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.

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 *sumOfBound6IsBound12*, of type $\{x : \mathbb{N}\} \to x \leqslant 6 \to \{y : \mathbb{N}\} \to y \leqslant 6 \to x + y \leqslant 12$, which takes proofs that $x \leqslant 6$[1] and $y \leqslant 6$, and returns a proof that $x + y \leqslant 12$.[2]

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 *twoDice*, which is implemented in terms of an underlying operation *die*:

```
die : IO ℕ
die = ...

twoDice : IO ℕ
twoDice = do x ← die
             y ← die
             return (x + y)
```

That is, we are interested in proving that *twoDice* returns, upon execution, a natural number not greater than *12*.

An intrusive solution We can of course reimplement *die* and make it return not just a natural number n but also a proof that n is bounded by 6.[3] Likewise, we can reimplement *twoDice* – with the help of the *sumOfBound6IsBound12* function – so that it returns a natural number n and a proof that n is bounded by 12:[4]

```
die : IO (Σ[ n ∈ ℕ ] n ⩽ 6)
die = ...

twoDice : IO (Σ[ n ∈ ℕ ] n ⩽ 12)
twoDice =
  do (x^ℕ , px^{x ⩽ 6}) ← die^{IO (Σ[ n ∈ ℕ ] n ⩽ 6)}
     (y^ℕ , py^{x ⩽ 6}) ← die^{IO (Σ[ n ∈ ℕ ] n ⩽ 6)}
     return ( (x + y)^ℕ
            , (sumOfBound6IsBound12 px py)^{x + y ⩽ 12} )
```

[1] The type constructor $\_\leqslant\_$ maps natural numbers a and b to the type $a \leqslant b$ representing the proposition that a is at most b. Following the Curry-Howard correspondence [12], the type $a \leqslant b$ 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

[3] That is, upon execution, *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 $(n\ ,\ p)$ where $n : \mathbb{N}$ and $p : n \leqslant 6$. Such pairs are exactly the inhabitants of the type $\Sigma[\, n \in \mathbb{N} \,]\ n \leqslant 6$.

[4] To aid the reader, we sometimes annotate functions and variables with their type like $^{\text{this}}$. These superscripts are not part of the code.

TyDe 2019, August 2019, Berlin, Germany

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This approach is referred to as strong specification [13]; the type IO ($\Sigma[\ n \in \mathbb{N}\ ]\ n \leqslant 12$) 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 implementation.

However, modifying the original code to use strong specification 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 properties 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 $\mathbb{N}$ untouched and prove their properties extrinsically, in a separate definition.[5] Let us imagine what such a solution could look like.

Dreaming up a better solution Of course, we cannot simply write $twoDice \leqslant 12$ as $\lambda\ a \to a \leqslant 12$ is a predicate[6] on $\mathbb{N}$, not IO $\mathbb{N}$. This suggests that what we need is an operator, say Lift, which takes a predicate on $\mathbb{N}$, say $\lambda\ a \to a \leqslant 12$, and returns the desired, lifted predicate on IO $\mathbb{N}$. That is, for an $ioa : \mathsf{IO}\ A$, the type Lift P 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 inhabitant *dieBound6* of type Lift $(\lambda\ a \to a \leqslant 6)$ *die*, how can we prove Lift $(\lambda\ a \to a \leqslant 12)$ *twoDice*? One way to think about this is that intuitively, Lift $(\lambda\ a \to a \leqslant 6)$ *die* is true because *die could* be reimplemented as a computation of type IO $(\Sigma[\ n \in \mathbb{N}\ ]\ n \leqslant 6)$. Analogously, Lift $(\lambda\ a \to a \leqslant 12)$ *twoDice* is true *intuitively* because *twoDice* could be reimplemented at type IO $(\Sigma[\ n \in \mathbb{N}\ ]\ n \leqslant 12)$. Perhaps, even though Lift is obviously not a monad, it would be nice if we could write a proof of Lift $(\lambda\ a \to a \leqslant 12)$ *twoDice* in a form of do-notation, where we essentially replicate this hypothetical reimplementation, only now in terms of *dieBound6*.

```
dieBound6 : Lift (λ a → a ⩽ 6) die^[IO ℕ]
dieBound6 = ...

twoDiceBound12 : Lift (λ a → a ⩽ 12) twoDice^[IO ℕ]
twoDiceBound12 =
  do (x^[ℕ] , px^[x ⩽ 6]) ← dieBound6^[Lift (λ a → a ⩽ 6) die]
     (y^[ℕ] , py^[y ⩽ 6]) ← dieBound6^[Lift (λ a → a ⩽ 6) die]
     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
                  y ← die
                  return (x + y)

dieBound6twoTimesBound12 :
  (die : IO ℕ) → Lift (λ a → a ⩽ 6) die →
  Lift (λ a → a ⩽ 12) (twoTimes die)
dieBound6twoTimesBound12 die 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 $die : \mathsf{IO}\ \mathbb{N}$ or $dieBound6 : \mathsf{Lift}\ (\lambda\ n \to n \leqslant 6)\ die$ 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
                          y ← 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

[5] Completely analogous to the pure case, we defined *sumOfBound6IsBound12* extrinsically instead some intrinsic specification of addition, which would have the type $\Sigma[\ n \in \mathbb{N}\ ]\ n \leqslant 6 \to \Sigma[\ n \in \mathbb{N}\ ]\ n \leqslant 6 \to \Sigma[\ n \in \mathbb{N}\ ]\ n \leqslant 12$.

[6] A predicate, say P, on a type A is a function of type $A \to \mathsf{Set}$. That is, for each inhabitant of A, it returns a type $P\ a$. For example, the predicate $\lambda\ a \to a \leqslant 12$ on $\mathbb{N}$ takes a natural number n, and maps it to the proposition $n \leqslant 12$.

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 between *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 $(\Sigma[\, n \in \mathbb{N}\,]\ n \leqslant 6)$:[7][8]

$$
\begin{aligned}
&dieListWithProofs : \mathsf{List}\ (\Sigma[\, n \in \mathbb{N}\,]\ n \leqslant 6)\\
&dieListWithProofs =\\
&\quad [\,(1, p16^{\boxed{1 \leqslant 6}}), (2, p26^{\boxed{2 \leqslant 6}}), (3, p36^{\boxed{3 \leqslant 6}})\\
&\quad , (4, p46^{\boxed{4 \leqslant 6}}), (5, p56^{\boxed{5 \leqslant 6}}), (6, p66^{\boxed{6 \leqslant 6}})\,]
\end{aligned}
$$

This *dieListWithProofs* : List $(\Sigma[\, n \in \mathbb{N}\,]\ n \leqslant 6)$ 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 $fmap\ proj_1\ dieListWithProofs$ (i.e. executing *dieListWithProofs* and using $proj_1 : \Sigma[\, n \in \mathbb{N}\,]\ n \leqslant 6 \to \mathbb{N}$ to forget the proof) is equal to the original *dieList*.[9]

$$
\begin{aligned}
&dieListProofsCorr : fmap\ proj_1\ dieListWithProofs \equiv dieList\\
&dieListProofsCorr = refl
\end{aligned}
$$

We can prove this equality using *refl* because $fmap\ proj_1$ *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 $fa : F\ A$:

$$
\begin{aligned}
&\textbf{record}\ \mathsf{Lift}\ (P : \mathsf{Predicate}\ A)\ (fa : F\ A) : \mathsf{Set}\ \textbf{where}\\
&\quad \textbf{field}\ witness : F\ (\Sigma\ A\ P)\\
&\qquad\quad corresponds : fa \equiv fmap\ proj_1\ witness
\end{aligned}
$$

Lift *P fa* is a record type with two fields: *witness* of type $F\ (\Sigma\ A\ P)$ and *corresponds*: a proof of equality between *fa* and $fmap\ proj_1\ witness$.

Coming back to our example, we now have the following:

$$
\begin{aligned}
&dieListBound6 : \mathsf{Lift}\ (\lambda\ a \to a \leqslant 6)\ dieList\\
&dieListBound6 =\\
&\quad \textbf{record}\ \{witness\ \ \ = dieListWithProofs\\
&\qquad\quad ; corresponds\ = dieListProofsCorr\}
\end{aligned}
$$

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 $(\lambda\ s \to length\ s \geqslant 20)$ *choosePassword* expresses that *choosePassword* will only produce passwords of length $\geqslant 20$. For a non-deterministic operation $range : \mathbb{N} \to \mathbb{N} \to \mathsf{List}\ \mathbb{N}$ in the List monad, we can implement $\forall\ \{n\ m\} \to \mathsf{Lift}\ (\lambda\ x \to (n \leqslant x))\ (range\ n\ m)$ and $\forall\ \{n\ m\} \to \mathsf{Lift}\ (\lambda\ x \to (x \leqslant m))\ (range\ n\ m)$, to formalise that every element in $range\ n\ m$ is between *n* and *m*. Considering a stateful implementation of Euclid's algorithm, $gcd : \mathbb{N} \to \mathbb{N} \to \mathsf{State}\ \mathbb{N}\ \mathbb{N}$, we could implement Lift $(\lambda\ d \to (Divisible\ d\ n) \times (Divisible\ d\ m))\ (gcd\ n\ m)$, encoding the fact that, for arbitrary $s : \mathbb{N}$, we have $Divisible\ (evalState\ (gcd\ n\ m)\ s)\ n$ and $Divisible\ (evalState\ (gcd\ n\ m)\ s)\ m$.

In general, if we think of a functorial values $fa : F\ A$ as a container storing values of type *A*, then Lift *P fa* encodes the proposition that every value inside this container satisfies *P*. If we think of a monadic value $ma : M\ A$ as the encoding of an effectful computation that gives output in *A*, then Lift *P ma* encodes the proposition that every output will satisfy *P*.

[7] When 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 $^{\boxed{\text{this}}}$.

[8] For clarity, we use a Haskell inspired pseudo syntax to denote lists.

[9] Actually, we have $proj_1 : \Sigma\ A\ P \to A$ for all types *A* and predicates *P* on *A*. That is, fixing some *A* and *P*, $proj_1$ is just the function $\lambda\ \{(a, p) \to a\}$ discarding the second component of each pair.

2.2 Binding Proofs

But let us consider again the operation *twoDice* : IO $\mathbb{N}$ from the introduction. Now that we can lift properties, let us attempt to prove *twoDiceBound12*. We can do this by manually implementing the *witness* and *corresponds* fields:

```
twoDiceBound12 : Lift (λ a → a ⩽ 12) twoDice
twoDiceBound12 = record {witness    = aWitness;
                         corresponds = aCorresponds}
  where
    aWitness : IO (Σ[ n ∈ ℕ ] n ⩽ 12)
    aWitness =
      do (x , px) ← (witness dieBound6)   [IO (Σ[ n ∈ ℕ ] n ⩽ 6)]
         (y , py) ← witness dieBound6
         return (x + y , sumOfBound6IsBound12 px py)
    aCorresponds : twoDice ≡ fmap proj₁ witness
    aCorresponds = ???
```

While the candidate witness is easy, the *corresponds* field is a bit tricky. We have a good intuition for why the latter must be so, but formalising this intuition requires no less than 34 lines of code, that we show in appendix A. This manual proof consists of 7 equational reasoning steps, of which 5 tedious applications of the monad laws. Moreover, we have the same burden of proof when proving Lift $(\lambda\ a \to a \leqslant 6)$ *die* and again for the lower level implementations for which we need to prove some lifted properties. In sum, manually proving these *corresponds* fields is not a viable strategy.

A better alternative is to define operators that work directly with these lifted properties. Enter the *lifted operators*, $return_L$, $\_\gg\!=_L\_$ and $\_\gg_L\_$.

Let us begin with the easiest one, $return_L$. For an $a : A$, and a proof of $P\ a$, the $return_L$ operator proves Lift P (*return a*):

```
return_L : ((a , p) : Σ A P) → Lift P (return a)
return_L (a , p) =
  record {witness    = return (a , p);
          corresponds = ...[return a ≡ fmap proj₁ (return (a , p))]}
```

We leave out the exact implementation for the *corresponds* field; it just boils down to the left unit monad law.

The lifted bind operator $\_\gg\!=_L\_$ is more interesting. Consider an $ma : M\ A$ and $f : A \to M\ B$. 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 $\_\gg\!=_L\_$ states that also Lift Q $(ma \gg\!= f)$:

```
_≫=_L_ : Lift P ma → (((a , p) : Σ A P) → Lift Q (f a)) →
         Lift Q (ma ≫= f)
lp ≫=_L flp =
  record {witness    = (witness lp) ≫= (witness ∘ flp);
          corresponds = ...}
```

A proof of the *corresponds* field is fairly long and technical. It boils down to the use of monadic laws, combined with the correspondences from *lp* and *flp*.

And of course we also have $\_\gg_L\_$ as a special case of $\_\gg\!=_L\_$:

```
_≫_L_ : Lift P ma → Lift Q mb → Lift Q (ma ≫ mb)
lPma ≫_L lQmb = lPma ≫=_L λ _ → lQmb
```

Now that we have defined these lifted operators, we can write down a concise proof for Lift $(\lambda\ a \to a \leqslant 12)$ *twoDice*:

```
twoDiceBound12 : Lift (λ a → a ⩽ 12) twoDice
twoDiceBound12 =
  dieBound6[Lift (λ a → a ⩽ 6) die] ≫=_L λ (x , px) →
  dieBound6[Lift (λ a → a ⩽ 6) die] ≫=_L λ (y , py) →
  return_L (x + y , sumOfBound6IsBound12 px py)
```

The *corresponds* 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 *witness* 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 out the $\_\gg\!=_L\_$ operator, can't we just **do** it? The **do**-notation desugars recursively to expressions with $\_\gg\!=\_$ and $\_\gg\_$ as described in [2]. It was developed with the classical monadic $\_\gg\!=\_$ and $\_\gg\_$ operators in mind to restructure monadic compositions using these operators, making them more readable.

In Agda however, it is fully up to us which $\_\gg\!=\_$ and $\_\gg\_$ we want to have in scope. By locally renaming $\_\gg\!=_L\_$ to $\_\gg\!=\_$, we can reuse the **do**-notation to structure our proof more easily. We have the following:[10]

```
open import Monads hiding (return; _≫=_; _≫_)
open import LiftOperators
  renaming (return_L to return;
            _≫=_L_ to _≫=_; _≫_L_ to _≫_)
twoDiceBound12 : Lift (λ a → a ⩽ 12) twoDice
twoDiceBound12 =
  do (x , px) ← dieBound6[Lift (λ a → a ⩽ 6) die]
     (y , py) ← dieBound6[Lift (λ a → a ⩽ 6) die]
     return (x + y , sumOfBound6IsBound12 px py)
```

In other words, our pseudo-proofs from the introduction have been valid formal ones all along!

3 Some More Elaborate Examples

So our framework can tackle the property Lift $(\lambda\ a \to a \leqslant 12)$ *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.

[10] For consistency, we rename our $return_L$ operator to *return* as well here.

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:

```
relabel : Tree A → State ℕ (Tree ℕ)
relabel (leaf a)   = do n ← fresh
                        return (leaf n)
relabel (node l r) = do l′ ← relabel l
                        r′ ← relabel r
                        return (node l′ r′)
```

Here, we have defined *fresh* as follows:

```
fresh : State ℕ ℕ
fresh = do n ← get
           modify suc
           return n
```

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 $(\lambda\ t' \to t' \cong t)$ $(relabel\ t)$ 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:

```
data _≅_ : Tree A → Tree B → Set where
  leafISO : leaf a ≅ leaf b
  nodesISO : tal ≅ tbl → tar ≅ tbr →
    node tal tar ≅ node tbl tbr
```

No duplicates Lift *NoDups* (*relabel t*) where *NoDups* is a predicate on Tree ℕ such that *NoDups tree* encodes the property that *tree* : Tree ℕ contains no duplicate values. That is, any tree resulting from *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:[11]

```
relabel≅ : (t : Tree A) → Lift (λ t′ → t ≅ t′) (relabel t)
relabel≅ (leaf a)   = do (n , _) ← nothing2Prove fresh
                         return (leaf n , leafISO)
relabel≅ (node l r) =
  do (l′ [Tree ℕ] , pl≅l′ [l ≅ l′]) ← relabel≅ l
     (r′ [Tree ℕ] , pr≅r′ [r ≅ r′]) ← relabel≅ r
     return (node l′ r′ , nodesISO pl≅l′ pr≅r′)
```

The above uses a simple helper function *nothing2Prove* of type $(fa : F\ A) \to \text{Lift}\ (\lambda\ a \to \top)\ (fa)$, which proves a trivial predicate for the result of an arbitrary computation $fa : F\ A$. The predicate $(\lambda\ a \to \top)$ 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 Lift, we cannot prove it with our lifted bind operators as our induction hypothesis is not strong enough. Indeed, given trees l' : Tree ℕ and r' : Tree ℕ that satisfy *NoDups*, it is generally not the case that *node* l' r' satisfies *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 : 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 *queenConfigs* of type ℕ → ℕ → List QConfig. Given a chessboard of size n : ℕ and an amount of queens k : ℕ, *queenConfigs n k* : List QConfig returns a list of k-sized queen configurations. We represent queen configurations as lists of natural numbers, i.e. we write QConfig for List ℕ. The encoding is depicted in Figure 1: queens are placed in the k leftmost columns and the ith 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 ℕ is quite loose. That is, many terms of type List 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* : QConfig, and the candidate row of a new rightmost queen, say q : ℕ, *qs*

[11] From here on, we leave the renaming of our lifted operators implicit.

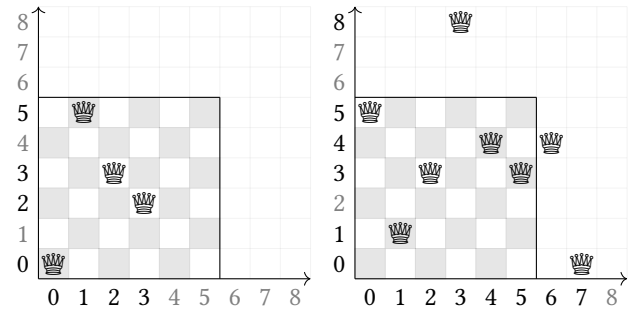

Figure 1. Two chessboards of size *6*, represented by lists [*0* , *5* , *3* , *2*] and [*5* , *1* , *3* , *8* , *4* , *3* , *4* , *0*]. The *i*th number in the list represents the row number for the queen in column *i-1*.

areNotAttacking q will evaluate to a boolean encoding what its name implies:

```
_areNotAttacking_ : QConfig → ℕ → Bool
qs areNotAttacking q =
  upwardDiagonal qCoordinate
    ∉ fmap upwardDiagonal qsCoordinates &&
  downwardDiagonal qCoordinate ∉
    fmap downwardDiagonal qsCoordinates &&
  row qCoordinate ∉ fmap row qsCoordinates
    where
      qsCoordinates : List (ℕ × ℕ)
      qsCoordinates = toCoordinates qs
      qCoordinate : ℕ × ℕ
      qCoordinate = length qs , q
```

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

```
toCoordinates : List ℕ → List (ℕ × ℕ)
toCoordinates qs = zip (range 0 (length qs - 1)) qs
```

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

```
upwardDiagonal : ℕ × ℕ → ℕ
upwardDiagonal (x , y) = x + y
downwardDiagonal : ℕ × ℕ → ℤ
downwardDiagonal (x , y) = x - y
row : ℕ × ℕ → ℕ
row (x , y) = y
```

These definitions are easily verified by glancing at Figure 2. Notice that the encoding already enforces that no two queens reside in the same column.

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

```
queenConfigs : ℕ → ℕ → List QConfig
queenConfigs n zero    = return []
```

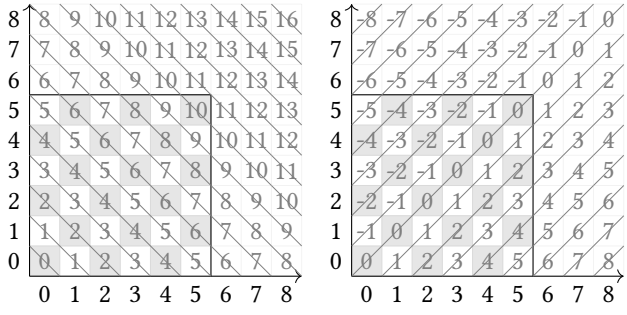

Figure 2. The downward diagonals are easily identified by summing the coordinates (left). The upward diagonals can be determined by their subtraction (right).

```
queenConfigs n (suc k) =
  do qs[QConfig] ← queenConfigs n k
     q[ℕ]        ← filter (λ 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 (*λ q → q ⩽ n - 1*) on QConfig which we denote as Fitting *n*. We can then prove that *every* obtained configuration in *queenConfigs n k* : List QConfig fits, that is, we prove Lift (Fitting *n*) (*queenConfigs n k*):

```
queenConfigsFit : (n : ℕ) → (k : ℕ) →
  Lift (Fitting n) (queensConfigs n k)
queenConfigsFit n zero =
  return ([][QConfig] , emptyConfigFits[Fitting n []])
queenConfigsFit n (suc k) = do
  (qs[QConfig] , qsFit[Fitting qs]) ← queenConfigsFit n k
  (q[ℕ] , qFit[q ⩽ n - 1]) ←
    filterPreserves[(f : A → Bool) → Lift P as → Lift P (filter f as)]
      (λ q → qs areNotAttacking q)
      (rangeUpBound 0 (n - 1))[Lift (λ r → (r ⩽ n - 1)) (range 0 (n - 1))]
  return (qs ∺ [ q ] , (qsFit ∺L [ qFit ]L)[Fitting (qs ∺ [ q ])])
```

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 [_]$_L$ operator of type *P a* → Lift *P* [*a*] where *a* is instantiated by *q*.

The peacefulness 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)
queenConfigsPeaceful n (suc k) = do
  (qs , qsP) ← queensConfigsPeaceful n k
  (q , qsNAq) ← filterNew (λ q → qs areNotAttacking q)
                          (range 0 (n - 1))
  return ((qs ++ [ q ]) , addPeacefully qs q qsP qsNAq)
```

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 $P_1 \wedge P_2$ represents the conjunction predicate $\lambda\ a \to P_1\ a \times P_2\ a$. Additionally, we use a _→$_L$_ operator defined of type Lift *P as* [List *A*] → 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[12]. 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 [10] to define effectful computations.

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

```
twoDice : IO ℕ
twoDice = ⦇ die + die ⦈

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]) that ⦇ *e* a_1 .. a_n ⦈ desugares to *pure e* ⊛ a_1 ⊛ .. ⊛ a_n.[13]

[12] That is, computations that are composed using _≫=_ and *return*.

[13] Note that ⦇ *genericDie* + *genericDie* ⦈ should be interpreted as ⦇ _+_ *genericDie genericDie* ⦈ for the purpose of this desugaring.

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

```
twoDiceBound12 : Lift (λ a → a ⩽ 12) twoDice
twoDiceBound12 =
  ⦇ sumOfBound6IsBound12 dieBound6 dieBound6 ⦈
relabel≅ : (t : Tree A) → Lift (λ t' → t ≅ t') (relabel t)
relabel≅ (leaf a) =
  ⦇ (λ _ → leafISO) (nothing2Prove fresh) ⦈
relabel≅ (node l r) = ⦇ nodesISO (relabel≅ l) (relabel≅ 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 $pure_L$ of type $\forall\ \{x : X\} \to (p : P\ x) \to \text{Lift}\ P\ (pure\ x)$[14]. More interesting is the $\_\circledast\_$ operator.

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

```
_⟹_ : Predicate X → Predicate Y → Predicate (X → Y)
(P ⟹ Q) f = ∀ {x : X} → P x → 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 $\_\circledast_L\_$ gives us the same in the impure context of an applicative functor F. There, we have a particular instance of $\_\circledast\_$ to encode application of an effectful function $af : F\ (X \to Y)$ to an effectful argument $ax : F\ X$. Analogously now, if ax satisfies Lift P and af satisfies Lift $(P \Longrightarrow Q)$, we can now use $\_\circledast_L\_$ to prove that $af \circledast ax$ satisfies Lift Q. This is summarised in Figure 3.

The actual implementation of $\_\circledast_L\_$ is straightforward but tedious; the witness implementation is as expected and the proof of the *corresponds* field is a long and tedious application of the applicative laws.

With $\_\circledast_L\_$ and $pure_L$ in scope as $\_\circledast\_$ and *pure* respectively, we can recycle the bracket notation as demonstrated above.[15]

4.2 Simplifying *relabel* - General Functors

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

```
relabel : Tree A → State ℕ (Tree ℕ)
relabel (leaf a) = fmap leaf fresh
relabel (node l r) = ⦇ node (relabel l) (relabel r) ⦈
```

[14]The slight difference being that we let the initial x be inferred for us automatically.

[15]Again, 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.

Luckily, we also have a lifted version of *fmap*, $fmap_L$ of type $(P \Longrightarrow Q)\ f \to \text{Lift}\ P\ af \to \text{Lift}\ Q\ (fmap\ f\ af)$. 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 $f : A \to B$, if we can prove $(P \Longrightarrow Q)\ f$ and Lift $P\ af$, we can conclude Lift Q $(fmap\ f\ af)$. With this in mind, we have the following proof.

```
relabel≅ : (t : Tree A) → Lift (λ t' → t ≅ t') (relabel t)
relabel≅ (leaf a) =
  fmap_L (λ _ → leafISO) (nothing2Prove fresh)
relabel≅ (node l r) =
  ⦇ nodesISO (relabel≅ l) (relabel≅ 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 *queensConfigs* was valid, we separately proved them to be both Fitting and Peaceful. Remember how we have then used the operator $\_,_L\_$ to combine these two into a proof of the combined property $Valid\ n = \text{Fitting}\ n \wedge \text{Peaceful}$:

```
Valid : ℕ → QConfig → Set
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)
```

Crucial for this to work is the operator $\_,_L\_ : \{fa : F\ A\} \to \text{Lift}\ P\ fa \to \text{Lift}\ Q\ fa \to \text{Lift}\ (P \wedge Q)\ fa$. 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 $\_,_L\_$ operator. Examples of such functors include List, Writer, Maybe, State and all other polynomial functors (also called container functors [1, 5]). Moreover, beside the existence of $\_,_L\_$, we have that pullback-preserving functors *and only those* have $\_,_L\_$ operators that behave universally. Universal behaviour of an operator $\_,_L\_$ characterises that

$$uncurry\ \_,_L\_ : \text{Lift}\ P\ fa \times \text{Lift}\ Q\ fa \to \text{Lift}\ (P \wedge Q)\ fa$$

is inverse to the canonical map

$$split_L : \text{Lift}\ (P \wedge Q)\ fa \to \text{Lift}\ P\ fa \times \text{Lift}\ Q\ fa$$
$$split_L\ faPQ = apply_L\ proj_1\ faPQ\ ,\ apply_L\ proj_2\ faPQ$$

where we used $apply_L$ of type $(\{a : A\} \to P\ a \to Q\ a) \to \text{Lift}\ P\ fa \to \text{Lift}\ Q\ fa$. We prove both these statements 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

	Application	Proof about application
Pure setting	$\_\$\_ : (X \to Y) \to X \to Y$	$(\lambda\ H\ p \to H\ p) : (P \Longrightarrow Q)\ f^{[X \to Y]} \to P\ x^{[X]} \to Q\ (f\ x)$
Applicative	$\_\circledast\_ : F\ (X \to Y) \to F\ X \to F\ Y$	$\_\circledast_L\_ : \mathsf{Lift}\ (P \Longrightarrow Q)\ af^{[F\ (X \to Y)]} \to \mathsf{Lift}\ P\ ax^{[F\ X]} \to \mathsf{Lift}\ Q\ (af \circledast ax)$

Figure 3. The lifted applicative bind operator ($\_\circledast_L\_$) generalises pure function predicate application, just like the applicative bind ($\_\circledast\_$) generalises regular function application.

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 $\_\to_L\_$ 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 validness 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 $[]_L : \mathsf{Lift}\ P\ []$ where P an arbitrary predicate. Moreover, it is easy to implement $\_::_L\_ : (P\ a) \to \mathsf{Lift}\ P\ xs \to \mathsf{Lift}\ P\ (a :: xs)$. And finally, we also have *invertP-cons* of type $\mathsf{Lift}\ P\ (x :: xs) \to P\ x \times (\mathsf{Lift}\ P\ xs)$. 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 *filter* function:

```
filter : {A : Set} → (A → Bool) → List A → List A
filter f [] = []
filter f (x :: xs) with f x =? true
filter f (x :: xs) | yes _ = x :: filter f xs
filter f (x :: xs) | no _  = filter f xs
```

With the $[]_L$ and $\_::_L\_$ operators, it is easy to prove, for instance, that every element in the resulting filtered list satisfies the filtered predicate:

```
filterNew : (f : A → Bool) → (as : List A) →
  Lift (λ a → f a ≡ true) (filter f as)
filterNew f [] = []_L
filterNew f (x :: as) with f x =? true
filterNew f (x :: as) | yes p = p ::_L filterNew f as
filterNew f (x :: as) | no ¬p = filterNew f as
```

Now consider a typical *range* function as defined below.

```
range : ℕ → ℕ → List ℕ
range zero zero = [ 0 ]
range (suc n) zero = []
range zero (suc m) = range zero m ++ [ suc m ]
range (suc n) (suc m) = fmap suc (range n m)
```

Proving that the resulting list is bounded below by the first argument is just as easy:[16]

```
rangeDownBound :
  (d : ℕ) → (u : ℕ) → Lift (λ x → d ⩽ x) (range d u)
rangeDownBound zero zero = [ p00[0 ⩽ 0] ]_L
rangeDownBound (suc n) zero = []_L
rangeDownBound zero (suc m) =
  rangeDownBound zero m ++_L [ pzsn[0 ⩽ suc n] ]_L
rangeDownBound (suc n) (suc m) =
  fmap_L imp[{a : ℕ} → n ⩽ a → suc n ⩽ suc a]
    (rangeDownBound n m)
```

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

```
_ ++_L _ : Lift P xs → Lift P ys → Lift P (xs ++ ys)
_ ++_L _ {xs = []}     _[Lift P []]      ysP[Lift P ys] = ysP
_ ++_L _ {xs = a :: as} aasP[Lift P (a :: as)] ysP
  with invertP-cons aasP
_ ++_L _ {xs = a :: as} aasP             ysP
  | pa[P a] , asP[Lift P as] = pa ::_L (asP ++_L ysP)
```

Finally, for Lists, we can also prove the $\_\to_L\_$ operator discussed earlier:

```
_→L_ : Lift P as → Lift Q as → Lift (P ∧ Q) as
_→L_ {as = []}     _[Lift P []]      _[Lift Q []] = []_L
_→L_ {as = x :: xs} xxsP[Lift P (x :: xs)] xxsQ[Lift Q (x :: xs)]
  with invertP-cons xxsP | invertP-cons xxsQ
_→L_ {as = x :: xs} xxsP             xxsQ
  | px[P x] , xsP[Lift P xs] | qx[Q x] , xsQ[Lift Q xs] =
    (px , qx) ::_L (xsP ,_L xsQ)
```

The use of *invertP-cons* in the last two examples is still a bit suboptimal. Ideally, we would like to reuse the pattern matching syntax on lifted property proofs, rewriting $\_ ++_L \_$ to something like the following:

[16] Here, we have $[\_]_L$ the singleton operator of type $P\ a \to \mathsf{Lift}\ P\ [a]$. As $[\_]$ is easily implemented by $\lambda\ a \to a :: []$, we have $[\_]_L$ trivially implemented as $\lambda\ pa \to pa ::_L []_L$.

```
_++L_ : Lift P xs → Lift P ys → Lift P (xs ++ ys)
_++L_ {xs = xs}     xsP          ysP by indListL xsP
_++L_ {xs = []}     []L[Lift P []]          ysP = ysP
_++L_ {xs = a :: as} px[P a] ::L asP[Lift P as] ysP =
  pa ::L (asP ++L ysP)
```

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 $1000\ /\ (13 - n)$, where n is the number of pips he/she throws. A naive implementation might look like this:

```
fine : IO ℕ
fine = fmap amount twoDice where
  amount : ℕ → ℕ
  amount n = case (n ⩽? 12) of
    λ {(yes p[n ⩽ 12]) → (1000 div (13 - n , easy[13 - n ≢ 0]));
       (no ¬p[¬ (n ⩽ 12)]) → ??} where
         _div_ : ℕ → (Σ[ n ∈ ℕ ] n ≢ 0) → ℕ
         n div (d , p) = ...
```

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

```
fmapR : {fa : F A} → (Σ A P → B) → (Lift P fa) → F B
fmapR fR lp = fmap fR (witness lp)
```

[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).

Using $fmap_R$, we now have the following:

```
fine : IO ℕ
fine = fmapR amountR twoDiceBounded12 where
  amountR : Σ[ n ∈ ℕ ] n ⩽ 12 → ℕ
  amountR (n , p) = 1000 div (13 - n , easy[13 - n ≢ 0])
```

It is in fact easy to prove that this new implementation of *fine* is equal to the old one. Intuitively, this is because for $d : \mathbb{N}$ and $p : n \leqslant 12$, we have that *amount d* is equal to $amount_R\ (d\ ,\ p)$. In general, it is easily proven formally that *fmap f*[A → B] *fa*[F A] equals $fmap\ f_R$[Σ A P → B] *lp*[Lift P fa] if f agrees with f_R on P.[18]

We have an analogous operator for _⊛_.

```
_⊛R_ : {ax : F X} → F (Σ X P → Y) → Lift P ax → F Y
fR ⊛R axP = fR ⊛ witness axP
```

So given an applicative value $ax : F\ X$ for which we have a proof *axP* of Lift $P\ ax$, it is enough to have f_R of type $F\ (\Sigma\ X\ P \to Y)$ instead of $F\ (X \to Y)$.

And finally, the analogous operator for _≫=_.

```
_≫=R_ : {ma : M A} →
  (Lift P ma) → (Σ A P → M B) → M B
lp ≫=R fR = witness lp ≫= fR
```

That is, given a monadic value $ma : M\ A$ and a proof of Lift $P\ ma$, when binding *ma* with a function $f : A \to M\ B$, it is enough to define $f\ a$ given that a satisfies P. That is, it is enough to define f of type $\Sigma\ A\ P \to M\ B$.

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.

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 Lift, $\_\gg\!=_L\_$, $return_L$, $pure_L$, $\_\circledast_L\_$ and $fmap_L$.

Moreover, we can easily reuse existing notation to structure compositions with $\_\gg\!=_L\_$ (**do**-notation), and $\_\circledast_L\_$ together with $pure_L$ (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.

[18] That is, if for all $(a\ ,\ p) : \Sigma\ A\ P$ we have that $f\ a$ equals $f_R\ (a\ ,\ p)$.

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 correspoding 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 *interestingOperation* : IO ⊤ 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 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 *twoDice*) extrinsically, as we did in *twoDiceBound12* : Lift ($\lambda\ a \rightarrow a \leqslant 12$) *twoDice* IO ℕ. Ornamentation libraries, on the other hand, are often designed to combine an extrinsic proof like *twoDiceBound12* with the underlying implementation *twoDice* into an intrinsically correct implementation.

Secondly, the results in the ornament literature simply do not seem to be applicable in our setting. Our Lift operator is designed to reason about forgetful maps of the form $proj_1 : \Sigma\ A\ P \rightarrow A$. We like to point out that, since it is possible in dependent type theory to define the inverse image of a function, any function $f : B \rightarrow A$ factors as $B \cong \Sigma(x : A)\ g^{-1}(x) \rightarrow A$. 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

```
cumbersomeProof : twoDice ≡ fmap proj₁ aWitness
cumbersomeProof =
  begin
    (do x ← die
        y ← die
        return (x + y))
  ≡ ⟨ cong (flip _>>=_ _) (corresponds dieBound6) ⟩
    (do x ← fmap proj₁ (witness dieBound6)
        y ← die
        return (x + y))
  ≡ ⟨ fmap-bind _ _ _ ⟩
    (do (x , px) ← witness dieBound6
        y ← die
        return (x + y))
  ≡ ⟨ cong (_>>=_ (witness dieBound6))
           (funext λ _ → cong (flip _>>=_ _)
                              (corresponds dieBound6)) ⟩
    (do (x , px) ← witness dieBound6
        y ← fmap proj₁ (witness dieBound6)
        return (x + y))
  ≡ ⟨ cong (_>>=_ _) (funext λ _ → fmap-bind _ _ _) ⟩
    (do (x , px) ← witness dieBound6
        (y , py) ← witness dieBound6
        return (x + y))
```

[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.

```
≡ ⟨ cong (_>>=_ _) (funext λ a → cong (_>>=_ _)
                    (funext λ _ → sym (fmap-return proj₁ _))) ⟩
  (do (x , px) ← witness dieBound6
      (y , py) ← witness dieBound6
      fmap  (proj₁ {B = λ n → n ⩽ 12})
            (return (x + y , sumOfBound6IsBound12 px py)))
≡ ⟨ cong (_>>=_ _) (funext λ _ → sym (fmap-move-bind _ _ _)) ⟩
  (do (x , px) ← witness dieBound6
      fmap (proj₁ {B = λ n → n ⩽ 12})
        do (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.

```
fmap-bind g f mx :
  (fmap f mx) >>= g ≡ mx >>= (g ∘ f)
fmap-return f a :
  fmap f (return a) ≡ return (f a)
fmap-move-bind f ma g :
  (fmap g (ma >>= f)) ≡ (ma >>= (fmap g ∘ f))
```

B Manual Proof *relabel*

```
relabel≅ : (t : Tree A) → (n : ℕ) → evalState (relabel t) n ≅ t
relabel≅ (leaf a) n    = leafISO
relabel≅ (node l r) n  =
  nodesISO (relabel≅ l n) (relabel≅ r (execState (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.

C On Pullback Preservation

In this appendix, we show that a well-behaved operator $\_\rightarrow_L\_$ 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.[20]

C.1 Definitions

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

Definition C.1. A (propositionally) commutative diagram

$$\begin{array}{ccc} T & \xrightarrow{p_1} & A \\ {\scriptstyle p_2}\downarrow & & \downarrow{\scriptstyle f} \\ B & \xrightarrow[g]{} & C \end{array}$$

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

$$\Sigma[\, a \in A \,]\, \Sigma[\, b \in B \,]\, \Sigma[\, c \in C \,]\, (f\ a \equiv c) \times (g\ b \equiv c)$$

with the maps $p_1 : T \to A$ and $p_2 : T \to B$ corresponding to the appropriate projections from this type of quintuples.

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

$$\begin{array}{ccc} F\ T & \xrightarrow{fmap\ p_1} & F\ A \\ {\scriptstyle fmap\ p_2}\downarrow & & \downarrow{\scriptstyle fmap\ f} \\ F\ B & \xrightarrow[fmap\ g]{} & F\ C \end{array}$$

that is also a pullback square.

C.2 Combining Properties Requires Pullback Preservation

We seek to prove the following:

Proposition C.3. *The function $split_L$ (§4.3) is an isomorphism if and only if F preserves pullbacks.*

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

$$\begin{array}{ccc} \Sigma(F\ C)(\mathsf{Lift}\ (P \wedge Q)) & \longrightarrow & \Sigma(F\ C)(\mathsf{Lift}\ P) \\ \downarrow & & \downarrow{\scriptstyle proj_1} \\ \Sigma(F\ C)(\mathsf{Lift}\ Q) & \xrightarrow{proj_1} & F\ C. \end{array}$$

Now this diagram is a pullback square if and only if $\Sigma\ (F\ C)\ (\mathsf{Lift}\ (P \wedge Q))$ is isomorphic to $\Sigma\ (F\ C)\ (\mathsf{Lift}\ P \wedge \mathsf{Lift}\ Q)$ in a manner compatible with the canonical maps to $\Sigma\ (F\ C)\ (\mathsf{Lift}\ P)$ and $\Sigma\ (F\ C)\ (\mathsf{Lift}\ Q)$. In other words, we require that $\mathsf{Lift}\ (P \wedge Q)\ fc$ be isomorphic to $\mathsf{Lift}\ P\ fc \times \mathsf{Lift}\ Q\ fc$ in a manner compatible with the projections to $\mathsf{Lift}\ P\ fc$ and $\mathsf{Lift}\ Q\ fc$. This compatibility criterion equivalently says that one side of the isomorphism is given by $split_L$. This proves the proposition. □

[20] Throughout the section, we assume uniqueness of identity proofs.

[21] In the right hand type of this isomorphism, the *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 *witness* field of type $F\ (\Sigma\ X\ R)$ as actual information.

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 Lift $(P\ \wedge\ Q)\ ma^{[M\ A]}$ is not necessarily isomorphic to Lift $P\ ma\ \times$ Lift $Q\ ma$. As a warm-up, let us consider what Lift even means for the continuation monad.

A value $ma : M\ A$ 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, ma may call the continuation zero or multiple times and combine the results.

If we have a value $ma : M\ A$ and a function $f : A \to B$, then $fmap\ f\ ma : M\ B$ will take a continuation $k : B \to R$, compose it with f and feed the result to ma. A value $ma : M\ A$ satisfies Lift P if it arises by applying $fmap\ proj_1 : M\ (\Sigma\ A\ P) \to M\ A$ to some computation of type $M\ (\Sigma\ A\ P)$. In other words, a proof of Lift $P\ ma$ indicates that, even though ma 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, Lift P also becomes proof-relevant, but in a blown-up manner. Indeed, $ma : M\ A$ takes 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 $ma : M\ A$ satisfies Lift P needs to respect the behaviour of ma 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 ma can become enormous.

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

$$\begin{aligned}&\Sigma\ (M\ \top)\ (\mathsf{Lift}\ P\ \times\ \mathsf{Lift}\ Q)\\ &\quad\hookrightarrow\ \Sigma\ (M\ \top)\ (\mathsf{Lift}\ P)\ \times\ \Sigma\ (M\ \top)\ (\mathsf{Lift}\ Q)\\ &\qquad\cong M\ \mathsf{Bool}\ \times\ M\ \mathsf{Bool}\end{aligned}$$

where $\hookrightarrow$ denotes an injection. Thus, if M were to preserve pullbacks, then $M\ (\mathsf{Bool}\ \times\ \mathsf{Bool})$ should have lower cardinality than $M\ \mathsf{Bool}\ \times\ M\ \mathsf{Bool}$. However, the cardinality of the former type is $2^{2^{2\cdot 2}} = 2^{16}$, whereas the latter type has cardinality $2^{2^2} \cdot 2^{2^2} = 2^8$ 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

[1] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing strictly positive types. *Theor. Comput. Sci.* 342, 1 (2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002

[2] Agda Developers. 2019. Syntactic Sugar - Do notation - Agda 2.6.0 documentation. https://agda.readthedocs.io/en/v2.6.0/language/syntactic-sugar.html#do-notation. (Accessed on 05/14/2019).

[3] Agda Developers. 2019. Syntactic Sugar - Idiom brackets - Agda 2.6.0 documentation. https://agda.readthedocs.io/en/v2.6.0/language/syntactic-sugar.html#idiom-brackets. (Accessed on 05/14/2019).

[4] Pierre-Évariste Dagand. 2017. The essence of ornaments. *J. Funct. Program.* 27 (2017). https://doi.org/10.1017/S0956796816000356

[5] Nicola Gambino and Martin Hyland. 2003. Wellfounded Trees and Dependent Polynomial Functors. In *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*. 210–225. https://doi.org/10.1007/978-3-540-24849-1_14

[6] Jeremy Gibbons and Ralf Hinze. 2011. Just Do It: Simple Monadic Equational Reasoning. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 2–14. https://doi.org/10.1145/2034773.2034777

[7] Graham Hutton and Diana Fulger. 2008. Reasoning about effects: Seeing the wood through the trees. In *Pre-proceedings of the Ninth Symposium on Trends in Functional Programming*.

[8] Conor McBride. 2010. Ornamental algebras, algebraic ornaments. (2010).

[9] Conor McBride and James McKinna. 2004. The view from the left. *Journal of Functional Programming* 14, 1 (2004), 69111. https://doi.org/10.1017/S0956796803004829

[10] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of functional programming* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

[11] U. Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. Dissertation. Chalmers.

[12] Morten Heine Sørensen and Pawel Urzyczyn. 2006. *Lectures on the Curry-Howard isomorphism*. Vol. 149. Elsevier.

[13] Wouter Swierstra. 2009. A Hoare logic for the state monad. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 440–451. https://doi.org/10.1007/978-3-642-03359-9_30