Equational Reasoning with Applicative Functors

Andreas Lochbihler and Joshua Schneider

Institute of Information Security, Department of Computer Science, ETH Zurich
andreas.lochbihler@inf.ethz.ch, joshuas@student.ethz.ch

Abstract. In reasoning about effectful computations, it often suffices to focus on the effect-free parts. We present a package for automatically lifting equations to effects modelled by applicative functors. It exploits properties of the concrete functor thanks to a modular classification based on combinators. We formalise the meta theory and demonstrate the usability of our Isabelle/HOL package with two case studies. This is a first step towards practical reasoning with effectful computations.

1 Introduction

In functional languages, effectful computations are often captured by monads. Monadic effects also feature in many verification projects and formalisations (e.g., [4,8,20,21,22]). The reasoning support is typically tailored to the specific monad under consideration. Thus, the support must be designed and implemented anew for every monad. In contrast, reasoning about monadic effects in general has been largely neglected in the literature on both mechanised and pen-and-paper reasoning—one notable exception is [11]. One reason might be that the monadic operators can be used in too many different ways for one generic technique covering all of them.

Applicative functors (a.k.a. *idioms*) [25] are a less well-known alternative for modelling effects. Compared to monads, sequencing is restricted in idioms such that the effects of the second computation may not depend on the result of the first. In return, the structure of the computation becomes fixed. So, idiomatic expressions can be analysed statically and reasoned about. Every monad is an applicative functor and many real-world monadic programs can be expressed idiomatically [24].

In reasoning about effectful computations, only some steps involve reasoning about the effects themselves. Typically, many steps deal with the effect-free parts of the computations. In this case, one would like to get the effects out of the way, as they needlessly complicate the reasoning. *Lifting*, which transfers properties from the pure world to the effectful, can formally capture such an abstraction.

In this paper, we present a new package to automate the lifting for equational reasoning steps over effects modelled by applicative functors. We choose applicative functors (rather than monads) because they enjoy nicer properties: the computational structure is fixed and they compose. We focus on equational reasoning as it is the fundamental reasoning principle in the verification of functional programs. The theory is inspired by Hinze's work on lifting [15] (see §5 for a comparison). We formalised, refined and implemented the theory in Isabelle/HOL. Our work is not specific to Isabelle; any HOL-based proof assistant could have been used.

Our package consists of two parts (see §1.2 for a usage example). First, the command **applicative** allows users to register HOL types as applicative functors. Second, two proof methods *applicative-nf* and *applicative-lifting* implement the lifting of equations as backwards-style reasoning steps over registered functors.

Crucially, lifting is generic in the applicative functor. That is, the implementation works uniformly for any applicative functor by relying only on the laws for applicative functors (§3). Yet, not all equations can be lifted in all idioms. If the functor provides additional laws like commutativity or idempotence of effects, then more equations can be lifted. So, it makes sense to specialise the reasoning infrastructure to some extent. To strike a balance between genericity and applicability, we identified classes of idioms for which the liftable equations can be characterised syntactically (§4). We achieve modularity in the implementation by using the same algorithm schema (borrowed from combinatory logic) for all classes.

Moreover, we have formalised a core idiomatic language and most of the meta-theory in HOL itself (§2). In fact, we manually derived the implementation of the package from this formalisation. Thus, not only does the inference kernel check every step of our proof method, but we know that the algorithm is indeed correct.

Two small case studies on tree labelling (§1.2) and the Stern-Brocot tree (§4.4) demonstrate the reasoning power of the package and indicate directions for future extension (§6). The implementation and the examples are available online [23,9].

1.1 Background on Applicative Functors

McBride and Paterson [25] introduced the concept of applicative functors to abstract a recurring theme they observed in the programming language Haskell. An applicative functor (or idiom) is a unary type operator F (here written postfix) with two polymorphic operations $\mathsf{pure}_\mathsf{F} :: \alpha \Rightarrow \alpha\ \mathsf{F}$ and $(\diamond)_\mathsf{F} :: (\alpha \Rightarrow \beta)\ \mathsf{F} \Rightarrow \alpha\ \mathsf{F} \Rightarrow \beta\ \mathsf{F}$. The functor F models the effects of a computation with result type α, $\mathsf{pure}_\mathsf{F}\ x$ represents a value x without effects, and $f \diamond_\mathsf{F} x$ applies the function resulting from the computation f to the value of the computation x and combines their effects. That is, $(\diamond)_\mathsf{F}$ lifts function application to effectful computations. When the functor F is clear from the context, we omit the subscript F. The infix operator $(\diamond)$ associates to the left like function application. Idioms must satisfy the following four laws called the applicative laws.

$$\begin{aligned} \mathsf{pure}_\mathsf{F}\ \mathsf{id} \diamond_\mathsf{F} x &= x && \text{(identity)}\\ \mathsf{pure}_\mathsf{F}\ (\circ) \diamond_\mathsf{F} f \diamond_\mathsf{F} g \diamond_\mathsf{F} x &= f \diamond_\mathsf{F} (g \diamond_\mathsf{F} x) && \text{(composition)}\\ \mathsf{pure}_\mathsf{F}\ f \diamond_\mathsf{F} \mathsf{pure}_\mathsf{F}\ x &= \mathsf{pure}_\mathsf{F}\ (f\ x) && \text{(homomorphism)}\\ f \diamond_\mathsf{F} \mathsf{pure}_\mathsf{F}\ x &= \mathsf{pure}_\mathsf{F}\ (\lambda f.\ f\ x) \diamond_\mathsf{F} f && \text{(interchange)} \end{aligned}$$

Every monad is an applicative functor—take $\mathsf{pure} = \mathsf{return}$ and $f \diamond x = f \gg\!= (\lambda f'.\ x \gg\!= (\lambda x'.\ \mathsf{return}\ (f'\ x')))$—but not vice versa. Thus, applicative functors are more general. For example, streams (**codatatype** α stream $= \alpha \prec \alpha$ stream) host an idiom (1) which cannot be extended to a monad [25]. More examples are given in App. A.

$$\mathsf{pure}\ x = x \prec \mathsf{pure}\ x \qquad (f \prec fs) \diamond (x \prec xs) = f\ x \prec (fs \diamond xs) \tag{1}$$

The more restrictive signature of $(\diamond)$ imposes a fixed structure on the computation. In fact, any expression built from the applicative operators can be transformed into canonical form $\mathsf{pure}\ f \diamond x_1 \diamond \ldots \diamond x_n$ using the applicative laws (see §3.1), namely "a single pure function [...] applied to the effectful parts in depth-first order" [25].

1.2 Motivating Example: Tree Labelling

To illustrate lifting and its benefits, we consider the problem of labelling a binary tree with distinct numbers. This example has been suggested by Hutton and Fulger [19]; Gibbons et al. [10,11] explore it further. The classic solution shown below uses a state monad with an operation $\mathsf{fresh} = \mathbf{do}\ \{\ x \leftarrow \mathsf{get};\ \mathsf{put}\ (x+1);\ \mathsf{return}\ x\ \}$ to generate the labels, where we use Haskell-style **do** notation.

$$\begin{aligned}
&\mathbf{datatype}\ \alpha\ \mathsf{tree} = \mathsf{L}\ \alpha \mid \mathsf{N}\ (\alpha\ \mathsf{tree})\ (\alpha\ \mathsf{tree})\\
&\mathsf{lbl}\ (\mathsf{L}\ \_) \quad\ = \mathbf{do}\ \{\ x \leftarrow \mathsf{fresh};\ \mathsf{return}\ (\mathsf{L}\ x)\ \}\\
&\mathsf{lbl}\ (\mathsf{N}\ l\ r) = \mathbf{do}\ \{\ l' \leftarrow \mathsf{lbl}\ l;\ r' \leftarrow \mathsf{lbl}\ r;\ \mathsf{return}\ (\mathsf{N}\ l'\ r')\ \}
\end{aligned}$$

Hutton and Fulger expressed lbl concisely in the state idiom as follows.

$$\mathsf{lbl}\ (\mathsf{L}\ \_) = \mathsf{pure}\ \mathsf{L} \diamond \mathsf{fresh} \qquad\qquad \mathsf{lbl}\ (\mathsf{N}\ l\ r) = \mathsf{pure}\ \mathsf{N} \diamond \mathsf{lbl}\ l \diamond \mathsf{lbl}\ r$$

The task is to prove that the labels in the resulting tree are distinct, i.e., $\mathsf{pure}\ \mathsf{lbls} \diamond \mathsf{lbl}\ t$ returns only distinct lists where the function lbls given below extracts the labels in a tree and $(+\!\!+)$ concatenates two lists.

$$\mathsf{lbls}\ (\mathsf{L}\ x) = [x] \qquad\qquad \mathsf{lbls}\ (\mathsf{N}\ l\ r) = \mathsf{lbls}\ l +\!\!+ \mathsf{lbls}\ r \tag{2}$$

As a warm-up, we prove that the list of labels in a relabelled tree equals a relabelling of the list of labels in the original tree. Formally, define relabelling for lists by $\mathsf{lbl}'\ [\,] = \mathsf{pure}\ [\,]$ and $\mathsf{lbl}'\ (x \cdot l) = \mathsf{pure}\ (\cdot) \diamond \mathsf{fresh} \diamond \mathsf{lbl}'\ l$. We show $\mathsf{pure}\ \mathsf{lbls} \diamond \mathsf{lbl}\ t = \mathsf{lbl}'\ (\mathsf{lbls}\ t)$ by induction on t. In each case, we first unfold the defining equations for lbl, lbl′ and lbls, possibly the induction hypotheses and the auxiliary fact $\mathsf{lbl}'\ (l +\!\!+ l') = \mathsf{pure}\ (+\!\!+) \diamond \mathsf{lbl}'\ l \diamond \mathsf{lbl}'\ l'$, which we prove similarly by induction on l and lifting the defining equations of $(+\!\!+)$. Then, the two subgoals below remain.

$$\begin{aligned}
\mathsf{pure}\ \mathsf{lbls} \diamond (\mathsf{pure}\ \mathsf{L} \diamond \mathsf{fresh}) \qquad\quad &= \mathsf{pure}\ (\cdot) \diamond \mathsf{fresh} \diamond \mathsf{pure}\ [\,]\\
\mathsf{pure}\ \mathsf{lbls} \diamond (\mathsf{pure}\ \mathsf{N} \diamond \mathsf{lbl}\ l \diamond \mathsf{lbl}\ r) &= \mathsf{pure}\ (+\!\!+) \diamond (\mathsf{pure}\ \mathsf{lbls} \diamond \mathsf{lbl}\ l) \diamond (\mathsf{pure}\ \mathsf{lbls} \diamond \mathsf{lbl}\ r)
\end{aligned} \tag{3}$$

Observe that they are precisely liftings of (2). We recover the latter equations if we remove all pures, replace $\diamond$ by function application and generalise fresh to a variable x.

Our new proof method *applicative-nf* performs this transition after the state idiom has been registered with the package using the command **applicative**. Registration takes the name of the idiom (here "state") and HOL terms for the applicative operations. Then, the applicative laws must be proven, which the proof method *auto* automates in this case.

$$\mathbf{applicative}\ \text{state}\ \mathbf{for}\ \mathsf{pure} : \mathsf{pure}_{\mathsf{state}} \qquad \mathsf{ap} : (\diamond)_{\mathsf{state}} \qquad \mathbf{by}(\textit{auto simp:}\ (\diamond)_{\mathsf{state}}\text{-def})$$

After the registration, both subgoals in (3) are discharged automatically using the new proof method *applicative-nf* and term rewriting. The crucial point is that we have never unfolded the definitions of the state idiom or fresh. Thus, we do not break the abstraction.

Let us now return to the actual task. The main difficulty is stating distinctness of labels without looking into the state monad, as this would break the abstraction. Gibbons and Hinze [11] suggested to use an error monad; we adapt their idea to idioms. We consider the composition of the state idiom with the error idiom derived from the option monad (see App. A). Then, the correctness of fresh is expressed abstractly as

$$\forall n.\ \mathsf{pure}_{\mathsf{state}}\ (\mathsf{assert}\ \mathsf{distinct}) \diamond \mathsf{nfresh}\ n = \mathsf{nfresh}\ n \tag{4}$$

```
lemma 1: assumes nfresh: ∀n. pure_state (assert distinct) ⋄ nfresh n = nfresh n
             shows pure_state dlbls ⋄_state lbl t = nfresh (lvs t)
proof (induction t)
  show pure dlbls ⋄ lbl (L x) = nfresh (lvs (L x)) for x
    unfolding lbl.simps lvs.simps repeat.simps by applicative-nf simp
next
  fix l r
  assume IH₁ : pure dlbls ⋄ lbl l = nfresh (lvs l) and IH₂ : pure dlbls ⋄ lbl r = nfresh (lvs r)
  let ?f = λl r. pure ⌈(++)⌉ ⋄ (assert ⌈disjoint⌉ (pure Pair ⋄ l ⋄ r))
  have pure dlbls ⋄ lbl (N l r) = pure ?f ⋄ (pure dlbls ⋄ lbl l) ⋄ (pure dlbls ⋄ lbl r)
    unfolding lbl.simps by applicative-nf simp
  also have ... = pure ?f ⋄ (pure (assert distinct) ⋄ nfresh (lvs l)) ⋄
                            (pure (assert distinct) ⋄ nfresh (lvs r))
    unfolding IH₁ IH₂ nfresh ..
  also have ... = pure (assert distinct) ⋄ nfresh (lvs (N l r))
    unfolding lvs.simps repeat-plus by applicative-nf simp
  also have ... = nfresh (lvs (N l r)) by (rule nfresh)
  finally show pure dlbls ⋄ lbl (N l r) = nfresh (lvs (N l r)) .
qed
```

Fig. 1. Isar proof of Lemma 1. Our proof method is highlighted in grey. X.simps refers to the defining equations of the function X, and repeat-plus to distributivity of repeat over $(+)$.

where $\mathsf{pure}_{\mathsf{state}}$ lifts the assertion from the error idiom to the state-error idiom. Further, the function nfresh n = $\mathsf{pure}_{\mathsf{state}}$ $\mathsf{pure}_{\mathsf{option}}$ $\diamond$ repeat n fresh produces n fresh symbols, where repeat n x repeats the computation x for n times and collects the results in a list. Again, observe that $\mathsf{pure}_{\mathsf{state}}$ $\mathsf{pure}_{\mathsf{option}}$ embeds the computation repeat n fresh from the state idiom into the state-error idiom.

Moreover, in the error idiom, we can combine the extraction of labels from a tree and the test for disjointness in the subtrees of a node into a single function dlbls :: α tree $\Rightarrow$ α list option. Here, disjoint $l\ l' \longleftrightarrow$ set $l \cap$ set $l' = \emptyset$ tests whether the lists l and l' are disjoint and $\lceil f \rceil$ uncurries the function f.

$$\begin{aligned}\mathsf{dlbls}\ (\mathsf{L}\ x) &= \mathsf{pure}\ [x]\\ \mathsf{dlbls}\ (\mathsf{N}\ l\ r) &= \mathsf{pure}\ \lceil(+\!\!+)\rceil \diamond (\mathsf{assert}\ \lceil\mathsf{disjoint}\rceil \diamond (\mathsf{pure\ Pair} \diamond \mathsf{dlbls}\ l \diamond \mathsf{dlbls}\ r))\end{aligned}$$

Finally, we can state correctness of lbl as follows (lvs t counts the leaves in t).

Lemma 1. *If* (4) *holds, then* $\mathsf{pure}_{\mathsf{state}}\ \mathsf{dlbls} \diamond_{\mathsf{state}} \mathsf{lbl}\ t = \mathsf{nfresh}\ (\mathsf{lvs}\ t)$.

Figure 1 shows the complete proof in Isar. The base case for L merely lifts the equation dlbls (L x) = pure $[x]$, which lives in the option idiom, to the state idiom. As our package performs the lifting, the proof in Isabelle is automatic. The case for N requires four reasoning steps, two of which involve lifting identities from the error idiom to the state-error idiom; the other steps apply the induction hypotheses and the assumption (4). This compares favourably with Gibbons' and Hinze's proof for the monadic version [11], which requires one and a half columns on paper and has not been checked mechanically.

2 Modelling Applicative Functors in HOL

We model applicative functors in HOL twice. In our first model, a functor F appears as a family of HOL types $\alpha\ \mathsf{F}$ with HOL terms for the applicative operations. The package implementation rests on this basis. The second model is used for the meta theory: we formalise a deep embedding of the idiomatic language in order to establish the proof procedure and argue for its correctness.

Applicative functors in HOL. The general notion of an applicative functor cannot be expressed in HOL for the same reasons as for monads [16]: there are no type constructor variables in HOL, and the applicative operations occur with several different type instances in the applicative laws. This implies that the first model cannot be based on definitions and theorems that are generic in the functor. Instead, we necessarily always work with a concrete applicative functor. The corresponding terms and theorems can be expressed in HOL, as HOL constants may be polymorphic. Our package keeps track of a set of applicative functors. Thus, the user must register a functor using the command **applicative** before the package can use it. During the registration, the user must prove the applicative laws (and possibly additional properties, see §4).

The package follows the traditional LCF style of prover extensions. The proof procedures are written in ML, where they analyse the HOL terms syntactically and compose the inference rules accordingly. This approach shifts the problem of (functor) polymorphism to the program level, where it is easily solved. As usual, the logical kernel ensures that all the proofs are sound. Conversely, the proof procedures themselves should be correct, namely terminate and never attempt to create an invalid proof. Arguments to support this are in turn supplied by the meta theory studied in the second model.

Deep embedding of applicative functors. The second model serves two purposes: it formalises the meta theory and we derive our package implementation from it. The model separates the notion of idiomatic terms from the concrete applicative functor and represents them syntactically. Idiomatic terms consist of pure terms $\mathsf{Pure}\ t$, opaque terms $\mathsf{Opq}\ x$, and applications $t_1 \mathbin{\overline{\diamond}} t_2$ of two idiomatic terms.

$$\textbf{datatype}\ \alpha\ \mathsf{iterm} = \mathsf{Pure\ term} \mid \mathsf{Opq}\ \alpha \mid \alpha\ \mathsf{iterm} \mathbin{\overline{\diamond}} \alpha\ \mathsf{iterm}$$

Opaque terms represent impure (effectful) values of the functor, or variables in general; their representation is left abstract as it is irrelevant to most definitions in the model. In contrast, Pure's argument needs some structure such that the applicative laws can be stated. To that end, we reuse Nipkow's formalisation of the untyped λ-calculus with de Bruijn indices [26]: **datatype** $\mathsf{term} = \mathsf{Var\ nat} \mid \mathsf{term} \mathbin{\$} \mathsf{term} \mid \mathsf{Abs\ term}$. For readability, we write such terms as abstractions with named variables, e.g. $\overline{\lambda}x.\ x \equiv \mathsf{Abs}\ (\mathsf{Var}\ 0)$, where the notation $\overline{\lambda}$ distinguishes them from HOL terms. The relation $\simeq_{\beta\eta}$ on term denotes equivalence of λ-terms due to $\beta\eta$-conversion.

The model ignores types, as they are not needed for the meta theory. Thus, we cannot express type safety of our algorithms, either. However, we do not foresee any difficulties in extending our model with types, e.g., in the style of Berghofer [1].

Equational reasoning on the applicative functor is formalised by an equivalence relation $\simeq$ on α iterm. It is the least equivalence relation satisfying the rules in Fig. 2. They represent the applicative laws and the embedding of $\beta\eta$-equivalence on λ-terms.

$$\frac{}{\mathsf{Pure}\ \overline{\mathbf{B}}\ \overline{\diamond}\ f\ \overline{\diamond}\ g\ \overline{\diamond}\ x \simeq f\ \overline{\diamond}\ (g\ \overline{\diamond}\ x)}\ (\text{composition}) \qquad \frac{}{\mathsf{Pure}\ \overline{\mathbf{I}}\ \overline{\diamond}\ x \simeq x}\ (\text{identity})$$

$$\frac{}{\mathsf{Pure}\ f\ \overline{\diamond}\ \mathsf{Pure}\ x \simeq \mathsf{Pure}\ (f\ \$\ x)}\ (\text{homomorphism}) \qquad \frac{t \simeq_{\beta\eta} t'}{\mathsf{Pure}\ t \simeq \mathsf{Pure}\ t'}\ (\text{cong-Pure})$$

$$\frac{}{f\ \overline{\diamond}\ \mathsf{Pure}\ x \simeq \mathsf{Pure}\ (\overline{\lambda} f.\ f\ \$\ x)\ \overline{\diamond}\ f}\ (\text{interchange}) \qquad \frac{t_1 \simeq t'_1 \qquad t_2 \simeq t'_2}{t_1\ \overline{\diamond}\ t_2 \simeq t'_1\ \overline{\diamond}\ t'_2}\ (\text{cong-}\overline{\diamond})$$

Fig. 2. Equivalence of idiomatic terms, where $\overline{\mathbf{I}} \equiv \overline{\lambda} x.\ x$ and $\overline{\mathbf{B}} \equiv \overline{\lambda} f\ g\ x.\ f\ \$\ (g\ \$\ x)$.

Clearly, if we interpret two idiomatic terms s and t in an applicative functor F in the obvious way as s' and t', and if s' and t' are type correct, then $s \simeq t$ implies $s' = t'$.

Connection between the two models. It is natural to ask how the verified meta model could be leveraged as part of the proofs in the shallow embedding. We decided to leave the connection informal and settled for the two-model approach for now. Formally bridging the gap is left as future work, for which two approaches appear promising.

Computational reflection makes the correspondence between objects of the logic and their representation explicit by an interpretation function with correctness theorems [3]. For idiomatic terms, interpretation cannot be defined directly in HOL, as a single term may refer to an arbitrary collection of types. Schropp and Popescu [29] circumvent this limitation by modelling the type universe as a single type parameter to the meta theory; additional machinery injects the actual types into this universe and transfers the obtained results. Similar injections could be crafted for idiomatic terms, but the connection would have to be built anew upon each usage. It is not clear that the overhead incurred is compensated by the savings in avoiding the replay of the lifting proof in the shallow embedding.

Alternatively, Tuong and Wolff [30] model the Isabelle API in HOL syntactically and can thus generate code for packages from the HOL formalisation. This could be used to express our proof tactics as HOL terms. Then, we could formally verify them and thus obtain a verified package. Before we can apply this technique in our setting, two challenges must be solved. First, their model merely defines the syntax, but lacks a semantics for the API. Hence, one would first have to model the semantics and validate it. Second, the additional code for usability aspects like preserving the names of bound variables would also have to be part of the HOL terms. This calls for some notion of refinement or abstraction, which is not yet available; otherwise, these parts would clutter the formalisation.

3 Lifting with Applicative Functors

The pure_F operation of an applicative functor F lifts values of type α to $\alpha\ \mathsf{F}$. If we view HOL terms as functions of their free variables, we can also lift terms via the following syntactic modification: free variables of type α are replaced by those of type $\alpha\ \mathsf{F}$, constants and abstractions[1] are embedded in pure_F, and function application is replaced by $(\diamond)_\mathsf{F}$. Lifting extends to equations, where both sides are treated

[1] As lifting wraps the types of *free* variables in F, it does not look into abstractions, but treats them like constants. For example, $\lambda x.\ x :: \alpha \Rightarrow \alpha$ is lifted to $\mathsf{pure}\ (\lambda x.\ x) :: (\alpha \Rightarrow \alpha)\ \mathsf{F}$ rather than $\lambda x.\ x :: \alpha\ \mathsf{F} \Rightarrow \alpha\ \mathsf{F}$. Thus, lifting effectively operates on first-order terms.

$$(\mathsf{Pure}\ x)\!\downarrow\ = \mathsf{Pure}\ x \qquad (\mathsf{Opq}\ x)\!\downarrow\ = \mathsf{Pure}\ \bar{\mathbf{I}}\ \bar{\diamond}\ \mathsf{Opq}\ x \qquad (t\ \bar{\diamond}\ t')\!\downarrow\ = \mathsf{norm_{nn}}\ (t\!\downarrow)\ (t'\!\downarrow)$$

$$\begin{aligned}
&\mathsf{norm_{nn}}\ n\ (\mathsf{Pure}\ x) = \mathsf{norm_{pn}}\ ((\bar{\lambda} a\ b.\ b\ \$\ a)\ \$\ x)\ n && \mathsf{norm_{pn}}\ f\ (\mathsf{Pure}\ x) = \mathsf{Pure}\ (f\ \$\ x)\\
&\mathsf{norm_{nn}}\ n\ (n'\ \bar{\diamond}\ x) = \mathsf{norm_{nn}}\ (\mathsf{norm_{pn}}\ \bar{\mathbf{B}}\ n)\ n'\ \bar{\diamond}\ x && \mathsf{norm_{pn}}\ f\ (n\ \bar{\diamond}\ x) = \mathsf{norm_{pn}}\ (\bar{\mathbf{B}}\ \$\ f)\ n\ \bar{\diamond}\ x
\end{aligned}$$

Fig. 3. Specification of the normalisation function $t\!\downarrow$.

separately. (We assume that the free variables in an equation are implicitly quantified universally, i.e., in the interpretation as functions, an equation denotes an equality of two functions.) Associativity $(x + y) + z = x + (y + z)$, e.g., gets lifted to $\mathsf{pure}\ (+) \diamond (\mathsf{pure}\ (+) \diamond x \diamond y) \diamond z = \mathsf{pure}\ (+) \diamond x \diamond (\mathsf{pure}\ (+) \diamond y \diamond z)$. Conversely, unlifting removes the functor from an idiomatic expression or equation by dropping pures and $(\diamond)$ and replacing opaque terms with fresh variables. An equation is liftable in F iff the equation implies itself lifted to F. When we consider a term or equation and its lifted counterpart, we say that the former is at base level (relative to this lifting).

Hinze [15] characterised equations that are liftable in any idiom and showed that the proof of the lifting step follows a simple structure if both sides are in canonical form. In this section, we adapt his findings to our setting, formalise the lifting lemma in our deep model, and discuss its implementation in the package.

3.1 Conversion to Canonical Form

The first step of lifting converts an idiomatic expression into canonical form. Recall from §1.1 that an idiomatic term is in canonical form iff it consists of a single pure _ applied to the effectful (opaque) terms, i.e., $\mathsf{pure}\ f \diamond x_1 \diamond \ldots \diamond x_n$. We formalise canonicity as the inductive set CF defined by (i) $\mathsf{Pure}\ x \in \mathsf{CF}$, and (ii) $t\ \bar{\diamond}\ \mathsf{Opq}\ x \in \mathsf{CF}$ if $t \in \mathsf{CF}$. Borrowing from Hinze's terminology, we say that n is a normal form of an idiomatic term t iff n is in canonical form and equivalent to t, i.e., $n \in \mathsf{CF}$ and $t \simeq n$. If $t \in \mathsf{CF}$, we refer to the $\mathsf{Pure}\ x$ part as the single pure subterm of t.

Hinze [15, Lemma 1] gives an algorithm to compute a normal form in the monoidal representation of idioms, which is essentially an uncurried variant of the applicative representation from §1.1. Since HOL functions are typically curried, we want to retain the applicative style in lifted expressions (to which normalisation is applied). Therefore, we stick to curried functions and adapt the normalisation function accordingly. In the following, we first formalise the normalisation function $\downarrow$ in the deep model and then explain how a proof-producing function for the corresponding equation can be extracted. The latter step is a recurring theme in our implementation.

Figure 3 shows the specification for $\downarrow$. The cases of pure and opaque terms are easy. For applications, $\downarrow$ first normalises both arguments and combines the results using the auxiliary functions $\mathsf{norm_{nn}}$ and $\mathsf{norm_{pn}}$. The auxiliary function $\mathsf{norm_{pn}}$ handles the simplest case of applying a pure function f to a term in canonical form. By repeated application of the composition law, $\mathsf{norm_{pn}}$ splits the variables off until only two pure terms remain which can be combined by the homomorphism law. The other function $\mathsf{norm_{nn}}$ assumes that both arguments are already in canonical form. The base case $n' = \mathsf{Pure}\ x$ reduces to the domain of $\mathsf{norm_{pn}}$ via the interchange law. In case of an application, $\mathsf{norm_{pn}}$ incorporates the added term $\bar{\mathbf{B}}$ into n before $\mathsf{norm_{nn}}$ recurses. Note that the equations for $\mathsf{norm_{nn}}$ and $\mathsf{norm_{pn}}$ are complete for terms in canonical form.

Lemma 2 (Correctness of $\downarrow$). *Let $t :: \alpha$ iterm, $f ::$ term and $n, n' \in$ CF. Then,*
(a) pure $f \diamond n \simeq \mathsf{norm}_{pn}\ f\ n$ and $\mathsf{norm}_{pn}\ f\ n \in$ CF;
(b) $n \diamond n' \simeq \mathsf{norm}_{nn}\ n\ n'$ and $\mathsf{norm}_{nn}\ n\ n' \in$ CF;
(c) $t \simeq t\downarrow$ and $t\downarrow \in$ CF.

Proof. We prove each of (a)–(c) by structural induction. As a representative example, we focus on the three cases for (c): (i) The case $t = \mathsf{Pure}\ \_$ is trivial. (ii) For $t = \mathsf{Opq}\ x$, we justify $\mathsf{Opq}\ x \simeq \mathsf{Pure}\ \bar{\mathsf{I}} \mathbin{\bar\diamond} \mathsf{Opq}\ x$ by the identity law (Fig. 2) and symmetry. (iii) For $(t \mathbin{\bar\diamond} t')\downarrow$, the induction hypotheses are $t \simeq t\downarrow$ and $t\downarrow \in \mathsf{CF}$, and analogously for t'. Thus, $t \mathbin{\bar\diamond} t' \simeq t\downarrow \mathbin{\bar\diamond} t'\downarrow \simeq \mathsf{norm}_{nn}\ (t\downarrow)\ (t'\downarrow) = (t \mathbin{\bar\diamond} t')\downarrow$ by (b).

In the shallow embedding, the proof method *applicative-nf* not only computes a normal form t' for an idiomatic term t. It also must prove them being equivalent, i.e., $t = t'$. Such a function from terms to equational theorems is known as a conversion. Paulson [27] designed a library of combinators for composing conversions, e.g., by transitivity. This way, each of (a)–(c) in Lemma 2 becomes one conversion which establishes the part about $\simeq$. (We ignore the part about $\_ \in \mathsf{CF}$, as it is computationally irrelevant.) The inductive proofs yield the implementation of the conversions: the induction hypotheses are obtained by recursively calling the conversion on the subterms; case distinction is implemented by matching; and the concrete applicative laws are known to the package and instantiated directly. Thus, each proof step involving $\simeq$ indicates which conversion combinator has to be used.

3.2 Lifting

Hinze's condition for equations that can be lifted in all idioms is as follows: The list of variables, when reading from left to right, must be the same on both sides, and no variable may appear twice on either side. Then, the normal forms of the two lifted terms differ only in the the pure functions, which are just the base-level terms abstracted over all variables. The base equation implies that these functions are extensionally equal.

It is not entirely obvious that the normal form has this precise relationship with lifting, so we prove it formally. This gives us confidence that our proof procedure always succeeds if the conditions on the variables are met.

For practical reasons, our proof method performs unlifting rather than lifting. It takes as input an equality between idiomatic expressions, and reduces it to the weakest base-level equation that entails it—independent of the applicative functor. Unlifting has two advantages. First, the user can apply the method to instantiations of lifted equations, where the variables are replaced with concrete effects such as fresh. Thus, there is no need to manually generalise the lifted equation itself. Second, in the lifted equation, the pure terms distinguish constants (to be lifted) from opaque terms, but there are no such markers on the base level. Rather than lifting, we therefore formalise unlifting, which replaces each opaque term by a new bound variable ($|x|$ denotes the length of the list x).

$\mathsf{unlift}\ t = (\mathsf{let}\ n = |\mathsf{opq}\ t|\ \mathsf{in}\ \mathsf{Abs}^n\ (\mathsf{unlift'}\ n\ 0\ t))$

$\mathsf{unlift'}\ n\ i\ (\mathsf{Pure}\ x) = \mathsf{shift}\ x\ n$
$\mathsf{unlift'}\ n\ i\ (\mathsf{Opq}\ x) = \mathsf{Var}\ i$
$\mathsf{unlift'}\ n\ i\ (t \mathbin{\bar\diamond} t') = \mathsf{unlift}\ n\ (i + |\mathsf{opq}\ t'|)\ t \mathbin{\$} \mathsf{unlift}\ n\ i\ t'$

Here, the function $\mathsf{opq}\ t$ returns the list of all opaque terms from left to right, so $|\mathsf{opq}\ t|$ counts the occurrences of Opq in t. Nipkow's function $\mathsf{shift}\ x\ n$ increments all loose variables in x by n. For example, $\mathsf{unlift}\ (\mathsf{Opq}\ a \mathbin{\overline{\diamond}} (\mathsf{Pure}\ f \mathbin{\overline{\diamond}} \mathsf{Opq}\ b)) = \overline{\lambda} g\ x.\ g\ (f\ x)$, as expected. Note that this holds independent of a and b.

The benefit of the meta model is that we can characterise the normal form.

Lemma 3. *Let* $\mathsf{Pure}\ f$ *be the single pure subterm in* $t{\downarrow}$. *Then* $f \simeq_{\beta\eta} \mathsf{unlift}\ t$.

Equality in a real theory generally has more axioms than those for term reductions. Let $\simeq'_{\beta\eta}$ be an extension of $\simeq_{\beta\eta}$, and $\simeq'$ the corresponding extension of $\simeq$. Then, we obtain the following lifting rule, which follows from Lemmas 2 and 3 and $\mathsf{opq}\ (t{\downarrow}) = \mathsf{opq}\ t$.

Lemma 4. *Let* $\mathsf{opq}\ s = \mathsf{opq}\ t$. *Then,* $\mathsf{unlift}\ s \simeq'_{\beta\eta} \mathsf{unlift}\ t$ *implies* $s \simeq' t$.

To implement its proof, we rewrite both idiomatic terms of the input equation with the normal form conversion, i.e., we are left with the subgoal of the form $\mathsf{pure}\ f \diamond x_1 \ldots \diamond x_n = \mathsf{pure}\ g \diamond x_1 \ldots \diamond x_n$. It suffices to prove $f = g$, which follows from the base-level equation $\forall x_1 \ldots x_n.\ f\ x_1 \ldots x_n = g\ x_1 \ldots x_n$ by extensionality.

Moreover, we get that the normal form is indeed unique.

Lemma 5. *If* $s, t \in \mathsf{CF}$ *and* $s \simeq' t$, *then* s *and* t *have the same structure, and the pure terms are related by* $\simeq'_{\beta\eta}$.

4 Combinators

Lifting works for equations whose both sides contain the same list of variables without repetitions. Many equations, however, violate this conditions. Therefore, Hinze studied the class of idioms in which all equations can be lifted [15]. He proved that every equation can be lifted if the functor satisfies the two equations

$$\mathsf{pure}\ \mathbf{S} \diamond f \diamond g \diamond x = f \diamond x \diamond (g \diamond x) \qquad\qquad \mathsf{pure}\ \mathbf{K} \diamond x \diamond y = x \tag{5}$$

for all f, g, x, and y, where $\mathbf{S} = (\lambda f\ g\ x.\ f\ x\ (g\ x))$ and $\mathbf{K} = (\lambda x\ y.\ x)$ denote the well-known combinators from combinatory logic. Similar to bracket abstraction for the λ-calculus, Hinze defines an abstraction algorithm $[x]t$ which removes an opaque term x from an idiomatic expression t such that $[x]t \mathbin{\overline{\diamond}} x \simeq_E t$, where $\simeq_E$ extends $\simeq$ with the combinators' laws. For lifting, Hinze uses the abstraction algorithm to remove all variables from both sides of the equation in the same order (which may introduce $\mathbf{S}$ and $\mathbf{K}$ in the pure part), then applies the lifting technique and finally removes the combinators again.

However, only few applicative functors satisfy (5). In this section, we subject Hinze's idea to a finer analysis of equational lifting for various sets of combinators, present the implementation as a proof method in Isabelle and an application to the Stern-Brocot tree, and sketch the formalisation in the deep embedding.[2] The new proof method *applicative-lifting* subsumes the one from §3.

[2] Hinze briefly considers functors with the combinators $\mathbf{S}$ and $\mathbf{C}$ and notes that the case with only the combinator $\mathbf{C}$ might be interesting, too, but omits the details.

applicative functor	**B**	**I**	**C**	**K**	**W**	**S**	abstraction algorithm
environment, stream, non-standard numbers	√	√	√	√	√	√	(kibtcs)
option, zip list	√	√	√		√	√	(ibtcs)
probability, non-empty set	√	√	√	√			(kibtc)
subprobability, set, commutative monoid	√	√	√				(ibtc)
either, idempotent monoid	√	√			√		(ibt w)
distinct non-empty list	√	√		√			(kibt)
state, list, parser, monoid	√	√					(ibt)

Table 1. List of applicative functors, their combinators and the abstraction algorithm.

4.1 The Combinatorial Basis BCKW

While **SK** has become the canonical approach to combinatory logic, we argue that Curry's set of combinators **BCKW** works better for applicative lifting, where **B** $= (\lambda f\ g\ x.\ f\ (g\ x))$ and **C** $= (\lambda f\ x\ y. f\ y\ x)$ and **W** $= (\lambda f\ x.\ f\ x\ x)$. We say that a functor has a combinator if the equation defining the combinator is liftable. For **BICKW** (where **I** $= (\lambda x.\ x)$), the lifted equations are the following.

$$\text{pure } \mathbf{B} \diamond f \diamond g \diamond x = f \diamond (g \diamond x) \qquad\qquad \text{pure } \mathbf{C} \diamond f \diamond x \diamond y = f \diamond y \diamond x$$

$$\text{pure } \mathbf{K} \diamond x \diamond y = x \qquad\qquad \text{pure } \mathbf{W} \diamond f \diamond x = f \diamond x \diamond x \qquad\qquad \text{pure } \mathbf{I} \diamond x = x$$

Note that every applicative functor has combinators **B** and **I** as their equations are exactly the composition and identity law, respectively.

We focus on **BCKW** for two reasons. First, the combinators can be intuitively interpreted. A functor has **C** if effects can be swapped, it has **K** if effects may be omitted, and it has **W** if effects may be doubled. In contrast, Hinze's combinator **S** mixes doubling with a restricted form of swapping; full commutativity additionally requires **K**. Second, our set of combinators yields a finer hierarchy of applicative functors. Thus, the proof method is more widely applicable because it can exploit more precisely the properties of the particular functor, although its implementation remains generic in the functor.

Table 1 lists for a number of applicative functors the combinators they possess. For reference, the functors are defined in App. A. The table is complete in the sense that there is a tick √ iff the functor has this combinator.

Most of the functors are standard, but some are worth mentioning. Hinze [15] proved that all functors which are isomorphic to the environment functor (a.k.a. the reader idiom, e.g., streams and infinite binary trees) have combinators **S** and **K**. Thus, they also have the combinators **C** and **W**, as the two can be expressed in terms of **S** and **K**. However, some functors with combinators **S** and **K** are not isomorphic to the environment functor. One example is Huffman's construction of non-standard numbers in non-standard analysis [17].

Every monoid yields an applicative functor known as the writer idiom. Given a monoid on β with binary operation + and neutral element 0, we turn the functor (β,α) monoid $= \beta \times \alpha$ into an applicative one via

$$\text{pure}_{\text{monoid}}\ x = (0, x) \qquad\qquad (a, f) \diamond_{\text{monoid}} (b, x) = (a + b, f\ x)$$

Commutative monoids have the combinator **C**, idempotent ones have **W**.

The idioms "probability" and "non-empty set" are derived from the monads for probabilities and non-determinism without failure. When the latter is implemented by distinct non-empty lists, commutativity is lost because lists respect the order of elements.

The attentive reader might have noticed that one combination of combinators is missing, namely **BIKW**, i.e., only **C** is excluded. As **BCKW** is a minimal basis for combinatory logic, **C** cannot be expressed in terms of **BIKW**. Surprisingly, an applicative functor always has **C** whenever it has **BIKW**, as the following calculation shows, where $\mathsf{Pair}\ x\ y = (x, y)$ and $\pi_1\ (x, y) = x$ and $\pi_2\ (x, y) = y$ and G abbreviates $\lambda f\ p\ q.\ \mathbf{C}\ f\ (\pi_2\ p)\ (\pi_1\ q))$. Steps (i) and (iii) are justified by the equations for **K** and **I** and **W**; steps (ii) and (iv) hold by lifting of the identities $\mathbf{K}\ (\mathbf{C}\ f\ (\mathbf{K}\ \mathbf{I}\ z\ x)\ y)\ w = G\ f\ (z,x)\ (y,w)$ and $\mathbf{W}\ (G\ f)\ (y,x) = f\ y\ x$, respectively.

$$\begin{aligned}
\mathsf{pure}\ \mathbf{C} \diamond f \diamond x \diamond y &\overset{\text{(i)}}{=} \mathsf{pure}\ \mathbf{K} \diamond (\mathsf{pure}\ \mathbf{C} \diamond f \diamond (\mathsf{pure}\ \mathbf{K} \diamond \mathsf{pure}\ \mathbf{I} \diamond y \diamond x) \diamond y) \diamond x \\
&\overset{\text{(ii)}}{=} \mathsf{pure}\ G \diamond f \diamond (\mathsf{pure}\ \mathsf{Pair} \diamond y \diamond x) \diamond (\mathsf{pure}\ \mathsf{Pair} \diamond y \diamond x) \\
&\overset{\text{(iii)}}{=} \mathsf{pure}\ \mathbf{W} \diamond (\mathsf{pure}\ G \diamond f) \diamond (\mathsf{pure}\ \mathsf{Pair} \diamond y \diamond x) \\
&\overset{\text{(iv)}}{=} f \diamond y \diamond x
\end{aligned}$$

The crucial difference between combinatory logic and idioms can be seen by looking at G, which is equivalent to $\mathbf{B}\ (\mathbf{B}\ (\mathbf{T}\ \pi_1))\ (\mathbf{B}\ (\mathbf{B}\ \mathbf{B})\ (\mathbf{B}\ (\mathbf{T}\ \pi_2)\ (\mathbf{B}\ \mathbf{B}\ \mathbf{C})))$ where $\mathbf{T} = (\lambda x\ f.\ f\ x)$. By the interchange and homomorphism laws, we have $\mathsf{pure}\ (\mathbf{T}\ x) \diamond f = f \diamond \mathsf{pure}\ x$ in every idiom. This is the very bit of reordering that **C** adds to **BKW**. Note, however, that **T** is different from the other combinators: it may only occur applied to a term without Opq (as such terms are lifted to pure terms by the homomorphism law). In fact, if **T** was like the others, every applicative functor would have **C** thanks to $\mathbf{C} = \mathbf{B}\ (\mathbf{T}\ (\mathbf{B}\ \mathbf{B}\ \mathbf{T}))\ (\mathbf{B}\ \mathbf{B}\ \mathbf{T})$ [6].

4.2 Characterisation of Liftable Equations

The lifting technique from §3.2 requires that the list of opaque terms be the same on both sides and free of duplicates. With additional combinators, we can try to rewrite both sides such that the lists satisfy this condition. In this section, we derive for each set of combinators a simple criterion whether this can be achieved. Simplicity is important because users should be able to easily judge whether an equation can be lifted to a particular functor using our proof method. Our analysis heavily builds on the literature on representable λ-terms in various combinator bases [5]. Therefore, we refer to opaque terms as variables in the rest of this section.

By using normal forms (cf. Lemma 2), it suffices to consider only the list of variables on each side of the equation, say v_l and v_r. Our goal is to find a duplicate-free variable list v^* such that v_l and v_r can both be transformed into v^*. The permitted transformations are determined by the combinators:

- If **C** is available, we may reorder any two variables.
- If **K** is available, we may insert a variable anywhere.
- If **W** is available, we may duplicate any contiguous subsequence or drop a repetition of a contiguous subsequence (the repetition must be adjacent).

This yields the following characterisation of liftable equations. (The conditions for all the cases which include the combinator **C** are equal to the representation conditions for λ-terms with the given combinators [5].)

BI No transformation is possible. So we require $v^* = v_l = v_r$.

BIC v_l and v_r must be duplicate-free and permutations of each other. We choose for v^* any permutation of v_l.

BICK v_l and v_r must be duplicate-free. We choose for v^* any duplicate-free list of the union of the variables in v_l and v_r.

BICW v_l and v_r must contain the same variables, but need not be duplicate-free. We choose for v^* any duplicate-free list of the variables.

BICKW No constraints on v_l or v_r. We choose for v^* any duplicate-free list of the union of variables in v_l and v_r. (This is the case considered by Hinze [15].)

BIK v_l and v_r must be duplicate-free and the shared variables must occur in the same order. Take for v^* any merge of v_l and v_r, i.e., a duplicate-free sequence which contains v_l and v_r as subsequences.

BIW In this case, we work in the free idempotent monoid (FIM) whose letters are the variables in v_l and v_r. So, our task boils down to finding a duplicate-free word v^* such that $v_l \sim v^* \sim v_r$ where $\sim$ denotes equivalence in the FIM.
Green and Rees [13] characterised $\sim$ recursively: For a word x, let $\overrightarrow{x}$ and $\overleftarrow{x}$ denote the longest prefix or suffix of x that contains all but one letters of x. Then, $x \sim y$ iff x and y contain the same letters and $\overrightarrow{x} \sim \overrightarrow{y}$ and $\overleftarrow{x} \sim \overleftarrow{y}$.
This criterion yields the following conditions: (i) v_l and v_r contain the same variables; (ii) the orders in which the variables occur for the first or for the last time must be all the same in v_l and v_r (we choose v^* as the list of variables in this order); and (iii) recursively the same conditions hold for $\overrightarrow{v_l}$ and $\overrightarrow{v_r}$, and for $\overleftarrow{v_l}$ and $\overleftarrow{v_r}$. For example, the equation $\forall a\ b\ c.\ \mathsf{f}\ a\ b\ c = \mathsf{g}\ a\ b\ c\ a\ c\ b\ a\ b\ c$ satisfies this condition with $v^* = abc$.[3]

4.3 Implementation via Bracket Abstraction

Bracket abstraction converts a λ-calculus term into combinator form. The basic algorithm $[x]t$ abstracts the variable x from the term t (which must not contain any abstraction). Like λ-terms, applicative terms are built from constants (Pure _), variables (Opq _) and applications. So, bracket abstraction also makes sense for applicative terms. What is interesting about bracket abstraction is that the algorithm is modular in the combinators. That is, bracket abstraction allows us to deal with all the different combinator bases in a uniform way. In detail, we first abstract the variables on both sides of the equation in the order given by $v^* = v_1 \ldots v_n$. As $l \simeq_E ([v_1](\ldots([v_n]l))) \mathbin{\bar{\diamond}} v_1 \mathbin{\bar{\diamond}} \ldots \mathbin{\bar{\diamond}} v_n$ and $r \simeq_E ([v_1](\ldots([v_n]r))) \mathbin{\bar{\diamond}} v_1 \mathbin{\bar{\diamond}} \ldots \mathbin{\bar{\diamond}} v_n$ by the correctness of bracket abstraction, we thus obtain an equation whose two sides are in normal form. From there, our implementation proceeds as before (§3).

As usual, we specify a bracket abstraction algorithm by a list of rules, say (kibtcs). This means that the corresponding rules should be tried in that order and the first one

[3] The following shows the equivalence (bold face denotes doubling and underlining dropping of a repetition): $abcacb\mathbf{abc} \sim abcacbabc\mathbf{abc} \sim abcacbabcab\mathbf{ca}bc \sim abcacbabca\mathbf{bcac}abc \sim abcacbabcabc\mathbf{acb}cacabc \sim abcacbabcabcac\mathbf{ba}cbcacabc \sim abcacbabcabca\mathbf{cbab}acbcacabc \sim \underline{abcacbabcabcacbabc}babacbcacabc \sim abca\underline{cbabcbab}acbcacabc \sim abcac\underline{baba}cbcacabc \sim abc\underline{acbacb}cacabc \sim a\underline{bcacbcac}abc \sim ab\underline{caca}bc \sim \underline{abcabc} \sim abc$.

$[x]t$	$= \mathsf{Pure}\ \overline{\mathbf{K}} \mathbin{\bar\diamond} t$	if $x \notin \mathcal{V}(t)$	(k)
$[x]x$	$= \mathsf{Pure}\ \overline{\mathbf{I}}$		(i)
$[x](s \mathbin{\bar\diamond} t)$	$= \mathsf{Pure}\ \overline{\mathbf{B}} \mathbin{\bar\diamond} s \mathbin{\bar\diamond} [x]t$	if $x \notin \mathcal{V}(s)$	(b)
$[x](s \mathbin{\bar\diamond} t)$	$= \mathsf{Pure}\ \overline{\mathbf{T}} \mathbin{\bar\diamond} t \mathbin{\bar\diamond} [x]s$	if $\mathcal{V}(t) = \emptyset$	(t)
$[x](s \mathbin{\bar\diamond} t)$	$= \mathsf{Pure}\ \overline{\mathbf{C}} \mathbin{\bar\diamond} [x]s \mathbin{\bar\diamond} t$	if $x \notin \mathcal{V}(t)$	(c)
$[x](s \mathbin{\bar\diamond} t)$	$= \mathsf{Pure}\ \overline{\mathbf{S}} \mathbin{\bar\diamond} [x]s \mathbin{\bar\diamond} [x]t$		(s)
$[x](s \mathbin{\bar\diamond} t)$	$= \mathsf{Pure}\ \overline{\mathbf{W}} \mathbin{\bar\diamond} (\mathsf{Pure}\ \overline{\mathbf{B}} \mathbin{\bar\diamond} (\mathsf{Pure}\ \overline{\mathbf{T}} \mathbin{\bar\diamond} [x]t) \mathbin{\bar\diamond} (\mathsf{Pure}\ (\overline{\mathbf{B}}\ \overline{\mathbf{B}}) \mathbin{\bar\diamond} [x]s))$	if $\mathcal{V}([x]t) = \emptyset$	(w)

Table 2. Bracket abstraction rules for applicative expressions.

matching should be taken. The algorithm for each set of combinators is listed in the last column of Table 1. The rules are shown in Table 2, where $\mathcal{V}(t)$ denotes the set of variables in t. All but (t) and (w) correspond to the standard abstraction rules for the λ-calculus [5]. The side condition of (t) reflects the restriction of the interchange law to pure computations (cf. §4.1). Rule (w) is used only if **C** is not available—otherwise (s) is used as **S** = **B** (**B W**) (**B B C**). It uses **T** to allow for pure computations between two occurrences of the same variable. This way, we avoid repeatedly converting the term to normal form, as otherwise **W** could only be used for terms of the form $t \mathbin{\bar\diamond} x \mathbin{\bar\diamond} x$ for some variable x.

Our bracket abstraction algorithm (ibtw) is not complete for **BIW**. A dedicated algorithm would be needed, as it seems not possible to construct equivalence proofs like in Footnote 3 using bracket abstraction, because the transformations are not local. Bersten's and Reutenauer's elementary proof [2, Thm. 2.4.1] of Green's and Rees' characterisation contains an algorithm, but we settle with (ibtw) nevertheless. Thus, our implementation imposes stronger conditions on v_l and v_r than those described in §4.2, namely v_l and v_r must use the same variables in the same order, but each variable may be repeated any number of times (with no other variable between the repetitions). In practice, we have not yet encountered a liftable sequence of variables that needs the full generality.

Again, we verify unlifting in the deep embedding. We show that the implementation with bracket abstraction yields the same equation (after reducing the combinators) as unlifting the lifted equation directly, where v^* determines the quantifier order. Thus, it suffices to rearrange the quantifiers according to v^*.

Unlike to §3.2, unlift must map identical opaque terms to the same variable. So, we assume that Opq's argument denotes the variable name. Then, the new function unlift* replaces Opq i with Var i, $(\bar\diamond)$ with \$ and Pure x with shift x $|v^*|$.

Further, we abstract from the concrete bracket abstraction algorithm. We model the algorithm as two partial functions $\overline{[\_]}$ and $[\_]$ on term and nat iterm and assume that they are correct ($\lfloor\_\rfloor$ denotes definedness): (i) if $\overline{[i]}t = \lfloor t' \rfloor$, then t' \$ Var $i \simeq_{\beta\eta} t$ and i is not free in t', (ii) if $[i]t = \lfloor t' \rfloor$, then $t' \diamond \mathsf{Opq}\ i \simeq_E t$ and set (opq t') = set (opq t) − { Opq i }, and (iii) they commute with unlifting: $\overline{[i]}$ (unlift* n t) = unlift* n ($[i]t$) for $i < n$. Formalising and verifying bracket abstraction is left as future work. In the theorem below, the congruence relation $\simeq'_E$ combines $\simeq_E$ with the additional axioms from $\simeq'$.

Theorem 1. *Let s, t :: nat iterm and let v^* be a permutation of $\{0, \ldots, n-1\}$. Assume that set (opq s) $\cup$ set (opq t) $\subseteq$ set v^* and that $[\_]$ succeeds to abstract s and t over v^*. Then, Abs^n (unlift* n s) $\simeq'_{\beta\eta}$ Abs^n (unlift* n t) implies $s \simeq'_E t$.*

4.4 Application: The Stern-Brocot Tree

Hinze uses his theory of lifting to reason about infinite trees of rational numbers [14]. In particular, he shows that a linearisation of the Stern-Brocot tree yields Dijsktra's fusc function [7]. We have formalised his reasoning in Isabelle as a benchmark for our package [9]. Here, we report on our findings.

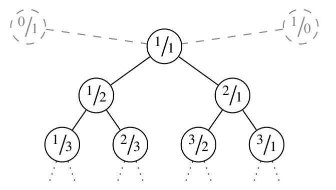

Fig. 4. The Stern-Brocot tree

The Stern-Brocot tree stern-brocot enumerates all the rationals in their lowest terms (see [12]). It is an infinite binary tree of type frac cotree containing formal fractions (**type-synonym** frac = nat × nat). Each node is labelled with the mediant of its right-most and left-most ancestor (Fig. 4), where mediant (a, c) (b, d) = $(a + b, c + d)$. Formally, stern-brocot = sb-gen $(0, 1)$ $(1, 0)$ with

codatatype α cotree = Node (root: α) (α cotree) (α cotree)
primcorec sb-gen l u = (let m = mediant l u in Node m (sb-gen l m) (sb-gen m u))

The type constructor cotree forms an idiom analogous to stream, i.e., $(\diamond)$ corresponds to zipping trees with function application. Combinators **C**, **K**, and **W** exist. The idiom homomorphism stream :: α cotree $\Rightarrow$ α stream linearises a tree to a stream.

primcorec chop (Node x l r) = Node (root l) r (chop l)
primcorec stream t = root t $\prec$ stream (chop t)

Hinze shows that stream stern-brocot equals fusc $\circledast$ fusc′ for Dijkstra's fusc and fusc′ given by

$$\mathsf{fusc} = 1 \prec \mathsf{fusc}' \qquad\qquad \mathsf{fusc}' = 1 \prec (\mathsf{fusc} + \mathsf{fusc}' - 2 * (\mathsf{fusc}\ \mathsf{mod}\ \mathsf{fusc}'))$$

where all arithmetic operations are lifted to streams, e.g., $s + t$ denotes pure $(+) \diamond s \diamond t$, and $(\circledast)$ :: α stream $\Rightarrow$ β stream $\Rightarrow$ $(\alpha \times \beta)$ stream zips two streams. The proof shows that stream stern-brocot satisfies the same recursion equation as fusc $\circledast$ fusc′, so they must be equal. The crucial step is to show that chop den = num + den − 2 ∗ (num mod den) where num = pure π_1 $\diamond$ stern-brocot and den = pure π_2 $\diamond$ stern-brocot project the Stern-Brocot tree to numerators and denominators. Hinze proves this equality by lifting various arithmetic identities from integers to trees.

We instantiate Isabelle/HOL's fine-grained arithmetic type class hierarchy for cotree and stream up to the class for rings with characteristic 0. This way, we can use the algebraic operators and reason directly on trees and streams. Almost all algebraic laws are proven by our lifting package from the base equation. The only exception are the two cancellative laws in semigroups, namely $a = b$ whenever $a + c = b + c$ or $c + a = c + b$. Such conditional equations are not handled by our lifting machinery. So, we prove these two laws conventionally by coinduction.

Moreover, we discovered that Hinze's lifting framework cannot prove the identity for chop den, contrary to his claims. In detail, the proof relies on the identity x mod $(x + y)$ = x on natural numbers, but this holds only for $y > 0$. Hinze does not explain how to lift and handle such preconditions. As the combinators **K** and **W** exist, we express the lifted precondition as $\mathsf{pure}\ (>) \diamond y \diamond 0 = \mathsf{pure\ True}$ and split the proof into the three steps shown below: (i) and (iii) hold by lifting and (ii) by assumption.

$$\begin{aligned} x \ \mathsf{mod}\ (x+y) &\overset{(i)}{=} \mathsf{pure}\ (\lambda b\ x.\ \mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ 0) \diamond (\mathsf{pure}\ (>) \diamond y \diamond 0) \diamond x \overset{(ii)}{=} \\ &\quad \mathsf{pure}\ (\lambda b\ x.\ \mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ 0) \diamond \mathsf{pure\ True} \qquad\qquad \diamond x \overset{(iii)}{=} x \end{aligned}$$

Overall, we found that the lifting package works well for algebraic reasoning and that we should extend lifting to handle arbitrary relations and preconditions.

5 Related Work

Most closely related to our work is Hinze's [15] on lifting. He focuses on the two extremes in the spectrum: the class of equations liftable in all idioms, and the idioms in which all equations are liftable. Our implementation for the former merely adapts his ideas to the HOL setting. For the latter, Hinze requires idioms to be strongly extensional in addition to them having **S** and **K**. This ensures that the idiom can emulate λ-abstraction, so lifting is defined for all λ-terms. Therefore, his proof of the Lifting Lemma does not carry over to weaker sets of combinators. As we focus on unlifting, we do not need such emulations and instead use bracket abstraction, which is hidden in Hinze's emulation of abstraction, uniformly for all sets of combinators. Hinze also models idiomatic expressions syntactically using GADTs, which ensures type correctness. He defines equivalence on idiomatic terms semantically. As the interpretation cannot be expressed in HOL, we use the syntactic relation $\simeq$ instead. This has the advantage that we can prove uniqueness of normal forms (Lemma 5) by induction.

Several other kinds of lifting are available as Isabelle/HOL packages. Huffman's transfer tactic [17] lifts properties to non-standard analysis (NSA) types like the hyperreals, which are formalised by the idiom star. The tactic can lift arbitrary first-order properties by exploiting the properties of star. To that end, the tactic first unlifts the property similar to our operation unlift and then proves equivalence by resolving with rules for logical and star operators. Our package subsumes Huffman's for equations, but it cannot lift first-order connectives yet.

The package Lifting [18] creates quotient types via partial equivalence relations. The companion package Transfer, which is different from aforementioned transfer tactic, exploits parametricity and representation independence to prove equivalences or implications between properties on the raw type and the quotient. Like for NSA, resolution guides the equivalence proof. Lifting and Transfer cannot handle lifting to applicative functors, as the functor's values are usually more complex than the base values, instead of more abstract. In comparison, our lifting step is much simpler, as it just considers pairs of extensionally equal functions; the whole automation is needed to extract these functions from idiomatic expressions. The other packages preserve the term structure and relate each component of the term as determined by the rules.

6 Conclusion and Future Work

This paper presents a first step towards a infrastructure for reasoning about effectful programs. Like applicative functors help in delimiting pure and effectful parts of the computation, our proof method supports separating the effectful and the pure aspects of the reasoning. The results from our case studies indicate that applicative functors are a suitable abstraction for reasoning. They seem to be better suited than monads, as applicative expressions can be analysed statically. Thus, one should prefer applicative functors over monads whenever possible.

There is much to be done before proof assistants support reasoning about effects smoothly. As a next step, we will investigate how to extend the scope of lifting. Going from equations to arbitrary relations should be easy: if the functor has a relator for which pure and $(\diamond)$ are relationally parametric [28], then the lifting technique should work unchanged. The extension to preconditions and other first-order connectives seems to be harder. In any ring with $0 \neq 1$, e.g., $x = x + 1 \longrightarrow x = 2x$ holds, but it does not when interpreted in the set idiom over the same ring. We expect that combinators will help there, too. Moreover, we would like to study whether one should further refine the set of combinators. For example, the idiom "either" derived from the exception monad has the stronger combinator **H** with pure **H** $\diamond f \diamond x \diamond y = f \diamond x \diamond y \diamond x$, which cannot be expressed by **BIW**. Experience will tell when specialisation is needed and when it goes too far.

The combinator laws can also be interpreted monadically. For example, **C** exists in commutative monads and **K** demands that $x \gg\!= (\lambda\_.y) = y$. Therefore, we experimented with lifting for monads, too. As $(\diamond)$ and $(\gg\!=)$ are related (cf. §1.1), one can express certain parts of a monadic term applicatively using $(\diamond)$ and apply the lifting approach to those parts. In particular, the monadic laws for **C** and **K** can only be utilised if the affected part can be expressed applicatively. In a first attempt, we applied this idea to a security proof of the Elgamal encryption scheme [22], which uses the subprobability monad (which only has **C**). Our package successfully automates the arguments about commutativity in this proof, which previously were conducted by manual applications of the commutativity law. At present, we have to manually identify the right parts and rewrite them into applicative form. One reason is that monadic expressions in general contain several overlapping applicative subparts and consecutive applications of commutativity may require different parts for each application. Overall, the new Isar proof is more declarative, but also longer due to the manual rewrite steps. It will be an interesting problem to automate the identification of suitable parts and to combine the appropriate rewrites with lifting.

Acknowledgements Peter Gammie triggered our interest in reasoning about applicative functors and helped with the Stern-Brocot tree. We thank Dmitriy Traytel, Joachim Breitner, and the anonymous reviewers for suggesting many textual improvements. The first author was supported by SNSF grant 153217 "Formalising Computational Soundness for Protocol Implementations".

References

1. Berghofer, S.: Proofs, Programs and Executable Specifications in Higher Order Logic. Ph.D. thesis, Institut für Informatik, Technische Universität München (2003)
2. Berstel, J., Reutenauer, C.: Square-free words and idempotent semigroups. In: Lothaire, M. (ed.) Combinatorics on Words, pp. 18–38. Cambridge University Press, second edn. (1997)
3. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Abadi, M., Ito, T. (eds.) TACS 1997. LNCS, vol. 1281, pp. 515–529. Springer (1997)
4. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer (2008)

5. Bunder, M.W.: Lambda terms definable as combinators. Theoretical Computer Science 169(1), 3–21 (1996)
6. Church, A.: The calculi of lambda-conversion. Princeton University Press (1941)
7. Dijkstra,E.W.: An exercise for Dr.R.M.Burstall. In: Selected writings on computing: a personal perspective, pp. 215–216. Texts Monogr Comput Sci, Springer (1982)
8. Eberl, M., Hölzl, J., Nipkow, T.: A verified compiler for probability density functions. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 80–104. Springer (2015)
9. Gammie, P., Lochbihler, A.: The Stern-Brocot tree. Archive of Formal Proofs (2015), http://isa-afp.org/entries/Stern_Brocot.shtml, Formal proof development
10. Gibbons, J., Bird, R.: Be kind, rewind: A modest proposal about traversal (2012), http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/backwards.pdf
11. Gibbons, J., Hinze, R.: Just do it: Simple monadic equational reasoning. In: ICFP 2011. pp. 2–14. ACM (2011)
12. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics–a Foundation for Computer Science. Addison-Wesley, 2nd edn. (1994)
13. Green, J.A., Rees, D.: On semi-groups in which $x^r = x$. Mathematical Proceedings of the Cambridge Philosophical Society 48, 35–40 (1952)
14. Hinze, R.: The Bird tree. J. Func. Programm. 19(5), 491–508 (2009)
15. Hinze, R.: Lifting operators and laws (2010), http://www.cs.ox.ac.uk/ralf.hinze/Lifting.pdf
16. Homeier, P.V.: The HOL-Omega logic. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 244–259. Springer (2009)
17. Huffman, B.: Transfer principle proof tactic for nonstandard analysis. In: Kanovich, M., White, G.,Gottliebsen, H.,Oliva, P. (eds.) NetCA 2005. pp. 18–26. Queen Mary, University of London, Dept. of Computer Science, Research report RR-05-06 (2005)
18. Huffman, B., Kunčar, O.: Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In: CPP 2013. LNCS, vol. 8307, pp. 131–146. Springer (2013)
19. Hutton, G., Fulger, D.: Reasoning about effects: Seeing the wood through the trees. In: Trends in Functional Programming (TFP 2008) (2008)
20. Krebbers, R.: The C standard formalized in Coq. Ph.D. thesis, Radboud University (2015)
21. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer (2012)
22. Lochbihler, A.: Probabilistic functions and cryptographic oracles in higher order logic. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 503–531. Springer (2016)
23. Lochbihler, A., Schneider, J.: Applicative lifting. Archive of Formal Proofs (2015), http://isa-afp.org/entries/Applicative_Lifting.shtml
24. Marlow, S., Peyton Jones, S., Kmett, E., Mokhov, A.: Desugaring Haskell's do-notation into applicative operations (2016), http://research.microsoft.com/en-us/um/people/simonpj/papers/list-comp/applicativedo.pdf
25. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming 18(1), 1–13 (2008)
26. Nipkow, T.: More Church-Rosser proofs (in Isabelle/HOL). J. Automat. Reason. 26, 51–66 (2001)
27. Paulson, L.: A higher-order implementation of rewriting. Sci. Comput. Program 3(2), 119–149 (1983)
28. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP 1983. Information Processing, vol. 83, pp. 513–523. North-Holland/IFIP (1983)
29. Schropp, A., Popescu, A.: Nonfree datatypes in Isabelle/HOL. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 114–130. Springer (2013)
30. Tuong, F., Wolff, B.: A meta-model for the Isabelle API. Archive of Formal Proofs (2015), http://isa-afp.org/entries/Isabelle_Meta_Model.shtml

A Definitions of Applicative Functors

This appendix lists Isabelle/HOL definitions for the idioms mentioned in this paper. The definitional packages and their syntaxes are documented in the Isabelle/Isar reference manual. The proofs of the applicative laws and combinators are available online [23].

Environment (Reader)

type-synonym (α, β) env = $(\beta \Rightarrow \alpha)$
definition $\mathsf{pure}_{\mathsf{env}}\ x = (\lambda\_.\ x)$
definition $f \diamond_{\mathsf{env}} x = (\lambda y.\ f\ y\ (x\ y))$

Stream

codatatype α stream = $\alpha \prec \alpha$ stream
primcorec $\mathsf{pure}_{\mathsf{stream}}\ x = x \prec \mathsf{pure}_{\mathsf{stream}}\ x$
primcorec $(f \prec fs) \diamond_{\mathsf{stream}} (x \prec xs) = f\ x \prec (fs \diamond_{\mathsf{stream}} xs)$

Infinite binary tree

codatatype α cotree = Node α (α cotree) (α cotree)
primcorec $\mathsf{pure}_{\mathsf{cotree}}\ x = \mathsf{Node}\ x\ (\mathsf{pure}_{\mathsf{cotree}}\ x)\ (\mathsf{pure}_{\mathsf{cotree}}\ x)$
primcorec $(\mathsf{Node}\ f\ g\ h) \diamond_{\mathsf{cotree}} (\mathsf{Node}\ x\ y\ z) = \mathsf{Node}\ (f\ x)\ (g \diamond_{\mathsf{cotree}} y)\ (h \diamond_{\mathsf{cotree}} z)$

Non-standard numbers as used in non-standard analysis in Isabelle/HOL [17]. The type α star is the quotient of the environment idiom (α, nat) env over equality in some free ultrafilter $\mathcal{U}$ on nat.

quotient-type α star = (α, nat) env / $(\lambda X\ Y.\ (\lambda n.\ X n = Y n) \in \mathcal{U})$
lift-definition $\mathsf{pure}_{\mathsf{star}}$ is $\lambda x\ \_.\ x$
lift-definition $(\diamond)_{\mathsf{star}}$ is $\lambda f\ x\ y.\ f\ y\ (x\ y)$

Option

datatype α option = None | Some α
abbreviation $\mathsf{pure}_{\mathsf{option}}$ = Some
fun $(\diamond)_{\mathsf{option}}$ where $(\mathsf{Some}\ f) \diamond_{\mathsf{option}} (\mathsf{Some}\ x) = \mathsf{Some}\ (f\ x) \mid \_ \diamond_{\mathsf{option}} \_ = \mathsf{None}$

Zip list

codatatype α llist = [ ] | $\alpha \cdot \alpha$ llist
primcorec $\mathsf{pure}_{\mathsf{llist}}\ x = x \cdot \mathsf{pure}_{\mathsf{llist}}\ x$
primcorec $(f \cdot fs) \diamond_{\mathsf{llist}} (x \cdot xs) = f\ x \cdot (fs \diamond_{\mathsf{llist}} xs) \mid \_ \diamond_{\mathsf{llist}} \_ = [\,]$

Probability

typedef α pmf = $\{ f :: \alpha \Rightarrow \mathsf{real}.\ (\forall x.\ f\ x \geq 0) \wedge (\sum_x f\ x) = 1 \}$
lift-definition $\mathsf{pure}_{\mathsf{pmf}}$ is $\lambda x\ y.$ if $x = y$ then 1 else 0
lift-definition $(\diamond)_{\mathsf{pmf}}$ is $\lambda F\ X\ y.\ \sum_{\{(f,x).\ f\ x = y\}} F\ f \cdot X\ x$

Subprobability

type-synonym α spmf = α option pmf
definition $\mathsf{pure}_{\mathsf{spmf}} = \mathsf{pure}_{\mathsf{pmf}}\ \mathsf{pure}_{\mathsf{option}}$
definition $f \diamond_{\mathsf{spmf}} x = \mathsf{pure}_{\mathsf{pmf}}\ (\diamond)_{\mathsf{option}} \diamond_{\mathsf{pmf}} f \diamond_{\mathsf{pmf}} x$

Set

definition $\mathsf{pure}_{\mathsf{set}}\ x = \{\, x \,\}$
definition $F \diamond_{\mathsf{set}} X = \{\, f\ x.\ f \in F \wedge x \in X \,\}$

Non-empty set

typedef α neset = $\{\, A :: \alpha\ \mathsf{set}.\ A \neq \emptyset \,\}$
lift-definition $\mathsf{pure}_{\mathsf{neset}}$ is $\mathsf{pure}_{\mathsf{set}}$
lift-definition $(\diamond)_{\mathsf{neset}}$ is $(\diamond)_{\mathsf{set}}$

Monoid, commutative monoid, idempotent monoid

type-synonym (α,β) monoid-ap = $\alpha \times \beta$
definition $\mathsf{pure}_{\mathsf{monoid}}\ x = (0, x)$
fun $(\diamond)_{\mathsf{monoid}}$ where $(a, f) \diamond_{\mathsf{monoid}} (b, x) = (a + b, f\ x)$

The type variable α must have sort monoid-add. If α has sort comm-monoid-add, then monoid-ap has **C**. If α has sort idemp-monoid-add, then monoid-ap has **W**.

Either

datatype (α,β) either = Left α | Right β
definition $\mathsf{pure}_{\mathsf{either}}$ = Left
fun $(\diamond)_{\mathsf{either}}$ where

```
     Left f   ◇either Left x   = Left (f x)
   | _        ◇either Right y  = Right y
   | Right y  ◇either Left _   = Right y
```

Distinct non-empty list The function remdups removes duplicates from a list by retaining only the last occurrence of each element.

typedef α dnelist = $\{\, xs :: \alpha\ \mathsf{list}.\ \mathsf{distinct}\ xs \wedge xs \neq [\,] \,\}$
lift-definition $\mathsf{pure}_{\mathsf{dnelist}}$ is $\mathsf{pure}_{\mathsf{list}}$
lift-definition $(\diamond)_{\mathsf{dnelist}}$ is $\lambda f\ x.\ \mathsf{remdups}\ (f \diamond_{\mathsf{list}} x)$

State

type-synonym (α,σ) state = $\sigma \Rightarrow \alpha \times \sigma$
definition $\mathsf{pure}_{\mathsf{state}}$ = Pair
definition $f \diamond_{\mathsf{state}} x = (\lambda s.\ \mathsf{case}\ f\ s\ \mathsf{of}\ (f', s') \Rightarrow \mathsf{case}\ x\ s'\ \mathsf{of}\ (x', s'') \Rightarrow (f'\ x', s''))$

List

datatype α list = $[\,]$ | $\alpha \cdot \alpha$ list
definition $\mathsf{pure}_{\mathsf{list}}\ x = [x]$
definition $f \diamond_{\mathsf{list}} x = \mathsf{concat\text{-}map}\ (\lambda f'.\ \mathsf{map}\ f'\ x)\ f$

Parser The function apfst applies a function to the first component of a tuple.

type-synonym (α,σ) parser = $\sigma \Rightarrow (\alpha \times \sigma)$ list
definition $\mathsf{pure}_{\mathsf{parser}}\ x = (\lambda s.\ [(x, s)])$
definition $f \diamond_{\mathsf{parser}} x = (\lambda s.\ \mathsf{concat\text{-}map}\ (\lambda (f', s').\ \mathsf{map}\ (\mathsf{apfst}\ f')\ (x\ s'))\ (f\ s))$