Computing Correctly with Inductive Relations

Zoe Paraskevopoulou
z.paraskevopoulou@northeastern.edu
Northeastern University
USA

Aaron Eline
aeline@umd.edu
University of Maryland, College Park
USA

Leonidas Lampropoulos
leonidas@umd.edu
University of Maryland, College Park
USA

Abstract

Inductive relations are the predominant way of writing specifications in mechanized proof developments. Compared to purely functional specifications, they enjoy increased expressive power and facilitate more compositional reasoning. However, inductive relations also come with a significant drawback: they can't be used for computation.

In this paper, we present a unifying framework for extracting three different kinds of computational content from inductively defined relations: semi-decision procedures, enumerators, and random generators. We show how three different instantiations of the same algorithm can be used to generate all three classes of computational definitions inside the logic of the Coq proof assistant. For each derived computation, we also derive *mechanized proofs* that it is sound and complete with respect to the original inductive relation, using Ltac2, Coq's new metaprogramming facility.

We implement our framework on top of the QuickChick testing tool for Coq, and demonstrate that it covers most cases of interest by extracting computations for the inductive relations found in the Software Foundations series. Finally, we evaluate the practicality and the efficiency of our approach with small case studies in randomized property-based testing and proof by computational reflection.

Keywords: inductive relations, partial decidability, enumeration, random generation, proof assistants, Coq, QuickChick

1 Introduction

In interactive proof assistants like Coq [10] and Agda [27], specifications are formalized either as inductively defined relations or as pure functions, each with their own benefits and limitations. Inductive relations allow for a more liberal style of writing specifications as they can naturally express a variety of features that functions in a total language cannot capture, such as nontermination and nondeterminism. Moreover, they facilitate compositional reasoning as their elimination and inversion principles enable proofs by induction and case analysis.

However, these benefits don't come without a price. Unlike functions, inductive relations are not executable, even though that would be highly desirable in various situations, from leveraging computation in proofs using reflection [4] to establishing confidence in the correctness of developments using QuickCheck-style random testing [9]. Indeed, it is not uncommon for proof assistant users to write hand-written functional variants of inductive relations and then prove their correspondence to ensure that the two artifacts are—and remain—in sync.

In this work, we focus on the problem of extracting computational content from inductive relations, as well as mechanically verifying the correctness of derived code post-hoc, in the style of translation validation [32]. Prior work has focused on extracting computations for specific aspects of inductive relations with various degrees of correctness guarantees. For example, Tollitte et al. [34] extract total functions from relations that either check the validity of a fully applied relation or generate an output that satisfies the relation, given the rest of its arguments as inputs; Bulwahn [6] enumerate inhabitants of a given predicate rather than just a unique value that satisfies it; Lampropoulos et al. [24] extract random generators for data that satisfy inductively defined relations. However, as we will see, these three computational aspects are *interdependent*: enumerators and generators that produce data often need to check whether an inductive relation holds on a given set of inputs, while a procedure that checks whether an inductive relation holds might need to produce (all possible) values that satisfy a premise in order to handle existentially quantified variables (i.e., ones that are not bound in the result of a constructor).

In this paper, we present a unifying framework that captures all of these aspects, deriving both *checkers*, i.e., semi-decision procedures that determine whether an inductive relation holds for a given set of arguments, and *producers*, i.e., functions that produce data that satisfy an inductive relation either deterministically (enumerators) or randomly (generators). We target inductive relations P of the form:

$$\begin{array}{l}\texttt{Inductive}\ P\ (A_1\ \ldots : Type) : T_1 \to \cdots \to \texttt{Prop} := \\ \mid C_1\ : \forall x_1 \ldots, (Q_1\ e_{11}\ \ldots) \to \ldots \to P\ e_1\ \ldots\ e_n \quad \mid \cdots\end{array}$$

where each T_i is a first-order type and each Q_i is an inductive relation of the same class (or its negation). Crucially, for each generated computation, we automatically derive a *machine checked* proof of correctness that states that the computation is *sound* and *complete* with respect to P. Our unified approach gives us the ability to handle the interdependencies between checkers and producers, allowing us to extract computations for a much broader and practical class of relations than previous work.

PLDI '22, June 13–17, 2022, San Diego, CA, USA
2022. ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523707

We implement this framework on top of the QuickChick property-based testing tool for Coq [25]. Our unified derivation procedure is written in OCaml as a Coq plugin; our proof derivation procedures are written in Ltac2 [29], Coq's new metaprogramming language. That is, we developed generic proof scripts that produce correctness proofs for our derived procedures, in what is the first large-scale application of Ltac2 metaprogramming capabilities. Finally, we explore its applications and we demonstrate its capabilities through a variety of case studies. Concretely, our contributions are:

- We present an algorithm for deriving semi-decision procedures for a large class of inductive relations, but leaving out existentially quantified variables (Section 3).
- We generalize this algorithm to also derive *producers*, a novel abstraction that unifies enumerators and random generators. By exploiting the interdependencies between checkers and producers, this algorithm can handle *all* inductive relations (Section 4).
- We use a translation validation scheme to prove that each derived checker or producer is *correct*—sound and complete—with respect to the inductive relation it was derived from. We present a proof sketch of this scheme that also serves as an informal proof that our algorithm works on every inductive relation (Section 5).
- We implement all of these on top of QuickChick (publicly available here), leveraging its extensive generic programming facilities, and evaluate our work on three axes (Section 6): (1) we show that we can handle a wide variety of interesting inductive relations in practice, by targeting those in Software Foundations; (2) we evaluate the efficiency of derived procedures for *testing* by comparing against handwritten ones in QuickChick benchmarks with minimal slowdown (less than 2%); and (3) we demonstrate how the usefulness of derived checkers for *proving* with a small case study in proof by reflection.

We begin with a concrete example of a checker for the simply typed lambda calculus, that will serve as the starting point for the introduction of our derivation algorithm (Section 2), discuss limitations in Section 8, related work in Section 7, and conclude by proposing future work in Section 9.

2 Example: STLC

Throughout this paper, we will use as our running example the typing relation for terms in the simply typed lambda calculus (STLC) calculus: a relation ubiquitous amongst programming languages literature, but also one that is out of reach of prior approaches that attempt to extract computational content from inductive relations.

Following standard practice, we represent types and terms of STLC as inductive datatypes: `types` (natural numbers `N` or arrows `Arr`); and `terms` (constant natural numbers `Con`, additions `Add`, lambda abstractions in DeBruijn form `Abs`, variables `Var`, or applications `App`).

```
Inductive type : Type :=
| N : type
| Arr : type -> type -> type.

Inductive term : Type :=
| Con : nat -> term
| Add : term -> term -> term
| Var : nat -> term
| App : term -> term -> term
| Abs : type -> term -> term.
```

We then define an inductive relation that describes the STLC typing rules, characterizing when a `term` has a specific `type` in an environment (a list of `types`), where `lookup` is an auxiliary inductive relation for indexing into the environment (elided for brevity).

```
Inductive typing Γ : term -> type -> Prop :=
| TCon : forall n, typing Γ (Con n) N
| TAdd : forall e1 e2,
    typing Γ e1 N -> typing Γ e2 N ->
    typing Γ (Add e1 e2) N
| TAbs : forall e t1 t2,
    typing (t1 :: Γ) e t2 ->
    typing Γ (Abs t1 e) (Arr t1 t2)
| TVar : forall x t,
    lookup Γ x t -> typing Γ (Var x) t
| TApp : forall e1 e2 t1 t2,
    typing Γ e2 t1 -> typing Γ e1 (Arr t1 t2) ->
    typing Γ (App e1 e2) t2.
```

If we ignore the application case (we will return to it later in the paper), deciding whether a given term `e` has a given type `t` in an environment `Γ` is straightforward:

```
Fixpoint typing_dec Γ e t : bool :=
  match e, t with
  | Con n, N => true
  | Add e1 e2, N =>
    typing_dec Γ e1 N && typing_dec Γ e2 N
  | Abs t1 e, Arr t1' t2 =>
    (t1 = t1')? && typing_dec (t1 :: Γ) e t2
  | Var x, _ => (lookup Γ x t)?
  | _, _ => false
  end.
```

This fixpoint closely mirrors the structure of the `typing` relation. We match on both the term and the type: a constant always has type `N`; an addition has type `N` if both of its operands also do, an abstraction `Abs t1 e` has a function type `Arr t1 t2`, if its body has type `t2` in an extended environment recursively; and a variable has type `t` if that is its binding in the environment–expressed using the `?` notation that invokes a checker for a property using typeclasses [25].

But even ignoring application, how would one arrive at such a definition? The answer comes from the structure of the `typing` inductive definition itself. For `typing Γ e t`

to hold, there *must* be some constructor that was used to construct an inhabitant of that type. Therefore we can look at the conclusion of each constructor to figure out what shape that constructor expects the input arguments to be. These shapes become patterns to match inputs against in the checker; for example, the `TAbs` constructor expects the term to be an `Abs` and the type an `Arr`. Each precondition in a constructor then gives rise to an invocation of either the checker itself (if it is a recursive constraint) or of an external function that is the (possibly automatically derived) checker of this precondition.

This suggests a straightforward algorithm for deciding whether an inductive relation holds automatically: try to see if there is *any* constructor that could be used to produce an inhabitant by matching against the patterns in its conclusion, and checking whether its preconditions hold. We used this approach to produce a *derived checker* for `typing`, shown in Figure 1. This checker is, unsurprisingly, more verbose than the hand-written one, as it has to account for a broad class of things that can go wrong—deciding whether an inductive relation holds in the general case is undecidable after all!

The first thorn in the derived checker's side is nontermination: for Coq to be consistent as a proof assistant, all functions must be total and therefore terminating; however, inductive relations can be used to encode nonterminating computation, and therefore a checker for such a relation can't be a total function. To address this, we include two additional arguments in every checker: a standard `size` parameter as fuel to bound the recursion, and a `top_size` parameter that serves as fuel for calls to other checkers (such as the one for `lookup` in the abstraction case).

But then what should the checker do when it runs out of fuel? Our solution is to change the output type from a boolean to a *boolean option*. In this three-valued type:

1. `Some true` means that the checker could positively conclude that `typing Γ e t` holds;
2. `Some false` means that it could conclude that `typing Γ e t` doesn't hold;
3. `None` signifies that it needs additional fuel to reach a conclusion one way or another.

As a result, to compose together multiple checks for recursive constraints, we need some form of optional conjunction:

```
a .&& b := match a with
           | Some false => Some false
           | None => None
           | Some true => b
```

Finally, the hand-written checker can afford to be smart about merging the handling of different constructors in a single pattern match. On the other hand, the derived checker is more generic: each constructor gives rise to a *handler* comprising of a pattern match and some additional checks, while the pattern match returns something other than `Some false` in only one case—the one that is prescribed by the conclusion

```
Fixpoint rec size top_size Γ e t : option bool :=
  match size with
  | O =>
    backtracking [
      fun tt => match e, t with
                | Con _, N => Some true
                | _, _     => Some false
                end;
      fun tt => None ]
  | S size' =>
    backtracking [
      fun tt => match e, t with
                | Con _, N => Some true
                | _, _     => Some false
                end;
      fun tt => match e, t with
                | Add e1 e2, N =>
                  rec size' top_size Γ e1 N .&&
                  rec size' top_size Γ e2 N
                | _, _ => Some false
                end;
      fun tt => match e, t with
                | Var x, _ =>
                  check top_size (lookup Γ e t)
                | _, _ => Some false
                end;
      fun tt => match e, t with
                | Abs t1 e, Arr t1' t2 =>
                  check top_size (t1 = t1') .&&
                  rec size' top_size (t1 :: Γ) e t2
                | _, _ => Some false
                end ]
  end.
```

Figure 1. Derived checker for `typing` ignoring `App`

of the inductive constructor. Each such handler is then tried via the use of the `backtracking` combinator, which returns `Some true` if *any* of its options does so, `Some false` if *all* of its options do so, and `None` otherwise. All of its options are also thunked, to avoid unnecessary evaluation in a non-lazy setting. Moreover, one of the options in the base case is `None`, to correctly encode running out of fuel.

Discussion. At a high level, `typing_dec` is much cleaner and more optimized compared to the derived checker of Figure 1. However, the design decisions underlying its derivation that were described above make it much easier to implement generically, and more importantly to *reason* about its correctness with respect to the `typing` inductive relation. Still, as we will see later on, the overhead imposed in practice by these derived checkers is minimal, as any backtracking that occurs is extremely localized (Section 6.2). Moreover, the same algorithm can be used to derive proper decision procedures, albeit in a quite user-unfriendly way (Section 8).

3 Deriving Partial Decidability Procedures

The fully general derivation algorithm is quite intricate, so we will gradually build up to it in the rest of the paper. In this section, we will start with a restricted subset of inductive types, describe how to derive partial decidability procedures for it, and then slowly build back up to the complete class.

The Derivation Core. We begin by targeting inductive relations ranging over constructor terms: variables or fully applied constructors whose arguments are constructor terms. Additionally, we require that the result of each constructor does not contain *non-linear patterns*, i.e., each variable occurs exactly once in the conclusion, and that all universally quantified variables are bound in it. The grammar is otherwise the same as the one in the introduction:

$$\texttt{Inductive}\ P\ (A_1\ \ldots : Type) : T_1 \to \cdots \to \texttt{Prop} :=$$
$$\mid C_1\ : \forall x_1 \ldots, (Q_1\ e_{11}\ \ldots) \to \ldots \to P\ e_1\ \ldots\ e_n \quad \mid \cdots$$

We will relax these constraints later, but this core is simple enough to describe the essence of our approach, which is shown in pseudocode as Algorithm 1. We depict the logic of the algorithm with black, show the produced code in red, and allow for anti-quoted escapes with blue.

The first thing the algorithm does (lines 3-6) is to iterate through all constructors ctr of P in order to compute a handler for each one. Each handler c will check whether ctr can be used to produce a proof of P given the arguments provided, and is computed by the CTR_LOOP helper function, explained below. After the loop, the variable Cs holds a handler for each constructors, while Bs holds only the handlers of nonrecursive ones (such as `Con`).

The rest of DERIVE_CHECKER produces the top-level structure of the semi-decision procedure, which is always the same (in red) and is essentially identical to the one depicted in Figure 1. It is a fixpoint that takes as parameters: a `size` that will be used to bound the recursion, a `top_size` that will be used as the fuel parameter for all external calls, and any actual inputs P requires.

If the `size` is nonzero (lines 13-15), the algorithm iterates through all handlers computed earlier and anti-quotes the code of the handler, wrapping all of the handlers in a `backtracking` combinator, just like the checker of Figure 1. If the `size` is zero (lines 9-12), the checker ran out of fuel. In this case, the algorithm will only check if a base (i.e., nonrecursive) constructor can be satisfied. Finally, the algorithm checks if there were recursive handlers that were skipped (line 11) to quote an additional option to return `None` (signifying that the checker requires a larger `size` parameter as input to make a decision).

The final piece of the puzzle is the implementation of CTR_LOOP. Assume (line 20) that the constructor we're trying to check is of the form $\forall\ x_1 \cdots x_n.\ Q_1\ e_{11}\ \cdots\ e_{1n_1} \to \cdots \to Q_m\ e_{m1}\ \cdots\ e_{mn_m} \to P\ e_1\ \cdots\ e_p$. The first thing the algorithm does is to ensure that the inputs to the checker

Algorithm 1. Core Checker Derivation Algorithm

```
1:  function DERIVE_CHECKER(P)
2:      Bs, Cs ← ∅
3:      for each ctr ∈ P do
4:          c ← ctr_loop(ctr)
5:          Cs ← Cs ∪ c
6:          if not (is_rec ctr) then Bs ← Bs ∪ c
7:      let fix rec size top_size in_1 in_2 ... in_p :=
8:      match size with
9:      | 0 => backtracking [
10:     foreach c ∈ Bs do fun tt => c;
11:     if Cs ≠ Bs then fun tt => None
12:     ]
13:     | S size' => backtracking [
14:     foreach c ∈ Cs then fun tt => c;
15:     ]
16:     in fun size in_1 in_2 ... in_p =>
17:     rec size size in_1 in_2 ... in_p
18:
19: function CTR_LOOP(ctr)
20:     (∀ x̄. Q_i e_i1 ··· e_in_i → P e_1 ··· e_p) ← ctr
21:     match in_1, in_2, ..., in_p with
22:     | e_1, e_2, ..., e_p =>
23:     for each Q_i e_i1 ··· e_in_i do
24:         if P = Q_i then rec size' e_i1 ... e_in_i .&&
25:         else check top_size (Q_i e_i1 ... e_in_i) .&&
26:     Some true
27:     | _, _, ..., _ => Some false
```

are of the form described in the constructor's conclusion $P\ e_1\ \cdots\ e_p$. To do so, it produces a pattern match (lines 21-22) that matches the inputs $in_1, \ldots, in_p$ against those constructor terms—since in our restricted setting all terms are either variables or constructors applied to variables they can also serve as patterns. For instance, in the STLC example, the match generated for `TAdd` would be:[1]

```
match in1, in2, in3 with
| Γ, Add e1 e2, N => ...
```

If the pattern match succeeds, the algorithm needs to handle any constraints imposed by the constructor. Each such constraint $Q_i\ e_{i1}\ \cdots\ e_{in_i}$ gives rise to either a recursive call to the checker being defined (if $P = Q_i$, line 24), or to a checker for a different inductive relation (if $P \neq Q_i$, line 25). Going back to the STLC example, the `typing Γ e1 N` constraint of `TAdd` gives rise to a recursive call `rec size' top_size Γ e1 N`, while the `lookup Γ x` constraint of `TVar` results in a call to `check top_size (lookup Γ x t)`.

Successive constraints are chained together using the three-valued conjunction `.&&`. After all constraints have been processed successfully, only then does the checker return

[1] For presentation purposes, we elided the trivial pattern match on the environment in Figure 1.

`Some true` (line 26), completing the code that handles a particular constructor.

On the other hand, if the inputs did not match against the constructor's patterns (line 27), then the constructor handler cannot be used to produce an inhabitant of P and the checker has to return `Some false`.

Extending to Full Inductive Relations

The restricted form of inductive relations targeted is useful enough to capture many interesting inductive datatypes, but *non-linear patterns*, *function calls* in the conclusion of a constructor, and *existentially quantified variables* still need to be addressed. The first two are similar in nature: by leaving them out, we were able to get away with using the expressions in the conclusion of constructors as patterns. We will show how we can add them back by handling these constraints as additional premises. The last feature is trickier: `forall` quantifiers in constructors can be used to encode existential quantifiers, when variables appearing in the premises of a constructor but not its conclusion. We will address this challenge by introducing a way of instantiating such variables: enumeration.

Non-Linear Patterns. Taking a closer look at the abstraction case, we can spot a non-linear pattern:

```
TAbs : forall e t1 t2, typing (t1 :: Γ) e t2 ->
       typing Γ (Abs t1 e) (Arr t1 t2)
```

If we tried to blindly apply Algorithm 1, we would get syntactically invalid code, as Coq's pattern matching does not allow for binding the same variable multiple times:

```
match in1, in2, in3 with
| Γ, Abs t1 e, Arr t1 t2 => ...
```

To get around this, we observe that if we convert the nonlinear pattern to an explicit equality check, we obtain an equivalent formulation:

```
TAbs : forall e t1 t2 t1',
       t1 = t1' -> typing (t1 :: Γ) e t2 ->
       typing Γ (Abs t1 e) (Arr t1' t2)
```

Then, CTR_LOOP produces the following handler:

```
match in1, in2, in3 with
| Γ, Abs t1 e, Arr t1' t2 =>
  check top_size (t1 = t1') .&& ...
```

which is a perfectly valid checker for the original type of `TAbs`! We do this conversion as part of a preprocessing phase before the derivation process.

Function Calls. To see why function calls pose a similar problem, consider the following inductive relation, borrowed from the series Software Foundations [31], that holds when a natural number is the square of another:

```
Inductive square_of : nat -> nat -> Prop :=
| sq : forall n, square_of n (n * n).
```

Once again, if we tried to blindly apply Algorithm 1, we would get syntactically invalid code in the pattern match:

```
match in1, in2 with
| n, (n * n) => ...
```

Just like in the case of non-linear patterns, we can get around this issue by introducing a fresh variable and an equality constraint between it and the function call. Then, for instance, the `sq` constructor would be (automatically) rewritten to:

```
sq : forall n m, n * n = m -> square_of n m,
```

which yields a perfectly functional checker. Notice that function calls are not a problem when they appear in constraints: if a constraint `P (n * n)` is encountered, the checker for `P` with `n * n` as its argument can simply be invoked.

Existentially Quantified Variables. Unlike non-linear patterns and functions calls which posed a problem when handling conclusions of inductive constructors, existentially quantified variables are a problem that arises when handling premises. Recall the typing rule for application:

```
TApp : forall e1 e2 t1 t2,
    typing Γ e2 t1 -> typing Γ e1 (Arr t1 t2) ->
    typing Γ (App e1 e2) t2.
```

As `t1` does not occur in the result, when processing the constraint `typing Γ e2 t1` it will not be instantiated with a concrete value, which means we can't simply call the checker recursively as in Algorithm 1. One possible way of getting around this issue would be to start enumerating possible types for `t1` and check whether they are a valid depth for the given tree. Of course, this is too inefficient. What if instead we could enumerate `t1`s such that the predicate `typing Γ e2 t1` holds given some concrete values for `Γ` and `e2`? That is precisely the focus of the following section.

4 Generalizing the Derivation Procedure

Our goal in this section is to generalize the algorithm of the previous section that derives *checkers* to also derive *producers*, which can create (enumerate or randomly generate) arguments such that $P\ e_1\ \cdots\ e_n$ holds.

Producers. Producing structured data lies at the core of property-based testing, and is usually achieved through generation [1, 8, 12–14, 21, 23, 28] or enumeration [5, 7, 22, 26]. However, they are usually treated as completely distinct approaches, with distinct implementations. Rather than replicating infrastructure for both, we identify a common monadic core between the two, introducing the notion of *producers*: bounded value-producing monadic actions that admit the same possibilistic reasoning, facilitating maximal code—and more importantly proof—reuse.

Consider the following types:

```
Inductive G A := MkGen : (nat -> Rand -> A) -> G A.
Inductive E A := MkEnum : (nat -> List A) -> E A.
```

The type `G A` represents generators for some type `A`, and is a wrapper around functions from some size parameter and a random seed to `A`. Similarly, the type `E A` represents enumerators for `A` and is a wrapper around function from sizes to (lazy) lists of `A`.

We implement four monadic operations for both `G` and `E`: return, bind, and two ways of encoding failure: $\texttt{fail}_m$ (signifying failure to produce an inhabitant) and $\texttt{fuel}_m$ (signifying running out of fuel), each subscripted by $m \in \{\texttt{G},\texttt{E}\}$:

```
ret_m : A -> m option A
bind_m : m option A -> (A -> m option B) -> m option B
fail_m, fuel_m : m option A
```

Note that the monadic operations operate on `option A` instead of just `A`, in line with established QuickChick conventions [24].

Example: Type Inference. The key piece of intuition behind this generalization is that checkers and producers for a given inductive relation P are extremely similar. Consider the derived enumerator for types `t`, such that `typing Γ e t` holds for given `Γ` and `e`, depicted in Figure 2. A direct comparison with the checker from Figure 1 reveals the high-level structure to be basically identical. We first match on `size` to ensure termination by handling only nonrecursive constructors; we create a handler for each constructor that enumerates all possible `type`s it supports; and we group all handlers together with `enumerating`—a function that takes a list of enumerators and concatenates their results. Just like for checkers, we include $\texttt{fuel}_\texttt{E}$ as an option when running out of fuel (elided for brevity along with the `Con` and `Abs` cases and each handler's thunks).

Handlers for individual constructors are also similarly derived. Each one still performs a match against the patterns of the constructor's result, and then processes any preconditions in order. For example, the handler for `TAdd` first ensures that the input term `e` is of the form `Add e1 e2`, and then makes recursive calls for `e1` and `e2` to enumerate types `t1` and `t2` such that `typing Γ e1 t1` and `typing Γ e2 t2` hold respectively. These recursive calls are executed in sequence using the monadic $\texttt{bind}_\texttt{E}$. Finally, `t1` and `t2` are *checked* to be equal to `N`, before returning the singleton enumeration `N`—the only type `TAdd` prescribes in its conclusion.

For a nonrecursive example consider the handler for the `TVar` constructor. It only makes a single call to another enumerator (using the `enumST` typeclass method), which enumerates `t` such that `lookup Γ e t` holds. This enumerator can also be derived automatically using the same algorithm.

Finally, taking a closer look at the handler for `TApp` reveals an interesting situation: when processing the `typing Γ e1 (Arr t1 t2)` constraint, `t1` is fixed (from the first recursive call) while `t2` still needs to be enumerated. The derived enumerator opts for a recursive call to generate a fresh variable `t12`, matches against `Arr t1' t2` to create a binding for `t2`,

```
Fixpoint rec size top_size Γ e : E (option type) :=
  match size with
  | O => ... base cases elided for brevity ...
  | S size' =>
    enumerating [
      ... Con, Abs handlers elided for brevity ...
      match e with
      | Add e1 e2 =>
        bind_E (rec size' top_size Γ e1) $ fun t1 =>
        bind_E (rec size' top_size Γ e2) $ fun t2 =>
        if (t1 = N)? && (t2 = N)?
        then ret_E N else fail_E
      | _ => fail_E end;
      match e with
      | Var x =>
        enumST top_size (fun t => lookup Γ e t)
      | _ => fail_E end;
      match e with
      | App e1 e2 =>
        bind_E (rec size' top_size Γ e2) $ fun t1 =>
        bind_E (rec size' top_size Γ e1) $ fun t12 =>
        match t12 with
        | Arr t1' t2 =>
          if t1 = t1'? then ret_E t2 else fail_E
        | _ => fail_E end
      | _ => fail_E end ]
  end.
```

Figure 2. Derived enumerator for typing (simplified).

and then checks that `t1` and `t1'` are equal (similarly to how non-linear patterns were handled).

To arrive at such an enumerator automatically, we need to answer two questions. How can we sequence different operations (producers, checkers, or recursive calls) in a generic fashion? And how do we decide when we need to call a producer or a checker when processing a constraint?

Sequencing computations, generically. Converting the enumerator of Figure 2 to a generator is trivial: we substitute the enumerator monadic actions `ret`, `bind`, and `fail` of `E` with the corresponding generator ones, change the typeclass method called from `enumST` to its generator variant `genST`, and change `enumerating` for QuickChick's `backtrack` combinator that has an equivalent `G`-based type.

To unify producers and checkers, the key realization is that the three-valued conjunction `.&&` can also be seen as a monadic bind by treating `option bool` as a monad—or, more precisely, a "then" (`>>`). We also need to sequence computations in different monads together. For example, let's finally complete the checker of Figure 1 by constructing the `TApp` handler. The problem was that when processing the constraint `typing Γ e2 t1`, we could not recursively invoke `rec` as `t1` is not bound. Using the derived enumerator of Figure 2,

we can *produce* a `t1` that satisfies this constraint, but we would need to sequence that with the rest of the checker. To that end, we implement another "bind" combinator $\texttt{bind}_{\texttt{EC}}$ with type `E (option A) -> (A -> option bool) -> option bool`, which simply iterates through all enumerator results, yielding the following handler:

```
match e, t  with
| App e1 e2, t2 =>
  bindEC (enumST top_size (fun t => typing Γ e2 t))
  $ fun t1 =>
  bindC (rec size' top_size Γ e1 (Arr t1 t2))
  $ fun _ => retC true
| _, _ => failC
```

We also sometimes need to invoke checkers while deriving a producer, which requires the converse $\texttt{bind}_{\texttt{CE}}$ and $\texttt{bind}_{\texttt{GE}}$ "binds", which are simple pattern matches like `.&&`, but whose continuation is a producer rather than a checker. Appendix A contains additional implementation details.

Processing constraints. The only thing remaining is to decide, when processing each constraint, whether we need to invoke a checker, a producer, or make a recursive call. To decide that, we will keep track of a mapping *vars* from variables to (statically known) information about them. Each variable can either be: (1) fully instantiated with a value that is statically unknown but fixed, when it either is a top-level argument to the fixpoint (such as `Γ` or `e`), a variable in a pattern match (such as `e1` or `e2` in the `TAdd` case), or the result of a producer call (such as `t1` and `t2` in the `TAdd` case); (2) undefined, when it is an output to be produced; or (3), (potentially partially) instantiated via a pattern match (such as `e` after performing the match of `TAdd`, when it is known to have the form `Add e1 e2`, for fixed but unknown `e1` and `e2`).

The *vars* map is initialized for each constructor by inspecting its result type (after any rewriting to handle non-linear patterns or function calls), as seen in Algorithm 2. It relies on a user-provided *out_set*, a set describing which variables are meant to be produced (similar to *modes* in functional-logic programming). At the end of this procedure each function input $\texttt{in}_i$ maps to its corresponding pattern, and every variable in these patterns is marked either as an input or an output, propagating the *out_set* information.

Given a mapping *vars*, let's assume that we are processing a constraint $Q_i\ e_{i1}\ \cdots\ e_{in_i}$ for a constructor of a relation P. If the type constructor Q being handled is not the one being derived, then it clearly can't be solved by a recursive call. In this case, the algorithm looks up whether there exists a producer or checker typeclass instance that can be used to instantiate any variables that are still marked as *output*. It will favor producers over checkers as in the `TVar` case of Figure 2: using a producer to enumerate `ts` that satisfy `lookup Γ e t` directly is preferable to enumerating `ts` arbitrarily and then checking whether `lookup Γ e t` holds.

Algorithm 2. Environment Initialization

```
1:  function INIT_ENV(ctr, out_set)
2:      (∀ x̄. Q_i e_i1 ··· e_in_i → P e_1 ··· e_p) ← ctr
3:      vars ← ∅
4:      for each in_i ∈ in_1, ..., in_p do
5:          vars ← vars ∪ {in_i ↦ e_i}
6:          for each x_i ∈ variables(e_i) do
7:              if in_i ∉ out_set then
8:                  vars ← vars ∪ {x_i ↦ input}
9:              else
10:                 vars ← vars ∪ {x_i ↦ output}
11:     return vars
```

The interesting case is when the constructor being handled is P. The essence of this case is to decide: if the arguments can be used as arguments to a recursive call as is; if some arguments need to be instantiated before doing so; and if any arguments are more instantiated than they should be (tagged as inputs while expected to be outputs). This gives rise to a notion of *compatibility*, which takes a variable x and an expression e, and returns either $\bot$ (if the expression imposes constraints on x that can't be satisfied) or a set of variables in e that need to be instantiated before a recursive call can be made, in addition to an (optional) pattern to ensure produced values agree with potentially partially instantiated forms (such as in the handling of `TApp`).

$$
\begin{aligned}
&\mathit{compatible\ vars\ x\ y}\ = \\
&\quad \mid \mathit{if}\ vars(x) = vars(y) \rightarrow (\emptyset, \cdot) \\
&\quad \mid \mathit{if}\ vars(x) = input, vars(y) = output \rightarrow (\{y\}, \cdot) \\
&\quad \mid \mathit{otherwise} \rightarrow \bot \\
&\mathit{compatible\ vars\ x\ (C\ \overline{e})}\ = \\
&\quad \mid \mathit{if}\ vars(x) = output \rightarrow (\emptyset, C\ \overline{e}) \\
&\quad \mid \mathit{otherwise} \rightarrow (variables(\overline{e}), \cdot) \\
&\mathit{compatible\ vars\ x\ (f\ \overline{e})}\ = \\
&\quad \mid \mathit{if}\ vars(x) = output \rightarrow \bot \\
&\quad \mid \mathit{otherwise} \rightarrow (variables(\overline{e}), \cdot)
\end{aligned}
$$

Armed with the notion of *compatibility* there are three possibilities to consider:

- Every argument to P in the constraint is compatible with its corresponding argument in the computation being defined, and no variables need additional instantiation. In this case, we can just make a recursive call, perhaps followed by a pattern match as in `TApp`.
- There exists some incompatible argument that is more instantiated than expected. That can only happen when deriving a producer (as in a checker all arguments need to be instantiated). In this case, we simply invoke a checker for the relation (as in the checker for `TApp`).
- Every argument to P in the constraint is compatible with its corresponding argument in the computation being defined, but some variables do need additional instantiation. In this case, we have a choice: we can produce them and then make a recursive call; or we can make an external

call to a constrained producer. We favor enumeration over checking (either recursively or by invoking an appropriate typeclass method should such an instance exist), as that generally tends to lead to more efficient code.

5 Correctness via Translation Validation

To establish the correctness of derived computations we follow a translation validation approach [32]. Using Ltac2 [29], we construct, for each derived computation, a proof that it is sound and complete with respect to the inductive relation that it was derived from. This approach allows us to generate proofs inside the Coq proof assistant (as opposed to proving a meta-theroem about the derivation procedure). Then these proofs can be used in other Coq proofs, for example when carrying out proofs by reflection (Section 6.3) or when using the derived programs as components in larger mechanized developments. Here, we describe the formal properties that we prove, and then we sketch the translation validation scheme that we use. This sketch also serves as a metatheorem that our derivation procedure is sound and complete for all the inductive relations that it handles. In the following we use `typewriter` font for Coq code and *italics* to indicate metavariables ranging over Coq terms.

5.1 Formal Properties

We formally state and prove that each derived computation is sound and complete with respect to the inductive relation it was derived from.

Checkers. For a checker of a predicate P of arity m, soundness means that if it returns `true` for some fuel parameter s and some given inputs then the inductive predicate holds on the given inputs. Or, put formally,

$$\forall\ \mathtt{s},\ \mathtt{check}\ \mathtt{s}\ (P\ e_1\ \ldots\ e_m) = \mathtt{Some}\ \mathtt{true} \rightarrow P\ e_1\ \ldots\ e_m$$

Conversely, completeness requires that if the predicate holds for some given inputs, then there exists some size parameter for which the checker returns true on these inputs.

$$P\ e_1\ \ldots\ e_m \rightarrow \exists\ \mathtt{s},\ \mathtt{check}\ \mathtt{s}\ (P\ e_1\ \ldots\ e_m) = \mathtt{Some}\ \mathtt{true}$$

Checkers should also be *monotonic*: providing a larger size parameter should only increase the precision of the computation. Monotonicity means that once the checker has decided about the validity of the predicate, this decision cannot be changed. Formally,

$$\begin{aligned}&\forall\ \mathtt{s_1}\ \mathtt{s_2}\ \mathtt{b},\ \mathtt{s_1} \leq \mathtt{s_2} \rightarrow\\ &\qquad \mathtt{check}\ s_1\ (P\ e_1\ \ldots\ e_m) = \mathtt{Some}\ \mathtt{b} \rightarrow\\ &\qquad \mathtt{check}\ s_2\ (P\ e_1\ \ldots\ e_m) = \mathtt{Some}\ \mathtt{b}\end{aligned}$$

Using monotonicity and completeness one can also derive soundness for the negations of the checker. That is,

$$\begin{aligned}&\forall\ \mathtt{s},\ \mathtt{check}\ \mathtt{s}\ (P\ e_1\ \ldots\ e_m) = \mathtt{Some}\ \mathtt{false} \rightarrow\\ &\quad \neg\ (P\ e_1\ \ldots\ e_m)\end{aligned}$$

Unfortunately, completeness for negation cannot be derived and it does not hold in general as the ability of inductive relations to encode nonterminating computations gets in the way. Consider the following inductive predicate over natural numbers that holds for `0` and no other number, but in a rather roundabout way:

```
Inductive zero : nat -> Prop :=
| Zero    : zero 0
| NonZero : forall n, zero (S n) -> zero n.
```

A derived checker for `zero` will always return `None` for every non zero number, no matter how much fuel we give it, even though the property does not hold. It will keep trying to satisfy the `zero (S n)` premise of the `NonZero` constructor (which can never be satisfied), until it runs out of fuel.

Producers. Producers satisfy similar properties, but have more complicated semantics. To reason about them, we use the *set of outcomes* semantics that was used by [24] to reason about generators. In particular, for a producer of elements of type A, $prod$: `m` A, we define $[\![prod]\!]_{\mathtt{s}}$ to be the set of elements of type A that are produced for the given size.

Producers are also size-monotonic. That is, any element that can be produced using some size can also be produced using a larger size.

$$\forall\ \mathtt{s_1}\ \mathtt{s_2},\ \mathtt{s_1} \leq \mathtt{s_2} \rightarrow [\![prod]\!]_{\mathtt{s_1}} \subseteq [\![prod]\!]_{\mathtt{s_2}}$$

To specify correctness, we first define the set of elements that can be generated for *any* size parameter as

$$[\![\mathtt{prod}]\!] \stackrel{\text{def}}{=} \{x \mid \exists\ s,\ x \in [\![\mathtt{prod}]\!]_{\mathtt{s}}\}$$

Now, let $prod$ be a producer for the i-th argument of a predicate P with arity m, and let $e_1\ \ldots\ e_{i-1}\ e_{i+1}\ \ldots\ e_m$ be inputs for the rest of the arguments. We say that the producer is sound if every generated `x` satisfies the predicate, and complete if any `x` that satisfies the predicate can be generated. Formally,

$$\forall\ \mathtt{x},\ \mathtt{x} \in [\![prod]\!] \leftrightarrow P\ e_1\ \ldots\ e_{i-1}\ \mathtt{x}\ e_{i+1}\ \ldots\ e_m$$

5.2 Proof Derivation

Our Ltac2 scripts automatically derive proofs for the formal properties we stated in the previous section. The translation validation scheme is expected to succeed for all generated programs. The only exception to that is inductive predicates that have constructors with one or more negated premises. As we explain below, in such cases we cannot derive a completeness proof, unless the negated predicate is fully decidable. We only outline the proofs of soundness and completeness for checkers; the proofs for producers follow the same principles.

Let $P : T_1 \rightarrow \cdots \rightarrow T_m \rightarrow \mathtt{Prop}$ be an inductive predicate of arity m for which the derivation algorithm generates a semi-decision procedure of the form

```
Fixpoint rec (size top_size : nat) (in1 : T1) ... :=
  match s with
  | 0 => backtrack [ ... ] | S n => backtrack [ ... ]
  end.
```

where size is the decreasing size argument and top_size the size parameter for external calls to already defined checkers and producers.

To proceed, we need a specification for the backtrack combinator. This states that backtrack returns true if and only if there exist some checker in its input list that returns true.

```
∀ l, backtrack l = Some true ↔
∃ ch, ch ∈ l ∧ ch tt = Some true
```

5.2.1 Soundness. First, we focus on the soundness proof. Assuming rec size top_size in_1 ... in_m = Some true, we need to show that P in_1 ... in_m. The proof proceeds by induction on the size argument size.

The proof strategy is mostly the same in both the zero (size = 0) and successor (size = S size') cases. In both cases, we know that backtrack [ch_1; ...] = Some true for a list of checkers, and we need to show that P in_1 ... in_m holds. We apply the specification of backtrack to the hypothesis and we obtain that there is some ch in [ch_1; ...] such that ch tt = Some true. Therefore, we need to show that for any ch_i, if ch_i tt = Some true then P in_1 ... in_p. Each ch_i is either a checker that was produced by CTR_LOOP and checks whether a particular constructor of the type can be used to satisfy P with the current inputs, or the trivial checker fun _ => None, that is appended to the list of checkers when when the size is 0 and P has recursive constructors that are not checked. The latter case is trivial: it follows by contradiction, as Some true = None can't hold.

To prove the rest of the cases we perform an iterative process that examines the shape of the checker in the hypothesis. Each case in the loop matches a construct generated by CTR_LOOP.

- **Pattern matching.** The hypothesis is of the form:

  ```
  match x with
  | C p1 ... pl => ch   | _ => Some false
  end = Some true
  ```

  The pattern matching checks whether an input or an enumerated variable has the right form. By case analysis on x, we know that it has to be of the form C p_1 ... p_l, otherwise we derive a contradiction. We are now left with the hypothesis ch = Some true.
- **Checker matching.** The hypothesis is of the form:

  ```
  check top_size Q .&& ch = Some true
  ```

  where check top_size Q checks the validity of Q, which is a premise of the constructor that is currently checked. From this we obtain that Q holds (using the soundness proof for the checker of Q) and that ch = Some true.
- **Checker matching (negation).** When ¬Q is a premise of the constructor the hypothesis will be of the form:

  ```
  ~(check top_size Q) .&& ch = Some true
  ```

  where ~ maps Some true (resp. Some false) to Some false (resp. Some true), and leaves None unaffected. From this we obtain that check top_size Q = Some false and that ch = Some true. Using the soundness for the negation of the checker (which, as we explain in section 5.1, we can derive for free), we can conclude ¬Q.
- **Recursive call.** The hypothesis is of the form:

  ```
  rec size' top_size e1 ... ep .&& ch = Some true
  ```

  that performs a recursive call to the checker. We conclude that rec size' top_size e_1 ... e_p = Some true and that ch = Some true. Using the induction hypothesis, we obtain that P e_1 ... e_p.
- **Enumeration.** The hypothesis is of the form :

  ```
  bindEC (enumST top_size (fun t => Q t))
         (fun x => ch) = Some true
  ```

  where $bind_{EC}$ is the combinator that sequences enumeration and checker operations as in the previous section. From this hypothesis we derive that there exist some x in the set of outcomes of enumST top_size (fun t => Q t) such that ch x = Some true. From the soundness property of the enumerator we also derive that Q x holds.

We repeat this process until the checker in the hypothesis has been reduced to Some true. At this point, all the premises that are required to satisfy the constructor that the current ch_i checks will be in the context and all of the inputs will match the result of the constructor, concluding the proof.

5.2.2 Completeness. To show completeness we need to prove that if P in_1 ... in_p, there exist parameters size and top_size such that rec size top_size in_1 ... = Some true. We proceed by induction on the derivation of P in_1 ... in_m. It suffices to show that there exists some size such that rec (S size) size in_1 ... in_p = Some true, which further simplifies to backtrack [ch_1; ...; ch_m] = Some true.

By applying the specification of backtrack, we just need to prove that there exists some ch in the list [ch_1; ...; ch_m], such that ch tt = Some true. Depending on which constructor was used to derive the hypothesis P in_1 ... in_p, we can figure out which of the handlers ch_i to choose as a witness for ch. Then we can prove that ch tt = Some true by iteratively applying the following process.

- **Pattern matching.** The goal is of the form:

  ```
  ∃ size, match C e1 ... el with
          | C p1 ... pl => ch | _ => Some false
          end = Some true
  ```

  In this case, the scrutinee already has the right form, as this is enforced by the constructor that was used to derived the hypothesis (which corresponds to the handler

we provided as witness). Therefore, the goal simplifies to ch `= Some true`, where all pattern variables $\mathtt{p}_1, \ldots, \mathtt{p}_l$ have been substituted with the corresponding sub-expressions $e_1, \ldots, e_l$ of the scrutinee.

- **Checker matching.** The goal is of the form:

  `∃ size, check size` Q `.&&` ch `= Some true`

  where Q is an inductive relation that is a premise of the rule. Because both checkers `check size` Q and ch are monotonic in the size parameter, we can find witnesses for the size parameter independently: the maximum of the two is then used as a witness for the outer `size`. Therefore, it suffices to show that `∃ size, check size` Q `= Some true` and that `∃ size,` ch `= Some true`. The solve the former obligation, we use the completeness of the checker together with proof of Q should be present be present in the context as it is generated by the induction on the derivation. To solve the latter goal we proceed recursively.
- **Checker matching (negation).** The goal is of the form:

  `∃ size, ~(check size` Q`) .&&` ch `s = Some true`

  where Q is an relation whose negation is a premise of the constructor we are currently examining. Again, it suffices to show that `∃ size, check size` Q `= Some false` and that `∃ size,` ch `= Some true`. For the former, a proof of $\neg Q$ must have been generated in our context from the induction. However, as we discuss in section 5.1, completeness for the negation of our semi-decision procedures does not generally hold. We can obtain completeness for negation if Q is a fully decidable proposition (a common case for this is when Q is equality for a decidable type). Therefore, we can show completeness for this case only if a decidability procedure has been provided for Q. The latter goal is handled recursively.
- **Recursive call.** The goal is of the form

  `∃ size, rec size size` $e_1 \ldots e_m$ `.&&` ch `= Some true`

  As before, we need to show that `∃ size, rec size size` $\mathtt{e}_1 \ldots \mathtt{e}_m$ `= Some true` and that `∃ size,` ch `= Some true`. As usual, the latter is being handled recursively. For the former, since $P\ e_1 \ldots e_m$ must be in context from the induction, we obtain the goal by applying the induction hypothesis.
- **Enumeration.** The goal is of the form

  ```
  ∃ size, bind_EC (enumST size (fun t => Q t))
                  (fun x => ch) = Some true
  ```

  To handle this case, we must show that there exists some `x` such that there exists some `size` for which `x` is in the set of outcomes of `enumST size (fun t =>` Q `t)`, and that `∃ size,` ch `= Some true`. Since a proof for $Q\ x$ for some x should already be in our context, generated by the induction, we derive the former goal using completeness of the enumerator. The latter case is handled recursively.

This process is repeated until the goal is of the form `∃ size, Some true = Some true`, which is trivial.

5.3 Proof Engineering

We briefly discuss our experience using Coq's new tactic language, Ltac2. We found Ltac2 to be an indispensable tool in our arsenal. Compared to regular Ltac, it gives a much more comprehensive ability to inspect and construct Coq terms, which makes it a powerful metaprogramming facility. Unlike Ltac, Ltac2 is typed, which made it substantially easier to identify errors. However, we also encountered some limitations. Ltac2 does not provide a way of finding the current goal number, e.g., after performing induction or case analysis. As a result, we could not immediately select the witness `ch` in the proof of completeness by indexing into the list, but we have to check if the proof succeeds for each one of the checkers. Naturally, this hurts the efficiency of our proof scripts that becomes quadratic in the number of constructors instead of linear.

Our proof engineering heavily relies on Coq's typeclass mechanism [33] to resolve proof obligations for previously derived procedures. In particular, we turn monotonicity and correctness into typeclasses and derive instances of these typeclasses using our Ltac2 scripts. This way, the correctness and monotonicity proof obligations generated by our translation validation proof scripts can be resolved automatically without having to apply specific lemmas.

6 Evaluation

6.1 Practicality: Software Foundations

To evaluate whether the class of inductive relations handled is useful in practice, we target a large body of representative relations: the Software Foundations series of online textbooks. In particular, we extract and verify checkers and producers from inductive relations found in the first two volumes (Logical Foundations—LF [31] and Programming Language Foundations—PLF [30]). These volumes contain a particularly appealing body of relations to target, ranging from predicates on lists and natural numbers to regular expression matchers, and from stateful evaluators for imperative languages to typing relations for lambda calculi. Moreover, as part of an introductory textbook for Coq, they showcase diverse ways of writing down a relation to stress test the robustness of our implementation to stylistic changes.

For each inductive relation defined in the first two volumes, (including exercise solutions), we automatically derive

	Inductive Relations	Computations Derived	Baseline (Algorithm 1)
LF	38	30	11
PLF	71	67	25

Table 1. Number of inductive relations, derived checkers and producers from the Software Foundation textbooks.

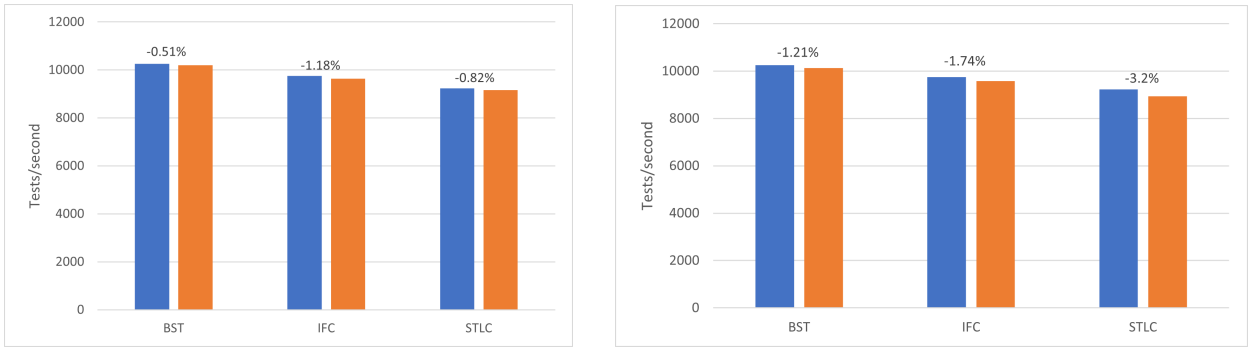

Figure 3. Throughput (tests/sec) of QuickChick case studies using handwritten (blue) or derived (orange) checkers (left) and generators (right).

checkers and producers as well as proofs of their correctness and completeness. We only made a single change, converting the representation of `Maps` (used for environments) from functions to association lists. Generating or checking predicates over functions is beyond the scope of this work—how would one, for example, check equality between two functions over natural numbers? Table 1 shows the number of inductive relations found in both volumes, and the number of them for which we were able to derive enumerators, checkers, and proofs of correctness. Furthermore, as a baseline, we report how many inductive relations could be handled by Algorithm 1. Out of the 38 inductively defined relations found in LF and the 71 found in PLF, 30 and 67 of them respectively do not involve any computations over higher order data, and are therefore within scope of the algorithm presented in this paper. Our implementation could handle all of them correctly. In contrast, Algorithm 1 can only handle 11 and 25 respectively, showing that being able to handle the interdependencies between producers and checkers is vital to capture a practical fragment of inductive relations.

6.2 Random Testing

The primary motivation behind this line of work was to facilitate testing in Coq using QuickChick by establishing confidence in a design before attempting a proof. To that end, testing feedback should be quick and require less effort than proving. However, if users write specifications using inductive relations, requiring them to also write checkers to obtain such feedback is a non-starter. On the other hand, automatically deriving (correct!) checkers is helpful, even if they are slightly less efficient than handwritten ones.

To evaluate the efficiency of our derived procedures we turn to QuickChick's microbenchmark suite, consisting of three case studies from the literature that target: binary search trees [20], information flow control abstract machines [18, 19], and STLC [15]. These cover diverse verification domains (data structures, security, and type systems), and exercise all interesting features of inductive relations not handled by prior work (such as nonlinear patterns, function calls, negation, and existentials), making them excellent candidates to evaluate the performance of our derived computations. In addition, they already come with handcrafted generators and checkers to serve as the baseline.

We first evaluate the efficiency of the *derived checkers*. To do that, we use the same handcrafted generator to produce hundreds of thousands of test inputs, and use both handcrafted and derived checkers to test whether a given property holds. All such properties were already proved to be correct in Coq, and therefore the only metric of interest is throughput: the number of tests per second that can be executed. The left-hand of Figure 3 shows this throughput for the three benchmarks using both handwritten (in blue) and derived (in orange) checkers. In all three benchmarks, the slowdown is minimal, showing less than 2% decrease in performance—a small price to pay to avoid writing and verifying additional code.

We then evaluate the efficiency of the *derived generators*. Just like before, we use the same handcrafted checker to decide whether a property holds, and compare the throughput of the handcrafted and derived generators. The results appear in the right-hand side of Figure 3. The decrease in performance is slightly larger than the derived checkers (between 1% and 3.5%), as the derived generators are more likely to perform some backtracking locally, but still very much worth the cost to avoid handcrafting generators.

At the same time, for a fair comparison, it is important to ensure that the distributions of test data produced by the two generators are similar. Anecdotally, upon manual inspection, their structure seems similar (as expected). However, mechanically demonstrating this similarity remains an open problem in property-based testing. Still, can instead show that the generators are similarly effecting at finding bugs, by leveraging the second component of QuickChick's microbenchmark suite—injecting mutations that cause properties to fail. In particular, the suite introduces errors in: the insertion function in binary search trees (causing the result of an insertion to sometimes violate the search tree

invariant); the substitution and lifting functions of the STLC (causing violations of preservation); and in the label propagation of the information flow control abstract machines (causing violations of noninterference). After running both the handwritten and the derived generators against these microbenchmarks, we found that their mean tests to failure were indistinguishable.

Benchmarking was carried out in a Dell workstation running i7-8700 @ 3.2GHz with 16GB of RAM.

6.3 Computational Reflection

Another application of derived checkers is to facilitate verification, allowing for quick proofs by computational reflection. Proof by computational reflection [4] is a technique that is commonly used in proof assistants in order to reduce the size of proof terms and, consequently, the time of typechecking. To demonstrate how checkers can help, consider the following `Sorted` predicate on lists:

```
Inductive Sorted : list nat -> Prop :=
| Sorted_nil : Sorted []
| Sorted_sing : forall x, Sorted [x]
| Sorted_cons : forall x y l, x <= y ->
    Sorted (y :: l) -> Sorted (x :: y :: l).
```

Let's try to prove that the list `repeat 1 2000`, consisting of the number 1 2000 times is sorted. A straightforward proof repeatedly applies the suitable `Sorted` constructor.

```
Lemma sorted_2000 : Sorted (repeat 1 2000).
Proof. time (repeat
        (first [ eapply Sorted_cons; [ apply le_n | ]
               | eapply Sorted_sing ])).
(* ... 11.202 secs (11.171u,0.019s) ... *)
Time Qed. (* ... 16.283 secs (16.23u,0.03s) ... *)
```

Unfortunately, the proof term produced this way is huge: constructing it takes 11.202 seconds and typechecking it 16.283 seconds! The standard solution is to write a decision procedure for `Sorted`, prove it sound, apply the soundness theorem to the goal, and compute the value of the decision procedure for the list. Using our framework, writing and proving correct such a procedure can be fully automated. We only have to write the following commands:

1. Derive a checker for `Sorted`:

   ```
   Derive DecOpt for (Sorted l).
   ```

2. Derive its proof of soundness:

   ```
   Instance Sort_sound l : DecOptSoundPos (Sorted l).
   Proof. derive_sound. Qed.
   ```

3. Use the `sound` typeclass method in the proof:

   ```
   Lemma sorted_2000' : Sorted (repeat 1 2000).
   Proof. time (eapply sound with (s := 2000);
                compute; reflexivity).
   (* ... 0.05 secs (0.05u,0.s) ... *)
   Time Qed. (* ... 0.059 secs (0.058u,0.s) ... *)
   ```

Constructing and typechecking the proof term now take less than 60 milliseconds each, with minimal user effort!

7 Related Work

As mentioned throughout the paper, extracting forms of computation from inductive relations has received a lot of attention. While we would have liked to evaluate our derived computations against prior work, the closest one is no longer maintained. Instead, in this section, we thoroughly discuss the most directly relevant approaches. At a high level, the key distinguishing feature of our work is that by unifying the treatment of checkers and producers in the same framework, we are able to leverage the interdependency between them and significantly expand the class of inductive relations targeted, while providing strong fully mechanized correctness guarantees (including both soundness and completeness). We also maximize code and proof reuse, simplifying the maintenance of the implementation in the process.

Checkers and Functional Content. Arguably, the closest line of work is the pioneering work of Catherine Dubois' group. Starting with Delahaye et al. [11], they present a method for extracting functional content from Coq to OCaml out of a fragment of inductive relations using mode analysis, and present a metatheoretic proof of its soundness. In Tollitte et al. [34], they adapt their approach to extract functions inside the logic of Coq, with mechanized soundness proofs. In their work, functions can either be partial mode (single output-multiple inputs) or full mode (i.e., a checker). Our work generalizes this approach by extending partial-mode functions to sound and complete enumerators, allowing for a broader class of inductive relations to be checked. Existential variables can only be handled if they are arguments to a premise that has a partial mode in that argument. But if this premise is not deterministic in that argument, then the extracted full mode is necessarily incomplete (i.e., the full mode might return false when the relation holds). Other limitations of this work that are not present to ours are that it requires inductive relations to be structurally recursive in order to handle nontermination (which excludes, for example, substitution-based evaluation relations), non-linear patterns, and negated premises.

Similarly, Berghofer and Nipkow [3] provide a way of executing a fragment of higher-order logic in the context of Isabelle inside the logic itself. Later, Berghofer et al. [2] adapt this approach using mode analysis to transform Isabelle predicates to functional equations, which can in turn be used for code generation [16]. However, they only justify their approach metatheoretically (i.e., no mechanized proofs of correctness), they can't mix functions and predicates arbitrarily, and their translation amounts to unrestricted depth-first search which can lead to nonterminating computation (something that is ruled out conclusively by a completeness proof such as the one in Section 5.2.2).

Enumerators. The most prominent line of work on deriving enumerators from inductive relations is Bulwahn [6] in Isabelle's QuickCheck [5]. Bulwahn presents a view of Isabelle specifications as logic programs that allows for enumerating data directly satisfying the preconditions of such specifications. They then use these enumerators to greatly reduce the number of tests needed to falsify properties compared to simpler type-based approaches. However, they do not reason (even informally) about the correctness of their approach, which invites subtle errors. For instance, they only pass around a single `size` parameter to limit the depth of their enumeration, which can lead to incompleteness.

In the simply typed world, Yakushev and Jeuring [36] use generic programming based on spine views [17] to enumerate GADTs, such as well-typed terms. However, GADTs are far less expressive than inductive relations and there is naturally no mechanized proof of correctness (though proof assistants like Liquid Haskell [35] could be used to that end).

Random Generators. The other closely related line of work is that by Lampropoulos et al. [24] on deriving random generators that produce constrained random data based on an inductive relation, along with a proof of correctness using the same notion of correctness (originating from Dybjer et al. [12]). However, their derived generators rely on manually written and verified checkers. Moreover, their approach was to generate *proof terms* in OCaml using the same generic code that derived the generators themselves. This significantly complicated the code, and created a maintenance nightmare which ultimately lead to leaving the proof derivation component out of the Coq CI when QuickChick became a core package. In contrast, in this work, by unifying generators, enumerators, and checkers we provide a framework that fully subsumes their work, all while greatly simplifying the code and proof base, and allowing users to leverage it not only to fully automate their testing workflow, but also incorporate it into their proof scripts.

8 Limitations and Discussion

While our approach handles a much broader class than prior work, there are still certain limitations, besides the quadratic (in the number of constructors) completeness proof terms discussed in Section 5.3. To begin with, the class we target itself does not allow for `let` expressions between different premises (a feature that Coq `Inductive`s allow). While it seems straightforward to simply substitute the body of a `let` in the subsequent premises, we have not yet explored the implications of removing expression sharing. Moreover, our algorithm currently processes constraints sequentially, which means their order matters: switching premises around could instantiate variables in a different order, resulting in potentially different performance. We leave figuring out how to best traverse constraints for future work.

Finally, the implementation does not currently support two features that the algorithm does in principle: mutually inductive types and multiple producer outputs. The former is a direct result of our choice to rely on Coq's typeclasses: while this choice significantly aided proof automation, since Coq's typeclasses cannot be mutually recursive, neither can our derived checkers and producers that rely on them. The latter was a pragmatic choice: we didn't have a use case for it and it would take a non-trivial engineering effort to implement; still, this shouldn't be a fundamental limitation, as the algorithm already supports this through *out_set*.

On the other hand, deriving decision procedures directly (instead of semi-decision ones) should be straightforward to implement using a minor variation of the algorithm presented. Simply removing the fuel parameter and converting each enumerative producer to a functional version (by using an `Identity` monad instead of `E option`), would yield a decision procedure in many cases of interest. However, the result would be quite user-unfriendly, as as in many cases attempting such a derivation would result in Coq's termination checker failing on a program the user didn't write. Taking also into account that arguably the most appealing aspect of inductive relations is to precisely encode features that don't allow for total and elegant decision procedures (e.g. nontermination, nondeterminism, etc.), we chose not to pursue this endeavor. Instead, we focused on handling as large a class of inductives as possible, while keeping things compositional enough to allow for automatic proof construction.

9 Conclusion and Future Work

In this paper, we described a generic algorithm for deriving computational content from inductive relations in the form of checkers, enumerators, and random generators in a unified manner, together with mechanized proofs of their correctness. We showed that our framework handles many practical cases of interest and that it can be useful for both testing and verification. In the immediate future, we would like to explore how to support user-defined datatype refinements (such as a function-based representation in proofs with an efficient implementation used in computations). It would also be interesting to explore how deep the connection is between generators and enumerators, and whether there are other useful forms of producers.

Acknowledgments

We thank Harrison Goldstein, Antal Spector-Zabusky, Benjamin Pierce, Michael Hicks, and the anonymous reviewers for their helpful comments. This work was supported by NSF award #2107206, *Efficient and Trustworthy Proof Engineering* (any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF).

References

[1] Thomas Arts, Laura M. Castro, and John Hughes. 2008. Testing Erlang Data Types with QuviQ QuickCheck. In *7th ACM SIGPLAN Workshop on Erlang* (Victoria, BC, Canada). ACM, 1–8. https://doi.org/10.1145/1411273.1411275

[2] Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. 2009. Turning Inductive into Equational Specifications. In *22nd International Conference on Theorem Proving in Higher Order Logics (Lecture Notes in Computer Science, Vol. 5674)*. Springer, 131–146. http://www4.in.tum.de/~haftmann/pdf/turning_inductive_into_equational_specifications_berghofer_bulwahn_haftmann.pdf

[3] Stefan Berghofer and Tobias Nipkow. 2002. Executing Higher Order Logic. In *International Workshop on Types for Proofs and Programs (TYPES) (Lecture Notes in Computer Science, Vol. 2277)*. Springer, 24–40. http://www4.in.tum.de/publ/papers/TYPES2000.pdf

[4] Yves Bertot and Pierre Castéran. 2004. ** Proof by Reflection*. Springer Berlin Heidelberg, Berlin, Heidelberg, 433–448. https://doi.org/10.1007/978-3-662-07964-5_16

[5] Lukas Bulwahn. 2012. The New Quickcheck for Isabelle - Random, Exhaustive and Symbolic Testing under One Roof. In *2nd International Conference on Certified Programs and Proofs (CPP) (Lecture Notes in Computer Science, Vol. 7679)*. Springer, 92–108. https://www.irisa.fr/celtique/genet/ACF/BiblioIsabelle/quickcheckNew.pdf

[6] Lukas Bulwahn. 2012. Smart Testing of Functional Programs in Isabelle. In *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) (Lecture Notes in Computer Science, Vol. 7180)*. Springer, 153–167. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.229.1307&rep=rep1&type=pdf

[7] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. 2011. Integrating Testing and Interactive Theorem Proving. In *10th International Workshop on the ACL2 Theorem Prover and its Applications (EPTCS, Vol. 70)*. 4–19. http://arxiv.org/abs/1105.4394

[8] Koen Claessen, Jonas Duregård, and Michał H. Pałka. 2014. Generating Constrained Random Data with Uniform Distribution. In *Functional and Logic Programming (Lecture Notes in Computer Science, Vol. 8475)*. Springer, 18–34. https://doi.org/10.1007/978-3-319-07151-0_2

[9] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 268–279. http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf

[10] The Coq Development Team. 2021. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.4501022

[11] David Delahaye, Catherine Dubois, and Jean-Frédéric Étienne. 2007. Extracting Purely Functional Contents from Logical Inductive Types. In *20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs) (Lecture Notes in Computer Science, Vol. 4732)*. Springer, 70–85. http://cedric.cnam.fr/~delahaye/papers/pred-exec%20(TPHOLs'07).pdf

[12] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. 2004. Random Generators for Dependent Types. In *First International Colloquium Theoretical Aspects of Computing (Lecture Notes in Computer Science, Vol. 3407)*. Springer, 341–355. https://doi.org/10.1007/978-3-540-31862-0_25

[13] Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *24th European Symposium on Programming (Lecture Notes in Computer Science, Vol. 9032)*. Springer, 383–405. http://users.eecs.northwestern.edu/~baf111/random-judgments/

[14] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *32nd ACM/IEEE International Conference on Software Engineering*. ACM, 225–234. https://doi.org/10.1145/1806799.1806835

[15] Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. 2021. Do Judge a Test by its Cover - Combining Combinatorial and Property-Based Testing. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 264–291. https://doi.org/10.1007/978-3-030-72019-3_10

[16] Florian Haftmann and Tobias Nipkow. 2010. Code Generation via Higher-Order Rewrite Systems. In *Functional and Logic Programming*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 103–117.

[17] Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. 2006. "Scrap Your Boilerplate" Reloaded. In *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3945)*, Masami Hagiya and Philip Wadler (Eds.). Springer, 13–29. https://doi.org/10.1007/11737414_3

[18] Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing Noninterference, Quickly. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 455–468. http://prosecco.gforge.inria.fr/personal/hritcu/publications/testing-noninterference-icfp2013.pdf

[19] Cătălin Hriţcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo de Amorim, Maxime Dénès, John Hughes, Benjamin C. Pierce, and Dimitrios Vytiniotis. 2016. Testing Noninterference, Quickly. *Journal of Functional Programming (JFP); Special issue for ICFP 2013* 26 (April 2016), e4 (62 pages). https://doi.org/10.1017/S0956796816000058 Technical Report available as arXiv:1409.0393.

[20] John Hughes. 2019. How to Specify It! *20th International Symposium on Trends in Functional Programming* (2019).

[21] Casey Klein and Robert Bruce Findler. 2009. Randomized Testing in PLT Redex. In *Workshop on Scheme and Functional Programming (SFP)*. http://www.eecs.northwestern.edu/~robby/pubs/papers/scheme2009-kf.pdf

[22] Ivan Kuraj and Viktor Kuncak. 2014. SciFe: Scala framework for efficient enumeration of data structures with invariants. In *Proceedings of the Fifth Annual Scala Workshop*. ACM, 45–49. https://doi.org/10.1145/2637647.2637655

[23] Leonidas Lampropoulos. 2018. *Random Testing for Language Design*. Ph.D. Dissertation. University of Pennsylvania.

[24] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating good generators for inductive relations. *PACMPL* 2, POPL (2018), 45:1–45:30. https://doi.org/10.1145/3158133

[25] Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickCHick: Property-Based Testing In Coq*. Electronic textbook. http://www.cis.upenn.edu/~bcpierce/sf

[26] Max S. New, Burke Fetscher, Robert Bruce Findler, and Jay A. McCarthy. 2017. Fair enumeration combinators. *J. Funct. Program.* 27 (2017), e19. https://doi.org/10.1017/S0956796817000107

[27] Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming* (Heijen, The Netherlands) *(AFP'08)*. Springer-Verlag, Berlin, Heidelberg, 230–266.

[28] Manolis Papadakis and Konstantinos F. Sagonas. 2011. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011*. 39–50. https://doi.org/10.1145/2034654.2034663

[29] Pierre-Marie Pédrot. 2019. Ltac2: Tactical Warfare. The Fifth International Workshop on Coq for Programming LanguagesCoqPL.

https://www.pédrot.fr/articles/coqpl2019.pdfpdf

[30] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. 2018. *Programming Language Foundations*. Electronic textbook, Version 5.5. http://www.cis.upenn.edu/~bcpierce/sf

[31] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2018. *Logical Foundations*. Electronic textbook, Version 5.5. http://www.cis.upenn.edu/~bcpierce/sf

[32] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1384)*, Bernhard Steffen (Ed.). Springer, 151–166. https://doi.org/10.1007/BFb0054170

[33] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics* (Montreal, P.Q., Canada) *(TPHOLs '08)*. Springer-Verlag, Berlin, Heidelberg, 278–293. https://doi.org/10.1007/978-3-540-71067-7_23

[34] Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. 2012. Producing Certified Functional Code from Inductive Specifications. In *Second International Conference on Certified Programs and Proofs (CPP) (Lecture Notes in Computer Science, Vol. 7679)*. Springer. http://cedric.cnam.fr/~delahaye/papers/relext-coq%20%28CPP%2712%29.pdf

[35] Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover*. Ph.D. Dissertation. University of California, San Diego, USA. http://www.escholarship.org/uc/item/8dm057ws

[36] Alexey Rodriguez Yakushev and Johan Jeuring. 2010. Enumerating Well-Typed Terms Generically. In *Approaches and Applications of Inductive Programming*, Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer (Eds.). Lecture Notes in Computer Science, Vol. 5812. Springer Berlin Heidelberg, 93–116. https://doi.org/10.1007/978-3-642-11931-6_5