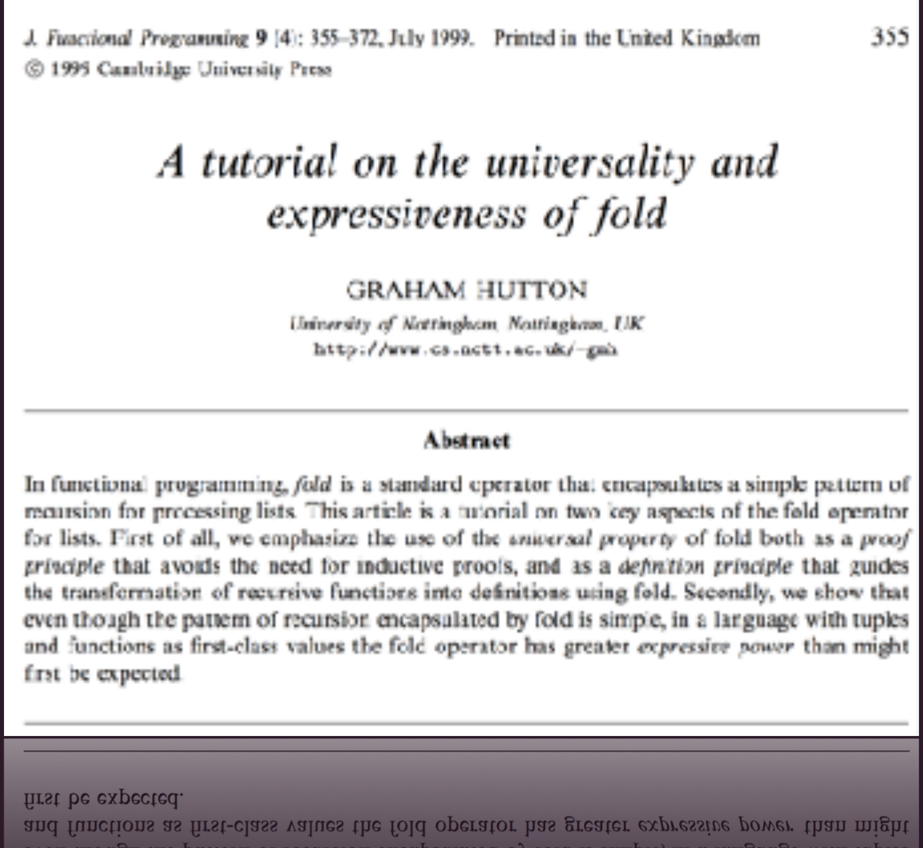
Transducers Internals

Clojure Exchange 2017 - Workshops

On the universality of fold

- **reduce** as the prototypical recursive iterative process
- Redefinition of sequential processing in terms of **reduce**
- Extract transformations and I/O details
- Possible? Let's refactor **map** and **filter** to find out



J. Functional Programming 9 (4): 355–372, July 1999. Printed in the United Kingdom 355
© 1999 Cambridge University Press

A tutorial on the universality and expressiveness of fold

GRAHAM HUTTON
University of Nottingham, Nottingham, UK
http://www.cs.nott.ac.uk/~gmh

Abstract

In functional programming, *fold* is a standard operator that encapsulates a simple pattern of recursion for processing lists. This article is a tutorial on two key aspects of the fold operator for lists. First of all, we emphasize the use of the *universal property* of fold both as a *proof principle* that avoids the need for inductive proofs, and as a *definition principle* that guides the transformation of recursive functions into definitions using fold. Secondly, we show that even though the pattern of recursion encapsulated by fold is simple, in a language with tuples and functions as first-class values the fold operator has greater *expressive power* than might first be expected.

Step 1: express like reduce

```clojure
(defn map* [f result coll]
  (if (not= '() coll)
    (map* f
      (f result (first coll))
      (rest coll))
    result))

; use like
(map*
  (fn [result el] (conj result (inc el)))
  []
  (range 10))
```

```clojure
(defn filter* [f result coll]
  (if (not= '() coll)
    (filter* f
      (f result (first coll))
      (rest coll))
    result))

(filter*
  (fn [result el]
    (if (odd? el)
      (conj result el)
      result))
  []
  (range 10))
```

Step 2: rename it reduce, which is what it is!

```
(defn reduce* [f result coll]
  (if (not= '() coll)
    (reduce* f
      (f result (first coll))
      (rest coll))
    result))

;; And use like this:
(reduce*
  (fn [result el] (conj result (inc el)))
  []
  (range 10))
```

```
(defn reduce* [f result coll]
  (if (not= '() coll)
    (reduce* f
      (f reduce
        (first coll))
        (rest coll))
    result))

(reduce*
  (fn [result el]
    (if (odd? el)
      (conj result el)
      result))
  []
  (range 10))
```

Step 3: That's just normal reduce!

```
;; We can throw away our reduce* and just
;; use Clojure's:

;; map
(reduce
  (fn [result el]
    (conj result (inc el)))
  []
  (range 10))
```

```
;; filter
(reduce
  (fn [result el]
    (if (odd? el)
      (conj result el) result))
    []
    (range 10))
```

Step 4: separate essence into fn

```clojure
(defn mapping [result el]
  (conj result (inc el)))


(reduce mapping [] (range 10))
```

```clojure
(defn filtering [result el]
  (if (odd? el)
    (conj result el)
    result))

(reduce filtering [] (range 10))
```

Step 5: introduce "rf", extract "conj" away

```
(defn mapping [rf]
  (fn [result el]
    (rf result (inc el))))


(reduce (mapping conj) [] (range 10))
```

```
(defn filtering [rf]
  (fn [result el]
    (if (odd? el)
      (rf result el)
      result)))

(reduce (filtering conj) [] (range 10))
```

Step 6: introduce param "f" and "pred?"

```
(defn mapping [f]
  (fn [rf]
    (fn [result el]
      (rf result (f el)))))


(reduce
  ((mapping inc) conj)
  []
  (range 10))
```

```
(defn filtering [pred?]
  (fn [rf]
    (fn [result el]
      (if (pred? el)
        (rf result el)
        result))))


(reduce
  ((filtering odd?) conj)
  []
  (range 10))
```

Step 7: finally, extract transduce* fn

```
(defn mapping [f]
  (fn [rf]
    (fn [result el]
      (rf result (f el)))))
```

```
(defn transduce* [xf rf coll]
  (reduce (xf rf) (rf) coll))
```

```
;; Use like
(transduce* (mapping inc) conj (range 10))
```

```
(defn filtering [pred]
  (fn [rf]
    (fn [result el]
      (if (pred? el)
        (rf result el)
        result))))
```

```
(defn transduce* [xf rf coll]
  (reduce (xf rf) (rf) coll))
```

```
(transduce* (filtering odd?) conj (range 10))
```

Creating your own transducers

- Our **mapping** and **filtering** are the same in the std lib.
- But that's not all you need to know.
- A "good transducer" also knows how to behave when other transducers are around.

Some aspects about designing a transducer

- Deal with the end of the reduction (1-arg arity).
- Providing an initial value (0-arg arity).
- Where to initialize state (if needed).
- How to terminate early (if needed).
- Surrounding transducers awareness (mandatory calls).

Resources

- A tutorial on the universality and expressiveness of fold
- uSwitch Labs transducers articles

Lab 02
Custom Transducers

Goal of Lab2

- **Task 1**: create a logging transducer to print intermediate results from a transducers chain.
- **Task 2**: create a stateful moving average transducer

Open transducers-workshop.lab02 to start.