook proofs that use a shortcut without using this shortcut.

All in all, the path taken by problem solvers is too steep for newcomers. What about the path a theorizer could take? This path would probably start with building a comfort zone with the basics, which includes understanding each proof step to build muscle memory in order to acquire a sense of how one mechanizes a proof in this proof assistant.

Over the last decade Danvy (2011 – 2021), the author has been patiently designing such a path for newcomers Danvy (2019b). Against this backdrop, the goal of the present tutorial is to describe how to mechanize equational reasoning about recursive programs. In the

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

course of the 1970s, this equational reasoning was promoted by Burstall & Landin (1969), Manna (1974) and Burstall and Darlington (1977), and since the 1980s, it is in common use to reason about functional programs (Bird & Wadler, 1988; Thompson, 1995; Thompson, 2011), e.g., to manipulate them Jones *et al.* (1993). At the heart of this mechanical reasoning lie fold–unfold steps to relate (a) calls to a structurally recursive function with arguments that correspond to one of its base cases or to one of its induction steps and (b) the corresponding case in its definition. It is the author's thesis that these fold–unfold steps, once stated as fold–unfold lemmas, provide normal Coq users with an actionable understanding through a tractable middle path – the path of a theorizer – where they can distinguish between the trivial and the nontrivial and explain the whole process with clarity. The present tutorial is an overview of this middle path.

Roadmap: We start off with equational reasoning about the inductive specifications of programs (Section 2). Fold–unfold lemmas account for the clauses in the inductive specification of a program (Section 3), and they make it possible to reason about recursive programs (Section 4). In fact, they typically coincide with these clauses (Section 5), which we illustrate next with continuation-passing style, functionals that abstract structurally recursive computations (i.e., fold functions), mutual recursion, and with proving properties of recursive and co-recursive functions as well as the course-of-values induction principle (Section 6). We then turn to proving fold–unfold lemmas (Section 7), be it directly or using the functional-induction plugin or the Equations plugin, and to interfacing with preexisting recursive functions. Thus equipped, we describe how to reason not only about recursive equations but also about block-structured programs that use lexical scope (Section 8). We then put the present work into perspective (Section 9) before concluding (Section 10).

Prequisites and notations: An elementary knowledge of recursive programs, e.g., in OCaml, and no particular knowledge of the Coq Proof Assistant are expected from the reader. A deliberately reduced vocabulary of tactics is used here, each of them with one clear effect in the `*goals*` window (we are using Proof General), and all of them reviewed in appendix.

The term "fold–unfold" is due to Burstall and Darlington (1977) and is read "fold after unfold:"

- The term "unfold" is used when a name is replaced by is denotation, and the Coq tactic `unfold` does exactly that. When the name denotes a function and occurs in a function call, this call is "unfolded" when this function is inlined: an instance of its body replaces the call where the actual parameters are substituted for its formal parameters. For example, if `foo` denotes `fun x => if x then 1 else 0`, then the call `foo y` is unfolded into `if y then 1 else 0` using the tactic `unfold foo`. Some simplifications can take place during this inlining. For example, the call `foo true` is unfolded into `1` using the tactic `unfold foo`.
- The term "fold" is used when a denotation is replaced by its name, and the Coq tactic `fold` does exactly that. When the name denotes a function and an expression is an instance of the body of this function, this instance is "folded" when this instance is replaced by a call to this function with suitable actual parameters. For example, if `foo` denotes `fun x => if x then 1 else 0`, then the expression `if y then 1 else 0` is folded into `foo y` using the tactic `fold (foo y)`, and the expression `1` is folded into `foo true` using the tactic `fold (foo true)`.

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

As defined by Burstall et al., a fold–unfold step consists in (1) unfolding the call to a function and then (2) folding another call to this function in the resulting expression. Fold–unfold steps are most relevant for recursive functions because they make it possible to focus on what a recursive function does and disregard how it is implemented. So for example, if `list_map` denotes the recursive map function over lists, the two expressions `list_map f (v :: vs)` and `f v :: list_map f vs` are interderivable using one fold–unfold step.

Target audience and goal: This tutorial is written for people who are interested in reasoning about programs and who are curious about formulating this reasoning on a computer in a domain-specific language for writing proofs. The proofs and their pace are elementary to give the reader a chance to build a new knowledge on top of an existing knowledge instead of building yet another independent knowledge from scratch, kaleidoscopically. A not-uncommon side effect of this paced approach is a consolidation of this existing knowledge. And as in martial arts, the reader has then a chance to reach a mindful position where they can accelerate their moves safely, a side effect of having been patiently introduced to a logic of computing instead of having been made to start with its automation.

I want to give you
what I wish someone had given me
when I was your age.
– Samson Raphaelson (1949)

2 Equational reasoning about specifications of programs

In the Coq Proof Assistant, equational reasoning is achieved using the `rewrite` tactic. This tactic, when applied to the Leibniz equality of two expressions, replaces one of these expressions by the other. Consider, for an elementary example, the mirroring function over binary trees in Gallina, the resident pure and total functional language in Coq:

```
Inductive binary_tree (V : Type) : Type :=
| Leaf : V -> binary_tree V
| Node : binary_tree V -> binary_tree V -> binary_tree V.

Definition specification_of_mirror
               (mirror : forall V : Type, binary_tree V -> binary_tree V) : Prop :=
  (forall (V : Type) (v : V),
      mirror V (Leaf V v) = Leaf V v)
  /\
  (forall (V : Type) (t1 t2 : binary_tree V),
      mirror V (Node V t1 t2) = Node V (mirror V t2) (mirror V t1)).
```

The first declaration defines the polymorphic type of binary trees with payloads in the leaves. The second specifies the polymorphic mirror function inductively with two conjuncts, one for the leaves (base case) and one for the nodes (induction step).

To illustrate equational reasoning already, let us prove that at most one function satisfies this specification:

```
Proposition there_is_at_most_one_mirror :
  forall mirror1 mirror2 : forall V : Type, binary_tree V -> binary_tree V,
    specification_of_mirror mirror1 ->
    specification_of_mirror mirror2 ->
    forall (V : Type) (t : binary_tree V),
      mirror1 V t = mirror2 V t.
```

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

In words: any two functions `mirror1` and `mirror2` that satisfy the specification of the mirror function map the same input to the same output. The proof starts by naming assumptions and isolating the equality in the goal. Consistently with the policy that each tactic has one clearly identifiable effect in the `*goals*` window, we use an unfold step to replace the name `specification_of_mirror` by its denotation, i.e., a conjunction, before naming its two components:

```
Proof.
  intros mirror1 mirror2.
  unfold specification_of_mirror.
  intros [S_mirror1_Leaf S_mirror1_Node] [S_mirror2_Leaf S_mirror2_Node].
  intros V t.
```

And indeed the `*goals*` buffer then contains named assumptions and the equality as a goal:

```
  mirror1 : forall V : Type, binary_tree V -> binary_tree V
  mirror2 : forall V : Type, binary_tree V -> binary_tree V
  S_mirror1_Leaf : forall (V : Type) (v : V), mirror1 V (Leaf V v) = Leaf V v
  S_mirror1_Node : forall (V : Type) (t1 t2 : binary_tree V),
                   mirror1 V (Node V t1 t2) = Node V (mirror1 V t2) (mirror1 V t1)
  S_mirror2_Leaf : forall (V : Type) (v : V), mirror2 V (Leaf V v) = Leaf V v
  S_mirror2_Node : forall (V : Type) (t1 t2 : binary_tree V),
                   mirror2 V (Node V t1 t2) = Node V (mirror2 V t2) (mirror2 V t1)
  V : Type
  t : binary_tree V
  ============================
  mirror1 V t = mirror2 V t
```

This equality is proved by structural induction. We first focus on the base case:

```
  induction t as [v | t1 IHt1 t2 IHt2].
  -
```

The `*goals*` buffer is the same as before, modulo the following changes:

- the goal now reads `mirror1 V (Leaf V v) = mirror2 V (Leaf V v)`, manifesting that we are in the base case where `t` is `Leaf v`; and
- `v : V` stands instead of `t : binary_tree V` as the last assumption; `v` is the argument of the `Leaf` constructor.

Instantiating `S_mirror1_Leaf` and `S_mirror2_Leaf` with `V` and `v` provides two Leibniz equalities, one that matches the left-hand side of the goal and the other that matches the right-hand side of the goal. Based on these equalities, one can replace equals by equals in the goal using the `rewrite` tactic from left to right:

```
    rewrite -> (S_mirror1_Leaf V v).
    rewrite -> (S_mirror2_Leaf V v).
```

The goal then reads `Leaf V v = Leaf V v`, which can be proved using the `reflexivity` tactic since Leibniz equality is reflexive. The base case being established, one can next focus on the induction step:

```
    reflexivity.
  -
```

The `*goals*` buffer now reads:

```
  ...
  V : Type
  t1, t2 : binary_tree V
  IHt1 : mirror1 V t1 = mirror2 V t1
  IHt2 : mirror1 V t2 = mirror2 V t2
  ============================
  mirror1 V (Node V t1 t2) = mirror2 V (Node V t1 t2)
```

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

It is again the same as before, modulo the following changes:

- the goal now reads `mirror1 V (Node V t1 t2) = mirror2 V (Node V t1 t2)`, manifesting that we are in the induction step where `t` is `Node V t1 t2`; and
- `t1, t2 : binary_tree V` stands instead of `t : binary_tree V` in the last assumption; `t1` and `t2` are the arguments of the `Node` constructor, and they stand along their associated induction hypotheses `IHt1` and `IHt2`.

Instantiating `S_mirror1_Node` and `S_mirror2_Node` with `V`, `t1`, and `t2` provides two Leibniz equalities, one that matches the left-hand side of the goal and the other that matches its right-hand side. Based on these equalities, one can replace equals by equals in the goal using the `rewrite` tactic:

```
  rewrite -> (S_mirror1_Node V t1 t2).
  rewrite -> (S_mirror2_Node V t1 t2).
```

The goal then reads `Node V (mirror1 V t2) (mirror1 V t1) = Node V (mirror2 V t2) (mirror2 V t1)`. The induction hypotheses provide two Leibniz equalities which can be used to replace equals by equals in the goal:

```
  rewrite -> IHt1.
  rewrite -> IHt2.
```

The goal then reads `Node V (mirror2 V t2) (mirror2 V t1) = Node V (mirror2 V t2) (mirror2 V t1)`, which again can be proved using the `reflexivity` tactic, which concludes the proof:

```
  reflexivity.
Qed.
```

Likewise, one can prove equationally that a function satisfying the specification of mirror is involutory, i.e., that composing it with itself yields the identity function:

```
Theorem a_function_that_satisfies_the_specification_of_mirror_is_involutory :
  forall mirror : forall V : Type, binary_tree V -> binary_tree V,
    specification_of_mirror mirror ->
    forall (V : Type) (t : binary_tree V),
      mirror V (mirror V t) = t.
```

The proof is much as the one above, each step explicit and each of them with a simple explanation:

```
Proof.
  intro mirror.
  unfold specification_of_mirror.
  intros [S_mirror_Leaf S_mirror_Node].
  intros V t.
  (* goal: mirror V (mirror V t) = t *)
  induction t as [v | t1 IHt1 t2 IHt2].
  { (* goal: mirror V (mirror V (Leaf V v)) = Leaf V v *)
    rewrite -> (S_mirror_Leaf V v).
    (* goal: mirror V (Leaf V v) = Leaf V v *)
    rewrite -> (S_mirror_Leaf V v).
    (* goal: Leaf V v = Leaf V v *)
    reflexivity. }
  { (* IHt1 : mirror V (mirror V t1) = t1 *)
    (* IHt2 : mirror V (mirror V t2) = t2 *)
    (* goal: mirror V (mirror V (Node V t1 t2)) = Node V t1 t2 *)
    rewrite -> (S_mirror_Node V t1 t2).
    (* goal: mirror V (Node V (mirror V t2) (mirror V t1)) = Node V t1 t2 *)
    rewrite -> (S_mirror_Node V (mirror V t2) (mirror V t1)).
    (* goal: Node V (mirror V (mirror V t1)) (mirror V (mirror V t2)) = Node V t1 t2 *)
    rewrite -> IHt1.
    (* goal: Node V t1 (mirror V (mirror V t2)) = Node V t1 t2 *)
    rewrite -> IHt2.
    (* goal: Node V t1 t2 = Node V t1 t2 *)
    reflexivity. }
Qed.
```

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

Both proofs feature equational reasoning, witness their extensive use of the `rewrite` tactic. The parts specific to the mirror function (`S_mirror_Leaf`, `S_mirror_Node`) are carried out based on its specification. Each time, a call to `mirror` on a leaf or on a node is replaced by the corresponding expression equated by the specification, which is called a "fold–unfold step" since Burstall and Darlington.

Would not it be nice if we could mechanically reason about programs – instead of about their specification – using fold–unfold steps?

For example, consider the following (obvious) implementation of the mirror function in direct style:

```
Fixpoint mirror (V : Type) (t : binary_tree V) : binary_tree V :=
  match t with
  | Leaf _ v     => Leaf V v
  | Node _ t1 t2 => Node V (mirror V t2) (mirror V t1)
  end.
```

This implementation begs for the two following statements, one to the effect that it satisfies the specification and one to the effect that it is involutory. (The second is proved as a corollary of the first just below, but we would also like a direct proof.)

```
Proposition there_is_at_least_one_mirror_function :
  specification_of_mirror mirror.
Admitted. (* for now *)

Theorem mirror_is_involutory :
  forall (V : Type) (t : binary_tree V),
    mirror V (mirror V t) = t.
Proof.
  exact (a_function_that_satisfies_the_specification_of_mirror_is_involutory
           mirror
           there_is_at_least_one_mirror_function).
Qed.
```

In words: applying the previous theorem to suitable arguments yields a quantified Leibniz equality that coincides with the statement of the theorem.

Besides `rewrite`, the Coq Proof Assistant offers two tactics to carry out fold–unfold steps: `unfold` that replaces a name by its denotation, and `fold` that replaces a denotation by an existing name. Using the `unfold` tactic over a recursive function exposes its internal representation and using the `fold` tactic is often chancy in this context, especially when the name is parameterized. Worse, this strategy does not scale as the programs to reason about become more complex.

The source of the problem is that the Coq users need to strategize every time they need to perform a fold–unfold step, i.e., every time a function is called.

Instead, we suggest that fold–unfold lemmas be stated once, when a recursive function is declared, and used when a function is called, as a predefined strategy.

3 Fold–unfold lemmas

Let us revisit the implementation of the mirror function:

```
Fixpoint mirror (V : Type) (t : binary_tree V) : binary_tree V :=
  match t with
  | Leaf _ v     => Leaf V v
  | Node _ t1 t2 => Node V (mirror V t2) (mirror V t1)
  end.
```

By completeness, the mirror function can either be applied to a leaf or to a node, which elicits two fold–unfold lemmas:

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

```
Lemma fold_unfold_mirror_Leaf :
  forall (V : Type) (v : V),
    mirror V (Leaf V v) = ....

Lemma fold_unfold_mirror_Node :
  forall (V : Type) (t1 t2 : binary_tree V),
    mirror V (Node V t1 t2) = ...
```

The first lemma aims to state the Leibniz equality associated to a call to the mirror function on a leaf and the second the Leibniz equality associated to a call to the mirror function on a node. Now what should one write instead of the ellipses? Answer – one should copy-paste the corresponding branch in the implementation:

```
Lemma fold_unfold_mirror_Leaf :
  forall (V : Type) (v : V),
    mirror V (Leaf V v) = Leaf V v.
Admitted. (* for now *)

Lemma fold_unfold_mirror_Node :
  forall (V : Type) (t1 t2 : binary_tree V),
    mirror V (Node V t1 t2) = Node V (mirror V t2) (mirror V t1).
Admitted. (* for now *)
```

4 Equational reasoning about recursive programs

Witness the following proof, the fold–unfold lemmas from Section 3 enable one to reason equationally as in the proofs in Section 2:

```
Theorem mirror_is_involutory_alt :
  forall (V : Type) (t : binary_tree V),
    mirror V (mirror V t) = t.
```

The proof starts by naming assumptions and isolating the equality in the goal:

```
Proof.
  intros V t.
```

And indeed the `*goals*` buffer then contains named assumptions and the equality as a goal. Compared to the `*goals*` buffer at the beginning of Section 2, only `V` and `t` are declared in the assumptions: the other assumptions (declaration of `mirror` and fold–unfold lemmas are now global:

```
  V : Type
  t : binary_tree V
  ============================
  mirror V (mirror V t) = t
```

The equality is proved by structural induction. We first focus on the base case:

```
  induction t as [v | t1 IHt1 t2 IHt2].
  -
```

The `*goals*` buffer is the same as before, modulo the following changes:

- the goal now reads `mirror V (mirror V (Leaf v)) = Leaf v`, manifesting that we are in the base case where `t` is `Leaf v`; and
- `v : V` stands instead of `t : binary_tree V` as the last assumption; `v` is the argument of the `Leaf` constructor.

Instantiating `fold_unfold_mirror_Leaf` with `V` and `v` provides a Leibniz equality that matches a subterm in the left-hand side of the goal. Based on this equality, one can replace equals by equals in the goal using the `rewrite` tactic:

```
    rewrite -> (fold_unfold_mirror_Leaf V v).
```

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

The goal now coincides with this Leibniz equality, which makes it possible to complete the base case and to focus next on the induction step:

```
  exact (fold_unfold_mirror_Leaf V v).
-
```

The `*goals*` buffer now reads:

```
V : Type
t1, t2 : binary_tree V
IHt1 : mirror V (mirror V t1) = t1
IHt2 : mirror V (mirror V t2) = t2
============================
mirror V (mirror V (Node V t1 t2)) = Node V t1 t2
```

It is again the same as before, modulo the following changes:

- the goal now reads `mirror V (mirror V (Node V t1 t2)) = Node V t1 t2`, manifesting that we are in the induction step where `t` is `Node V t1 t2`; and
- `t1, t2 : binary_tree V` stands instead of `t : binary_tree V` in the last assumption; `t1` and `t2` are the arguments of the `Node` constructor, and they stand along their associated induction hypotheses `IHt1` and `IHt2`.

One instantiation of `fold_unfold_mirror_Node` provides a Leibniz equality that matches a subterm in the left-hand side of the goal. Based on this equality, one can replace equals by equals in the goal using the `rewrite` tactic:

```
  rewrite -> (fold_unfold_mirror_Node V t1 t2).
```

The goal then reads `mirror V (Node V (mirror V t2) (mirror V t1)) = Node V t1 t2`. Another instantiation of `fold_unfold_ mirror_Node` provides a Leibniz equality that matches its left-hand side. Based on this equality, one can replace equals by equals in the goal using the `rewrite` tactic:

```
  rewrite -> (fold_unfold_mirror_Node V (mirror V t2) (mirror V t1)).
```

The goal then reads `Node V (mirror V (mirror V t1)) (mirror V (mirror V t2)) = Node V t1 t2`. The induction hypotheses provide two Leibniz equalities which can be used to replace equals by equals in the goal:

```
  rewrite -> IHt1.
  rewrite -> IHt2.
```

The goal then reads `Node V t1 t2 = Node V t1 t2`, which is proved using the `reflexivity` tactic, which concludes the proof:

```
  reflexivity.
Qed.
```

We find this formal proof remarkable because both in form and in content, it mechanizes an informal proof (Manna, 1974; Burge, 1975; Bird & Wadler, 1988). Such a pen-and-paper proof is typically a series of equalities, each of them with a justification. Here, the successive goals enumerate these equalities, and each proof step propels the proof from goal to goal. This mechanization comforts the newcomers that their knowledge is not only actionable but also building up.

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

5 A closer look at the fold–unfold lemmas

Witness the following proof, the two fold–unfold lemmas above coincide with the two conjuncts in the specification of the mirror function, a consequence of the implementation being in direct style:

```
Proposition there_is_at_least_one_mirror_function :
  specification_of_mirror mirror.
Proof.
  unfold specification_of_mirror.
  exact (conj fold_unfold_mirror_Leaf fold_unfold_mirror_Node).
Qed.
```

This coincidence exemplifies how fold–unfold lemmas provide the same expressive power to reason about recursive programs as the inductive specification of these recursive programs.

6 Applications

Fold–unfold lemmas scale seamlessly, e.g., to continuation-passing style (Section 6.1), fold functions (Section 6.2), and mutual recursion (Section 6.3), as well as to proving properties of recursive and co-recursive functions (Section 6.4) and to proving the course-of-values induction principle (Section 6.5).

6.1 Continuation-passing style

Consider the left-to-right continuation-passing implementation of the mirror function:

```
Fixpoint mirror'_cps (Ans V : Type) (t : binary_tree V) (k : binary_tree V -> Ans) :
                                                                          Ans :=
  match t with
  | Leaf _ v     =>
    k (Leaf V v)
  | Node _ t1 t2 =>
    mirror'_cps Ans V t1 (fun t1_m =>
                            mirror'_cps Ans V t2 (fun t2_m =>
                                                    k (Node V t2_m t1_m)))
  end.

Definition mirror' (V : Type) (t : binary_tree V) : binary_tree V :=
  mirror'_cps (binary_tree V) V t (fun t_m => t_m).
```

In words, the implementation is tail-recursive and uses a continuation that is initialized in the main function, `mirror'`. As a continuation-passing function, `mirror'_cps` is now polymorphic both with respect to the payloads in the leaves of the binary tree to mirror and with respect to a domain of answers and is also parameterized with a continuation that maps an intermediate result to an answer. In the base case, the computation continues by applying the current continuation to a new leaf. In the induction step, and given a current continuation, `mirror'_cps` tail-calls itself on the left subtree with a new continuation that, when applied, tail-calls `mirror'_cps` on the right subtree with another continuation that, when applied, sends a new node to the current continuation.

By completeness, the auxiliary function can either be applied to a leaf or to a node, which elicits two fold–unfold lemmas:

```
Lemma fold_unfold_mirror'_cps_Leaf :
  forall (Ans V : Type) (v : V) (k : binary_tree V -> Ans),
    mirror'_cps Ans V (Leaf V v) k = ...
Proof. ...
```

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

```
Lemma fold_unfold_mirror'_cps_Node :
  forall (Ans V : Type) (t1 t2 : binary_tree V) (k : binary_tree V -> Ans),
    mirror'_cps Ans V (Node V t1 t2) k = ...
Proof. ...
```

The first lemma aims to state the Leibniz equality associated to a call to the auxiliary function on a leaf, and the second the Leibniz equality associated to a call to the auxiliary function on a node. Now what should one write instead of the ellipses? Answer – one should copy-paste the corresponding branch in the implementation, just as in Section 3:

```
Lemma fold_unfold_mirror'_cps_Leaf :
  forall (Ans V : Type) (v : V) (k : binary_tree V -> Ans),
    mirror'_cps Ans V (Leaf V v) k =
    k (Leaf V v).
Admitted. (* for now *)

Lemma fold_unfold_mirror'_cps_Node :
  forall (Ans V : Type) (t1 t2 : binary_tree V) (k : binary_tree V -> Ans),
    mirror'_cps Ans V (Node V t1 t2) k =
    mirror'_cps Ans V t1 (fun t1_m =>
                            mirror'_cps Ans V t2 (fun t2_m =>
                                                    k (Node V t2_m t1_m))).
Admitted. (* for now *)
```

These fold–unfold lemmas enable one to prove the completeness of `mirror'_cps` equationally and its soundness relationally:

```
Lemma completeness_of_mirror'_cps :
  forall (V : Type) (t t_m : binary_tree V),
    mirror V t = t_m ->
    forall (Ans : Type) (k : binary_tree V -> Ans),
      mirror'_cps Ans V t k = k t_m.
Proof. ... Qed. (* see the accompanying .v file *)
```

In words: if `mirror` maps a binary tree `t` to its mirror image `t_m`, then applying `mirror'_cps` to `t` and a continuation `k` leads to `k` being applied to `t_m`. The equivalence of `mirror'` and `mirror` is a corollary:

```
Theorem equivalence_of_mirror'_and_mirror :
  forall (V : Type) (t : binary_tree V),
    mirror' V t = mirror V t.
Proof.
  intros V t.
  unfold mirror'.
  exact (completeness_of_mirror'_cps
           V t (mirror V t) (eq_refl (mirror V t)) (binary_tree V) (fun t_m => t_m)).
Qed.
```

Witness the accompanying .v file, the completeness lemma is proved by structural induction on `t`, using the fold–unfold lemmas in the base case and in the induction step. As for the soundness lemma, which admittedly is not needed here, it requires one to prove that continuations are injective if the initial continuation is also injective, which is the case here (see the accompanying .v file).

6.2 Functionals that abstract structurally recursive computations

Consider the following fold functionals over Peano numerals:

```
Definition nat_fold_left (V : Type) (z : V) (s : V -> V) (n : nat) : V :=
  let fix loop n a :=
    match n with
    | O    => a
    | S n' => loop n' (s a)
    end
  in loop n z.

Definition nat_fold_right (V : Type) (z : V) (s : V -> V) (n : nat) : V :=
  let fix visit n :=
```

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

```
    match n with
    | O    => z
    | S n' => s (visit n')
    end
  in visit n.
```

The first functional abstracts a tail-recursive computation over a natural number using an accumulator and the second an ordinary recursive computation over a natural number. And as it happens, these two functionals are equivalent Danvy (2019a):

```
Theorem folding_left_and_right :
  forall (V : Type) (z : V) (s : V -> V) (n : nat),
    nat_fold_left V z s n = nat_fold_right V z s n.
Proof. ... Qed. (* see the accompanying .v file *)
```

This theorem is proved by induction on `n`, using either of the following lemmas, which are also proved by induction on `n`, witness the accompanying .v file:

```
Lemma about_nat_fold_left :
  forall (V : Type) (z : V) (s : V -> V) (n : nat),
    nat_fold_left V (s z) s n = s (nat_fold_left V z s n).
Proof. ... Qed. (* see the accompanying .v file *)

Lemma about_nat_fold_right :
  forall (V : Type) (z : V) (s : V -> V) (n : nat),
    nat_fold_right V (s z) s n = s (nat_fold_right V z s n).
Proof. ... Qed. (* see the accompanying .v file *)
```

These three proofs all use the fold–unfold lemmas associated to the two functionals, making it possible for the user to focus on what to prove instead of also having to worry about how to proceed at every proof step:

```
Lemma fold_unfold_nat_fold_left_O :
  forall (V : Type) (z : V) (s : V -> V),
    nat_fold_left V z s O = z.
Admitted. (* for now *)

Lemma fold_unfold_nat_fold_left_S :
  forall (V : Type) (z : V) (s : V -> V) (n' : nat),
    nat_fold_left V z s (S n') = nat_fold_left V (s z) s n'.
Admitted. (* for now *)

Lemma fold_unfold_nat_fold_right_O :
  forall (V : Type) (z : V) (s : V -> V),
    nat_fold_right V z s O = z.
Admitted. (* for now *)

Lemma fold_unfold_nat_fold_right_S :
  forall (V : Type) (z : V) (s : V -> V) (n' : nat),
    nat_fold_right V z s (S n') = s (nat_fold_right V z s n').
Admitted. (* for now *)
```

Both Bird and Wadler's duality theorems about lists (1988) and their ancestors Burge (1975) are formalized in a similar manner.

6.3 Mutual recursion

Consider the canonical implementation of the parity predicates:

```
Fixpoint evenp (n : nat) : bool :=
  match n with
  | O    => true
  | S n' => oddp n'
  end
with oddp (n : nat) : bool :=
       match n with
       | O    => false
       | S n' => evenp n'
       end.
```

This implementation gives rise to the following 4 fold–unfold lemmas, admitted for now:

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

```
Lemma fold_unfold_evenp_O : evenp 0 = true.  Admitted.

Lemma fold_unfold_evenp_S : forall n' : nat, evenp (S n') = oddp n'.  Admitted.

Lemma fold_unfold_oddp_O : oddp 0 = false.  Admitted.

Lemma fold_unfold_oddp_S : forall n' : nat, oddp (S n') = evenp n'.  Admitted.
```

These lemmas were mechanically stated by enumerating all the possible calls to `evenp` and `oddp` and by copy-pasting the corresponding branch in the implementation.

They enable one to prove the soundness and the completeness of the canonical implementation:

```
Theorem soundness_and_completeness_of_evenp_and_oddp :
  forall n : nat,
    (evenp n = true <-> exists m : nat, n = 2 * m)
    /\
    (oddp n = true <-> exists m : nat, n = S (2 * m)).
Proof. ... Qed. (* see the accompanying .v file *)
```

The theorem is proved by structural induction on `n`, using the fold–unfold lemmas in the base case and in the induction step. One can then verify, for example, that the converse cases are corollaries of soundness and completeness:

```
Corollary soundness_and_completeness_of_evenp_and_oddp_too :
  forall n : nat,
    (evenp n = false <-> exists m : nat, n = S (2 * m))
    /\
    (oddp n = false <-> exists m : nat, n = 2 * m).
Proof. ... Qed. (* see the accompanying .v file *)
```

6.4 Proving properties of recursive and co-recursive functions

Witness the accompanying .v file, fold–unfold lemmas come handy to prove, e.g., that list concatenation is associative, which is a standard example. Ditto for for proving that the number of leaves in a binary tree equals its number of nodes, plus 1.

The idea readily transfers to streams:

```
CoInductive stream (V : Type) : Type :=
| Cons : V -> stream V -> stream V.
```

Consider, for a playful example, a function that ostensibly concatenates one stream to another:

```
CoFixpoint stream_append (V : Type) (v1s v2s : stream V) :=
  match v1s with
  | Cons _ v1 v1s' =>
    Cons V v1 (stream_append V v1s' v2s)
  end.
```

In order to prove that concatenating a given stream to another yields a stream that is bisimilar to the given stream, the following fold–unfold lemma provides a very useful stepping stone:

```
Lemma fold_unfold_stream_append :
  forall (V : Type) (v1 : V) (v1s' v2s : stream V),
    stream_append V (Cons V v1 v1s') v2s = Cons V v1 (stream_append V v1s' v2s).
Proof. ... Qed. (* see the accompanying .v file *)
```

6.5 Proving the course-of-values induction principle

Given (1) a predicate `P` indexed by a natural number and (2) a natural number `n`, the following function computes `P n /\ ... /\ P 1 /\ P 0`:

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

```
Fixpoint course_of_values (P : nat -> Prop) (n : nat) : Prop :=
  match n with
  | 0    => P 0
  | S n' => P (S n') /\ course_of_values P n'
end.
```

Thus equipped, the course-of-values induction principle is stated as follows:

```
Lemma nat_ind_course_of_values :
  forall P : nat -> Prop,
    P 0 ->
    (forall k : nat,
        course_of_values P k -> P (S k)) ->
    forall n : nat,
      P n.
```

In the accompanying .v file, this induction principle is proved by mathematical induction using the fold–unfold lemmas associated to `course_of_values`.

7 Proving the fold–unfold lemmas

7.1 Using the unfold tactic and then the fold tactic

By their very name, fold–unfold lemmas are proved using the `unfold` tactic immediately followed by the `fold` tactic, which yields a Leibniz equality. So the proof of each fold–unfold lemma can be stated as an instance of the following one-liner:

```
Ltac fold_unfold_tactic name := intros; unfold name; fold name; reflexivity.
```

In other words, the proof of both fold–unfold lemmas for the `mirror` function, for example, reads:

```
Proof.
  fold_unfold_tactic mirror.
Qed.
```

That being said, the `reflexivity` tactic itself uses `fold` and `unfold` as part of its normalization steps toward comparing normal forms, and therefore there is no need for these two tactics in the proof of fold–unfold lemmas, witness the following shorter proof script that does not even need the name of the function to unfold and to fold:

```
Ltac fold_unfold := intros; reflexivity.
```

In other words, the proof of each fold–unfold lemma can read:

```
Proof.
  fold_unfold.
Qed.
```

Equivalently, it can read:

```
Proof.
  intros; reflexivity.
Qed.
```

Since the last two proofs look like magic, i.e., require explanations that use concepts that are more advanced than the concepts the students have been presented so far, the previous version (namely `fold_unfold_tactic` applied to the name of the function at hand) does the job better. At some point, a student may wonder why the proof assistant accepts the proof `fold_unfold_tactic foo` for a fold–unfold lemma about `bar`, at which point they are mature enough to be told about the unusual effectiveness of the `reflexivity` tactic in the Coq Proof Assistant (to say nothing of its efficiency Grégoire & Leroy (2002)).

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

Further up the road, the limitation of the fold–unfold tactic will come to light, namely for functions whose recursion pattern depends on more than one argument. For example, consider `List.nth`, which has type `forall V : Type, nat -> list V -> V -> V`. Applying `List.nth` to an index *i*, a list *vs*, and a default value *d* yields *d* if *i* is "out of bounds" and otherwise the element at index *i* in *vs*. Accessing a given list at a given index can equivalently be defined by induction on the index or by induction on the list. Therefore, two pairs of fold–unfold lemmas arise.

- The first pair is by cases on the given index:

```
Lemma fold_unfold_nth_0 :
  forall (V : Type) (vs : list V) (default : V),
    nth 0 vs default = match vs with
                       | nil      => default
                       | v :: vs' => v
                       end.

Lemma fold_unfold_nth_S :
  forall (V : Type) (i' : nat) (vs : list V) (default : V),
    nth (S i') vs default = match vs with
                            | nil      => default
                            | v :: vs' => nth i' vs' default
                            end.
```

- The second pair is by cases on the given list:

```
Lemma fold_unfold_nth_nil :
  forall (V : Type) (i : nat) (default : V),
    nth i nil default = match i with
                        | 0    => default
                        | S i' => default
                        end.

Lemma fold_unfold_nth_cons :
  forall (V : Type) (i : nat) (v : V) (vs' : list V) (default : V),
    nth i (v :: vs') default = match i with
                               | 0    => v
                               | S i' => nth i' vs' default
                               end.
```

One is even tempted to declare 4 fold–unfold lemmas to account for the 4 possibilities:

```
Lemma fold_unfold_nth_0_nil :
  forall (V : Type) (default : V),
    nth 0 nil default = default.

Lemma fold_unfold_nth_0_cons :
  forall (V : Type) (v : V) (vs' : list V) (default : V),
    nth 0 (v :: vs') default = v.

Lemma fold_unfold_nth_S_nil :
  forall (V : Type) (i' : nat) (default : V),
    nth (S i') nil default = default.

Lemma fold_unfold_nth_S_cons :
  forall (V : Type) (i' : nat) (v : V) (vs' : list V) (default : V),
    nth (S i') (v :: vs') default = nth i' vs' default.
```

Each of these 4 lemmas is seamlessly proved using the fold–unfold tactic since the user has done the work of enumerating all cases in the statements of the lemmas. (Incidentally, stating these 4 lemmas could be used to *define* a function to access a given list at a given index, as done, e.g., in Isabelle Nipkow *et al.* (2002).) Both Lemmas `fold_unfold_nth_nil` and `fold_unfold_list_nth_cons` are also seamlessly proved using the fold–unfold tactic. However, since `List.nth` is implemented by recursion on the list, the Coq Proof Assistant balks at proving the two previous lemmas like that. Instead, one needs to customize the following proof that explicitly distinguishes the two cases of a list:

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

```
Proof. (* for the O case *)
  intros V [ | v vs'] default; reflexivity.
Qed.

Proof. (* for the S case *)
  intros V i' [ | v vs'] default; reflexivity.
Qed.
```

And conversely, for a version of `nth` that is implemented by recursion on the index, the two first lemmas are seamlessly proved using the fold–unfold tactic, but for the two others, one needs to customize the following proof that explicitly distinguishes the two cases of an index:

```
Proof. (* for the nil case *)
  intros V [ | i'] default; reflexivity.
Qed.

Proof. (* for the cons case *)
  intros V [ | i'] v vs' default; reflexivity.
Qed.
```

In other words, to state fold–unfold lemmas for a recursive function, one either needs to know how this function is implemented or one needs to enumerate all possible configurations of its arguments by eta expanding them according to their type, as it were.

7.2 Using the functional-induction plugin

The functional-induction plugin (`FunInd`) provides a `Function` vernacular. This vernacular appeared in Coq v8.0 as part of this plugin, which was loaded by default. Since Coq v8.7, this plugin must be explicitly loaded with `Require FunInd`. When using `Function` instead of `Fixpoint`, an equation is generated – its "fixpoint equality," to quote the section about Advanced Recursive Functions in the Coq Reference Manual:

https://coq.github.io/doc/V8.11.2/refman/language/gallina-extensions.html#advanced-recursive-functions

This equation has direct relevance to fold–unfold lemmas.

For example, let us revisit the mirror function, declaring it with `Function` instead of with `Fixpoint`:

```
Function mirror_alt (V : Type) (t : binary_tree V) : binary_tree V :=
  match t with
  | Leaf _ v     => Leaf V v
  | Node _ t1 t2 => Node V (mirror_alt V t2) (mirror_alt V t1)
  end.
```

A by-product of this declaration is that from then on, `mirror_alt_equation` (i.e., the name of the function declared with `Function`, concatenated with `_equation`) has type

```
forall (V : Type) (t : binary_tree V),
  mirror_alt V t =
  match t with
  | Leaf _ v     => Leaf V v
  | Node _ t1 t2 => Node V (mirror_alt V t2) (mirror_alt V t1)
  end
```

It is then immediate to declare the fold–unfold lemmas:

```
Definition fold_unfold_mirror_alt_Leaf (V : Type) (v : V) :=
  mirror_alt_equation V (Leaf V v).

Definition fold_unfold_mirror_alt_Node (V : Type) (t1 t2 : binary_tree V) :=
  mirror_alt_equation V (Node V t1 t2).
```

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

And indeed their type reads as follows:

```
forall (V : Type) (v : V),
  mirror_alt V (Leaf V v) = Leaf V v

forall (V : Type) (t1 t2 : binary_tree V),
  mirror_alt V (Node V t1 t2) = Node V (mirror_alt V t2) (mirror_alt V t1)
```

These lemmas hold as instances of the equation.

So all in all, for recursive functions that are declared globally, i.e., for recursive equations, the `Function` vernacular provides fold–unfold lemmas for free. (And by that book it would be great to have a similar `coFunction` vernacular for co-recursive functions.)

7.3 The Equations plugin

Ostensibly Sozeau & Mangin (2019), the EQUATIONS plugin "provides a syntax for defining programs by dependent pattern-matching and structural or well-founded recursion." Concretely, and because to a person with a hammer, the world looks like a nail, this syntax is that of block-structured, scope-sensitive recursive functions which are then λ lifted Johnsson (1985) into toplevel fold–unfold equations, a practical progress over the functional-induction plugin. Conceptually, and to the author's eye, Sozeau and Mangin made the heretofore unidentified connection between functional elimination and λ lifting in their EQUATIONS plugin. And so practically, the extra parameters in the resulting fold–unfold equations can also make sense, intuitively, in terms of λ lifting.

7.4 Interfacing with preexisting recursive functions

When using library functions, one cannot assume that they were declared with `Function` or with `Equation`, the vernacular provided by the EQUATIONS plugin. To reason about them uniformly, one can then state their associated fold–unfold lemmas at the outset, by enumerating the possible configurations of its arguments, as described in Section 7.1.

8 Reasoning about block-structured programs with lexical scope

There is more to functional programming than recursive equations, witness the implementations of `nat_fold_left` and `nat_fold_right` in Section 6.2 that use block structure (`loop` and `visit` are declared locally) and lexical scope (`V`, `z`, and `s` are global to `loop` and `visit`). It feels like pure luck that the fold–unfold tactic proves their fold–unfold lemmas.

Consider the following implementation for computing self-convolutions using the TABA (There and Back Again Danvy & Goldberg (2005)) recursion pattern, where the list to convolve is traversed both through the calls to an auxiliary function `visit` and through the subsequent returns:

```
Definition convolve_thyself (V : Type) (vs : list V) : option (list (V * V)) :=
  let fix visit (vs_sfx : list V) : option (list V * list (V * V)) :=
      match vs_sfx with
      | nil          => Some (vs, nil)
      | v :: vs_sfx' => match visit vs_sfx' with
                        | Some (ws, ps) => match ws with
                                           | nil      => None (* impossible case *)
                                           | w :: ws' => Some (ws', (v, w) :: ps)
                                           end
                        | None          => None
```

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

```
                    end
    end
  in match visit vs with
     | Some (ws, ps) => match ws with
                        | nil      => Some ps
                        | w :: ws' => None (* impossible case *)
                        end
     | None          => None
     end.
```

In words: during its successive recursive calls, `visit` traverses the given list, until it reaches the empty list, i.e., the base case. It then returns a pair holding the given list and an empty list of pairs. During the subsequent returns, the given list is traversed again and the convolution is constructed. The option type is only there to appease the type inferencer.

How does one proceed to prove, e.g., the soundness of this implementation?

```
Theorem soundness_and_totality_of_convolve_thyself :
  forall (V : Type) (vs : list V) (ops : option (list (V * V))),
    convolve_thyself V vs = ops ->
    exists ps : list (V * V),
      ops = Some ps /\ map fst ps = vs /\ map snd ps = rev vs.
```

In words: if applying `convolve_thyself` to a given list yields an optional result, then this optional result was constructed with `Some` and the argument of `Some` is a list of pairs that can be unzipped into the given list and its reverse, which is the definition of a symbolic self-convolution. This lemma establishes that `convolve_thyself` is total since it always returns a result constructed with `Some`, and that it is sound in that it does yield a self-convolution of the given list.

To prove this soundness, one option is (1) to λ lift this implementation into two recursive equations and declare the new auxiliary function with `Function`, since it provides the two associated fold–unfold lemmas for free:

```
Function visit_lifted (V : Type) (vs_sfx vs : list V)
                                              : option (list V * list (V * V)) :=
  match vs_sfx with
  | nil          => Some (vs, nil)
  | v :: vs_sfx' => match visit_lifted V vs_sfx' vs with
                    | Some (ws, ps) => match ws with
                                       | nil      => None (* impossible case *)
                                       | w :: ws' => Some (ws', (v, w) :: ps)
                                       end
                    | None          => None
                    end
  end.

Definition convolve_thyself_lifted (V : Type) (vs : list V)
                                                    : option (list (V * V)) :=
  match visit_lifted V vs vs with
  | Some (ws, ps) => match ws with
                     | nil      => Some ps
                     | w :: ws' => None (* impossible case *)
                     end
  | None          => None
  end.
```

and (2) to proceed with the following auxiliary lemma that characterizes both the control flow and the data flow of TABA:

```
Lemma about_visit_lifted :
  forall (V : Type) (vs_sfx ws_pfx : list V),
    length vs_sfx = length ws_pfx ->
    forall ws_sfx : list V,
      exists ps : list (V * V),
        visit_lifted V vs_sfx (ws_pfx ++ ws_sfx) = Some (ws_sfx, ps) /\
        map fst ps = vs_sfx /\ map snd ps = rev ws_pfx.
```

The reader is referred to Section Section 2 of "Getting There and Back Again" Danvy (2022) for more detail, but in essence, the third argument of `visit_lifted`, `vs`, is the given

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

list and its second argument, `vs_sfx`, is the list traversed by the calls so far. So `vs_sfx` is a suffix of `vs`. In this lemma, this given list is also expressed as the concatenation of a prefix, `ws_pfx`, and a suffix, `ws_sfx`, which are such that the length of this prefix is the same as the length of the list traversed so far, `vs_sfx`. The lemma captures how the given list is traversed at call time and at return time: the lengths of `vs_sfx` and of `ws_pfx` are the number of remaining calls to traverse `vs_sfx`. By the very nature of structural recursion, this number is also the number of returns that yield `Some (ws_sfx, ps)`. Therefore, the returns have traversed the current prefix of the given list, `ws_pfx`, and the returned list is its current suffix, `ws_sfx`. The lemma also captures that the returned list of pairs is a symbolic convolution of the list that remains to be traversed at call time, namely `vs_sfx`, and of the list that has been traversed at return time, namely `ws_pfx`. The control-flow aspect of the lemma expresses that `ws_pfx` has been traversed by the returns and that `vs_sfx` remains to be traversed by the calls. The data-flow aspect of the lemma expresses that the returned list of pairs is a symbolic convolution of these two lists. The lemma is proved by induction on `vs_sfx`, and the theorem follows as a corollary.

Is λ lifting the only option? Can one only reason about λ-lifted programs, i.e., recursive equations, with the Coq Proof Assistant? Well, no. One can reason about λ-dropped programs Danvy & Schultz (2000), i.e., programs with block structure and lexical scope, with "λ-dropped proofs," i.e., proofs with block structure and local names. Here, the λ-dropped proof starts by naming the recursive function as `visit` using the `remember` tactic. One then declares local lemmas using `assert`, starting with fold–unfold ones and continuing with an analogue of the one above, as illustrated in the accompanying .v file. The point here is that local recursive declarations, block structure, and lexical scope make sense for writing programs, and they make just as much sense for writing proofs, the Curry–Howard correspondence notwithstanding. And whereas the `Function` vernacular can only be used for recursive equations, the Equations plugin can be used for the worker/wrapper pattern.

To drum this drum some more, what if one wants to make do without the intermediate option type in the auxiliary function to self-convolve a list? After all, this computation is a showcase for continuations:

```
Definition convolve_thyself_c (V : Type) (vs : list V) : option (list (V * V)) :=
  let fix visit vs_sfx k :=
      match vs_sfx with
      | nil          => k vs nil
      | v :: vs_sfx' => visit vs_sfx' (fun ws ps =>
                                        match ws with
                                        | nil      => None (* impossible case *)
                                        | w :: ws' => k ws' ((v, w) :: ps)
                                        end)
      end

  in visit vs (fun ws ps => match ws with
                            | nil      => Some ps
                            | w :: ws' => None (* impossible case *)
                            end).
```

In words: instead of returning an optional pair of values, `visit` is called with a continuation to which to send these two values. In the base case, the current continuation is sent the given list and an empty list of pairs. In the induction step, `visit` traverses the rest of the list tail-recursively with a new continuation. This new continuation, given a suffix of the given list and a list of pairs, will send the tail of this suffix and an extended list of pairs to the current continuation. The initial continuation is sent the empty list and the self-convolution: it discards one and returns the other.

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

Now for proving the soundness and the completeness of this continuation-based implementation, there is no need to massage it first (here: with λ lifting) to make it fit any particular plugin, whether explicitly with the functional-induction plugin or implicitly with the Equations plugin. One can write a λ-dropped proof similar to the one above. This is of course not a criticism of these plugins, on the contrary: once a beginner is then introduced to them, their reaction is less likely to be "Whatever. Will this be at the exam?" – it is more likely to be one of informed appreciation. Furthermore, no new concept is needed to write a λ-dropped proof, which makes one feel secure in one's knowledge, seeing that it is actionable as it is, a welcome relief to the perpetual (and doubt-inducing) need for new tools to solve new problems.

Let us conclude this section with a simple proof of the simple fact that the lambda-lifted and the lambda-dropped versions of `nat_fold_right` are functionally equivalent (the accompanying .v file also contains a similar proof for `nat_fold_left`):

```
Definition nat_fold_right_dropped (V : Type) (z : V) (s : V -> V) (n : nat) : V :=
  let fix visit n :=
    match n with
    | O    => z
    | S n' => s (visit n')
    end
  in visit n.

Fixpoint nat_fold_right_lifted (V : Type) (z : V) (s : V -> V) (n : nat) : V :=
  match n with
  | O    => z
  | S n' => s (nat_fold_right_lifted V z s n')
  end.
```

This proof is a routine mathematical induction that features both the global fold–unfold lemmas associated to `nat_fold_right_lifted` and the local fold–unfold lemmas associated to `nat_fold_right_dropped`:

```
Lemma fold_unfold_nat_fold_right_lifted_O :
  forall (V : Type) (z : V) (s : V -> V),
    nat_fold_right_lifted V z s O = z.
Proof.
  fold_unfold_tactic nat_fold_right_lifted.
Qed.

Lemma fold_unfold_nat_fold_right_lifted_S :
  forall (V : Type) (z : V) (s : V -> V) (n' : nat),
    nat_fold_right_lifted V z s (S n') = s (nat_fold_right_lifted V z s n').
Proof.
  fold_unfold_tactic nat_fold_right_lifted.
Qed.

Lemma functional_equivalence_of_nat_fold_right_lifted_and_nat_fold_right_dropped :
  forall (V : Type) (z : V) (s : V -> V) (m : nat),
    nat_fold_right_lifted V z s m = nat_fold_right_dropped V z s m.
```

The proof starts with the local declaration of `visit` as denoting the recursive function that is local to `nat_fold_right_dropped` in an assumption that is named `D_visit`:

```
Proof.
  intros V z s m.
  unfold nat_fold_right_dropped.
  remember (fix visit (n : nat) : V := match n with
                                       | O => z
                                       | S n' => s (visit n')
                                       end) as visit eqn:D_visit.
```

Witness the `*goals*` buffer, this named assumption is in the scope of `V`, `z`, and `s`, just like `visit` in the definition of `nat_fold_right_dropped`:

```
  V : Type
  z : V
```

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

```
  s : V -> V
  m : nat
  visit : nat -> V
  D_visit : visit = (fix visit (n : nat) : V := match n with
                                                | O => z
                                                | S n' => s (visit n')
                                                end)
  ============================
  nat_fold_right_lifted V z s m = visit m
```

We then state and prove two local fold–unfold lemmas for `visit`:

```
  assert (fold_unfold_visit_O :
            visit O = z).
  { rewrite -> D_visit.
    reflexivity. }
  assert (fold_unfold_visit_S :
            forall n' : nat,
              visit (S n') = s (visit n')).
  { rewrite -> D_visit.
    reflexivity. }
```

We are then in position to conduct a routine induction proof where the global and the local fold–unfold lemmas are used in synchrony (the local ones with fewer arguments):

```
  induction m as [ | m' IHm'].
  { (* goal:  nat_fold_right_lifted V z s O = visit O *)
    rewrite -> (fold_unfold_nat_fold_right_lifted_O V z s).
    rewrite -> fold_unfold_visit_O.
    reflexivity. }
  { (* IHm' : nat_fold_right_lifted V z s m' = visit m' *)
    (* goal: nat_fold_right_lifted V z s (S m') = visit (S m') *)
    rewrite -> (fold_unfold_nat_fold_right_lifted_S V z s m').
    rewrite -> (fold_unfold_visit_S m').
    rewrite -> IHm'.
    reflexivity. }
Qed.
```

The very facts that this induction proof is routine and that the fold–unfold lemmas are used in synchrony consolidate the newcomer's grasp of block structure and lexical scope.

9 Reflections

The present work – harnessing the Coq Proof Assistant's fold and the unfold tactics into fold–unfold lemmas to mechanize equational reasoning about functional programs – takes place in a wider context: other ways to reason equationally in Coq and other proof assistants. As other proof assistants have their own paradigms – and taking due note that the reviewers unanimously consider fold–unfold lemmas to be trivial in Agda, Isabelle, and Haskell – we only consider Coq. Pretty universally, all the other approaches the author is aware of use the `simpl` tactic, which performs simplifications that encompass fold–unfold steps. Very often, these simplifications save precious time, though at the cost of explainability: it is not always clear how some expressions were simplified and why some others were not. Also, simplifications can overshoot, which hides opportunities for shared lemmas. Take the following lemma, for example, which comes handy when reasoning about the parity predicates:

```
Lemma twice_S :
  forall n : nat,
    S (S (2 * n)) = 2 * S n.
```

This lemma suffuses a problem solver with boredom (to quote Rota (1996)), but a theorizer is happy to see an undergraduate student displaying awareness and potential by

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

stating a helper lemma `twice : forall x : nat, x + x = 2 * x`, and using it in tandem with `plus_Sn_m` and `plus_n_Sm` to prove `about_twice`:

```
Proof.
  intro n.
  (* goal: S (S (2 * n)) = 2 * S n *)
  rewrite <- (twice n).
  (* goal: S (S (n + n)) = 2 * S n *)
  rewrite -> (plus_n_Sm n n).
  (* goal: S (n + S n) = 2 * S n *)
  rewrite <- (plus_Sn_m n (S n)).
  (* goal: S n + S n = 2 * S n *)
  exact (twice (S n)).
Qed.
```

At any rate, the `simpl` tactic is generally seen as a time saver for experienced users.

Debussy once retorted that of course, he knew the rules of the fugue, which is why he could break them. The situation appears to be similar here: shortcuts can be used fruitfully when one knows what they cut short. Unfortunately, using shortcuts, it is perfectly possible to learn the Coq Proof Assistant with very little understanding and then use it as one would play whack-a-mole. The proof of Lemma `completeness_of_mirror'_cps` in Section 6.1, for example, can proceed by first simplifying and then by mechanically writing the tactic that fits the current goal (and never mind the semiotics of `H` nor why one needs to conclude the proof by invoking `reflexivity` three times, Lewis–Carroll style):

```
  induction t; simpl; intros.
  - rewrite H.
    reflexivity.
  - (* rewrite IHt1. (* Error: Unable to find an instance for the variable t_m. *) *)
    rewrite (IHt1 (mirror V t1)). (* <- guess *)
    rewrite (IHt2 (mirror V t2)). (* <- educated guess (learning by doing, yay) *)
    rewrite H.
    reflexivity.
    reflexivity.
    reflexivity.
Qed.
```

This gaming strategy may work when one uses the Coq Proof Assistant in anger, but it is not sustainable because one learns nothing from what one does mindlessly: exercises and exams are not referentially transparent, nor are they an end in themselves – they are a means to acquire an actionable understanding.

Now what to do when a facetious undergraduate student swaps the joy of figuring things out for the glee of not figuring things out and getting away with it, because, hey, Coq accepted their proof? Rather than fighting this wave, we can surf on it and introduce Paul Erdös and his notion of a proof being from The Book. Since some proofs are from The Book, others are not. Extra credit can be given for proofs that are not from The Book if they come with an explanation or at least with an analysis, something students take seriously if 50% and 50% about the quality of one's understanding being reflected by the clarity of one's narrative (1815).

10 Summary and conclusion

Keep calm and write fold–unfold lemmas.

Fold–unfold lemmas complement the Coq Proof Assistant's `rewrite` tactic to reason about recursive programs equationally. Their simplicity is deceiving. For beginners, not being able to state them is a tell-tale sign. For rookies, not being able to explain them is

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

another tell-tale sign. Fold–unfold lemmas often feel beneath more advanced users, and it is impressive how often these more advanced users subsequently feel that their proof is above them. The reason for this feeling, however, is merely quantitative: fold–unfold lemmas offer expressive power to reason about calls to recursive functions, and it is more efficient to harness this expressive power at the outset, when declaring a recursive function locally or globally, than to re-invent the wheel each time a recursive function is called. As such, fold–unfold lemmas – be them stated implicitly using a plugin or stated explicitly by hand – provide a reliable foundation for reasoning about recursive programs using the Coq Proof Assistant.

Acknowledgments

The author is grateful to Julia Lawall for preliminary comments and for pointing to the functional-induction plugin in the Coq Proof Assistant and its undocumented feature. Thanks are also due to the anonymous reviewers for perceptive comments, which very much include pointing to the Equations plugin (Sozeau & Mangin, 2019), and to Matthieu Sozeau for a subsequent email exchange in the fall of 2021. And of course pure and total thanks go to the Archimedean designers, developers, and maintainers of the Coq Proof Assistant and of Proof General for offering such a fulcrum to lift the world of computing. This research received no specific grant from any funding agency, commercial, or not-for-profit sectors.

Conflicts of interest

The author reports no conflict of interest.

Supplementary material

For supplementary material for this article, please visit https://doi.org/10.1017/S0956796822000107.

References

Bertot, Y. (2006) *Coq in a hurry*. CoRR. Available at: http://arxiv.org/abs/cs/0603118v2.
Bertot, Y. & Castéran, P. (2004) *Interactive Theorem Proving and Program Development*. Springer.
Bird, R. & Wadler, P. (1988) *Introduction to Functional Programming*. Prentice-Hall International.
Boileau, N. (1815) *L'art poétique*. Aug. Delalain.
Burge, W. H. (1975) *Recursive Programming Techniques*. Addison-Wesley.
Burstall, R. M. & Darlington, J. (1977) A transformational system for developing recursive programs. *J. ACM* **24**(1), 44–67.
Burstall, R. M. & Landin, P. J. (1969) Programs and their proofs: An algebraic approach. In *Machine Intelligence*, Meltzer, B. & Michie, D. (eds), vol. 4. Edinburgh University Press, pp. 17–43.
Chlipala, A. (2013) *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press.
Danvy, O. (2011–2021). *Functional Programming and Proving, de l'âne au coq*. Lecture Notes.
Danvy, O. (2019a) Folding left and right over Peano numbers. *J. Funct. Program.* **29**(e6).

Danvy, O. (2019b) Mystery functions: Making specifications, unit tests, and implementations coexist in the mind of undergraduate students. In *IFL'19: Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, Stutterheim, J. & Chin, W. N. (eds), Singapore: ACM Press, pp. 1–9. Article No. 2.

Danvy, O. (2022) Getting there and back again. *Fundamenta Informaticae* **185**(2), 115–183.

Danvy, O. & Goldberg, M. (2005) There and back again. *Fundamenta Informaticae* **66**(4), 397–413. A preliminary version was presented at the 2002 ACM SIGPLAN International Conference on Functional Programming (ICFP 2002).

Danvy, O. & Schultz, U. P. (2000) Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoret. Comput. Sci.* **248**(1–2), 243–287.

Gowers, T. (2000) The two cultures of mathematics. In *Mathematics: Frontiers and Perspectives*. AMS.

Grégoire, B. & Leroy, X. (2002) A compiled implementation of strong reduction. In *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, Peyton Jones, S. (ed). SIGPLAN Notices, vol. 37, No. 9. Pittsburgh, Pennsylvania: ACM Press, pp. 235–246.

Johnsson, T. (1985) Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, Jouannaud, J.-P. (ed). Lecture Notes in Computer Science, vol. 201. Nancy, France: Springer-Verlag, pp. 190–203.

Jones, N. D., Gomard, C. K. & Sestoft, P. (1993) *Partial Evaluation and Automatic Program Generation*. London, UK: Prentice-Hall International.

Manna, Z. (1974) *Mathematical Theory of Computation*. McGraw-Hill.

Nipkow, T., Paulson, L. C. & Wenzel, M. (2002) *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, vol. 2283. Springer.

Pierce, B. C., Casinghino, C., Greenberg, M., Hriţcu, C., Sjöberg, V. & Yorgey, B. (2012) *Software foundations*. Available at: http://www.cis.upenn.edu/~bcpierce/sf/.

Raphaelson, S. (1949) *The Human Nature of Playwriting*. Macmillan.

Rota, G.-C. (1996) *Indiscrete Thoughts*. Birkhaüser.

Sozeau, M. & Mangin, C. (2019) Equations reloaded: High-level dependently-typed functional programming and proving in Coq. In *ICFP*, Dreyer, D. & Pottier, F. (eds). Proceedings of the ACM on Programming Languages, vol. 3. Berlin, Germany: ACM Press, pp. 1–29. Article No. 86.

Thompson, S. (1995) *Miranda: The Craft of Functional Programming*, 1st ed. International Computer Science Series. Addison-Wesley.

Thompson, S. (2011) *Haskell: The Craft of Functional Programming*, 3rd ed. International Computer Science Series. Addison-Wesley Professional.

Appendix

A Coq in no hurry

This appendix provides a minimalistic introduction to the Coq Proof Assistant, in the manner of Bertot (2006). This proof assistant, like Agda, is based on the proposition-as-types paradigm, but unlike Agda, it has a separate tactics language. This tactics language has the imperative flavor of algebraic proofs in that the goal is incrementally modified until the completion of each proof. A proof follows the statement of a theorem/lemma/proposition/etc. Proofs are prefixed by `Proof.` and suffixed with `Qed.`, unless, e.g., they are admitted.

A.1 Libraries

The Coq Proof Assistant comes with many libraries, e.g, the one for arithmetics over natural numbers, that have type `nat` and are represented as Peano numerals, with `O` and `S` as constructors. This library is imported as follows:

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

```
Require Import Arith.
```

Given the name of a lemma in this library, the `Check` command tells us its type. For example, `Arith` features lemmas that correspond to the inductive definition of addition (base case and induction step) as well as its associativity:

```
Check Nat.add_0_l. (* : forall m : nat, 0 + m = m *)

Check Nat.add_succ_l. (* forall n m : nat, S n + m = S (n + m) *)

Check Nat.add_assoc. (* : forall n m p : nat, n + (m + p) = n + m + p *)
```

Likewise, `Arith` features lemmas that correspond to the inductive definition of multiplication over the multiplier as well as its distribution over addition:

```
Check Nat.mul_0_l. (* : forall m : nat, 0 * m = 0 *)

Check Nat.mul_succ_l. (* : forall n m : nat, S n * m = n * m + m *)

Check Nat.mul_add_distr_r. (* : forall n m p : nat, (n + m) * p = n * p + m * p *)

Check Nat.mul_add_distr_l. (* : forall n m p : nat, n * (m + p) = n * m + n * p *)
```

A.2 Applying lemmas

Lemmas that name universally quantified formulas are instantiated by applying their name to arguments:

```
Check (Nat.mul_add_distr_l 2). (* : forall m p : nat, 2 * (m + p) = 2 * m + 2 * p *)

Check (Nat.mul_add_distr_l 2 3). (* : forall p : nat, 2 * (3 + p) = 2 * 3 + 2 * p *)

Check (Nat.mul_add_distr_l 2 3 4). (* : 2 * (3 + 4) = 2 * 3 + 2 * 4 *)
```

Natural numbers are parsed into Peano numerals:

```
Check (Nat.add_succ_l 3). (* : forall m : nat, 4 + m = S (3 + m) *)
```

A.3 Definitions

To align the lemmas that correspond to the inductive definition of addition and multiplication with the corresponding fold–unfold lemmas, we can rename them:

```
Definition fold_unfold_add_O : forall m : nat, 0 + m = m :=
  Nat.add_0_l.

Definition fold_unfold_add_S : forall n m : nat, S n + m = S (n + m) :=
  Nat.add_succ_l.

Definition fold_unfold_mul_O : forall m : nat, 0 * m = 0 :=
  Nat.mul_0_l.

Definition fold_unfold_mul_S : forall n m : nat, S n * m = n * m + m :=
  Nat.mul_succ_l.
```

As a functional language, Gallina makes it possible to declare functions. The square function, for example, is written `fun (n : nat) => n * n` and can be named using the traditional syntactic sugar of functional programs:

```
Definition square (n : nat) : nat :=
  n * n.
```

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

A.4 Structure of a proof

A proof is stated as a series of updates over the current goal, using tactics. Some of these tactics might create subgoals (e.g., for proofs by cases or proofs by induction). Then we focus on the current subgoal using itemization signs or using curly braces.

A.5 The tactics used in the present tutorial

To prove a universally quantified statement $\forall x.s$, one typically picks an x and proceeds with s. This picking is achieved using the `intro` tactic. If the type of x is composite (e.g., a product or a sum), one uses the `intros` tactic with syntactic sugar to name the components.

As described at the end of Section 1, the `fold` tactic replaces a name by its denotation and the `unfold` tactic replaces a denotation by its name.

As described at the beginning of Section 2, given an equality, the `rewrite` tactic replaces either side by the other.

Given an argument whose type is inductive, the `induction` tactic is used to conduct a structural induction proof at that type. So if this type is `nat`, this proof corresponds to mathematical induction since natural numbers are represented as Peano numerals.

The `reflexivity` tactic is used to prove the current goal when this goal is an equality with the same left-hand side and right-hand side.

Given an argument that coincides with the current goal, the `exact` tactic is used to prove this goal.

The `assert` and `remember` tactics are used to declare new assumptions.

Using a semicolon (`;`) instead of a period (`.`) between two proof steps conflates them into one. And if the first proof step creates subgoals, the second proof step takes place for each of these subgoals.

A.6 Three simple lemmas and their proof

We are now in position to mechanize algebraic proofs.

Proving that 1 is neutral on the left of multiplication is carried out with three successive fold–unfold steps:

```
Lemma mul_1_l :
  forall n : nat,
    1 * n = n.
Proof.
  intro n.
  (* goal: 1 * n = n *)
  rewrite -> (fold_unfold_mul_S 0 n).
  (* goal: 0 * n + n = n *)
  rewrite -> (fold_unfold_mul_0 n).
  (* goal: 0 + n = n *)
  rewrite -> (fold_unfold_add_0 n).
  (* goal: n = n *)
  reflexivity.
Qed.
```

Proving Lemma `twice` from Section 9 is carried out with one fold–unfold step and using Lemma `mul_1_l`:

```
Lemma twice :
  forall n : nat,
    n + n = 2 * n.
Proof.
  intro n.
  (* goal: n + n = 2 * n *)
  rewrite -> (fold_unfold_mul_S 1 n).
```

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press

```
  (* goal: n + n = 1 * n + n *)
  rewrite -> (mul_1_l n).
  (* goal: n + n = n + n *)
  reflexivity.
Qed.
```

Likewise, and as illustrated in the accompanying .v file, proving the binary expansion at rank 2 is carried out from left to right by unfolding the left call to `square`, distributing multiplication over addition 3 times, reassociating addition twice, commuting a multiplication, applying Lemma `twice`, and folding two calls to `square`. (The accompanying .v file also contain a proof for binary expansion at rank 2 that is carried out from right to left.)

A.7 Addition is commutative

Since addition is defined recursively over its first argument, its commutativity is proved by nested induction, systematically using its fold–unfold lemmas. The outer induction is over the first argument on the left-hand side and the inner induction over the first argument on the right-hand side:

```
Lemma add_comm :
  forall i j : nat,
    i + j = j + i.
Proof.
  intro i.
  (* goal: forall j : nat, i + j = j + i *)
  induction i as [ | i' IHi']. (* outer induction *)
  { (* goal: forall j : nat, 0 + j = j + 0 *)
    intro j.
    (* goal: 0 + j = j + 0 *)
    rewrite -> (fold_unfold_add_0 j).
    (* goal: j = j + 0 *)
    induction j as [ | j' IHj']. (* inner induction in the outer base case *)
    { (* goal: 0 = 0 + 0 *)
      rewrite -> (fold_unfold_add_0 0).
      (* goal: 0 = 0 *)
      reflexivity. }
    { (* IHj' : j' = j' + 0 *)
      (* goal: S j' = S j' + 0 *)
      rewrite -> (fold_unfold_add_S j' 0).
      (* goal: S j' = S (j' + 0) *)
      rewrite <- IHj'.
      (* goal: S j' = S j' *)
      reflexivity. } }
  { (* IHi' : forall j : nat, i' + j = j + i' *)
    (* goal: forall j : nat, S i' + j = j + S i' *)
    intro j.
    (* goal: S i' + j = j + S i' *)
    rewrite -> (fold_unfold_add_S i' j).
    (* goal: S (i' + j) = j + S i' *)
    induction j as [ | j' IHj']. (* inner induction in the outer induction step *)
    { (* goal: S (i' + 0) = 0 + S i' *)
      rewrite -> (IHi' 0).
      (* goal: S (0 + i') = 0 + S i' *)
      rewrite -> (fold_unfold_add_0 i').
      (* goal: S i' = 0 + S i' *)
      rewrite -> (fold_unfold_add_0 (S i')).
      (* goal: S i' = S i' *)
      reflexivity. }
    { (* IHj' : S (i' + j') = j' + S i' *)
      (* goal: S (i' + S j') = S j' + S i' *)
      rewrite -> (IHi' (S j')).
      (* goal: S (S j' + i') = S j' + S i' *)
      rewrite -> (fold_unfold_add_S j' i').
      (* goal: S (S (j' + i')) = S j' + S i' *)
      rewrite -> (fold_unfold_add_S j' (S i')).
      (* goal: S (S (j' + i')) = S (j' + S i') *)
      rewrite <- IHj'.
      (* goal: S (S (j' + i')) = S (S (i' + j')) *)
      rewrite -> (IHi' j').
      (* goal: S (S (j' + i')) = S (S (j' + i')) *)
      reflexivity. } }
Qed.
```

https://doi.org/10.1017/S0956796822000107 Published online by Cambridge University Press