Functional Pearl: Replaying the stack for parsing and pretty printing

Mathieu Boespflug
McGill University
mboes@cs.mcgill.ca

Abstract

Modulo inessential details, parsers and pretty printers, to and from algebraic datatypes, offer an uncanny resemblance and yet are all too often defined separately, in gross violation of the "don't repeat yourself" principle. We present a family of reversible parser/printer combinators that allows one to define both at once in a type-safe manner, compositionally and without any need for a preprocessor or program generator, in HASKELL'98 + rank 2 types.

1. Introduction

This story is an exercise in symmetry.

A full-fledged program, if it must be of any use at all, effects change to the World around it. This change, to the programmer often amounts to "data goes in, data goes out". Much as the scientist cleans and prepares collected data in order to process it, so does the program: we recover the structure of the *in* data by parsing it, and we package the *out* data by printing it, optionally in a pretty form, before shipping it off. So much is apparent from the type of one of the early functions in a program, as well as the type of one of the later ones:

$$\mathsf{parse} :: \mathsf{String} \to D$$
$$\mathsf{print} :: D \to \mathsf{String}$$

where D is the type we choose to confer to data in structured form. But parsing may not consume all of the input, here represented as a string, and it might also fail to actually recover any meaningful structure from the input data. A more appropriate signature for parse is therefore:

$$\mathsf{parse} :: \mathsf{String} \to (\mathsf{Maybe}\ D, \mathsf{String})$$

which we might as well change to

$$\mathsf{parse} :: \mathsf{String} \to \mathsf{Maybe}\ (D, \mathsf{String})$$

because we're not interested in what remains of the input string if the parser fails. But by now we have lost the symmetry with the type of print.

Structured data will in general be classified by distinct types — let then the following type macro capture the common pattern in the types of all the parsing functions we might introduce:

$$\mathbf{type}\ \mathsf{Parser}\ a = \mathsf{String} \to \mathsf{Maybe}\ (a, \mathsf{String})$$

We can compose parse or any other parsing function of the same type with the $\gg$ combinator, yielding a new parser that parses two consecutive pieces of data and returns the last one:

$$\begin{aligned}&(\gg) :: \mathsf{Parser}\ a \to \mathsf{Parser}\ b \to \mathsf{Parser}\ b\\ &p_1 \gg p_2 = (p_2 \circ \mathsf{snd})\ \tilde{\circ}\ p_1\\ &\quad \mathbf{where}\ (\tilde{\circ})\ f\ g\ s \mid \mathsf{Just}\ s' \leftarrow g\ s = f\ s'\\ &\qquad\qquad\qquad\quad \mid \mathsf{otherwise} = \mathsf{Nothing}\end{aligned}$$

The general form of parser composition allows the result of the first parser to depend on that of the second. That is, the second argument is a function from the result to a parser:

$$\begin{aligned}&(\ggg) :: \mathsf{Parser}\ a \to (a \to \mathsf{Parser}\ b) \to \mathsf{Parser}\ b\\ &p_1 \ggg f = \mathsf{uncurry}\ f\ \tilde{\circ}\ p_1\end{aligned}$$

Sometimes, a parser can produce an output of the required type without consuming any part of the input string. Such a parser is said to be pure because it has no effect on the input string:

$$\begin{aligned}&\mathsf{pure} :: a \to \mathsf{Parser}\ a\\ &\mathsf{pure}\ x = \lambda s \to \mathsf{Just}\ (x, s)\end{aligned}$$

The Parser type constructor forms a monad, with the $(\ggg)$ combinator as the bind of the monad and pure as the unit. In fact, the Parser monad composes two effects: state and failure. This can be made explicit by expressing Parser as the application of the two standard ErrorT and StateT monad transformers on top of the identity monad (the monad with no effects):

$$\mathbf{type}\ \mathsf{Parser}\ a = \mathsf{ErrorT}\ (\mathsf{StateT}\ \mathsf{String}\ (\mathsf{Identity}\ a))$$

Piecing parsers together using combinators is convenient because it means we can construct bigger parsers compositionally from existing ones. Just how we construct the bigger parsers is guided by the grammar of the data we are trying to parse.

But we have now headed even further astray from the simple type we started with and lost any kind of symmetry with the type of print along the way. Besides, writing down a parser of the above type doesn't magically afford us a printer: we still have to write that one separately, even though how to compositionally construct a printer is entirely determined by the same grammar that determines the construction of the parser.

In essence, the effect of a parser constructed with the above combinators on a given input string could be described using a tape, consisting of a sequence of primitive instructions. It would be nice if, to obtain a printer, we could just play that tape in reverse!

That's the topic of the next section.

2. How to play a cassette tape

Compact cassettes of old consisted of two miniature spools, between which was wound a plastic magnetized tape. This tape con-

sisted of two tracks, the A-side and the B-side. Playing the tape in one direction accessed the first track — flipping the tape to play it in the reverse direction accessed the second track. Such tapes live on in the virtual world that useful programs effect change upon — if we define them.

$$\mathbf{type}\ a \leftarrow b = b \to a$$
$$\mathbf{data}\ \mathsf{K7}\ a\ b\ c\ d = \mathsf{K7}\ \{\mathsf{sideA} :: a \to b, \mathsf{sideB} :: c \leftarrow d\}$$

Avoiding the margins calls for brevity in our naming, which is justified by either 1) the German spelling *Kassette* where 7 letters follow the K, or 2) the phonetization of the word using French pronunciation of numerals and letters, according to whichever side of the Rhine one is most accustomed to. Either way, a compact cassette consists of two sides, labelled accordingly.

Tapes of any type can be flipped, so that the A-side becomes the B-side and *vice versa*:

$$\mathsf{flip} :: \mathsf{K7}\ a\ b\ c\ d \to \mathsf{K7}\ d\ c\ b\ a$$
$$\mathsf{flip}\ (\mathsf{K7}\ x\ y) = (\mathsf{K7}\ y\ x)$$

Playing a tape built compositionally from existing tapes is simply a matter of pressing play:

$$\mathsf{play} :: \mathsf{K7}\ a\ b\ c\ d \to a \to b$$
$$\mathsf{play}\ k7\ x = \mathsf{sideA}\ k7\ x$$

One can of course play the B-side by flipping the cassette first.

Another useful thing to do with cassettes is to make new ones by splicing the tapes of existing ones:

$$(\otimes) :: \mathsf{K7}\ b\ c\ c'\ b'$$
$$\quad \to \mathsf{K7}\ a\ b\ b'\ a'$$
$$\quad \to \mathsf{K7}\ a\ c\ c'\ a'$$
$$\sim(\mathsf{K7}\ f\ f') \otimes \sim(\mathsf{K7}\ g\ g') = \mathsf{K7}\ (f \circ g)\ (g' \circ f')$$

For convenience later, we can make also more of the same glue, that works in reverse and with lower precedence:

$$k7_1 \longrightarrow k7_2 = k7_2 \otimes k7_1$$

Splicing is associative, and has a neutral element. Indeed, in the special case where both tracks are inverses of each other, cassettes form a category under splicing[1]:

$$\mathbf{type}\ \mathsf{Sym}\ a\ b = \mathsf{K7}\ a\ b\ a\ b$$
$$\mathbf{instance}\ \mathsf{Category}\ \mathsf{Sym}\ \mathbf{where}$$
$$\quad \mathsf{id} = \mathsf{K7}\ \mathsf{id}\ \mathsf{id}$$
$$\quad (\circ) = (\otimes)$$

3. Rewinding a parser by playing in reverse

Consider the following rudimentary language:

$$\mathsf{Term} \quad t \quad ::= \quad x \mid \lambda x\ t \mid (t_1\ t_2)$$

This grammar is readily captured in HASKELL with the following datatype declaration[2]:

$$\mathbf{type}\ \mathsf{Id} = \mathsf{String}$$
$$\mathbf{data}\ \mathsf{Term}\ \mathbf{where}$$
$$\quad \mathsf{Var} :: \mathsf{Id} \to \mathsf{Term}$$
$$\quad \mathsf{Lam} :: \mathsf{Id} \to \mathsf{Term} \to \mathsf{Term}$$
$$\quad \mathsf{App} :: \mathsf{Term} \to \mathsf{Term} \to \mathsf{Term}$$

[1] Actually, in standard HASKELL one would need to make Sym a **newtype** for this instance declaration to work. We gloss over such detail here to avoid the distraction of the contingent **newtype** wrappers and unwrappers.

[2] To make the types of the constructors explicit, we use the notation introduced in GHC to support GADTs, even if this datatype is a regular algebraic datatype.

A parser for abstractions, for instance, can be written as

$$\mathsf{abstraction} =$$
$$\quad \mathsf{char}\ \text{'}\lambda\text{'} \gg$$
$$\quad \mathsf{ident} \gg\!= \lambda x \to$$
$$\quad \mathsf{term} \gg\!= \lambda t \to$$
$$\quad \mathsf{pure}\ (\mathsf{Lam}\ x\ t)$$

where, for now, we take as given char, ident and term as parsers that match the given character, yield an identifier and yield a term, respectively.

Intuitively, a parser sequentially introduces a number of subresults in scope, before building the final result.

Intuitively, the corresponding printer should destruct the final result, before sequentially printing each of the component pieces.

We essentially have an embedded domain specific language (DSL) of parsers. The effects of expressions (or *specifications*) of this language are a function of the semantics that we choose to assign to the primitives of this language (such as that of the $(\gg\!=)$ and pure combinators, whose semantics we gave definitionally as HASKELL programs in Section 1). The crux of the problem is: can the same language be an embedded DSL of printers, under different semantics? If so, we could simultaneously write the interpretations of specifications under the two semantics on each track of a cassette tape.

But it is not obvious how to construct this alternative semantics for the monadic language we have currently. The two intuitions above tell us that actions of a parser must occur inverted and in reverse in the corresponding printer, but the asymmetry of the main composition combinator $(\gg\!=)$ — it expects a parser on the left but a function on the right — means that it is impossible to read a specification in reverse. Fortunately, for context free grammars, parsers don't need the full generality of a monad, as first observed by S.D. and Swierstra [2001], who instead provides the following two basic combinators to build parsers:

$$\mathsf{pure} :: a \to \mathsf{Parser}\ a$$
$$(\langle * \rangle) :: \mathsf{Parser}\ (a \to b) \to \mathsf{Parser}\ a \to \mathsf{Parser}\ b$$
$$p_1\ \langle * \rangle\ p_2 = p_1 \gg\!= \lambda f \to p_2 \gg\!= \lambda x \to \mathsf{pure}\ (f\ x)$$

$p_1\ \langle * \rangle\ p_2$ first invokes the p_1 parser, followed by p_2, where p_1 yields a function that is applied to the result of p_2 to construct the result of this sequential composition. This interface corresponds in fact to that of applicative functors [McBride and Paterson 2008], a standard generalization of monads.

Applicative style parsers for abstractions and applications read as:

$$\mathsf{abstraction} =$$
$$\quad \mathsf{pure}\ (\lambda\_\ x\ t \to \mathsf{Lam}\ x\ t)\ \langle * \rangle$$
$$\quad \mathsf{char}\ \text{'}\lambda\text{'}\ \langle * \rangle\ \mathsf{ident}\ \langle * \rangle\ \mathsf{term}$$
$$\mathsf{application} =$$
$$\quad \mathsf{pure}\ (\lambda\_\ t_1\ \_\ t_2\ \_ \to \mathsf{App}\ t_1\ t_2)\ \langle * \rangle$$
$$\quad \mathsf{char}\ \text{'('}\ \langle * \rangle\ \mathsf{term}\ \langle * \rangle\ \mathsf{char}\ \text{' '}\ \langle * \rangle\ \mathsf{term}\ \langle * \rangle\ \mathsf{char}\ \text{')'}$$

The important point is that applicative parsers for constructors of an arbitrary number of fields are built inductively, by applying the $(\langle * \rangle)$ combinator the appropriate number of times, much like growing an onion to the right size, one layer at a time. The second point is that one can just as easily peel off the layers of the proverbial onion to make another one of the same shape and size.

The inductive structure of parsers corresponding to one alternative of one grammar production mirrors the shape of the types of constructors. These types grow along their spine, which is the trail of outermost arrows:

$$A_1 \to \ldots \to A_n \to A$$

A parser for one alternative of one grammar production, such as abstraction or application, is built from as many $(\langle * \rangle)$'s as there are arrows in the type of the constructor of the value it returns.

In applicative style, the pure element of such a parser is a constructor. In applicative style, the pure element of a printer is a destructor, *i.e.* a case analysis, such as this one:

$$\lambda t \to \mathbf{case}\ t\ \mathbf{of}\ \mathsf{Lam}\ x\ t' \to (x, t')$$

whose type is

$$\mathsf{Term} \to (\mathsf{Id}, \mathsf{Term})$$

In general the spine of a destructor of this form will look like

$$A \to (A_1, \ldots, A_n)$$

However, there is no inductive structure to this spine. Tuples are not types that are built inductively the way the types of curried functions (and indeed constructors) are. Moreover, there is something unsatisfactory about this style of destructor: the spine of their types is not symmetrical to that of the corresponding constructors. Surely, if parsing is the dual of printing, there should be a symmetry to the components they are built from. To recover this symmetry, we would have to uncurry the constructors,

$$\begin{aligned} &\textbf{constructor types (tupled):} && (A_1, \ldots, A_n) \to A \\ &\textbf{destructor types (tupled):} && A \to (A_1, \ldots, A_n) \end{aligned}$$

but doing so would make them too lose the inductive structure of their types!

Destructors are, intuitively, multivalue functions, just as constructors are multi-argument functions. In direct style, currying deals with multiple arguments, but not multiple values. But in continuation passing style (CPS), a multivalued function can be encoded as a function with a curried continuation, just as n-ary functions can be encoded as a curried functions. In CPS, multivalue functions look no different from single value functions and have no extra level of indirection through tupling[3]:

$$\begin{aligned} &\textbf{constructor types (CPS):} && (A \to r) \to (A_1 \to \ldots \to A_n \to r) \\ &\textbf{destructor types (CPS):} && (A_1 \to \ldots \to A_n \to r) \to (A \to r) \end{aligned}$$

In CPS, all functions always take a continuation as their first argument, whose *answer type* is r. Functions in pure CPS, who do not perform any control effects, always make a tail-call to their continuation eventually, so that the return type of all functions is the answer type of the continuation[4]. In CPS, our constructors and destructors are just as symmetric as in tupled style (sprinkling some redundant parentheses makes this clearer). But unlike tupled style, the spines of the types have an inductive structure to them.

Turning all parsers and printers into CPS, and threading state (the string that a parser consumes, or the string that a printer produces) through them, begets the following type schemas:

parser types (CPS):

$$(\mathsf{String} \to A_1 \to \ldots \to A_n \to r) \to (\mathsf{String} \to r)$$

printer types (CPS):

$$(\mathsf{String} \to r) \to (\mathsf{String} \to A_1 \to \ldots \to A_n \to r)$$

In general, a parser takes a string as input and produces a string and zero or more values as output, which it passes to its continuation. A printer accepts as input a string and zero or more values, which it prints onto the string. We have come back full swing to a symmetrical presentation of printers, just like we had in the very beginning of Section 1, as reverse parsers, whose type is that of parsers but with an arrow flipped in the other direction. But unlike the parsers, printers and types we started with, these are composable...

[3] Observe that composing an n-ary destructor with its corresponding constructor gives the Church encoding of an n-ary tuple.

[4] Strictly speaking, though we postpone further discussion to Section 6, the arguments' types are part of the answer type, so that functions move from one answer type to another answer type. The type of a destructor is actually a suffix of the answer type of its continuation, and conversely for the answer type of constructors.

4. Recording cassettes

Cassettes that contain a parser producing values of type a on their A-side and printers for values of that type on their B-side are called P/P pairs:

$$\begin{aligned} \mathbf{type}\ \mathsf{PP}\ a = \forall r\ r'.\mathsf{K7}\ &(\mathsf{String} \to a \to r)\ (\mathsf{String} \to r) \\ &(\mathsf{String} \to r')\quad (\mathsf{String} \to a \to r') \end{aligned}$$

The parsers and printers are functions in CPS that are oblivious to the actual answer type. This invariant is enforced at the level of types by requiring that functions on each side of the cassette work *for any possible answer type*[5], not just a specific one[6].

Of course, some P/P pairs are only interesting for their side effects and do not produce or consume any value at all, as their type informs:

$$\begin{aligned} \mathbf{type}\ \mathsf{PP0} = \forall r\ r'.\mathsf{K7}\ &(\mathsf{String} \to r)\ \ (\mathsf{String} \to r) \\ &(\mathsf{String} \to r')\ (\mathsf{String} \to r') \end{aligned}$$

An example is a cassette that parses/prints a given string literal:

$$\begin{aligned} &\mathsf{lit} :: \mathsf{String} \to \mathsf{PP0} \\ &\mathsf{lit}\ x = \mathsf{K7}\ (\lambda k \to k \circ \mathsf{stripPrefix}\ x)\ (\lambda k \to k \circ (x +\!\!+)) \end{aligned}$$

where $\mathsf{stripPrefix} :: \mathsf{String} \to \mathsf{String} \to \mathsf{String}$ is a partial function that, provided the first argument is a prefix of the second argument, chops this prefix off.

Additional primitive and composed cassettes can also be manufactured:

$$\begin{aligned} &\mathsf{char} :: \mathsf{Char} \to \mathsf{PP0} \\ &\mathsf{char}\ x = \mathsf{lit}\ [x] \\ &\mathsf{anyChar} :: \mathsf{PP\ Char} \\ &\mathsf{anyChar} = \mathsf{K7}\ (\lambda k\ (x : s) \to k\ s\ x)\ (\lambda k\ s\ x \to k\ (x : s)) \\ &\mathsf{digit} :: \mathsf{PP\ Int} \\ &\mathsf{digit} = \mathsf{K7}\ (\lambda k\ (x : s) \to k\ s\ (\mathsf{read}\ x)) \\ &\qquad\qquad (\lambda k\ s\ x \to k\ (\mathsf{show}\ x +\!\!+ s)) \end{aligned}$$

More ambitious basic building blocks will be given in Section 5, where we will also properly define

$$\mathsf{ident} :: \mathsf{PP\ String}$$

which parses/prints any identifier (*i.e.* a string of alphanumeric characters), but as things stand we can already assemble a cassette from these basic ones to handle variables, abstractions and applications.

The fundamental glue is the $(\otimes)$ combinator of Section 2 that composes in opposite directions two pairs of functions. In CPS, as observed by [Danvy 1998] and put to good use a number of times [Rhiger 2009; Fridlender and Indrika 2000], composing an n-ary function with an m-ary function with polymorphic answer types yields an $(n + m)$-ary function. Provided

$$\begin{aligned} &f :: (\mathsf{String} \to r_1) \to (\mathsf{String} \to A_1 \to \ldots \to A_n \to r_1) \\ &g :: (\mathsf{String} \to r_2) \to (\mathsf{String} \to B_1 \to \ldots \to B_m \to r_2) \end{aligned}$$

[5] In principle, this property lets us establish the absence of certain control effects, as a consequence of parametricity alone.

[6] This is the only definition in this paper that steps outside of HASKELL'98. It requires support for rank 2 types, implemented as an extension in virtually every extant HASKELL compiler and in OCAML.

we have that

$$f \circ g :: (\mathsf{String} \to r_2) \to \mathsf{String} \to A_1 \to \ldots \to A_n \to B_1 \to \ldots \to B_m \to r_2$$

since the types of f, g are unifiable only under the constraint

$$r_1 = B_1 \to \ldots \to B_m \to r_2$$

Conversely, composing an n-value function with an m-value function yields an $(n+m)$-value function. Beware, however, that dually to multiple argument functions, the values of the resulting multi-value function are returned in reverse. Provided

$$h :: (\mathsf{String} \to A_1 \to \ldots \to A_n \to r_1) \to (\mathsf{String} \to r_1)$$
$$l :: (\mathsf{String} \to B_1 \to \ldots \to B_m \to r_2) \to (\mathsf{String} \to r_2)$$

we have that

$$h \circ l :: (\mathsf{String} \to B_1 \to \ldots \to B_n \to A_1 \to \ldots \to A_m \to r_1) \to \mathsf{String} \to r_1$$

since the types of h, l are unifiable only under the constraint

$$r_2 = A_1 \to \ldots \to A_m \to r_1$$

This glue allows us for instance to make a piece of tape for the components of an abstraction,

$$\mathsf{char}\ \text{'}\lambda\text{'} \otimes \mathsf{ident} \otimes \mathsf{term}$$

whose inferred type is

$$\mathsf{K7}\ (\mathsf{String} \to \mathsf{Id} \to \mathsf{Term} \to r)\ (\mathsf{String} \to r)\ (\mathsf{String} \to r')\ (\mathsf{String} \to \mathsf{Term} \to \mathsf{Id} \to r')$$

i.e. the type of a function that parses multiple values in sequence, paired with a function that prints multiple arguments in sequence.

Reading the above specification from left to right tells us the story of the parser and reading it from right to left tells us of the printer. All good stories have an beginning and an end, however, so that the we must splice a special lead segment on one end of the tape. On the A-side, the lead marks the end of the tape (lead-out) and on the B-side, since we are reading it in the opposite direction, it marks the beginning (lead-in). When parsing, the lead finishes the computation by combining all the parsed values into a single value, by applying a constructor or performing a fold. When printing, the lead initiates the computation by eliminating a single value, by case analysis or performing an unfold.

In Section 3, we introduced constructors and destructors in CPS. Their types have the same shape as that of printers and parsers, respectively. In as much as one can view a parser as a multiplexer (creating many outputs from one input), a constructor demultiplexes its output. One the B-side, a printer demultiplexes the outputs of a destructor. With the adjunction of a lead to the above piece of tape, we obtain a complete parser and printer for abstractions, simultaneously:

```
abstraction :: PP Term
abstraction = absL ⟶ char 'λ' ⊗ ident
  where absL = K7 leadout leadin
        leadout k s t x = k s (Lam x t)
        leadin k s (Lam x t) = k s t x
```

Remember that a parser, being a composition of multivalue functions, produces its values in reverse order. Hence an artifact of our approach is that the lead-out accepts arguments in reverse order compared to the order in the specification. Likewise, the lead-in feeds the components of the destructed value to the printer in reverse order, *i.e.* the order of a right-to-left reading of a specification.

5. Choice points

The discussion so far steered clear from managing failure and backtracking. The absence of these control effects confines our parser and printers to a single alternative of a single production in a grammar.

Such effects are easily performed in CPS. Dropping the current continuation (or *aborting* it) simulates failure, for instance. However, control effects alter the answer type, which we require to remain polymorphic for cassette splicing to work[7]. Moreover, terms performing control effects are not in fact in the image of CPS transform. Taking our cue from [Danvy and Filinski 1990], we simply iterate the CPS transform one more time, thereby translating away control effects into terms that are in CPS and therefore parametric in their answer type.

Starting with the simplest parsers.

After two iterations of the CPS transform, parsers that produce no output have types of the following form:

$$((t \to r) \to \mathsf{String} \to r) \to (t \to r) \to \mathsf{String} \to r$$

We now have two continuations, the first of which we can use as a success continuation, and the second as a failure continuation. In essence, the failure continuation is (additional) state threaded through parsing, just as the input string is, and can therefore change at every step. The r type to the failure continuation can be chosen arbitrarily. We do not require it to be abstract, and we will find it convenient to make this argument a String:

$$((\mathsf{String} \to r) \to \mathsf{String} \to r) \to (\mathsf{String} \to r) \to \mathsf{String} \to r$$

Generalizing to multivalue functions, we impose that the number of arguments to the failure continuation available to the success continuation remain consistent with the number of arguments to the success continuation itself. That is, baking failure into parsers alters their types to ones of the following form:

$$((\mathsf{String} \to \vec{A} \to r) \to \mathsf{String} \to \vec{A} \to r) \to (\mathsf{String} \to r) \to \mathsf{String} \to r$$

Thus, if the success continuation fails, when calling the failure continuation it should pass on the arguments that it was given. Maintaining synchrony between the arity of the success continuation and that of the failure continuation makes parsers composable just as before (and as Section 6 will demonstrate, it is no surprise that they should). Assuming (for brevity, $\vec{A}$ stands for $A_1 \to \ldots \to A_n$)

$$h :: ((\mathsf{String} \to \vec{A} \to r_1) \to \mathsf{String} \to \vec{A} \to r_1) \to ((\mathsf{String} \to r_1) \to \mathsf{String} \to r_1)$$
$$l :: ((\mathsf{String} \to \vec{B} \to r_2) \to \mathsf{String} \to \vec{B} \to r_2) \to ((\mathsf{String} \to r_2) \to \mathsf{String} \to r_2)$$

we have that

$$h \circ l :: ((\mathsf{String} \to \vec{B} \to \vec{A} \to r_1) \to \mathsf{String} \to \vec{B} \to \vec{A} \to r_1) \to ((\mathsf{String} \to r_1) \to \mathsf{String} \to r_1)$$

Printers have the same type, with the outermost arrow reversed. That is, we redefine

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

[7] Answer type polymorphism is also the reason why cannot simply lift functions into the exception monad; we would then have answer types of the form $m\ r$, leading to unsatisfiable unification constraits of the form $m\ r = \mathsf{String} \to \ldots \to m\ r$ when splicing.

```
K7 ((String → r) → String → r)
   ((String → r) → String → r)
   ((String → r') → String → r')
   ((String → r) → String → r')
```

Cassettes can now be combined not just "horizontally", as with the $(\otimes)$ combinator to form one alternative of a grammar, but also vertically, to form one full grammar production, using the choice combinator:

```
(⊕) :: PP a → PP a → PP a
K7 f f' ⊕ K7 g g' =
  K7 (λk_s k_f s → f  k_s (λs' → g  k_s k_f s) s)
     (λk_s k_f s → f' k_s (λs' → g' k_s k_f s) s)
```

The vertical composition of two cassettes $pp_1 \oplus pp_2$ (of lower precedence than $(\otimes)$) gives a new cassette where the A-side and B-side of pp_2 are the failure continuation of the A-side and B-side of pp_1, respectively. Notice that the failure continuation restarts parsing or printing using the same initial input string[8].

Furthermore, allowing for failure makes it possible to write *total* leads, where the destructors fail gracefully if the input is not of the expected shape. For instance, a destructor for a list cons cell cannot destruct the empty list, and *vice versa*:

```
cons_L = K7 leadout leadin where
  leadout k_s k_f s xs' x = k_s (λs _ → k_f s xs' x) s (x : xs')
  leadin k_s k_f s xs@(x : xs') = k_s (λs _ _ → k_f s xs) s xs' x
  leadin k_s k_f s xs           = k_f xs
nil_L = K7 leadout leadin where
  leadout k_s k_f s = k_s (λs _ → k_f) s []
  leadin k_s k_f s xs@[] = k_s (λs → k_f s xs) s
  leadin k_s k_f s xs    = k_f xs
```

Because constructors map n inputs to 1 output, the failure continuation must be applied n times and abstracted over once, to make it unary. Because destructors map 1 input to n outputs, the failure continuation must be applied once and abstracted over n times, to make it n-ary.

We now have the final missing ingredient to implement one very fundamental combinator, repetition ("zero or more" and "one or more"):

```
many :: PP a → PP [a]
many pp = many1 pp ⊕ nil_L
many1 :: PP a → PP [a]
many1 pp = cons_L ⟶ pp ⊗ many pp
```

as well as a generalization of anyChar whose success is predicated on a boolean function:

```
satisfy :: (Char → Bool) → PP Char
satisfy p = K7 f g where
  f k_s k_f (x : xs) | p x = k_s (λs _ → k_f s) xs x
  f k_s k_f _              = k_f s
  g k_s k_f s x | p x      = k_s (λs → k_f s x) (x : s)
                | otherwise = k_f s x
```

An identifier is one or more character that are alphanumerical:

```
ident = many1 (satisfy isAlphaNum)
```

[8] This is the source of a space leak because it means that the $(\oplus)$ combinator holds on to the input string indefinitely in order to allow for backtracking. In practice, parser combinator libraries limit the lookahead for backtracking to 1 by default, so that the initial input string may be dropped immediately upon successfully consuming just one character of the input. This more limited choice combinator can just as easily be implemented in the framework presented here, if so desired.

Thus equipped, writing out the full parser for the pure λ-calculus, for example, gives:

```
var_L = K7 leadout leadin where
  leadout k_s k_f s x = k_s (λs _ → k_f s x) s (Var x)
  leadin k_s k_f s t@(Var x) = k_s (λs _ → k_f s t) s x
  leadin k_s k_f s t         = k_f s t
abs_L = K7 leadout leadin where
  leadout k_s k_f s t' x = k_s (λs _ → k_f s t' x) s (Lam x t')
  leadin k_s k_f s t@(Lam x t) = k_s (λs _ _ → k_f s t) s t x
  leadin k_s k_f s t           = k_f s t
app_L = K7 leadout leadin where
  leadout k_s s t2 t1 = k_s (λs _ → k_f s t2 t1) s (App t1 t2)
  leadin k_s k_f s t@(App t1 t2) = k_s (λs _ _ → k_f s t) s t2 t1
  leadin k_s k_f s t             = k_f s t
term :: PP Term
term = var_L ⟶ ident
     ⊕ abs_L ⟶ char 'λ' ⊗ ident ⊗ term
     ⊕ app_L ⟶ parens (term ⊗ space ⊗ term)
```

where we defined

```
parens pp = char '(' ⊗ pp ⊗ char ')'
space = char ' '
```

The essential difference with the definitions in Section 4 is that abstraction and application are allowed to fail, triggering backtracking at the choice points $(\oplus)$ if they do.

Taking a step back, focusing out, it is hard to overlook the relative verbosity of leads and the syntactic noise that is the bookkeeping of continuations. Moreover, the types of leads can grow rather unwieldy, so much so that the columns of this page are too snug a fit to include them. Once the satisfy, $\oplus$ and $\otimes$ primitives are defined, we only need to deal with continuations explicitly when writing the leads, which we could unburden from the user with a modicum of meta-programming. Still, in the next section, we take the more tasteful approach of implementing aggressive noise attenuation measures.

6. Cassettes in direct style

A program in CPS is said to be *pure* when the continuation that is passed as an extra argument to every function is used by the latter with some discipline. In particular, functions may only transfer control through [Asai 2009]

- a tail-call to another function, or
- a tail-call to another continuation.

By contrast, ordinary functions with no continuations threaded through them are said to be in *direct style*. Ordinary functions are typically easier to write and understand than functions in CPS. Moreover, pure CPS functions can readily be translated into direct style by inverting the CPS transform [Danvy 1992]. Therefore, it is seldom useful to write pure CPS functions. The point of writing functions in CPS is to profit from selectively bend the rules, locally stepping outside of the boundaries of purity and into impurity.

Direct style primitives: We can apply the same analysis of the solution of Danvy [1998] given by Asai [2009] to the parsing and pretty printing cassettes worked out in the previous sections. While most of the introduced cassettes and glue are in pure CPS, a select few are not. For instance, η-expanding $(\otimes)$ shows that this combinator is in fact in pure CPS:

```
K7 f f' ⊗ K7 g g' =
  K7 (λk_s k_f s → f (λs → g k_s k_f s) s)
     (λk_s k_f s → g' (λs → f' k_s k_f s) s)
```

This alternative definition makes clear that composition of P/P pairs stacks the right hand function onto the continuation of the left hand function on the one side, and the left hand function onto the continuation of the right hand on the other side of the tape. All calls are tail calls. In the anyChar cassette (in Section 4), in contrast, the call to the provided continuation isn't really a tail call, because the call is guarded by an abstraction. This is most easily observed in the following equivalent definition (without failure):

$$\begin{aligned}\mathsf{anyChar} = \mathsf{K7}\ &(\lambda k_s\ (x : s) \to (k_s\ s)\ x)\\ &(\lambda k_s\ s \to (\lambda x \to k_s\ (x : s)))\end{aligned}$$

The anyChar function does not preserve the answer type of the continuation it is passed, because on the A-side the interceding application moves us from an answer type of the form $a \to r$ to r and conversely, the interceding abstraction on the B-side moves us from an answer type of the form r to a type of the form $a \to r$.

Fundamentally, all of the continuations we are manipulating in Section 4 have but one argument, the String that we are threading through computations, and all other argument types are actually part of the answer type.

Conversely, all functions we are manipulating in Section 4 have but one non-continuation argument, the String that we are threading through computations, and all other argument types are actually part of the answer type.

Iterating the CPS transform as we did in Section 5 merely adds one extra argument to every continuation and one extra argument to every function: the failure continuation. Again, all other arguments types are really part of the answer type.

Therefore, the way continuations are used in anyChar, as well as all other cassettes of type $\mathsf{PP}\ a$ for some type a, is *non-standard*. That is, these CPS functions are impure and make use of control effects. The typed sprintf solution of Danvy [1998] arises as a special case of the printers presented here. Asai [2009] identifies these effects as uses of the shift and reset operators introduced by Danvy and Filinski [1990]. We observe that our pairs of parsers and printers make use of shift on both tracks of the tape to capture the context up to a delimiting reset. In addition, the $(\oplus)$ combinator aborts delimited continuations up to the same reset to achieve backtracking behaviour.

The semantics of shift in direct style can be explained by mapping it to a term of the pure λ-calculus in CPS. A term of the form $\mathsf{shift}\ (\lambda k \to M[k])$ corresponds, in CPS, to the term

$$M'[k]\ (\lambda x \to x)$$

where M' is the result of applying the CPS transform on M.

In general, only part of the context is captured as a continuation and bound to k by shift. The delimiter for the continuation is reset. A term of the form $\mathsf{reset}\ M$ corresponds, in CPS, to the term

$$\lambda k \to k\ (M'\ (\lambda x \to x))$$

where M' is the result of applying the CPS transform to M.

The abort operator is a special case of shift, where the captured continuation is never called.

Using shift, it is possible to give an implementation in direct style of anyChar (still without failure), without having to resort to continuation passing style:

$$\begin{aligned}\mathsf{anyChar} = \mathsf{K7}\ &(\lambda(x : s) \to \mathsf{shift}\ (\lambda k \to k\ s\ x))\\ &(\lambda s \to \mathsf{shift}\ (\lambda k\ x \to k\ (x : s)))\end{aligned}$$

We can readily translate this direct style definition into CPS using the above semantics for shift, to verify that this function is indeed the same function as before.

Assigning a type to this definition is a more involved matter, because the two occurrences of shift move us between two distinct answer types. Types of functions in pure CPS are in bijective correspondence with types in direct style: CPS types, of the form $(A \to r) \to B \to r$, correspond to type $A \to B$ in direct style, and *vice versa*. But if the answer type changes as we move across the outermost arrow, than one cannot give a corresponding direct style type in this way.

Danvy and Filinski [1989] enunciate a type and effects system to for functions in direct style that use control effects. This type system accounts for the change in answer type that would occur in the corresponding CPS style terms. Without repeating the full system here in all its glory, we can provide an intuition as to how it works. In essence, the source and destination of an arrow type are annotated to reflect the change in answer type as we move across the arrow. More concretely, the type of functions has the form $A/R_1 \to B/R_2$, where R_1, R_2 can be two arbitrary answer types. In the absence of any effects, the types of the CPS terms corresponding to functions in direct style are of the form $(A \to r) \to B \to r$, which polymorphic in the answer type, meaning that effect-free direct style functions have types of the form $A/r \to B/r$. In general, a type of the form $A/R_1 \to B/R_2$ maps to $(A \to R_1) \to B \to R_2$ in CPS. Thus, in this system, the types of the two tracks of direct style anyChar are:

A-side type:	$\mathsf{String}/r \to \mathsf{String}/(\mathsf{Char} \to r)$
B-side type:	$\mathsf{String}/(\mathsf{Char} \to r) \to \mathsf{String}/r$

Factoring in failure, which we can illustrate with the more general satisfy primitive, we have the following definition after inverting the CPS transform once:

$$\begin{aligned}&\mathsf{satisfy}\ p =\\ &\quad \mathsf{K7}\ (\lambda k_f\ s \to \mathsf{shift}\ (\lambda k \to \mathbf{case}\ s\ \mathbf{of}\\ &\qquad\quad x : s' \mid p\ x \to k\ (\lambda s\ \_ \to k_f\ s)\ s\ x\\ &\qquad\quad \_ \qquad\quad \to k_f\ s)\\ &\qquad (\lambda k_f\ s \to \mathsf{shift}\ (\lambda k\ x \to\\ &\qquad\quad \mathbf{if}\ p\ x\ \mathbf{then}\ k\ (k_f\ s\ x)\ (x : s)\ \mathbf{else}\ k_f\ s\ x))\end{aligned}$$

As before, the shift moves us from one answer type to another. Here, shift also alters the answer type in the failure continuation. Indeed, there is an interceding application and abstraction on the A-side and the B-side (respectively) before the tail call to the failure continuation. Looking at the type of above expression we have:

$$\begin{aligned}\textbf{A-side type:}\quad &\mathsf{String}/((\mathsf{String} \to \mathsf{Char} \to r) \to \mathsf{String} \to r)\\ &\to \mathsf{String}/(((\mathsf{String} \to r) \to \mathsf{String} \to \mathsf{Char} \to r))\\ \textbf{B-side type:}\quad &\mathsf{String}/(((\mathsf{String} \to r) \to \mathsf{String} \to \mathsf{Char} \to r))\\ &\to \mathsf{String}/((\mathsf{String} \to \mathsf{Char} \to r) \to \mathsf{String} \to r)\end{aligned}$$

We can notice, from the text itself and also from its type, that this definition is still in continuation passing style — it must then itself be the result of CPS transforming some function with effects. Thus, we can invert the transform one more time, to get:

$$\begin{aligned}&\mathsf{satisfy}\ p =\\ &\quad \mathsf{K7}\ (\lambda s \to \mathsf{shift}_1\ (\lambda k_1\ \_ \to \mathsf{shift}_2\ (\lambda k_2 \to \mathbf{case}\ s\ \mathbf{of}\\ &\qquad\quad x : s \mid p\ x \to k_2\ s\ x\\ &\qquad\quad \_ \qquad\quad \to k_1)))\\ &\qquad (\lambda s \to \mathsf{shift}_1\ (\lambda k_1 \to \mathsf{shift}_2\ (\lambda k_2\ x \to\\ &\qquad\quad \mathbf{if}\ p\ x\ \mathbf{then}\ k\ (x : s)\ \mathbf{else}\ k_1\ x)))\end{aligned}$$

Just as the first transformation to direct style introduced control operators to account for the non-standard use of continuations, so does this second iteration. However, following Danvy and Filinski [1990], because the effects we are witnessing are in fact happening at two different levels care must be taken to give distinct names to the associated control operators of each level.

The final result is now properly in direct style. The function we obtain is not entirely dissimilar to anyChar, its more rudimentary non-failing kin, in that it still just a string transformer, mapping a

String to a String, as its type informs. Danvy and Filinski [1989] only give a type and effects system for the first level of the CPS hierarchy. But here as we iterated the direct style transform twice, we might be tempted to write a type that succinctly reflects the effects occurring at both levels:

$$\begin{aligned}\textbf{A-side type:}\quad & \mathsf{String}//(\mathsf{String}/r \to \mathsf{String}/(\mathsf{Char} \to r)) \\ & \to \mathsf{String}//(\mathsf{String}/(\mathsf{Char} \to r) \to \mathsf{String}/r) \\ \textbf{B-side type:}\quad & \mathsf{String}//(\mathsf{String}/(\mathsf{Char} \to r) \to \mathsf{String}/r) \\ & \to \mathsf{String}//(\mathsf{String}/r \to \mathsf{String}/(\mathsf{Char} \to r))\end{aligned}$$

where, to avoid confusion, we demarcate a type from its effect with / and // according to the level of the effect. Interestingly, we can notice that on both sides, at the inner level answer types are shifted exactly as they are in the outer level, but in the opposite direction.

Direct style leads: Inherently, our cassettes have string transformers on each side — it's just that these transformers are side-effecting ones. Switching to direct style from the CPS style of the previous sections makes this plain. We have seen how primitive cassettes can be written in this style, which advantageously makes continuations explicit only at the points where a control effect will occur, without polluting pure snippets of code. However, most primitives can be written once and for all and included in a library to be reused among many parsers and printers for many grammars in many different projects, so this syntactic advantage is nice but perhaps not essential. Leads, on the other hand, must be written for every constructor of every datatype that parsers target and printers consume. One can therefore expect a very handsome payoff to making the leads shorter to write and easier to read.

We identified in Section 4 that constructors can be viewed as pure printers — it does not transform the string in any way — and that destructors can be view as pure parsers. We can therefore expect to be able to write leads as pairs of effectful string transformers, just as primitives can.

The answer types in leads differ slightly to primitive parsers and printers. A primitive parser has no arguments other than its input string and a primitive printer has no outputs beyond its output string, while a destructor ignores the input string and creates multiple outputs from a single input, and a constructor also ignores the input string and a single output from multiple inputs. In other words, a primitive parser only modifies the answer type on the left of outermost arrow and a primitive printer only modifies the answer type on the right on arrow, while constructors and destructors do both.

Again, these effects can succinctly be captured by shifting. In direct style, a lead-out for a constructor C of a value of type T given arguments of type $A_1, \ldots, A_n$ has the form:

$$\begin{aligned}&\mathsf{leadout} = \lambda s \to \mathsf{shift}_1\ (\lambda k_1\ \_ \to \mathsf{shift}_2 \\ &\quad (\lambda k_2\ x\_n\ \ldots\ x_1 \to k_2\ s\ (\mathsf{C}\ x_1\ \ldots\ x\_n))\end{aligned}$$

A lead-in for C (ie a destructor) has the form:

$$\begin{aligned}&\mathsf{leadin} = \lambda s \to \mathsf{shift}_1\ (\lambda k_1\ \_\ \ldots\ \_ \to \mathsf{shift}_2 \\ &\quad (\lambda k_2\ v \to \mathbf{case}\ v\ \mathbf{of} \\ &\qquad \mathsf{C}\ x_1\ \ldots\ x\_n \to k_2\ s\ x\_n\ \ldots\ x_1 \\ &\qquad \_ \qquad\qquad\ \to k_1\ s\ v))\end{aligned}$$

The input string passes through lead-ins and lead-outs untouched.

Direct style in HASKELL***:*** While the work in all previous sections made do with standard HASKELL'98 with one extension, in a typed setting the price to pay to write shorter and more readable leads is to move to a language that supports effect annotations in the types. This is not an issue in an untyped programming language of course, but even in a typed language Kiselyov [2007] shows that one could simulate the control effects used here using a parameterized monad, which is a generalized notion of a monad. Kiselyov constructs a specific instance of a parameterized monad that allows a type-safe embedding in HASKELL'98 of a language with a polymorphic type and effects system [Asai and Kameyama 2007].

7. Discussion and related work

The most effective way of maintaining synchrony between disjoint but related components is to share the bulk of their code. The same changes then do not need to be repeated across multiple compilation units and the consistency between components is ensured mostly by construction and statically checked. Inadvertently compromising the consistency between data consumers and data producers can be a particularly tricky problem to spot.

The approach we have taken here[9] is to define an embedded domain specific language for parsing and pretty printing where the primitives building blocks are kept to the smallest size possible. In this way, most of the domain specific knowledge is encoded at the DSL layer, rather than the layer beneath, that of the host language. The more happens at the DSL layer, the more we get for free, because whatever logic remains within the primitives is logic that must necessarily be duplicated, for one direction and the other.

We started with the standard monadic/applicative framework for parsers. While the control effects of primitives and leads could be implemented using monads, the final solution does not use the usual monadic interface for parsers, however, because monads are not closed under the gluing combinators we provide. We argue that our framework achieves better "horizontal" composability, however, because the $(\otimes)$ combinator is associative and therefore cassettes can be build piecemeal and extended horizontally from either end at any time.

A number of solutions to at least part of the problem have been proposed. Perhaps the most closely related to our work are the solutions to the typed sprintf problem put forth by Danvy [1998] and Asai [2009]. While their work only deals with printing, we follow the same general idea of inductively constructing higher-order functions. Danvy shows how to define adhoc formatters for polymorphic algebraic datatypes, but these formatters are mostly defined within the host language, rather than in terms of alternation and sequencing primitives exclusively as we do. In as sense, we have taken this approach to its logical conclusion by increasing the granularity of primitives even more.

Kennedy [2004] defines pickling combinators for serializing data and deserializing it. As in our approach, the primitive picklers are paired with their unpickler. The picklers are less composable, however. In particular, all datatypes need to have their fields packed into tuples, and distinct combinators handle each tuple arity. By contrast, we support rows of data of arbitrary length, through currying. Also, variants in datatypes are either binary coded using nested Either a b values, or tagged with an integer. Adding a new constructor to a datatype potentially requires to renumber the tags everywhere in the code. The solution of Smetsers et al. [2009] using bidirectional arrows suffers from the same lack of "vertical" composability. Rendel and Ostermann [2010] achieve excellent horizontal and vertical composability by layering a set of combinators atop a formalism of partial isomorphisms, and define alternation combinators in a similar way to ours. The main difference lies in the fact that they achieve horizontal composability through nested tuples, rather than currying, which can cause extra allocation at runtime and potentially impacts performance substantially.

[9] A prototype is available at: `http://www.cs.mcgill.ca/~mboes/src/cassette/`

References

K. Asai. On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation*, 22(3):275–291, 2009.

K. Asai and Y. Kameyama. Polymorphic delimited continuations. In Z. Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2007. ISBN 978-3-540-76636-0.

O. Danvy. Back to direct style. In B. Krieg-Brückner, editor, *ESOP*, volume 582 of *Lecture Notes in Computer Science*, pages 130–150. Springer, 1992. ISBN 3-540-55253-7.

O. Danvy. Functional unparsing. *J. Funct. Program.*, 8(6):621–625, 1998.

O. Danvy and A. Filinski. A functional abstraction of typed contexts. Technical report, DIKU, 1989.

O. Danvy and A. Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.

D. Fridlender and M. Indrika. Do we need dependent types? *J. Funct. Program.*, 10(4):409–415, 2000.

A. Kennedy. Pickler combinators. *J. Funct. Program.*, 14(6):727–739, 2004.

O. Kiselyov, Dec. 2007. URL `http://okmij.org/ftp/continuations/implementations.html#genuine-shift`.

C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

T. Rendel and K. Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing. In J. Gibbons, editor, *Haskell*, pages 1–12. ACM, 2010. ISBN 978-1-4503-0252-4.

M. Rhiger. Type-safe pattern combinators. *J. Funct. Program.*, 19(2):145–156, 2009.

S.D. and Swierstra. Combinator parsers: From toys to tools. *Electronic Notes in Theoretical Computer Science*, 41(1):38 – 59, 2001. ISSN 1571-0661.

S. Smetsers, A. van Weelden, and R. Plasmeijer. Efficient and type-safe generic data storage. *Electr. Notes Theor. Comput. Sci.*, 238(2):59–70, 2009.