Invertible Syntax without the Tuples (Functional Pearl)

Mathieu Boespflug
Tweag
Paris, France
m@tweag.io

Arnaud Spiwack
Tweag
Paris, France
arnaud.spiwack@tweag.io

Abstract

In the seminal paper *Functional unparsing*, Olivier Danvy used continuation passing to reanalyse printf-like format strings as combinators. In the intervening decades, the conversation shifted towards a concurrent line of work — applicative, monadic or arrow-based combinator libraries — in an effort to find combinators for invertible syntax descriptions that simultaneously determine a parser as well as a printer, and with more expressive power, able to handle inductive structures such as lists and trees. Along the way, continuation passing got lost. This paper argues that Danvy's insight remains as relevant to the general setting as it was to the restricted setting of his original paper. Like him, we present three solutions that exploit continuation-passing style as an alternative to both dependent types and monoidal aggregation via nested pairs, in our case to parse and print structured data with increasing expressive power.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Parsers**.

Keywords: continuation-passing style, biparser, parser combinators, bidirectional programming, indexed monad

ACM Reference Format:
Mathieu Boespflug and Arnaud Spiwack. 2025. Invertible Syntax without the Tuples (Functional Pearl). In *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday (OLIVIERFEST '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3759427.3760381

1 The Problem

Mcbride and Paterson [2008] captured a common programming pattern as the `Applicative` class:

```
class Functor f where
  (<$>)  :: (a → b) → f a → f b

class Functor f ⇒ Applicative f where
  pure   :: a → f a
  (<*>)  :: f (a → b) → f a → f b
```

While the above is now standard in Haskell, Mcbride and Paterson note that the following alternative definition is equivalent:

```
class Functor f ⇒ Monoidal f where
  unit   :: f ()
  mult   :: f a → f b → f (a, b)
```

One instance of programming with applicatives is parser combinator libraries in the style of Röjemo [1995], also called *deserializers*. Conversely, combinators for pretty printing, also called *serializers*, are an instance of coapplicative programming, characterized by reversing the arrow in the definition of the `Functor` class:

```
class Contravariant f where
  (>$<)  :: (a → b) → f b → f a

class Contravariant f ⇒ Comonoidal f where
  counit :: f ()
  comult :: f a → f b → f (a, b)
```

What we want: dual-use combinators, which we will call *format descriptors*, for both parsing and pretty printing.

We could characterize descriptors as profunctors (see *e.g.* [Xia et al. 2019]), which generalize both functors and contravariants:

```
class Profunctor p where
  dimap :: (b → a) → (c → d)
        → p a c → p b d
```

with an instance like the following:

```
data (pr ⊗ pa) a b = pr a ⊗ pa b

instance (Contravariant pr, Functor pa) ⇒
  Profunctor (pr ⊗ pa) where
    dimap f g (pr ⊗ pa) =
      (f >$< pr) ⊗ (g <$> pa)
```

Tuple Troubles. We could, furthermore, define a new sequencing operator for profunctors analogous to `(<*>)` above:

```
(***) :: (Comonoidal pr, Monoidal pa)
      ⇒ (pr ⊗ pa) a a'
```


OLIVIERFEST '25, Singapore, Singapore

ACM ISBN 979-8-4007-2150-2/25/10
https://doi.org/10.1145/3759427.3760381

```
      → (pr ⊗ pa) b b'
      → (pr ⊗ pa) (a, b) (a', b')
(pr ⊗ pa) *** (pr' ⊗ pa') =
    comult pr pr' ⊗ mult pa pa'
```

But herein lies **the problem**: chaining `pr ⊗ pa` combinators in a sequence leads to deeply nested pairs. Effectively, it appears that we are forced to work in an awkward recursively uncurried style, when currying is more natural, avoids noisy wrapping/unwrapping of constructors, and potentially has better performance.

Example 1.1. Assume a record `Rec` with three fields. Given (de)serializers `bool`, `char` and `int`, a (de)serializer for `Rec` might be of the form given in `ex1` below:

```
data Rec = Rec {a :: Bool, b :: Char, c :: Int}

unwrapRec Rec{a, b, c} = ((a, b), c)
wrapRec ((a, b), c) = Rec{a, b, c}

ex1 :: _ ⇒ (pr ⊗ pa) Rec Rec
ex1 = dimap unwrapRec wrapRec spec where
  spec = bool *** char *** int
```

The (un)wrapping functions (a pair of them per constructor of the data type) can, and should, be derived generically. Nevertheless, such an API requires frequent calls to adaptor functions to mediate between elements of nested tuples and components of data types. Worse, different programming patterns call for nesting to the left like `((a, b), c)` or to the right like `(a, (b, c))`. But associativity doesn't come free. We have to resolve any impedance mismatch by hand with conversion functions. When an API has this tendency, we shall say that the API exhibits *tuple troubles*.

The uncurried style leaks into many derived combinators, including higher-order combinators like folds and generic combinators from standard abstractions like `Applicative` and `Alternative`, which either need to be redefined based on tupling, or duplicated [Rendel and Ostermann 2010].

In short, we seek to dualize Röjemo's immensely popular style of parser combinators to parsing/printing pairs, but,

1. without compromise on the ergonomics, and
2. without compromise on expressivity, to support arbitrary structured data, including collections and trees (sums of products), not just records (products).

In this paper, we present three solutions. The latter two are novel and satisfy all of the above requirements, moreover in a setting limited to Haskell'98 + rank-2 polymorphic types.

2 Analysis

Consider the following variations of class methods introduced above:

```
comult' :: Comonoidal f ⇒ (r → (a, b))
        → f a → f b → f r
comult' split f g = split >$< comult f g

mult'   :: Monoidal f ⇒ (a → b → r)
        → f a → f b → f r
mult' combine f g =
  uncurry combine <$> mult f g
```

They reveal a deeply significant asymmetry between splitting comonoidally and combining monoidally: `combine` is a curried function of two arguments, but flipping the arrow in the type of `split` gives us an uncurried function. This is the crux of the problem: there is no notion of "curried" multiple-result function. This is why `Comonoidal` exhibits tuple troubles, while `Applicative` manages to avoid them. We need symmetry to sequence covariant and contravariant functors simultaneously; uncurrying functions everywhere is one way to recover that symmetry. But there is another way.

The key insight in this paper is to invoke the folklore observation that in continuation-passing style (CPS), multiple-argument functions and multiple-result functions are completely symmetric, and the types can be written with arrows alone. We need not pack multiple results in a tuple, like in the `Comonoidal` class.

Compare the types of 2-argument functions with 2-value functions in CPS (with continuations passed first, not last):

```
(s → r) → (a → b → r) -- two arguments
(a → b → r) → (s → r) -- two return values
```

3 An Applicative Solution

Swierstra and Duponcheel [1996] propose a surprisingly succinct variation of the following interface for the primitives atop which most other Röjemo-style parser combinators are derived:

```
class _ p ⇒ Parsing p where
  satisfy :: (Char → Bool) → p Char
```

We can vary the expressive power of the combinators with different choices to fill the constraint hole in the above. Swierstra and Duponcheel [1996] use `Alternative`. Röjemo [1995] uses `Monad`. In this section, we will match the power of `Applicative` i.e. `Alternative` without choice. However, inspired by the polyvariadic printing of Danvy [1998], our setting will be a more general notion of polyvariadic parsers, instead of the unary parsers above.

Danvy postulates an abstract syntax of *format descriptors*:

- `lit` for declaring literal strings;
- `int` for specifying integers;
- `str` for specifying strings;
- `compose` to "glue [descriptor] components together".

This language of descriptors is used to specify how to pretty print (or *format*) input data, but can serve just as well to

specify how to parse it [Asai et al. 2011]. How would we do so, and can we relate this to the `Parsing` interface above?

3.1 String Transformers

Recall the definition of a `Category` as a structure with objects and morphisms `cat a b` from object `a` to object `b`, including identity morphisms, all of which can be composed:

```
class Category cat where
  id :: cat a a
  (∘) :: cat b c → cat a b → cat a c
```

Now, consider the type of string transformers, and their type in CPS:

$$\text{String} \to \text{String} \tag{0}$$

$$(\text{String} \to r) \to (\text{String} \to r) \tag{1}$$

Functions always take a *continuation* as their first argument. The return type r of continuations is called the *answer type*, which in general is allowed to change from continuation to continuation. Answer-type modification (ATM) is considered a *control effect*. So the most general type of string transformers can be defined as below. String transformers form a category under function composition:

```
newtype Tr r r' =
  Tr ((String → r) → String → r')

instance Category Tr where
  id = Tr id
  Tr f ∘ Tr g = Tr (f ∘ g)
```

Notice that parsing and pretty printing both fit within the more general framework of CPS string transformers:

```
Tr r (a → r)                -- For printing
Tr (a → r) r                -- For parsing
```

We can check this by expanding the definition of `Tr`. A string transformer for parsing provides the parsed value and the remaining string to the continuation. For printing, a string transformer expects an input string and a value to print, prints the value to another string, and provides its continuation with the concatenation of these two strings:

```
(String → r) → String → a → r  -- print
(String → a → r) → String → r  -- parse
```

Notice that here string transformers for parsing are those for printing with a flipped top-most arrow. The first type is an instance of the type of (printing) format descriptors in [Danvy 1998]. In printers, `r` is typically polymorphic, such that with polymorphic functions `f :: Tr (a → r) r` and `g :: Tr (b → r) r`, instantiating `r` in the type of `f` gives, `(f :: Tr (a → b → r) (b → r))`, so that the composition `f ∘ g` has type `Tr (a → b → r) r`, again for an arbitrary `r`. Type instantiation, like this, is the essence of composition in [Danvy 1998].

The type for parsing is novel[1]. But indeed, there are multiple ways to construct a type of parsing descriptors with `Tr`. An alternative is to instantiate the answer types for printing with flipped polarities[2]:

```
Tr r (a → r)                -- For printing
Tr (r → t) ((a → r) → t)    -- For parsing
```

This type for parsing string transformers is an instance of the type of (parsing) format descriptors proposed by Asai et al. [2011].

Like for the type of printing descriptors, gluing parsing descriptors to form new descriptors is composition of morphisms, *i.e.* just function composition. Therefore, a pair of printing/parsing string transformers, which we'll call a *cassette* (or `K7` for short), is itself an instance of `Category`[3]:

```
data K7 p a b = K7
  { sideA :: p a b,
    sideB :: ∀t. p (a → t) (b → t) }

instance Category p ⇒ Category (K7 p) where
  id = K7 id id
  ~(K7 f f') ∘ ~(K7 g g') =
      K7 (f ∘ g) (f' ∘ g')

type PP a = ∀r. K7 Tr r (a → r)
type PP0 = ∀r. K7 Tr r r
```

Cassettes have a tape with a pair of tracks. The `Category` instance tells us that the tapes of two cassettes can be spliced to form a new cassette. Crucially, splicing is associative: `a ∘ (b ∘ c)` and `(a ∘ b) ∘ c` denote the same cassette.

A `PP a` is simultaneously a descriptor for printing and for parsing values of type `a`. In the framework of string transformers, this is but a special case of descriptors of any arity, so we introduce type synonym `PP0` for nullary descriptors.

3.2 Primitive and Derived String Transformers

`satisfy` is a primitive descriptor to print/parse a single character satisfying the given predicate:

```
satisfy :: (Char → Bool) → PP Char
satisfy p = K7 (Tr pr) (Tr pa) where
  pr k s c = k (s ++ [c])
  pa k s@(c:cs) u | p c = k cs (u c)
```

We now have enough to implement `lit`, the descriptor that prints/parses a literal string, as a derived combinator:

```
lit :: String → PP0
lit [] = id
```

[1] Novel but flawed: if `f :: Tr r (a → r)` and `g :: Tr r (b → r)`, then `f ∘ g :: Tr r (b → a → r)`. We want `a`, `b` in the oppposite order.
[2] Recall that a type occurrence is positive if it lies to the right of an even number of arrows starting from the top level, and negative if to the right of an odd number.
[3] We will see later why using lazy irrefutable pattern matching is important.

```
lit (c:cs) = satisfy (== c) ∘ lit cs
```

We can translate between curried and uncurried style with the following primitive:

```
pairL :: K7 Tr (a → b → r) ((a, b) → r)
pairL = K7 (Tr (uncurry ∘)) (Tr (curry ∘))
```

Finally, it is useful to afford ourselves an equivalent to (<$>) in applicative-style parsing, to map over a descriptor. In our setting, this is possible given any pair of morphisms that are inverses of each other (i.e. an isomorphism), one for printing and one for parsing:

```
isoL :: (s → a) → (a → s)
     → K7 Tr (a → r) (s → r)
isoL to from = K7 (Tr pr) (Tr pa) where
  pr k s t = k s (to t)
  pa k s u = k s (\x → u (from x))
```

Example 3.1. Typed `sprintf()` [Asai 2009; Danvy 1998] and its inverse `sscanf()` [Asai et al. 2011] are both implementable in our framework:

```
sprintf :: K7 Tr String r → r
sprintf (K7 (Tr pr) _) = pr id ""

sscanf :: K7 Tr r r' → String → r' → r
sscanf (K7 _ (Tr pa)) s k = pa (const id) s k
```

Given the following derived combinators,

```
char :: PP Char
char = satisfy (const True)

digit :: PP Int
digit = conv ∘ satisfy isDigit where
  conv = isoL (head ∘ show) (\c → read [c])
```

and the following format descriptor,

```
spec = digit ∘ lit "-th character after " ∘
       char ∘ lit " is " ∘ char
```

we can implement an example due to Asai et al. [2011]:

```
>>> sprintf spec 5 'a' 'f'
"5-th character after a is f"

>>> sscanf spec "5-th character after a is f" (,,)
(5, 'a', f')
```

The inferred type of spec is `K7 Tr r (Int → Char → r)`. This is the type of a binary descriptor.

Where in the `Parsing` interface, descriptors are applicative combinators, which can be sequenced (<*>) and mapped over (<$>), here descriptors are modeled as morphisms, which compose (∘) and can be mapped over with `isoL`. The respective operators are in 1:1 correspondence. `pure` corresponds to mapping over `id` with `isoL`. We have the same expressive power, in a different formalism. What we have gained is printing with the descriptors, not just parsing.

4 An Alternative Solution

The `char` and `digit` descriptors of Example 3.1 only match a single character from the input. The full `str` and `int` descriptors of Danvy are, on the parsing side, examples of *maximal munch*, which means matching as long of a portion of the input as possible. To derive them as iterating `char` and `digit` requires a new primitive encoding the pattern, or in general, a notion of choice, to specify when to continue munching or to stop.

Returning to the `Parsing` interface, in this section we choose `Alternative` as the superclass, whose two methods serve to add failure and choice to applicatives, respectively:

```
class Applicative f ⇒ Alternative f where
  empty  :: f a
  (<|>)  :: f a → f a → f a
```

This interface has sufficient expressive power to express context free grammars [Swierstra and Duponcheel 1996].

4.1 Vertical Composition

Our morphisms are not applicatives, but we can likewise add similar structure:

```
class Monoid a where
  mempty :: a
  (<>) :: a → a → a
```

If we could make string transformers an instance of `Monoid`, we would horizontally compose morphisms using (∘), and vertically compose using (<>). Both operators have units.

Example 4.1. We can optionally apply a nullary descriptor. If it fails, that's fine — we'll just do nothing in that case:

```
optional :: PP0 → PP0
optional p = p <> id
```

Cassettes are monoids if the underlying category is itself a monoid:

```
instance (∀r r'. Monoid (p r r')) ⇒
    Monoid (K7 p r r') where
  mempty = K7 mempty mempty
  K7 f f' <> K7 g g' = K7 (f <> g) (f' <> g')
```

So we need to adapt `Tr` to make it a monoid. Parsers that sometimes fail are often implemented the same way we model partiality: with the `Maybe` monoid. For example, if a parser fails, it does not yield a value nor the rest of the string. It returns `Nothing`. We'll need partiality for printing as well, so that both parse descriptors and descriptors can uniformly be composed vertically to incrementally build a total descriptor describing how to both print and parse.

Example 4.2. Assume a variant `lit' :: PP ()` equivalent to `lit :: PP0` and a primitive `prismL` to lift a prism [Pickering et al. 2017] to string transformers (more on prisms in Section 4.3). We can construct a (total) descriptor for formatting boolean literals piecewise from (partial) descriptor components, one for each constructor of data type `Bool`:

```
_True :: Prism' Bool ()
_False :: Prism' Bool ()

true, false :: PP Bool
true = prismL _True ⟶ lit' "T" where
false = prismL _False ⟶ lit' "F" where

bool = true <> false
```

The problem is that there is no uniform way to introduce `Maybe` in the answer types of both sides of a cassette. On the printing side, say the continuation has return type `Maybe r`. Then we can no longer compose two printing string transformers. For example,

```
>>> let p = _ :: Tr (Maybe r) (a → r')
>>> p ∘ p
-- Type error: can't match a->r' with Maybe r.
```

So what would Olivier Danvy do? Perhaps observe that there is no problem that cannot be solved with an extra pass of the CPS transform[4]! Well then, let's continue the series of string transformer types one more level — that corresponding to level 2 in the CPS hierarchy [Danvy and Filinski 1990]:

```
                String → String                    (0)
        (String → r) → (String → r)                (1)
((String → r) → r) → ((String → r) → r)            (2)
```

Iterating the CPS transform one more time gives us access to a second continuation. We will use this continuation as a failure continuation [Röjemo 1995]. By convention, we will say `k` for the success continuation and `k'` for the failure continuation.

The definition of `Tr` now reads as follows, again allowing for ATM:

```
type C r = (String → r) → String → r

newtype Tr r r' = Tr { unTr :: C r → C r' }
```

As before, `Tr` is an instance of `Category`. The definitions of these instances remain unchanged. But we can now add a new instance:

```
instance Monoid (Tr r r') where
  mempty = Tr \_ k' s → k' s
  Tr f <> Tr g =
    Tr \k k' s → f k (\_ → g k k' s) s
```

[4]See *e.g.* [Danvy 1996, §3] for one instance of that.

`mempty` is the string transformer that throws away the success continuation because it always fails. And where `(∘)` is (horizontal) composition of success continuations, `(<>)` is (vertical) composition of failure continuations. If the first transformer fails, the computation proceeds with the second transformer, with the same input string and success continuation as initially.

We need to adapt `satisfy` to the new definition of `Tr`:

```
satisfy :: (Char → Bool) → PP Char
satisfy p = K7 (Tr pr) (Tr pa) where
  pr k k' s c
    | p c = k (\s → k' s c) (s ++ [c])
    | otherwise = k' s c
  pa k k' (c:cs) u
    | p c = k (\cs _ → k' cs u) cs (u c)
  pa _ k' s u = k' s u
```

Unlike in Section 3, `satisfy` is now a total function. Printing and parsing both only succeed if the supplied predicate succeeds. Otherwise, we call the failure continuation `k'`, instead of leaving that case undefined.

4.2 Leads

Vertical composition opens up new possibilities, not just of failure. Consider again the type of `spec` in Example 3.1:

```
spec :: K7 Tr r r' where r' = Int → Char → r
```

We can view a cassette as an abstract machine with a tape of instructions and a stack. The types `r` and `r'` respectively give us the precondition and postcondition of the stack when the machine reads the tape in one direction and executes the sequence of instructions that if finds, represented by `spec`. Of course, since this paper is about invertible syntax, the `spec` can also be executed in the other direction, in which case the pre- and post- conditions are reversed.

In Section 5.1, we'll see how to define the generic `push` and `pop` operations one would expect to manipulate the stack. For now, consider that primitive descriptors like `char`, `digit` also affect the stack:

```
id               :: K7 Tr r r
char ∘ id        :: K7 Tr r (Char → r)
int ∘ char ∘ id  :: K7 Tr r (Int → Char → r)
```

We can read this sequence of primitive descriptors as instructions. From right to left, each instruction pushes a new type on the stack. In the opposite direction, from left to right, these instructions pop off the stack.

In this light, we can view quantifying over all `r` as a kind of frame rule: a descriptor of type `PP a` behaves the same no matter the state of the stack.

Some format descriptors simultaneously push and pop from the stack. Consider the following string transformer `consL`. It pushes the two types of the components of the `(:)`

constructor on the stack, after having popped the result type off the stack, or *vice versa*:

```
id          :: K7 Tr r r
consL ∘ id :: K7 Tr (a → [a] → r) ([a] → r)
```

On the parsing side, `consL` constructs a list using `(:)`. To do so, it pops the components off the stack and pushes the result on the stack. On the printing side, time flows in reverse. `consL` expects a list on the stack. It pops the list to deconstruct (i.e. pattern matches on) it, before pushing the components onto the stack if the list is non-empty.

We call such string transformers *leads*, by analogy to the lead-ins and lead-outs at the beginning and at the end of the tape in a cassette. In our setting, we have lead-ins on the printing side and lead-outs on the parsing side. Lead-ins and lead-outs have types of the following form, respectively:

```
Tr (b₁ → ... → bₙ → r)          (a → r)
Tr((b₁ → ... → bₙ → r) → t) ((a → r) → t)
```

These types generalize the types of printing and parsing string transformers from Section 3.1, respectively.

Intuitively,

- a *lead-in* deconstructs a value if possible and passes the components to its continuation, or fails otherwise;
- a *lead-out* constructs a value from components and passes that to its continuation. It never fails.

A pair of a lead-in and a lead-out is a lead. By convention, we name leads with an `*L` suffix. We can define a lead like `consL` by hand:

```
consL = K7 (Tr leadin) (Tr leadout) where
  leadin k k' s xs@(x:xs') =
    k (\s _ _ → k' s xs) s x xs'
  leadin _ k' s xs = k' s xs
  leadout k k' s u =
    k (\s _ → k' s u) s (\x xs' → u (x:xs'))
```

But more typically, we'll build leads from existing prisms, as in the next section.

4.3 Prisms

A *prism* is an optic used to work with sum types [Pickering et al. 2017]. Like a lens, it is a bidirectional structure, relating the value of a data type to its components. It offers a *review* operation to inject components into a sum type, and a *preview* to project these components. Reviewing always succeeds, whereas previewing can fail. Notice the similarity with leads: a lead-in is a preview and a lead-out is a review. Therefore, we can lift all prisms to lead string transformers:

```
prismL :: Prism' s a → K7 Tr (a → r) (s → r)
prismL l = K7 (Tr leadin) (Tr leadout) where
 leadin k k' s t = case preview l t of
   Nothing → k' s t
   Just x → k (\s _ → k' s t) s x
 leadout k k' s u =
   k (\s _ → k' s u) s (\x → u (review l x))
```

This is convenient because prisms are well understood and well supported. In practice, they are derived systematically, whether through meta- or generic programming [Adams and DuBuisson 2012; Kiss et al. 2018].

4.4 Derived String Transformers

We now have enough rope to implement a raft of new combinators. For lack of space, we'll demonstrate the use of leads to implement just two of them that are standard in Röjemo-style parser combinator libraries:

```
many :: PP a → PP [a]
many p = some p <> nilL

some :: PP a → PP [a]
some p = consL ∘ p ∘ many p
```

These higher-order descriptors repeat the given descriptor, collecting the result at each step. With these we can finally implement `int`:

```
int :: PP Int
int = isoL show read ∘ some (satisfy isDigit)
```

Recursive combinators like these crucially depend on the composition operator being lazy in both arguments, lest evaluation never terminate. This explains the irrefutable patterns in Section 3.1.

4.5 Case Study: The λ-Calculus

We defined `sprintf` and `sscanf` in Example 3.1. Analogues in the more general case, where we handle sums in addition to products, are given below:

```
pretty :: PP a → a → Maybe String
pretty (K7 (Tr f) _) =
  f (\_ → Just) (\_ _ → Nothing) ""

parse :: PP a → String → Maybe a
parse (K7 _ (Tr f')) s =
  f' (\_ _ x → Just x) (\_ _ → Nothing) s id
```

The initial success continuation of `pretty` throws away the failure continuation and returns just the result string, nothing otherwise. For `parse`, we throw away the failure continuation as well as the unparsed rest of the string in case of success, nothing otherwise.

We now have enough infrastructure in place to demonstrate the generalization of Danvy's format descriptors to context-free grammars. Figure 1 specifies the syntax of a small programming language, the pure λ-calculus, in the style of a grammar expressed in Backus-Naur form (BNF). We have one production, `term`, with several alternatives. For effect, we use `(⟶)`, a synonym for `(∘)`, to separate leads (which are pure string transformers) from the descriptor for

```
type Idnt = String
data Term =
  Var Idnt | Abs Idnt Term | App Term Term

varL = prismL _Var
absL = prismL _Abs ∘ pairL
appL = prismL _App ∘ pairL

parens p = lit "(" ∘ p ∘ lit ")"
sepSpace = lit " "
idnt = consL ⟶ letter ∘ many alphaNum

term :: PP Term
term = varL ⟶ idnt <>
       absL ⟶ lit "λ" ∘ idnt ∘ lit "." ∘ term <>
       appL ⟶ parens (term ∘ sepSpace ∘ term)
```

Figure 1. The syntax of the pure λ-calculus.

components. The leads in this grammar are constructed from ambient prisms derived automatically [Kiss et al. 2018].

At the prompt, parsing the self-application term gives the result we expect, and we can pretty print it back:

```
>>> parse term "λx.(x x)"
Just (Abs "x" (App (Var "x") (Var "x")))

>>> (parse term "λx. (x x)") ⩾> pretty term
Just "λx.(x x)"
```

5 A Monadic Solution

Expressive power is a limitation of the K7-style format descriptors. Parsers' control flow is static; they can't observe an intermediate result to change how the rest of the input is parsed. In effect, only context-free grammars can be parsed. While this is a standard limitation – parser generators also exhibit static control flow – we now turn to investigating how to parse languages generated by context-sensitive grammars, *e.g.* XML, in a single pass.

In this section, we seek to match the expressive power of `Parsing` when we take `Monad` as its superclass. In doing so, we will have to forgo some of the symmetry of cassettes. In exchange, we will build some reusable abstraction and gain the ability to reuse existing parsing and pretty printing libraries, instead of rolling our own.

5.1 Stacked Monads

As a starting point, let us turn to an established interface combining features of categories (like in Sections 3 and 4) and monads, namely *indexed monads* [Atkey 2009]. For the sake of completeness, Figure 2 lists the derived combinators from the `IxMonad` class that we'll be using.

```
class IxMonad m where
  return :: a → m r r a
  (⩾>) ::
    m r' r a → (a → m r'' r' b) → m r'' r b
```

In `m r' r a`, `r` and `r'` are the shapes of the stack before and after printing, respectively, while the parser has `a` as the type of its result, instead of pushing its result to the stack like in Sections 3 and 4. In effect, we're decoupling the covariant and the contravariant arguments, in a similar style to the profunctors of Section 1.

A pair of categories form a category. A cassette is then a pair of two categories: print and parse format descriptors. Similarly, a pair of indexed monads is an indexed monad, the detailed implementation of which can be found, together with all the code supporting this section, in Appendix A.

So we're looking for an indexed monad for printing, and another for parsing. As remarked in [Xia et al. 2019], the parsing case is easily solved: take any regular parser monad `m`, and lift it to an indexed monad by ignoring the stack:

```
newtype Fwd m r r' a = Fwd { unFwd :: m a }

instance Monad m ⇒ IxMonad (Fwd m) where
  return x = Fwd (return x)
  (Fwd a) ⩾> f = Fwd (a ⩾> (unFwd ∘ f))
```

We shall therefore turn, in the rest of this section, our attention to the much less obvious case of printers. Indeed, in order to make an (indexed) monad, printers can't ignore the return type, since monad lets you branch on the return value. The solution to this conundrum can be found, again, in [Xia et al. 2019]: make printers return the value they print.

Continuations as an Indexed Monad. To implement printers, we shall turn, again, to continuations. Answer-type modifying continuations can be packaged as an indexed monad. This was, in fact, one of the original motivations for the introduction of indexed monads [Atkey 2009]:

```
newtype Cont r r' a
  = Cont { runCont :: (a → r) → r' }

instance IxMonad Cont where
  return x = Cont \k → k x
  (Cont c) ⩾> f = Cont \k →
    c \x → runCont (f x) k
```

To connect this to what we've been doing so far, the type `Tr r r'` of Section 3.1 is isomorphic to a type of Kleisli arrows: `Tr r r' ≈ String → Cont r r' String`.

Abstracting the Stack. In order to implement leads, like in Section 4.2, we will need some way to access and modify the stack. Indexed monads, by themselves, don't give us any

```
(<$>) :: IxMonad m ⇒ (a → b) → m r r' a
      → m r r' b
f <$> ma = ma ≫= \a → return (f a)

(<*>) :: IxMonad m ⇒ m r' r (a → b) → m r'' r' a
      → m r'' r b
mf <*> ma = mf ≫= \f → ma ≫= \a → return (f a)
```

```
(<*) :: IxMonad m ⇒ m r' r a → m r'' r' ()
     → m r'' r a
ma <* mu = ((\a _ → a) <$> ma) <*> mu

(*>) :: IxMonad m ⇒ m r' r () → m r'' r' a
     → m r'' r a
mu *> ma = ((\_ a → a) <$> mu) <*> ma
```

Figure 2. Functorial and applicative combinators for indexed monads

access to the stack. We will need more primitives. Let continuations guide us once again. The quintessential primitive of `Cont` is `shift` [Danvy and Filinski 1990]:

```
shift :: ((a→r)→Cont k r' k) → Cont r r' a
shift f = Cont \k → runCont (f k) id
```

And, indeed, `shift` lets us manipulate the stack. For instance, we can implement a function to return the top of the stack:

```
pop :: Cont r (a → r) a
pop = shift \k → return (\a → k a)
```

However, `Fwd` can't implement pop, hence `Fwd` can't implement `shift`. Indeed, in `Fwd` the stack type is phantom. There isn't any stack to return a value from. That is, we can't have stack-manipulating functions have return values. Specializing `shift` to the unit type gives us just that, and forms the next step of our abstraction:

```
class Stacked m where
  shift_ :: (r → m k r' k) → m r r' ()

instance Stacked Cont where
  shift_ f = shift \k → f (k ())
```

The `Stacked` type class lets us implement all sorts of stack-manipulating operations. Note how, crucially, we can only implement pop_, a variant of pop which simply discards the value from the stack.

```
push :: (IxMonad m, Stacked m)
     ⇒ a → m (a → r) r ()
push a = shift_ \k → return (k a)

pop_ :: (IxMonad m, Stacked m) ⇒ m r (a→r) ()
pop_ = shift_ \k → return (\_a → k)

stack :: (IxMonad m, Stacked m)
      ⇒ (r → r') → m r r' ()
stack f = shift_ \k → return (f k)
```

The `stack` function, in particular, is a very flexible tool. It's worth paying attention to the fact that because of how the answer types `r` and `r'` are structured, a function `r → r'` really maps the stack `r'` to the stack `r`. For instance in

```
curryStack :: Cont ((a, b)→r) (a→b→r) ()
curryStack = stack \k a b → k (a, b)
```

the type `((a, b) → r) → (a → b → r)` is really the type `a → b → (a, b)` in CPS.

Together with, *mutatis mutandis*, the interface of Swierstra and Duponcheel [1996], we now have our complete abstraction:

```
class (Stacked m, IxMonad m) ⇒ Descr m where
  satisfy ::
    (Char → Bool) → m r (Char → r) Char
```

Effects. We have a complete interface of monadic format descriptors. But we don't actually know how to print yet. In Section 3 we used the type

```
String → Cont r r' String
```

as the type of printers. But, of course, this isn't a monad. We could try to fix up this type until we have a monad. But, since monads are all about effects, it's better to think of printing as an effect that we will add to our continuation monad.

It's tempting to try to encode effects with the usual continuation monad transformer:

```
newtype ContT m r r' a
  = ContT { runContT :: (a → m r) → m r' }

instance IxMonad (ContT m) where
  return x = ContT \k → k x
  (ContT c) ≫= f = ContT \k →
    c (\x → runContT (f x) k)
```

But `ContT m` isn't an instance of `Stacked`. In the below,

```
shift_ :: Monad m ⇒ (r → ContT m k r' k)
       → ContT m r r' ()
shift_ f = ContT \k → runCont (f _) return
```

we need a value of type `r` to fill the hole, but we cannot extract one from `k :: () → m r`. Furthermore, we cannot change `shift_` to take a monadic continuation either,

```
badShift_ :: Monad m ⇒ (m r → ContT m k r' k)
          → ContT m r r' ()
badShift_ f = ContT \k →
  runCont (f (k ())) return
```

as then we cannot, for instance, implement push anymore. Perhaps more concretely, how could we print a character with `ContT`? In the below,

```
satisfy :: ContT ((String,)) (Char → r) r Char
satisfy _ = ContT \(k :: Char→(String, r)) →
  (_ :: (String, Char → r))
```

how can we fill the hole? Per the type, it must be of the form `(printed,\c → _)`. That is, whatever is printed is independent of `c`. This isn't what we want. We would really want the hole to be of type `Char → (String, r)`. But this isn't a type that the monad transformer style can give us.

Let's look for our solution somewhere else: there's another, if less travelled, way of complementing continuations with effects: using comonads [Kmett 2011]. And this one enables us to implement `Stacked`.

```
class Comonad w where
  extract :: w a → a
  extend :: (w a → b) → w a → w b

newtype ContW w r r' a
  = ContW { runContW :: w (a → r) → r' }

instance Comonad w ⇒ IxMonad (ContW w) where
  return x = ContW \wk → extract wk x
  (ContW a) >>= f = ContW \wk →
      a (extend k' wk)
    where k' wk x = runContW (f x) wk

instance Comonad w ⇒ Stacked (ContW w) where
  shift_ f = ContW \wk →
    -- Continuation and its comonadic context
    -- are split, but both are passed to f.
    runContW (f (extract wk ()))
             (const id <$> wk)

yield :: (Comonad w)
      ⇒ (w r → r) → ContW w r r ()
yield eff = ContW \wk →
  (eff ((\k → k ()) <$> wk))
```

To recover the type `Tr` of Section 3, we can use the `Store` comonad:

```
newtype Store s a = Store (s → a, s)
```

Checking that `ContW (Store String)` is indeed isomorphic to `Tr` is left as an exercise to the reader.

However, in order to demonstrate the flexibility of this approach, and since printing doesn't need the full power of state-passing, we can turn instead to the `Traced` comonad:

```
newtype Traced m a
  = Traced { runTraced :: m → a }

instance Monoid m ⇒ Comonad (Traced m) where
  extract (Traced a) = a mempty
  extend f (Traced a) = Traced \m →
    f (Traced \m' → a (m <> m'))
```

Note that `∀ r. ContW (Traced m) r r a` is isomorphic to `(m, a)`. That is to say, `ContW` is more precise than `ContT`: `∀ r. ContT (m,) r r a` contains more computations than `(m, a)`. This is related to the observation that `ContT m` is a monad even if `m` itself isn't a monad. Key for us is the fact that the following two types are isomorphic,

```
ContW w (Char → r) r' a
Char → ContW w r r' a
```

which is precisely what we were calling for. This enables us to implement the `satisfy` function:

```
class Comonad w ⇒ ComTraced s w where
  trace :: s → w a → a

instance Monoid s ⇒
    ComTraced s (Traced s) where
  trace x (Traced f) = f x

instance ComTraced String w ⇒
    Descr (ContW w) where
  satisfy _ = pop >>= \c →
    yield (trace [c]) *> return c
```

We can opt for different pretty-printing libraries by varying the monoid `s`. We use `String` in our examples for the sake of simplicity, but a pretty-printing library in the style of [Wadler 2003] would use some `Doc` type instead. Of course, there is no conceptual difficulty in printing with an arbitrary pretty-printing library with `K7`-like combinators (reusing a parser library, on the other hand, doesn't seem possible).

And with this, we have our monadic format descriptors. Let's revisit Example 3.1 with stacked monads. Notice that because monads don't have multiple return values, `sscanf` isn't as flexible as in Section 3: we need to choose the type in which the result is packaged in `spec` instead of being able to wait until the call to `sscanf`. This is a mild instance of tuple troubles which is inherent to monads (and applicatives). It's usually fine because, contrary to the example of Section 2, it doesn't tend to create nested tuples. The full code for this section can be found in Appendix A.

Example 5.1. We construct our type of format descriptors by pairing print and parse format descriptors:

```
data (pr ⊗ pa) r r' a = pr r r' a ⊗ pa r r' a

type D = ContW (Traced String) ⊗ Fwd Pa
```

On the parser side, a value of type `Fwd Pa r r' a` is an action of an indexed monad, over a monad `Pa`. We can take `∀ r. Tr r (a → r)` as `Pa`, or just as simply:

```
newtype Pa a = Pa (String → (a, String))
```

`sprintf`, `sscanf` are given as follows:

```
sprintf :: D String r a → r
sprintf (ContW pr ⊗ _) = pr (Traced \s _ → s)
```

```
sscanf :: D r r' a → String → a
sscanf (_ ⊗ Fwd (Pa pa)) s = fst (pa s)
```

Our spec is as in Example 3.1, but this time it is not polyvariadic — we have to map a tuple constructor over it:

```
spec = (,,) <$>
  digit <* lit "-th character after "
  <*> char <* lit " is " <*> char
```

We can use spec for both printing and parsing:

```
>>> sprintf spec 5 'a' 'f'
"5-th character after a is f"

>>> sscanf spec "5-th character after a is f"
Just (5, 'a', f')
```

The inferred type of spec is,

```
spec :: (Descr m)
     ⇒ m r (Int → Char → Char → r)
           (Int, Char, Char)
```

5.2 Returning to Partiality

As a final part of our exploration, let us now handle failure and choice in the monadic context. For regular monads, this is the role of the `MonadPlus` class. In our setting, we add an extra `Monoid` superclass as in Section 4:

```
class (Stacked m, IxMonad m
      , ∀r r' a. Monoid (m r r' a))
       ⇒ Descr m where
  satisfy ::
    (Char → Bool) → m r (Char → r) Char
```

We might hope that we can use the comonad `w` to add partiality as an effect. However, this seems to be a dead end. Indeed, there is no comonad `W` such that `ContT W r r` is isomorphic to `Maybe` (or to `List` for that matter).

Therefore, like in Section 4, we use two continuations. Remarkably, the implementation of the indexed monad instance is exactly the same as that for `ContW`:

```
newtype Cont2W w r r' a = Cont2W
  { runCont2W :: w (a → r → r) → r' → r' }

instance Comonad w ⇒ IxMonad (Cont2W w) where
  return x = Cont2W \wk → extract wk x
  Cont2W a >>= f = Cont2W \wk →
      a (extend k' wk)
    where
      k' wk x = runCont2W (f x) wk
instance Comonad w ⇒
    Monoid (Cont2W w r r' a) where
  mempty = Cont2W \_ fl → fl
  (Cont2W a) <> (Cont2W b) = Cont2W \wk fl →
    a wk (b wk fl)
```

We will still use `Fwd` and `(⊗)`, as they preserve monoid. Their instances, and the code supporting this section can be found in Appendix B.

Iterating the CPS transform gives rise to the so-called CPS hierarchy [Danvy and Filinski 1990], and `Cont2W` is the (comonad-enriched) second iteration. There is, however, a bit of a design choice here. Indeed, `Cont2W` isn't the only type that we can give to said second iteration. For instance:

```
type C2 h r r' = (a → h → r) → h → r'
```

We could also give a type with four type parameters which is more general than both, but it's also rather hard to tame into an abstraction.

The practical implication of this choice is that `C2` doesn't support a choice monoid at every type (indeed `C2` doesn't even support `mempty` at every type). On the one hand, `C2` implements the same `shift` as Section 5.1[5]. On the other hand, `Cont2` does support choice at every type, but requires a different type for `shift`. So we need to modify the `Stacked` type class to account for the particular type of its continuations. `Fwd` and `(⊗)` are unaffected.

```
class IxMonad m ⇒ Stacked m where
  shift_ :: ((r → r) → r' → m k r' k)
         → m r r' ()

instance Comonad w ⇒ Stacked (Cont2W w) where
  shift_ f = Cont2W \wk k' →
    runCont2W (f (extract wk ())) k')
              ((\_k → \x _ → x) <$> wk) k'

push :: Stacked m ⇒ a → m (a → r) r ()
push x = shift_ \k k' →
  return (k (\_ → k') x)

pop_ :: Stacked m ⇒ m r (a → r) ()
pop_ = shift_ \k k' → return (\a → k (k' a))

stack :: Stacked m ⇒ (r' → r → r')
      → (r' → r) → m r r'
stack f u =
  shift_ \k k' → return (f k' (k (u k')))
```

We can see two things happening in the type of `stack`. First, the function `f` takes an additional argument of type `r'`. This lets us declare a failure to `stack` — useful to implement leads. Then, `stack` takes an extra argument `u`: it's a "stack unrolling" function. The role of `u` is to restore the stack to its initial state in case of failures. To illustrate this, implement the lead for the `(:)` constructor explicitly:

```
consL :: Stacked m
      ⇒ m (a → [a] → r) ([a] → r)
           (a → [a] → [a])
consl = stack uncons unroll *> return (:)
```

[5] In fact, `C2 h` can be defined as `ContW (StoreT h)`.

```
  where
    uncons k' k (x : xs) = k x xs
    uncons k' k [] = k' []

    unroll k' x xs = k' (x:xs)
```

This extra bookkeeping seems to be the price to pay to have a monoid at every `Cont2W w r r' a`, to match the usual expressive power of `MonadPlus`.

Following the same recipe as `consL`, and just like in Section 4.3, we can implement a lead for any prism:

```
prismL :: Stacked m ⇒ Prism' s a
       → m (a → r) (s → r) (a → s)
prismL l = stack rev u *> return (review l)
  where
    rev k' k t = case preview l t of
      Nothing → k' t
      Just x → k x

    u k' a = k' (review l a)
```

Finally, to parse a string, we apply the format descriptor, throwing away what remains of the string. The implementation of pretty printing is analogous to Section 4.5:

```
type D2 = Cont2W (Traced String) ⊗ Fwd Pa

parse :: D2 r r' b → String → Maybe b
parse (_ ⊗ Fwd (Pa pa)) s = fst <$> pa s

pretty
  :: D2 (Maybe String) (a → Maybe String) b
  → a → Maybe String
pretty (Cont2W pr ⊗ _) =
  pr (Traced \s _ _ → Just s) (\_ → Nothing)
```

And this is it. We now have the material to build the λ-calculus example of Section 4.5. We do so in Figure 3. Notice the following differences.

Where descriptors as cassettes are glued together uniformly using `(∘)`, descriptors as stacked monads rely on a variety of operators to selectively drop the return value on the left or right-hand sides. This is needed because we no longer have true nullary descriptors. Here, `lit` returns `()`, when it previously returned nothing at all.

A more instructive observation is that in the `App` case we write

```
parens (appL <*> term <* sepSpace <*> term)
```

where in Section 4.5 we could write

```
appL ⟶ parens (term ∘ sepSpace ∘ term)
```

It is not possible to write the same with stacked monads:

```
appL <*> parens (term <* sepSpace <*> term)
  -- Type error
```

We could work around that by tupling and untupling. This is another instance of tuple troubles lurking right around the corner in the stacked monad style. There is an as yet unresolved tension here, between the convenience of polyvariadic descriptors and the expressive power of context sensitivity afforded by monads.

6 Assessment

Danvy [1998] sought to illustrate the "expressive power of ML" by implementing `printf` without dependent types. The technique is furthermore a demonstration that *continuations get you out of tuple troubles*. This is again demonstrated with the construction of `scanf` by Asai et al. [2011] along the same lines, or by Rhiger [2009] delightfully applying the same technique to first-class patterns for pattern-matching.

Hinze [2003] demonstrates a direct-style solution, in a language richer than ML, with type classes and functional dependencies to build a type-level function over types. Hinze relies on functor composition, which isn't associative in Haskell. Therefore, similar troubles to tuple troubles arise. In particular, `a <> b` may or may not type check depending on how functor compositions were nested in the respective types of `a`, `b`.

The original motivation of the arrows of Hughes [2000] was to capture the essence of the parser combinators of Swierstra and Duponcheel [1996]. But in the end, Hughes uses only a weaker fragment [Lindley et al. 2011] of arrows that Mcbride and Paterson [2008] identify as `Alternative`. In general though, programming with nested tuples is at the core of the arrow style [Hughes 2004], to the point that Paterson [2001] proposes dedicated syntactic sugar to hide the tupling. Alimarine et al. [2005] uses this syntactic sugar to implement combinators as invertibility-preserving pairs, much like our format descriptors. More recent developments also adopt this style [Xie et al. 2025]. Rendel and Ostermann [2010] achieve much the same in applicative style, but again perform monoidal aggregation via nested pairs.

To the best of our knowledge, the `K7` descriptors of Sections 3 and 4 is the only set of combinators that avoids tuple troubles entirely, but like the above cited works, at the price of only parsing context-free grammars. Xia et al. [2019] achieve greater expressivity, still with nested pairs. Even the stacked monads of Section 5 exhibit some tuple troubles. More research would be needed to find a tuple-trouble-free context-sensitive set of format descriptors. In particular, it feels like what we are grasping at there is a notion of multiple-result monads yet to be developed, akin to the multiple-result functions of Section 3. In essence, it would appear that the "potential equivalence" between the CPS hierarchy and monads that Danvy and Filinski [1990] were musing about has not, 35 years later, been fully resolved.

Xia et al. [2019] introduce a notion of *partial monadic profunctor* (PMP) to give an interface to format descriptors:

```
parens p = lit "(" *> p <* lit ")"
sepSpace = lit " "
idnt = consL <*> letter <*> many alphaNum
```

```
term :: (Descr m) ⇒ m r (Term → r) Term
term =
  varL <*> idnt <>
  absL <* lit "λ" <*> idnt <* lit "." <*> term <>
  parens (appL <*> term <* sepSpace <*> term)
```

Figure 3. λ-calculus with stacked monads

```
term :: Biparser Term Term
term = (Var <$> idnt) `uponr` _Var <>
       (pure Abs <* lit "λ" <*> idnt `upon'` fst <* lit "." <*> term `upon'` snd) `uponr` _Abs <>
       parens ((App <$> term `upon'` fst <* sepSpace <*> term `upon'` snd) `uponr` _App)
```

```
upon' :: PMP p ⇒ p b c → (a → b) → p a c
upon' p f = p `upon` (Just ∘ f)
```

```
uponr :: PMP p ⇒ p a b → Prism' s a
uponr p pat = p `upon` review pat
```

Figure 4. λ-calculus with partial monadic profunctors

```
class (∀a. MonadPlus (p a)) ⇒ PMP p where
  upon :: p a b → (a' → Maybe a) → p a' b
```

The intent is that a `p a b` can print an `a` and parse into a `b`. This design was the main inspiration behind the stacked monads of Section 5.1, which can be summarised as applying a continuation-based stack to the contravariant argument to overcome the inherent tuple troubles of the `Comonoidal` type class (as analysed in Section 2). Format descriptors implemented with stacked monad are significantly less verbose. Compare for instance the syntax of the λ-calculus with PMPs in Figure 4, with that of Figure 3.

Note that `PMP`s are, indeed, comonoidal as another equivalent type for `comult` is

```
comult'' :: Comonoidal f ⇒ f r → f r → f r
```

There's no tuple in this interface, but, as is apparent in Figure 4, `comult''` only replaces automatic nested tupling with the need to manually re-adjust the base type everywhere; *e.g.* pairing each descriptor with a call to `upon`.

It's worth noting, however, that PMPs can implement failure and choice abstractly using a monad transformer (`MaybeT`), whereas this doesn't seem possible for stacked monads as discussed in Section 5.2.

Further Considerations. The developments presented in this paper ignore several concerns that are important in practice, like good error messages, precise location of the cause of these errors, potential space leaks due to unrestricted backtracking [Leijen and Meijer 2001], threading user state, and controlling the printer's layout. All of the above could in principle be worked into our framework. But a more practical and maintainable approach is to reuse off-the-shelf, mature parser and pretty printing combinator libraries. This is demonstrated in the libraries accompanying this paper[6]. We don't know, however, how to reuse parsing libraries in the `K7` style. Once again there's a tension left to be resolved between `K7`'s tuple trouble freedom and practical needs.

Another venue for further investigation is the relation between the comonads of Section 5 and delimited control. Indeed, Asai [2009] demonstrates how Danvy's `printf` can be reinterpreted without explicit continuation passing in a (typed, answer-type-modifying) language with `shift` and `reset`. Yet, if passing continuations explicitly lets us wrap the continuation in a comonad, how could we do the same in a language with delimited control instead? In this way, Asai's solution could extend, generically, to more applications than just printing.

Beyond the applications to format descriptors, we might imagine that stacked functors or categories can find other uses in functional programming. Xia et al. [2019] also apply PMPs to defining multiple-focus lenses, Goldstein et al. [2023] proposes several applications to random generation. Presumably, wherever PMPs are useful, stacked monads should apply too. But beyond even bidirectional programming, any interface using contravariant functors or profunctors as an abstraction is liable to have some amount of tuple trouble. We argue that even though they aren't as grounded in mathematical practice, stacked functors and categories may be more natural in functional programming languages. Therefore, we look forward to seeing whether such interfaces can be revisited with stacks to good effect.

References

Michael D. Adams and Thomas M. DuBuisson. 2012. Template your boilerplate: using template haskell for efficient generic programming. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell*

[6] https://github.com/mboes/cassette/ and https://github.com/tweag/pup

'12). Association for Computing Machinery, New York, NY, USA, 13–24. doi:10.1145/2364506.2364509

Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. 2005. There and back again: arrows for invertible programming. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell* (Tallinn, Estonia) *(Haskell '05)*. Association for Computing Machinery, New York, NY, USA, 86–97. doi:10.1145/1088348.1088357

Kenichi Asai. 2009. On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation* (2009). doi:10.1007/s10990-009-9049-5

Kenichi Asai, Oleg Kiselyov, and Chung-chieh Shan. 2011. Functional un|unparsing. *Higher-Order and Symbolic Computation* (2011). doi:10.1007/s10990-012-9087-2

Robert Atkey. 2009. Parameterised notions of computation. *Journal of Functional Programming* 19, 3–4 (2009), 335–376. doi:10.1017/S095679680900728X

Olivier Danvy. 1996. Type-directed partial evaluation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 242–257. doi:10.1145/237721.237784

Olivier Danvy. 1998. Functional unparsing. *Journal of Functional Programming* 8, 6 (1998), 621–625. doi:10.1017/S0956796898003104

Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. Association for Computing Machinery, New York, NY, USA, 151–160. doi:10.1145/91556.91622

Harrison Goldstein, Samantha Frohlich, Meng Wang, and Benjamin C. Pierce. 2023. Reflecting on Random Generation. *Proc. ACM Program. Lang.* 7, ICFP, Article 200 (Aug. 2023), 34 pages. doi:10.1145/3607842

Ralf Hinze. 2003. Formatting: a class act. *Journal of Functional Programming* 13, 5 (Sept. 2003), 935–944. doi:10.1017/S0956796802004367

John Hughes. 2000. Generalising monads to arrows. *Sci. Comput. Program.* 37, 1–3 (May 2000), 67–111. doi:10.1016/S0167-6423(99)00023-4

John Hughes. 2004. Programming with arrows. In *Proceedings of the 5th International Conference on Advanced Functional Programming* (Tartu, Estonia) *(AFP'04)*. Springer-Verlag, Berlin, Heidelberg, 73–129. doi:10.1007/11546382_2

Csongor Kiss, Matthew Pickering, and Nicolas Wu. 2018. Generic deriving of generic traversals. *Proc. ACM Program. Lang.* 2, ICFP, Article 85 (July 2018), 30 pages. doi:10.1145/3236780

Edward Kmett. 2011. Monads from Comonads. https://web.archive.org/web/20241213000619/http://comonad.com/reader/2011/monads-from-comonads/.

Daan Leijen and Erik Meijer. 2001. Parsec: Direct Style Monadic Parser Combinators for the Real World. (2001). Draft manuscript.

Sam Lindley, Philip Wadler, and Jeremy Yallop. 2011. Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous. *Electron. Notes Theor. Comput. Sci.* 229, 5 (March 2011), 97–117. doi:10.1016/j.entcs.2011.02.018

Conor Mcbride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (Jan. 2008), 1–13. doi:10.1017/S0956796807006326

Ross Paterson. 2001. A new notation for arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (Florence, Italy) *(ICFP '01)*. Association for Computing Machinery, New York, NY, USA, 229–240. doi:10.1145/507635.507664

Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. 2017. Profunctor Optics: Modular Data Accessors. *The Art, Science, and Engineering of Programming* 1, 2 (April 2017). doi:10.22152/programming-journal.org/2017/1/7

Tillmann Rendel and Klaus Ostermann. 2010. Invertible syntax descriptions: unifying parsing and pretty printing. *SIGPLAN Not.* 45, 11 (Sept. 2010), 1–12. doi:10.1145/2088456.1863525

Morten Rhiger. 2009. Type-safe pattern combinators. *Journal of Functional Programming* 19, 2 (2009), 145–156. doi:10.1017/S0956796808007089

Niklas Röjemo. 1995. Highlights from nhc—a space-efficient Haskell compiler. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*. Association for Computing Machinery, New York, NY, USA, 282–292. doi:10.1145/224164.224217

S. Doaitse Swierstra and Luc Duponcheel. 1996. Deterministic, Error-Correcting Combinator Parsers. In *Advanced Functional Programming, Second International School-Tutorial Text*. Springer-Verlag, Berlin, Heidelberg, 184–207.

Philip Wadler. 2003. A prettier printer. *The Fun of Programming, Cornerstones of Computing* (2003), 223–243.

Li-yao Xia, Dominic Orchard, and Meng Wang. 2019. Composing Bidirectional Programs Monadically. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 147–175.

Ruifeng Xie, Tom Schrijvers, and Zhenjiang Hu. 2025. Biparsers: Exact Printing for Data Synchronisation. *Proc. ACM Program. Lang.* 9, POPL, Article 74 (Jan. 2025), 27 pages. doi:10.1145/3704910

A Full Linear Scanning Example with Stacked Monads

```
class (Functor w) ⇒ Comonad w where
  extract :: w a → a
  extend :: (w a → b) → w a → w b

newtype Traced m a = Traced {runTraced :: m → a}
  deriving (Functor)

instance (Monoid m) ⇒ Comonad (Traced m) where
  extract (Traced a) = a mempty
  extend f (Traced a) = Traced \m →
    f (Traced \m' → a (m <> m'))

class (Comonad w) ⇒ ComTraced m w where
  trace :: m → w a → a

instance (Monoid m) ⇒ ComTraced m (Traced m) where
  trace x (Traced f) = f x

newtype Pa a
  = Pa {runPa :: String → (a, String)}
  deriving (Functor)

instance Monad Pa where
  -- return a = Pa \s -> Just (a, s)
  (Pa p) >>= f = Pa \s →
    let (a, s') = p s
     in runPa (f a) s'

instance Applicative Pa where
  pure a = Pa \s → (a, s)
  (<*>) = ap

data (f ⊗ g) r r' a
  = (⊗) {ifst :: (f r r' a), isnd :: (g r r' a)}
```

```
instance
  (IxMonad f, IxMonad g) ⇒
  IxMonad (f ⊗ g)
  where
  return x = return x ⊗ return x
  ~(l ⊗ r) >>= f =
    (l >>= (ifst ∘ f)) ⊗ (r >>= (isnd ∘ f))

newtype Fwd m r r' a = Fwd {unFwd :: m a}

instance (Monad m) ⇒ IxMonad (Fwd m) where
  return x = Fwd (return x)
  (Fwd a) >>= f = Fwd (a >>= (unFwd ∘ f))

class Stacked m where
  shift_ :: (r → m k r' k) → m r r' ()

push :: (IxMonad m, Stacked m)
     ⇒ a → m (a → r) r ()
push a = shift_ \k → return (k a)

pop_ :: (IxMonad m, Stacked m) ⇒ m r (a → r) ()
pop_ = shift_ \k → return (\_a → k)

stack :: (IxMonad m, Stacked m)
      ⇒ (r → r') → m r r' ()
stack f = shift_ \k → return (f k)

newtype Cont r r' a
  = Cont {runCont :: (a → r) → r'}

instance IxMonad Cont where
  return x = Cont \k → k x
  (Cont c) >>= f = Cont \k →
    c (\x → runCont (f x) k)

instance Stacked Cont where
  shift_ f = shift \k → f (k ())

shift :: ((a → r) → Cont k r' k) → Cont r r' a
shift f = Cont \k → runCont (f k) id

pop :: Cont r (a → r) a
pop = shift \k → return (\a → k a)

instance
  (Stacked f, Stacked g) ⇒
  Stacked (f ⊗ g)
  where
  shift_ f = (shift_ (ifst ∘ f) ⊗ shift_ (isnd ∘ f))

instance (Monad m) ⇒ Stacked (Fwd m) where
  shift_ _f = Fwd (return ())

class (Stacked m, IxMonad m) ⇒ Descr m where
  satisfy :: (Char → Bool) → m r (Char → r) Char

instance (Descr f, Descr g) ⇒ Descr (f ⊗ g) where
  satisfy p = satisfy p ⊗ satisfy p

instance Descr (Fwd Pa) where
  satisfy p = Fwd (Pa go)
    where
      go (c : s) | p c = (c, s)

newtype ContT m r r' a
  = ContT {runContT :: (a → m r) → m r'}

instance IxMonad (ContT m) where
  return x = ContT \k → k x
  (ContT c) >>= f = ContT \k →
    c \x → runContT (f x) k

lift :: (Monad m) ⇒ m a → ContT m r r a
lift act = ContT \k → act >>= k

newtype ContW w r r' a
  = ContW {runContW :: w (a → r) → r'}

shiftw :: (Comonad w)
       ⇒ ((a → r) → ContW w k r' k)
       → ContW w r r' a
shiftw f = ContW \wk →
  runContW (f (extract wk)) (const id <$> wk)

popw :: (Comonad w) ⇒ ContW w r (a → r) a
popw = shiftw \k → return (\a → k a)

instance (Comonad w) ⇒ IxMonad (ContW w) where
  return x = ContW \wk → extract wk x
  (ContW a) >>= f = ContW \wk → a (extend k' wk)
    where
      k' wk x = runContW (f x) wk

instance (Comonad w) ⇒ Stacked (ContW w) where
  shift_ f = ContW \wk →
    -- Notice how the continuation and its
    -- comonadic context are split, but both
    -- are passed to f.
    runContW (f (extract wk ())) (const id <$> wk)

yield :: (Comonad w)
      ⇒ (w r → r) → ContW w r r ()
yield eff = ContW \wk →
  (eff ((\k → k ()) <$> wk))

instance (ComTraced String w)
  ⇒ Descr (ContW w) where
  satisfy _ =
    popw >>= \c →
      yield (trace [c]) *> return c
```

```
type D = ContW (Traced String) ⊗ Fwd Pa

lit :: (Descr m) ⇒ String → m r r ()
lit [] = return ()
lit (c : cs) =
  push c *> satisfy (== c) ≫= \_ → lit cs

char :: (Descr m) ⇒ m r (Char → r) Char
char = satisfy (const True)

digit :: (Descr m) ⇒ m r (Int → r) Int
digit = return (\c → read [c]) <*
        stack (\k → k ∘ head ∘ show) <*>
        satisfy isDigit

spec :: (Descr m)
     ⇒ m r (Int → Char → Char → r)
           (Int, Char, Char)
spec = (,,) <$>
  digit <* lit "-th character after "
  <*> char <* lit "is" <*> char

-- | Ex:
-- >>> sprintf spec 5 'a' 'f'
-- "5-th character after a is f"
sprintf :: D String r a → r
sprintf (ContW pr ⊗ _) = pr (Traced (\s _ → s))

-- | Ex:
-- >>> sscanf spec "5-th character after a is f"
-- Just (5, 'a', f')
sscanf :: D r r' a → String → a
sscanf (_ ⊗ Fwd (Pa pa)) s = fst (pa s)
```

B Full λ-Calculus Example with Stacked Monads

```
newtype Pa a
  = Pa {runPa :: String → Maybe (a, String)}
  deriving (Functor)

instance Monad Pa where
  -- return a = Pa \s -> Just (a, s)
  (Pa p) ≫= f = Pa \s → do
    ~(a, s') <- p s
    runPa (f a) s'

instance Applicative Pa where
  pure a = Pa \s → Just (a, s)
  (<*>) = ap

instance MonadPlus Pa

instance Alternative Pa where
  empty = Pa \_ → empty
  (Pa pa1) <|> (Pa pa2) = Pa \s → pa1 s <|> pa2 s

class (Stacked m, IxMonad m
      , ∀r r' a. Monoid (m r r' a))
      ⇒ Descr m where
  satisfy :: (Char → Bool) → m r (Char → r) Char

instance (Descr f, Descr g) ⇒ Descr (f ⊗ g) where
  satisfy p = satisfy p ⊗ satisfy p

instance Descr (Fwd Pa) where
  satisfy p = Fwd (Pa go)
    where
      go (c : s) | p c = Just (c, s)
      go _ = Nothing

newtype Cont2W w r r' a = Cont2W
  {runCont2W :: w (a → r → r) → r' → r'}

instance (Comonad w) ⇒ IxMonad (Cont2W w) where
  return x = Cont2W \wk → extract wk x
  Cont2W a ≫= f = Cont2W \wk → a (extend k' wk)
    where
      k' wk x = runCont2W (f x) wk

instance (Comonad w)
  ⇒ Monoid (Cont2W w r r' a) where
  mempty = Cont2W \_ fl → fl
  (Cont2W a) <> (Cont2W b) = Cont2W \wk fl →
    a wk (b wk fl)

shiftw :: (Comonad w)
       ⇒ ((a → r → r) → r' → Cont2W w k r' k)
       → Cont2W w r r' a
shiftw f = Cont2W \wk k' →
  runCont2W (f (extract wk) k')
            (const (\k _ → k) (<$>) wk) k'

pop :: (Comonad w) ⇒ Cont2W w r (a → r) a
pop = shiftw \k k' → return (\a → k a (k' a))

instance (ComTraced String w)
  ⇒ Descr (Cont2W w) where
  satisfy _ =
    pop ≫= \c →
      yield (trace [c]) *> return c

yield :: (Comonad w)
      ⇒ (w r → r) → Cont2W w r r ()
yield eff = Cont2W \wk k' →
  eff ((\k → k () k') <$> wk)

instance (MonadPlus m)
```

```
  ⇒ Monoid (Fwd m r r' a) where
  mempty = Fwd empty
  (Fwd a) <> (Fwd b) = Fwd (a <|> b)

instance MonadPlus Prs

instance Alternative Prs where
  empty = Prs \_ → empty
  (Prs pa1) <|> (Prs pa2) = Prs \s →
    pa1 s <|> pa2 s

instance
  (Monoid (f r r' a), Monoid (g r r' a)) ⇒
  Monoid ((f ⊗ g) r r' a)
  where
  mempty = mempty <> mempty
  ~(fl ⊗ fr) <> ~(gl ⊗ gr) =
    (fl <> gl) ⊗ (fr <> gr)

class (IxMonad m) ⇒ Stacked m where
  shift_ ::
    ((r → r) → r' → m k r' k) →
    m r r' ()

instance (Comonad w) ⇒ Stacked (Cont2W w) where
  shift_ f = Cont2W \wk k' →
    runCont2W
      (f (extract wk ()) k')
      ((\_k → \x _ → x) <$> wk)
      k'

push :: (Stacked m) ⇒ a → m (a → r) r ()
push x = shift_ \k k' → return (k (\_ → k') x)

pop_ :: (Stacked m) ⇒ m r (a → r) ()
pop_ = shift_ \k k' → return (\a → k (k' a))

stack ::
  (Stacked m) ⇒
  (r' → r → r') →
  (r' → r) →
  m r r' ()
stack f u =
  shift_ \k k' → return (f k' (k (u k')))

instance (Monad m) ⇒ Stacked (Fwd m) where
  shift_ _f = Fwd (return ())

instance (Stacked f, Stacked g)
  ⇒ Stacked (f ⊗ g) where
  shift_ f =
    (shift_ (\k k' → ifst (f k k'))) ⊗
    (shift_ (\k k' → isnd (f k k')))

consL ::
  (Stacked m) ⇒
  m (a → [a] → r)
    ([a] → r)
    (a → [a] → [a])
consL = stack uncons unroll *> return (:)
  where
    uncons k' k (x : xs) = k x xs
    uncons k' k [] = k' []

    unroll k' x xs = k' (x : xs)

data Prism' s a
  = Prism' { review :: a → s
           , preview :: s → Maybe a }

prismL ::
  (Stacked m) ⇒
  Prism' s a →
  m (a → r) (s → r) (a → s)
prismL l = stack rev u *> return (review l)
  where
    rev k' k t = case preview l t of
      Nothing → k' t
      Just x → k x

    u k' a = k' (review l a)

some :: (Descr m)
     ⇒ (∀r. m r (a → r) a)
     → m r' ([a] → r') [a]
some p = consL <*> p <*> many p

many :: (Descr m)
     ⇒ (∀r∘ m r (a → r) a)
     → m r' ([a] → r') [a]
many p = some p <> (pop_ *> return [])

lit :: (Descr m) ⇒ String → m r r ()
lit [] = return ()
lit (c : cs) =
  push c *> $satisfy (== c) >>= \_ → lit cs

letter =
  satisfy (\c → isLetter c && isAscii c)

alphaNum =
  satisfy (\c → isAlphaNum c && isAscii c)

parens :: D2 r r' a → D2 r r' a
parens p = lit "(" *> p <* lit ")"

sepSpace = lit " "

idnt = consL <*> letter <*> many alphaNum
```

```
type Idnt = String

data Term =
  Var Idnt | Abs Idnt Term | App Term Term
  deriving (Show)

type D2 = Cont2W (Traced String) ⊗ Fwd Pa

term :: D2 r (Term → r) Term
term =
  varL <*> idnt <>
  absL <* lit "λ" <*> idnt <* lit "." <*> term <>
  parens (appL <*> term <* sepSpace <*> term)

varL :: (Descr m)
     ⇒ m (Idnt → r) (Term → r) (Idnt → Term)
varL = prismL _Var
  where
    _Var = Prism'
      { review = Var
      , preview = \case
          Var x → Just x
          _ → Nothing }

absL :: (Descr m)
     ⇒ m (Idnt → Term → r)
         (Term → r)
         (Idnt → Term → Term)
absL =
  stack (\k' k → \case Abs x u → k x u; t → k' t)
        (\k x u → k (Abs x u))
  *> return Abs

appL :: (Descr m)
     ⇒ m (Term → Term → r)
         (Term → r)
         (Term → Term → Term)
appL =
  stack (\k' k → \case App u v → k u v; t → k' t)
        (\k u v → k (App u v))
  *> return App

-- |
-- >>> parse term "λx.(x x)"
-- Just (Abs "x" (App (Var "x") (Var "x")))
parse :: D2 r r' b → String → Maybe b
parse (_ ⊗ Fwd (Pa pa)) s = fst <$> pa s

-- |
-- >>> (parse term "λxx.(x x)") >>= pretty term
-- Just "λx.(x x)"
pretty :: D2 (Maybe String) (a → Maybe String) b
       → a → Maybe String
pretty (Cont2W pr ⊗ _) =
  pr (Traced (\s _ _ → Just s)) (\_ → Nothing)
```

Received 2025-06-09; accepted 2025-07-31