Backtracking, Interleaving, and Terminating Monad Transformers

Andrew Lelechenko
1@dxdy.ru

Barclays, London

Papers We Love, London, 29.10.2019

Backtracking, Interleaving, and Terminating Monad Transformers

(Functional Pearl)

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Harvard University
ccshan@post.harvard.edu

Daniel P. Friedman
Indiana University
dfried@indiana.edu

Amr Sabry
Indiana University
sabry@indiana.edu

Abstract

We design and implement a library for adding backtracking computations to any Haskell monad. Inspired by logic programming, our library provides, in addition to the operations required by the *MonadPlus* interface, constructs for fair disjunctions, fair conjunctions, conditionals, pruning, and an expressive top-level interface. Implementing these additional constructs is easy in models of backtracking based on streams, but not known to be possible in continuation-based models. We show that all these additional constructs can be *generically* and monadically realized using a single primitive *msplit*. We present two implementations of the library: one using success and failure continuations; and the other using control operators for manipulating delimited continuations.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.6 [*Programming Techniques*]: Logic Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives

General Terms Languages

Keywords continuations, control delimiters, Haskell, logic programming, Prolog, streams.

monad interface, has found many practical applications [20], ranging from those envisioned for McCarthy's *amb* operator [15] and its descendents [25], to transactions [28], pattern combinators [27], and failure handling [21].

In a functional pearl [11], Hinze describes backtracking monad transformers that support non-deterministic choice and a Prolog-like *cut* with delimited extent. Hinze aimed to systematically derive his monad transformers in two ways, yielding a term implementation and a (more efficient) context-passing implementation. The most basic backtracking operations, failure and non-deterministic choice, are indeed systematically derived from their specifications. But when it came to *cut*, creative insight was still needed. Furthermore, the resulting term implementation is no longer based on a free term algebra, and the corresponding context-passing implementation performs pattern-matching on the context. As Hinze notes [11], this context-passing implementation differs from a traditional continuation-passing-style (CPS) implementation that handles continuations abstractly. In other words, the implementation is not directly amenable to a direct-style implementation using control operators.

Most existing backtracking monad transformers, including the ones presented by Hinze, suffer from three deficiencies in practical use: unfairness, confounding negation with pruning, and a limited ability to collect and operate on the final answers of a non-deterministic computation. First, the straightforward depth-first search

Better List Monad

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Harvard University
ccshan@post.harvard.edu

Daniel P. Friedman
Indiana University
dfried@indiana.edu

Amr Sabry
Indiana University
sabry@indiana.edu

Abstract

We design and implement a library for adding backtracking computations to any Haskell monad. Inspired by logic programming, our library provides, in addition to the operations required by the *MonadPlus* interface, constructs for fair disjunctions, fair conjunctions, conditionals, pruning, and an expressive top-level interface. Implementing these additional constructs is easy in models of backtracking based on streams, but not known to be possible in continuation-based models. We show that all these additional constructs can be *generically* and monadically realized using a single primitive *msplit*. We present two implementations of the library: one using success and failure continuations; and the other using control operators for manipulating delimited continuations.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.6 [*Programming Techniques*]: Logic Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives

General Terms Languages

Keywords continuations, control delimiters, Haskell, logic programming, Prolog, streams.

monad interface, has found many practical applications [20], ranging from those envisioned for McCarthy's *amb* operator [15] and its descendents [25], to transactions [28], pattern combinators [27], and failure handling [21].

In a functional pearl [11], Hinze describes backtracking monad transformers that support non-deterministic choice and a Prolog-like *cut* with delimited extent. Hinze aimed to systematically derive his monad transformers in two ways, yielding a term implementation and a (more efficient) context-passing implementation. The most basic backtracking operations, failure and non-deterministic choice, are indeed systematically derived from their specifications. But when it came to *cut*, creative insight was still needed. Furthermore, the resulting term implementation is no longer based on a free term algebra, and the corresponding context-passing implementation performs pattern-matching on the context. As Hinze notes [11], this context-passing implementation differs from a traditional continuation-passing-style (CPS) implementation that handles continuations abstractly. In other words, the implementation is not directly amenable to a direct-style implementation using control operators.

Most existing backtracking monad transformers, including the ones presented by Hinze, suffer from three deficiencies in practical use: unfairness, confounding negation with pruning, and a limited ability to collect and operate on the final answers of a non-deterministic computation. First, the straightforward depth-first search

Better list comprehensions

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Harvard University
ccshan@post.harvard.edu

Daniel P. Friedman
Indiana University
dfried@indiana.edu

Amr Sabry
Indiana University
sabry@indiana.edu

Abstract

We design and implement a library for adding backtracking computations to any Haskell monad. Inspired by logic programming, our library provides, in addition to the operations required by the *MonadPlus* interface, constructs for fair disjunctions, fair conjunctions, conditionals, pruning, and an expressive top-level interface. Implementing these additional constructs is easy in models of backtracking based on streams, but not known to be possible in continuation-based models. We show that all these additional constructs can be *generically* and monadically realized using a single primitive *msplit*. We present two implementations of the library: one using success and failure continuations; and the other using control operators for manipulating delimited continuations.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.6 [*Programming Techniques*]: Logic Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives

General Terms Languages

Keywords continuations, control delimiters, Haskell, logic programming, Prolog, streams.

monad interface, has found many practical applications [20], ranging from those envisioned for McCarthy's *amb* operator [15] and its descendents [25], to transactions [28], pattern combinators [27], and failure handling [21].

In a functional pearl [11], Hinze describes backtracking monad transformers that support non-deterministic choice and a Prolog-like *cut* with delimited extent. Hinze aimed to systematically derive his monad transformers in two ways, yielding a term implementation and a (more efficient) context-passing implementation. The most basic backtracking operations, failure and non-deterministic choice, are indeed systematically derived from their specifications. But when it came to *cut*, creative insight was still needed. Furthermore, the resulting term implementation is no longer based on a free term algebra, and the corresponding context-passing implementation performs pattern-matching on the context. As Hinze notes [11], this context-passing implementation differs from a traditional continuation-passing-style (CPS) implementation that handles continuations abstractly. In other words, the implementation is not directly amenable to a direct-style implementation using control operators.

Most existing backtracking monad transformers, including the ones presented by Hinze, suffer from three deficiencies in practical use: unfairness, confounding negation with pruning, and a limited ability to collect and operate on the final answers of a non-deterministic computation. First, the straightforward depth-first search

List comprehension

- Mathematics:

$$\{(x, y) \mid x \in [1, 3], y \in [x, 4], x + y \leq 5\} = \\ = \{(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3)\}.$$

- Haskell:

```
[ (x, y) | x <- [1..3], y <- [x..4], x + y <= 5 ]
```

- Python:

```
{ (x, y)
  for x in range(1, 3)
  for y in range(x, 4)
  if x + y <= 5 }
```

- PostgreSQL:

```
SELECT x, y
FROM  GENERATE_SERIES(1, 3) AS x,
      GENERATE_SERIES(x, 4) AS y
WHERE x + y <= 5;
```

Infinite lists and $\cup$

- Mathematics:

$$\{x \mid x \in 2\mathbb{N}, x \equiv 0 \pmod 3\} = \{0, 6, 12 \dots\}.$$

- Haskell:

```
[ x | x <- [0, 2..], x `mod` 3 == 0 ]
```

- Mathematics:

$$\{x \mid x \in 2\mathbb{N} \cup 3\mathbb{N}, x \equiv 1 \pmod 2\} = \{3, 9, 15 \dots\}.$$

- Haskell:

```
[ x | x <- [0, 2..] ++ [0, 3..], x `mod` 2 == 1 ]
```

- But this computation stucks forever! Because `(++)` is *left-biased.*

Fair interleave aka unbiased $+\!\!+$

`interleave [0,2,4] [1,3,5,7,9] = [0,1,2,3,4,5,7,9]`

Simple implementation:

```
interleave :: [a] -> [a] -> [a]
interleave [] ys = ys
interleave xs [] = xs
interleave (x:xs) (y:ys) = x : y : interleave xs ys
```

Simpler implementation:

```
interleave :: [a] -> [a] -> [a]
interleave [] ys = ys
interleave (x:xs) ys = x : interleave ys xs
```

Bonus level: is `interleave` associative?

Infinite lists and $\times$

- Mathematics:

$$\{(x, y) \mid x \in \mathbb{N}, y \in \mathbb{N}, x > y\}$$

- Haskell:

```
[ (x, y) | x <- [0..], y <- [0..], x > y ]
```

- But this computation stucks forever! Imagine it as two nested loops:

```
for(int x = 0; ; x++)
  for(int y = 0; ; y++)
    if(x > y)
      printf("(%d, %d)", x, y);
```

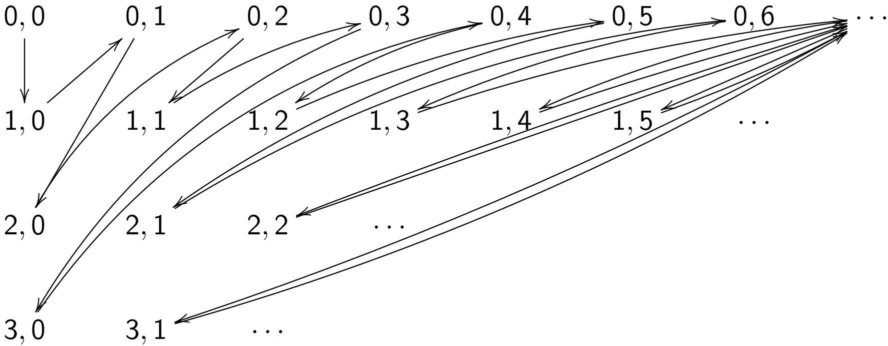
Unfair interweave aka simplified $\ggg$

```
interweave :: [a] -> [b] -> [(a, b)]
interweave [] ys = []
interweave (x:xs) ys =
  map (\y -> (x, y)) ys ++ interweave xs ys
```

$$
\begin{tikzcd}
0,0 \arrow[r] & 0,1 \arrow[r] & 0,2 \arrow[r] & 0,3 \arrow[r] & \cdots \arrow[dllll] \\
1,0 \arrow[r] & 1,1 \arrow[r] & 1,2 \arrow[r] & 1,3 \arrow[r] & \cdots \arrow[dllll] \\
2,0 \arrow[r] & 2,1 \arrow[r] & 2,2 \arrow[r] & 2,3 \arrow[r] & \cdots \arrow[dllll] \\
3,0 \arrow[r] & 3,1 \arrow[r] & 3,2 \arrow[r] & 3,3 \arrow[r] & \cdots
\end{tikzcd}
$$

Fair interweave aka simplified ≫−

```
interweave :: [a] -> [b] -> [(a, b)]
interweave [] ys = []
interweave (x:xs) ys =
  map (\y -> (x, y)) ys `interleave` interweave xs ys
```



Quantifier ∃

An integer is called *composite* if it has at least one non-trivial divisor (other than 1 and itself). E. g., 4 is the smallest composite.

$$\{x \mid x \in [2, 10], \exists y \in [2, x-1]\ x \equiv 0 \pmod{y}\} = \{4, 6, 8, 9, 10\}.$$

It is not easily expressible in Haskell:

```
> [(x, y) | x <- [2..10], y <- [2..x-1], x `mod` y == 0]
[(4,2),(6,2),(6,3),(8,2),(8,4),(9,3),(10,2),(10,5)]
```

```
> [x | x <- [2..10], y <- [2..x-1], x `mod` y == 0]
[4,6,6,8,8,9,10,10]
```

Fun fact: it is perfectly expressible in SQL:

```
SELECT x
FROM  GENERATE_SERIES(2, 10) AS x,
      GENERATE_SERIES(2, x-1) AS y
WHERE x % y = 0
GROUP BY x;
```

Quantifier ∃ and once

$$\{x \mid x \in [2,10], \exists y \in [2, x-1]\ x \equiv 0 \pmod{y}\} = \{4,6,8,9,10\}.$$

We can express ∃ via *nested list comprehensions:*

```
[ x | x <- [2..10],
  [ y | y <- [2..x-1], x `mod` y == 0 ] /= []
]
```

Even better: introduce once to keep track of the evidence of compositeness:

```
once :: [a] -> [a]
once [] = []
once (x:_) = [x]

[ (x, y1) | x <- [2..10], y1 <- once
  [ y | y <- [2..x-1], x `mod` y == 0 ]
]
```

Quantifier ∀

An integer is called *prime* if it is divisible only by 1 and itself.

$$\{x \mid x \in [2, 10], \forall y \in [2, x-1]\ x \not\equiv 0 \pmod{y}\} = \{2, 3, 5, 7\}.$$

We again need nested list comprehensions to express it in Haskell:

```
[ x | x <- [2..10],
  [ y | y <- [2..x-1], x `mod` y == 0 ] == []
]
```

Fun fact: it is yet again perfectly expressible in SQL.

```
SELECT x
FROM GENERATE_SERIES(2, 10) AS x
LEFT JOIN GENERATE_SERIES(2, x-1) AS y ON x % y = 0
WHERE y IS NULL;
```

Quantifier $\forall$ and `lnot`

Can we express $\forall$ via $\exists$? Apply De Morgan's law:

$$\{x \mid x \in [2,10], \forall y \in [2, x-1]\ x \not\equiv 0 \pmod{y}\} = \{x \mid x \in [2,10], \neg(\exists y \in [2, x-1]\ x \equiv 0 \pmod{y})\}.$$

Introduce `lnot` combinator (logical negation):

```
lnot :: [a] -> [()]
lnot [] = [()]
lnot (_:_) = []
```

```
[ x  | x <- [2..10], _ <- lnot
  [ y | y <- [2..x-1], x `mod` y == 0 ]
]
```

For alert readers: the paper uses a more general combinator `ifte`.

What's next?

- `interleave`, `interweave`, `once` and `lnot` form a pretty expressive DSL for backtracking and relational (logic) programming on lists.
- How to generalize this notion to other data types?
  - Infinite streams.
  - Probability distributions: `[CatAlive, CatDead]` vs. `[(0.7, CatAlive), (0.3, CatDead)]`.
- How to handle side effects?
  - Read input lists from IO (disk, network, whatever).
  - Cache intermediate computations.
  - Track progress and write results.

Back to Kiselyov et al., 2005

- Introduce a type class `MonadLogic`, consisting of `interleave`, `interweave`, `once` and `lnot`, for an arbitrary `MonadPlus` (not just for lists).
- Propose laws, allowing equational reasoning about relational programming.
- Express all functions by means of a single combinator `msplit`:

```
msplit :: [a] -> [Maybe (a, [a])]
msplit [] = [Nothing]
msplit (x:xs) = [Just (x, xs)]
```

- Explain how to add `MonadLogic` capabilities atop any other `Monad`, giving rise to `LogicT` monad transformer.

Final remarks

- I happened to maintain a Haskell implementation of `LogicT` at hackage.haskell.org/package/logict
- Modern developments in Haskell and relational programming: twitch.tv/ekmett/videos
- SQL is a logic programming language.

@ andrew.lelechenko@gmail.com

Bodigrim

Thank you!