

Inductive Reasoning about Effectful Data Types

Andrzej Filinski

DIKU, University of Copenhagen, Denmark
andrzej@diku.dk

Kristian Støvring

DAIMI, University of Aarhus, Denmark
kss@daimi.au.dk

Abstract

We present a pair of reasoning principles, *definition* and *proof* by *rigid induction*, which can be seen as proper generalizations of lazy-datatype induction to monadic effects other than partiality. We further show how these principles can be integrated into logical-relations arguments, and obtain as a particular instance a general and principled proof that the success-stream and failure-continuation models of backtracking are equivalent. As another application, we present a monadic model of general search trees, not necessarily traversed depth-first. The results are applicable to both lazy and eager languages, and we emphasize this by presenting most examples in both Haskell and SML.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (functional) programming; F.3.2 [Semantics of Programming Languages]: Denotational semantics

General Terms Languages, Theory

Keywords monads, recursive types, equational reasoning, logical relations, abstract effects, streams, backtracking

1. Introduction

Consider the SML type of lists:

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

Although this is an instance of a general recursive data type, we frequently think of it as an inductive, set-theoretic definition. We can then define functions on lists by structural induction:

```
fun append Nil l = l
  | append (Cons (h,t)) l = Cons h (append t l)

fun rev Nil l = l
  | rev (Cons (h,t)) l = rev t (Cons (h,l))
fun reverse l = rev l Nil
```

We can also reason about such functions by structural induction on their arguments. For example, to verify that `append` is associative, i.e., that for all list-typed values l_1, l_2 , and l_3 ,

```
append (append l1 l2) l3 = append l1 (append l2 l3),
```

we use the clauses of the function definition as valid equational axioms, and proceed by induction over l_1 , checking the equation

for $l_1 = \text{Nil}$ and $l_1 = \text{Cons } (h, t)$ with the induction hypothesis that the equation holds for t .

Similarly, we can prove that `reverse` is an involution. We start by proving, by induction on l_1 , that

$$\text{reverse } (\text{rev } l_1 l_2) = \text{rev } l_2 l_1 \quad (1)$$

from which `reverse (reverse l) = reverse (rev l Nil) = rev Nil l = l`.

Consider now the analogous definitions in Haskell, where lists can be partial and/or infinite. The principle of “lazy-list induction” (Reade 1989, Section 8.5) says that that when reasoning about lazy lists, one must consider, in addition to the proper constructors, the case of an undefined value, usually written Ω or \perp .

For associativity of `append`, the extra case $l_1 = \perp$ goes through easily, with both sides equal to \perp ; but for Equation (1) with $l_1 = \perp$ and $l_2 = \text{Cons } 0 \text{ Nil}$, we get

$$\begin{aligned} \text{reverse } (\text{rev } \perp (\text{Cons } 0 \text{ Nil})) &= \text{reverse } \perp = \perp \\ &\neq \text{Cons } 0 \perp = \text{rev } (\text{Cons } 0 \text{ Nil}) \perp \end{aligned}$$

which is just as well, since `reverse` is evidently not an involution on infinite lists.

The problem with infinite lists is actually a bit subtler than just checking the \perp -case: For example, the predicate

$$P(l) \Leftrightarrow \exists n :: \text{Int. take } n l = l$$

holds for all finite (even \perp -terminated) lists, but still not for infinite ones. Domain-theoretically, the problem is that this predicate is not *chain-complete*: knowing that it holds for all elements in an ascending chain does not guarantee that it holds of the supremum of the chain. We say that a predicate P is *admissible* if $P(\perp)$ and P is chain-complete. We then have

Theorem 1.1 (informal). *Let P be an admissible predicate on the type $\text{List } \tau$. If $P(\text{Nil})$, and $\forall h, t. P(t) \Rightarrow P(\text{Cons } h t)$, then $\forall l. P(l)$.*

(Note that the condition $P(\perp)$ is part of the definition of admissibility, rather than another pseudo-constructor case; this view will prove useful later, when we consider effects other than divergence.)

Fortunately, we rarely need to verify admissibility by explicit appeal to its definition. It is fairly easy to see that if terms e_1 and e_2 with free variables x, y_1, \dots, y_n are strict in x , then the predicate

$$P(x) \Leftrightarrow \forall y_1, \dots, y_n. e_1 = e_2$$

is admissible. This codifies three important properties of admissible predicates: (1) equality (i.e., the diagonal predicate on $\tau \times \tau$) is admissible; (2) inverse images by strict functions of admissible predicates are admissible; and (3) arbitrary intersections (conjunctions) of admissible predicates are admissible.

Further, an easy way to see that a function is strict in an argument is when it induces on that argument. Indeed, the problem with (1) was that the induction on the right-hand side was not on l_1 .

Consider finally the generalization to streams, where producing the next stream element (Cons or Nil) may involve arbitrary computation, including but not limited to I/O. We can still easily define an append function for such streams, but can we be sure it remains associative, even in the presence of effects embedded in the streams? Equally importantly, can we (once the proper framework is in place) show it with effort comparable to proving the original append associative, i.e., merely verify the two constructor cases, and check that the associativity equation is built up in a way that admits the induction principle?

In the remainder of the paper we present such a framework, based on a category-theoretical generalization of strictness: the notion of *monad-algebra morphisms*, which we call *rigid functions*. Strict functions are then precisely those that are rigid with respect to the lifting monad. We show how an associated induction principle allows us to both *define* rigid functions on streams and similar data types, and *prove* properties about them. We consider both the domain-theoretic foundations, and how they are reflected in reasoning about ML and Haskell programs.

As an application of the setup, we show how to relate several models of backtracking search, based on solution streams, success/failure continuations, and decision trees. In particular, we generalize the main result of Wand and Vaillancourt (2004), which relates the former two models in the special case where the underlying effect is just partiality.

2. Equational reasoning

2.1 Domain-theoretic foundations

We develop the basic results in the setting of the category Cpo of ω -cpo's and (total) continuous functions. Let us briefly recall some standard definitions:

Definition 2.1. A *monad* is a triple $\bar{T} = (T, \eta, \star)$, where T maps cpo's to cpo's, and $\eta : A \rightarrow TA$ and $\star : TA \times (A \rightarrow TB) \rightarrow TB$ are function families satisfying

$$\eta a \star f = f a \quad (2)$$

$$t \star \eta = t \quad (3)$$

$$(t \star f) \star g = t \star (\lambda f. f a \star g). \quad (4)$$

An alternative presentation defines a monad as a triple (T, η, μ) where T is an endofunctor, and η and μ are natural transformations satisfying a few additional equations. The two formulations are equivalent: $T(f) = \lambda t. t \star (\eta \circ f)$, $\mu = \lambda t. t \star id$; and conversely, $t \star f = \mu \circ T(f)$. The *lifting monad* takes $T^\perp A = A_\perp$, $\eta^\perp a = \lfloor a \rfloor$, $\perp \star^\perp f = \perp$, and $\lfloor a \rfloor \star^\perp f = f(a)$.

Definition 2.2. A *monad morphism* from (T, η, \star) to (T', η', \star') is a family of functions $\iota : TA \rightarrow T'A$, satisfying

$$\iota(\eta a) = \eta' a \quad (5)$$

$$\iota(t \star f) = \iota t \star' (\iota \circ f). \quad (6)$$

Monad morphisms can be used to model *inclusion* or *lifting* of behaviors from one monad to a more general one.

Definition 2.3. An *algebra* for a functor F is a pair (X, γ) , where $\gamma : FX \rightarrow X$ is called the *structure* of the algebra. An algebra for a monad (T, η, \star) must also satisfy a few equations, relating γ to η and \star . For readability, it is often convenient to say instead that a monad algebra is a pair $\tilde{D} = (D, \otimes)$, where the function family $\otimes : TA \times (A \rightarrow D) \rightarrow D$ satisfies equations similar to those for \star :

$$\eta a \otimes f = f a \quad (7)$$

$$(t \star f) \otimes g = t \otimes (\lambda a. f a \otimes g). \quad (8)$$

Again the formulations are equivalent: we can take $\gamma(t) = t \otimes id_D$, and conversely $t \otimes f = \gamma(t \star (\lambda a. \eta(f a))) = \gamma(T(f)(t))$.

A cpo D is *pointed* (i.e., has a least element \perp_D) precisely when it can be extended to an algebra for the lifting monad, by a structure $\gamma : D_\perp \rightarrow D$. One easily sees that this γ must be unique if it exists at all. In the alternative formulation, \otimes determines the *strict extension* of a function $f : A \rightarrow D$ to $\lambda l. l \otimes f : A_\perp \rightarrow D$.

Proposition 2.4. *There are canonical ways to construct algebras for a monad (T, η, \star) :*

- For any cpo A , (TA, \star) is the free algebra on A .
- For algebras $\tilde{D}_1 = (D_1, \otimes_1)$ and $\tilde{D}_2 = (D_2, \otimes_2)$ we form the product algebra $\tilde{D}_1 \tilde{\times} \tilde{D}_2 = (D_1 \times D_2, \otimes_\times)$ where $t \otimes_\times f = (t \otimes_1 (\pi_1 \circ f), t \otimes_2 (\pi_2 \circ f))$.
- For any cpo B and algebra $\tilde{D} = (D, \otimes)$, we form the function algebra $B \tilde{\rightarrow} \tilde{D} = ([B \rightarrow D], \otimes_\rightarrow)$, where $t \otimes_\rightarrow f = \lambda b. t \otimes (\lambda a. f a b)$.

Definition 2.5. An *algebra morphism* or *rigid function* between \bar{T} -algebras (D, \otimes) and (D', \otimes') is a function $h : D \rightarrow D'$ satisfying that, for all cpo's A , $t \in TA$, and $f : A \rightarrow D$,

$$h(t \otimes f) = t \otimes' (h \circ f). \quad (9)$$

We write $h : (D, \otimes) \rightarrow (D', \otimes')$ when this is the case.

When \bar{T} is lifting, a rigid function is precisely one that is strict, i.e. satisfies that $h(\perp_D) = \perp_{D'}$. Such functions may be moved inside an evaluation-forcing construct. In general, rigid functions can be thought of as those that perform their arguments' computations before any other.

Proposition 2.6. *The following are valid principles for constructing rigid functions:*

1. $id : \tilde{D} \rightarrow \tilde{D}$; and if $f : \tilde{D}_1 \rightarrow \tilde{D}_2$ and $g : \tilde{D}_2 \rightarrow \tilde{D}_3$, then $g \circ f : \tilde{D}_1 \rightarrow \tilde{D}_3$.
2. For any (D, \otimes) and $f : A \rightarrow D$, $\lambda t. t \otimes f : (TA, \star) \rightarrow (D, \otimes)$.
3. $\pi_i : \tilde{D}_1 \tilde{\times} \tilde{D}_2 \rightarrow \tilde{D}_i$; and if each $f_i : \tilde{D} \rightarrow \tilde{D}_i$, then $\lambda d. (f_1(d), f_2(d)) : \tilde{D} \rightarrow \tilde{D}_1 \tilde{\times} \tilde{D}_2$.
4. For each $b \in B$, $\lambda g. g b : (B \tilde{\rightarrow} \tilde{D}) \rightarrow \tilde{D}$; and if $f : D' \rightarrow (B \rightarrow D)$ satisfies that for each $b \in B$, $\lambda d'. f(d') b : \tilde{D}' \rightarrow \tilde{D}$, then $f : \tilde{D}' \rightarrow (B \tilde{\rightarrow} \tilde{D})$.

We also say that a function of multiple arguments is rigid in one of them, when it is rigid (in the original sense) for all values of the other arguments. (A function can be rigid in more than one of its arguments, if \bar{T} is a commutative monad, such as lifting, read-only state, or complexity.) By Prop. 2.6(4), this is equivalent to requiring that the corresponding carried function into the function algebra is rigid.

Depending on the specific monad \bar{T} , there may be additional principles for constructing rigid functions. For example, in a monad modeling idempotent effects (such as divergence, $TA = A_\perp$), the computation-duplicating function $\lambda t. t \star \lambda a. t \star \lambda a'. \eta(a, a') : TA \rightarrow T(A \times A)$ is rigid; and in one modeling discardable effects (such as read-only state, $TA = S \rightarrow A$), the function $\lambda t. t \star \lambda a. \eta() : TA \rightarrow T1$ is rigid. The more functions that are rigid, the stronger the equational reasoning principles we shall obtain in the following.

Finally, there is a close relationship between monad algebras and monad morphisms:

Proposition 2.7. *Let $\bar{T} = (T, \eta, \star)$ and $\bar{T}' = (T', \eta', \star')$ be monads. If for every A , (TA, \otimes'_A) is a \bar{T}' -algebra, and for every $f : A \rightarrow TB$, $\lambda t. t \star f : (TA, \otimes'_A) \rightarrow (TB, \otimes'_B)$, then $\iota = \lambda t'. t' \otimes'_A \eta : T'A \rightarrow TA$ is a monad morphism. In this situation, we say that the monad \bar{T} is layered over \bar{T}' .*

Proof. Straightforward calculation, using the monad-morphism and monad-algebra laws. \square

2.2 Reasoning about programs

2.2.1 Metalanguage

To get a coherent account of reasoning about both SML and Haskell programs with computational effects, it is convenient to conceptually express both languages in terms of a common computational metalanguage. In this way, we abstract from concrete syntax, as well as from the differences between lazy/strict evaluation, classes/modules, and monad-parameterization/effect-refinement.

Our metalanguage, a slightly simplified variant of the Multi-Monadic MetaLanguage M^3L (Filinski 2007), is essentially the monadic metalanguage of Moggi (1991), extended with the FPC-style general recursive types (Fiore and Plotkin 1994), and a little syntactic support for talking about algebras for a collection of monads. Its types are parameterized over a set of *effect names* e :

$$\begin{aligned}\tau &::= 1 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \alpha \mid \sigma \\ \sigma &::= \mathbf{T}^e\tau \mid \tau \rightarrow \sigma \mid \sigma_1 \times \sigma_2\end{aligned}$$

σ -types are called *computational*, and will denote monad algebras. Note that most conventional base types are definable in the metalanguage, e.g., $\text{Nat} \equiv \mu\alpha.1 + \alpha$.

The terms are parameterized over a set of potentially polymorphic constants $c_{\bar{\alpha}}$, which must be type-instantiated at each use:

$$\begin{aligned}M &::= x \mid c_{\bar{\tau}} \mid () \mid (M_1, M_2) \mid \mathbf{fst}(M) \mid \mathbf{snd}(M) \mid \mathbf{inl}(M) \\ &\mid \mathbf{inr}(M) \mid \mathbf{case}(M, x_1.M_1, x_2.M_2) \mid \mathbf{in}_{\mu\alpha.\tau} M \\ &\mid \mathbf{out}_{\mu\alpha.\tau} M \mid \lambda x^\tau.M \mid M_1 M_2 \mid \mu x^\sigma.M \mid \mathbf{val}^e M \\ &\mid \mathbf{let}^e x \leftarrow M_1.M_2\end{aligned}$$

The typing relation, $\Gamma \vdash M : \tau$, is defined in the obvious way. Note that term-level recursion, $\mu x^\sigma.M$, is only allowed at σ -types. We will occasionally omit the type tags, and use a straightforward pattern-matching notation for **fst** and **snd**.

We write $e \preceq \sigma$ when σ is a product or function space of \mathbf{T}^e -types, in which case we can define a *generalized let*, $\mathbf{glet}_{\sigma}^e x \leftarrow M_1.M_2$, allowing $M_2 : \sigma$, by induction on σ :

$$\begin{aligned}\mathbf{glet}_{\mathbf{T}^e\tau}^e x \leftarrow M_1.M_2 &= \mathbf{let}^e x \leftarrow M_1.M_2 \\ \mathbf{glet}_{\tau \rightarrow \sigma}^e x \leftarrow M_1.M_2 &= \lambda a^\tau. \mathbf{glet}_{\sigma}^e x \leftarrow M_1.M_2 a \\ \mathbf{glet}_{\sigma_1 \times \sigma_2}^e x \leftarrow M_1.M_2 &= \\ (\mathbf{glet}_{\sigma_1}^e x \leftarrow M_1. \mathbf{fst}(M_2), \mathbf{glet}_{\sigma_2}^e x \leftarrow M_1. \mathbf{snd}(M_2))\end{aligned}$$

This corresponds to the \otimes -formulation of a monad algebra; equivalently, we can define a term family $\mathbf{glue}_{\sigma}^e : \mathbf{T}^e\sigma \rightarrow \sigma$ corresponding to the γ . As noted in Def. 2.3, they are interdefinable:

$$\begin{aligned}\mathbf{glet}_{\sigma}^e x \leftarrow M_1.M_2 &= \mathbf{glue}_{\sigma}^e (\mathbf{let}^e x \leftarrow M_1. \mathbf{val}^e M_2) \\ \mathbf{glue}_{\sigma}^e &= \lambda t^{\mathbf{T}^e\sigma}. \mathbf{glet}_{\sigma}^e x \leftarrow t.x\end{aligned}$$

For the denotational semantics of types, let θ map type variables in $\Delta = \{\alpha_1, \dots, \alpha_n\}$ to cpos, and for every effect name e , let $\bar{T}^e = (T^e, \eta^e, \star^e)$ be a monad layered over lifting. Then to every τ with free type variables in Δ , we associate a cpo $\llbracket \tau \rrbracket_{\theta}$; and to every σ with $e \preceq \sigma$, a \bar{T}^e -algebra $\llbracket \sigma \rrbracket_{\theta}^{\bar{a}}$:

$$\begin{aligned}\llbracket \alpha \rrbracket_{\theta} &= \theta(\alpha) \\ \llbracket 1 \rrbracket_{\theta} &= 1 = \{()\} \\ \llbracket \tau_1 \times \tau_2 \rrbracket_{\theta} &= \llbracket \tau_1 \rrbracket_{\theta} \times \llbracket \tau_2 \rrbracket_{\theta} \\ \llbracket \tau_1 + \tau_2 \rrbracket_{\theta} &= \llbracket \tau_1 \rrbracket_{\theta} + \llbracket \tau_2 \rrbracket_{\theta} \\ \llbracket \sigma \rrbracket_{\theta} &= D \text{ where } (D, \otimes) = \llbracket \sigma \rrbracket_{\theta}^{\bar{a}} \\ \llbracket \mu\alpha.\tau \rrbracket_{\theta} &= \mu X. \llbracket \tau \rrbracket_{\theta[\alpha \mapsto X]} = \{\phi_{\alpha.\tau} m \mid m \in \llbracket \tau[\mu\alpha.\tau/\alpha] \rrbracket_{\theta}\} \\ \llbracket \mathbf{T}^e\tau \rrbracket_{\theta}^{\bar{a}} &= (T^e \llbracket \tau \rrbracket_{\theta}, \star^e) \\ \llbracket \sigma_1 \times \sigma_2 \rrbracket_{\theta}^{\bar{a}} &= \llbracket \sigma_1 \rrbracket_{\theta}^{\bar{a}} \bar{\otimes} \llbracket \sigma_2 \rrbracket_{\theta}^{\bar{a}} \\ \llbracket \tau \rightarrow \sigma \rrbracket_{\theta}^{\bar{a}} &= \llbracket \tau \rrbracket_{\theta} \bar{\rightarrow} \llbracket \sigma \rrbracket_{\theta}^{\bar{a}}\end{aligned}$$

To properly give the semantics of μ -types, we actually need *functorial actions* of all type constructors, which for \mathbf{T}^e -types can be derived from η^e and \star^e . Once this scaffolding is removed, however, the above set of equations remains.

We also take $\llbracket \Gamma \rrbracket = \prod_{x \in \text{dom } \Gamma} \llbracket \Gamma(x) \rrbracket$; then for $\Gamma \vdash M : \tau$, we define $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket \tau \rrbracket$ as usual. We take complete programs to be closed terms of *observable* type, typically excluding function spaces and some effect types. When the set of *base* effects only contains partiality, the proof of computational adequacy, i.e., that the denotation of a complete program agrees with that program's observable operational behavior, is also standard (Fiore and Plotkin 1994); the extensions to most other common base effects are also straightforward. (We will cover *programmer-defined* effects in Section 3.) As a standard consequence of adequacy and compositionality of the semantics, we have that $\llbracket M \rrbracket = \llbracket M' \rrbracket$ guarantees that M and M' are observationally indistinguishable.

2.2.2 Embedding functional programming languages

A substantial fragment of core SML can be straightforwardly embedded in the metalanguage. However, to obtain stronger reasoning principles, the conceptual embedding $\langle - \rangle$ of types and terms is expressed for an effect-annotated source language, where a function type written as $\tau_1 \rightarrow (\star e \star) \tau_2$ in the code is translated as $\langle \tau_1 \rangle \rightarrow \mathbf{T}^e \langle \tau_2 \rangle$. Unannotated function arrows $\tau_1 \rightarrow \tau_2$ (only allowed when τ_2 is itself a well-formed function type, or a product of such types) are translated as simply $\langle \tau_1 \rangle \rightarrow \langle \tau_2 \rangle$. Terms of such type must be *manifestly pure* functions.

The embedding of Haskell is a bit more problematical. The translation of any Haskell type should be a σ -type over the partiality effect (i.e., a pointed type), which for sum types is ensured by adding an extra lifting \mathbf{T}^{\perp} around their translations. However, to ensure that, e.g., the syntactic state monad does indeed denote a proper monad, we require the η -conversion law to be valid; and this is not the case for full Haskell, when observations of termination at functional types are allowed (at the top level, or using **seq**). We shall disallow such observations, allowing us to consider a slightly “stricter”, PCF-like semantics of the resulting Haskell fragment, where meanings of function types are not further lifted.

For expressing monads and algebras in Haskell and ML, we define some basic infrastructure in Figures 1–2. The Haskell class definitions of monads, algebras, and layering are straightforward transcriptions of the mathematical/metalanguage constructions. Note, however, that we cannot add the free algebra as a class instance; instead, we will need to add it explicitly for every type isomorphic (by **newtype**) to the free algebra.

In ML, monad layering is expressed somewhat differently. First, lacking the overloading notation, it is more convenient to work with the **glue**- rather than the **glet**-formulation of algebras. Also, monad-transformer definitions are not explicitly parameterized over the base effect; rather, all effects are considered as subeffects of the language's native notion of effects. This is particularly useful when the native effect is *universal*, as in the SML/NJ dialect, whose first-class continuations allow any syntactically definable monadic effect to be so embedded. Such an imperative realization of monadic effects is shown in Figure 3, adapted from Filinski (1999) for the case of a single layer of defined effects. Briefly, **reflect** converts a *transparent*, monadic-type representation of an effect into its *opaque*, imperative counterpart, while **reify** converts an imperative computation back to a pure data type. We use this implementation for concrete programming examples, but we stress that it is not essential: absent a native-embedding facility, one can also write client programs explicitly in monadic style, like in Haskell.

```

--class Monad m where
-- return :: a -> m a
-- (>>=) :: m a -> (a -> m b) -> m b

class (Monad m) => MonAlg m b where
  (>>>=) :: m a -> (a -> b) -> b

--overlaps with others
--instance (Monad m) => MonAlg m (m a) where
-- (>>>=) = (>>=)

instance (Monad m, MonAlg m d) => MonAlg m (z -> d) where
  t >>>= f = \z -> t >>>= \a -> f a z

instance (Monad m, MonAlg m d1, MonAlg m d2) =>
  MonAlg m (d1,d2) where
  t >>>= f = (t >>>= (fst . f), t >>>= (snd . f))

class (Monad m, Monad (t m)) => MonadT m t where
  lift :: m a -> t m a

```

Figure 1. Monads and algebras in Haskell

```

type 'd glue = (unit ->(*m*) 'd) -> 'd

signature MONAD = sig
  type 'a t
  val unit : 'a -> 'a t
  val bind : 'a t * ('a -> 'b t) -> 'b t
  val glue : 'a t glue
end;

fun glue_m t = fn () => t () ()

fun glue_f (g: 't glue): ('a -> 't) glue =
  fn t => fn a => g (fn () => t () a)

fun glue_p (g1:'t1 glue, g2:'t2 glue): ('t1 * 't2) glue =
  fn t => (g1 (#1 o t), g2 (#2 o t))

```

Figure 2. Monads and algebras in SML

2.3 Minimal invariance

Consider an SML data type of effectful streams:

```

datatype 'a stream_ =
  Nil
  | Cons of 'a * 'a stream
withtype 'a stream = unit ->(*e*) 'a stream_

```

Streams of type `'a stream` are effectful since functions of type `unit -> 'a stream` can perform computational effects, including divergence, when called. Suppose that these underlying effects are modeled by a monad $\bar{T} = (T, \eta, \star)$. Then, in order to give a domain-theoretic model of the type `'a stream`, we need for each cpo A , a cpo $\text{Str}(A)$ such that

$$\text{Str}(A) \cong T(1 + A \times \text{Str}(A)).$$

In other words, we need to solve a (covariant) domain equation. A solution is given by the interpretation of the open metalanguage type $\mathbf{T}^e(\mu\alpha.1 + \alpha_0 \times \mathbf{T}^e\alpha)$ when \mathbf{T}^e is interpreted as \bar{T} and α_0 is interpreted as A . But in order to prove properties about elements of $\text{Str}(A)$ by induction, it is not enough to know that $\text{Str}(A)$ is *any* solution to the domain equation above. The standard methods for solving domain equations, including the one used to interpret our metalanguage, produce solutions that also satisfy a so-called *minimal invariance* condition, and this condition gives rise to general induction and co-induction principles (Pitts 1996). In this article we investigate consequences of minimal invariance for solutions of covariant domain equations involving monads, such as the one for effectful streams above.

```

signature RMONAD = sig
  include MONAD
  val reflect : 'a t ->(*t*) 'a
  val reify : (unit ->(*t*) 'a) -> 'a t
end;

functor Represent (M : MONAD) : RMONAD =
struct
  open M SMLofNJ.Cont
  val mk : exn M.t cont option ref = ref NONE

  fun abort x = let val SOME k = !mk in throw k x end

  fun reset t =
    let val m = !mk
        val r = callcc (fn k =>
          (mk := SOME k; abort (t ())))
        in mk := m; r end

  fun shift h =
    callcc (fn k =>
      abort (h (fn v => reset (fn () => throw k v))))

  fun reflect m =
    shift (fn k =>
      M.bind (m, fn a => M.glue (fn () => k a)))

  fun reify t =
    let exception D of 'a
        in M.glue (fn () =>
          M.bind (reset (fn () => M.unit (D (t ()))),
            fn (D d) => M.unit d))
        end
    end;
end;

```

Figure 3. Monadic reflection in SML/NJ (condensed)

Definition 2.8. Let $\bar{T} = (T, \eta, \star)$ be a monad layered over the lifting monad, and let $F : \text{Cpo} \rightarrow \text{Cpo}$ be a locally continuous functor. A *minimal T -invariant* for F is a cpo TC together with an isomorphism $i : F(TC) \rightarrow C$ satisfying that

$$\text{fix}(\lambda g^{TC \rightarrow TC}. T(i \circ F(g) \circ i^{-1})) = \text{id}_{TC}. \quad (10)$$

(The least fixed-point is well-defined since TC is pointed whenever \bar{T} is layered over the lifting monad.)

In the example with effectful streams, take $FX = 1 + A \times X$. If (TC, i) is a minimal T -invariant for F , then $C \cong F(TC) = 1 + A \times TC$ and hence $TC \cong T(1 + A \times TC)$. Therefore TC satisfies the domain equation for effectful streams above. Furthermore, the minimal invariance condition (10) is that the SML function f given by

```

fun f s ()
  = (case s ()
     of Nil => (fn () => Nil)
      | Cons (a, s')
         => (fn () => Cons (a, f s')) ())

```

denotes the domain-theoretic identity function on streams. Intuitively, this is an extensionality property: an effectful stream is completely characterized by what happens if you try to access its elements number 1, 2, \dots . That is why effectful streams are amenable to a variant of structural induction.

One can show that for all \bar{T} and F as in Definition 2.8, a minimal T -invariant for F exists. We do not need that general fact. In the next section we will, however, see that interpretations of certain metalanguage types are minimal T -invariants for suitable functors.

Remark. In categorical terms, Definition 2.8 is equivalent to saying that TC is an initial “ F -and- \bar{T} ”-algebra in Cpo (see Section 2.6). Based on that observation, a categorical formulation of the induc-

tion principle shown in Section 2.5 can be developed and proved in a relatively straightforward way. Indeed, the induction principle is a variant of the one presented by Lehmann and Smyth (1981, Section 5.2) and by Crole and Pitts (1992): instead of considering F -algebras we consider F - and \bar{T} -algebras.

2.4 Effectful data types

The *effectful data types* consist of the following subset of metalanguage types:

$$\delta ::= 1 \mid \delta_1 \times \delta_2 \mid \delta_1 + \delta_2 \mid \mathbf{T}^e \delta \mid \mu\alpha.\delta \mid \alpha$$

Notice that such types contain no function arrows. Consequently, all occurrences of type variables are strictly positive.

We aim towards deriving induction principles for data types of the form $\mathbf{T}^e(\mu\alpha.\delta[\mathbf{T}^e\alpha/\alpha])$. As an example, the type

$$\text{Str}(\alpha_0) \equiv \mathbf{T}^e(\mu\alpha.1 + \alpha_0 \times \mathbf{T}^e\alpha)$$

of effectful streams has that form, taking $\delta = 1 + \alpha_0 \times \alpha$.

For the purpose of deriving induction principles for effectful data types, an important fact is that each effectful data type gives rise to a functor. For example, consider the type $\delta = 1 \times \alpha$. For any cpo A , let $F(A) = \llbracket \delta \rrbracket_{[\alpha \mapsto A]} = 1 \times A$. If now $f : A_1 \rightarrow A_2$ is a continuous function, then we can in a natural way define a function $F(f) = \llbracket \delta \rrbracket^d([\alpha \mapsto f]) : F(A_1) \rightarrow F(A_2)$ by $F(f)(u, a_1) = (u, f(a_1))$.

More generally, for δ containing n free type variables we define a functor $\llbracket \delta \rrbracket^d$ of n arguments. First, a few definitions. For a finite set of type variables Δ , let $\llbracket \Delta \rrbracket$ be the set of maps from variables in Δ to cpos. Furthermore, for $\theta, \theta' \in \llbracket \Delta \rrbracket$, the notation $\varphi : \theta \rightarrow \theta'$ means that φ is a map from variables in Δ to continuous functions such that $\varphi(\alpha) : \theta(\alpha) \rightarrow \theta'(\alpha)$ for all α in Δ . The identity map $id_\theta : \theta \rightarrow \theta$ has $id_\theta(\alpha) = id_{\theta(\alpha)}$ for all α , and for $\varphi_1 : \theta \rightarrow \theta'$ and $\varphi_2 : \theta' \rightarrow \theta''$, the composition $\varphi_2 \circ \varphi_1 : \theta \rightarrow \theta''$ is defined pointwise.

Proposition 2.9. *Let $\theta, \theta' \in \llbracket \Delta \rrbracket$ and $\varphi : \theta \rightarrow \theta'$. For every effectful data type δ with free type variables in Δ , there exists a functor $\llbracket \delta \rrbracket^d : \text{Cpo}^\Delta \rightarrow \text{Cpo}$ with $\llbracket \delta \rrbracket^d(\theta) = \llbracket \delta \rrbracket_\theta$ and such that*

$$\llbracket \alpha \rrbracket^d(\varphi) = \varphi(\alpha)$$

$$\llbracket 1 \rrbracket^d(\varphi) = id_1$$

$$\llbracket \delta_1 \times \delta_2 \rrbracket^d(\varphi) = \lambda(d_1, d_2). (\llbracket \delta_1 \rrbracket^d(\varphi) d_1, \llbracket \delta_2 \rrbracket^d(\varphi) d_2)$$

$$\llbracket \delta_1 + \delta_2 \rrbracket^d(\varphi) = \lambda d. \text{case } d \text{ of } \left\{ \begin{array}{l} \text{inl}(d_1). \text{inl}(\llbracket \delta_1 \rrbracket^d(\varphi) d_1) \\ \text{inr}(d_2). \text{inr}(\llbracket \delta_2 \rrbracket^d(\varphi) d_2) \end{array} \right\}$$

$$\llbracket \mathbf{T}^e \delta \rrbracket^d(\varphi) = T^e(\llbracket \delta \rrbracket^d(\varphi)) = \lambda t. T^{T^e \llbracket \delta \rrbracket^d(\varphi)}. t \star \lambda d. \eta^e(\llbracket \delta \rrbracket^d(\varphi) d)$$

$$\llbracket \mu\alpha.\delta' \rrbracket^d(\varphi) = h \quad \text{where } h \text{ is the unique function satisfying} \\ h = \phi_{\alpha.\delta'} \circ \llbracket \delta' \rrbracket^d(\varphi[\alpha \mapsto h]) \circ \phi_{\alpha.\delta'}^{-1}.$$

Furthermore, $\llbracket \delta \rrbracket^d$ is compositional in the following sense:

$$\llbracket \delta_1[\delta_2/\alpha] \rrbracket_\theta^d = \llbracket \delta_1 \rrbracket_{\theta[\alpha \mapsto \llbracket \delta_2 \rrbracket_\theta^d]}^d$$

$$\llbracket \delta_1[\delta_2/\alpha] \rrbracket^d(\varphi) = \llbracket \delta_1 \rrbracket^d(\varphi[\alpha \mapsto \llbracket \delta_2 \rrbracket^d(\varphi)])$$

Proof sketch. Notice that $\llbracket \delta \rrbracket^d(\varphi)$ cannot be defined by a simple induction on δ , since it is not clear that there exists a (unique) function satisfying the requirement on $\llbracket \mu\alpha.\delta' \rrbracket^d(\varphi)$ above. Instead, the existence of $\llbracket \delta \rrbracket^d$ must be argued directly from properties of the functors $\llbracket \tau \rrbracket^f$ in Filinski (2007, Fig. 6) used to construct the interpretations of types in the first place.

Some notation: Let pCpo be the Kleisli category of Cpo with respect to the lifting monad (equivalently, the category of cpos and *partial* continuous functions). For a total function $f : A \rightarrow B$, let $\bar{f} = \eta^\perp \circ f : A \rightarrow B_\perp$ be its inclusion in pCpo .

Since all type variable occurrences in δ are strictly positive, one can show that the functor $\llbracket \delta \rrbracket^f$ is essentially a *covariant* functor

from pCpo^Δ to pCpo . The functorial action of $\llbracket \mu\alpha.\delta' \rrbracket^f$ is then the least fixed-point q of a certain operator on partial functions; we need the fact that q is actually the unique fixed-point of that operator. This can be shown using the minimal-invariant property of the domain $\llbracket \mu\alpha.\delta' \rrbracket_\theta$ and the associated isomorphism.

One then shows that the result of applying $\llbracket \delta \rrbracket^f$ to *total* functions φ is itself a total function: $\llbracket \delta \rrbracket^f(\bar{\varphi}) = \bar{f}$ for some (unique) function f . Now define $\llbracket \delta \rrbracket^d(\varphi)$ as the unique function f such that $\llbracket \delta \rrbracket^f(\bar{\varphi}) = \bar{f}$. The desired equations for $\llbracket \delta \rrbracket^d(\varphi)$ then follow directly from the definition of $\llbracket \delta \rrbracket^f$ and from the uniqueness property for $\llbracket \mu\alpha.\delta' \rrbracket^f$ mentioned above. Similarly, functoriality and compositionality of $\llbracket \delta \rrbracket^d$ follow from the analogous properties of $\llbracket \delta \rrbracket^f$. \square

Spelling out the fact that $\llbracket \delta \rrbracket^d$ is a functor, we have

$$\llbracket \delta \rrbracket^d(id_\theta) = id_{\llbracket \delta \rrbracket_\theta}$$

$$\llbracket \delta \rrbracket^d(\varphi_1 \circ \varphi_2) = \llbracket \delta \rrbracket^d(\varphi_1) \circ \llbracket \delta \rrbracket^d(\varphi_2).$$

Recall that we aim towards deriving induction principles for data types of the form $\mathbf{T}^e(\mu\alpha.\delta[\mathbf{T}^e\alpha/\alpha])$. To that end we need to know that the interpretation of such a type is a minimal T^e -invariant for a certain functor F_θ derived from the type δ .

Let in the following δ be a data type with free type variables in $\Delta \cup \{\alpha\}$. For each $\theta \in \llbracket \Delta \rrbracket$ we define a (one-argument) functor F_θ by “partially applying $\llbracket \delta \rrbracket^d$ to θ ” such that only α varies:

Lemma 2.10. *Let $\theta \in \llbracket \Delta \rrbracket$. Then $F_\theta : \text{Cpo} \rightarrow \text{Cpo}$ defined by*

$$F_\theta(X) = \llbracket \delta \rrbracket_{\theta[\alpha \mapsto X]}$$

$$F_\theta(f) = \llbracket \delta \rrbracket^d(id_\theta[\alpha \mapsto f])$$

is a functor.

Proof. Immediate from the fact that $\llbracket \delta \rrbracket^d$ is a functor. \square

For example, in the case for streams we have that $\delta = 1 + \alpha_0 \times \alpha$ and hence $\Delta = \{\alpha_0\}$. For every fixed cpo A , the lemma above gives a functor $F_A^{\text{str}} = F_{[\alpha_0 \mapsto A]}^{\text{str}}$ such that $F_A^{\text{str}}(X) = 1 + A \times X$, and for $f : X \rightarrow Y$, the function $F_A^{\text{str}}(f) : 1 + A \times X \rightarrow 1 + A \times Y$ is given by

$$F_A^{\text{str}}(f)(m) = \text{case } m \text{ of } \left\{ \begin{array}{l} \text{inl}(). \text{inl}() \\ \text{inr}(a, x). \text{inr}(a, f(x)) \end{array} \right\}. \quad (11)$$

The interpretation of the type $\mathbf{T}^e(\mu\alpha.\delta[\mathbf{T}^e\alpha/\alpha])$ with respect to θ is a minimal T^e -invariant for F_θ :

Lemma 2.11. *Let $\delta_0 = \mu\alpha.\delta[\mathbf{T}^e\alpha/\alpha]$, let $C = \llbracket \delta_0 \rrbracket_\theta$, and let $i : \llbracket \delta[\mathbf{T}^e\delta_0/\alpha] \rrbracket_\theta \rightarrow \llbracket \delta_0 \rrbracket_\theta$ be the associated isomorphism. Furthermore, let F_θ be as in Lemma 2.10. Then $(T^e C, i)$ is a minimal T^e -invariant for F_θ .*

Proof. Let g be any fixed-point of the function

$$\lambda g. T^{T^e C \rightarrow T^e C}. T^e(i \circ F_\theta(g) \circ i^{-1}).$$

We show that $g = id_{T^e C}$. Define $h = i \circ F_\theta(g) \circ i^{-1}$. Then $T^e(h) = g$ by the fixed-point property of g . Now observe that

$$\begin{aligned} F_\theta(g) &= \llbracket \delta \rrbracket^d(id_\theta[\alpha \mapsto g]) = \llbracket \delta \rrbracket^d(id_\theta[\alpha \mapsto T^e(h)]) \\ &= \llbracket \delta[\mathbf{T}^e\alpha/\alpha] \rrbracket^d(id_\theta[\alpha \mapsto h]) \end{aligned}$$

by compositionality of $\llbracket \delta \rrbracket^d$. But then

$$h = i \circ \llbracket \delta[\mathbf{T}^e\alpha/\alpha] \rrbracket^d(id_\theta[\alpha \mapsto h]) \circ i^{-1}$$

satisfies the defining property of $\llbracket \mu\alpha.\delta[\mathbf{T}^e\alpha/\alpha] \rrbracket^d(id_\theta)$. Hence by uniqueness, and by the fact that $\llbracket \mu\alpha.\delta[\mathbf{T}^e\alpha/\alpha] \rrbracket^d$ is a functor, $h = \llbracket \mu\alpha.\delta[\mathbf{T}^e\alpha/\alpha] \rrbracket^d(id_\theta) = id_C$. Finally, since T^e is a functor, $g = T^e(h) = T^e(id_C) = id_{T^e C}$. \square

Returning to the example with effectful streams, let $C_A = \llbracket \mu\alpha.1 + \alpha_0 \times \mathbf{T}^e\alpha \rrbracket_{[\alpha_0 \mapsto A]}$, and let $i : 1 + A \times T^e C_A \rightarrow C_A$

be the associated isomorphism. Then define

$$\text{Str}(A) = \llbracket \text{Str}(\alpha_0) \rrbracket_{[\alpha_0 \mapsto A]} = T^e C_A.$$

The lemma above gives that $(\text{Str}(A), i)$ is a minimal T^e -invariant for F_A^{str} . In the metalanguage, we can define terms $\text{vnil} : \text{Str}(\alpha_0)$ and $\text{vcons} : \alpha_0 \times \text{Str}(\alpha_0) \rightarrow \text{Str}(\alpha_0)$ by

$$\begin{aligned} \text{vnil} &\equiv \text{val}^e \text{in}_{\mu\alpha.1+\alpha_0 \times T^e \alpha}(\text{inl}()) \\ \text{vcons} &\equiv \lambda p. \text{val}^e \text{in}_{\mu\alpha.1+\alpha_0 \times T^e \alpha}(\text{inr}(p)). \end{aligned}$$

Taking $\text{vnil} = \llbracket \text{vnil} \rrbracket_{[\alpha_0 \mapsto A]}$ and $\text{vcons} = \llbracket \text{vcons} \rrbracket_{[\alpha_0 \mapsto A]}$, we get $\text{vnil} \in \text{Str}(A)$ and $\text{vcons} \in A \times \text{Str}(A) \rightarrow \text{Str}(A)$ such that

$$\begin{aligned} \text{vnil} &= \eta^e(i(\text{inl}())) \\ \text{vcons}(a, s) &= \eta^e(i(\text{inr}(a, s))). \end{aligned}$$

2.5 Proof by rigid induction

We now develop a principle for carrying out general proofs on induction on effectful data types, with the explicit goal that the resulting proofs are almost identical to textbook structural induction proofs. The key is to factor out the treatment of effects into an effect-specific notion of admissibility, which leaves the main proof obligations essentially unmodified from the pure case.

2.5.1 T -admissibility

We start by defining a notion of a well-behaved subset P of a general \bar{T} -algebra (D, \otimes) . This amounts to requiring not only that $P \subseteq D$ is a sub-cpo (i.e., that the order relation on D is also a complete partial order on P) but also a sub-algebra (i.e., that the inclusion function is also a T -algebra morphism):

Definition 2.12. Let (D, \otimes) be a \bar{T} -algebra, A subset P of D is T -admissible (with respect to \otimes) if it is chain-complete and satisfies the following condition:

For every cpo E , continuous function $g : E \rightarrow D$ whose range is a subset of P , and element t of TE , the element $t \otimes g$ belongs to P .

We will not mention \otimes explicitly when it is clear from the context.

In particular, when \bar{T} is lifting, the only elements of $TE = E_\perp$ are \perp and $[e]$. Since $\perp \otimes g = \perp_D$, and $[e] \otimes g = g(e)$, a predicate on a pointed cpo D is lift-admissible precisely when it is chain-complete and pointed, i.e., admissible in the sense of the Introduction.

The standard principles for constructing admissible predicates generalize naturally to T -admissible ones:

Proposition 2.13.

1. The equality relation, $P = \{(d_1, d_2) \in D \times D \mid d_1 = d_2\}$ on $(D, \otimes) \times (D, \otimes)$ is T -admissible.
2. If P' is a T -admissible predicate on (D', \otimes') and $h : (D, \otimes) \rightarrow (D', \otimes')$ is a rigid continuous function, then the predicate $P = \{d \in D \mid h(d) \in P'\}$ is T -admissible on (D, \otimes) .
3. If $(P_j)_{j \in J}$ is an arbitrary family of T -admissible predicates on (D, \otimes) , then $P = \bigcap_{j \in J} P_j$ is also T -admissible on (D, \otimes) .

Proof. Straightforward calculations. \square

In particular, any predicate of the form $P = \{x \mid \forall y. f_1(x, y) = f_2(x, y)\}$, where f_1 and f_2 are continuous and rigid in their first arguments, is T -admissible.

2.5.2 F -closure

Next, we define a criterion generalizing that a subset of a data type is closed under constructor applications:

Definition 2.14. Let (TC, i) be a minimal T -invariant for F . Notice that then $\eta \circ i$ is a function from $F(TC)$ to TC . A subset P of TC is F -closed if it satisfies the following condition:

For every cpo E , continuous function $g : E \rightarrow TC$ whose range is a subset of P , and element m of FE , the element $(\eta \circ i)(F(g) m)$ belongs to P .

In the case where F is derived from a simple sum-of-products type δ , the closure condition can be expressed in terms of the data type constructors. For example, let us consider streams (recalling that $F_A^{\text{str}}(X) = 1 + A \times X$):

Proposition 2.15. A subset $P \subseteq \text{Str}(A)$ is F_A^{str} -closed precisely when it satisfies:

1. The element vnil belongs to P .
2. For every a in A and s in P , the element $\text{vcons}(a, s)$ belongs to P .

Proof. First, let $g : E \rightarrow \text{Str}(A)$ be such that $\forall e \in E. g(e) \in P$, and let $m \in F_A^{\text{str}} E = 1 + A \times E$; we must then show that

$$(\eta^e \circ i)(F_A^{\text{str}}(g) m) \in P.$$

For $m = \text{inl}()$, $\eta^e(i(F_A^{\text{str}}(g)(\text{inl}()))) = \eta^e(i(\text{inl}())) = \text{vnil} \in P$ by condition (1); and for $m = \text{inr}(a, e)$, we have

$$\begin{aligned} \eta^e(i(F_A^{\text{str}}(g)(\text{inr}(a, e)))) &= \eta^e(i(\text{inr}(a, g(e)))) \\ &= \text{vcons}(a, g(e)) \in P \end{aligned}$$

by condition (2) and the assumption on g .

Conversely, assume that P is F_A^{str} -closed. Take $E = P$ (viewed as a discrete cpo), and g as the (trivially continuous) inclusion function. Then we get $\text{vnil} \in P$ by considering $m = \text{inl}()$, and $\text{vcons}(a, s) \in P$ using $m = \text{inr}(a, s)$. \square

The generalization to other sum-of-products data type constructors F is evident. When F involves effects and/or recursion, however, we will need the formulation in terms of the functorial action on generators g . We will see an example in Section 2.8.

2.5.3 Proof principle

We can now prove the main theorem, an abstract proof principle:

Theorem 2.16 (Proof by rigid induction). Let (TC, i) be a minimal T -invariant for F . Assume that P is a subset of TC that is both T -admissible and F -closed. Then P is the entire set TC .

Proof. First we show that $\perp_{TC} \in P$. For this, apply the condition in Definition 2.12 with $E = \emptyset$ and $g = \emptyset$ (the empty function from \emptyset to TC): for every $t \in T\emptyset$ we have $t \star g \in P$. In particular $\perp_{T\emptyset} \star g \in P$. But since the monad $\bar{T} = (T, \eta, \star)$ is layered over lifting, \star is strict in its first argument, so $\perp_{T\emptyset} \star g = \perp_{TC}$.

We now aim to show by fixed-point induction and the minimal-invariance condition (10) that id_{TC} belongs to the set

$$Q = \{e : TC \rightarrow TC \mid \forall t \in TC. e(t) \in P\}.$$

This implies that $P = TC$, which ends the proof.

First, Q is chain-complete since P is. Also, by the argument in the beginning of the proof, $\lambda t. \perp_{TC} \in Q$. Finally, assume that $e \in Q$: we now show that then $T(i \circ F(e) \circ i^{-1}) \in Q$. Let $t \in TC$; we must show that the element $T(i \circ F(e) \circ i^{-1})(t) = t \star (\eta \circ i \circ F(e) \circ i^{-1})$ belongs to P . Since P is T -admissible, it suffices to show that the range of the function $\eta \circ i \circ F(e) \circ i^{-1}$ is a subset of P . But this follows from the fact that P is F -closed, taking $g = e$ in the condition in Definition 2.14. In conclusion, by fixed-point induction and minimal invariance (10), $\text{id}_{TC} \in Q$. \square

Together with Lemma 2.11, Theorem 2.16 immediately gives an induction principle for data types of the form $\mathbf{T}^e(\mu\alpha. \delta[\mathbf{T}^e \alpha / \alpha])$. In particular, for effectful streams, the rigid-induction principle, together with the observation after Def. 2.12 (instantiating T -admissibility for lifting) and Prop. 2.15 (characterizing F^{str} -closure in terms of the stream constructors), properly generalizes Theorem 1.1 from the Introduction to arbitrary effects.

As an example of reasoning by rigid induction, suppose that $\text{append} : \text{Str}(A) \rightarrow \text{Str}(A) \rightarrow \text{Str}(A)$ is a function that is rigid in its first argument and satisfies

$$\text{append } \text{vnil } s = s \quad (12)$$

$$\text{append } (\text{vcons } (a, s_1)) s_2 = \text{vcons } (a, \text{append } s_1 s_2). \quad (13)$$

(We will see in the next section that there in fact exists exactly one such function.) We want to show that for all $s \in \text{Str}(A)$, including effectful and/or infinite streams,

$$\text{append } s \text{vnil} = s. \quad (14)$$

Let $P \subseteq \text{Str}(A)$ be the set of streams s for which (14) does hold. By the remark after Prop. 2.13 and the assumption that append is rigid, P is T^e -admissible; and by equations (12) and (13), it clearly contains $s = \text{vnil}$ and $s = \text{vcons } (a, s')$ when $s' \in P$, so P is F_A^{str} -closed by Prop. 2.15. Thus, by Theorem 2.16, P is all of $\text{Str}(A)$, i.e., equation (14) holds universally.

2.6 Definition by rigid induction

As a consequence of proof by rigid induction, we also obtain a convenient principle for defining rigid functions by induction.

Theorem 2.17 (Definition by rigid induction). *Let (TC, i) be a minimal T -invariant for F . For every \bar{T} -algebra (D, \otimes) and every $k : FD \rightarrow D$, the function $\text{fold}_{\otimes}^F(k) : TC \rightarrow D$ defined by*

$$\text{fold}_{\otimes}^F(k) = \text{fix}(\lambda f. \lambda t. t \otimes (k \circ F(f) \circ i^{-1}))$$

is the unique rigid function f from (TC, \star) to (D, \otimes) such that

$$\forall m \in F(TC). f(\eta(i(m))) = k(F(f)(m)). \quad (15)$$

Proof. By the fixed-point equation, $f = \text{fold}_{\otimes}^F(k)$ evidently satisfies that

$$f(t) = t \otimes (k \circ F(f) \circ i^{-1}).$$

Prop. 2.6(2) then immediately tells us that f is rigid; and

$$\begin{aligned} f(\eta(i(m))) &= \eta(i(m)) \otimes (k \circ F(f) \circ i^{-1}) \\ &= k(F(f)(i^{-1}(i(m)))) = k(F(f)(m)) \end{aligned}$$

as required. Moreover, suppose f' is another rigid function satisfying (15). Let $P = \{t \in TC \mid f(t) = f'(t)\}$; we want to show that P is all of TC . By Prop. 2.13, P is T -admissible, so by Theorem 2.16, it suffices to show that P is F -closed. So suppose $g : E \rightarrow TC$ satisfies $\forall e \in E. g(e) \in P$, i.e., $f \circ g = f' \circ g$. Then

$$\begin{aligned} f \circ (\eta \circ i) \circ F(g) &= k \circ F(f) \circ F(g) = k \circ F(f \circ g) \\ &= k \circ F(f' \circ g) = \dots = f' \circ (\eta \circ i) \circ F(g). \end{aligned}$$

So for all $m \in FE$, $(\eta \circ i)(F(g)m) \in P$, as required. \square

Again, for streams, the definition can be expressed more readably in terms of the constructors:

Corollary 2.18. *Let (D, \otimes) be a \bar{T}^e -algebra, $b \in D$, and $g : A \times D \rightarrow D$. Then the function $\text{sfold}_{\otimes} b g : \text{Str}(A) \rightarrow D$ defined by*

$$\begin{aligned} &\text{sfold}_{\otimes} b g \\ &= \text{fix} \left(\lambda f. \lambda t. t \otimes \lambda c. \text{case } i^{-1}(c) \text{ of } \left\{ \begin{array}{l} \text{inl } (). b \\ \text{inr } (a, s). g(a, f s) \end{array} \right\} \right) \end{aligned}$$

is the unique $f : (\text{Str}(A), \star^e) \rightarrow (D, \otimes)$ such that

$$f(\text{vnil}) = b \quad (16)$$

$$f(\text{vcons}(a, s)) = g(a, f(s)). \quad (17)$$

Proof. Follows from Theorem 2.17, by taking $k : (1 + A \times D) \rightarrow D$ as

$$k(m) = \text{case } m \text{ of } \left\{ \begin{array}{l} \text{inl } (). b \\ \text{inr } (a, d). g(a, d) \end{array} \right\}$$

and recalling the definition of $F_A^{\text{str}}(f)$ from (11). \square

Note how we are specifying f on (top-level) pure elements only; rigidity determines its value on all other ones. However, b and g are allowed to contain arbitrary \bar{T}^e -effects; and there is no requirement that g be rigid, or even strict, in its second argument.

The function sfold_{\otimes} can be straightforwardly defined in the metalanguage: for $e \preceq \sigma$, take

$$\begin{aligned} \text{sfold}_{\sigma} b^{\sigma} g^{\alpha_0 \times \sigma \rightarrow \sigma} &\equiv \\ \mu f^{\text{Str}(\alpha_0) \rightarrow \sigma}. \lambda s. \mathbf{glet}_{\sigma}^e c \leftarrow s. \mathbf{case}(\mathbf{out}_{\mu\alpha. 1 + \alpha_0 \times \mathbf{T}^e \alpha}(c), \\ &\quad (). b, \\ &\quad (a, s). g(a, f s)) \end{aligned}$$

Then $\llbracket \text{sfold}_{\sigma} \rrbracket \emptyset = \text{sfold}_{\otimes}$ where $(D, \otimes) = \llbracket \sigma \rrbracket_{[\alpha_0 \mapsto A]}^{\alpha}$.

As an example, we can define the stream-concatenating append function from the previous section. Taking

$$\text{append } s_1 s_2 \equiv \text{sfold}_{\text{Str}(\alpha)} s_2 \text{vcons } s_1$$

we get

$$\text{append} = \llbracket \text{append} \rrbracket \emptyset = \lambda s_1. \lambda s_2. \text{sfold}_{\star^e} s_2 \text{vcons } s_1.$$

It then follows directly from (16) and (17) that append satisfies (12) and (13), respectively. Also, by Corollary 2.18, the function $\lambda s_1. \text{sfold}_{\star^e} s_2 \text{vcons } s_1$ is rigid for each fixed s_2 . Thus, by Prop. 2.6(4), $\text{append} : (\text{Str}(A), \star^e) \rightarrow \text{Str}(A) \rightarrow (\text{Str}(A), \star^e)$.

(In the following, we shall generally omit the obvious metalanguage term xyz whose denotation is the semantic object xyz .)

Uniqueness primarily allows us to talk about *the* rigid function satisfying equations (16) and (17), instead of merely *the least* one. However, we can also use it to reason about general equivalence. For example, to show that

$$\text{append } s_1 (\text{append } s_2 s_3) = \text{append} (\text{append } s_1 s_2) s_3, \quad (18)$$

let s_2 and s_3 be given, and take $b = \text{append } s_2 s_3$ and $g = \text{vcons}$. Then there can be at most one rigid function f satisfying

$$f(\text{vnil}) = \text{append } s_2 s_3$$

$$f(\text{vcons}(a, s)) = \text{vcons}(a, f(s)).$$

But it follows directly from the append equations that both $f_1 = \lambda s. \text{append } s (\text{append } s_2 s_3)$ and $f_r = \lambda s. \text{append} (\text{append } s s_2) s_3$ satisfy the conditions. Since both are rigid, they must be the same function, so $f_1(s_1) = f_r(s_1)$, i.e., (18) holds.

We could of course also have defined append using higher-order iteration: taking $(D, \otimes) = \text{Str}(A) \rightarrow (\text{Str}(A), \star^e)$,

$$\text{append}' = \text{sfold}_{\otimes} (\lambda s_2. s_2) (\lambda(a, h). \lambda s_2. \text{vcons}(a, h s_2)).$$

Since append' is also rigid and evidently satisfies Equations (12-13), it must be equal to the previously defined append .

Note, however, that the uniqueness is *only* among rigid functions. For example, consider the rigid function $f : \text{Str}(A) \rightarrow \text{Str}(A)$ determined by

$$f(\text{vnil}) = f(\text{vcons}(a, s)) = \perp_{\text{Str}(A)}.$$

Can we show that $f(s) = \perp_{\text{Str}(A)}$ for all $s \in \text{Str}(A)$? Clearly $f'(s) = \perp_{\text{Str}(A)}$ also satisfies the equations. When \bar{T}^e models a non-control effect, such as state ($TX = S \rightarrow (X \times S)_{\perp}$), f' is easily seen to be rigid, and hence indeed equal to f . But for other effects, such as exceptions ($TX = (X + E)_{\perp}$), f' is *not* rigid; and indeed, a stream starting with an exception-raising computation is mapped to itself by f , but to \perp by f' .

Finally, it is not important that definitions by rigid induction be expressed explicitly through sfold . For example, we can use a more general primitive-recursion schema: for $b \in D$ and $h : A \times D \times \text{Str}(A) \rightarrow D$, we recursively define $r : \text{Str}(A) \rightarrow D$ by:

$$r(s) = s \otimes \lambda c. \text{case } i^{-1}(c) \text{ of } \left\{ \begin{array}{l} \text{inl } (). b \\ \text{inr } (a, s'). h(a, r(s'), s') \end{array} \right\}.$$

(In particular, we get a constant-time tail-function by letting h simply return its last argument.) By uniqueness, r must be equivalent

to (but more efficient than) the directly `sfold`-definable function that returns a copy of its argument together with the result, allowing the combining function g access to both.

2.7 The stream monad transformer

Using the function `append` defined in the previous section, we can define a monad structure on effectful streams. First, the unit of the stream monad:

$$\eta^{\text{Str}} a = \text{vcons}(a, \text{vnil}). \quad (19)$$

The bind function of the stream monad is defined by rigid induction, similarly to the definition of `append`:

$$s \star^{\text{Str}} h = \text{sfold}_{\star^e} \text{vnil} (\lambda(a, d). \text{append} (h a) d) s.$$

Then \star^{Str} is rigid in its left argument, and satisfies

$$\text{vnil} \star^{\text{Str}} h = \text{vnil} \quad (20)$$

$$(\text{vcons}(a, s)) \star^{\text{Str}} h = \text{append} (h a) (s \star^{\text{Str}} h). \quad (21)$$

Now we aim towards showing that $(\text{Str}, \eta^{\text{Str}}, \star^{\text{Str}})$ is actually a monad, i.e., that η^{Str} and \star^{Str} satisfy the three monad laws. The proof is virtually identical to the finite-list case, since the relevant functions are all rigid by construction.

We have already established that $(\text{Str}(A), \text{append}, \text{vnil})$ form a monoid (12, 14, 18). We will also need that, for $s_1, s_2 \in \text{Str}(A)$ and $f : A \rightarrow \text{Str}(B)$,

$$(\text{append } s_1 s_2) \star^{\text{Str}} f = \text{append} (s_1 \star^{\text{Str}} f) (s_2 \star^{\text{Str}} f). \quad (22)$$

The proof is again by rigid induction on s_1 , using (18).

Proposition 2.19. *The families of continuous functions η^{Str} and \star^{Str} satisfy the three monad laws from Definition 2.1. Hence $(\text{Str}, \eta^{\text{Str}}, \star^{\text{Str}})$ is a monad.*

Proof. Equation (2) follows directly from (14). Equations (3) and (4) are shown by rigid induction on t , using Equation (22). The relevant subsets of $\text{Str}(A)$ are T^e -admissible by construction. \square

Finally, Proposition 2.7 implies that the resulting stream monad is layered over \bar{T}^e . In Haskell terminology, we have defined a *stream monad transformer*.

The ML and Haskell counterparts of the definitions above can be seen in Figures 4–5.

2.8 Streams as a data type constructor

Recall that $\text{Str}(A) = \llbracket \text{Str}(\alpha_0) \rrbracket_{[\alpha_0 \mapsto A]}$. By Proposition 2.9, the stream cpo constructor Str can therefore be extended to a functor by taking $\text{Str}(f) = \llbracket \text{Str}(\alpha_0) \rrbracket^d([\alpha_0 \mapsto f])$. As could be expected, $\text{Str}(f)$ is the function that “maps f ” over its input stream:

Proposition 2.20. *Let $f : A_1 \rightarrow A_2$. The function $\text{Str}(f)$ is the unique rigid function $g : (\text{Str}(A_1), \star^e) \rightarrow (\text{Str}(A_2), \star^e)$ satisfying $g(\text{vnil}) = \text{vnil}$ and $g(\text{vcons}(a, s)) = \text{vcons}(f(a), g(s))$.*

Proof. It follows directly from the definition of $\text{Str}(f)$ that $\text{Str}(f)$ is rigid (by Proposition 2.6(2)) and that it satisfies the requirements on g above. Therefore Corollary 2.18 implies that $\text{Str}(f)$ is the unique rigid function satisfying these requirements. \square

Remark. In the previous section we extended Str to a monad $(\text{Str}, \eta^{\text{Str}}, \star^{\text{Str}})$. Since every monad is also a functor, we have an alternative extension of Str to a functor, namely $\text{Str}'(f) = \lambda s. s \star^{\text{Str}} (\eta^{\text{Str}} \circ f)$. But the proposition above implies that the two resulting functors are identical: $\text{Str}(f) = \text{Str}'(f)$.

Let $\text{map } f = \text{Str}(f) = \llbracket \text{Str}(\alpha_0) \rrbracket^d([\alpha_0 \mapsto f])$. The fact that $\llbracket \delta \rrbracket^d$ is a functor for all δ immediately gives a “map fusion” law for effectful streams: $\text{map } (f \circ g) = (\text{map } f) \circ (\text{map } g)$. As another example of proof by rigid induction, we show that, additionally, the following “map-fold fusion” law holds:

```
data Stream_ m a = Nil
                | Cons a (StreamT m a)
newtype StreamT m a = ST { rST :: m (Stream_ m a) }

instance Monad m => MonAlg m (StreamT m b) where
  t >>= f = ST (t >>= \a -> rST (f a))

instance Monad m => MonadT m StreamT where
  lift m = m >>= return

vnil :: Monad m => StreamT m a
vnil = ST (return Nil)

vcons :: Monad m => a -> StreamT m a -> StreamT m a
vcons a s = ST (return (Cons a s))

sfold :: (Monad m, MonAlg m b) =>
        b -> (a -> b -> b) -> StreamT m a -> b
sfold n c s =
  rST s >>= \s -> case s of
    Nil -> n
    Cons a s -> c a (sfold n c s)

append :: Monad m =>
        StreamT m a -> StreamT m a -> StreamT m a
append s1 s2 = sfold s2 vcons s1
append' :: Monad m =>
        StreamT m a -> StreamT m a -> StreamT m a
append' =
  sfold (\s2 -> s2) (\a -> \h -> \s2 -> vcons a (h s2))

instance Monad m => Monad (StreamT m) where
  return a = vcons a vnil
  t >>= f = sfold vnil (\a -> \d -> append (f a) d) t
```

Figure 4. Streams in Haskell

```
datatype 'a stream_ =
  Nil
  | Cons of 'a * 'a stream
withtype 'a stream = unit -> (*m*) 'a stream_

val glue_stream : 'a stream glue = glue_m

val vnil = fn () => Nil
fun vcons (a,s) = fn () => Cons (a,s)

fun sfold gl n c s = gl (fn () =>
  case s () of Nil => n
              | Cons (h,s) => c (h, sfold gl n c s))

fun append s1 s2 = sfold glue_stream s2 vcons s1
val append' = (* ignoring value restriction *)
  sfold (glue_f glue_stream)
        (fn s2 => s2)
        (fn (a,h) => fn s2 => vcons (a, h s2))

structure StreamM : MONAD = struct
  type 'a t = 'a stream
  fun unit a = vcons (a, vnil)
  val glue = glue_stream
  fun bind (t,f) =
    sfold glue vnil (fn (a,d) => append (f a) d) t
end
```

Figure 5. Streams in SML

Proposition 2.21. *Let (D, \otimes) be a \bar{T} -algebra, and let $b \in D$, $f : A_1 \rightarrow A_2$, $g : A_2 \times D \rightarrow D$, and $s \in \text{Str}(A_1)$. Then*

$$\text{sfold}_{\otimes} b g (\text{map } f s) = \text{sfold}_{\otimes} b (\lambda(a_1, d). g(f(a_1), d)) s.$$

Proof. By rigid induction on s . The relevant predicate on s is T^e -admissible since the functions $\text{map } f$ and $\text{sfold}_{\otimes} b g$ and $\text{sfold}_{\otimes} b (\lambda(a_1, d). g(f(a_1), d))$ are rigid. \square

These fusion laws will be used in Section 4.4.

3. Relational reasoning

Equational reasoning suffices for many purposes, but when proving properties of higher-order functions, one must usually generalize to *logical relations*, because the relationships are no longer expressible as simple equations. When the programs also involve recursive, and especially reflexive types, however, the natural specification of the relation family also becomes circular, and one must carefully exploit how the recursive domains were constructed in the first place to make sure that the desired relations even exist (Reynolds 1974; Pitts 1996; Filinski 2007). We will not go into the technical details here, concentrating instead on demonstrating how the equational results from the previous section can be applied in such a relational setting.

The setup is that we want to treat an effect as an *abstract data type*, with a monad and its proper operations seen as a module implementing an interface. We can then talk about contextual equivalence of two such modules as substitutability in all programs respecting the abstraction boundary, i.e., not manipulating the monadic values directly, but always going through the interface. The monad-representation operators of Figure 3 are one particular instance of this: here the operations `reflect` and `reify` actually fully expose the monad type up to (observational) isomorphism; nevertheless, this still leaves room for radically different implementations. In this section, we will show two further instances of showing different implementations of an effect-ADT equivalent.

To formalize implementations of effects, it is convenient to talk of a *signature* Σ of effect names and constants. The semantics of our metalanguage is given with respect to a base signature Σ_0 , representing a fixed programming language. If a program is written over a larger signature than Σ_0 , it also needs a *realization* Φ of the extensions, i.e., definitions of the additional effects and constants, so that the expanded program $M[\Phi]$ can be executed; this corresponds to simple module linking, or instantiation of overloaded constants. We use $\Phi.c \equiv M$ to define constants in realizations. (We sometimes write $c x \equiv M$ for $c \equiv \lambda x. M$, but never recursively.) For defining new effects, we introduce the following:

Definition 3.1. A formal monad over an effect e_0 consists of a type constructor $\sigma_T(\cdot)$ such that $e_0 \preceq \sigma_T(\alpha)$, and closed terms $M_\eta : \alpha \rightarrow \sigma_T(\alpha)$ and $M_* : \sigma_T(\alpha) \times (\alpha \rightarrow \sigma_T(\alpha')) \rightarrow \sigma_T(\alpha')$.

Semantically, we say that such a formal monad denotes a (proper) monad if the denotations of the terms (with type variables interpreted as arbitrary cpos) satisfy the monad laws and the layering condition from Prop. 2.7.

Syntactically, a realization containing an effect definition $\Phi.e \equiv (\sigma_T, M_\eta, M_*)$ can be used to eliminate types and terms associated with the effect:

$$(\mathbf{T}^e \tau)[\Phi] = \sigma_T(\tau[\Phi])$$

$$(\mathbf{val}^e M)[\Phi] = M_\eta(M[\Phi])$$

$$(\mathbf{let}^e x \leftarrow M_1.M_2)[\Phi] = M_* (M_1[\Phi], \lambda x. M_2[\Phi])$$

One can check that the meaning of a program linked with a formal monad agrees with the meaning of the program in an interpretation extended with that monad's denotation. For relating two different monadic implementations of an effect, we introduce:

Definition 3.2. An *admissible relation* R between two cpos A and A' is a chain-complete subset of $A \times A'$, i.e., satisfying that if for all $i \in \omega$, $(a_i, a'_i) \in R$, then also $(\bigsqcup_i a_i, \bigsqcup_i a'_i) \in R$. We write $\mathbf{ARel}(A, A')$ for the set of all such relations.

Definition 3.3. A *relational action* \mathcal{R} for a pair of monads (T_1, η_1, \star_1) and (T_2, η_2, \star_2) maps relations $R \in \mathbf{ARel}(A_1, A_2)$ to $\mathcal{R}(R) \in \mathbf{ARel}(T_1 A_1, T_2 A_2)$, satisfying that

1. $(\perp_{T_1 A_1}, \perp_{T_2 A_2}) \in \mathcal{R}(R)$.

2. If $(a, a') \in R$ then $(\eta_1 a, \eta_2 a') \in \mathcal{R}(R)$.
3. If $(t, t') \in \mathcal{R}(R)$ and for all $(a, a') \in R$, $(f(a), f'(a')) \in \mathcal{R}(S)$, then $(t \star_1 f, t' \star_2 f') \in \mathcal{R}(S)$.

Let there now be given two realizations Φ_1 and Φ_2 of Σ , and for every $e \in \Sigma$, a relational action \mathcal{R}^e for the monads denoted by $\Phi_1.e$ and $\Phi_2.e$. We can then show:

Proposition 3.4. Let ψ be a relation map for $\theta_1, \theta_2 \in \llbracket \Delta \rrbracket$, i.e., for all $\alpha \in \Delta$, $\psi(\alpha) \in \mathbf{ARel}(\theta_1(\alpha), \theta_2(\alpha))$. Then for every type τ , with effect names from Σ , and $F\text{TV}(\tau) \subseteq \Delta$, there exists a relation $(\sim_\tau^\psi) \in \mathbf{ARel}(\llbracket \tau[\Phi_1] \rrbracket, \llbracket \tau[\Phi_2] \rrbracket)$ such that:

$$a \sim_\alpha a' \Leftrightarrow (a, a') \in \psi(\alpha)$$

$$u \sim_1 u' \Leftrightarrow \text{true}$$

$$p \sim_{\tau_1 \times \tau_2} p' \Leftrightarrow \pi_1(p) \sim_{\tau_1} \pi_1(p') \wedge \pi_2(p) \sim_{\tau_2} \pi_2(p')$$

$$s \sim_{\tau_1 + \tau_2} s' \Leftrightarrow (\exists a \sim_{\tau_1} a'. s = \text{inl}(a) \wedge s' = \text{inl}(a')) \vee$$

$$(\exists a \sim_{\tau_2} a'. s = \text{inr}(a) \wedge s' = \text{inr}(a'))$$

$$f \sim_{\tau \rightarrow \sigma} f' \Leftrightarrow \forall a \sim_\tau a'. f a \sim_\sigma f' a'$$

$$t \sim_{\mathbf{T}^e \tau} t' \Leftrightarrow (t, t') \in \mathcal{R}^e(\sim_\tau)$$

$$m \sim_{\mu\alpha.\tau} m' \Leftrightarrow \phi_{\alpha,\tau}^{-1}(m) \sim_{\tau[\mu\alpha.\tau/\alpha]} \phi_{\alpha,\tau}^{-1}(m')$$

(Since ψ remains fixed, we have omitted it, to avoid clutter.)

Proof sketch. Again, the last clause prevents us from simply defining the relation \sim_τ by induction on τ . Instead, existence and uniqueness of the relation family must once more be argued from the minimal-invariant property of the recursive-type interpretations (Pitts 1996), exploiting the conditions on relational actions of effects (Filinski 2007). \square

We require that for observable types o , $a \sim_o a' \Rightarrow a = a'$, and that for any $(c : \tau) \in \Sigma_0$, $\llbracket c \rrbracket \emptyset \sim_\tau \llbracket c \rrbracket \emptyset$; this is usually easily checked. For the defined constants and effects, we take:

Definition 3.5. Realizations $\Phi_1, \Phi_2 : \Sigma$ are related by relational-action family $(\mathcal{R}^e)_{e \in \Sigma}$ if for every $(c_\alpha : \tau) \in \Sigma$; $\theta_1, \theta_2 \in \llbracket \bar{\alpha} \rrbracket$; and $\psi \in \mathbf{ARel}(\theta_1, \theta_2)$, $\llbracket \Phi_1.c \rrbracket_{\theta_1} \sim_\tau^\psi \llbracket \Phi_2.c \rrbracket_{\theta_2}$.

Theorem 3.6 (Fundamental Lemma). Let Φ_1 and Φ_2 be related realizations, and for every $(x : \tau) \in \Gamma$, $\rho(x) \sim_\tau^\psi \rho'(x)$. Then for every $\Gamma \vdash_\Sigma M : \tau$, $\llbracket M[\Phi_1] \rrbracket_{\theta_1} \rho \sim_\tau^\psi \llbracket M[\Phi_2] \rrbracket_{\theta_2} \rho'$.

Proof. By induction on M , with most cases standard. For the case $\mu x^\sigma.M'$ we use fixed-point induction, exploiting that the relation \sim_σ is admissible and pointed; the cases for **in** and **out** are immediate from the characterization of $\sim_{\mu\alpha.\tau}$; and the ones for **val** and **let** follow from the definition of relational actions. \square

In particular, if we can find a relational action for the monads underlying two realizations of an effect-ADT, such that their operations are related, the two realizations are observationally indistinguishable. Candidate relational actions are often easy to construct:

Proposition 3.7. The following are valid principles for constructing monadic relational actions:

1. *Syntactic.* If the effect e is defined by the same formal monad $(\sigma_T(\cdot), M_\eta, M_*)$ in both Φ_1 and Φ_2 , then $\mathcal{R}(R) = (\sim_{\sigma_T(\alpha)}^{\llbracket \alpha \mapsto R \rrbracket})$ is a relational action for the corresponding monads.
2. *Inverse image.* If $\iota_1 : \bar{T}_1 \rightarrow \bar{T}'_1$ and $\iota_2 : \bar{T}_2 \rightarrow \bar{T}'_2$ are monad morphisms, and \mathcal{R}' is a relational action for \bar{T}'_1 and \bar{T}'_2 , then $\mathcal{R}(R) = \{(t_1, t_2) \mid (\iota_1 t_1, \iota_2 t_2) \in \mathcal{R}'(R)\}$ is a relational action for \bar{T}_1 and \bar{T}_2 .
3. *Intersection.* If $\mathcal{R}_j \in \mathcal{J}$ are all relational actions for \bar{T}_1 and \bar{T}_2 , then so is $\mathcal{R}(R) = \bigcap_{j \in \mathcal{J}} \mathcal{R}_j(R)$.

Proof. Each part follows easily:

1. $\mathcal{R}(R)$ is admissible by construction, and relates the denotations of M_η and M_* by the Fundamental Lemma.

2. $\mathcal{R}(R)$ is admissible by being an inverse image by strict continuous functions of another admissible relation, and relates the unit and bind functions because of the monad-morphism laws.
3. Straightforward. \square

4. Application: relating nondeterministic-search monad transformers

As a practical example of using the techniques, we present two instances of relating implementations of an abstract data type, in the context of backtracking search. The first one is the well-known, but surprisingly tricky to formalize, relationship between success-stream and failure-continuation models of depth-first backtracking. The second is about relating the higher-level abstraction of search trees and traversal strategies to solution streams.

4.1 Relating streams and 2-continuations

4.1.1 The 2-continuation monad

In formal models of backtracking (notably for embedding Prolog in functional languages), a frequent alternative to the natural solution-stream approach is a representation of computations with success and failure continuations. This model can be seen as a standard continuation monad, but with the answer type itself being a continuation-computation type. The “outer”, success continuation is thus effectively itself written in continuation-passing style, and is invoked on each solution and the “inner”, failure continuation.

(Sometimes the failure continuation is left implicit as the normal call stack. A sequence of answers is then represented as a series of calls to the success continuation, with the return values ignored. Ultimately, such answers must therefore be recorded using some computational effect, such as interactive I/O.)

When R is a computational type with $e \preceq R$, we define the formal R -continuation monad over e by

$$\begin{aligned} \text{Cont}(\alpha) &\equiv (\alpha \rightarrow R) \rightarrow R \\ \text{unit}_{\alpha}^{\text{Cont}} a &\equiv \lambda k^{\alpha \rightarrow R}. k a \\ \text{bind}_{\alpha, \alpha'}^{\text{Cont}} (t, f) &\equiv \lambda k^{\alpha' \rightarrow R}. t (\lambda a^{\alpha}. f a k) \end{aligned}$$

Note that when $e \preceq R$, then also $e \preceq \text{Cont}(\tau)$. Analogously, we can fulfill the condition on R by taking $R \equiv (1 \rightarrow S) \rightarrow S$ for some S with $e \preceq S$. Since the failure continuation takes no meaningful argument, we will actually simplify this to just $R \equiv S \rightarrow S$.

The Haskell and ML representations of the general continuation monad are shown in Figures 6–7.

4.1.2 Backtracking operations

In the presentation of Wand and Vaillancourt (2004), the abstract effect of *backtracking*, BT, can be thought of as having a signature with the following operations:

$$\begin{aligned} \text{bfail}_{\alpha} &: \mathbf{T}^{\text{BT}}_{\alpha} \\ \text{bdisj}_{\alpha} &: \mathbf{T}^{\text{BT}}_{\alpha} \rightarrow \mathbf{T}^{\text{BT}}_{\alpha} \rightarrow \mathbf{T}^{\text{BT}}_{\alpha} \\ \text{answers} &: \mathbf{T}^{\text{BT}}_{\text{o}} \rightarrow \text{Str}(\text{o}) \end{aligned}$$

`bfail` and `bdisj` represent failure and nondeterministic choice, respectively. `answers` is the implicit operation turning a backtracking computation – however it is implemented – into an observable stream of answers of some fixed observable type `o`.

In the stream-based realization $\Phi_{\text{Str}}^{\text{BT}}$, we directly take

$$\begin{aligned} \Phi_{\text{Str}}^{\text{BT}}.\text{BT} &\equiv (\text{Str}(\cdot), \text{unit}^{\text{Str}}, \text{bind}^{\text{Str}}) \\ \Phi_{\text{Str}}^{\text{BT}}.\text{bfail} &\equiv \text{vnil} \\ \Phi_{\text{Str}}^{\text{BT}}.\text{bdisj } s_1 s_2 &\equiv \text{append } s_1 s_2 \\ \Phi_{\text{Str}}^{\text{BT}}.\text{answers } s &\equiv s \end{aligned}$$

```
newtype MonAlg m r =>
  ContT r m a = CT { rCT :: (a -> r) -> r }

instance (Monad m, MonAlg m r) =>
  MonAlg m (ContT r m a) where
  t >>= f = CT (t >>= \a -> rCT (f a))

instance Monad m => Monad (ContT r m) where
  return a = CT (\k -> k a)
  m >>= f = CT (\k -> rCT m (\a -> rCT (f a) k))

instance (Monad m, MonAlg m r) => MonadT m (ContT r) where
  lift m = m >>= return
```

Figure 6. Continuations in Haskell

```
functor Cont (type r val glue_r : r glue) : MONAD =
struct
  type 'a t = ('a -> r) -> r

  fun unit a = fn k => k a
  fun bind (t, f) = fn k => t (fn a => f a k)
  fun glue t = glue_f glue_r t
end;
```

Figure 7. Continuations in SML

For the 2-continuation monad, we choose the “inner” answer type S as $\text{Str}(\text{o})$, i.e.,

$$R \equiv \text{Str}(\text{o}) \rightarrow \text{Str}(\text{o})$$

We can then take

$$\begin{aligned} \Phi_{\text{Cont}}^{\text{BT}}.\text{BT} &\equiv (\text{Cont}(\cdot), \text{unit}^{\text{Cont}}, \text{bind}^{\text{Cont}}) \\ \Phi_{\text{Cont}}^{\text{BT}}.\text{bfail} &\equiv \lambda k. \lambda c. c \\ \Phi_{\text{Cont}}^{\text{BT}}.\text{bdisj } u_1 u_2 &\equiv \lambda k. \lambda c. u_1 k (u_2 k c) \\ \Phi_{\text{Cont}}^{\text{BT}}.\text{answers } u &\equiv u (\lambda a. \lambda s. \text{vcons}(a, s)) \text{vnil} \end{aligned}$$

(Note that only the last operation depends on the above choice of the type S).

The corresponding code is shown in Figures 8–9.

To relate the two realizations, we first define a function `stoc` : $\text{Str}(A) \rightarrow \text{Cont}(A)$ by rigid induction;

$$\text{stoc } \text{vnil } k c = c \quad (23)$$

$$\text{stoc } (\text{vcons}(h, t)) k c = k h (\text{stoc } t k c) \quad (24)$$

(`stoc` is of course explicitly definable in the metalanguage, but we do not rely on this.)

Lemma 4.1. $\text{stoc}(\text{append } s_1 s_2) k c = \text{stoc } s_1 k (\text{stoc } s_2 k c)$.

Proof. Straightforward rigid induction on s_1 . \square

Lemma 4.2. *stoc is a monad morphism from streams to continuations.*

Proof. Equation (5) follows directly from (23) and (24); (6) is shown by rigid induction on the stream t , using Lemma 4.1. \square

Lemma 4.3. $\text{stoc } s (\lambda a. \lambda s'. \text{vcons}(a, s')) \text{vnil} = s, s \in \text{Str}(\text{o})$.

Proof. Straightforward rigid induction on s . \square

We now use the equations shown above to establish the relational correspondence.

Proposition 4.4. *The realizations $\Phi_{\text{Str}}^{\text{BT}}$ and $\Phi_{\text{Cont}}^{\text{BT}}$ are related, and hence observationally equivalent in all well-typed contexts.*

Proof. Since `stoc` is a monad morphism (Lemma 4.2), as is the identity on $\text{Cont}(A)$, by Prop. 3.7(1,2), we can define the relational action of BT as an inverse image of the syntactic $\text{Cont}(\alpha)$ -relation:

$$s \sim_{\mathbf{T}^{\text{BT}}_{\alpha}} u \stackrel{\text{def}}{\iff} \text{stoc } s \sim_{\text{Cont}(\alpha)} u$$

```

type Obs = String

class (Monad m) => BTT m t where
  bfail :: t m a
  bdisj :: t m a -> t m a -> t m a
  answers :: t m Obs -> StreamT m Obs

instance (Monad m) => BTT m StreamT where
  bfail = vnil
  bdisj = append
  answers s = s

type SAns m = StreamT m Obs -> StreamT m Obs

instance (Monad m) => BTT m (ContT (SAns m)) where
  bfail = CT (\k -> \c -> c)
  bdisj g1 g2 = CT (\k -> \c -> rCT g1 k (rCT g2 k c))
  answers u = rCT u vcons vnil

```

Figure 8. Backtracking in Haskell

```

type obs = string

signature BT = sig
  val bfail : unit ->(*bt*) 'a
  val bdisj :
    (unit ->(*bt*) 'a) -> (unit ->(*bt*) 'a) ->(*bt*) 'a
  val answers : (unit -> (*bt*) obs) -> obs stream
end;

structure StreamBT : BT =
struct
  structure SR = Represent(StreamM) open SR
  fun bfail () = reflect vnil
  fun bdisj t1 t2 = reflect (append (reify t1) (reify t2))
  fun answers t = reify t
end;

type r = obs stream -> obs stream
val glue_r : r glue = glue_f glue_stream

structure ContM = Cont (type r = r val glue_r = glue_r);

structure ContBT : BT =
struct
  structure CR = Represent(ContM) open CR
  fun bfail () = reflect (fn k => fn c => c)
  fun bdisj t1 t2 =
    reflect (fn k => fn c => reify t1 k (reify t2 k c))
  fun answers t = reify t (fn a => fn c => vcons(a,c)) vnil
end;

```

Figure 9. Backtracking in SML

We must now check that the constants are related. The case for `bfail` is effectively a degenerate variant of `bdisj`, so let us only consider the latter. Looking at its type, and expanding the definition of \sim for function types (twice), suppose $stoc\ s_1 \sim_{Cont(\alpha)} u_1$ and $stoc\ s_2 \sim_{Cont(\alpha)} u_2$; we must show that

$$stoc\ ([\Phi_{Str}^{BT}.bdisj]\ s_1\ s_2) \sim_{Cont(\alpha)} [[\Phi_{Cont}^{BT}.bdisj]]\ u_1\ u_2. \quad (25)$$

But by Lemma 4.1 on the LHS,

$$\begin{aligned} stoc\ ([\Phi_{Str}^{BT}.bdisj]\ s_1\ s_2) &= stoc\ (append\ s_1\ s_2) \\ &= \lambda k. \lambda c. stoc\ s_1\ k\ (stoc\ s_2\ k\ c) \\ &= [[\Phi_{Cont}^{BT}.bdisj]]\ (stoc\ s_1)\ (stoc\ s_2) \end{aligned}$$

and hence (25) follows from the Fundamental Lemma for the term $\Phi_{Cont}^{BT}.bdisj$. (Note that we did not even have to expand $\sim_{Cont(\alpha)}$.)

Finally, for answer extraction, suppose $s \sim_{T^{BT}_o} u$, i.e., $stoc\ s \sim_{Cont(o)} u$; we must show that

$$[[\Phi_{Str}^{BT}.answers]]\ s \sim_{Str(o)} [[\Phi_{Cont}^{BT}.answers]]\ u. \quad (26)$$

Using Lemma 4.3 on the LHS,

```

dfpick :: (Monad m, MonadT m t, BTT m t) => [a] -> t m a
dfpick [] = bfail
dfpick (h:t) = bdisj (return h) (dfpick t)

instance MonAlg IO (IO a) where
  (>>=) = (>>=)

printStats :: StreamT IO String -> IO ()
printStats =
  sfold (return ()) (\a -> \d -> putStr (a ++ "\n") >> d)

evens :: StreamT IO Int -- or ContT (SAns IO) IO Int
evens =
  do x1 <- dfpick [3,4,5]
     () <- lift (putStr ("x1=" ++ show x1 ++ "? "))
     x2 <- dfpick [5,6,7]
     () <- lift (putStr ("x2=" ++ show x2 ++ "? "))
     let v = x1 * x2
         () <- if v `mod` 2 == 1 then bfail else return ()
     return v

printevens :: IO ()
printevens =
  printStream (answers (evens >>= (return . show)))

```

Figure 10. Tracing example in Haskell

```

open StreamBT; (* or ContBT *)

fun dfpick [] = bfail ()
  | dfpick (h::t) = bdisj (fn () => h) (fn () => dfpick t)

fun printstream s =
  sfold glue_m
    (fn () => ())
    (fn (s, d) => fn () => (print (s ^ "\n"); d ()))
  s ()

fun evens () =
  let val x1 = dfpick [3,4,5]
      val () = print ("x1=" ^ Int.toString x1 ^ "? ")
      val x2 = dfpick [5,6,7]
      val () = print ("x2=" ^ Int.toString x2 ^ "? ")
      val v = x1 * x2
      val () = if v mod 2 = 1 then bfail() else ()
  in v end

fun printevens () =
  printstream (answers (fn () => Int.toString (evens ())))

```

Figure 11. Tracing example in SML

```

x1=3? x2=5? x2=6? 18!
x2=7? x1=4? x2=5? 20!
x2=6? 24!
x2=7? 28!
x1=5? x2=5? x2=6? 30!
x2=7?

```

Figure 12. Output from `printevens`

$$\begin{aligned} [[\Phi_{Str}^{BT}.answers]]\ s &= s = stoc\ s\ (\lambda a. \lambda s'. vcons\ (a, s'))\ vnil \\ &= [[\Phi_{Cont}^{BT}.answers]]\ (stoc\ s) \end{aligned}$$

so (26) follows from Theorem 3.6 for the term $\Phi_{Cont}^{BT}.answers$. \square

Note that the theorem also applies directly to backtracking layered above some other effect, such as I/O. Effectful operations in the underlying monad are still represented in the stream, even if they do not ultimately lead to solutions. For example, in Figures 10–12, we show how tracing output gets embedded in the answer stream, so that when the latter is eventually printed, the output elements are properly interleaved with the computations leading up to them. The output is identical for the ML and Haskell versions,

```

data Tree_ m a = Leaf a
               | Node [TreeT m a]
newtype TreeT m a = TT { rTT :: m (Tree_ m a) }

instance Monad m => MonAlg m (TreeT m b) where
  t >>= f = TT (t >>= \a -> rTT (f a))

vleaf :: Monad m => a -> TreeT m a
vleaf a = TT (return (Leaf a))

vnode :: Monad m => [TreeT m a] -> TreeT m a
vnode l = TT (return (Node l))

tfold :: (Monad m, MonAlg m b) =>
  (a -> b) -> ([b] -> b) -> TreeT m a -> b
tfold f n t =
  rTT t >>= \m -> case m of
    Leaf a -> f a
    Node l -> n (map (tfold f n) l)

instance Monad m => Monad (TreeT m) where
  return a = vleaf a
  t >>= f = tfold f vnode t

pick :: Monad m => [a] -> TreeT m a
pick l = vnode (map return l)

catstr :: Monad m => [StreamT m a] -> StreamT m a
catstr = foldr append vnll

ttos :: Monad m => TreeT m a -> StreamT m a
ttos = tfold return catstr

```

Figure 13. Trees in Haskell (SML version is analogous)

and for the stream-based and 2-continuation based implementations of backtracking.

4.2 Relating trees and streams

We now consider a higher-level interface for backtracking, which allows the programmer to define alternative search strategies. The key concept is the *tree monad transformer* presented next; it allows one to construct potentially effectful search trees.

4.2.1 The tree monad transformer

Consider the metalanguage type

$$\mathbf{Tree}(\alpha_0) \equiv \mathbf{T}^e(\mu\alpha. \alpha_0 + \mathbf{List}(\mathbf{T}^e\alpha))$$

where $\mathbf{List}(\alpha) = \mu\alpha'. 1 + \alpha \times \alpha'$. For every cpo A , define $C_A = \llbracket \mu\alpha. \alpha_0 + \mathbf{List}(\mathbf{T}^e\alpha) \rrbracket_{[\alpha_0 \mapsto A]}$ and

$$\mathbf{Tree}(A) = \llbracket \mathbf{Tree}(\alpha_0) \rrbracket_{[\alpha_0 \mapsto A]} = T^e C_A.$$

Then $\mathbf{Tree}(A)$ is a domain of effectful, finitely-branching trees with leaves from A .

Also, let $\mathbf{List}(A) = \llbracket \mathbf{List}(\alpha_0) \rrbracket_{[\alpha_0 \mapsto A]}$ and let $i : A + \mathbf{List}(\mathbf{Tree}(A)) \rightarrow C_A$ be the isomorphism associated with C_A . Then define constructors $vleaf \in A \rightarrow \mathbf{Tree}(A)$ and $vnode \in \mathbf{List}(\mathbf{Tree}(A)) \rightarrow \mathbf{Tree}(A)$ by

$$\begin{aligned} vleaf a &= \eta^e(i(\text{inl } a)) \\ vnode l &= \eta^e(i(\text{inr } l)). \end{aligned}$$

The type $\mathbf{Tree}(\alpha_0)$ has the form $\mathbf{T}^e(\mu\alpha. \delta[\mathbf{T}^e\alpha/\alpha])$ considered in Section 2.4, defining $\delta = \alpha_0 + \mathbf{List}(\alpha)$. Therefore, in order to derive an induction principle for effectful trees (elements of $\mathbf{Tree}(A)$), we need to consider the definition of $\llbracket \delta \rrbracket^d$.

First, the type $\mathbf{List}(\alpha)$ gives rise to a functor “in α ”: let $\mathbf{List}(f) = \text{map } f = \llbracket \mathbf{List}(\alpha) \rrbracket^d([\alpha \mapsto f])$. One can show that \mathbf{List} is in fact a finite list functor: $\mathbf{List}(X)$ is the cpo of finite lists with elements from X , and $\mathbf{List}(f)$ is indeed the familiar “map” function that applies f to each element of its argument.

Proposition 4.5. *Let P be a T^e -admissible subset of $\mathbf{Tree}(A)$. Suppose that P satisfies the following conditions:*

1. *For every a in A , the element $vleaf a$ belongs to P .*
2. *For every cpo E , continuous function $g : E \rightarrow \mathbf{Tree}(A)$ such that the range of g is a subset of P , and $l \in \mathbf{List}(E)$, the element $vnode(\text{map } g l)$ belongs to P .*

Then P is the entire set $\mathbf{Tree}(A)$.

Proof. By Lemma 2.10 we obtain a functor F_A^{tree} with $F_A^{\text{tree}}(X) = \llbracket \delta \rrbracket_{[\alpha_0 \mapsto A, \alpha \mapsto X]} = A + \mathbf{List}(X)$ and

$$F_A^{\text{tree}}(f) = \llbracket \delta \rrbracket^d([\alpha_0 \mapsto id_A, \alpha \mapsto f]) = id_A + \mathbf{List}(f).$$

In order to apply Theorem 2.16 we must check that P is F_A^{tree} -closed. But this is easily seen to be equivalent to the two conditions on P above. \square

Remark. Using the fact that \mathbf{List} is a finite list functor, condition (2) in the above proposition can be reformulated in a more natural-looking way: For all t_1, \dots, t_n in P , the element $vnode[t_1, \dots, t_n]$ belongs to P . One advantage of the formulation in the proposition is that it generalizes to infinitely branching trees (see Section 4.4).

The principle for definition by induction given by Theorem 2.17 is:

Proposition 4.6. *Let (D, \otimes) be a T^e -algebra, let $g_1 : A \rightarrow D$, and $g_2 : \mathbf{List}(D) \rightarrow D$. Then the function $\text{tfold}_{\otimes} g_1 g_2 : \mathbf{Tree}(A) \rightarrow D$ defined by*

$$\begin{aligned} \text{tfold}_{\otimes} g_1 g_2 \\ = \text{fix} \left(\lambda f. \lambda t. t \otimes \lambda c. \text{case } i^{-1}(c) \text{ of } \left\{ \begin{array}{l} \text{inl } a. g_1(a) \\ \text{inr } l. g_2(\text{map } f l) \end{array} \right\} \right) \end{aligned}$$

is the unique $f : (\mathbf{Tree}(A), \star^e) \rightarrow (D, \otimes)$ such that

$$\begin{aligned} f(vleaf a) &= g_1(a) \\ f(vnode l) &= g_2(\text{map } f l). \end{aligned}$$

Like for streams, the function tfold is definable by a term in our metalanguage.

For example, we can define a function ttos flattening an effectful tree to an effectful stream in depth-first order:

$$\begin{aligned} \text{catstr} &= \text{foldr append vnll} \in \mathbf{List}(\mathbf{Str}(A)) \rightarrow \mathbf{Str}(A) \\ \text{ttos} &= \text{tfold}_{\star^e} \eta^{\text{Str}} \text{catstr} \in \mathbf{Tree}(A) \rightarrow \mathbf{Str}(A) \end{aligned}$$

(Here foldr is the familiar “fold right” function on lists.) A Haskell implementation is shown in Figure 13; the SML implementation is similar. However, the Haskell version actually implements *infinitely branching* effectful trees; we return to this point in Section 4.4.

Monad structure We use Proposition 4.6 to define a monad structure $(\mathbf{Tree}, \eta^{\text{Tree}}, \star^{\text{Tree}})$ on effectful trees, similarly to what we did for effectful streams:

$$\begin{aligned} \eta^{\text{Tree}} a &= vleaf a \\ t \star^{\text{Tree}} h &= \text{tfold}_{\star^e} h vnode t \end{aligned}$$

Then \star^{Tree} is rigid (in its first argument) and satisfies

$$\begin{aligned} (vleaf a) \star^{\text{Tree}} h &= h a \\ (vnode l) \star^{\text{Tree}} h &= vnode(\text{map } (\lambda t. t \star^{\text{Tree}} h) l). \end{aligned}$$

Proposition 4.7. *The families of continuous functions η^{Tree} and \star^{Tree} satisfy the three monad laws of Definition 2.1. Hence $(\mathbf{Tree}, \eta^{\text{Tree}}, \star^{\text{Tree}})$ is a monad.*

Proof. Similar to the corresponding proof for streams, but simpler. Equation (2) is a simple calculation, while Equations (3) and (4) follow directly by rigid induction on t (Proposition 4.5), using one of the functor laws for \mathbf{List} : $\text{map}(f \circ g) = (\text{map } f) \circ (\text{map } g)$. \square

As for streams, Proposition 2.7 implies that the resulting tree monad $(Tree, \eta^{Tree}, \star^{Tree})$ is layered over T^e . In Haskell terminology, we have defined a *tree monad transformer*. Since the tree monad transformer can be layered on top of monad transformers for state and exception effects, one can in particular use it to implement search strategies that terminate early after a certain number of backtracking operations, or search strategies such as branch-and-bound that depend on search-global state which is *not* restored on backtracking.

4.2.2 Search operations

We consider the signature of the abstract search effect ND to contain two operations:

$$\begin{aligned} \text{pick}_\alpha &: \text{List}(\alpha) \rightarrow \mathbf{T}^{\text{ND}}\alpha \\ \text{collect}_\alpha &: \mathbf{T}^{\text{ND}}\alpha \rightarrow \text{Str}(\alpha) \end{aligned}$$

The first one expresses a nondeterministic choice between the values in a list (with an empty list representing failure); the second produces a stream of all successful answers from a nondeterministic computation. Note that, in general, it need not be the case that $\text{pick}[a] = \eta a$: a choice point with only one alternative may be observably different from a completely pure computation.

We can implement this abstract effect with two realizations: one based on search trees, and one based on solution streams:

$$\begin{aligned} \Phi_{Tree}^{\text{ND}}.\text{ND} &\equiv (\text{Tree}(\cdot), \text{unit}^{Tree}, \text{bind}^{Tree}) \\ \Phi_{Tree}^{\text{ND}}.\text{pick } l &\equiv \text{vnode}(\text{map unit}^{Tree} l) \\ \Phi_{Tree}^{\text{ND}}.\text{collect } r &\equiv \text{ttos } r \\ \Phi_{Str}^{\text{ND}}.\text{ND} &\equiv (\text{Str}(\cdot), \text{unit}^{Str}, \text{bind}^{Str}) \\ \Phi_{Str}^{\text{ND}}.\text{pick } l &\equiv \text{foldr vcons vnul } l \\ \Phi_{Str}^{\text{ND}}.\text{collect } s &\equiv s \end{aligned}$$

In the tree-based one, a pick is represented as a choice node with trivial computations in the branches; the collect performs a depth-first traversal of the tree. In the stream-based representation, pick immediately converts the list of choices to a stream of answers, making collect trivial. Although (as we will show), the observable net effect is the same, the tree-based representation also makes it possible for collect to use traversals other than *ttos*, while the stream-based one commits to depth-first already at the choice point.

Lemma 4.8. *ttos is a monad morphism from trees to streams.*

Proof. By rigid induction on trees (Proposition 4.5). One needs the following lemma, which follows from Equation (22) by induction on the finite list l : $(\text{catstr } l) \star^S f = \text{catstr}(\text{map}(\lambda s. s \star^S f) l)$. \square

Lemma 4.9. *For all $l \in \text{List}(A)$,*

$$\text{ttos}(\text{vnode}(\text{map } \eta^{Tree} l)) = \text{foldr vcons vnul } l.$$

Proof. Straightforward verification, using the map/fold fusion laws for finite lists. \square

Proposition 4.10. *The realizations Φ_{Tree}^{ND} and Φ_{Str}^{ND} are observationally equivalent.*

Proof. Like for streams, we define the relational action of ND as an inverse image: $t \sim_{\mathbf{T}^{\text{ND}}\alpha} s \Leftrightarrow \text{ttos } t \sim_{\text{Str}(\alpha)} s$. That the realizations of pick are related follows from Lemma 4.9 and the Fundamental Lemma for $\Phi_{Str}^{\text{ND}}.\text{pick}$; and the case for collect follows directly from the definition. \square

4.3 Breadth-first traversal

The Prolog-style depth-first search semantics embodied in the stream and 2-continuation models of backtracking is not appropriate for all programming tasks. For a programmer, the main ad-

vantage of programming to the ND API rather than the BT one, is that the former retains the possibility of other tree traversal orders.

Here we consider the example of breadth-first search. In order to implement breadth-first search using the search tree model we simply need to construct a stream that represents a traversal of the effectful search tree in breadth-first order. This can be done with a standard algorithm that uses a queue of subtrees. Alternatively, one can define breadth-first traversal compositionally, using the general *tfold* function derived from Proposition 4.6; the Haskell code is shown in Figure 14. The resulting definition can actually be seen as the refunctionalized counterpart (Danvy and Millikin 2007) of the standard algorithm. The idea is to use a recursive type to allow one to process all children of a node “in advance” when first encountering them. Hofmann (1993) analyzes a variant due to Tofte that uses a similar idea.

As an example of a programming task that admits a natural solution by breadth-first backtracking, consider the problem of constructing a stream of all untyped, closed λ -terms, up to α -equivalence. The solution, in Haskell and SML, is shown in Figures 15 and 16; in both cases, `terms` contains the desired stream, which can be inspected lazily.

4.4 Infinitely branching effectful trees

Strictly speaking (no pun intended), the Haskell implementation of Trees in Figure 13 is not quite faithful to the metalanguage version, because it is expressed using the native Haskell data type of (lazy) lists, and thus more accurately corresponds to *infinitely branching* effectful trees.

It turns out that the development in the previous sections goes through when substituting lazy lists for eager lists. Instead of $\text{Tree}(\alpha_0)$, consider the type $\text{Tree}'(\alpha_0) = \mu\alpha. \alpha_0 + \text{Str}(\mathbf{T}^e\alpha)$. In Section 2.8 it was shown that the functorial action of $\text{Str}(\alpha)$ is a “map” function on streams. Therefore, we obtain induction principles for Tree' completely similar to those for Tree (Propositions 4.5 and 4.6), except that *map* is now the map function on streams instead of lists. Furthermore, if we take the effect in the definition of $\text{Str}(\cdot)$ to be lifting, we obtain precisely the data type of Haskell’s lazy lists. The development in the previous sections then goes through mostly as before, using the fusion laws for streams from Section 2.8. In effect, one uses rigid induction on streams instead of structural induction on lists.

4.5 Other models of backtracking

There exist a number of other models of backtracking and search strategies related to monads in the literature (Wand and Vaillancourt 2004; Danvy et al. 2002; Hughes 1995; Hinze 2000; Kiselyov et al. 2005). Spivey (2006) gives a general categorical account of search strategies based on monads, including one based on finitely branching trees analogous to our *Tree*, but without embedded effects. With the exception of the models of Wand and Vaillancourt (2004), these models have not been investigated in terms of object languages with formal semantics. We hope that the techniques presented in this article could be useful in such a formalization.

5. Conclusions

With a little care, the well-known induction principles for reasoning about algebraic data types scale to settings where the data values contain embedded computational effects, not necessarily limited to partiality. Moreover, such reasoning extends smoothly to programs with higher-order and reflexive types: although the reasoning framework will generally have to be relational, most concrete verifications still boil down to familiar equational reasoning by induction. In particular, we have systematized and generalized the result of Wand and Vaillancourt about relating the solution-stream and 2-

```

data Rec m a = Fun { rFun :: [Rec m a] -> StreamT m a }

instance Monad m => MonAlg m (Rec m a) where
  t >>= f = Fun (t >>= (rFun . f))

bfs_aux :: Monad m => [Rec m a] -> StreamT m a
bfs_aux [] = vnil
bfs_aux (Fun f : fs) = f fs

bfs :: Monad m => TreeT m a -> StreamT m a
bfs t = bfs_aux [f]
  where f = tfold (\a -> Fun (\fs -> vcons a (bfs_aux fs)))
                 (\fs' -> Fun (\fs -> bfs_aux (fs ++ fs')))
                 t

```

Figure 14. Breadth-first search in Haskell (SML is analogous)

```

data Term = Var String
          | App Term Term
          | Lam String Term
  deriving Show

genTerm :: (Monad m) => TreeT m Term
genTerm = gen [] where
  gen vs = pick [do x <- pick vs; return (Var x),
                do t1 <- gen vs; t2 <- gen vs;
                 return (App t1 t2),
                let v = "x" ++ show (length vs) in
                 do t <- gen (v:vs); return (Lam v t)]
  >>= id

terms :: Monad m => StreamT m Term
terms = bfs genTerm

```

Figure 15. Lambda-term generation in Haskell

```

datatype term = Var of string
              | App of term * term
              | Lam of string * term

structure N = Represent(TreeM);
fun rpick l = N.reflect (pick l)
fun rbfs t = bfs (N.reify t)

fun genTerm () =
  let fun gen vs =
        rpick [fn () => Var (rpick vs),
               fn () => App (gen vs, gen vs),
               fn () => let val v =
                           "x" ^ Int.toString (length vs)
                       in Lam(v, gen (v :: vs)) end] ()
    in gen [] end

val terms : term stream = rbfs genTerm

```

Figure 16. Lambda-term generation in SML

continuation models of depth-first backtracking, and sketched how more general search strategies can fit into the framework.

More broadly, we believe our results illustrate that the category-theoretical foundations of monads and related concepts provide a rich source of reasoning principles for effective functional programs, whether the monadic structure is explicit as in Haskell, or only implicit, as in most ML programs. Functional programs *should* be easy to reason about, and proofs by structural induction on data types are even presented in most introductory functional-programming texts. We hope to have shown that these reasoning techniques remain valid and easy to use even in “imperative functional” settings.

In the examples, we have concentrated on backtracking models atop some existing notion of computation, such as partiality or I/O, which represents *persistent* effects that should not be undone or disappear on backtracking. An equally important technique is

to layer further monadic effects on top of backtracking, such as state-based destructive unification, as found in most realistic Prolog implementations. Monadic layering allows us to separate concerns, decoupling the underlying search strategy from higher-level computations using choice as a black-box abstraction. This is currently work in progress, and we hope to report on it at a later time. However, we believe the concept of rigid induction is a powerful yet easy-to-use principle in its own right, and represents a useful addition to the functional programmer’s toolbox.

Acknowledgments

We thank the anonymous reviewers for comments and suggestions that helped improve the presentation.

References

- Roy L. Crole and Andrew M. Pitts. New foundations for fixpoint computations: FIX-hyperdoctrines and the FIX-logic. *Information and Computation*, 98:171–210, 1992.
- Olivier Danvy and Kevin Millikin. Refunctionalization at work. BRICS Research Report RS-07-7, Department of Computer Science, University of Aarhus, 2007.
- Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unifying approach to goal-directed evaluation. *New Generation Computing*, 20(1):53–73, 2002. ISSN 0288-3635.
- Andrzej Filinski. Representing layered monads. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.
- Andrzej Filinski. On the relations between monadic semantics. *Theoretical Computer Science*, 375(1):41–75, 2007.
- Marcelo Fiore and Gordon D. Plotkin. An axiomatisation of computationally adequate domain theoretic models of FPC. In *Proceedings of the Ninth Symposium on Logic in Computer Science*, pages 92–102, Paris, France, 1994.
- Ralf Hinze. Deriving backtracking monad transformers. In *ICFP ’00: Proceedings of the 5th International Conference on Functional programming*, pages 186–197. ACM Press, 2000.
- Martin Hofmann. Non strictly positive datatypes for breadth first search. TYPES Forum posting, 1993. Available from <http://www.seas.upenn.edu/~sweirich/types/archive/1993/msg00027.html>.
- John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming, First International Spring School*, volume 925 of LNCS, pages 53–96, 1995.
- Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers. In *ICFP ’05: Proceedings of the 10th International conference on Functional programming*, pages 192–203. ACM Press, 2005.
- Daniel J. Lehmann and Michael B. Smyth. Algebraic specification of data types: A synthetic approach. *Mathematical Systems Theory*, 14:97–139, 1981.
- Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.
- Andrew M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.
- Chris Reade. *Elements of Functional Programming*. Addison Wesley, 1989.
- John C. Reynolds. On the relation between direct and continuation semantics. In *2nd Colloquium on Automata, Languages and Programming*, volume 14 of LNCS, pages 141–156, Saarbrücken, Germany, July 1974.
- Michael Spivey. Algebras for combinatorial search. In *Workshop on Mathematically Structured Functional Programming*, Kuressaare, Estonia, July 2006.
- Mitchell Wand and Dale Vaillancourt. Relating models of backtracking. In *ICFP ’04: Proceedings of the 9th International conference on Functional programming*, pages 54–65. ACM Press, 2004.