How to do Proofs

Practically Proving Properties about Effectful Programs’ Results (Functional Pearl)

Koen Jacobs  
KU Leuven  
Belgium  
koen.jacobs@cs.kuleuven.be

Andreas Nuyts  
KU Leuven  
Belgium  
andreas.nuyts@cs.kuleuven.be

Dominique Devriese  
Vrije Universiteit Brussel  
Belgium  
dominique.devriese@vub.be

Abstract
Dependently-typed languages are great for stating and proving properties of pure functions. We can reason about them modularly (state and prove their properties independently of other functions) and non-intrusively (without modifying their implementation). But what if we are interested in properties about the results of effectful computations? Ideally, we could keep on stating and proving them just as nicely.

This pearl shows we can. We formalise a way to lift a property about values to a property about effectful computations producing such values, and we demonstrate that we need not make any sacrifices when reasoning about them. In addition to this modular and non-intrusive reasoning, our approach offers independence of the underlying monad and allows for readable proofs whose structure follows that of the code.

CCS Concepts  • Theory of computation → Type theory; Equational logic and rewriting; • Software and its engineering → Formal software verification.

Keywords  Agda, dependently-typed programming, effectful, extrinsic proofs, monad, functor, applicative, strong specification, equational reasoning

ACM Reference Format:
The ACM Reference Format can be found at https://www.acm.org/publications/proceedings-reference-format.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format: 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6815-5/19/08...
https://doi.org/10.1145/3331554.3342603

1 Some Pseudo-Proofs about Effects
Imagine that we are developing a Monopoly game where players frequently throw a pair of dice. A single die roll is at most six, and the game rules sometimes rely on the fact that a pair of dice can never be greater than twelve.

In a dependently-typed language, we can prove such facts. That is, if natural numbers \( x \) and \( y \) are not greater than 6, we can prove that \( x + y \) is not greater than 12. To do this, we define a function, say \( \text{sumOfBound6IsBound12} \), of type \( \{ x : \mathbb{N} \} \to x \leq 6 \to \{ y : \mathbb{N} \} \to y \leq 6 \to x + y \leq 12 \), which takes proofs that \( x \leq 6 \) and \( y \leq 6 \), and returns a proof that \( x + y \leq 12 \).

However, in our Monopoly implementation, we are not interested in this property for individual rolls, but as a property over the results of the effectful operation \( \text{twoDice} \), which is implemented in terms of an underlying operation \( \text{die} \):

\[
\text{die} : \text{IO} \mathbb{N} \\
\text{die} = \ldots \\
\text{twoDice} : \text{IO} \mathbb{N} \\
\text{twoDice} = \text{do } x \leftarrow \text{die} \\
\quad \quad \quad \text{y} \leftarrow \text{die} \\
\quad \quad \quad \text{return } (x + y)
\]

That is, we are interested in proving that \( \text{twoDice} \) returns, upon execution, a natural number not greater than 12.

1 The type constructor \( \leq \) maps natural numbers \( a \) and \( b \) to the type \( a \leq b \) representing the proposition that \( a \) is at most \( b \). Following the Curry-Howard correspondence [12], the type \( a \leq b \) is inhabited if this proposition is true, and its inhabitants represent proofs of the proposition.

2 All code samples that appear here, are written in the dependently-typed functional programming language Agda [11]. The Agda files can be found at https://github.com/scaup/agda-liftprop.

3That is, upon execution, \( \text{die} \) will return a pair in which the first component is a natural number, and the second component a proof that the first component is not greater than 6. More concisely, it will return a pair of the form \( (n, \rho) \) where \( n : \mathbb{N} \) and \( \rho : n \leq 6 \). Such pairs are exactly the inhabitants of the type \( \Sigma[ n \in \mathbb{N} ] n \leq 6 \).

4To aid the reader, we sometimes annotate functions and variables with their type like \( \text{io die :: IO} \mathbb{N} \). These superscripts are not part of the code.
die : IO (Σ[n ∈ N] n ≤ 6)
die = . . .
twoDice : IO (Σ[n ∈ N] n ≤ 12)
twoDice =
  do (x : N, px : N) ← die : IO (Σ[n ∈ N] n ≤ 6);
    (y : N, py : N) ← die : IO (Σ[n ∈ N] n ≤ 6);
  return ((x + y) : N, (sumOfBound6IsBound12 px py) : N)

This approach is referred to as strong specification [13]:
the type IO (Σ[n ∈ N] n ≤ 12) of this reimplemented
twoDice captures the desired property about the function’s
result by making it return a proof along with the result. The
function itself needs to be modified to return this proof and
so the proof of the property is baked into the new implementa-
tion.

However, modifying the original code to use strong speci-
fication is not always desirable. For one, the code becomes
more verbose, unnecessarily so for readers who do not care
about the property. Also, we run the risk of introducing new
bugs, simply because we have to modify the original code to
return the extra proof. Furthermore, proving several prop-
erties about a single function is only possible by including
them all in the original definition.

Because of these downsides, we would prefer to avoid
reimplementing die and twoDice. Instead, we would like
to leave their original versions of type IO N untouched and
prove their properties extrinsically, in a separate defini-
tion.5 Let us imagine what such a solution could look like.

Dreaming up a better solution Of course, we cannot sim-
ply write twoDice ≤ 12 as λ a → a ≤ 12 is a predicate6 on N,
not IO N. This suggests that what we need is an operator,
say Lift, which takes a predicate on N, say λ a → a ≤ 12,
and returns the desired, lifted predicate on IO N. That is, for
an ioa : IO A, the type Lift ioa encodes the proposition
that ioa returns – upon execution – an a : A that satisfies P.

But even if we had such a Lift operator, how would we
prove such a proposition? More concretely, given an inhab-
itant dieBound6 of type Lift (λ a → a ≤ 6) die, how can we
prove Lift (λ a → a ≤ 12) twoDice? One way to think
about this is that intuitively, Lift (λ a → a ≤ 6) die is true
because die could be reimplemented as a computation of
type IO (Σ[n ∈ N] n ≤ 6). Analogously, Lift (λ a → a ≤ 12)
twoDice is true intuitively because twoDice could be
reimplemented at type IO (Σ[n ∈ N] n ≤ 12). Perhaps,
even though Lift is obviously not a monad, it would be nice

5Completely analogous to the pure case, we defined sumOfBound6IsBound12
extrinsically instead some intrinsic specification of addition, which would
have the type Σ[n ∈ N] n ≤ 6 → Σ[n ∈ N] n ≤ 6 → Σ[n ∈ N] n ≤ 12.
6A predicate, say P, on a type A is a function of type A → Set. That is, for
each inhabitant of A, it returns a type P a. For example, the predicate λ a → a ≤ 12
on N takes a natural number n, and maps it to the proposition
n ≤ 12.

if we could write a proof of Lift (λ a → a ≤ 12) twoDice in a
form of do-notation, where we essentially replicate this hypo-
thesitical reimplemention, only now in terms of dieBound6.

dieBound6 : Lift (λ a → a ≤ 6) die

dieBound6 = . . .
twoDiceBound12 : Lift (λ a → a ≤ 12) twoDice

twoDiceBound12 =
  do (x : N, px : N) ← dieBound6
    (y : N, py : N) ← dieBound6
  return (x + y, sumOfBound6IsBound12 px py)

Ideally, the proof twoDiceBound12 should not depend on
the precise implementation of die, nor on that of dieBound6.
For example, we should also be able to write the following:

twoTimes : IO N → IO N
twoTimes die = do x ← die
  return (x + y)
dieBound6twoTimesBound12 :
  (die : IO N) → Lift (λ a → a ≤ 6) die →
  Lift (λ a → a ≤ 12) (twoTimes die)
dieBound6twoTimesBound12 dieBound6 =
  do (x , px) ← dieBound6
    (y , py) ← dieBound6
  return (x + y, sumOfBound6IsBound12 px py)

This modularity is important to accommodate situations
where die and dieBound6 are implemented independently
and abstractly by another party, in some other module; in
such a scenario, changes to the internals of die : IO N or
dieBound6 : Lift (λ n → n ≤ 6) die should not affect the
validity of the proof twoDiceBound12.

In principle, the above pseudo-proof also does not really
depend on the fact that we are in the IO monad. For example,
why shouldn’t we be able to use the same approach for
computations in the List monad:

twoTimesList : List N → List N
twoTimesList dieList = do x ← dieList
  return (x + y)
dieBound6twoTimesListBound12 :
  (dieList : List N) → Lift (λ a → a ≤ 6) dieList →
  Lift (λ a → a ≤ 12) (twoTimesList dieList)
dieBound6twoTimesListBound12 dieList dieListBound6 =
  do (x , px) ← dieListBound6
    (y , py) ← dieListBound6
  return (x + y, sumOfBound6IsBound12 px py)

Although the meaning of Lift changes for this other monad,
the intuitive reasoning does not change, so why should the
proofs?
Overview  It turns out that we can in fact implement the Lift operator sketched above, and in the next section, we explain how. Moreover, we can provide convenient abstractions to work with lifted properties, and we can even allow formal proofs that resemble the above pseudo-code quite closely. In §3, we will demonstrate this framework on some more elaborate examples. In §4, we broaden our view to applicative and arbitrary functors, zoom in on a particular kind of functors classified by an important property related to Lift, and prove some properties about List functions that we introduced in §3.2. Afterwards, in §5, we will see that our framework provides not only an elegant way to prove lifted properties, but also an interesting way to make use of these properties. Finally, we conclude in §6, where we summarise the advantages and limitations of our approach, and lay out some related and future work.

Remark  Although this paper’s subtitle is ‘Practically proving properties about effectful programs results’, we do not claim that our approach supports proving all such properties. Particularly, when a program’s result depends on the behavior of effectful operations, proving properties about this result will require other techniques for reasoning about the behavior of those effects. We encounter such an example in §3.1.

Note also that this paper is specifically about extrinsic or retroactive proofs. In other words, we want to keep proofs like dieBound6twoTimesBound12 separate from the implementation of twoTimes because of the advantages described before: twoTimes does not need to be modified, it remains readable to non-experts and we can easily formulate and prove many separate properties about it. The flip side of the choice for extrinsic proofs is a certain amount of repetition: dieBound6twoTimesBound12 repeats some of the structure of the underlying program twoTimes. However, this code duplication is not the same as that which is widely discouraged in software engineering courses, as there is no risk for inconsistencies bet-ween twoTimes and dieBound6twoTimesBound12. Any such inconsistency will be reliably detected and reported by the type-checker.

2 How Does It Work?
In this section, we formalise the notations from §1: the meaning of Lift (§2.1) and of the corresponding do-notation (§2.2).

2.1 Lifting a Predicate
So how can we define this hypothetical Lift operator? Consider, for example, a list of natural numbers dieList = [1, 2, 3, 4, 5, 6]. How can we encode the fact that every element in dieList is not greater than 6? Notice, again, that dieList could just as well have been implemented to be of type List (Σ n ∈ ℕ n ≤ 6):8
dieListWithProofs : List (Σ n ∈ ℕ n ≤ 6)
dieListWithProofs = [(1, p16), (2, p26), (3, p36), (4, p46), (5, p56), (6, p66)]

This dieListWithProofs corresponds to dieList in the sense that executing dieListWithProofs and then forgetting the returned proofs, is equivalent to the original dieList. More formally, we can show that fmap proj1 dieListWithProofs (i.e. executing dieListWithProofs and using proj1 : Σ n ∈ ℕ n ≤ 6 → ℕ to forget the proof) is equal to the original dieList.9
dieListProofsCorr : fmap proj1 dieListWithProofs ≡ dieList
dieListProofsCorr = refl

We can prove this equality using refl because fmap proj1 dieListWithProofs simply reduces to dieList. Intuitively, we can think of dieListWithProofs as a witness to the fact that every result of dieList is indeed at most 6.

Generalising this example, we can define the proposition Lift P fa as follows, for an arbitrary functor, say F, a type A, a predicate P on A, and a functorial value fa : FA:

record Lift (P : Predicate A) (fa : F A) : Set where

field witness : F (Σ A P)
corresponds : fa ≡ fmap proj1 witness

Lift P fa is a record type with two fields: witness of type F (Σ A P) and corresponds: a proof of equality between fa and fmap proj1 witness.

Coming back to our example, we now have the following:
dieListBound6 : Lift (λ a → a ≤ 6) dieList
dieListBound6 =

record {witness = dieListWithProofs
; corresponds = dieListProofsCorr}

Intuitively, the meaning of Lift is slightly different for different functors. To get a better feel for the full generality of Lift, let us consider some more examples.

For an IO-operation choosePassword : IO String, a proof that Lift (λ s → length s ≥ 20) choosePassword expresses that choosePassword will only produce passwords of length ≥ 20. For a non-deterministic operation range : ℕ → ℕ → List ℕ in the List monad, we can implement ∀ [n m] → Lift (λ x → (x ≤ m)) (range n m) and ∀ [n m] → Lift (λ x → (n ≤ x)) (range n m), to formalise that every element in range n m is between n and m. Considering a stateful

7When the implementation of a term of a specific type is irrelevant and distracting, we give it a generic name together with its type in superscript like P.
8For clarity, we use a Haskell inspired pseudo syntax to denote lists.
9Actually, we have proj1 : Σ A P → A for all types A and predicates P on A. That is, fixing some A and P, proj1 is just the function λ (a . p) → a discarding the second component of each pair.
The lifted bind operator \( \Rightarrow_{L} \) is more interesting. Consider an \( ma : M A \) and \( f : A \rightarrow M B \). If we have a proof of \( Lift \; P \; ma \) (for some predicate \( P \) over \( A \)), and if we know that for each \( (a, p) : \Sigma \; A \; P \), we can prove \( Lift \; Q \; f \; a \) (for some predicate \( Q \) over \( B \)), then \( \Rightarrow_{L} \) states that also \( Lift \; Q \; (ma \Rightarrow f) \):

\[
\Rightarrow_{L} \Downarrow \Delta \quad \text{corresponds to} \quad \Rightarrow_{L} \Downarrow \Delta
\]

A proof of the \( \Downarrow \Delta \) field is fairly long and technical. It boils down to the use of monadic laws, combined with the 

And of course we also have \( \Rightarrow_{L} \) as a special case of \( \Rightarrow_{L} \).

Now that we have defined these lifted operators, we can write down a concise proof for \( Lift \; (\lambda \; a \rightarrow a \leq 12) \) twoDice:

\[
\text{twoDiceBound12} : \Lambda \rightarrow \text{IO} \Downarrow \Delta \Rightarrow_{L} \Downarrow \Delta
\]

The \( \Downarrow \Delta \) field is now completely taken care of by our lifted operators! Notice also that the outline of our full proof is identical to that of the \( \Downarrow \Delta \) field in our first attempt; a programmer can just use these lifted operators, and pretend that he/she is doing a retroactive proof by strong specification.

But can’t we make this a bit more readable still? Instead of writing the \( \Rightarrow_{L} \) operator, can’t we just \textit{do} it? The \textit{do}-notation desugars recursively to expressions with \( \Rightarrow_{L} \) and \( \Rightarrow_{L} \) as described in [2]. It was developed with the classical monadic \( \Rightarrow_{L} \) and \( \Rightarrow_{L} \) operators in mind to restructure monadic compositions using these operators, making them more readable.

In Agda however, it is fully up to us which \( \Rightarrow_{L} \) and \( \Rightarrow_{L} \) we want to have in scope. By locally renaming \( \Rightarrow_{L} \) to \( \Rightarrow_{L} \), we can reuse the \textit{do}-notation to structure our proof more easily. We have the following.\(^\text{10}\)

open import Monads hiding (return, \( \Rightarrow_{L} \), \( \Rightarrow_{L} \))

open import LiftOperators

renaming (returnL to return;
\( \Rightarrow_{L} \) to \( \Rightarrow_{L} \);
\( \Rightarrow_{L} \) to \( \Rightarrow_{L} \);
\( \Rightarrow_{L} \) to \( \Rightarrow_{L} \))

twoDiceBound12 : Lift (\( \lambda a \rightarrow a \leq 12 \)) twoDice
twoDiceBound12 =

\(^{10}\)For consistency, we rename our returnL operator to return as well here.
3 Some More Elaborate Examples

So our framework can tackle the property lift \((\lambda \ a \rightarrow a \leq 12)\) twoDice. But does it also work for bigger, more realistic examples? To appreciate its generality, and to see how we could use it in practice, let us have a look at some more elaborate examples.

3.1 Tree Relabelling

Our first example is the stateful tree relabelling function from Hutton and Fulger [7]. Consider a straightforward Tree data type:

data Tree (A : Set) : Set where
leaf : (a : A) → Tree A
node : Tree A → Tree A → Tree A

Now consider the relabel function, which walks over the tree and replaces all node values with freshly-generated numbers:

\[
\begin{align*}
\text{relabel} : & \quad \text{Tree} \ A \rightarrow \text{State} \ N \ (\text{Tree} \ N) \\
\text{relabel} \ (\text{leaf} \ a) = & \quad \text{do} \ n \leftarrow \text{fresh} \\
& \quad \text{return} \ (\text{leaf} \ n) \\
\text{relabel} \ (\text{node} \ l \ r) = & \quad \text{do} \ l' \leftarrow \text{relabel} \ l \\
& \quad r' \leftarrow \text{relabel} \ r \\
& \quad \text{return} \ (\text{node} \ l' \ r')
\end{align*}
\]

Here, we have defined fresh as follows:

\[
\begin{align*}
\text{fresh} : & \quad \text{State} \ N \ N \\
\text{fresh} = & \quad \text{do} \ n \leftarrow \text{get} \\
& \quad \text{modify suc} \\
& \quad \text{return} \ n
\end{align*}
\]

The aim of this function is to relabel a tree with unique labels while leaving its shape unchanged. To establish that it fulfils this objective, we prove the following two lifted propositions:

- **Isomorphic**: Lift \((\lambda \ t' \rightarrow t' \equiv t)\) (relabel t) That is, given some input tree t : Tree A, any resulting tree should be isomorphic to it, where isomorphism is implemented in the obvious way:

\[
\begin{align*}
data \ \equiv_- : & \quad \text{Tree} \ A \rightarrow \text{Tree} \ B \rightarrow \text{Set} \\
\text{leafISO} : & \quad \text{leaf} \ a \equiv \text{leaf} \ b \\
\text{nodelsISO} : & \quad \text{tal} \equiv \text{tbl} \rightarrow \text{tar} \equiv \text{tbr} \\
\text{node tal tar} \equiv & \quad \text{node tbl tbr}
\end{align*}
\]

- **No duplicates**: Lift \(\text{NoDups} \ (\text{relabel} \ t)\) where \(\text{NoDups}\) is a predicate on Tree N such that \(\text{NoDups} \ \text{tree}\) encodes the property that \(\text{tree} \ : \ \text{Tree} \ N\) contains no duplicate values. That is, any tree resulting from \(\text{relabel} \ t\) does not contain duplicates.

Let us consider the isomorphism property. While it is often overlooked in the literature for being too trivial, formalising our intuition for it is not. In appendix B, we showcase an elementary but awkward proof to illustrate this. In our framework, however, it is a breeze:\footnote{From here on, we leave the renaming of our lifted operators implicit.}

\[
\begin{align*}
\text{relabel} : & \quad (t : \text{Tree} \ A) \rightarrow \text{Lift} \ (\lambda \ t' \rightarrow t' \equiv t) \ (\text{relabel} \ t) \\
\text{relabel} \ (\text{leaf} \ a) = & \quad \text{do} \ (n, _) \leftarrow \text{nothing2Prove fresh} \\
& \quad \text{return} \ (\text{leaf} \ n, \ \text{leafISO})
\end{align*}
\]

The above uses a simple helper function \(\text{nothing2Prove}\) of type \((\text{fa} : \text{F} \ A) \rightarrow \text{Lift} \ (\lambda \ a \rightarrow \top) \ (\text{fa})\), which proves a trivial predicate for the result of an arbitrary computation \(\text{fa} : \text{F} \ A\). The predicate \((\lambda \ a \rightarrow \top)\) is always satisfied as \(\top\) is the unit type with single inhabitant \(tt\).

Now for the no-duplicates property... Although we can state this property using \(\text{Lift}\), we cannot prove it with our lifted bind operators as our induction hypothesis is not strong enough. Indeed, given trees \(l' : \text{Tree} \ N\) and \(r' : \text{Tree} \ N\) that satisfy \(\text{NoDups}\), it is generally not the case that \(\text{node} \ l' \ r'\) satisfies \(\text{NoDups}\).

Of course, we could strengthen our hypothesis by also proving that the resulting tree is appropriately bounded with respect to the initial and final state. However, such a strengthened hypothesis cannot be expressed as a lifted property, as it does not merely refer to the computational output, but also to the computation’s effect on the state. Since Lift and our lifted operators are defined over arbitrary functor/monad, this naturally falls outside the scope of our approach. In §6, we briefly come back to this problem statement, and discuss how we can solve it by explicitly supporting the State monad.

3.2 n-Queens

A second larger example that we look at is a simple solver for the \(n\)-queens problem. Given a natural number \(n : \mathbb{N}\), the problem is to find all configurations of \(n\) queens on a chessboard of size \(n\) for which no two queens are attacking each other. As in standard chess, two queens attack each other if and only if they are on the same column, row, or diagonal.

We define a List computation \(\text{queenConfigs}\) of type \(\mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{List} \ \text{QConfig}\). Given a chessboard of size \(n : \mathbb{N}\) and an amount of queens \(k : \mathbb{N}\), \(\text{queenConfigs} \ n \ k : \text{List} \ \text{QConfig}\) returns a list of \(k\)-sized queen configurations. We represent queen configurations as lists of natural numbers, i.e. we write \(\text{QConfig}\) for \(\text{List} \ \mathbb{N}\). The encoding is depicted in Figure 1:
queens are placed in the $k$ leftmost columns and the $i$th number in the list represents the row number for the queen in column $i-1$.

As can be seen in Figure 1, our encoding as List $\mathbb{N}$ is quite loose. That is, many terms of type List $\mathbb{N}$ represent queen configurations that are actually illegal. On the flip side however, we can now first focus on defining our algorithm, instead of being forced to simultaneously prove a strong specification about it.

We first define a function `areNotAttacking` as follows. Given a queen configuration, say $qs : \text{QConfig}$, and the candidate row of a new rightmost queen, say $q : \mathbb{N}$, $qs$ `areNotAttacking` $q$ will evaluate to a boolean encoding what its name implies:

```haskell
areNotAttacking :: QConfig -> N -> Bool
areNotAttacking q = upwardDiagonal qCoordinate
  &\& fmap upwardDiagonal qsCoordinates
  &\& downwardDiagonal qCoordinate
  &\& fmap downwardDiagonal qsCoordinates
  &\& row qCoordinate
  &\& fmap row qsCoordinates

where
  qsCoordinates : List (N x N)
  qsCoordinates = toCoordinates qs
  qCoordinate : N x N
  qCoordinate = length qs , q
```

Herein, we have used the `toCoordinates` function defined as follows:

```haskell
toCoordinates :: List N -> List (N x N)
toCoordinates qs = zip (range 0 (length qs - 1)) qs
```

Additionally, we have also used the following functions `upwardDiagonal`, `downwardDiagonal`, and `row`:

```haskell
upwardDiagonal :: N x N -> N
upwardDiagonal (x , y) = x + y
```

```haskell
downwardDiagonal :: N x N -> Z
downwardDiagonal (x , y) = x - y
row :: N x N -> N
row (x , y) = y
```

These definitions are easily verified by glancing at Figure 2.

To find valid configurations, we can now simply recurse on the amount of queens we place leftmost on the board.

```haskell
queenConfigs :: N -> N -> List QConfig
queenConfigs n zero = return []
queenConfigs n (suc k) =
do qsExpanded <- queenConfigs n k
    qExpanded <- filter (\q -> qs areNotAttacking q) (range 0 (n - 1))
    return (qs + [q])
```

Let us now prove that every obtained configuration solves the $n$-queens problem, i.e. that it is both fitting and peaceful:

**Fitting** A configuration is fitting if every element in it is actually smaller than $n$, that is, each queen actually fits on the board. For example, the list $[0 , 6 , 3 , 2]$ from Figure 1 is not fitting as the queen in the second column falls outside the board.

**Peaceful** A configuration is peaceful, if no two queens are in the same column, row or diagonal. For example, the list $[0 , 6 , 3 , 2]$ from Figure 1 is not peaceful as the queens from the third and fourth column are attacking each other.

For a board of size $n$, the fact that a configuration fits the board is formalised as a predicate `Lift (\lambda q -> q \leq n - 1)` on `QConfig` which we denote as Fitting $n$. We can then prove that every obtained configuration in `queenConfigs n k` fits, that is, we prove `Lift (Fitting n) (queenConfigs n k)`:

```haskell
queenConfigsFit :: (n : N) -> (k : N) ->
Lift (Fitting n) (queenConfigs n k)
```
queenConfigsFit n zero =
    return (([], emptyConfigFits(Fitting n [])))
queenConfigsFit n (suc k) =
    do
        (qs, queenConfigs n k) ← queenConfigsFit n k
        (q, qFit) ←
            filterPreserves
                (λ q → q areNotAttacking q)
                (rangeUpBound 0 (n - 1))
        return (qs + [q], (qsFit +L [qFit])

Here, we have the function filterPreserves encoding the fact that lifted properties on lists are preserved upon filtering. The preserved property in this scenario is indeed the fact that every element in range 0 (n - 1) is smaller than n, which is encoded by the function rangeUpBound. We finally have to prove Fitting (qs + [q]). Remember now, that we defined Fitting n as Lift (λ q → q ≤ n - 1). So to prove Lift (λ q → q ≤ n - 1) (qs + [q]), we used the operator _+L_ of type Lift P xs → Lift P ys → Lift P (xs + ys) where xs is instantiated by qs and ys by [q], and the _[ ]_ lift operator of type P a → Lift P [a] where a is instantiated by q.

The usefulness of a configuration is formalised in the following predicate:

Peaceful : Predicate (QConfig)
Peaceful qs =
    NoDups (fmap upwardDiagonal coordinates) ×
    NoDups (fmap downwardDiagonal coordinates) ×
    NoDups (fmap row coordinates)
    where coordinates = toCoordinates qs

We also implement an addPeacefully function of the following type:

    (qs : QConfig) → (q : N) → Peaceful qs →
    (qs areNotAttacking q ≡ true) → Peaceful (qs + [q])

Its implementation is unimportant here, so we omit it.

A proof that every obtained configuration in queenConfigs n k satisfies Peaceful is now feasible:

queenConfigsPeaceful : (n : N) → (k : N) →
    Lift Peaceful (queenConfigs n k)
queenConfigsPeaceful n zero =
    return (([], emptyConfigPeaceful[Peaceful []]))
queenConfigsPeaceful n (suc k) =
    do
        (qs, queenConfigs n k) ← queenConfigsPeaceful n k
        (q, qNAq) ←
            filterNew
                (λ q → q areNotAttacking q)
                (range 0 (n - 1))
        return (qs + [q], addPeacefully qs q qNAq)

Here, we have used the property filterNew, defined so that filterNew f as proves that filter f as satisfies Lift (λ a → f a ≡ true).

**Gradual proving** Note that we have split up validity of a configuration into two separate predicates Fitting n and Peaceful. This was quite handy, as we could focus on proving these weaker properties in isolation of each other. The contrast of our approach to the alternative in absence of Lift is stark; we would have to implement queenConfigs to be of type (n : N) → N → List (ValidQConfig) where ValidQConfig already fully specifies a valid configuration. In addition to the disadvantages mentioned before, we would have been forced to implement the specifications of ValidQConfig all at once.

In order for this to work, we have made the implicit assumption that we can easily combine both these properties afterwards to prove the validity of our algorithm:

Valid : N → Predicate QConfig
Valid n = Fitting n ∧ Peaceful
queenConfigsValid : (n : N) → (k : N) →
    Lift (Valid n) (queenConfigs n k)
queenConfigsValid n k =
    (queenConfigsFit n k) -L (queenConfigsPeaceful n k)

The above uses a predicate conjunction operator _∧_ where _P_ ∧ _Q_ represents the conjunction predicate _λ a → P a ∧ Q a_. Additionally, we use a _¬-L_ operator defined of type Lift P as _[[ ]_] Lift Q as → Lift (P ∧ Q) as. Interestingly, this _¬-L_ is less trivial than one might expect. We come back to this in §4.3.

## 4 Completing the Picture

Up until this point, we have only talked about properties of monadic computations. In this section, we complete our story to cover more ground. In §4.1, we will broaden our point of view by illustrating that our framework fits just as nicely in the context of applicative functors. In §4.2, we generalise further to arbitrary functors. Afterwards, in §4.3, we illustrate the need to zoom in on a particular class of functors for which we can sensibly implement _¬-L_ as we have seen for List in §3.2. Lastly, in §4.4, we will focus in on the List functor, and some of the List-specific properties that we used in §3.2.

### 4.1 Simplifying twoDice and relabel - Applicative Functors

Throughout this pearl, we have confined ourselves to monadic computations. Sadly, this presentation is incomplete as it’s often a good idea to use applicative functors to define effectful computations.

Indeed, twoDice and relabel can be nicely rephrased in this applicative style.

twoDice : IO N
twoDice = (die + die)

---

That is, computations that are composed using _≡_ and _return_.

---
relabel : Tree A → State N (Tree N)
relabel (leaf a) = (leaf fresh)
relabel (node l r) = (node (relabel l) (relabel r))

Here – in order make code a bit more readable – we have used Agda’s built-in idiom bracket notation (proposed by McBride and Paterson [10]). Intuitively, the idiom brackets denote function application lifted to effectful computation. Formally, we have ([3]) (\( e a_1 \ldots a_n \)) desugars to pure \( e \otimes a_1 \otimes \ldots \otimes a_n \).

Ideally, we could now prove the lifted properties just as easily:

twoDiceBound12 : Lift (\( \lambda \ a \rightarrow a \leq 12 \)) twoDice
twoDiceBound12 =
(\( \sumOf(sBound12 \ dieBound6 \ dieBound6) \))
relabel\( \equiv \) : (t : Tree A) → Lift (\( \lambda \ t' \rightarrow t \equiv t' \)) (relabel t)
relabel\( \equiv \) (leaf a) =
(\( \lambda (\_ \rightarrow \text{leafISO}) \) (nothing2Prove fresh))
relabel\( \equiv \) (node l r) = (nodesISO (relabel\( \equiv \) l) (relabel\( \equiv \) r))

Luckily we can! To make the above proofs formal, we first lift the operators of the applicative functor class. The lifted operator for pure is almost identical to the one return; we have \( \text{pure}_L \) of type \( \forall X \rightarrow \text{pure } \lambda \ x \rightarrow \text{pure } x \).

More interesting is the _ operator. There, we have a particular kind of type \( \forall X \rightarrow \text{pure } \lambda \ x \rightarrow \text{pure } x \).

The _ operator is easily explained by analogy with the pure case, as depicted in Figure 3. Consider some function \( f : X \rightarrow Y \), \( x : X \), \( P \) on \( X \), and a predicate \( Q \) on \( Y \). Now suppose that \( x \) satisfies some predicate \( P \) on \( X \), and suppose that \( f \) satisfies \( P \Rightarrow Q \) where \( \Rightarrow \) is defined as follows.

\[ P \Rightarrow Q \equiv \forall x : X \rightarrow P x \rightarrow Q (f x) \]

That is, suppose that \( f \) maps inputs satisfying \( P \) to outputs satisfying \( Q \). It is trivial now to prove that the application of \( f \) to \( x \) satisfies \( Q \).

The operator \( \_ \otimes \) gives us the same in the impure context of an applicative functor \( F \). There, we have a particular instance of \( \_ \otimes \) to encode application of an effectful function \( af : F (X \rightarrow Y) \) to an effectful argument \( ax : F X \). Analogously now, if \( ax \) satisfies \( Lift P \) and \( af \) satisfies \( Lift P \Rightarrow Q \), we can now use \( \_ \otimes \) to prove that \( af \otimes ax \) satisfies \( Lift Q \). This is summarised in Figure 3.

The actual implementation of \( \_ \otimes \) is straightforward but tedious; the witness implementation is as expected and the proof of the \textit{corresponds} field is a long and tedious application of the applicative laws.

\[ \_ \otimes \] and \( \text{pure} \) in scope as \( \_ \) and \( \text{pure} \) respectively, we can recycle the bracket notation as demonstrated above.

4.2 Simplifying relabel - General Functors

Consider again the definition of \( \text{relabel} \). In the first case (\( \text{leaf} a \)), we do not fully use the \( \_ \otimes \) operator as indeed, we could have used \( \text{fmap} \) instead.

relabel : Tree A → State N (Tree N)
relabel (leaf a) = \( \text{fmap } \lambda \text{ fresh } \text{ leaf } \) fresh
relabel (node l r) = (node (relabel l) (relabel r))

Luckily, we also have a lifted version of \( \text{fmap} \). \( \text{fmap}_L \) of type \( (P \Rightarrow Q) f \rightarrow Lift P af \rightarrow Lift Q \) (\( \text{fmap } f af \)). So in the context of a functor \( F \); types \( A \) and \( B \); predicates \( P \) on \( A \) and \( Q \) on \( B \); a functorial value \( fa : F A \); and a function \( f : A \rightarrow B \), if we can prove \( (P \Rightarrow Q) f \) and \( Lift \ P \ af \), we can conclude \( Lift Q \) (\( \text{fmap } f \ af \)). With this in mind, we have the following proof.

relabel\( \equiv \) : (t : Tree A) → Lift (\( \lambda \ t' \rightarrow t \equiv t' \)) (relabel t)
relabel\( \equiv \) (leaf a) =
\( \text{fmap}_L \) (\( \lambda (\_ \rightarrow \text{leafISO}) \) (nothing2Prove fresh))
relabel\( \equiv \) (node l r) =
(\( \text{nodesISO} \) (relabel\( \equiv \) l) (relabel\( \equiv \) r))

4.3 Combining Properties - Pullback Preserving Functors

Consider again the strategy that we used in § 3.2. In order to prove that every obtained configuration in \( \text{queensConfigs} \) was valid, we separately proved them to be both Fitting and Peaceful. Remember how we have then used the operator \( \Rightarrow \) to combine these two into a proof of the combined property \( \text{Valid } n = \text{Fitting } n \wedge \text{Peaceful } \).

\( \text{Valid} : \mathbb{N} \rightarrow \text{QConfig} \rightarrow \text{Set} \)
\( \text{Valid } n = \text{Fitting } n \wedge \text{Peaceful } \)
\( \text{queenConfigsValid} : (n : \mathbb{N}) \rightarrow (k : \mathbb{N}) \rightarrow \text{Lift } (\text{Valid } n) \) (\( \text{queenConfigs } n \) \( k \))
\( \text{queenConfigsValid } n \) \( k = \)
(\( \text{queenConfigsFit } n \) \( k \)) \( \_ \)
(\( \text{queenConfigsPeaceful } n \) \( k \))

Crucial for this to work is the operator \( \Rightarrow \) : \( \{fa : F A\} \rightarrow Lift P \) \( fa \) \( \rightarrow Lift Q \) \( fa \) \( \rightarrow Lift (P \wedge Q) \) \( fa \). Ideally, we would hope that such an operator exists for an arbitrary functor \( F \). Sadly however, it is far from obvious how we can implement the general case.

If we restrict ourselves to pullback-preserving functors, we have such a \( \Rightarrow \) operator. Examples of such functors include \( \text{List} \), \( \text{Writer} \), \( \text{Maybe} \), \( \text{State} \) and all other polynomial functors (also called container functors \( [1, 5] \)). Moreover, beside the existence of \( \Rightarrow \), we have that pullback-preserving functors

\[ \_ \otimes \] and \( \text{pure} \) in scope as \( \_ \) and \( \text{pure} \) respectively, we can recycle the bracket notation as demonstrated above.

13Again, we rely on the fact that the bracket notation – the implementation of it in Agda – is just syntax sugaring that gets desugared before type checking.
and only those have \( _{-L} \) operators that behave universally. Universal behaviour of an operator \( _{-L} \) characterises that

uncurry \( _{-L} \): Lift \( P \times Q \) \( \times \) Lift \( P \) \( \times \) Lift \( Q \) is inverse to the canonical map

\[
\text{split}_L : \text{Lift} \ (P \times Q) \to \text{Lift} \ P \times \text{Lift} \ Q \, \text{fa}
\]

where we used \( \text{apply}_L \) of type \( \langle a : A \to P \ a \to Q \rangle \to \text{Lift} \ P \, \text{fa} \to \text{Lift} \ Q \, \text{fa} \). We prove these both cases in appendix C.

Not all functors preserve pullbacks though. In appendix C, we prove that the continuation monad, \( M \ X = (X \to R) \to R \), for instance, does not. While this excludes the possibility that for continuations, there exists a well-behaved \( _{-L} \), we have not disproven its mere existence. For continuations, we can – under the assumption that \( P \) and \( Q \) are decidable and \( R \) is inhabited – construct said operator (which can be seen in the git repository) but the general case remains unclear.

To deal with this complication, we make the \( _{-L} \) operator available in a type class PullbackPreserving that we implement for monads like List, Writer etc.

### 4.4 Proving Our Low-Level Properties about List

Up until this section, we have only proven functor-independent lifted properties. In §3.2 however, we have assumed a lot of properties about everyday List functions, whose validity actually relies upon the structure of List itself.

Ideally, the proofs of these assumptions should of course be just as concise and readable. In this section, we show that this is indeed the case.

We begin by assuming three operators that directly correspond to the constructors of the List data type. First of all, note that for the empty list \([],\) it is trivial to prove \([\_] : \text{Lift} \ [\_] \) where \( P \) an arbitrary predicate. Moreover, it is easy to implement \( _{-\Rightarrow L} : (P \ a) \to \text{Lift} \ P \, \text{xs} \) \( \Rightarrow \) \( \text{Lift} \ P \, \text{a} \times \) \( \text{Lift} \ P \, \text{xs} \). And finally, we also have \( \text{invertP-cons} \) of type \( \text{Lift} \ P \, \text{a} \times \) \( \text{Lift} \ P \, \text{xs} \). However easy they are implemented, their proofs are a bit verbose, and we leave them out here. Given these primitive implementations specific to List, we can easily derive our interesting properties.

Consider a typical implementation of the \texttt{filter} function:

\[
\text{filter} \ : \ {\cdot} \ {\cdot} \to \ {\cdot} \to \text{List} \ A \to \text{List} \ A
\]

\[
\text{filter} \ \text{fa} \, \text{ax} \, \text{with} \ f \, \text{ax} = \ ? \ \text{true}
\]

\[\text{filter} \ f \ (\text{x} :: \text{xs}) \ | \ \text{yes} \_ = \ \text{x} :: \text{filter} \ f \ \text{xs} \]

\[\text{filter} \ f \ (\text{x} :: \text{xs}) \ | \ \text{no} \_ = \ \text{filter} \ f \ \text{xs} \]

With the \([\_] \) and \( _{-\Rightarrow L} \) operators, it is easy to prove, for instance, that every element in the resulting filtered list satisfies the filtered predicate:

\[
\text{filterNew} : (f : A \to \text{Bool}) \to \text{List} \ A \to \text{List} \ A
\]

\[
\text{filterNew} \ f \ [\_] \ = \ [\_] \ L
\]

\[
\text{filterNew} \ f \ (\text{x} :: \text{as}) \ with \ f \, \text{x} = \ ? \ \text{true}
\]

\[\text{filterNew} \ f \ (\text{x} :: \text{as}) \ | \ \text{yes} \ p = \ p :: \text{filterNew} \ f \ \text{as} \]

\[\text{filterNew} \ f \ (\text{x} :: \text{as}) \ | \ \text{no} \ a = \ \text{filterNew} \ f \ \text{as} \]

Now consider a typical \texttt{range} function as defined below.

\[\text{range} : \text{List} \ \Rightarrow \text{List} \ \Rightarrow \text{List} \ \Rightarrow \text{List} \]

\[
\text{range} \ \text{zero} \ = \ [\_]
\]

\[
\text{range} \ \text{succ} \ n \ = \ [\_]
\]

\[
\text{range} \ \text{succ} \ n \ = \ \text{range} \ \text{zero} \ m \ + \ [\_]
\]

\[
\text{range} \ \text{succ} \ n \ = \ \text{fmap} \ \text{succ} \ (\text{range} \ n \ m)
\]

Proving that the resulting list is bounded below by the first argument is just as easy.

\[\text{rangeDownBound} : \]

\[
(d : \text{N}) \to (u : \text{N}) \to \text{Lift} \ (\lambda \ x \to d \leq x) \ (\text{range} \ d \ u)
\]

\[
\text{rangeDownBound} \ \text{zero} \ = \ [\_]
\]

\[
\text{rangeDownBound} \ \text{succ} \ n \ = \ [\_]
\]

\[
\text{rangeDownBound} \ \text{zero} \ m \ = \ \text{fmap} \ \text{succ} \ (\text{range} \ n \ m)
\]

The above uses a lemma for combining a lifted property proofs when appending lists:

\[
\text{rangeDownBound} \ (\text{xs} :: \text{ys}) \ = \ [\_]
\]

\[
\text{rangeDownBound} \ (\text{xs} :: \text{ys}) \ = \ \text{fmap} \ \text{succ} \ (\text{range} \ n \ m)
\]

\[\text{with} \ \text{invertP-cons} \�\text{as} \]

\[\text{rangeDownBound} \ (\text{xs} :: \text{ys}) \ = \ [\_]
\]

\[\text{rangeDownBound} \ (\text{xs} :: \text{ys}) \ = \ \text{fmap} \ \text{succ} \ (\text{range} \ n \ m)
\]

\[\text{with} \ \text{invertP-cons} \�\text{as} \]

\[\text{rangeDownBound} \ (\text{xs} :: \text{ys}) \ = \ [\_]
\]

\[\text{rangeDownBound} \ (\text{xs} :: \text{ys}) \ = \ \text{fmap} \ \text{succ} \ (\text{range} \ n \ m)
\]
Finally, for Lists, we can also prove the \(-L_{-}\) operator discussed earlier:
\[
-L_{-} : \text{List } P \rightarrow \text{List } Q \rightarrow \text{List } (P \land Q) = \lambda \{ f : \text{List } P \rightarrow \text{List } Q \rightarrow \text{List } (P \land Q) \}
\]

\[
-L_{-} \{ \text{as } x : \text{xsP} \mid \text{xxsQ} \} = \{ \}
\]

\[
-L_{-} \{ \text{as } x : \text{xsP} \mid \text{xxsQ} \} = \lambda \{ f : \text{List } P \rightarrow \text{List } Q \rightarrow \text{List } (P \land Q) \}
\]

with \(\text{invertP-cons} \text{xsP} \mid \text{invertP-cons} \text{xxsQ} \)

The use of \(\text{invertP-cons}\) in the last two examples is still a bit suboptimal. Ideally, we would like to reuse the pattern matching syntax on lifted properties proofs, rewriting \(-L_{-}\) to something like the following:

\[
-L_{-} : \text{List } P \rightarrow \text{List } P \rightarrow \text{List } (P + P) = \lambda \{ f : \text{List } P \rightarrow \text{List } P \rightarrow \text{List } (P + P) \}
\]

\[
-L_{-} \{ \text{as } x : \text{xsP} \mid \text{ysP} \} = \text{indListL \text{xsP}}
\]

\[
-L_{-} \{ \text{as } x : \text{xsP} \mid \text{ysP} \} = \lambda \{ f : \text{List } P \rightarrow \text{List } P \rightarrow \text{List } (P + P) \}
\]

Unfortunately, such a rephrasing would require a generalisation of Agda’s pattern matching along the lines of Epigram’s induction by syntax [9].

5 Feeding Proofs to Programs

The above examples demonstrate how we can use Lift to prove lifted properties about existing programs. But what if we want to rely on such lifted properties when implementing other programs?

Imagine, for example, that we want to spice up our monopoly implementation with a new chance card. The card says the player is fined for speeding, that is, he/she has to pay an amount of \(1000 \div (13 - n)\), where \(n\) is the number of pips he/she throws. A naive implementation might look like this:

\[
\text{fine} : \text{IO } \mathbb{N}
\]

\[
\text{fine} = \text{fmap} \text{amount} \text{twoDice where}\n\]

\[
\text{amount} : \mathbb{N} \rightarrow \mathbb{N}
\]

\[
\text{amount } n = \text{case } (n \leq 12) \text{ of}\n\]

\[
\lambda \{(\text{yes } p) : \rightarrow (1000 \div (13 - n, \text{ easy}(13 - n \neq 0))) ;\}
\]

\[
(\text{no } r) : \rightarrow ??\}
\]

\[
\text{where}\n\]

\[
\text{div} : \mathbb{N} \rightarrow (\Sigma[n \in \mathbb{N}] n \neq 0) \rightarrow \mathbb{N}
\]

\[
n \text{div } (d, p) = \ldots
\]

Here, we have defined an \(\text{amount}\) function from \(\mathbb{N}\) to \(\mathbb{N}\), taking the outcome of a throw and mapping it to a fine. To avoid a division by zero, we do a case split to decide whether the roll is greater than 12 or not. However, the case that the roll is greater than 12 is spurious. Rather than returning an arbitrary result, it would be better to rule out the case explicitly. This will also prevent us from forgetting about this assumption in the future.\(^{17}\)

So how can we use the proof \(\text{twoDiceBound12}\) of type \(\text{Lift } (\lambda n : n \leq 12)\) \(\text{twoDice}\) to rule out the spurious case above. Given this proof, it should in fact suffice to define \(\text{amount}\) of type \(\Sigma[n \in \mathbb{N}] n \leq 12 \rightarrow \mathbb{N}\). To do this, we define a restricted \(\text{fmap}\) operator, \(\text{fmap}_{R}\), formalising the intuition that, when applying \(\text{fmap}\) with a function \(f : A \rightarrow B\) and a value \(\text{fa} : F A\) for which we have a proof of \(\text{Lift } f\text{fa}\), we can always assume that when defining \(f, a\) satisfies \(P\). That is, it is enough to define \(f\) as a function of type \(\Sigma A P \rightarrow B\):

\[
\text{fmap}_{R} : \{ f : F A \rightarrow (\Sigma A P \rightarrow B) \rightarrow (\text{Lift } f\text{fa}) \rightarrow F B \}
\]

\[
\text{fmap}_{R} f R p = \text{fmap}_{R} f (\text{witf_ine } p)
\]

Using \(\text{fmap}_{R}\), we now have the following:

\[
\text{fine} = \text{fmap}_{R} \text{amount}_{R} \text{twoDiceBound12 where}\n\]

\[
\text{amount}_{R} : \Sigma[n \in \mathbb{N}] n \leq 12 \rightarrow \mathbb{N}
\]

\[
\text{amount}_{R} (n, p) = 1000 \div (13 - n, \text{ easy}(13 - n \neq 0))
\]

It is in fact easy to prove that this new implementation of \(\text{fine}\) is equal to the old one. Intuitively, this is because for \(d : \mathbb{N}\) and \(p : n \leq 12\), we have that \(\text{amount } d\) is equal to \(\text{amount}_{R} (d, p)\). In general, it is easily proven formally that \(\text{fmap } f_{R} \{ \text{fa} : F A \rightarrow \Sigma A P \rightarrow B\} \) equals \(\text{fmap } f_{R} \{ \text{fa} : F A \rightarrow \Sigma A P \rightarrow B\} \) if \(f\) agrees with \(f_{R}\) on \(P\).\(^{18}\)

We have an analogous operator for \(\text{ax}\).

\[
\text{ax} : F X \rightarrow F (\Sigma X P \rightarrow Y) \rightarrow \text{Lift } \text{axP} \rightarrow F Y
\]

\[
\text{axP} = \text{fmap}_{R} \text{axP} \rightarrow \text{witf_ine } \text{axP}
\]

So given an applicative value \(\text{ax} : F X\) for which we have a proof \(\text{axP}\) of \(\text{Lift } \text{axP}\), it is enough to have \(\text{f}_{R}\) of type \(\Sigma (\Sigma X P \rightarrow Y)\) instead of \(\Sigma (X \rightarrow Y)\).

And finally, the analogous operator for \(\Rightarrow_{R}\):

\[
\Rightarrow_{R} : \{ ma : \Sigma A P \rightarrow \Sigma A \Rightarrow B\} \rightarrow (\text{Lift } \text{ma} \rightarrow (\Sigma A \Rightarrow B) \rightarrow B
\]

\[
\text{witf_ine } \Rightarrow_{R} f_{R} \Rightarrow_{R} f_{R}
\]

That is, given a monadic value \(\text{ma} : \Sigma A P\) and a proof of \(\text{Lift } \text{ma}\) when binding \(\text{ma}\) with a function \(f : A \rightarrow M B\), it is enough to define \(f\) and then \(\Rightarrow_{R}\) of \(f\) a given that \(a\) satisfies \(P\). That is, it is enough to define \(f\) of type \(\Sigma A P \rightarrow M B\).

6 Conclusion

In summary, this paper shows a nice way to express and prove lifted properties of effectful code. In general, we can express any property related to the results of an effectful computation, and prove it with our lifted operators provided that it holds independently of the specific functor/applicative/monad in use.

\(^{17}\)Imagine, for example, that we introduce a new rule that you can roll the dice again if you roll doubles the first time (same number on each die).

\(^{18}\)That is, if for all \((a, p) : \Sigma A P\) we have that \(f a\) equals \(f_{R} (a, p)\).
The properties can be formulated extrinsically (no need to modify the original code). The proofs are readable and easy to understand, and can be composed modularly. No language modifications are needed and in fact, the core of our framework is very small; it consists only of \( \text{Lift}, \_ \gg \_ \), \( \text{return}_L, \text{pure}_L, \_ \odot \_ \) and \( \text{fmap}_L \).

Moreover, we can easily reuse existing notation to structure compositions with \( \_ \gg \_ \) (do-notation), and \( \_ \odot \_ \) together with \( \text{pure}_L \) (idiom brackets) due to the flexible nature of their implementation in Agda. That is, both do-notation and idiom brackets are desugared before type checking, and so, we can easily experiment with operators whose types do not adhere to the prior intent of the syntax. Furthermore, the examples in this pearl provide a clear case for the virtues of said flexibility.

As we have explained, our approach is limited to properties that can be phrased in terms of the results of effectful operations. Moreover – confining ourselves to the lifted operators – we can only prove properties independent of the specific functor in use. We expect that supporting more expressive properties is possible however, by focusing in on a specific (class of) underlying monads/functors. We already demonstrated the feasibility of this for the List functor; by merely implementing some basic primitive properties corresponding to the data constructors, we were able to obtain nice and intuitive proofs about your everyday List functions.

More elaborately, we show in the git repository how can take Swierstra’s Hoare State monad for writing witnesses of lifted properties (over State) that are quantified over the initial and final state variable and prove the no-duplicates property extrinsically. In the future, it could be interesting to extend the approach to more specific properties. The idea would be to use strongly-specified witnesses that testify for these more expressive properties and require an appropriate correspondence to the underlying code.

Of course, not all properties regarding effectful computations concern themselves merely with the results of that computation; think \textit{interestingOperation} \( : \text{IO} \ \top \) for example. In such cases, the programmer should use a different approach, like equational reasoning [6, 7].

### 6.1 Related Work

#### Equational reasoning

The tree relabelling problem is often used in literature to showcase the use of equational reasoning to prove general properties about effectful computations [7]. So too is the n-queens problem [6]. We note however that the general objective in this setting is quite different from what we do here; one proves a desired property about a given effectful computation by arguing that it is contextually equivalent to a more naive computation, clearly satisfying said property. This is done by chaining together a link of equalities; hence the name ‘equational reasoning’.

A narrow version of said practice can be found in the framework presented in this paper though. We prove that computations satisfy specifications with regard to their result, and we do this by implementing a strongly specified version. Arguing that forgetting about this specification gives us back the original computation, is now done completely under the hood, formally encapsulated in our lifted operators. Moreover, the program that will be run is still the original one, untouched by our extrinsic proofs, so that our technique has no runtime cost (other than the one involved in migrating to a dependently typed functional language).

#### Ornaments

A key idea in the development of our framework is the way in which we define \text{Lift}; we demand an intrinsic, strongly specified version of our original computation together with a proof that forgetting about this specification, we obtain the original computation.

Of course, the practice of decorating structures is reminiscent of ornaments [8]. However, we believe that the theory of ornaments does not provide an alternative to our techniques, for two reasons. First, the goal of ornaments is different. We are ultimately interested in the ability to express and prove properties of effectful code (such as \text{twoDice}) extrinsically, as we did in \text{twoDiceBound12} : \text{Lift} (\lambda \ a \rightarrow a \leq 12) \text{twoDice} \ \text{Witnesses}. Ornamentation libraries, on the other hand, are often designed to combine an extrinsic proof like \text{twoDiceBound12} with the underlying implementation \text{twoDice} into an intrinsically correct implementation.

Secondly, the results in the ornament literature simply do not seem to be applicable in our setting. Our \text{Lift} operator is designed to reason about forgetful maps of the form \text{proj}_j : \Sigma \ A \ P \rightarrow A$. We like to point out that, since it is possible in dependent type theory to define the inverse image of a function, any function \text{f} : B \rightarrow A$ factors as $B \equiv \Sigma (x : A) g^{-1}(x) \rightarrow A$. Thus, we essentially consider all functions. Ornaments, on the other hand, consider only a restricted class of functions which can be seen as forgetting ornaments\(^{19}\). Alternatively, one might try to apply the theory of ornaments not to the result type but to the monad at hand, but then we would obtain results regarding effect specifications ignoring the result of a computation, which sounds interesting but is quite the opposite of the current paper’s subject.

### A Manual Proof

\[ \text{cumbersomeProof} : \text{twoDice} \equiv \text{fmap proj}_j \ \text{aWitness} \]

\[
\begin{align*}
\text{cumbersomeProof} &= \\
&= \begin{cases}
\text{do} \ x \leftarrow \text{die} \\
\text{\hspace{1cm}} y \leftarrow \text{die} \\
\text{\hspace{1cm}} \text{return} \ (x + y) \\
\end{cases}
\equiv \langle \text{cong} \ (\text{flip} \ _ \gg \_) \ (\text{corresponds dieBound6}) \rangle \\
&= \begin{cases}
\text{do} \ x \leftarrow \text{fmap proj}_j \ (\text{witness dieBound6}) \\
\end{cases}
\]

\(^{19}\)Dagand [4] characterizes them as functions between indexed inductive types (viewed as indexed W-types) that arise from cartesian natural transformations between their generating indexed polynomial functors.
y ← die
  return (x + y))
≡ ⟨fmap-bind _ _ _ _⟩
  (do (x, px) ← witness dieBound6
    y ← die
    return (x + y))
≡ (cong (_$\cong$_) (witness dieBound6))
  (funext $\lambda$ _ $\to$ cong (flip $\cong$ _ _ _ _)
    (corresponds dieBound6))
  (do (x, px) ← witness dieBound6
    y ← fmap f px
    return (x + y))
≡ (cong (_$\cong$_) (witness dieBound6))
  (do (x, px) ← witness dieBound6
    y, py ← witness dieBound6
    return (x + y))
≡ (cong (_$\cong$_) (witness dieBound6))
  (do (x, px) ← witness dieBound6
    y, py ← witness dieBound6
    fmap (proj1 [B = $\lambda$ n $\to$ n $\leq$ 12])
    return (x + y, sumOfBound6IsBound12 px py))
≡ (cong (_$\cong$_) (witness dieBound6))
  (do (x, px) ← witness dieBound6
    y, py ← witness dieBound6
    return (x + y, sumOfBound6IsBound12 px py))
≡ (sym (fmap-move-bind _ _ _ _ _ _ _ _ _ _ _ _ _ )
    fmap proj1 (do (x, px) ← witness dieBound6
      y, py ← witness dieBound6
      return (x + y, sumOfBound6IsBound12 px py))

Here, we have used the following helper functions.

\[
\text{fmap-bind } g \, f \, m x : \\
(f \text{map } f \, m x) \equiv g \equiv m x \equiv (g \circ f) \\
\text{fmap-return } f : \equiv \text{return } (f \, a) \\
\text{fmap-move-bind } f \, m a \, g : \equiv (f \, \text{map } g \, (m a)) \equiv (m a \equiv (f \, \text{map } g \, f))
\]

\section{Manual Proof relabel}

\text{relabel} : (t : \text{Tree } A) \to (n : \mathbb{N}) \to \text{evalState } (\text{relabel } t) \, n \equiv t \\
\text{relabel} (\text{leaf } a) \, n \equiv \text{leafISO} \\
\text{relabel} (\text{node } l \, r) \, n \equiv \text{nodesISO } (\text{relabel } l \, n) \, (\text{relabel } r \, (\text{execState } (\text{relabel } l) \, n))

However small, the above proof is quite awkward as it needs to take into account the particular details of State binding that are totally void in our intuitive understanding of the property.

\section{On Pullback Preservation}

In this appendix, we show that a well-behaved operator $\text{relabel}$ exists for a given functor $F$ if and only if $F$ preserves pullbacks.

Moreover, we show that the continuation monad $M \, X = (X \to R) \to R$ does not preserve pullbacks.\textsuperscript{20}

\subsection{Definitions}

Because this paper is not about formalizing category theory in type theory, we use the following down-to-earth definition of a pullback:

\textbf{Definition C.1.} A (propositionally) commutative diagram

\[
\begin{array}{ccc}
T & \xrightarrow{P_1} & A \\
\downarrow{P_2} & & \downarrow{f} \\
B & \xrightarrow{g} & C
\end{array}
\]

is called a pullback square (and $T$ is called the pullback of $A \xrightarrow{f} C \xleftarrow{g} B$) if $T$ is isomorphic to the type $\Sigma \{ a \in A \mid \Sigma \{ b \in B \mid \Sigma \{ c \in C \mid (f \, a \equiv c) \times (g \, b \equiv c) \} \} \}$ with the maps $P_1 : T \to A$ and $P_2 : T \to B$ corresponding to the appropriate projections from this type of quintuples.

\textbf{Definition C.2.} A functor $F$ preserves pullbacks if it maps any pullback square as in definition C.1 to a new diagram

\[
\begin{array}{ccc}
F \, T & \xrightarrow{F \, P_1} & F \, A \\
\downarrow{F \, P_2} & & \downarrow{\text{fmap } f} \\
F \, B & \xrightarrow{F \, g} & F \, C
\end{array}
\]

that is also a pullback square.

\section{Combining Properties Requires Pullback Preservation}

We seek to prove the following:

\textbf{Proposition C.3.} The function $\text{split}_L$ (§4.3) is an isomorphism if and only if $F$ preserves pullbacks.

\textbf{Proof.} If we define $P$ as the inverse image of $f$ and $Q$ as that of $g$, then $A \equiv \Sigma \, C \, P$ and $B \equiv \Sigma \, C \, Q$. Then $T$ will be the pullback of $f$ and $g$ if and only if $T \equiv \Sigma \, C \, (P \land Q)$. If we apply to this diagram the functor $F$ and observe that $F \, (\Sigma \, X \, R) \equiv \Sigma \, (F \, X) \, (\text{Lift } R)$,\textsuperscript{21} then we can write the result up to isomorphism as:

\[
\begin{array}{ccc}
\Sigma (F \, C) \, (Lift \, (P \land Q)) & \to & \Sigma (F \, C) \, (Lift \, P) \\
\downarrow & & \downarrow \\
\Sigma (F \, C) \, (Lift \, Q) & \xrightarrow{\text{proj}_1} & F \, C
\end{array}
\]

\textsuperscript{20}Throughout the section, we assume uniqueness of identity proofs.

\textsuperscript{21}In the right hand type of this isomorphism, the \textit{corresponds} field of the second component can be defined / pattern matched against using the reflexivity constructor of the identity type, fixing the first component of type $F \, X$ and leaving only the \textit{witness} field of type $F \, (\Sigma \, X \, R)$ as actual information.
Now this diagram is a pullback square if and only if $\Sigma (F C)$ (\text{Lift} $(P \land Q)$) is isomorphic to $\Sigma (F C)$ (\text{Lift} $P \land \text{Lift} Q$) in a manner compatible with the canonical maps to $\Sigma (F C)$ (\text{Lift} $P$) and $\Sigma (F C)$ (\text{Lift} $Q$). In other words, we require that \text{Lift} $(P \land Q) \, f \, c$ be isomorphic to \text{Lift} $P \, f \, c \times \text{Lift} Q \, f \, c$ in a manner compatible with the projections to \text{Lift} $P \, f \, c$ and \text{Lift} $Q \, f \, c$. This compatibility criterion equivalently says that one side of the isomorphism is given by $\text{split}_L$. This proves the proposition. \hfill $\Box$

### C.3 The Continuation Monad Does not Preserve Pullbacks

In this section, we consider the functor $M X = (X \to R) \to R$, and show via proposition C.3 that it does not preserve pullbacks. In other words, we will show that \text{Lift} $(P \land Q) \, m \, a$ is not necessarily isomorphic to \text{Lift} $P \, m \, a \times \text{Lift} Q \, m \, a$. As a warm-up, let us consider what Lift even means for the continuation monad.

A value $m \, a : M A$ is a computation that pretends to produce an output of type $A$ but in fact grabs the continuation of type $A \to R$ (a computation containing a hole of type $A$ and producing an overall result of type $R$) and manipulates it to produce some result of type $R$. A pure computation will simply feed a value of type $A$ to the continuation; in general, $m \, a$ may call the continuation zero or multiple times and combine the results.

If we have a value $m \, a : M A$ and a function $f : A \to B$, then $\text{fmap} \, f \, m \, a : M B$ will take a continuation $k : B \to R$, compose it with $f$ and feed the result to $m \, a$. A value $m \, a : M A$ satisfies \text{Lift} $P$ if it arises by applying $\text{fmap} \, \text{proj}_1 : M (\Sigma \, A \, P) \to M A$ to some computation of type $M (\Sigma \, A \, P)$. In other words, a proof of $\text{Lift} \, m \, a \, P$ indicates that, even though $m \, a$ takes a continuation of type $A \to R$, it will only invoke this computation on values $a : A$ for which it can prove $P \, a$.

A remarkable phenomenon arises when $P$ happens to be a proof-relevant predicate: a type family that for some values $a : A$ may contain more than a single element. In this case, \text{Lift} $P$ also becomes proof-relevant, but in a blown-up manner. Indeed, $m \, a : M A$ satisfies continuations of type $A \to R$, whereas a computation of type $M (\Sigma \, A \, P)$ takes continuations of type $\Sigma \, A \, P \to R$. The latter type of continuations is much bigger, as it contains continuations that distinguish between proofs of $P \, a$. Hence, a witness that $m \, a : M A$ satisfies \text{Lift} $P$ needs to respect the behaviour of $m \, a$ on continuations that ignore the proof of $P \, a$, but can behave arbitrarily on those that don’t. The result is that the number of witnesses corresponding to a single $m \, a$ can become enormous.

In fact, we can consider the dullest instance of a proof-relevant predicate and argue with a single cardinality argument that $M$ cannot preserve pullbacks. We take $A = \top$ (the unit type), $R = \text{Bool}$ and $P = Q = \lambda \_ \to \text{Bool}$. Then we have $\Sigma (M \top) (\text{Lift} (P \land Q)) \cong M (\Sigma \top (P \land Q)) \cong M (\text{Bool} \times \text{Bool})$ and $\Sigma (M \top) (\text{Lift} P \times \text{Lift} Q)$.

\[ \cong M \text{Bool} \times M \text{Bool} \]

where $\cong$ denotes an injection. Thus, if $M$ were to preserve pullbacks, then $M (\text{Bool} \times \text{Bool})$ should have lower cardinality than $M \text{Bool} \times M \text{Bool}$. However, the cardinality of the former type is $2^{2^{2^2}} = 2^{64}$, whereas the latter type has cardinality $2^{2^2} \cdot 2^2 = 2^9$ and this is not even a conservative estimate.

### Acknowledgments

This work was funded in part by Internal Funds KU Leuven grant C14/18/064.

Andreas Nuyts holds a Ph.D. Fellowship from the Research Foundation - Flanders (FWO).

### References


