**HOSC manuscript No.**
(will be inserted by the editor)

# Lifting Operators and Laws

**Ralf Hinze**

Received: date / Accepted: date

**Abstract** Mathematicians routinely lift operators to structures. For instance, almost every textbook on calculus lifts addition pointwise to functions: $(f+g)(x) = f(x)+g(x)$. In this particular example, the lifted operator inherits the properties of the base-level operator. Does this hold in general? In order to approach this problem, one has to make the concept of lifting precise. I argue that lifting can be defined generically using the notion of an applicative functor or idiom. In this setting, the paper answers two questions: "Which lifted base-level identities hold in every idiom?" and "Which idioms satisfy every lifted base-level identity?"

**Keywords** lifting · applicative functor · idiom · idiom homomorphism · environment idiom · monad · combinators · environment model · generalised algebraic datatypes

**CR Subject Classification** D.1.1 [Programming Techniques]: Applicative (Functional) Programming · D.2.4 [Software/Program Verification]: correctness proofs, formal methods · D.3.2 [Programming Languages]: Language Classifications — applicative (functional) languages · F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs — mechanical verification

## 1 Introduction

Mathematicians routinely lift operators to structures so that, for example, '$+$' not only denotes addition, but also addition lifted pointwise to sets, $A+B = \{a+b \mid a \in A, b \in B\}$, or addition lifted pointwise to functions, $(f+g)(x) = f(x)+g(x)$.

Haskell programmers routinely lift operators to container types so that '$+$' not only adds two integers or two floating point numbers, but also adds two elements of a pointed type (*Maybe* $\tau$ in Haskell, $\tau$ *option* in Standard ML), or sequences two parsers adding their semantic values.

I used lifting extensively in recent papers on codata (Hinze 2008, 2009a,b), where '$+$' either zips two streams or two infinite trees adding corresponding elements. The papers are

R. Hinze
Computing Laboratory, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, England
Tel.: +44-1865-610700
Fax: +44-1865-283531
E-mail: ralf.hinze@comlab.ox.ac.uk

mainly concerned with proofs, and I mentioned in passing that the usual arithmetic laws also hold when lifted to streams or infinite trees. I considered this statement to be self-evident, so I did not even bother to prove it. Lifting is, however, a more general concept and, in general, the situation is not so clear cut. For example, while lifted associativity $(x+y)+z = x+(y+z)$ holds in all the example structures above, lifted commutativity $x+y = y+x$ fails for parsers, and the lifted version of $x * 0 = 0$ fails for sets, pointed types and parsers.

I have referred to lifting as a general notion, but is there actually a mathematical concept that unites sets, functions, pointed types, parsers, streams and infinite trees? It turns out that the concept of an applicative functor or idiom (McBride and Paterson 2008) perfectly fits the bill. ('Idiom' was the name McBride originally chose, but he and Paterson now favour the less evocative term 'applicative functor'. I prefer the former over the latter, not least because it lends itself nicely to adjectival uses, as in 'idiomatic expression'. As an aside, Leroy's parametrised modules (1995) are a completely different kind of 'applicative functor'.) In a nutshell, idioms are functors with two additional operations that allow us to lift functions such as '$+$' to the functorial structure — but nothing more.

One can then extend the notion of lifting to applicative expressions and equations in the obvious way: we replace functions by their lifted counterparts and let variables range over the idiomatic structure. The examples above demonstrate that not every lifted law holds in every idiom. Can you spot a pattern? The purpose of this paper is exactly this, to identify patterns. In particular, it answers the following questions:

1. Which lifted base-level identities hold in every idiom?
2. Which idioms satisfy every lifted base-level identity?

In addition, the paper also explores some of the middle ground.

The rest of the paper is structured as follows. Section 2 provides a gentle introduction to idioms. Section 3 introduces idiomatic expressions and answers the first question (Normal-form Lemma). Sections 4, 5 and 6 answer the second question (Lifting Lemma). Section 7 explores the middle ground. Finally, Section 8 reviews related work.

The paper assumes some knowledge of the functional programming language Haskell (Peyton Jones 2003). In particular, we shall use Haskell both for the examples and as a meta-language for the formal development. For reference, Appendix A lists the standard combinators used in the main text.

## 2 Idioms

Categorically, idioms are *strong lax monoidal functors* (Mac Lane 1998). Programmatically, idioms arose as an interface for parsing combinators (Röjemo 1995). In Haskell, we can express the interface by a type class:

```
infixl 6 ⋄
class Idiom ι where
  pure :: α → ι α
  (⋄)  :: ι (α → β) → (ι α → ι β).
```

The constructor class abstracts over a container type; it introduces an operation for embedding a value into a structure, and an application operator that takes a structure of functions to a function that maps a structure of arguments to a structure of results. Like ordinary application, idiomatic application associates to the left. For reasons that become clear later on, we refer to *pure a* as a *pure computation* and to arbitrary elements of the container type as *(potentially) impure computations.*

In the case of parsing combinators, $\iota\ \tau$ is the type of a parser that returns a semantic value of type $\tau$; *pure a* parses the empty string (it always succeeds) and returns $a$ as the semantic value; the parser $p \diamond q$ sequences $p$ and $q$ and returns the semantic value of $p$ applied to the semantic value of $q$.

As a matter of fact, the first combinator-parsing libraries used a slightly different interface (Hutton 1992):

$$\begin{aligned}
&\textbf{infixl}\ 6\ \star\\
&\mathit{map} :: (\alpha \to \beta) \to (\iota\ \alpha \to \iota\ \beta)\\
&\mathit{unit} :: \iota\ ()\\
&(\star)\ \ :: \iota\ \alpha \to \iota\ \beta \to \iota\ (\alpha, \beta),
\end{aligned}$$

which happens to capture the categorical notion of a strong lax monoidal functor and which is equivalent to the interface above (McBride and Paterson 2008). The function *map* defines the morphism part of the functor $\iota$, *unit* creates a structure of empty tuples, and '$\star$' turns a pair of structures into a structure of pairs (the description doesn't quite match the type as Haskell's binary operators are always curried). The two sets of operations are inter-definable, see below, and we will freely mix them.

$$\begin{aligned}
\mathit{map}\ f\ u &= \mathit{pure}\ f \diamond u\\
\mathit{unit} &= \mathit{pure}\ ()\\
u \star v &= \mathit{pure}\ (,) \diamond u \diamond v
\end{aligned}
\qquad
\begin{aligned}
\mathit{pure}\ a &= \mathit{map}\ (\mathit{const}\ a)\ \mathit{unit}\\
u \diamond v &= \mathit{map}\ \mathit{app}\ (u \star v)
\end{aligned}$$

The curious '$(,)$' is Haskell's pairing constructor; *const* and *app* are defined in Appendix A.

We shall refer to the first interface as the *asymmetric interface*. In line with the language Haskell, it favours curried functions, whereas the second interface, the *symmetric interface*, is tailored towards the first-order language of category theory.

All the examples given in the introduction have the structure of an idiom. For instance, here is the instance declaration that turns the *environment functor* $\tau \to$ into an idiom:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Idiom}\ (\tau \to)\ \textbf{where}\\
&\quad \mathit{pure}\ a = \lambda x \to a\\
&\quad u \diamond v \ \ = \lambda x \to (u\ x)\ (v\ x).
\end{aligned}$$

Interestingly, *pure* is the combinator $\mathbb{K}$ and '$\diamond$' is the combinator $\mathbb{S}$ from combinatory logic (Curry and Feys 1958).

The identity type constructor, $\mathit{Id}\ \alpha = \alpha$, is an idiom. Idioms are closed both under type composition, $(\phi \cdot \psi)\ \alpha = \phi\ (\psi\ \alpha)$, and type pairing, $(\phi \times \psi)\ \alpha = (\phi\ \alpha, \psi\ \alpha)$. Every monad is an idiom, but not the other way round. In other words, idioms are weaker than monads and consequently more general. In particular, Haskell's predefined monad of IO computations is an idiom:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Idiom}\ \mathit{IO}\ \textbf{where}\\
&\quad \mathit{pure}\ a = \mathit{return}\ a\\
&\quad u \diamond v \ \ = \textbf{do}\ \{f \leftarrow u; x \leftarrow v; \mathit{return}\ (f\ x)\}.
\end{aligned}$$

So, *pure* is *return* and '$\diamond$' is implemented using two monadic binds. Non-commutative monads such as *IO* and $[\,]$ (the list monad) give, in fact, rise to two idioms as the order in which $u$ and $v$ are executed is significant. As as aside, the proximity to monads inspired the terminology of pure and impure computations.

Every instance of *Idiom* must satisfy four laws:

$$\mathit{pure}\ \mathit{id} \diamond u = u \qquad \text{(idiom identity)}$$

$$\mathit{pure}\ (\cdot) \diamond u \diamond v \diamond w = u \diamond (v \diamond w) \qquad \text{(idiom composition)}$$

$$\mathit{pure}\ f \diamond \mathit{pure}\ x = \mathit{pure}\ (f\ x) \qquad \text{(homomorphism)}$$

$$u \diamond \mathit{pure}\ x = \mathit{pure}\ (\diamond\ x) \diamond u. \qquad \text{(interchange)}$$

The '$\diamond$' in $\mathit{pure}\ (\diamond\, x)$ lives in the identity idiom, that is, $(\diamond\, x)$ takes a function and applies it to $x$.

The laws imply a normalform: every idiomatic expression can be rewritten into the form $\mathit{pure}\, f \diamond u_1 \diamond \cdots \diamond u_n$, a pure function applied to impure arguments. We shall prove this claim in Section 3.3. Put differently, applicative functors or idioms capture the notion of lifting: $\lambda u_1\ \cdots\ u_n \to \mathit{pure}\, f \diamond u_1 \diamond \cdots \diamond u_n$ is the lifted version of the $n$ary function $f$ (assuming that $f$ is curried). For instance, the environment idiom $\tau \to$ captures lifting operators to function spaces: $\mathit{pure}\ (+) \diamond f \diamond g = \mathbb{S}\ (\mathbb{S}\ (\mathbb{K}\ (+))\ f)\ g = \lambda x \to f\ x + g\ x$.

Above we have used the asymmetric interface, which assumes curried functions as the norm. For the symmetric interface, there is a corresponding set of six laws:

$$\begin{array}{lll}
\mathit{map}\ \mathit{id}\ u & = u & \text{(functor identity)}\\
\mathit{map}\ (f \cdot g)\ u & = \mathit{map}\, f\ (\mathit{map}\ g\ u) & \text{(functor composition)}\\
\mathit{map}\ (f \times g)\ (u \star v) & = \mathit{map}\, f\ u \star \mathit{map}\ g\ v & \text{(naturality of } \star)\\
\mathit{map}\ \mathit{snd}\ (\mathit{unit} \star v) & = v & \text{(left identity)}\\
\mathit{map}\ \mathit{fst}\ (u \star \mathit{unit}) & = u & \text{(right identity)}\\
\mathit{map}\ \mathit{assocl}\ (u \star (v \star w)) & = (u \star v) \star w, & \text{(associativity)}
\end{array}$$

and a corresponding notion of normalform: every idiomatic expression can be rewritten into the form $\mathit{map}\, f\ (u_1 \star \cdots \star u_n)$, a pure function applied to a tuple of impure arguments (here $f$ is uncurried).

We should note that currying is valid in every idiomatic structure, $\mathit{pure}\ (\mathit{curry}\, f) \diamond u \diamond v = \mathit{pure}\, f \diamond (u \star v)$, where $\mathit{curry}\, f\ x\ y = f\ (x,y)$, as a somewhat lengthy calculation shows:

$$\begin{array}{ll}
& \mathit{pure}\, f \diamond (u \star v)\\
= & \quad \{\ \text{definition of '}\star\text{'}\ \}\\
& \mathit{pure}\, f \diamond (\mathit{pure}\ (,) \diamond u \diamond v)\\
= & \quad \{\ \text{idiom composition}\ \}\\
& \mathit{pure}\ (\cdot) \diamond \mathit{pure}\, f \diamond (\mathit{pure}\ (,) \diamond u) \diamond v\\
= & \quad \{\ \text{idiom homomorphism}\ \}\\
& \mathit{pure}\ (f\ \cdot) \diamond (\mathit{pure}\ (,) \diamond u) \diamond v\\
= & \quad \{\ \text{idiom composition}\ \}\\
& \mathit{pure}\ (\cdot) \diamond \mathit{pure}\ (f\ \cdot) \diamond \mathit{pure}\ (,) \diamond u \diamond v\\
= & \quad \{\ \text{idiom homomorphism}\ \}\\
& \mathit{pure}\ ((f\ \cdot) \cdot (,)) \diamond u \diamond v\\
= & \quad \{\ ((f\ \cdot) \cdot (,))\ x\ y = (f \cdot (x,))\ y = f\ (x,y) = f\ (x,y) = \mathit{curry}\, f\ x\ y\ \}\\
& \mathit{pure}\ (\mathit{curry}\, f) \diamond u \diamond v.
\end{array}$$

The proof involves a lot of plumbing, which is typical of reasoning with idioms. Clearly, lifting the definitional equality $\mathit{curry}\, f\ x\ y = f\ (x,y)$ should involve less work. (The statement can, in fact, be generalised to $\mathit{pure}\ \mathit{curry} \diamond f \diamond u \diamond v = f \diamond (u \star v)$, whose even more laborious proof is left as an exercise to the reader.)

The interchange law allows us to swap pure and impure computations. This move possibly brings together pure computations, which we can subsequently merge using the homomorphism law. However, we cannot interchange impure computations, for instance, the *IO* computation *putStr* `"hi"` $\star$ *putStr* `"ho"` is different from *putStr* `"ho"` $\star$ *putStr* `"hi"`. These observations already hint at the answer to the first question, “Which lifted base-level identities hold in every idiom?” Let us consider an example first. The following calculation shows that lifted associativity holds in every idiom:

$$\begin{aligned}
&\quad \mathit{pure}\ (+) \diamond u \diamond (\mathit{pure}\ (+) \diamond v \diamond w)\\
&= \quad \{\ \text{Normalform Lemma}\ \}\\
&\quad \mathit{pure}\ (\lambda x\ y\ z \to x + (y + z)) \diamond u \diamond v \diamond w\\
&= \quad \{\ x + (y + z) = (x + y) + z\ \}\\
&\quad \mathit{pure}\ (\lambda x\ y\ z \to (x + y) + z) \diamond u \diamond v \diamond w\\
&= \quad \{\ \text{Normalform Lemma}\ \}\\
&\quad \mathit{pure}\ (+) \diamond (\mathit{pure}\ (+) \diamond u \diamond v) \diamond w.
\end{aligned}$$

The variables $u$, $v$ and $w$ range over the idiomatic structure, so they are potentially impure. Since the variables appear in the same order on both sides of the equation, we can normalise both sides and then apply the base-level identity. This approach does *not* work if the variables appear in a different order. We can still apply the Normalform Lemma, however, the two sequences of impure expressions won’t match. The approach also fails if a variable appears more than once. In this case, we can’t apply the base-level identity as the following example illustrates. Consider the made-up law $x \oplus x = x \otimes x$ where ‘$\oplus$’ and ‘$\otimes$’ are some invented operators. The lifted version of the law, $\mathit{pure}\ (\oplus) \diamond u \diamond u = \mathit{pure}\ (\otimes) \diamond u \diamond u$, is already in normalform. However, the base-level identity does not imply $(\oplus) = (\otimes)$, which is equivalent to $x \oplus y = x \otimes y$. And indeed, in the *IO* idiom $\mathit{readLn} \oplus \mathit{readLn}$ is different from $\mathit{readLn} \otimes \mathit{readLn}$ as the two input statements possibly yield different values.

Now, to formalise the Normalform Lemma we have to make the syntax and semantics of idiomatic expressions precise. This is what we do next. We employ Haskell also for the formal development, making intensive use of generalised algebraic datatypes, a recent extension to the language (Hinze 2003; Peyton Jones et al. 2006).

## 3 The $\iota$-calculus

### 3.1 Syntax

An idiomatic expression is basically a binary leaf tree, where the inner nodes represent applications and the outer nodes represent constants (pure computations) and variables (impure computations). Since we are only interested in well-formed expressions, we represent variables by De Bruijn indices and bake the typing rules into the datatype declaration:

```
data Ix :: * → * → * where
  Zero :: Ix (ρ, α) α
  Succ :: Ix ρ β → Ix (ρ, α) β
data Term :: * → * → * where
  Con  :: α → Term ρ α
  Var  :: Ix ρ α → Term ρ α
  App  :: Term ρ (α → β) → Term ρ α → Term ρ β.
```

An element of type $\mathit{Ix}\ \rho\ \tau$ is a De Bruijn index of type $\tau$ relative to the typing context $\rho$. For instance, $\mathit{Zero} :: \mathit{Ix}\ ((\rho, \tau_1), \tau_0)\ \tau_0$ and $\mathit{Succ}\ \mathit{Zero} :: \mathit{Ix}\ ((\rho, \tau_1), \tau_0)\ \tau_1$. For the purposes of this paper, the typing context is always a left-nested product type. Likewise, an element of type

$\mathit{Term}\ \rho\ \tau$ is a well-formed term of type $\tau$ relative to the typing context $\rho$. As an example, the idiomatic expression $\mathit{pure}\ (+) \diamond u \diamond v$, where $u$ and $v$ are variables, is represented by

$$\begin{aligned}
&\mathit{ex}_1 :: \mathit{Term}\ ((\rho, \mathit{Integer}), \mathit{Integer})\ \mathit{Integer}\\
&\mathit{ex}_1 = \mathit{App}\ (\mathit{App}\ (\mathit{Con}\ (+))\ (\mathit{Var}\ (\mathit{Succ}\ \mathit{Zero})))\ (\mathit{Var}\ \mathit{Zero}).
\end{aligned}$$

We sometimes use numeric literals for De Bruijn indices and abbreviate $\mathit{App}\ u\ v$ by $u \mathbin{\dot{\diamond}} v$. With these conventions in place, $\mathit{ex}_1$ can be written more succinctly as $\mathit{Con}\ (+) \mathbin{\dot{\diamond}} 1 \mathbin{\dot{\diamond}} 0$.

For the symmetric interface we introduce a corresponding set of constructors:

$$\begin{aligned}
&\mathit{Map} :: (\alpha \to \beta) \to (\mathit{Term}\ \rho\ \alpha \to \mathit{Term}\ \rho\ \beta)\\
&\mathit{Unit} :: \mathit{Term}\ \rho\ ()\\
&\mathit{Pair} :: \mathit{Term}\ \rho\ \alpha \to \mathit{Term}\ \rho\ \beta \to \mathit{Term}\ \rho\ (\alpha, \beta).
\end{aligned}$$

For conciseness of notation, we usually write $\mathit{Pair}$ infix as $\mathbin{\dot{\star}}$.

## 3.2 Semantics

Turning to the semantics of idiomatic terms, we first define the notion of an environment:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Env} :: (* \to *) \to * \to * \ \mathbf{where}\\
&\quad \mathit{Empty} :: \mathit{Env}\ \iota\ ()\\
&\quad \mathit{Push}\ \ :: \mathit{Env}\ \iota\ \rho \to \iota\ \alpha \to \mathit{Env}\ \iota\ (\rho, \alpha).
\end{aligned}$$

The type $\mathit{Env}$ is parametrised by the idiom and the typing context. An element of type $\mathit{Env}\ \iota\ \rho$ is a well-formed environment in the typing context $\rho$ that contains $\iota$ structures. We abbreviate $\mathit{Empty}$ by $\langle\rangle$ and $\mathit{Push}\ \eta\ u$ by $\langle \eta, u \rangle$. As an example, a suitable environment for interpreting the term $\mathit{ex}_1$ is

$$\begin{aligned}
&\eta_1 :: \mathit{Env}\ (\mathit{Integer} \to)\ (((), \mathit{Integer}), \mathit{Integer})\\
&\eta_1 = \langle \langle \langle\rangle, \lambda n \to 2 * n \rangle, \lambda n \to 2 * n + 1 \rangle.
\end{aligned}$$

Here $\iota$ is instantiated to the environment idiom $\mathit{Integer} \to$ . Consequently, the entries are functions from the integers.

Like the syntax, the semantics is split into two parts: we provide semantic equations for De Bruijn indices and semantic equations for idiomatic terms.

$$\begin{aligned}
&\mathit{acc} && :: \mathit{Ix}\ \rho\ \alpha \to \mathit{Env}\ \iota\ \rho \to \iota\ \alpha\\
&\mathit{acc}\ \mathit{Zero}\ \ \langle \eta, u \rangle && = u\\
&\mathit{acc}\ (\mathit{Succ}\ n)\ \langle \eta, u \rangle && = \mathit{acc}\ n\ \eta\\
&\mathscr{I}[\![]\!] && :: (\mathit{Idiom}\ \iota) \Rightarrow \mathit{Term}\ \rho\ \alpha \to \mathit{Env}\ \iota\ \rho \to \iota\ \alpha\\
&\mathscr{I}[\![\mathit{Con}\ u]\!]\eta && = \mathit{pure}\ u\\
&\mathscr{I}[\![\mathit{Var}\ n]\!]\eta && = \mathit{acc}\ n\ \eta\\
&\mathscr{I}[\![\mathit{App}\ e_1\ e_2]\!]\eta && = \mathscr{I}[\![e_1]\!]\eta \diamond \mathscr{I}[\![e_2]\!]\eta
\end{aligned}$$

The function $\mathit{acc}$ looks an index up in the environment; $\mathscr{I}[\![e]\!]\eta$ interprets the term $e$ in the environment $\eta$. As an example,

$$\mathscr{I}[\![\mathit{ex}_1]\!]\eta_1 = \mathit{pure}\ (+) \diamond (\lambda n \to 2 * n) \diamond (\lambda n \to 2 * n + 1) = \lambda n \to 4 * n + 1.$$

The idiomatic structure in which the term is interpreted is implicitly provided by the class context. In the example above, $\iota$ is instantiated to the environment idiom $\mathit{Integer} \to$ because of $\eta_1$'s type.

If $e$ is a closed term, we abbreviate $\mathscr{I}[\![e]\!]\langle\rangle$ by $\mathscr{I}[\![e]\!]$. Two terms $e_1$ and $e_2$ of type $\mathit{Term}\ \rho\ \tau$ are *equivalent* iff $\mathscr{I}[\![e_1]\!]\eta = \mathscr{I}[\![e_2]\!]\eta$ for all idioms $\iota$ and for all environments of type $\mathit{Env}\ \iota\ \rho$.

## 3.3 Normalisation

**Lemma 1 (Normalform)** *Let $e$ be an idiomatic term that contains the* list *of variables* $\mathit{Var}\ i_1, \ldots, \mathit{Var}\ i_n$*. Then $e$ is equivalent to*

1. $\mathit{Con}\ f \circledast \mathit{Var}\ i_1 \circledast \cdots \circledast \mathit{Var}\ i_n$ *for some suitable $f$, and to*
2. $\mathit{Map}\ g \circledast (\mathit{Var}\ i_1 \star \cdots \star \mathit{Var}\ i_n)$ *for some suitable $g$.*

*Proof* Part 1 is the curried version of Part 2, so it suffices to prove the latter. Since we have formalised the syntax and semantics of idiomatic expressions in Haskell, we can actually program the normalisation. Assuming the symmetric interface, we proceed in two steps. First, we move all occurrences of *Map* to the front: $e$ is transformed into $\mathit{Map}\ f\ u$ where $u$ is a nested pair of variables and units. Second, we turn $u$ into a left-linear tree, that is, a 'snoc-list' of variables.

```
norm   :: Term ρ α → Term ρ α
norm e = case norm₁ e of Map f u → case norm₂ u of Map g v → Map (f · g) v
```

Both transformations return a term of the form $\mathit{Map}\ f\ u$.

```
norm₁              :: Term ρ α → Term ρ α
norm₁ (Var n)      = Map id (Var n)      -- functor identity
norm₁ (Map f e) =                        -- functor composition
  case norm₁ e of Map g u → Map (f · g) u
norm₁ Unit         = Map id Unit         -- functor identity
norm₁ (e₁ ⋆ e₂)  =                       -- naturality of ⋆
  case (norm₁ e₁, norm₁ e₂) of (Map f₁ u₁, Map f₂ u₂) → Map (f₁ × f₂) (u₁ ⋆ u₂)
```

Each equation is based on the idiom law listed in the comment on the right.

The correctness of the second transformation relies on the monoidal laws.

```
norm₂                   :: Term ρ α → Term ρ α
norm₂ (Var n)           = Map id (Var n)                      -- functor identity
norm₂ Unit              = Map id Unit                         -- functor identity
norm₂ (Unit ⋆ e₂)       =
  case norm₂ e₂ of Map f₂ u₂ → Map (const () △ f₂) u₂       -- left identity
norm₂ (e₁ ⋆ Unit)       =
  case norm₂ e₁ of Map f₁ u₁ → Map (f₁ △ const ()) u₁       -- right identity
norm₂ (e₁ ⋆ Var n)      =
  case norm₂ e₁ of Map f₁ u₁ → Map (f₁ × id) (u₁ ⋆ Var n)
norm₂ (e₁ ⋆ (e₂ ⋆ e₃)) =                                       -- associativity
  case norm₂ ((e₁ ⋆ e₂) ⋆ e₃) of Map f u → Map (assocr · f) u
```

Note that $\mathit{id} \vartriangle \mathit{const}\ ()$ is the inverse of $\mathit{fst} :: (\alpha, ()) \to \alpha$. □

To summarise: The interchange law allows us to swap pure and impure computations, however, we can neither re-order, nor omit, nor duplicate arbitrary computations. Consequently, only laws that contain the same *list* of variables on the left- and right-hand side with no repeated variables are valid in every idiom.

## 4 The $\lambda\iota$K-calculus

Idiomatic expressions are tantalisingly close to expressions of the simply typed lambda-calculus: we have constants, variables and application; only lambda-abstraction is missing. It is, of course, easy to add the construct to the syntax

$$\mathit{Abs} :: \mathit{Term}\ (\rho, \alpha)\ \beta \to \mathit{Term}\ \rho\ (\alpha \to \beta),$$

but how can we assign a meaning to it? Borrowing from the semantics of the simply typed lambda-calculus (Mitchell 1996) we define (in pseudo-Haskell)

$$\mathscr{I}[\![\mathit{Abs}\ e]\!]\eta = \text{the unique } f \text{ such that } \forall u\ .\ f \diamond u = \mathscr{I}[\![e]\!]\langle \eta, u\rangle.$$

Clearly, we have to impose further conditions on an idiom in order for this definition to make sense: An idiom must be extensional so that the postulated 'function' is indeed unique and the idiom must contain "enough elements" to guarantee its existence.

**Definition 1 (Extensionality)**

- An idiom $\iota$ is *extensional* iff for all $f, g :: \iota\ (\sigma \to \tau)$,

  $$(\forall u :: \iota\ \sigma\ .\ f \diamond u = g \diamond u) \Longrightarrow f = g.$$

- An idiom $\iota$ is *strongly extensional* iff for all $f, g :: \iota\ (\sigma \to \tau)$,

  $$(\forall a :: \sigma\ .\ f \diamond \mathit{pure}\ a = g \diamond \mathit{pure}\ a) \Longrightarrow f = g.$$

Clearly, strong extensionality implies extensionality. The environment idiom, for instance, is strongly extensional, as the following straightforward calculation shows:

$$\begin{aligned}
& \forall a :: \sigma\ .\ f \diamond \mathit{pure}\ a = g \diamond \mathit{pure}\ a \\
\Longleftrightarrow\quad & \{\text{ definition of } \mathit{pure} \text{ and '}\diamond\text{' }\} \\
& \forall a :: \sigma\ .\ \lambda x \to (f\ x)\ a = \lambda x \to (g\ x)\ a \\
\Longrightarrow\quad & \{\text{ equality of functions }\} \\
& \forall a :: \sigma, x :: \tau\ .\ (f\ x)\ a = (g\ x)\ a \\
\Longleftrightarrow\quad & \{\text{ logic }\} \\
& \forall x :: \tau\ .\ \forall a :: \sigma\ .\ (f\ x)\ a = (g\ x)\ a \\
\Longrightarrow\quad & \{\text{ extensionality of functions, twice }\} \\
& f = g.
\end{aligned}$$

The *Maybe* idiom and the list idiom are also strongly extensional. The set idiom, on the other hand, is not extensional: $f = \{\mathit{const\ False}, \mathit{const\ True}\}$ and $g = \{\mathit{id}, \mathit{not}\}$ of type $\mathit{Set}\ (\mathit{Bool} \to \mathit{Bool})$ satisfy the antecedent $(\forall u :: \mathit{Set\ Bool}\ .\ f \diamond u = g \diamond u)$, but not the consequent $(f \neq g)$.

**Definition 2 (Combinatory model condition)** An idiom $\iota$ has K-combinators iff there exist elements $k :: \iota\ (\alpha \to \beta \to \alpha)$ and $s :: \iota\ ((\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma))$ satisfying

$$\begin{aligned}
k \diamond u \diamond v &= u \\
s \diamond u \diamond v \diamond w &= (u \diamond w) \diamond (v \diamond w),
\end{aligned}$$

for all $u$, $v$, $w$ of the appropriate types.

In our setting, the existence of $k$ means that we can omit 'effects'; the existence of $s$ implies that we can duplicate 'effects'. If both combinators exist, then we can also re-order 'effects'.

It is well-known that lambda-abstraction can be simulated with the combinators $k$, $s$ and $i = s \diamond k \diamond k$:

$$\begin{array}{ll}
\mathit{abs} & :: \mathit{Term}\,(\rho,\alpha)\,\beta \to \mathit{Term}\,\rho\,(\alpha \to \beta) \\
\mathit{abs}\,(\mathit{Con}\,u) & = K \mathbin{:\!\diamond} \mathit{Con}\,u \\
\mathit{abs}\,(\mathit{Var}\,\mathit{Zero}) & = I \\
\mathit{abs}\,(\mathit{Var}\,(\mathit{Succ}\,n)) & = K \mathbin{:\!\diamond} \mathit{Var}\,n \\
\mathit{abs}\,(e_1 \mathbin{:\!\diamond} e_2) & = S \mathbin{:\!\diamond} \mathit{abs}\,e_1 \mathbin{:\!\diamond} \mathit{abs}\,e_2.
\end{array}$$

Here, $K$, $S$ and $I$ are the syntactic counterparts of the combinators: $\mathscr{I}[\![K]\!]\eta = k$ etc.

**Lemma 2 (Abstraction)** *Let $\iota$ be an idiom with K-combinators, and let $e :: \mathit{Term}\,(\rho,\sigma)\,\tau$ be an idiomatic term. Then*

$$\mathscr{I}[\![\mathit{abs}\,e]\!]\eta \diamond u \;=\; \mathscr{I}[\![e]\!]\langle\eta,u\rangle,$$

*for all $u :: \iota\,\sigma$ and for all environments $\eta :: \mathit{Env}\,\rho\,\iota$.*

*Proof* Using a straightforward induction over the structure of idiomatic terms. □

Extensionality and the Abstraction Lemma then imply that $\mathscr{I}$ is well-defined.

Obvious candidates for $k$ and $s$ are *pure* $\mathbb{K}$ and *pure* $\mathbb{S}$ — note that the combinators have to be polymorphic. And indeed, if a strongly extensional idiom has K-combinators, then $k = \mathit{pure}\;\mathbb{K}$ and $s = \mathit{pure}\;\mathbb{S}$. For instance, the environment idiom has K-combinators:

$$\begin{array}{ll}
 & \mathit{pure}\;\mathbb{K} \diamond u \diamond v \\
= & \{\text{ definition of } \mathit{pure} \text{ and '}\diamond\text{' }\} \\
 & \lambda x \to \mathbb{K}\,(u\,x)\,(v\,x) \\
= & \{\text{ definition of } \mathbb{K} \,\} \\
 & \lambda x \to u\,x \\
= & \{\text{ extensionality }\} \\
 & u,
\end{array}
\qquad
\begin{array}{ll}
 & \mathit{pure}\;\mathbb{S} \diamond u \diamond v \diamond w \\
= & \{\text{ definition of } \mathit{pure} \text{ and '}\diamond\text{' }\} \\
 & \lambda x \to \mathbb{S}\,(u\,x)\,(v\,x)\,(w\,x) \\
= & \{\text{ definition of } \mathbb{S} \,\} \\
 & \lambda x \to ((u\,x)\,(w\,x))\,((v\,x)\,(w\,x)) \\
= & \{\text{ definition of '}\diamond\text{' }\} \\
 & (u \diamond w) \diamond (v \diamond w).
\end{array}$$

The set idiom, on the other hand, has no $k$ combinator, as $u \diamond v \diamond \emptyset = \emptyset$. Likewise, parser idioms don't have a $k$ combinator as $u \diamond v \diamond \mathit{fail} = \mathit{fail}$. Furthermore, a $k$ combinator doesn't exist in the $\mathit{IO}$ idiom because $u \diamond v \diamond \mathit{putStrLn}$ `"oops"` $\neq v$.

For strongly extensional idioms with K-combinators, left and right identity laws can be strengthened to

$$\mathit{pure}\;\mathit{fst} \diamond (u \star v) \;= u \tag{1}$$

$$\mathit{pure}\;\mathit{snd} \diamond (u \star v) = v. \tag{2}$$

The proofs are straightforward noting that $\mathit{curry}\;\mathit{fst} = \mathbb{K}$ and $\mathit{curry}\;\mathit{snd} = \mathbb{K}\,\mathbb{I}$.

The simplest idiom, the identity idiom $\mathit{Id}$, is, of course, strongly extensional and it has K-combinators. For $\mathit{Id}$, the idiomatic semantics specialises to the standard interpretation of the simply typed lambda-calculus:

$$\begin{array}{ll}
[\![\,]\!] & :: \mathit{Term}\,\rho\,\alpha \to \mathit{Env}\,\mathit{Id}\,\rho \to \alpha \\
[\![\mathit{Con}\,v]\!]\eta & = v \\
[\![\mathit{Var}\,n]\!]\eta & = \mathit{acc}\,n\,\eta \\
[\![\mathit{App}\,e_1\,e_2]\!]\eta & = ([\![e_1]\!]\eta)\,([\![e_2]\!]\eta) \\
[\![\mathit{Abs}\,e]\!]\eta & = \lambda u \to [\![e]\!]\langle\eta,u\rangle.
\end{array}$$

We can write the semantic equations more succinctly in point-free style using the environment idiom, hence its name.

$$\begin{aligned}
&acc && :: Ix\ \rho\ \alpha \to Env\ Id\ \rho \to \alpha\\
&acc\ Zero && = snd\\
&acc\ (Succ\ n) && = acc\ n \cdot fst
\end{aligned}$$

$$\begin{aligned}
&[\![\,]\!] && :: Term\ \rho\ \alpha \to Env\ Id\ \rho \to \alpha\\
&[\![Con\ v]\!] && = \mathbb{K}\ v\\
&[\![Var\ n]\!] && = acc\ n\\
&[\![App\ e_1\ e_2]\!] && = \mathbb{S}\ [\![e_1]\!]\ [\![e_2]\!]\\
&[\![Abs\ e]\!] && = curry\ [\![e]\!]
\end{aligned}$$

The point-free definition emphasises the fact that the interpretation function is a catamorphism: *Con* is replaced by $\mathbb{K}$, *Var* by *acc*, *App* by $\mathbb{S}$, and *Abs* by *curry*. (Strictly taken, *curry* should be defined as $curry\ f\ x\ y = f\ \langle x, y\rangle$; we silently ignore the difference between $(\,,\,)$ and $\langle,\rangle$.) This definition amounts to the interpretation of the simply typed lambda-calculus in a cartesian closed category. (Since category theory is a first-order language, $S\ f\ g$ is additionally replaced by $app \cdot (f \vartriangle g)$.)

## 5 The Lifting Lemma

If an idiomatic term contains no variables, then the Normalform Lemma implies that the term is equivalent to a pure one — the proof boils down to repeated applications of the homomorphism law. This statement holds true if abstractions enter the scene. If $e$ is a closed lambda-term, then

$$\mathscr{I}[\![e]\!] = pure\ [\![e]\!]. \tag{3}$$

Perhaps surprisingly, this simple statement answers the second question, "Which idioms satisfy every lifted base-level identity?" This is the case for strongly extensional idioms with K-combinators. To illustrate, consider the following generic proof of lifted distributivity. Recall that distributivity does not hold in every idiom, because the *list* of variables on the left-hand side and on the right-hand side is different.

$$\begin{aligned}
& pure\ (*) \diamond (pure\ (+) \diamond u \diamond v) \diamond w\\
=\ & \quad \{\text{ definition of } \mathscr{I} \}\\
& \mathscr{I}[\![Abs(Abs(Abs(Con(*) :\diamond (Con(+) :\diamond 2 :\diamond 1) :\diamond 0)))]\!] \diamond u \diamond v \diamond w\\
=\ & \quad \{\text{ Lifting Lemma }\}\\
& pure\ [\![Abs(Abs(Abs(Con(*) :\diamond (Con(+) :\diamond 2 :\diamond 1) :\diamond 0)))]\!] \diamond u \diamond v \diamond w\\
=\ & \quad \{\text{ definition of } [\![\,]\!] \}\\
& pure\ (\lambda x\ y\ z \to (x+y)*z) \diamond u \diamond v \diamond w\\
=\ & \quad \{\text{ arithmetic }\}\\
& pure\ (\lambda x\ y\ z \to x*z+y*z) \diamond u \diamond v \diamond w\\
=\ & \quad \{\text{ definition of } [\![\,]\!] \}\\
& pure\ [\![Abs(Abs(Abs(Con(+) :\diamond (Con(*) :\diamond 2 :\diamond 0) :\diamond (Con(*) :\diamond 1 :\diamond 0))))]\!] \diamond u \diamond v \diamond w\\
=\ & \quad \{\text{ Lifting Lemma }\}\\
& \mathscr{I}[\![Abs(Abs(Abs(Con(+) :\diamond (Con(*) :\diamond 2 :\diamond 0) :\diamond (Con(*) :\diamond 1 :\diamond 0))))]\!] \diamond u \diamond v \diamond w
\end{aligned}$$

$=\quad\{\text{ definition of } \mathscr{I}\ \}$
$\quad\mathit{pure}\ (+) \diamond (\mathit{pure}\ (*) \diamond u \diamond w) \diamond (\mathit{pure}\ (*) \diamond v \diamond w)$

The structure of the proof is simple; so simple that it could be easily mechanised: we reify the idiomatic expressions, appeal to the Lifting Lemma and then apply the base-level identity.

In order to prove the Lifting Lemma we need to generalise Equation 3 to terms with free variables. To this end, we introduce a function, known as an *applicative distributor*, that turns an environment of structures into a structure of environments (again, we identify $()$ and $\langle\rangle$, $(,)$ and $\langle,\rangle$):

$$\begin{aligned}
&\mathit{dist} && :: (\mathit{Idiom}\ \iota) \Rightarrow \mathit{Env}\ \iota\ \rho \to \iota\ (\mathit{Env}\ \mathit{Id}\ \rho)\\
&\mathit{dist}\ \langle\rangle && = \mathit{unit}\\
&\mathit{dist}\ \langle \mathit{vs}, v\rangle && = \mathit{dist}\ \mathit{vs} \star v\,.
\end{aligned}$$

**Lemma 3 (Lifting)** *Let $\iota$ be a strongly extensional idiom with K-combinators, and let $e :: \mathit{Term}\ \rho\ \tau$ be a lambda-term. Then*

$$\mathscr{I}[\![e]\!]\eta\ = \mathit{pure}\ [\![e]\!] \diamond \mathit{dist}\ \eta\,,$$

*for all environments $\eta :: \mathit{Env}\ \iota\ \rho$.*

*Proof* First of all, strong extensionality implies that $k = \mathit{pure}\ \mathbb{K}$ and $s = \mathit{pure}\ \mathbb{S}$. The proof then proceeds by induction over the structure of idiomatic terms. **Case** $e = \mathit{Con}\ v$:

$\quad\mathit{pure}\ [\![\mathit{Con}\ v]\!] \diamond \mathit{dist}\ \eta$
$=\quad\{\text{ definition of } [\![]\!]\ \}$
$\quad\mathit{pure}\ (\mathbb{K}\ v) \diamond \mathit{dist}\ \eta$
$=\quad\{\text{ idiom homomorphism }\}$
$\quad\mathit{pure}\ \mathbb{K} \diamond \mathit{pure}\ v \diamond \mathit{dist}\ \eta$
$=\quad\{\ \mathit{pure}\ \mathbb{K} \text{ is the } k \text{ combinator }\}$
$\quad\mathit{pure}\ v$
$=\quad\{\text{ definition of } \mathscr{I}\ \}$
$\quad\mathscr{I}[\![\mathit{Con}\ v]\!]\eta\,.$

**Case** $e = \mathit{Var}\ n$:

$\quad\mathit{pure}\ [\![\mathit{Var}\ n]\!] \diamond \mathit{dist}\ \eta$
$=\quad\{\text{ definition of } [\![]\!]\ \}$
$\quad\mathit{pure}\ (\mathit{acc}\ n) \diamond \mathit{dist}\ \eta$
$=\quad\{\text{ proof obligation: } \mathit{pure}\ (\mathit{acc}\ n) \diamond \mathit{dist}\ \eta = \mathit{acc}\ n\ \eta\ \}$
$\quad\mathit{acc}\ n\ \eta$
$=\quad\{\text{ definition of } \mathscr{I}\ \}$
$\quad\mathscr{I}[\![\mathit{Var}\ n]\!]\eta\,.$

The proof obligation can be discharged by a straightforward induction over the structure of De Bruijn indices using (1) and (2). Note that the type of $n$ ensures that $\eta$ is non-empty.

**Sub-case** $n = Zero$:

$$\begin{aligned}
& pure\ (acc\ Zero) \diamond dist\ \langle \eta, v \rangle \\
= \quad & \{ \text{definition of } acc \text{ and definition of } dist \} \\
& pure\ snd \diamond (dist\ \eta \star v) \\
= \quad & \{ (2) \} \\
& v \\
= \quad & \{ \text{definition of } acc \} \\
& acc\ Zero\ \langle \eta, v \rangle .
\end{aligned}$$

**Sub-case** $n = Succ\ m$:

$$\begin{aligned}
& pure\ (acc\ (Succ\ m)) \diamond dist\ \langle \eta, v \rangle \\
= \quad & \{ \text{definition of } acc \text{ and definition of } dist \} \\
& pure\ (acc\ m \cdot fst) \diamond (dist\ \eta \star v) \\
= \quad & \{ \text{idiom composition and homomorphism} \} \\
& pure\ (acc\ m) \diamond (pure\ fst \diamond (dist\ \eta \star v)) \\
= \quad & \{ (1) \} \\
& pure\ (acc\ m) \diamond dist\ \eta \\
= \quad & \{ \text{ex hypothesi} \} \\
& acc\ m\ \eta \\
= \quad & \{ \text{definition of } acc \} \\
& acc\ (Succ\ m)\ \langle \eta, v \rangle .
\end{aligned}$$

**Case** $e = App\ e_1\ e_2$:

$$\begin{aligned}
& pure\ [\![App\ e_1\ e_2]\!] \diamond dist\ \eta \\
= \quad & \{ \text{definition of } [\![\,]\!] \} \\
& pure\ (\mathbb{S}\ [\![e_1]\!]\ [\![e_2]\!]) \diamond dist\ \eta \\
= \quad & \{ \text{idiom homomorphism} \} \\
& pure\ \mathbb{S} \diamond pure\ [\![e_1]\!] \diamond pure\ [\![e_2]\!] \diamond dist\ \eta \\
= \quad & \{ pure\ \mathbb{S} \text{ is the } s \text{ combinator} \} \\
& (pure\ [\![e_1]\!] \diamond dist\ \eta) \diamond (pure\ [\![e_2]\!] \diamond dist\ \eta) \\
= \quad & \{ \text{ex hypothesi} \} \\
& \mathscr{I}[\![e_1]\!]\eta \diamond \mathscr{I}[\![e_2]\!]\eta \\
= \quad & \{ \text{definition of } \mathscr{I} \} \\
& \mathscr{I}[\![App\ e_1\ e_2]\!]\eta .
\end{aligned}$$

**Case** $e = Abs\ e$: Let $f = \mathscr{I}[\![Abs\ e]\!]\eta$, then

$$\begin{aligned}
& pure\ [\![Abs\ e]\!] \diamond dist\ \eta \\
= \quad & \{ \text{definition of } [\![\,]\!] \} \\
& pure\ (curry\ [\![e]\!]) \diamond dist\ \eta
\end{aligned}$$

$$
\begin{aligned}
&= \quad \{\text{ proof obligation }\}\\
&\quad f\\
&= \quad \{\text{ definition of } \mathscr{I} \}\\
&\quad \mathscr{I}[\![\mathit{Abs}\ e]\!]\eta\,.
\end{aligned}
$$

The proof of the obligation relies on the assumption that the idiom is strongly extensional and hence extensional.

$$
\begin{aligned}
&\quad \mathit{pure}\ (\mathit{curry}\ [\![e]\!]) \diamond \mathit{dist}\ \eta = f\\
&\iff \quad \{\text{ extensionality }\}\\
&\quad \mathit{pure}\ (\mathit{curry}\ [\![e]\!]) \diamond \mathit{dist}\ \eta \diamond v = f \diamond v\\
&\iff \quad \{\text{ definition of } f\text{: the unique element such that } \forall u\ .\ f \diamond u = \mathscr{I}[\![e]\!]\langle \eta, u\rangle \}\\
&\quad \mathit{pure}\ (\mathit{curry}\ [\![e]\!]) \diamond \mathit{dist}\ \eta \diamond u = \mathscr{I}[\![e]\!]\langle \eta, u\rangle\\
&\iff \quad \{\text{ currying, see Section 2 }\}\\
&\quad \mathit{pure}\ [\![e]\!] \diamond (\mathit{dist}\ \eta \star u) = \mathscr{I}[\![e]\!]\langle \eta, u\rangle\\
&\iff \quad \{\text{ definition of } \mathit{dist} \}\\
&\quad \mathit{pure}\ [\![e]\!] \diamond (\mathit{dist}\ \langle \eta, u\rangle) = \mathscr{I}[\![e]\!]\langle \eta, u\rangle
\end{aligned}
$$

The last equation holds ex hypothesi. □

**Corollary 1** *Let $\iota$ be a strongly extensional idiom with K-combinators, and let $e :: \mathit{Term}\ ()\ \tau$ be a closed lambda-term. Then*

$$\mathscr{I}[\![e]\!] = \mathit{pure}\ [\![e]\!].$$

*Proof*

$$
\begin{aligned}
&\quad \mathscr{I}[\![e]\!]\\
&= \quad \{\text{ definition of } \mathscr{I} \}\\
&\quad \mathscr{I}[\![e]\!]\langle\rangle\\
&= \quad \{\text{ Lifting Lemma }\}\\
&\quad \mathit{pure}\ [\![e]\!] \diamond \mathit{dist}\ \langle\rangle\\
&= \quad \{\text{ definition of } \mathit{dist} \}\\
&\quad \mathit{pure}\ [\![e]\!] \diamond \mathit{pure}\ \langle\rangle\\
&= \quad \{\text{ homomorphism }\}\\
&\quad \mathit{pure}\ ([\![e]\!]\ \langle\rangle)\\
&= \quad \{\text{ definition of } [\![\,]\!] \}\\
&\quad \mathit{pure}\ [\![e]\!]
\end{aligned}
$$

□

So, the generic proof of lifted distributivity (and other base-level identities) requires the idiom to be strongly extensional. We can fore-go this requirement if we assume instead that $k = \mathit{pure}\ \mathbb{K}$ and $s = \mathit{pure}\ \mathbb{S}$. We restrict ourselves to idiomatic terms (no abstraction) and additionally appeal to the Abstraction Lemma. Here is the first part of the proof modified accordingly:

$$
\begin{aligned}
& \mathit{pure}\ (*) \diamond (\mathit{pure}\ (+) \diamond u \diamond v) \diamond w \\
= \quad & \{\ \text{definition of } \mathscr{I}\ \} \\
& \mathscr{I}[\![\mathit{Con}(*) \mathbin{:\diamond} (\mathit{Con}(+) \mathbin{:\diamond} 2 \mathbin{:\diamond} 1) \mathbin{:\diamond} 0]\!]\langle\langle\langle\langle\rangle, u\rangle, v\rangle, w\rangle \\
= \quad & \{\ \text{Abstraction Lemma, thrice}\ \} \\
& \mathscr{I}[\![\mathit{abs}(\mathit{abs}(\mathit{abs}(\mathit{Con}(*) \mathbin{:\diamond} (\mathit{Con}(+) \mathbin{:\diamond} 2 \mathbin{:\diamond} 1) \mathbin{:\diamond} 0)))]\!] \diamond u \diamond v \diamond w \\
= \quad & \{\ \text{Lifting Lemma restricted to idiomatic terms}\ \} \\
& \mathit{pure}\ [\![\mathit{abs}(\mathit{abs}(\mathit{abs}(\mathit{Con}(*) \mathbin{:\diamond} (\mathit{Con}(+) \mathbin{:\diamond} 2 \mathbin{:\diamond} 1) \mathbin{:\diamond} 0)))]\!] \diamond u \diamond v \diamond w \\
= \quad & \{\ \text{property of } [\![\,]\!]\ \} \\
& \mathit{pure}\ (\lambda x\ y\ z \to (x + y) * z) \diamond u \diamond v \diamond w.
\end{aligned}
$$

Since the base-level identity is a first-order equation, the proof nicely goes through.

## 6 Idiom homomorphisms

In the previous section we have seen that strongly extensional idioms with K-combinators satisfy every lifted base-level identity. But which idioms actually fall into this category? On the negative side, the set idiom, pointed types and parser idioms have no $k$ combinator so the Lifting Lemma is not applicable — we explore a possible remedy in the next section. On the positive side, the environment idiom has all the required bits and pieces. But what about streams and infinite trees? It turns out that these structures also satisfy the requirements. In the rest of this section we explain how to prove this fact.

To start with, here is the definition of streams and the instance declaration that turns the datatype *Stream* into an idiom:

```
data Stream α = Cons {head :: α, tail :: Stream α}
instance Idiom Stream where
  pure a = s where s = Cons a s
  u ◇ v  = Cons ((head u) (head v)) ((tail u) ◇ (tail v)).
```

Streams are infinite sequences of elements. The type *Stream* $\alpha$ is like Haskell's list datatype $[\alpha]$, except that there is no base constructor so we cannot construct a finite stream.

The crucial insight is that streams are in a one-to-one correspondence to functions from the natural numbers: *Stream* $\alpha \cong \mathit{Nat} \to \alpha$ (Hinze 2000; Altenkirch 2001). A stream can be seen as the tabulation of a function from the natural numbers:

```
tabulate   :: (Nat → α) → Stream α
tabulate f = Cons (f 0) (tabulate (f · (+1))).
```

Conversely, a function of type $\mathit{Nat} \to \alpha$ can be implemented by looking up a memo-table:

```
lookup           :: Stream α → (Nat → α)
lookup s 0       = head s
lookup s (n + 1) = lookup (tail s) n.
```

The natural transformations *tabulate* and *lookup* are not only mutually inverse, they also preserve the idiomatic structure.

**Definition 3** Let $\iota$ and $\kappa$ be idioms. A natural transformation $h :: \iota\ \alpha \to \kappa\ \alpha$ is an *idiom homomorphism* iff $h$ preserves pure computations and idiomatic application, that is,

$$h\ (pure\ a) = pure\ a \tag{4}$$
$$h\ (u \diamond v) = h\ u \diamond h\ v, \tag{5}$$

for all $a$, $u$, $v$ of the appropriate types.

The function $pure :: \alpha \to \iota\ \alpha$ itself is a homomorphism from the identity idiom $Id$ to the idiom $\iota$. Condition 5 for $pure$ is equivalent to the homomorphism law (hence its name). Tabulation is an idiom isomorphism from $Nat \to$ to $Stream$, with look-up as its inverse.

**Lemma 4** *Let $\iota$ and $\kappa$ be idioms, and let $h :: \iota\ \alpha \to \kappa\ \alpha$ be an idiom isomorphism.*

1. *If $\iota$ is (strongly) extensional, then $\kappa$ is (strongly) extensional.*
2. *If $\iota$ has K-combinators, then $\kappa$ has K-combinators.*

*Proof* Straightforward. □

Since the $Stream$ idiom inherits the properties from the environment idiom, it is strongly extensional and it has K-combinators. An analogous argument applies to the type of infinite binary trees

$$\textbf{data}\ Tree\ \alpha = Node\ \{root :: \alpha, left :: Tree\ \alpha, right :: Tree\ \alpha\},$$

which is isomorphic to the environment idiom $[Bit] \to$, where $Bit$ is a two-element type.

## 7 The $\lambda\iota$I-calculus

Several idioms have zero elements, $zero \diamond v = zero = u \diamond zero$, which precludes the existence of the combinator $k$. In the $Maybe$ idiom ($\tau$ *option* in Standard ML), for instance,

$$
\begin{array}{l}
\textbf{data}\ Maybe\ \alpha = Nothing \mid Just\ \alpha \\
\textbf{instance}\ Idiom\ Maybe\ \textbf{where} \\
\quad pure\ a = Just\ a \\
\quad Nothing \diamond v \qquad\qquad = Nothing \\
\quad Just\ f \quad \diamond Nothing = Nothing \\
\quad Just\ f \quad \diamond Just\ a \quad = Just\ (f\ a),
\end{array}
$$

the 'added point' $Nothing$ is the zero element. The existence of $k$ is vital to prove, for instance, the lifted version of $x * 0 = 0$. However, for many other identities we can do without. As an example, consider the proof of lifted distributivity in Section 5. The crucial step in the calculation is the application of

$$\lambda x\ y\ z \to (x+y) * z = \lambda x\ y\ z \to x * z + y * z.$$

Note that the bound variables $x$, $y$ and $z$ appear in both bodies; the two abstractions are so-called $\lambda$I-terms (Church 1941). The $\lambda$I-calculus, Church's original version of the lambda-calculus, rules out abstractions where the bound variable does not appear in the body. Consequently, $\mathbb{K} = \lambda x\ y \to x$ is not a legal $\lambda$I-term. In fact, the essential difference between $\lambda$K-terms and $\lambda$I-terms is the combinator $\mathbb{K}$: all $\lambda$K-terms are definable from $\mathbb{K}$ and $\lambda$I-terms (Barendregt 1984).

We have seen that abstraction in the $\lambda\iota$K-calculus can be simulated with the combinators $k$ and $s$. For $\lambda$I-abstraction a 'less demanding' set of combinators is required.

**Definition 4 (Weak combinatory model condition)** An idiom $\iota$ has I-combinators iff there exist elements $i :: \iota\ (\alpha \to \alpha)$, $b :: \iota\ ((\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma))$, $c :: \iota\ ((\alpha \to \beta \to \gamma) \to \beta \to (\alpha \to \gamma)$ and $s :: \iota\ ((\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma))$ satisfying

$$i \diamond u = u \qquad (6)$$

$$b \diamond u \diamond v \diamond w = u \diamond (v \diamond w) \qquad (7)$$

$$c \diamond u \diamond v \diamond w = (u \diamond w) \diamond v \qquad (8)$$

$$s \diamond u \diamond v \diamond w = (u \diamond w) \diamond (v \diamond w), \qquad (9)$$

for all $u$, $v$, $w$ of the appropriate types.

In the *Id* idiom, $i$ is the identity *id*, $b$ is composition '$\cdot$' and $c$ is Haskell's *flip* operator.

We can now replay the development of Sections 4 and 5 for the $\lambda\iota$I-calculus. We confine ourselves to adapting the abstraction operator:

$$\begin{array}{l}
abs' :: Term\ (\rho, \alpha)\ \beta \to Term\ \rho\ (\alpha \to \beta) \\
abs'\ (Var\ Zero) = I \\
abs'\ (e_1 \mathbin{:\!\diamond} e_2) \\
\quad \mid \neg free\ e_1 \wedge free\ e_2 \quad = B \mathbin{:\!\diamond} dec\ e_1 \mathbin{:\!\diamond} abs'\ e_2 \\
\quad \mid free\ e_1 \quad \wedge \neg free\ e_2 = C \mathbin{:\!\diamond} abs'\ e_1 \mathbin{:\!\diamond} dec\ e_2 \\
\quad \mid free\ e_1 \quad \wedge free\ e_2 \quad = S \mathbin{:\!\diamond} abs'\ e_1 \mathbin{:\!\diamond} abs'\ e_2.
\end{array}$$

The function $abs'$ requires and maintains the invariant that the variable *Zero* occurs free in its argument. The Boolean function *free e* implements the check $Zero \in FV(e)$,

$$\begin{array}{ll}
free & :: Term\ \rho\ \alpha \to Bool \\
free\ (Con\ u) & = False \\
free\ (Var\ Zero) & = True \\
free\ (Var\ (Succ\ n)) & = False \\
free\ (e_1 \mathbin{:\!\diamond} e_2) & = free\ e_1 \vee free\ e_2,
\end{array}$$

and the function *dec* adjusts an idiomatic term that does not contain *Zero*:

$$\begin{array}{ll}
dec & :: Term\ (\rho, \alpha)\ \beta \to Term\ \rho\ \beta \\
dec\ (Con\ u) & = Con\ u \\
dec\ (Var\ (Succ\ n)) & = Var\ n \\
dec\ (e_1 \mathbin{:\!\diamond} e_2) & = dec\ e_1 \mathbin{:\!\diamond} dec\ e_2.
\end{array}$$

Every idiom possesses two of the four combinators: $i = pure\ \mathbb{I} = pure\ id$ and $b = pure\ \mathbb{B} = pure\ (\cdot)$. Condition (6) and (7) are idiom identity and idiom composition in disguise. Consequently, to establish the weak combinatory model condition it suffices to show the existence of $c$ and $s$. Furthermore, if a strongly extensional idiom has I-combinators, then $c = pure\ \mathbb{C}$ and $s = pure\ \mathbb{S}$. For instance, the *Maybe* idiom has I-combinators. On the other hand, stateful idioms such as *IO* and parser idioms don't possess I-combinators as we can't re-order or duplicate stateful computations.

The set and the list idiom have no $s$ combinator as a simple cardinality argument shows: for lists $|u \diamond v| = |u| * |v|$, but there is no $s$ such that $|s| * |u| * |v| * |w| = |u| * |v| * |w|^2$. However, the set idiom possesses a $c$ combinator. In other words, there is also a need for a linear version of the Lifting Lemma. We leave the details to the reader.

Table 1 summarises our findings. The environment idiom and consequently *Stream* and *Tree* satisfy every lifted base-level identity. For the *Maybe* idiom we have to make sure that every variable appears on both sides of the equation, and for the set idiom we have to additionally ensure that there are no repeated variables. For the remaining idioms only the Lifting Lemma is applicable.

| | ext. | str. ext. | $k$ | $s$ | $c$ |
|---|---|---|---|---|---|
| $\tau \rightarrow$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ |
| *Stream* | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ |
| *Tree* | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ |
| *Maybe* | $\surd$ | $\surd$ | no | $\surd$ | $\surd$ |
| *Set* | no | no | no | no | $\surd$ |
| [] | $\surd$ | $\surd$ | no | no | no |
| *State* | $\surd$ | $\surd$ | no | no | no |
| *Parser* | $\surd$ | $\surd$ | no | no | no |
| *IO* | ? | ? | no | no | no |

**Table 1** Properties of various idioms.

## 8 Related and future work

Almost every textbook on infinite calculus introduces lifted sums and products,

$$\begin{aligned}(f+g)(x) &= f(x)+g(x)\\ (f\cdot g)(x) &= f(x)\cdot g(x),\end{aligned}$$

so that the differentiation rules can be written down succinctly and attractively:

$$\begin{aligned}(f+g)' &= f'+g'\\ (f\cdot g)' &= f'\cdot g+f\cdot g'.\end{aligned}$$

(The textbook I resorted to during my studies (Dörfler and Peschek 1988) didn't quite trust its own conventions and defined somewhat half-heartedly $(f+g)'(x_0)=f'(x_0)+g'(x_0)$.) A formal treatment of lifting is almost always missing. In particular, it goes unnoticed that the lifted operators inherit the properties of the base-level operators. To the best of the author's knowledge this was first enunciated by Backhouse (1989). He writes

> "A large number of examples — I have yet to see any proof — suggest that $\langle\ldots\rangle$, given any valid judgement $\langle\ldots\rangle$, any lifted form of the judgement $\langle\ldots\rangle$ is also valid".

The Lifting Lemma and its variants generalise and close this case.

The fact that every idiomatic expression can be rewritten into the form $\mathit{pure}\,f \diamond u_1 \diamond \cdots \diamond u_n$, a pure function applied to impure arguments, has already been noted by McBride and Paterson (2008) in their seminal paper on applicative functors.

Applicative functors are closely related to typed applicative structures (Mitchell 1996), which are used to give a semantics to the simply typed lambda-calculus. Very briefly, an applicative structure consists of a family of carrier sets $A^{\sigma}$, a type-indexed family of mappings $\mathit{Const}^{\sigma}$ from term constants of type $\sigma$ to elements of $A^{\sigma}$, and a type-indexed family of mappings $\mathit{App}^{\sigma,\tau}{:}A^{\sigma\rightarrow\tau} \rightarrow (A^{\sigma} \rightarrow A^{\tau})$. An extensional structure that has "enough elements" is then a model, a so-called environment model or Henkin model. In a sense, idioms are a syntax-free variant of typed applicative structures, replacing the type-indexed set by a functor and the type-indexed mappings by natural transformations. Because of the close correspondence, it was straightforward to adapt the notion of an environment model to our setting.

A couple of problems are left for future work. In particular,

– Is every extensional idiom also strongly extensional?
– If this is not the case, are the combinators always pure embeddings ($k = \mathit{pure}\,\mathbb{K}$)?

**Acknowledgements** A big thank you to Daniel James for improving my English and for his heroic effort to formalise the Lifting Lemma in Coq.

## A Standard combinators

$$
\begin{aligned}
&\mathit{id}\;x &&= x\\
&(f \cdot g)\;x &&= f\;(g\;x)\\
&\mathit{const}\;x\;y &&= x\\
&\mathit{flip}\;f\;x\;y &&= f\;y\;x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fst}\;(x,y) &&= x\\
&\mathit{snd}\;(x,y) &&= y\\
&(f \vartriangle g)\;x &&= (f\;x, g\;x)\\
&(f \times g)\;(x,y) &&= (f\;x, g\;y)\\
&\mathit{assocl}\;(x,(y,z)) &&= ((x,y),z)\\
&\mathit{assocr}\;((x,y),z) &&= (x,(y,z))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{app}\;(f,x) &&= f\;x\\
&\mathit{curry}\;f\;x\;y &&= f\;(x,y)
\end{aligned}
$$

$$
\begin{aligned}
&\mathbb{I}\;x &&= x\\
&\mathbb{K}\;x\;y &&= x\\
&\mathbb{S}\;x\;y\;z &&= (x\;z)\;(y\;z)\\
&\mathbb{B}\;x\;y\;z &&= x\;(y\;z)\\
&\mathbb{C}\;x\;y\;z &&= (x\;z)\;y
\end{aligned}
$$

## References

Thorsten Altenkirch. Representations of first order function types as terminal coalgebras. In *Typed Lambda Calculi and Applications, TLCA 2001*, volume 2044 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 2001.

Roland Backhouse. Making formality work for us. *EATCS Bulletin*, 38:219–249, June 1989.

H. P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics*. North-Holland, Amsterdam New York Oxford, revised edition, 1984.

Alonzo Church. The calculi of lambda-conversion. Annals of Mathematics Studies No. 6, Princeton University Press, 1941.

H.B. Curry and R. Feys. *Combinatory Logic, Volume 1*. North-Holland, Amsterdam New York Oxford, 1958.

Willibald Dörfler and Werner Peschek. *Einführung in die Mathematik für Informatiker*. Hanser Verlag, 1988.

Ralf Hinze. Memo functions, polytypically! In Johan Jeuring, editor, *Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal*, pages 17–32, July 2000. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.

Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.

Ralf Hinze. Functional Pearl: Streams and unique fixed points. In Peter Thiemann, editor, *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*, pages 189–200. ACM Press, September 2008.

Ralf Hinze. Functional Pearl: The Bird tree. *J. Functional Programming*, 19(5):491–508, September 2009a.

Ralf Hinze. Scans and convolutions—a calculational proof of Moessner's theorem. In Sven-Bodo Scholz, editor, *Post-proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008), University of Hertfordshire, UK, September 10–12, 2008*, volume 5836 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009b.

Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

Xavier Leroy. Applicative functors and fully transparent higher-order modules. In Ron K. Cytron and Peter Lee, editors, *Proceedings of the22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), San Francisco, California, January 23–25*, pages 142–153. ACM New York, NY, USA, January 1995.

Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer-Verlag, Berlin, 2nd edition, 1998.

Conor McBride and Ross Paterson. Functional Pearl: Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, Cambridge, MA, 1996.

Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In Julia Lawall, editor, *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming, Portland, Oregon, USA, September 18-20, 2006*, pages 50–61. ACM Press, September 2006.

Niklas Röjemo. *Garbage collection, and memory efficiency, in lazy functional languages*. PhD thesis, Chalmers University of Technology, 1995.