An Intuition for R-stream — the Fundamental Data Structure of OCanren

Li Yue and Moiseenko Evgeniy
JetBrains Research, Saint-Petersburg, Russia

March 26, 2022

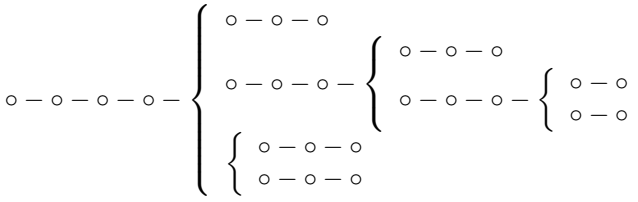
The fundamental data structure of the relational programming language OCanren is *r-stream*. Users of OCanren never need to work directly with r-streams, but behind the scenes all data computed by OCanren are organized as r-streams and all logical operations of OCanren are implemented as operations on r-streams. In this article we isolate the r-stream data type from OCanren, and study it in its own right. We shall discuss, *intuitively*, what an r-stream looks like (Sec. 1) and how the basic operation — interleaved merge, works on r-streams (Sec. 2). Sec. 3 shares the programming experience that provides these intuitions. Sec. 4 suggests future work on a formalized theory of r-streams. Sec. 5 discusses about alternative definitions for r-stream.

1 From List to R-stream

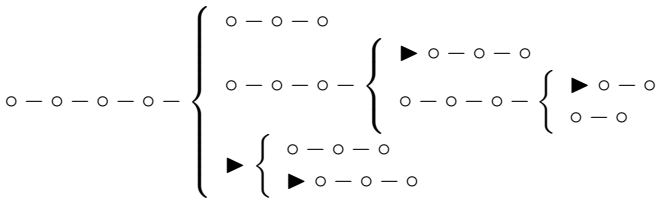
The reader might be familiar with a *list*. A typical list has the shape:

where the circles represent list members. A list is a finite object and it is readily generalized to an infinite one, known as a *stream*:

If we allow a list to *finitely* branch, this would give a new data structure that shall no longer be called a list — let us call it a *brancher* instead, the shape of which is typically:



We can further enrich a brancher with such a feature that every branch has a two-value state. The resulting structure shall be called a *binary-state brancher*. Using the presence or absense of a triangle ▶ to indicate the state of a branch, we depict a typical binary-state brancher as:



Binary-state brancher is to r-stream as list is to stream.

2 Interleaved Merge

We introduce the operation of *interleaved merge* on lists, and then extend the operation to branchers, and finally to binary-state branchers. We denote interleaved merge by the infix operator $\oplus$ (the o-plus symbol).

2.1 Merging Lists

The operator $\oplus$ is in general non-commutative. Below is an example.

l_1	1		2		3		4		5		6
l_2		a		b		c		d		e	
$l_1 \oplus l_2$	1	a	2	b	3	c	4	d	5	e	6

l_1		1		2		3		4		5	6
l_2	a		b		c		d		e		
$l_2 \oplus l_1$	a	1	b	2	c	3	d	4	e	5	6

In the example, the first row gives l_1 that is a list of six numbers; the second row gives list l_2 that is a list of five characters. The third row gives $l_1 \oplus l_2$. The indentation and spacing of the l_1, l_2 rows highlight *interleaving*. The fourth and fifth rows repeat l_1 and l_2 but in a different way of interleaving, which gives $l_2 \oplus l_1$. In general, elements of the left operand of $\oplus$ are systematically ahead of their counterparts in the right operand by one unit during interleaving.

The $\oplus$ is not associative either. Say $l_1 = 1$, $l_2 = 2$ and $l_3 = 3$. Then, $l_1 \oplus (l_2 \oplus l_3) = 1\ 2\ 3$ but $(l_1 \oplus l_2) \oplus l_3 = 1\ 3\ 2$.

2.2 Merging Branchers

If we can somehow convert a brancher into a list, then the task of interleaved merge of two branchers is reduced to the task of interleaved merge of their corresponding lists. Such a conversion from a brancher to a list is termed *flattening*.

A list is the most simple form of a brancher, the flattening of which is immediate. A slightly more complicated brancher is a table of lists, For example

$$\left\{\begin{array}{l} \circ_1 - \circ_2 - \circ_3 \\ \circ_4 - \circ_5 \\ \circ_6 - \circ_7 - \circ_8 - \circ_9 \end{array}\right.$$

The way we flatten a table of lists is to perform interleaved merge on all the lists. For example, if we name the three lists in the table as l_1, l_2, l_3 repectively,

and denote the table by $\mathcal{T}(l_1, l_2, l_3)$ and its flattening $\mathring{\mathcal{T}}(l_1, l_2, l_3)$, we can define $\mathring{\mathcal{T}}(l_1, l_2, l_3) = l_1 \oplus (l_2 \oplus l_3)$, that is,

l_2	$\circ_4 - \circ_5$
l_3	$\circ_6 - \circ_7 - \circ_8 - \circ_9$
$l_2 \oplus l_3$	$\circ_4 - \circ_6 - \circ_5 - \circ_7 - \circ_8 - \circ_9$
l_1	$\circ_1 - \circ_2 - \circ_3$
$l_1 \oplus (l_2 \oplus l_3)$	$\circ_1 - \circ_4 - \circ_2 - \circ_6 - \circ_3 - \circ_5 - \circ_7 - \circ_8 - \circ_9$

The next level of complication concerns a brancher of the form

$$\circ_1 - \circ_2 - \left\{ \begin{array}{l} \circ_3 - \circ_4 \\ \circ_5 \end{array} \right.$$

that is a list that branches into lists. To flatten it is to flatten the table first and then concatenate with the leading list. In this way we flatten the brancher into

$$\circ_1 - \circ_2 - \circ_3 - \circ_5 - \circ_4$$

Having analyzed the basic cases, now we are ready to flatten arbitrary branchers, such as this:

$$\circ_1 - \circ_2 - \circ_3 - \circ_4 - \left\{ \begin{array}{l} \circ_5 - \circ_6 - \circ_7 \\ \circ_8 - \circ_9 - \circ_{10} - \left\{ \begin{array}{l} \circ_{11} - \circ_{12} - \circ_{13} \\ \circ_{14} - \circ_{15} - \circ_{16} - \left\{ \begin{array}{l} \circ_{17} - \circ_{18} \\ \circ_{19} - \circ_{20} \end{array} \right. \end{array} \right. \\ \left\{ \begin{array}{l} \circ_{21} - \circ_{22} - \circ_{23} \\ \circ_{24} - \circ_{25} - \circ_{26} \end{array} \right. \end{array} \right.$$

We highlight some steps. We start from a sub-structure that is a table of lists, and flatten it by interleaved merge of the lists. Because a brancher is by definition finite, it is impossible that every table recursively contains a table. Therefore we can always find a table that contains only lists. We find a table of lists, which is circled below, and we shall flatten it.

$$\circ_1 - \circ_2 - \circ_3 - \circ_4 - \left\{ \begin{array}{l} \circ_5 - \circ_6 - \circ_7 \\ \circ_8 - \circ_9 - \circ_{10} - \left\{ \begin{array}{l} \circ_{11} - \circ_{12} - \circ_{13} \\ \circ_{14} - \circ_{15} - \circ_{16} - \boxed{\left\{ \begin{array}{l} \circ_{17} - \circ_{18} \\ \circ_{19} - \circ_{20} \end{array} \right.} \end{array} \right. \\ \left\{ \begin{array}{l} \circ_{21} - \circ_{22} - \circ_{23} \\ \circ_{24} - \circ_{25} - \circ_{26} \end{array} \right. \end{array} \right.$$

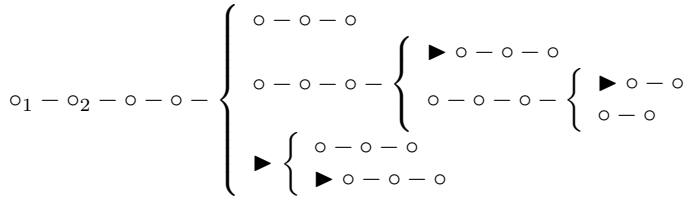
Once all child-tables are flattened, the parent-table becomes a table of lists, which we know how to flatten.

$$\circ_1 - \circ_2 - \circ_3 - \circ_4 - \left\{ \begin{array}{l} \circ_5 - \circ_6 - \circ_7 \\ \circ_8 - \circ_9 - \circ_{10} - \boxed{\left\{ \begin{array}{l} \circ_{11} - \circ_{12} - \circ_{13} \\ \circ_{14} - \circ_{15} - \circ_{16} - \circ_{17} - \circ_{19} - \circ_{18} - \circ_{20} \end{array} \right.} \\ \left\{ \begin{array}{l} \circ_{21} - \circ_{22} - \circ_{23} \\ \circ_{24} - \circ_{25} - \circ_{26} \end{array} \right. \end{array} \right.$$

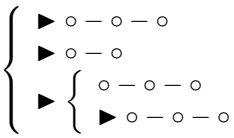
Repeat this process and we will finally get a list.

2.3 Merging Binary-state Branchers

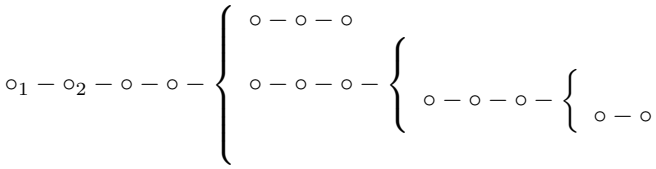
Just like merging single-state branchers, we want to flatten the two binary-state branchers before merging them. Without loss of generality, let us regard branches marked by the ▶ symbol as having state B, and the rest branches — state A. We want to merge all state A branches into a list l_A, and collect all state B branches into a single table $\mathcal{T}_B$. The list l_A and the table $\mathcal{T}_B$ is then connected, which is the result of flattening. For example, consider



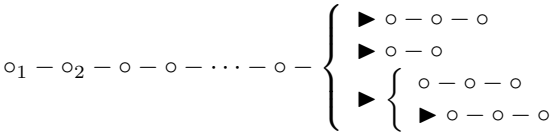
We first pick out all branches marked by ▶ and put them into a single table. If there are nested ▶, we do not go down, but stop at the top-level ▶. We get



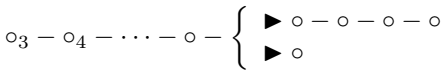
and the remaining part is a single-state brancher.



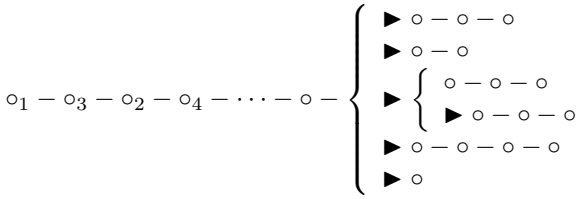
We flatten this single-state brancher and connect the resulting list to the collection of ▶ branches. Thus the original binary-state brancher is flattened into



To merge it with another flattened binary-state brancher, say, this one:



we perform interleaved merge with the leading lists, and pile the two tables, and connect to get



Since r-stream is binary-state brancher extended to allow infinite depth, by now we can see intuitively how interleaved merge works on r-streams.

3 Programming Experience

Where does our intuition, about the shape of r-streams and the work of their interleaved merge, come from? We start with the r-stream type definition, and in several steps remove its less essential features and reorganize the code to expose its very nature. On those simplified, refactored and essence-preserving versions of r-stream, we define interleaved merge and observe immediately how it works.

The language is OCaml version 4.12. R-stream is given by:

```
(* rstream.mli *)
type 'a t =
  | Nil
  | Cons    of 'a * 'a t
  | Thunk   of 'a thunk
  | Waiting of 'a suspended list
and 'a thunk = unit -> 'a t
and 'a suspended = {is_ready: unit -> bool; zz: 'a thunk}
```

Compared with the type definition of a list,

```
(* list.mli *)
type 'a t = Nil | Cons of 'a * 'a t
```

r-stream is still a regular recursive type, but has two extra value constructors `Thunk` and `Waiting`. A conventional lazy list[1] library such as OCaml's `Stdlib.Seq` does not have an expicit `Thunk` value constructor, but makes all list tails as well as the top-level list itself by default a thunk[2]. The explicit `Thunk` value constructor of r-stream allows mixing lazy and eager lists. The `Waiting` value constructor is for tabling which can be indispensable in certain relational programming tasks such as the jeep problem[3].

Therefore the most interesting part of r-stream is `Waiting`. To help us focus on this part, we define r-seq that is basically r-stream without the `Thunk` value constructor.

3.1 From R-stream to R-seq

Besides not having the `Thunk` value constructor, the concrete definition of r-seq differs from r-stream in several other ways. These are the changes that we introduce for pedagogical purposes. The effort is to disassemble the concept of an r-seq into more elementary parts. We now identify and highlight these parts (there are four) to make clearer the structure of an r-seq.

First, we generalize the `suspended` type to a type of binary-state (or boolean-state) values. The intuition is that the value is *conditional*, so that what we do to it depends on a boolean test on its state. This gives the `Cond.t` type consructor — a conceptual component of r-seq.

[1] The terms "stream" and "lazy list" are interchangeable.

[2] A *thunk* is an OCaml expression starting with `fun()->`.

[3] `https://github.com/YueLiPicasso/intro_ocaml/tree/master/OCanren_exercises/JeepProblem`

```
(* cond.mli *)
type 'a t = {btest  : bool;
             value  : 'a  }
```

The second conceptual component of an r-seq is the constructive core, given as `Abstract.t`.

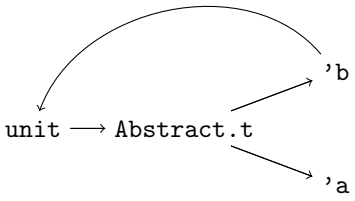
```
(* rseq.mli *)
module Abstract : sig
  type ('a, 'b) t =
      Nil
    | Cons  of 'a * 'b
    | Table of 'b Cond.t list
end
```

Note that we use `Table` instead of `Waiting` to give this type a wider appeal. Also, thanks to the identification of the `Cond.t` component, now it is clear that there are two channels for recursion indicated by the two occurrences of `'b` in the type representation.

The third and fourth component of r-seq are the thunk and the back loop, given together by the type equation:

```
(* rseq.mli *)
type 'a t = unit -> ('a, 'b) Abstract.t as 'b
```

In picture, `'a Rseq.t` is[4]



Now we summarize the five conceptual components that we identified for an r-stream:

1. The `Thunk` value constructor.
2. The conditional type `Cond.t`.
3. The non-recursive, constructive core `Abstract.t`.
4. The thunk `fun()->`.
5. The back loop.

R-seq removes component 1.

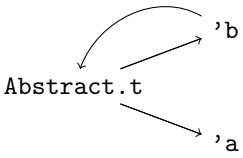
[4]We could see that `'a Rseq.t` and `'b` both refer to the same type.

3.2 From R-seq to Binary-state Brancher

We now provide an eager version of r-seq which is r-seq without thunks.This is also known as binary-state brancher. The motivation is to allow printing a finite r-seq value entirely in the OCaml REPL, otherwise the thunks would hide most of the interesting details. This is particularly helpful when we need to observe, given two r-seq's, the result of interleaved merge.

```
(* rseq.mli *)
module Eager : sig
  type 'a t = ('a, 'b) Abstract.t as 'b
end
```

In picture, `'a Rseq.Eager.t` is



A binary-state brancher thus removes components 1 and 4 from an r-stream. Worth noting that both types `Rseq.t` and `Rseq.Eager.t` are capable of typing infinite objects, the difference is that with thunks computations over infinite objects can be properly controlled.

3.3 From Binary-state Brancher to Single-state Brancher

The line "`Table of 'b Cond.t list`" in the definition of `Abstract.t` suggests that we can further remove the distinction of two states by simply deleting "`Cond.t`" from that line. Then all table items belong to the same and only state. This way we can ignore the handling of ▶ items and focus on how the ordinary items are merged with interleaving.

```
(* rseqnc.mli *)

type ('a, 'b) node =
    Nil
  | Cons  of 'a * 'b
  | Table of 'b list

type 'a t = ('a, 'b) node as 'b
```

The "`nc`" in "`rseqnc`" refers to "no conditional values". This is single-state brancher — the very core of r-stream, removing components 1, 2 and 4 , but only keeping the components 3 and 5.

3.4 Observing Interleaved Merge

By adapting the interleaved merge function for r-stream (available from the OCanren source distribution), we have those for single and binary state branch-

ers, given below[5]. By providing test data to them we can observe how interleaved merge works. This is where our intuition described in Sec.2 comes from.

```
(* rseqnc.ml, single-state brancher *)
let rec interleave xs ys =
  match xs with
  | Nil -> ys
  | Cons (h, t) -> Cons (h, interleave ys t)
  | Table nodes ->
    match flatten nodes, ys with
    | Table a , Table b -> Table (a @ b)
    | Table _ , _       -> interleave ys xs
    | l, _              -> interleave l ys
and flatten = function
  | []      -> Table []
  | n :: [] -> n
  | n :: ns -> interleave n (Table ns)
```

```
(* rseq.ml, binary-state brancher *)
let rec interleave xs ys =
    match xs with
    | Nil -> ys
    | Cons (h, t) -> Cons (h, interleave ys t)
    | Table rseqs ->
      match table_flatten rseqs, ys with
      | Table a , Table b -> Table (a @ b)
      | Table _ , _       -> interleave ys xs
      | l, _              -> interleave l ys
  and table_flatten = fun rseqs ->
      match find_map Cond.to_option rseqs with
    | Some rseq, []  -> rseq
    | Some rseq, rem -> interleave rseq (Table rem)
    | None     , _   -> Table rseqs
```

4 Beyond Intuition

Notation	Meaning & Example
l	$\circ_1 - \circ_2$
$\mathcal{T}(l,l)$	$\left\{ \begin{array}{l} \circ_1 - \circ_2 \\ \circ_1 - \circ_2 \end{array} \right.$
$l\,\mathcal{T}(l,l)$	$\circ_1 - \circ_2 - \left\{ \begin{array}{l} \circ_1 - \circ_2 \\ \circ_1 - \circ_2 \end{array} \right.$
$l \oplus l\,\mathcal{T}(l,l)$	$(\circ_1 - \circ_2) \oplus \left(\circ_1 - \circ_2 - \left\{ \begin{array}{l} \circ_1 - \circ_2 \\ \circ_1 - \circ_2 \end{array} \right. \right)$
$(l \oplus l)\,\mathcal{T}(l,l)$	$\circ_1 - \circ_1 - \circ_2 - \circ_2 - \left\{ \begin{array}{l} \circ_1 - \circ_2 \\ \circ_1 - \circ_2 \end{array} \right.$

[5]We use a customized `find_map` function, which works like OCaml `Stdlib.List.find_map`, but additionally returns the rest of the list after the search. The conversion from `'a Cond.t` to the type `'a option` is done in the obvious way.

After setting up the notation, we can formalize some useful theorems about interleaved merge and flattening of binary-state branchers. Below are some examples.

Flattening a table that has a single ▶ list we get the table itself.

$$\mathring{\mathcal{T}}(\blacktriangleright\, l) = \mathcal{T}(\blacktriangleright\, l)$$

Flattening a table of two lists, one of which has ▶, the result is independent from the order of the lists in the table. Moreover, the result is the same as the interleaved merge of the non-▶ list with a singletion table built from the ▶ list, which is the connection of the two.

$$\mathring{\mathcal{T}}(l_1, \blacktriangleright\, l_2) = \mathring{\mathcal{T}}(\blacktriangleright\, l_2, l_1) = l_1 \oplus \mathcal{T}(\blacktriangleright\, l_2) = l_1 \mathcal{T}(\blacktriangleright\, l_2)$$

These theorems may not be the most general ones that we can formulate for binary-state branchers, but they can be proved using some inductive argument based on the definitions of the data type and the functions for interleaved merge and flattening. Below are two more theorems.

$$l_1 \mathcal{T}(\blacktriangleright\, l) \oplus l_2 = l_1 \oplus l_2 \mathcal{T}(\blacktriangleright\, l) = (l_1 \oplus l_2)\mathcal{T}(\blacktriangleright\, l)$$

$$\mathring{\mathcal{T}}(l_1, \ldots, l_n) = l_1 \oplus (\ldots (l_{n-1} \oplus l_n))(n \geq 2)$$

It could be a good exercise to come up with more interesting and general theorems about branchers and r-streams and prove them in a mechanized theorem prover.

5 Alternative R-stream Definitions

The current r-stream definition merely translates Scheme version to OCaml and there are indeed alternatives. The single-state brancher (rseqnc) removes the boolean test that checks for the presence of new answers in the table; the `Table` only keeps thunks, and it returns the constructor `Nil` when there is no answer available. This looks more concise and math-y, and can be an alternative r-stream definition.

It is possible to make one step further and define type `node` as follows.

```
type ('a, 'b) node =
| Nil
| Cons of 'a * 'b
| Table of 'b * 'b   (* no list here *)
```

Now the `Table` constructor does not store a list of sequences, and instead contains a pair of thunk values. It is believed that the reason why it was storing the list in Scheme is very purely pragmatic (for some performance optimizations); essentially, storing a list allows to accumulate several suspended sequences in one place; from the point of view of clarity and conciseness (and for simpler formal reasoning), it might be beneficial to drop this list. Then it leads us to the following definition of r-streams,

```
type 'a stream =
| Nil
| Cons of 'a * 'a stream
| Thunk of unit -> 'a stream
| Table of 'a stream * 'a stream
```

where the first component of `Table` is a special handle that can be pulled to get new answers. In tabling implementation, the purpose of `Suspended` was to have a handle to *poll* for new answers in the table; it is possible to replace this `Suspended` by a thunk and instead of polling, just *pull* and see if there are new answers available. If there is no new answer, the pull will return a new thunk, so that we can try again later. Otherwise, it can return a `Cons` with some new answers, and a new `Thunk` at the end. The tricky part in this implementation would be to detect when a fixpoint is reached and the table will not produce any new answer.