Merging Inductive Relations

JACOB PRINZ, University of Maryland, USA
LEONIDAS LAMPROPOULOS, University of Maryland, USA

Inductive relations offer a powerful and expressive way of writing program specifications while facilitating compositional reasoning. Their widespread use by proof assistant users has made them a particularly attractive target for proof engineering tools such as QuickChick, a property-based testing tool for Coq which can automatically derive generators for values satisfying an inductive relation. However, while such generators are generally efficient, there is an infrequent yet seemingly inevitable situation where their performance greatly degrades: when multiple inductive relations constrain the same piece of data.

In this paper, we introduce an algorithm for merging two such inductively defined properties that share an index. The algorithm finds shared structure between the two relations, and creates a single merged relation that is provably equivalent to the conjunction of the two. We demonstrate, through a series of case studies, that the merged relations can improve the performance of automatic generation by orders of magnitude, as well as simplify mechanized proofs by getting rid of the need for nested induction and tedious low-level book-keeping.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: inductive relations, merging, QuickChick

ACM Reference Format:
Jacob Prinz and Leonidas Lampropoulos. 2023. Merging Inductive Relations. 1, 1 (February 2023), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

When using a proof assistant such as Coq [Coq Development Team 2021] or Agda [Norell 2008], proof engineers generally reach for inductive relations to express their specifications. When developing such specifications, having access to a testing tool that can quickly uncover errors before embarking on potentially costly proof efforts can be invaluable, which has led to the development of multiple such tools [Bulwahn 2012a; Chamarthi et al. 2011; Eastlund 2009; Lindblad 2007]. Most recently, Paraskevopoulou et al. [2022] extended QuickChick [Lampropoulos and Pierce 2018], the property-based testing tool for Coq, with facilities that specifically target inductive relations. In particular, given an inductive relation P, they showed how to automatically derive both a generator that produces random data satisfying P, as well as a partial decision procedure for P, allowing for rapid testing feedback. These derived generators are generally extremely efficient, with minimal overheads compared to handwritten generators that produce the same distribution. However, as their use became more widespread, one problem became increasingly apparent: when multiple inductive relations constrain the same piece of data, derived generators can only take one such relation into account during generation, with severe implications for generation performance.

Authors' addresses: Jacob Prinz, jacobeliasprinz@gmail.com, University of Maryland, College Park, USA; Leonidas Lampropoulos, leonidas@umd.edu, University of Maryland, College Park, USA.

XXXX-XXXX/2023/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

For concreteness, consider the type of binary trees in Coq [Coq Development Team 2021] with labels at the nodes, and an encoding of binary search trees, satisfying the *search invariant*: for every node with label `x`, every label in the left subtree is smaller than `x` and every label in the right is larger than `x`:

```
Inductive Tree A :=
| Leaf : Tree A
| Node : A -> Tree A -> Tree A -> Tree A.

Inductive bst : nat -> nat -> Tree nat -> Prop :=
| bst_leaf : forall lo hi, bst lo hi Leaf
| bst_node : forall lo hi x l r,
    lo < x < hi ->
    bst lo x l -> bst x hi r ->
    bst lo hi (Node x l r).
```

This `bst` relation characterizes binary search trees with elements between some lower and greater bounds `lo` and `hi`: `Leaf`s are always valid search trees, while `Node`s satisfy the invariant if the label is between `lo` and `hi` and its subtrees are valid search trees recursively with appropriately adjusted bounds.

A theorem one might want to prove about such trees is that `insert`ing an element in the tree preserves this search invariant, as long as the element being inserted is also between the lower and higher bounds:

```
Theorem insert_preserves_bst :
  forall x lo hi t, lo < x < hi ->
  bst lo hi t -> bst lo hi (insert x t).
```

The work of Paraskevopoulou et al. [2022] can be used to automatically derive efficient generators and checkers for the `bst` inductive relation:

```
Derive Generator for (fun t => bst lo hi t).
Derive Checker for (bst lo hi t).
```

These commands define appropriate typeclass instances that allow for generating a tree `t` such that `bst lo hi t` holds for given `lo` and `hi`, and for checking whether `bst lo hi t` holds when all three arguments are given. These instances can then be used to quickly uncover any errors before attempting a proof: [1]

```
QuickChick insert_preserves_bst.
  +++ Passed 10000 tests (0 discards)
  Time Elapsed: 0.56s
  Size Statistics: 0: 37.5%, 1: 12.5%, 2: 10%, ..., 6: 10%
```

QuickChick can quickly generate thousands of valid binary search trees, with a healthy distribution over various depths (albeit slightly skewed towards smaller ones). Indeed, given any inductive relation indexed by simply typed first-order data, QuickChick can generally derive efficient generators and checkers to automate the testing process.

But what happens when there are *additional* constraints on the values to be generated? What if, instead of binary search trees, our development involved AVL trees, which need to also be *balanced*:

[1]For presentation purposes, we have formatted the output of QuickChick to be more readable.

```
Inductive bal : nat -> Tree -> Prop :=
| bal_leaf0 : bal 0 Leaf
| bal_leaf1 : bal 1 Leaf
| bal_node : forall n t1 t2 m, bal n t1 -> bal n t2 -> bal (S n) (Node m t1 t2).
```

Here, the inductive relation `bal n t` characterizes binary trees `t` whose every path from root to a `Leaf` has length `n-1` or `n`.

The straightforward way to define AVL trees would be to simply take the conjunction of `bst` and `bal`. The insertion property of interest would then become:

```
Theorem insert_preserves_avl :
  forall x lo hi t, lo < x < hi ->
  bst lo hi t -> bal h t ->
  bst lo hi (insert x t) /\ exists h' (bal h' (insert x t)).
```

How would one go about testing this property using the automatically derived generators? We could either:

- *generate* trees `t` that are binary search trees and *check* if they are balanced, or
- *generate* trees `t` that are balanced and *check* whether they are valid search trees.

Only then could we check that the result of the insertion is a valid AVL tree.

Unfortunately, neither approach is anywhere close to being reasonably efficient—in fact, their performance renders testing with them essentially ineffective:

```
(* Generate bsts, check if balanced: *)
  *** Gave up! Passed only 2720 tests
  Discarded: 20000
  Time Elapsed: 1.31s
  Size Statistics: 0: 72.5%, 1: 25%, 2+: 2.5%
(* Generate balanced trees, check if bst: *)
  *** Gave up! Passed only 5726 tests
  Discarded: 20000
  Time Elapsed: 0.30s
  Size Statistics: 0: 35%, 1: 50%, 2: 15%
```

Either approach can only generate trivial valid trees, while wasting a lot of generation effort producing larger but invalid ones. The main issue is that *both relations are too sparsely inhabited to be ignored during generation.*

So what can we do? Current property-based testing practice dictates that users write, by hand, a generator that produces trees that are both balanced and valid search trees. However, that can be tedious and error-prone, and lies in stark contrast with QuickChick's intended goal of quickly checking if a goal is false before embarking on a proof effort.

An alternative approach would be to require that users write a *single* inductive relation that incorporates *both* properties. Unfortunately, that is also not ideal: this is a very non-compositional approach that does not allow for component reuse, separate reasoning, and can quickly become unwieldy. But, setting user-friendliness aside for a moment, what if we *did* have access to such a relation?

```
(* Generate balanced binary search trees directly: *)
  +++ Passed 10000 tests (0 discards)
  Time Elapsed: 0.95s
  Size Statistics: 0: 14.3%, ..., 6: 14.3%
```

It would completely solve all problems with the derived generator!

Naturally, one might wonder: *could we automatically obtain such an inductive relation that is the conjunction of two others?* That is precisely the main contribution of this paper. We develop an algorithm for merging inductive relations (like `bst` and `bal`) into a single relation that is provably equivalent to their conjunction, but often far more useful: for testing purposes, it can lead to dramatic speedups of multiple orders of magnitude; for proving purposes, it provides a more powerful induction principle that can be used for hassle-free reasoning.

Our approach is not a panacea: it remains (for now) up to the user to identify cases where merging inductives could be useful and explicitly invoke it. Moreover, when the recursive structure of the inductive relations is fundamentally different, it will provide little benefit in terms of generation. Still, in this paper we identify multiple cases where it does provide substantial benefit. In particular, we offer the following contributions:

- We develop an algorithm for merging two inductive relations into a single one that is equivalent to (but more useful than) their conjunction and implement this algorithm in Coq, using QuickChick's metaprogramming facilities (Section 2).
- We provide a generic proof script that, given two inductive relations P and Q that have been merged into a single one PQ, proves the equivalence of PQ to the conjunction of P and Q, showing in the process that the induction principle obtained is easier to work with (Section 3).
- We demonstrate through a series of case studies that generators derived for a merged inductive relation can be more efficient than generators that don't take both relations into account by orders of magnitude, and that the merging algorithm can apply to a wide range of inductive relations (Section 4).

We discuss limitations of our approach in Section 4.5 and related work in Section 5, before concluding and drawing directions for future work in Section 6.

2 THE ALGORITHM

In this section, we present an algorithm which merges inductive relations of the form:

$$\texttt{Inductive}\ R\ (A_1 \rightarrow \ \dots\ \rightarrow A_n : Type) : T_1 \rightarrow \ \dots\ \rightarrow T_m \rightarrow \texttt{Prop} :=$$
$$\mid C_1\ : \forall x_1\ \dots\ xk,\ (R_1\ e_{11}\ \dots) \rightarrow \dots \rightarrow R\ e_1\ \dots\ e_n \quad \mid \cdots$$

We assume that inductive relations can take an any number of type parameters and any number of (simply typed) indices which may depend on those type parameters. Each of the constructors C_i of the inductive relation can universally quantify over any number of (independent) variables x_i. Each constructor may also constrain these variables via any number of (potentially recursive) inductive relations R_i.

2.1 Formal Problem Statement

Given two inductive relations $P : T_{A1} \rightarrow \ \dots\ \rightarrow T_{An} \rightarrow T \rightarrow Prop$ and $Q : T_{B1} \rightarrow \ \dots\ \rightarrow T_{Bm} \rightarrow T \rightarrow Prop$ of this form, where the last index is of the same type, our goal will be to produce an inductive relation PQ of type $T_{A1} \rightarrow \ \dots\ \rightarrow T_{An} \rightarrow T_{B1} \rightarrow \ \dots\ \rightarrow T_{Bm} \rightarrow T \rightarrow Prop$ that is equivalent to the conjunction of the two:

$$\forall (a_i : T_{Ai})(b_i : T_{Bi})(t : T),\ P\ a_1\ \dots\ a_n\ t\ \wedge\ Q\ b_1\ \dots\ b_m\ t \iff PQ\ a_1\ \dots\ a_n\ b_1\ \dots\ b_m\ t$$

That is, if the relations P and Q hold for some number of unshared indices $a_1\ \dots\ a_n$ and $b_1\ \dots\ b_m$, and a single shared index t, then so will PQ for the same indices and vice-versa.

In the rest of this section, we present our algorithm for merging two relations. Our actual implementation can operate on arbitrarily positioned indices—the only requirement is that the types in these positions unify. For presentation purposes, however, we will assume that the index we're merging over is in the last position and that its type is T. Next, in Section 3, we describe how to prove the formal equivalence above in Coq.

Of course, the merging problem can be trivially solved by simply taking the conjunction of P and Q. Instead, we would like PQ to have more interesting recursive structure, without mentioning P or Q if possible. We demonstrate that our algorithm has this property empirically in Section 4.

2.2 The Algorithm, by Example

Suppose that we have some term t and some terms a_i and b_j for which both $P\ a_1\ \ldots\ a_n\ t$ and $Q\ b_1\ \ldots\ b_m\ t$ are inhabited. That means that there must be some constructor from P and some constructor from Q which create witnesses to these properties. However, not all pairs of constructors can create elements parameterized by the same term t.

For example, in the introduction, we looked at `bst` and `bal` as two relations over trees. Suppose that we would like to merge these into a single relation `AVL: nat -> nat -> nat -> Tree nat -> Prop`. How could a `Tree` t satisfy both `bst` and `bal`? Looking at their definitions, there are intuitively two ways that can happen: either t is a `Leaf`, and the constructors `bst_leaf`, `bal_leaf0`, or `bal_leaf1` were used; or t is a `Node` and the constructors `bst_node` and `bal_node` were used. The remaining constructor combinations cannot be used as their conclusions have incompatible shapes—a `Leaf` and a `Node` can never construct the same tree. This naturally gives rise to *unification* as the core mechanism used to determine which constructors of P and Q could conceivably create elements indexed by the same term.

Each such compatible pair of constructors from P and Q will then give rise to a constructor for PQ that captures the constraints that they impose. In our `AVL` example, that will lead to the following relation:

```
Merge (fun t => bst lo hi t) With (fun t => bal n t) As AVL.

Inductive AVL : nat -> nat -> nat -> Tree nat -> Prop :=
| bst_leaf_bal_leaf0 :
   forall lo hi : nat, AVL lo hi 0 Leaf
| bst_leaf_bal_leaf1 :
   forall lo hi : nat, AVL lo hi 1 Leaf
| bst_node_bal_node :
   forall (n : nat) (l r : Tree nat) (x lo hi : nat),
     lo < x < hi -> AVL lo x n l -> AVL x hi n r ->
     AVL lo hi (S n) (Node x l r).
```

Notably, the pairs of recursive calls to `bst` and `bal` on the left and right subtrees have been merged into a single calls to `AVL`.

2.3 Unification

The first building block of the merging algorithm is unification, which lets us both prune incompatible pairs of constructors (as described above), and allows us to relate variables that can appear in the different constructors. Looking back at our running example, the `Node` constructors have the following conclusions:

```
  bst_node : ... -> bst lo hi (Node x l r)
  bal_node : ... -> bal (S n) (Node m t1 t2)
```

The two trees in their conclusions can be made equal using a substitution $\{m \mapsto x, t1 \mapsto l, t2 \mapsto r\}$. In the general case, we will consider terms that can contain variables, constructors, and applications, as any functions that appear in those positions can be rewritten as equality constraints by QuickChick [Paraskevopoulou et al. 2022].

Formally, unification inputs two terms and outputs a substitution, or a mapping from variables to terms, such that the two terms are equal under that substitution. If such a substitution doesn't exist, it simply outputs $fail$. The following pseudocode represents this computation: a variable can unify with any expression in which it doesn't occur free, two constructors can unify if they are equal, and two applications need to unify in both the function and the argument.

$$
\begin{aligned}
&\texttt{unify} : \texttt{Term} \rightarrow \texttt{Term} \rightarrow \texttt{Maybe Sub}\\
&\texttt{unify}\ x\ e = \textbf{if}\ x \text{ occurs in } e\ \textbf{then}\ \texttt{fail},\ \textbf{else}\ \{x \rightarrow e\}\\
&\texttt{unify}\ e\ x = \textbf{if}\ x \text{ occurs in } e\ \textbf{then}\ \texttt{fail},\ \textbf{else}\ \{x \rightarrow e\}\\
&\texttt{unify}\ C\ C' = \textbf{if}\ C = C'\ \textbf{then}\ \{\}\ \textbf{else}\ \texttt{fail}\\
&\texttt{unify}\ (e_1\ e_2)\ (e_1'\ e_2') = \textbf{let}\ \sigma = \texttt{unify}\ e_1\ e_1'\ \textbf{in}\ \sigma \cup (\texttt{unify}\ (\sigma\ e_2)\ (\sigma\ e_2'))\\
&\texttt{unify}\ \_\ \_ = \texttt{fail}
\end{aligned}
\tag{1}
$$

2.4 Merging Constructors

Armed with unification, given two relations P and Q, we can find all pairs of constructors (c_P, c_Q) which could possibly produce elements parameterized by the same shared parameter. The merged relation PQ will need to have one constructor corresponding to each of these pairs, c_{PQ}. We can therefore reduce our goal of generating all of PQ to a simpler subproblem: generating a single constructor c_{PQ} from constructors c_P and c_Q, given a substitution σ that makes their conclusions equal.

To that end, we can decompose the type of a constructor c of a relation P as a quintuple:

- A set of forall-quantified variables v, such as `lo`, `hi`,`x`,`l`, and `r` in `bst_node`.
- A set of recursive constraints rs over these variables, such as `bst lo x l` and `bst x hi r` in `bst_node`.
- A set of non-recursive constraints os, such as `lo < x < hi` in `bst_node`.
- The list of not-shared terms in its conclusion as, such as `lo` and `hi` in `bst_node`.
- The shared term in its conclusion t, such as `Node x l r` in `bst_node`.

Given two such constructors $c_P = (v_P, rs_P, os_P, as_P, t_P)$ and $c_Q = (v_Q, rs_Q, os_Q, as_Q, t_Q)$, we need to produce a new quintuple to serve as a constructor in PQ. This construction is shown in Algorithm 1.

First, we unify the two shared terms t_P and t_Q. If it fails, this pair of constructors doesn't need to be merged. If successful, this yields a substitution σ, a mapping from some variables in $v_P \cup v_Q$ to terms. This substitution must then be applied to all possible terms (rs, os, as, and t) in both constructor representations. In particular, after applying this substitution, $\sigma(t_P) = \sigma(t_Q)$, and this term is the shared term for the conclusion of the merged constructor. Moreover, the set of variables quantified over in the new merged constructor is the union of the sets of variables quantified in c_P or c_Q, excluding those that were substituted away by unification. The non-shared parameters of the new constructor are simply the concatenation of those of c_P and c_Q after substitution.

What remains is figuring out what to do with the recursive and non-recursive constraints of the two input constructors. The latter is straightforward—every non-recursive constraint that appears in either c_P or c_Q should also appear in their merge, so we simply take their union post-substitution and place them in c_{PQ}.

To tackle recursive constraints, a first naive approach would be to also add the recursive constraints rs_P and rs_Q to the non-recursive (as they're now referring to a different inductive than that for which they are part of its definition) constraints of the new merged constructor. But that would not result in interesting shared recursive structure, and therefore would not facilitate testing or proving. However, looking back at our problem definition, if we have some constraint in rs_P of type $P\ a_1\ \dots\ a_n\ t$ and another constraint in rs_Q of type $Q\ b_1\ \dots\ b_m\ t$, that is equivalent to $PQ\ a_1\ \dots\ a_n\ b_1\ \dots\ b_m\ t$. Therefore, the final step of the merging algorithm is to look at the sets of recursive constraints from c_P and c_Q, find all matching pairs whose shared parameter is equal, and construct a single recursive constraint for c_{PQ} from each pair. Any remaining recursive constraints from the original constructors can then be added to the non-recursive arguments of c_{PQ}.

2.5 Unchanged Shared Parameters

While the algorithm above can handle the majority of cases of interest, there is an interesting interaction (or rather lack of) when a constructor treats the shared index more like a parameter—that is, it does not change across recursive calls. Consider, for instance, inequality over natural numbers defined as a relation:

```
Inductive less : nat -> nat -> Prop :=
  | less_n : forall n, less n n
  | less_S : forall m n, less n m -> less n (S m).
```

Suppose that we want to merge this relation with itself to create a relation $a \leq x \leq b$ for a fixed a and b—this naturally comes up in the `bst` example itself as the constraint on the value of a `Node`! We could do so by merging $less\ a\ x$ with $less\ x\ b$, exploiting the fact that our implementation doesn't actually require the merge to be over the last index.

Unfortunately, the merging procedure in this case is less useful, yielding the following relation:

```
Inductive between : nat -> nat -> nat -> Prop :=
| less_n_less_n :
    forall n' : nat, between n' n' n'.
| less_S_less_n :
    forall m n : nat, less n m -> between n (S m) (S m)
| less_n_less_S :
```

Algorithm 1. Merging Two Constructors

Inputs Two constructors $c_P = (v_P, rs_P, os_P, as_P, t_P)$ and $c_Q = (v_Q, rs_Q, os_Q, as_Q, t_Q)$
Output The merged constructor c_{PQ} or failure.

$\sigma \leftarrow \texttt{unify}\ t_P\ t_Q$
$t := \sigma(t_P)$
$as := \sigma(as_P) \cup \sigma(as_Q)$
$v := v_P \cup v_Q / dom(\sigma)$
$os := \sigma(os_P) \cup \sigma(os_Q) \cup \sigma(rs_P) \cup \sigma(rs_Q)$
$rs := \emptyset$
for $r_P = P\ a_1\ \dots\ a_n\ t_a \in \sigma(rs_P), r_Q = Q\ b_1\ \dots\ b_m\ t_b \in \sigma(rs_Q)$ **do**
    if $t_a = t_b$ **then**
        $os := os / \{r_P, r_Q\}$
        $rs := rs \cup \{PQ\ a_1\ \dots\ a_n\ b_1\ \dots\ b_m\ t_a\}$
return (v, rs, os, as, t)

```
    forall m n' : nat, less n' m -> between n' (S m) n'
| less_S_less_S :
    forall m' m n : nat, less (S m) m' -> less n m -> between n (S m') (S m)
```

The relation is not recursive, and instead simply refers back to the original *less* relation.

However, a simple extension of the algorithm can help, based on the following key idea: since the `less_S` constructor does not change its first parameter at all, it does not need to interact with the other relation it is being merged with. More generally, suppose that our relation P has a constructor with one recursive input, and it does not change the shared parameter from that input to its output. That is c_P is of the form:

$$\ldots \to P\ a_1\ \ldots\ a_n\ t \to \ldots \to P\ a'_1\ \ldots\ a'_n\ t$$

for some parameters a_i and a'_i.

Recall our original correctness criterion on general relations P and Q:

$$\forall (a_i : T_{Ai})(b_i : T_{Bi})(t : T),\ P\ a_1\ \ldots\ a_n\ t\ \wedge\ Q\ b_1\ \ldots\ b_m\ t \iff PQ\ a_1\ \ldots\ a_n\ b_1\ \ldots\ b_m\ t$$

This means that the implication in c_P can be lifted into an implication about PQ:

$$\ldots \to PQ\ a_1\ \ldots\ a_n\ b_1\ \ldots\ b_m\ t \to \ldots \to PQ\ a'_1\ \ldots\ a'_n\ b_1\ \ldots\ b_m\ t$$

As a result, we can add a constructor of this type to PQ, which fully accounts for the effect of c_P, and therefore we don't need to merge it with any constructors of Q.

Using this trick to deal with the `less_S` constructor for the right-hand `less` to be merged, we can perform unification on the remaining pairs of constructors, yielding the following improved result. The `less_S` constructor is transformed into the `less_S'` constructor below:

```
Inductive between : nat -> nat -> nat -> Prop :=
| less_S' : forall x m n : nat, between x m n -> between x (S m) n
| less_Sless_n : forall m n : nat, less n m -> between n (S m) (S m)
| less_nless_n : forall n' : nat, between n' n' n'.
```

While this relation isn't quite as nice as it might be if written by a human (`less_Sless_n` and `less_nless_n` could be combined and simplified), it is recursive and useful for generation and proving purposes.

2.6 Putting it All Together

Assembling all the individual pieces together, our complete algorithm for merging two inductive relations is shown in Algorithm 2. First, we identify opportunities to lift constructors of P (lines 2-6) and Q (lines 7-11) as described in Section 2.5. Then, for every remaining pair of constructors we invoke Algorithm 1 (lines 12-13), adding a constructor to our result for each one. Finally, we return the resulting list of constructor representations for PQ (line 14).

3 REASONING ABOUT AND WITH MERGED RELATIONS

While the algorithm described in the previous section intuitively results in a merged relation that should be equivalent to the conjunction of the two input relations, our implementation using QuickChick's metaprogramming facilities [Lampropoulos 2018] involves quite intricate manipulations of Coq's internal data representations. To ensure the correctness of our implementation, reasoning about the implementation itself is essentially infeasible: the metaprogramming facilities that allow for a maintainable implementation are all written in OCaml, without formally verified counterparts. Instead, we settled for the next best approach: translation validation [Pnueli et al. 1998].

For each merged inductive relation, we automatically prove (via generic proof scripts) *soundness* and *completeness* of the merge: that the merged inductive relation implies the conjunction of the two input inductive relations, and vice-versa. We then demonstrate that having access to the merged inductive relation can simplify proof developments, via a case study on proving the correctness of an efficient AVL tree search function.

3.1 Soundness and Completeness

Given two inductive relations $P : (T_{A1} \ \dots T : Prop)$ and $Q : (T_{B1} \ \dots T : Type)$ of the form described in the previous section, we showed how to produce an inductive relation PQ of type $T_{A1} \ \dots T_{B1} \dots \rightarrow T \rightarrow Prop$.

We can now state two theorems about the behavior of the derived relation PQ:

Theorem 3.1. *Soundness:* $\forall \ \overline{t_{Ai}} \ \overline{t_{Bj}} \ t, PQ \ \overline{t_{Ai}} \ \overline{t_{Bj}} \ t \ \rightarrow \ P \ \overline{t_{Ai}} \ t \ \wedge Q \ \overline{t_{Bj}} \ t$

Proof. By straightforward induction on the proof of PQ. □

Theorem 3.2. *Completeness:* $\forall \ \overline{t_{Ai}} \ \overline{t_{Bj}} \ t, P \ \overline{t_{Ai}} \ t \ \wedge Q \ \overline{t_{Bj}} \ t \rightarrow PQ \ \overline{t_{Ai}} \ \overline{t_{Bj}} \ t$

Proof. By induction on the proof of P, followed by a nested induction on the proof of Q, and finally using the inductive definition of PQ. □

Proof Script Details. [2] While the high level structure of the proof of soundness is fairly simple, the proof of completeness has a nested induction which requires additional low-level manipulations of the context in the general case.

For concreteness, let's revisit our AVL example from the introduction. First, we merge the `bst` and `bal` inductive relations:

```
Merge (fun t => bst lo hi t) With (fun t => bal n t) As AVL.
```

Then we can state and prove the soundness theorem:

[2]This subsection can be safely skipped by a reader who is not interested in low-level details of Coq proofs.

Algorithm 2. Merging Two Inductive Relations

Input Two lists of constructor representations for P and Q
Output A list of constructor representations for PQ

$PQ := []$
for each $c = (v, rs, os, as', t) \in P$ **do**
  if $rs = [P \ as \ t]$ **then**
    $bs := u_Q$ fresh variables
    $PQ := PQ \cup (v + bs, [PQ \ as \ bs \ t], os, as' + bs, t)$
    $P := P/c$
for each $c = (v, rs, os, bs', t) \in Q$ **do**
  if $rs = [Q \ bs \ t]$ **then**
    $as := u_P$ fresh variables
    $PQ := PQ \cup (v + as, [PQ \ as \ bs \ t], os, as + bs', t)$
    $Q := Q/c$
for each $(c_P, c_Q) \in P \times Q$ **do**
  $PQ := PQ \cup$ `merge_constructors` $c_P \ c_Q$
return PQ

```
Theorem AVL_sound :
  forall lo hi n t, AVL lo hi n t -> bst lo hi t /\ bal n t.
Proof. merge_sound. Qed.
```

...and the completeness theorem:

```
Theorem AVL_complete :
  forall lo hi t, bst lo hi t -> forall n, bal n t -> AVL lo hi n t.
Proof. merge_complete. Qed.
```

Focusing on the details of the completeness proof, after the first induction on `bst` and context manipulation, we're left (amongst other things) with a hypothesis of type `bal n (Node x l r)` that we would ideally want to induct on. However, as seasoned Coq users should expect at this point, the fact that the tree is not a variable but a concrete `Node` constructor stands in the way—we first need to generalize it but `remember` its shape. This is a standard trick [Pierce et al. 2018] when a straightforward proof is all that is required. However, we wanted to provide general proof scripts (`merge_sound` and `merge_complete`) to discharge all soundness and completeness theorems on merged relations.

To that end we turned to metaprogramming: we wrote a wrapper around the induction tactic (in OCaml), that first walks down the arguments of the hypothesis to be inducted upon and generalizes any arguments that are not abstract variables. Armed with this `remember_induct` tactic, we were able to construct the desired proof scripts, and discharge all soundness and theorems that we encountered.

3.2 Case study

The first indication that the merged inductives lend themselves better to reasoning is the difference in complexity of the soundness and completeness proofs: establishing the conjunction of the two original relations from the merged one is a straightforward induction, but the other way around requires nested induction and tedious low-level context manipulation. To further explore their effectiveness, we turn to our running AVL tree example and attempt to prove the correctness of an efficient search.

First, we specify tree membership with a straightforward traversal of the entire tree:

```
Fixpoint member (x : nat) (t : Tree) : bool :=
  match t with
  | Leaf => false
  | Node x' l r => (x =? x') || member x l || member x r
  end.
```

Then we write a version that relies on the search tree invariant to only search in one of the two subtrees of the node, and we include a fuel to allow for reasoning about the upper bound of recursive calls that need to be performed:

```
Fixpoint bst_search (n : nat) (x : nat) (t : Tree) : bool :=
  match n with
  | O => false
  | S n' =>
    match t with
    | Leaf => false
    | Node x' l r => if x <? x' then bst_search n' x l
                     else if x' <? x then bst_search n' x r
                     else true
```

```
    end
  end.
```

Finally, we state (and prove) our desired correctness theorem, that the efficient search agrees with `member` with a minimal amount of fuel:

```
Theorem bst_bal_search_member :
  forall n lo hi x t,
    bst lo hi t -> bal n t -> lo < x -> x < hi ->
    bst_search n x t = member x t.
```

We can also state the same theorem with a single precondition using `AVL`.

Just like when proving soundness and completeness of the merged relation, proving directly with `AVL` as the hypothesis allows for a straightforward inductive proof, while having both `bst` and `bal` requires a nested induction and similar low-level context manipulation. Alternatively, we could apply the completeness theorem first to simplify the rest of the proof. Overall, the merged inductive relation gives rise to an inductive hypothesis that lends itself to proof terms with a simpler recursive structure than the conjunction of the two original inductive relations.

4 EVALUATION

In this section, we demonstrate that using a merged inductive relation gives a significant performance boost to generation. We first demonstrate that the throughput of derived generators can increase by orders of magnitude through three case studies, using AVL trees, Red-Black Trees, and linear well-typed terms. Then, we show that our algorithm (and proofs) largely give useful results by merging a series of list-based inductive relations.

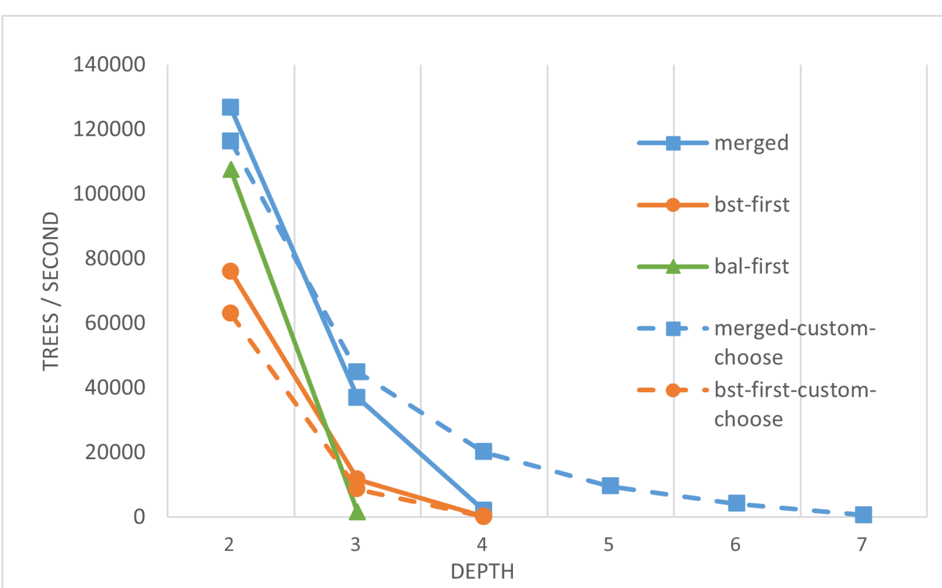

Fig. 1. Throughput of valid AVL tree generation.

4.1 Case study: AVL trees

Consider once again the example of AVL trees from the introduction. Using QuickChick, we derived three generators for AVL trees: one using the merged AVL relation, and two which generated terms satisfying one of `bst` or `bal` and checked against the other. Each generated tree t satisfies `bst 0 1000 t` and `bal d t` for some depth d. In Figure 1, we plot the number of (valid) trees that are successfully generated per second as a function of this depth. At a high enough depth, even after 100000 attempts, each generator failed to produce any trees, and we include data points up to the maximum depth which worked for each generator.

Particularly interesting is QuickChick's treatment of the inequality generation: generating `x` such that `lo < x < hi` for given `lo` and `hi`. The default generator for this inductive relation skews heavily towards the low numbers which leads to less than ideal coverage of the input space. However, QuickChick leverages its flexible typeclass infrastructure to provide a simple yet more effective such generator: `choose (lo, hi)`, which uniformly distributes `x` in the desired range. To provide a detailed account of the generator's performance, we show the throughput of the merged inductive relation both with, and without this improvement. This improvement could also affect `bst`-first generation so we show that combination as well, although in practice it doesn't help much.

We find that the merged inductive relation performs better than the generate-and-test variants even without using `choose`. More importantly, when using the standard `choose` combinator, the derived generator for merged AVL trees can generate thousands of AVL trees per second of depth up to 7. [3] In contrast, the generate-and-check variants are essentially unable to generate non-trivial trees, as even at depth 4 a random bst won't be balanced, and a random balanced tree won't satisfy the search invariant.

[3]Due to the balance requirement, every depth increment roughly doubles the size of trees that are being generated, which becomes the bottleneck after a while.

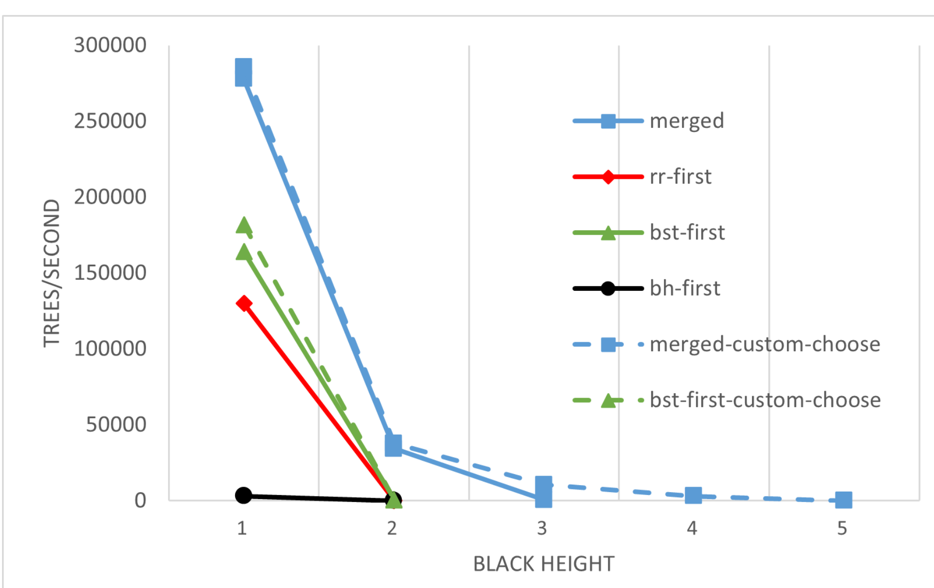

Fig. 2. Throughput of various generators for red-black trees.

```
(* Colors and Trees *)
Inductive color :=
| red : color
| black : color.

Inductive tree :=
| leaf : tree
| node : color -> nat -> tree -> tree -> tree.

(* No red node has a red child *)
Inductive rr : color -> tree -> Prop :=
| rbt_leaf : forall c, rr c leaf
| rbt_black_node : forall c1 c2 t1 t2 n,
    rr c1 t1 -> rr c2 t2 -> rr black (node black n t1 t2)
| rbt_red_node : forall t1 t2 n,
    rr black t1 -> rr black t2 -> rr red (node red n t1 t2).

(* Enforces the black height of the tree *)
Inductive bh : nat -> tree -> Prop :=
| bh_leaf : bh 1 leaf
| bh_red_node : forall t1 t2 h n,
    bh h t1 -> bh h t2 -> bh h (node red n t1 t2)
| bh_black_node : forall t1 t2 h n,
    bh h t1 -> bh h t2 -> bh (S h) (node black n t1 t2).
```

Fig. 3. Red-black tree inductive definitions

```
Merge (fun t => rr c t) With (fun t => bh c t) As red_black.
Merge (fun t => red_black color height t) With (fun t => bst lo hi t) As rbt.

Inductive rbt : color -> nat -> nat -> nat -> tree -> Prop :=
| rbt_leafbh_leafbst_leaf : forall lo hi c,
      rbt c 1 lo hi leaf
| rbt_black_nodebh_black_nodebst_node : forall lo hi x l r h c1 c2,
    lo < x < hi -> rbt c1 h lo x l ->
    rbt c2 h x hi r -> rbt black (S h) lo hi (node black x l r)
| rbt_red_nodebh_red_nodebst_node : forall lo hi x l r h,
    lo < x < hi -> rbt black h lo x l ->
    rbt black h x hi r -> rbt red h lo hi (node red x l r).
```

Fig. 4. Merged red-black tree definition.

4.2 Case study: red-black trees

A *red-black* tree is a binary tree where each node has a color and a number, satisfying three conditions: (1) no red node has a red child, (2) every path from a root to a leaf goes through the same number of black nodes (its *black height*), and (3) the tree satisfies the search tree invariant.

We can represent each of these three properties with an inductive relation. Figure 3 shows the inductive definitions involved (`bst` is elided for brevity as it is almost identical to the one earlier in the paper, with the exception that the `Node` constructor now takes an additional `color` argument).

We can merge all three of these relations together by invoking the merging algorithm twice, resulting in the code shown in Figure 4.

From these relations, we derived four generators: one using the merged `rbt` relation, and three which generate elements of one relation and check against the other two. In Figure 2, we plot the throughput of the various generators as a function of the black height of trees generated. Once again, we include all heights for which generators were able to produce any trees within 100000 attempts, and include the performance with and without `choose`. We find an even more substantial performance increase here: only the merged generator had any hope of producing non-trivial red-black trees of black height greater than two.

4.3 Case study: Typed and linear STLC terms

Another very common application of QuickChick is to test language developments, such as type system implementations, interpreters, or compilers. To test such systems effectively, given a typing relation in inductive form QuickChick can generate an efficient generator that only produces well-typed terms [Paraskevopoulou et al. 2022]. But once again, when multiple constraints need to be imposed (e.g. linearity—that functions use their arguments exactly once [Wadler 1990]), generators are once again found to be lacking.

For this case study, we implemented a typing judgment for the simply-typed lambda calculus as an inductive relation, as well as an inductive relation that encodes linearity of terms. We then merged them using our algorithm and evaluated the performance of different derived generators. The full code is quite large, but can be found in Appendix A.

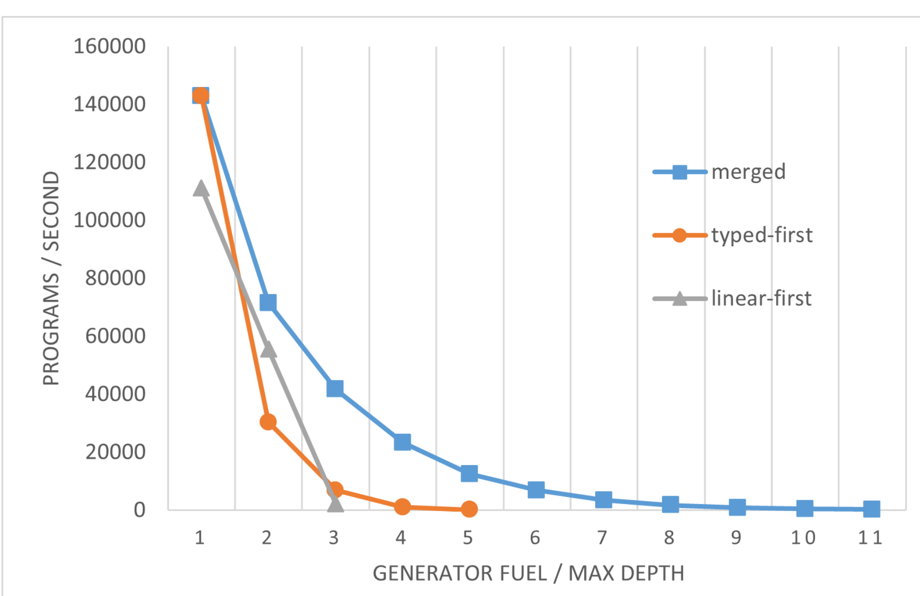

Fig. 5. Throughput of valid lambda term generators.

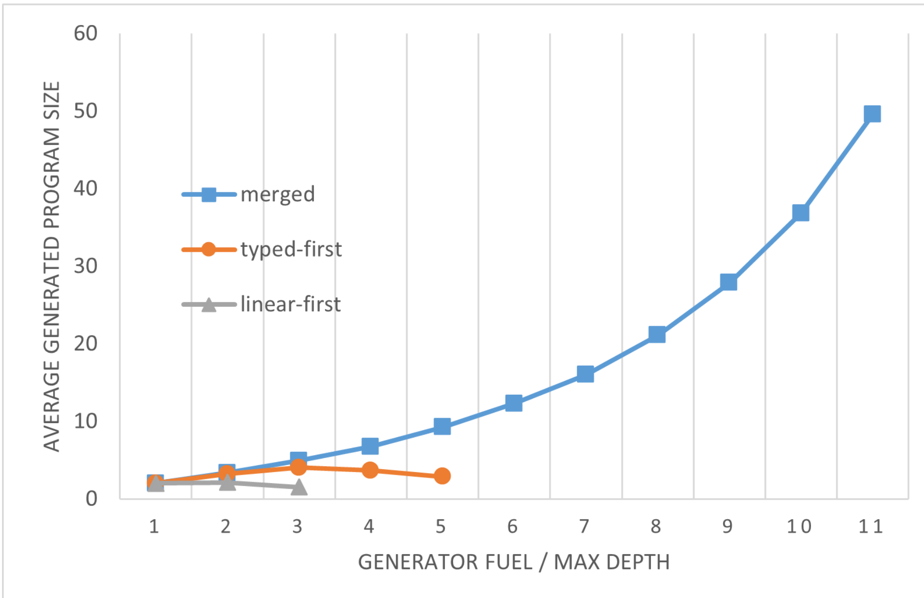

Fig. 6. Average size of lambda term generators

Just like in the previous case studies, we evaluated the performance of the generator that used the merged relation, as well as that of generators that generate for a single relation and check against the other. Unlike the previous case studies, we don't have a parameter of the relation to enforce the size of the generated program. Rather, we have to rely on QuickChick's fuel parameter to limit the maximum generation depth. To account for that, Figure 5 shows the throughput of the different generators as a function of this fuel, while Figure 6 plots the average size of the generated lambda terms.

The generator derived from the merged relation has no problem generating programs of an arbitrarily large size, given enough fuel and time. In contrast, as expected, the generate-and-test generators were not very efficient when trying to generate larger programs.

4.4 Case study: A variety of relations on lists

As discussed earlier, the task of producing a relation equivalent to the conjunction of two given relations could be trivially solved by simply returning the conjunction of the two relations. If a merged relation is to be useful, it needs to actually combine the constructors of the two relations. However, even this is not sufficient to guarantee that the merged result is useful; some of the resulting constructors may refer to both of the input relations! In that case, generating an element of the merged relation requires solving a sub-problem of generating an element satisfying both of the two relations anyway.

We attempt to quantify how often merging produces useful results by a percentage of constructors which reference at most one of P, Q, or PQ, by merging a variety of inductive predicates over lists. The definitions of the relations can be found in Appendix B.

Most natural inductive relations on lists turn out to merge well with each other, as shown in Figure 7, with an exception being permutations. In general, inductive relations which are defined recursively over the data will tend to merge together well. However, the definition of permutations

	sorted	prefix	suffix	sublist	permutation
sorted	100%	100%	100%	100%	86%
prefix	100%	100%	100%	100%	80%
suffix	100%	100%	100%	100%	86%
sublist	100%	100%	100%	100%	83%
permutation	86%	80%	86%	83%	75%

Fig. 7. Percentage of constructors which don't reference original relations.

has a transitivity constructor, which does not follow the recursive structure of the list, and is therefore a bad candidate for merging.

4.5 Limitations

The merging algorithm we presented in this paper can greatly speed up testing in scenarios where multiple inductive relations constrain the same piece of data. However, while given two such relations we can automatically derive a single one that is equivalent to their conjunction, and while QuickChick can automatically generate terms satisfying the generated relation, a human user still needs to identify a situation where the tool is useful. This paper develops a useful tool in the arsenal of an experienced property-based testing user, and a crucial building block for future work towards efficient, fully automatic testing of arbitrary Coq conjectures.

At the same time, relying on QuickChick for generating inputs on the merged inductive means that our approach inherits some of QuickChick's limitations. In particular, the order that constraints appear in an inductive constructor matters [Paraskevopoulou et al. 2022], as it dictates which universally quantified variables will be generated first. Using our algorithm to merge relations before deriving a generator adds another layer of indirection which can make dealing with any problems that arise from the order of constraints even more difficult.

Finally, our merging algorithm relies on identifying shared recursive structure between constructors. This structure exists for many, but not all pairs of inductive relations as the permutations example shows. Still, in the not uncommon cases where recursive structure *is* shared, the benefits are substantial.

5 RELATED WORK

Generating Test Inputs Satisfying Multiple Constraints. Random generation of inputs lies at the core of property-based testing and has been thoroughly studied since the emergence of Haskell QuickCheck [Claessen and Hughes 2000], both in the form of handwritten random generators for particularly challenging constraints [Hritcu et al. 2016; Midtgaard et al. 2017; Pałka et al. 2011; Yang et al. 2011] and as a general problem for automatically deriving such generators from a language of constraints [Bulwahn 2012b; Claessen et al. 2015; Fetscher et al. 2015; Lampropoulos et al. 2017, 2018]. Our work falls squarely in that last category, as we're building on top of the work of [Paraskevopoulou et al. 2022] to dramatically improve their generator performance when multiple inductive relations constrain the same piece of data.

Prior work also encountered the same complication. In particular, Lampropoulos et al. [2017] proposes a domain specific language for specifying generators as lightly annotated functional predicates that allow for explicitly delaying the instantiation of variables so that multiple different constraints can be taken into account. Their approach is much more modular, but is quite slow, reporting 35x overheads compared to handwritten generators.

In a different line of work, Claessen et al. [2015] exploit laziness to generate inputs satisfying Haskell predicates, by pruning large parts of the search space as soon as possible. In their work, they identify a parallel conjunction operator which allowed for exploring both predicates in a conjunction to more efficiently prune the search space. These generators can be quite effective when there is natural laziness to be exploited, but provide little benefit otherwise. Moreover, it is unclear how such an approach could translate to the strict setting of proof assistants like Coq.

Ornaments and Modularization. Another related line of work is that of Ko and Gibbons [Ko and Gibbons 2011, 2016]. Their goal is rather different: to make internalist representations of datatypes (where constraints are intrinsically part of the datatype such as vectors of a particular size) as easy to extend and manipulate as externalist representations (such as pairs of a list and a predicate constraining its length). They use ornaments [Dagan and McBride 2014] as the foundation of such predicates and introduce the notion of parallel composition of ornaments to address multiple refinements on data in a compositional manner. Instead, our work intends to construct a single representation of multiple constraints using only traditional inductive relations as inputs, and stays within the confines of the Coq proof assistant and its established ecosystem.

6 CONCLUSION AND FUTURE WORK

In this paper we identified a problem with prior work on deriving generators for data satisfying constraints in the form of inductive relations: when multiple constraints are imposed on the same piece of data, existing algorithms can fail to generate nontrivial values. We introduced an algorithm that addresses this problem by merging multiple inductive relations into one, leading to more effective generation and simpler proving.

One avenue of future work is further integrating our tool inside QuickChick's automated workflow. Currently, it is up to the user to identify a situation where this merging is needed and invoke our tool. It would be interesting to explore opportunities for QuickChick to automatically identify such cases leveraging the flexible typeclass mechanism of Coq.

ACKNOWLEDGMENTS

Zoe Paraskevopoulou, Alex Kavvos, Proof Engineering Grant

REFERENCES

Lukas Bulwahn. 2012a. 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

Lukas Bulwahn. 2012b. 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

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

Koen Claessen, Jonas Duregård, and Michal H. Palka. 2015. Generating constrained random data with uniform distribution. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S0956796815000143

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

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

Pierre-Évariste Dagan and Conor McBride. 2014. Transporting functions across ornaments. *Journal of Functional Programming* 24, 2-3 (2014), 316–383. https://doi.org/10.1017/S0956796814000069

Carl Eastlund. 2009. DoubleCheck Your Theorems. In *ACL2*. http://www.ccs.neu.edu/scheme/pubs/acl209-e.pdf

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/

Catalin Hritcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo de Amorim, Maxime Denes, John Hughes, Benjamin C. Pierce, and Dimitrios Vytiniotis. 2016. Testing Noninterference, Quickly. In *Journal of Functional Programming (JFP)*.

Hsiang-Shang Ko and Jeremy Gibbons. 2011. Modularising inductive families. In *WGP@ICFP*.

Hsiang-Shang Ko and Jeremy Gibbons. 2016. Programming with ornaments. *Journal of Functional Programming* 27 (12 2016). https://doi.org/10.1017/S0956796816000307

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

Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li yao Xia. 2017. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, (POPL)*.

Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating Good Generators for Inductive Relations. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*.

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

Fredrik Lindblad. 2007. Property Directed Generation of First-Order Test Data. In *8th Symposium on Trends in Functional Programming (Trends in Functional Programming, Vol. 8)*. Intellect, 105–123. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.2439&rep=rep1&type=pdf

Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-Driven QuickChecking of Compilers. *Proc. ACM Program. Lang.* 1, ICFP, Article 15 (aug 2017), 23 pages. https://doi.org/10.1145/3110259

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.

Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST '11)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615

Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing Correctly with Inductive Relations. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*.

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

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

Philip Wadler. 1990. Linear Types can Change the World!. In *Programming Concepts and Methods.*

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011.* 283–294. https://doi.org/10.1145/1993498.1993532

A TYPED AND LINEAR TERMS

In this Appendix we show the inductive relations for which we evaluated generation performance in Section 4.3. `Term` defines a grammar of programs and `Ty` defines their types, including a single base type `number`. `Var` and `typed` together define the typing rules for the language.

`linear` defines terms which use each variable exactly once. It is parameterized by a list of booleans, which represents whether each free variable in the context is used in this term. For closed terms, this list should be `nil`. In our evaluation, we generated terms `t` satisfying `typed nil number t` and `linear nil t`.

```
Inductive Term : Type :=
| var : nat -> Term
| app : Term -> Term -> Term
| lam : Term -> Term
| const : nat -> Term
| add : Term -> Term -> Term.

Inductive Ty : Type :=
| arr : Ty -> Ty -> Ty
| number : Ty.

Inductive Var : list Ty -> Ty -> nat -> Prop :=
| zero : forall t g, Var (cons t g) t 0
| suc : forall a b g n, Var g a n -> Var (cons b g) a (S n).

Inductive typed : list Ty -> Ty -> Term -> Prop :=
| t_var : forall g n t,
    Var g t n -> typed g t (var n)
| t_app : forall a b g e1 e2,
    typed g (arr a b) e1 -> typed g a e2 ->
    typed g b (app e1 e2)
| t_lam : forall a b g e,
    typed (cons a g) b e -> typed g (arr a b) (lam e)
| t_const : forall n g, typed g number (const n)
| t_add : forall e1 e2 g,
    typed g number e1 -> typed g number e2 ->
    typed g number (add e1 e2).

Inductive falses : nat -> list bool -> Prop :=
| Falses_nil  : falses 0 nil
| Falses_cons : forall n l, falses n l -> falses (S n) (cons false l).

Inductive nand : bool -> bool -> bool -> Prop :=
| nand_ff : nand false false false
| nand_ft : nand false true true
| nand_tf : nand true false true.

Inductive combine : list bool -> list bool -> list bool -> Prop :=
| combine_nil : combine nil nil nil
```

```
| combine_cons : forall as1 as2 as3 a1 a2 a3, nand a1 a2 a3
    -> combine as1 as2 as3 -> combine (cons a1 as1) (cons a2 as2) (cons a3 as3).

Inductive var_linear : list bool -> nat -> Prop :=
| zero_lin : forall n l, falses n l -> var_linear (true :: l) 0
| suc_lin : forall u n, var_linear u n -> var_linear (cons false u) (S n).

Inductive linear : list bool -> Term -> Prop :=
| l_var : forall u_ n_, var_linear u_ n_ -> linear u_ (var n_)
| l_app : forall u1_ u2_ u3_ e1_ e2_,
    combine u1_ u2_ u3_
    -> linear u1_ e1_ -> linear u2_ e2_
    -> linear u3_ (app e1_ e2_)
| l_con  : forall len_ n_ l,
    falses len_ l -> linear l (const n_)
| l_lam : forall u_ e_, linear (true :: u_) e_ -> linear u_ (lam e_)
| l_add : forall u1_ u2_ u3_ e1_ e2_,
    combine u1_ u2_ u3_ -> linear u1_ e1_ -> linear u2_ e2_
    -> linear u3_ (add e1_ e2_).
```

The two relations, `typed` and `linear`, can be merged using our plugin:

```
Merge (fun t => typed gamma ty t) With (fun t => linear used t) As typed_and_linear.
Merge (fun t => linear used t) With (fun t => typed gamma ty t) As linear_and_typed.
```

Which produces a merged relation:

```
Inductive linear_and_typed : list bool -> list Ty -> Ty -> Term -> Prop :=
| l_addt_add : forall e1 e2 g u1_ u2_ u3_,
      combine u1_ u2_ u3_ -> linear_and_typed u2_ g number e2 ->
      linear_and_typed u1_ g number e1 -> linear_and_typed u3_ g number (add e1 e2)
| l_lamt_lam : forall a b g e u_,
    linear_and_typed (true :: u_) (a :: g) b e ->
    linear_and_typed u_ g (arr a b) (lam e)
| l_cont_const : forall n g len_ l,
    falses len_ l -> linear_and_typed l g number (const n)
| l_appt_app : forall a b g e1 e2 u1_ u2_ u3_,
    combine u1_ u2_ u3_ -> linear_and_typed u2_ g a e2 ->
    linear_and_typed u1_ g (arr a b) e1 -> linear_and_typed u3_ g b (app e1 e2)
| l_vart_var : forall g n t u_,
    Var g t n -> var_linear u_ n -> linear_and_typed u_ g t (var n).
```

B LIST RELATIONS

In this Appendix we detail the inductive relations over lists which we evaluated the merging properties of in Section 4.4. `sorted` and `permutation` are taken directly from Software Foundations [Pierce et al. 2018]. In each relation with multiple inputs, we merged on the first index.

These relations also provide an example of nontrivial unification of shared parameters: we can merge `sorted` with the other relations even though its lists are parameterized by a `nat` while the others' are parameterized by a generic type parameter `t`.

```
Inductive sorted : list nat -> Prop :=
| sorted_nil : sorted nil
| sorted_1 : forall x, sorted (cons x nil)
| sorted_cons : forall x y l,
    le x y -> sorted (cons y l) -> sorted (cons x (cons y l)).

Inductive prefix {t : Type} : list t -> list t -> Prop :=
| prefix_nil : forall l, prefix nil l
| prefix_cons : forall l1 l2 t,
    prefix l1 l2 -> prefix (t :: l1) (t :: l2).

Inductive suffix {t : Type} : list t -> list t -> Prop :=
| suffix_same : forall l, suffix l l
| suffix_skip : forall l1 l2 t,
    suffix l1 l2 -> suffix (t :: l1) l2.

Inductive sublist {t : Type} : list t -> list t -> Prop :=
| sublist_nil : sublist nil nil
| sublist_skip : forall l1 l2 x,
    sublist l1 l2 -> sublist l1 (x :: l2)
| sublist_both : forall l1 l2 x,
    sublist l1 l2 -> sublist (x :: l1) (x :: l2).

Inductive permutation {t : Type} : list t -> list t -> Prop :=
| perm_nil :
   permutation nil nil
| perm_skip :
   forall x l l', permutation l l' -> permutation (x :: l) (x :: l')
| perm_swap :
   forall x y l, permutation (y :: x :: l) (x :: y :: l)
| perm_trans :
   forall l l' l'', permutation l l' -> permutation l' l'' -> permutation l l''.
```