EXTENDING AND COMPLETING ŸAUHAU

Justus Adam, supervised by Sebastian Ertel and Andres Goens

Motivation

```
(defn common-friends [x y]
  (intersection (friends-of x) (friends-of y)))

(defn friends-of [x]
  (fetch (FriendsOf. x)))
```

Motivation

```clojure
(defn common-friends [x y]
  (intersection (friends-of x) (friends-of y)))

(defn friends-of [x]
  (fetch (FriendsOf. x)))
```

- Batching, Caching and Concurrency desired

Motivation

```
(defn common-friends [x y]
  (intersection (friends-of x) (friends-of y)))

(defn friends-of [x]
  (fetch (FriendsOf. x)))
```

- Batching, Caching and Concurrency desired
- Retaining concise and straight forward code

Motivation

```
(defalgo common-friends [x y]
  (intersection (friends-of x) (friends-of y)))

(defalgo friends-of [x]
  (fetch (mk-req (FriendsOf. x) data-source)))
```

- Batching, Caching and Concurrency desired
- Retaining concise and straight forward code
- Ÿauhau solves issue with minimal difference in code

Goals

Goals

Current status

- Ÿauhau supports a base transformation on a dataflow graph

Goals

Current status

- Ÿauhau supports a base transformation on a dataflow graph
- The structure of the graph does not reflect control flow

Goals

Current status

- Ÿauhau supports a base transformation on a dataflow graph
- The structure of the graph does not reflect control flow

Goals

Current status

- Ÿauhau supports a base transformation on a dataflow graph
- The structure of the graph does not reflect control flow

Tasks

- Handling control flow

Goals

Current status

- Ÿauhau supports a base transformation on a dataflow graph
- The structure of the graph does not reflect control flow

Tasks

- Handling control flow
- Iteration/mapping with `smap`

Goals

Current status

- Ÿauhau supports a base transformation on a dataflow graph
- The structure of the graph does not reflect control flow

Tasks

- Handling control flow
- Iteration/mapping with `smap`
- Conditional execution with `if`

Goals

Current status

- Ÿauhau supports a base transformation on a dataflow graph
- The structure of the graph does not reflect control flow

Tasks

- Handling control flow
- Iteration/mapping with `smap`
- Conditional execution with `if`
- Semantics of side effects (writes)

Table of Contents

Table of Contents

Application

- Large scale, distributed systems (Facebook, Amazon, Twitter)
- Haskell library Haxl[1]
- Clojure library Muse[2]
- Scala implementation Stitch by Twitter[3] (closed source)
- Ÿauhau as an Ohua plugin[4]

1 Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is no fork: an abstraction for efficient, concurrent, and concise data access. ICFP '14.

2 Alexey Kachayev. 2015. Reinventing Haxl: Efficient, Concurrent and Concise Data Access. https://www.youtube.com/watch?v=T-oekV8Pwv8

3 Jake Donham. 2014. Introducing Stitch. https://www.youtube.com/watch?v=VVpmMfT8aYw

4 Sebastian Ertel, Andrés Goens, Justus Adam and Jeronimo Castrillon. Ÿauhau: Concise Code and Efficient I/O Straight from Dataflow. POPL '17. In submission.

Table of Contents

Ohua

- Ÿauhau hooks into the Ohua[5] compiler

[5] Sebastian Ertel, Christof Fetzer, and Pascal Felber. 2015. Ohua: Implicit Dataflow Programming for Concurrent Systems. PPPJ '15
[6] Arvind and David E. Culler. 1986. Dataflow architectures. In Annual review of computer science vol. 1
[7] J. B. Dennis. 1974. First version of a data flow procedure language. In Programming Symposium, Proceedings Colloque sur la Programmation

Ohua

- Ÿauhau hooks into the Ohua[5] compiler
- Parallelisation framework based on dataflow[6][7]

[5] Sebastian Ertel, Christof Fetzer, and Pascal Felber. 2015. Ohua: Implicit Dataflow Programming for Concurrent Systems. PPPJ '15

[6] Arvind and David E. Culler. 1986. Dataflow architectures. In Annual review of computer science vol. 1

[7] J. B. Dennis. 1974. First version of a data flow procedure language. In Programming Symposium, Proceedings Colloque sur la Programmation

Ohua

- Ÿauhau hooks into the Ohua[5] compiler
- Parallelisation framework based on dataflow[6][7]
- Stateful functions and algorithms (`defalgo`) as abstraction

[5] Sebastian Ertel, Christof Fetzer, and Pascal Felber. 2015. Ohua: Implicit Dataflow Programming for Concurrent Systems. PPPJ '15

[6] Arvind and David E. Culler. 1986. Dataflow architectures. In Annual review of computer science vol. 1

[7] J. B. Dennis. 1974. First version of a data flow procedure language. In Programming Symposium, Proceedings Colloque sur la Programmation

Base transformation

```
(defalgo common-friends [x y]
  (intersection
    (friends-of x)
    (friends-of y)))

(defalgo friends-of [x]
  (fetch (mk-req (FriendsOf. x)
                 data-source)))
```

Base transformation

- Rewrites high level algorithms

```
(defalgo common-friends [x y]
  (intersection
    (friends-of x)
    (friends-of y)))

(defalgo friends-of [x]
  (fetch (mk-req (FriendsOf. x)
                 data-source)))
```

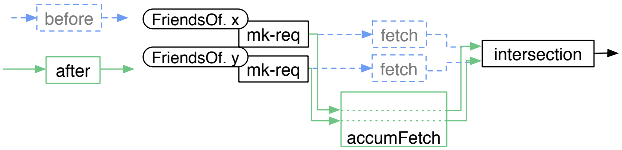
Ÿauhau base transformation

- Rewrites high level algorithms
- Operates on dataflow ir

```
(let [[x y] (algo-in)
      a (mk-req (FriendsOf. x) data-source)
      b (mk-req (FriendsOf. y) data-source)
      c (fetch a)
      d (fetch b)
      e (intersection a b)]
  e)
```

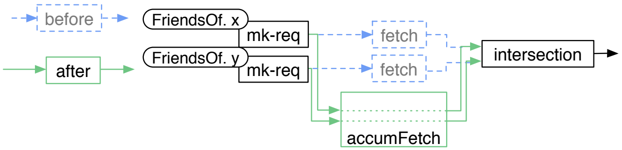
Ÿauhau base transformation

- Rewrites high level algorithms
- Operates on dataflow ir
- Traverses dataflow graph along dependencies



Ÿauhau base transformation

- Rewrites high level algorithms
- Operates on dataflow ir
- Traverses dataflow graph along dependencies
- Collect fetches, replace with accumulator



Ÿauhau base transformation

- Rewrites high level algorithms
- Operates on dataflow ir
- Traverses dataflow graph along dependencies
- Collect fetches, replace with accumulator
- Transformation is very simple and naive

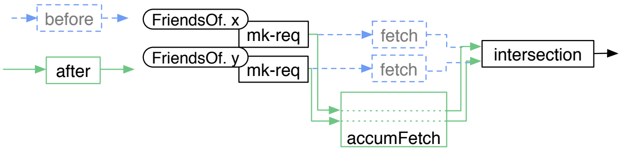


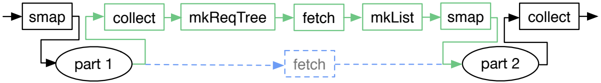
Table of Contents

smap rewrite

- Split inner graph around fetch

smap rewrite

- Split inner graph around fetch
- insert `collect` and `smap` around fetch



smap rewrite

- Split inner graph around fetch
- insert `collect` and `smap` around fetch
- Build tree of requests

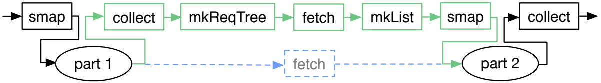
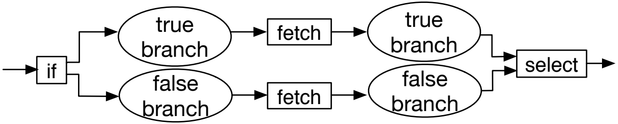


Table of Contents

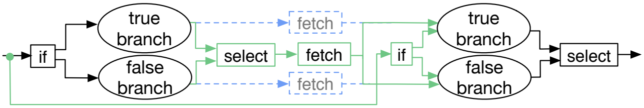
if rewrite

- Split inner graph around fetch



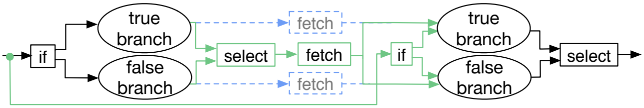
if rewrite

- Split inner graph around fetch
- Replace two fetches with one being selectively given the active request



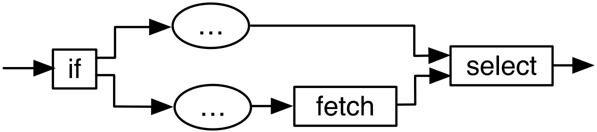
if rewrite

- Split inner graph around fetch
- Replace two fetches with one being selectively given the active request
- Push result to the continuation of the active branch using the initial condition



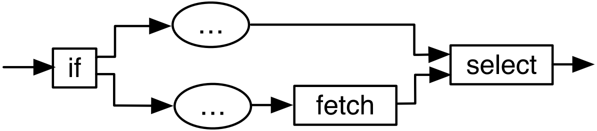
if rewrite

- Problem if unequal number of fetches on branches



if rewrite

- Problem if unequal number of fetches on branches
- Solution: Insert extra fetches



if rewrite

- Problem if unequal number of fetches on branches
- Solution: Insert extra fetches
- Use NoOp (empty) requests

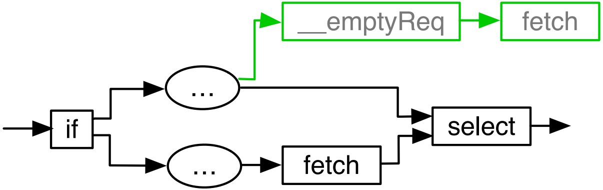


Table of Contents

Generalising to Context

- Handle arbitrary graphs with conditionals, maps etc

Generalising to Context

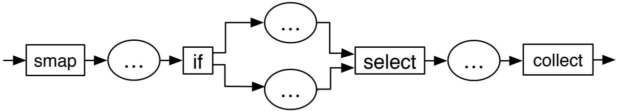
- Handle arbitrary graphs with conditionals, maps etc
- These structures are often interleaved

Generalising to Context

- Handle arbitrary graphs with conditionals, maps etc
- These structures are often interleaved
- Ease handling of complex nested stacks

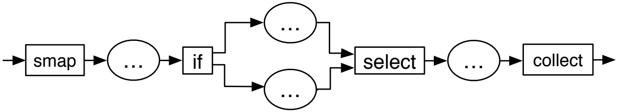
Context

- Hidden graph structures unified into context



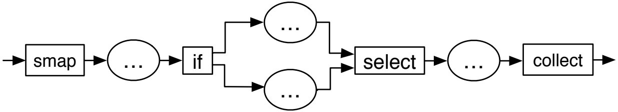
Context

- Hidden graph structures unified into context
- Context is property of subgraphs emerging from node labels



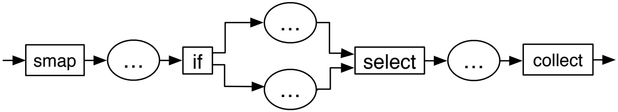
Context

- Hidden graph structures unified into context
- Context is property of subgraphs emerging from node labels
- Has a marker for begin and end



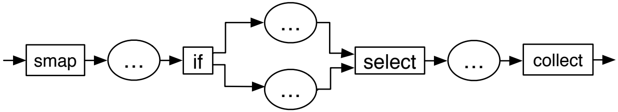
Context

- Hidden graph structures unified into context
- Context is property of subgraphs emerging from node labels
- Has a marker for begin and end
- Inherited property



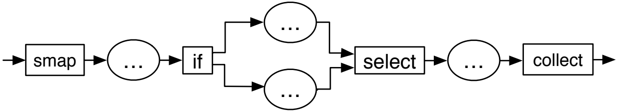
Context

- Hidden graph structures unified into context
- Context is property of subgraphs emerging from node labels
- Has a marker for begin and end
- Inherited property
- Subcontexts are always fully enclosed



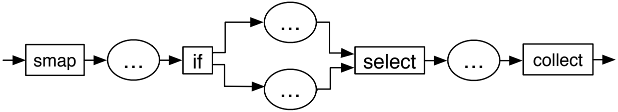
Finding context

- Context recognition is done at compile time



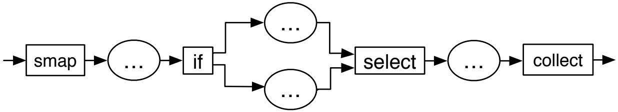
Finding context

- Context recognition is done at compile time
- If a context opening node is found annotate subgraph



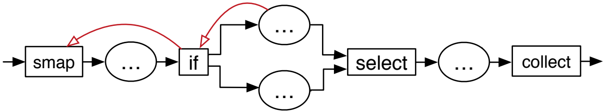
Finding context

- Context recognition is done at compile time
- If a context opening node is found annotate subgraph
- Resolve full context stack by following the parent context references



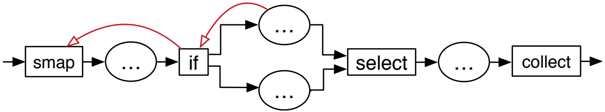
Finding context

- Context recognition is done at compile time
- If a context opening node is found annotate subgraph
- Resolve full context stack by following the parent context references



Finding context

- Context recognition is done at compile time
- If a context opening node is found annotate subgraph
- Resolve full context stack by following the parent context references
- Hinges on the "fully enclosing" property



Handling arbitrary graphs

- Calculate contexts for each fetch
- Contexts are unwound in order of descending nesting level
- Unwinding is interleaved

Table of Contents

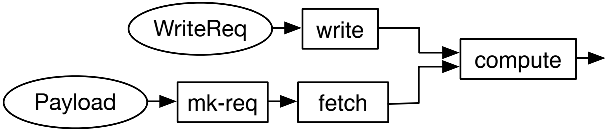
Side Effects in Algorithms

- Execution order depends entirely on data dependencies

```
(defalgo main []
  (let [write-result
          (write (WriteReq. "my-data")
                 data-source)
        fetch-data
          (fetch (mk-req (Payload. "constant")
                 data-source))]
    (compute fetch-data write-result)))
```

Side Effects in Algorithms

- Execution order depends entirely on data dependencies



Seq

- Execution order depends entirely on data dependencies
- Can be enforced using `seq` operator

```
(defalgo main []
  (let [write-result
          (write (WriteReq. "my-data")
                 data-source)
          enforced
          (seq write-result
               (fetch (mk-req
                        (Payload. "constant")
                        data-source)))]
    (compute enforced write-result)))
```

Side Effects across Algorithms

- Execution order depends entirely on data dependencies
- Can be enforced using `seq` operator
- Should be implicit from data dependencies

```
(defalgo does-read [x]
  (compute
    (fetch (mk-req (Payload. "constant")
                   data-source))
    x))
(defalgo does-write []
  (write (WriteReq. "my-data") data-source)
  ...)
(defalgo main []
  (does-read (does-write))
```

Enforcing Execution Order

- Exploring and evaluating different approaches

Enforcing Execution Order

- Exploring and evaluating different approaches
- Combining expected semantics and efficiency

Enforcing Execution Order

- Exploring and evaluating different approaches
- Combining expected semantics and efficiency

Enforcing Execution Order

- Exploring and evaluating different approaches
- Combining expected semantics and efficiency

Current approach

- `seq`‘ing fetches to algorithm boundary
- for ‘unconnected’ fetches following writes
- Vice versa for ‘unconnected’ writes following reads

Enforcing Execution Order

- Exploring and evaluating different approaches
- Combining expected semantics and efficiency

Current approach

- `seq`‘ing fetches to algorithm boundary
- for ‘unconnected’ fetches following writes
- Vice versa for ‘unconnected’ writes following reads

Open questions

- Always enabled?
- Scope?

Table of Contents

Verifying correctness (program semantics) and performance in comparison to Haxl and Muse for:

Experiments and Evaluation

Verifying correctness (program semantics) and performance in comparison to Haxl and Muse for:

1. Modularised graphs (functions/algorithms)
2. Graphs with map operations (`smap`)
3. Graphs with conditionals (`if`)

Experiments and Evaluation

Verifying correctness (program semantics) and performance in comparison to Haxl and Muse for:

1. Modularised graphs (functions/algorithms)
2. Graphs with map operations (`smap`)
3. Graphs with conditionals (`if`)

This requires extensions to our random code generator to allow generation of correct code with

Experiments and Evaluation

Verifying correctness (program semantics) and performance in comparison to Haxl and Muse for:

1. Modularised graphs (functions/algorithms)
2. Graphs with map operations (`smap`)
3. Graphs with conditionals (`if`)

This requires extensions to our random code generator to allow generation of correct code with

1. Randomly generated functions
2. Map operations using randomly generated functions
3. Conditionals, with and without forcibly prefetched branches

The end. Thank you for listening.

Slides are available at http://static.justus.science/presentations/extending-yauhau.pdf

Haxl Implementation details

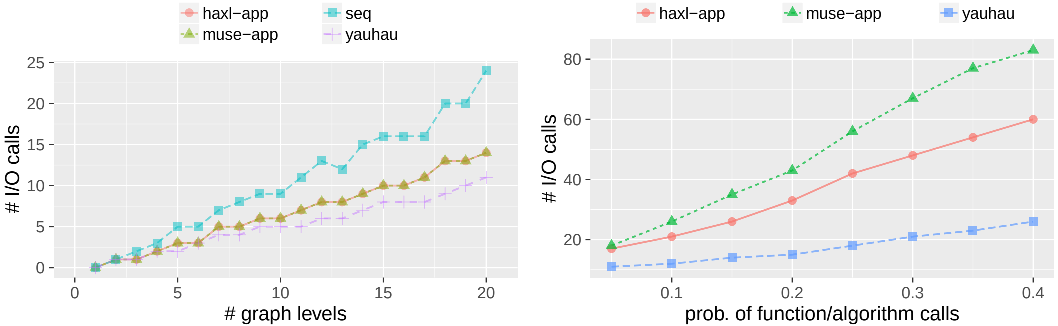
- Applicative functors denote independent data fetches
- Monad bind retrieved data
- Requests are GADT's encoding result type

```
commonFriends :: Id -> Id -> GenHaxl [Id]
commonFriends x y =
    intersection <$> friendsOf x
                 <*> friendsOf y

friendsOf :: Id -> GenHaxl [Id]
friendsOf = dataFetch . FriendsOf
```

Muse implementation detail

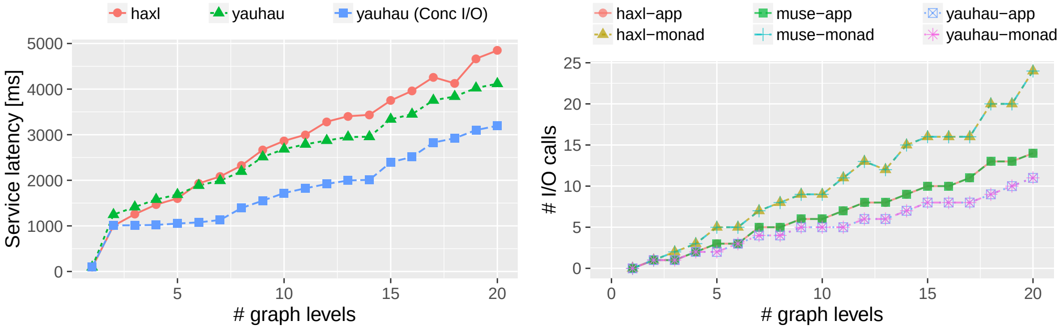
- Similar syntax to Haxl
- fmap/<$> (<$> and <*>), flat-map (>>=)
- Uses free monad to build an AST
- Traverses AST to find fetch rounds

```
(defn common-friends [x y]
  (<$> intersection
    (friends-of x)
    (friends-of y)))

(defn friends-of [x]
  (FriendsOf. x))

(defrecord FriendsOf [id]
  MuseAST
  (...))
```



Left: Baseline comparison. Right: Performance with functions.

Graphs



Left: Mixed latency data sources. Right: Performance across code styles.

Base transformation – Details

- Simulates program execution
- Maintains set of created data (bindings)
- At each step 'call' all non-fetch functions with satisfied inputs
- If no further function can be called dispatch round

```
(let [[x y] (algo-in)
        a (mk-req (FriendsOf. x) data-source)
        b (mk-req (FriendsOf. y) data-source)
        c (fetch a)
        d (fetch b)
        e (intersection a b)
  e)
```