TECHNISCHE UNIVERSITÄT DRESDEN

BACHELOR THESIS

Control Flow and Side Effects support in a Framework for automatic I/O batching

Author:
Justus ADAM

Supervisor:
Sebastian ERTEL

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Chair for Compiler Construction

November 30, 2016

Declaration of Authorship

I, Justus ADAM, declare that this thesis titled, “Control Flow and Side Effects support in a Framework for automatic I/O batching” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

TECHNISCHE UNIVERSITÄT DRESDEN

Abstract

Faculty for Computer Science
Chair for Compiler Construction

Bachelor of Science

Control Flow and Side Effects support in a Framework for automatic I/O batching

by Justus ADAM

The largest source of latency for many modern systems, particularly services, is the inherent latency of I/O operations. Disk I/O, database access and network bound communication is frequent, especially in data processing systems and distributed service infrastructures. Making I/O efficient is cumbersome, involves asynchronous programming and caching and hinders modularity when attempting to perform batch jobs.

There have been a few solutions proposed to automate this, first and foremost Haxl, a Haskell library. We proposed an alternate, improved implementation called Ÿauhau. It leverages dataflow to produce even better performance while being independent from any particular code writing style in a minimal Embedded Domain-Specific Language (EDSL).

In this thesis we first explore the techniques used by Ÿauhau to make I/O more efficient. Secondly we identify, explain and solve issues in Ÿauhau arising from control flow. Furthermore we suggest new optimisations for the compiler and implement safety guards for handling side effects.

Contents

List of Figures

Chapter 1

Introduction

Data processing applications, particularly online services, perform a large number of I/O bound transactions during processing. For instance when reading from disk or a database, querying a remote service or local cache. These I/O bound operations introduce significant latency in the application and developers combat this by running I/O operations asynchronously, adding internal caches and preparing batch jobs. However writing explicit code which uses these techniques is cumbersome at best if it doesn't prevent or destroy modularity and readability of code completely.

As a result engineers at Facebook developed a system called Haxl[13] which automates the process of batching, caching and concurrent execution of I/O data fetches. In the paper "Ÿauhau: Concise Code and Efficient I/O Straight from Dataflow."[5], we propose an improved solution in the form of a plugin for the Ohua[4][3] compiler, which we call Ÿauhau[1]. Our plugin rewrites a program at compile time, inserting caching and batching I/O operations whilst allowing the programmer to write concise and straight forward code. We have shown that Ÿauhau performs better on average than Haxl and imposes less restraints on the style of code a programmer has to write.

At compile time however certain information is not available. If we take the mapping operation `smap`[2] for instance, Figure 1.1. At compile time we cannot (always) know the size of `collection` and therefore not know the size of batch we need to produce to perform the fetches within. This is one of a number of open issues with the Ÿauhau plugin which we will address in this thesis, as well as showing opportunities for improvement.

```
(smap
  (algo [a] (fetch a))
  collection)
```

FIGURE 1.1: Simple smap operation

1.1 Contributions

1. Because of our compile time transformation model, Ÿauhau faces challenges when confronted with control flow structures such as `if` or iteration in the form of a mapping operation called `smap`. The basic implementation of these rewrites was done already for the paper, however no detailed description has yet been provided due to constraints for content length. Therefore this thesis provides a detailed description, including implementation considerations and decisions, of the **if and smap transformations**.

2. To handle nested `smap`s and `if`s in our rewrites we present a generalisation of non-structurally emergent dataflow graph properties into a concept called **Context**. Contexts are a broader concept which transcends the scope of only

[1]The source code is found on Bitbucket[1].
[2]Similar to `map` in other languages.

the control flow structures `if` and `smap` and is now a part of the Ohua framework. In this thesis we will define contexts, describe general properties, how it is detected and how it is used to handle nested control flow in Ÿauhau.

3. The aforementioned paper also introduces the notion of a mutative I/O action in the form of `write` requests. We explained in the paper how we ensure results of mutative actions are visible to the `read` requests in the presence of a cache, however there is still more semantic consistency to be desired between `read` and `write` requests. Namely that when algorithm[3] a depends on the result of running algorithm b, all reads and writes in b should be performed before reads or writes in a are performed. I provide an optional graph transformation in Ÿauhau used to **preserve write semantics** in programs using the Ÿauhau batching.

4. In the Ÿauhau paper we use a random code generator to generate test programs with which we can test the performance of Ÿauhau. To enable **support for mapping operations** as well as different **generation methods for conditionals** we **extend the code generator**.

5. Using the new extensions we can generate programs with certain properties, namely a certain percentage of map applications and/or conditional nodes. As a result I can now show new **experiments** comparing the performance of the Ÿauhau plugin to the existing technology Haxl in **programs with conditionals and mapping**.

6. Lastly we found that precomputing branches of conditionals can lead to higher performance in programs using the Ÿauhau plugin. In this thesis we further explore the possibility of **optimising programs for faster I/O by precomputing branches** by experimentally comparing the number of performed rounds and fetches in programs with both precomputed and non-precomputed conditionals.

[3]An “algorithm” is like a function in the Ohua framework, see Chapter 3.

Chapter 2

Related Work

2.1 Similar systems and inspiration

2.1.1 Haxl

Haxl[12][13] is a Haskell[2] library which provides a framework for performing data access operations with automatic batching and concurrency. It was first introduced in the paper "There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access"[13] by Simon Marlow, Louis Brandy, Jonathan Coens and Jon Purdy.

They describe a system where, by using the Haskell abstraction of an Applicative Functor[14], they are able to find which data access operations are independent from one another and therefore may be fetched as a batch job.

As a small example in Figure 2.1 we see a function `commonFriends`. The `intersection` function depends for its input on both the result of fetching the friends of `a` as well as the friends of `b`. However the two data fetches `friendsOf a` and `friendsOf b` are independent from one another, neither depends on the result of the other, which is what the `<*>` operator symbolises.

```
commonFriends a b =
    pure intersection
        <*> friendsOf a
        <*> friendsOf b
```

FIGURE 2.1: An operation in Haxl

A program written with Haxl runs all independent program graphs blocking once it encounters a fetching operation. When all program paths are either finished or blocking all the blocking requests are executed in what is called a 'round'. After the round finishes executing and the results are present the paths are resumed. A very simple example can be seen in Figure 2.3. Additionally all requests are cached for a run of the Haxl program.

In the paper "Ÿauhau: Concise Code and Efficient I/O Straight from Dataflow." [5] we criticise how the benefits of the Haxl framework require use of special functor operations, such as lifting the `intersection` operation into the Functor with `pure` and applying the arguments with `<*>`. It is much more difficult writing this kind of code than the obvious and straight forward monadic code from Figure 2.2. For small examples adopting this alternative style is not as difficult, but as programs gets larger and more complicated the complexity of maintaining this particular style of writing code increases.

```
commonFriends a b = do
    fa <- friendsOf a
    fb <- friendsOf b
    return (fa `intersection` fb)
```

FIGURE 2.2: Monadic version of `commonFriends`

```
main =
    pure compute
        <*> fetch (increment 23)
        <*> fetch (42 + 1337)
        <*> fetch (log_e 2.71828 + 1337)
```

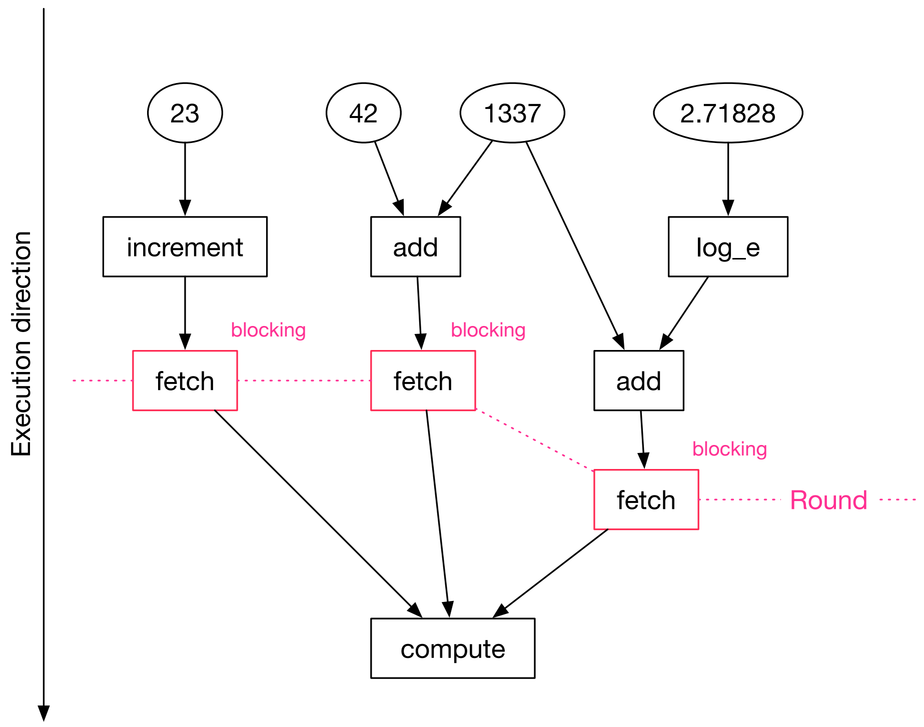


FIGURE 2.3: A visual example of the haxl execution

```
data FacebookRequest a where
    GetFriends :: UserID -> FacebookRequest [UserID]
    GetProfile :: UserID -> FacebookRequest Profile
```

FIGURE 2.4: Defining requests in Haskell

2.1.2 Muse

Muse[10] is a Clojure implementation of Haxl. Like Haxl the data access actions are defined with functors. This functor is a free monad[18] which builds an AST of the program. A runtime traverses that AST, finds parallel rounds of fetches and executes the code, batching the fetch operations.

2.1.3 Datasources

Datasources can be arbitrary user defined targets for I/O actions such as databases, network requests or filesystem I/O. However datasources must be defined separately and only for those defined datasources the optimisation can be leveraged.

Defining a datasource in Haxl means defining a set of actions on the source in form of a GADT[16], see Figure 2.4 .

Every time you want to make a request in the program it has to be encoded into one of those actions. A Haxl function by the name of `dataFetch` can be called with an encoded request to schedule its execution.

The actual execution of the I/O action is defined in a typeclass[19] called `DataSource` which has to be defined on the request GADT, see Figure 2.5. The `DataSource` typeclass has a method called `fetch` which does the actual work of performing

```
class DataSource u request where
    fetch :: ... -> [BlockedFetch request] -> PerformFetch
```

FIGURE 2.5: Defining a data source in Haxl

```
mappedFetches = mapM fetch [a, b, ..]
-- -> [Request a, Request b, ..]

conditionalFetches execute =
    if execute
        then fetch 0 -- -> [Request 0]
        else return () -- -> []
```

FIGURE 2.6: Control flow in Haxl

the desired I/O action and is user defined. Even though it allows arbitrary side effects due to the fact that Haxl caches results it is strongly recommended not to mutate the data source but only retrieve data. As its type signature `fetch` method indicates, see Figure 2.5, it performs multiple requests simultaneously and this is where the batching happens. The user defines how exactly these requests are batched, Haxl only concerns itself with finding requests that can be batched.

Additionally, for the cache, the `Hashable` typeclass needs to be implemented on this type as well as the `DataSourceName` and `StateKey` typeclass to identify the source.

When the framework executes the actions each datasources gets all the requests for that particular datasource. Therefore all actions which can be batched together should be defined on the same datasource (in the same GADT) and actions which cannot be batched should be defined on separate datasources.

2.2 Control flow

Both Haxl and Muse use runtime data structures to aggregate fetch rounds. In Haxl this is a `Data.Sequence.Seq`, which is a type of list. At runtime, if a data fetch is encountered it simply gets appended to this list of fetches to execute. This way Haxl builds a dynamic list of fetches to execute for each round each time the program is run.

This makes it easy to handle cases where the amount of fetches is not known at compile time. For instance in conditionals or when executing a fetch over a collection, see Figure 2.6.

2.3 Side effects

2.3.1 Haxl

Both Haxl and Muse make very strict assumptions about the immutability of their programs. In the case of Haxl side effects of any kind are prohibited. A Haxl computation cannot access the Network or Hard drive, only pure actions, or Haxl data fetches are allowed. This is enforced via the type system. Furthermore Haxl strongly recommends that data fetches not perform mutative operations on the data source. This is mainly because additionally to batching requests requests are also cached and when executed again are simply served from the cache. A sequence of actions like Figure 2.7 would yield `False` in every case, since the second fetch of `a` would be served from the cache.

```
data Request = Retrieve Int | Mutate Int

invalid :: GenHaxl u Bool
invalid = do
    let a = ...
    data1 <- fetch (Retrieve a)
    fetch (Mutate a)
    data2 <- fetch (Retrieve a)
    return (data1 /= data2)
```

FIGURE 2.7: Haxl invalid actions

2.3.2 Muse

Muse is more loose with its allowance of side effects, because in Clojure there is no type system which could be used to restrict I/O. However it also caches previous requests and therefore the effects of mutating the data source are not reflected in future, cached fetches. As a result muse also strongly recommends not to mutate the data source.

Chapter 3

Ohua

Ohua[4][3] is parallelisation framework implemented in Java[8] and Clojure[9].

At compile time Ohua transforms the program into a dataflow graph, see Figure 3.1. The program itself is written in a Clojure EDSL, input in the lower part of Figure 3.1 and composes of so called "stateful functions"3.1, input in the upper part of Figure 3.1.

The created dataflow graph is a representation of atomic parts of the program, the stateful functions, and the data dependencies between them. Since Ohua currently does not support recursion this graph is a DAG. Nodes are call sites[1] of atomic pieces of code, these pieces of code are called stateful functions. Edges represent data dependencies between the call sites. Edge direction indicates result and argument. The result of the source node is an argument to the target node. The result value of one node may be used by multiple other nodes and each node can have multiple inputs or none at all. An example of how this transformation looks can be seen in Figure 3.2. All of the called functions here, such as `vector`, `add` etc. have to be implemented as stateful functions, see Figure 3.3.

This dataflow graph can then be executed by a dataflow execution runtime which dynamically schedules the nodes of the graph. The runtime knows about the data dependencies between the nodes of the graph and therefore can schedule independent nodes in parallel. Data independent nodes can be executed in parallel. Also subsequent calls to the same node with unrelated data, as a result of a mapping operation for instance, can be executed in parallel. This property allows us to do deterministic state modifications on an object of the class the stateful function is attached to. As a result we obtain pipeline parallelism.

[1]A 'call site' of a function is the position in the code where control is handed to the function and it starts executing.

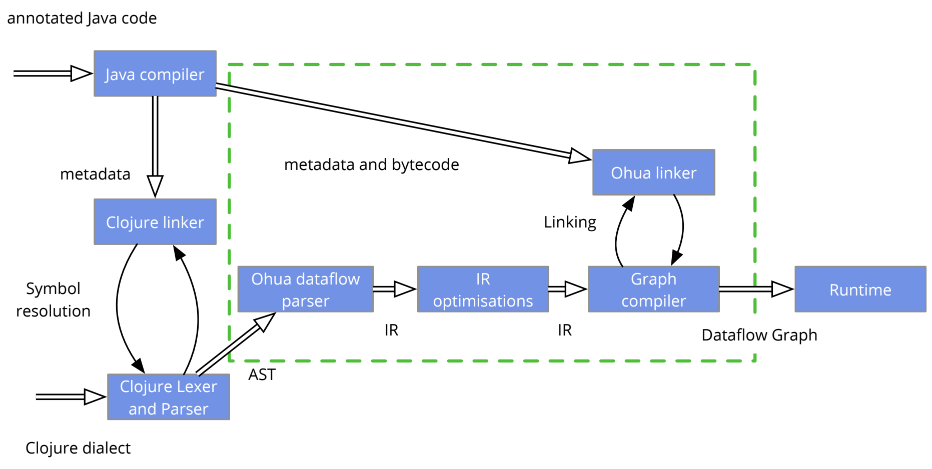


FIGURE 3.1: Compiler flow of Ohua

```
(def main
  (ohua
    (let [a (increment 1)
          b (add 2 a)
          c (some-action)]
      (vector a b c))))
```

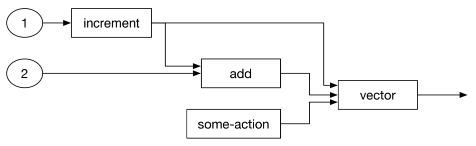


FIGURE 3.2: An example for code transformation in Ohua

```
class Add {
  @defsfn // <- the annotation to make a stateful function
  int add(int i1, int i2) {
    return i1 + i2;
  }
}
```

FIGURE 3.3: Example implementation for `add`

3.1 Stateful Functions

Stateful functions are the implementation for the nodes of the Ohua dataflow graph. They represent atomic pieces of code which the Ohua runtime schedules. Stateful functions can be implemented in Java or Scala[15] by annotating a method with `@defsfn`, or in Clojure by using the `defsfn` macro. Each stateful function has an associated class, which holds an internal, opaque state of the node. Whenever a stateful function is referenced in code the runtime creates a new instance of the class in which the stateful function is implemented. As a result invocations at the same call site of a stateful function implicitly share an opaque state, but different call sites do not share state implicitly. To share state across call sites the state has to be explicitly passed to the function as an argument. This ensures that state sharing is transparent to the runtime without requiring in depth knowledge about the structure of the state itself or the stateful function.

When executing the dataflow graph the runtime dynamically schedules nodes on a configurable number of cpu cores. As a result the precise order in which the graph is executed is indeterministic. However the runtime ensures that for each node the order of its subsequent invocations is preserved which ensures correct semantics not only in the high level algorithm but also with regard to the internal, opaque state of each stateful function itself.

3.2 Algorithms

Algorithms express high level, parallelisable computations in Ohua. These algorithms are written in Ohua's Clojure EDSL using the `algo` macro. Algorithms use and combine stateful functions to express complex computations. Algorithms are generally assumed to be pure in so far as there are no data dependencies between any two functions which are not directly visible via function arguments and return values. No two functions should share data in any way other then by explicitly passing them from one to the other via arguments and result. Furthermore there should not be any dependency on invocation ordering which is not expressed directly through data dependencies. The snippet of code in Figure 3.4 for example: If this were normal Clojure code, the read would be executed before the write due to the sequential, deterministic semantics of a Clojure `let` binding. In Ohua however, since there are no data dependencies between `read` and `write`, the execution order of those two stateful functions is nondeterministic and they could even be executed simultaneously. The graph in Figure 3.4 shows nicely how at this stage there is no visible order for those two actions in the

```
(defalgo my-algo []
  (let [a (read-database)
        b (write-database)]
    (compute a b)))
```

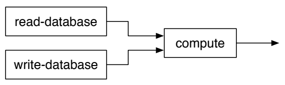


FIGURE 3.4: An example of indeterministic code

compiled graph. Both `read-database > write-database > compute`[2] and `write-database > read-database > compute` are valid topological orderings of the dataflow graph and Ohua only guarantees *a* topological call order, not which. When the algorithm is executed using the `ohua` or `<-ohua` macro the algorithms Clojure code is compiled into a dataflow graph and executed by the Ohua runtime.

3.3 Dataflow IR

Ohua was already using a data flow graph for its runtime execution model, but in order to implement Yauhau we also needed a (much simpler) compile time data flow graph representation. A simplified dataflow representation also lends itself nicely for performing high level optimisations on the graph itself as well as perform analysis such as type checking[3]. As a result the dataflow IR on which Ÿauhau operates is now directly part of Ohua and its compilation pipeline.

Like the final dataflow graph the IR encodes the dataflow relationships inside of the program. The description of the graph contains two elements:

1. **Bindings** are named data elements or alternatively can be interpreted as a data path. Each binding has to be unique and constant, which means it can only be assigned once, in which case we also speak of the *creation* of the binding or of *writing* the binding. Also for any operation on the IR itself it is assumed that the value of the binding itself is not mutated as the program runs. However the binding may be read any number of times (including 0).

2. A **Function** either represents the call site of a stateful function or a dataflow operator. A function name references the stateful function or operator implementation which should be executed. A list of input, or parameter, bindings references the data which is expected as argument to the stateful function and a list of result bindings creates references to the data produced by the stateful function. The same stateful function can be referenced by multiple IR functions, since IR functions represent call sites of a stateful function, not the stateful function itself. If a binding is present in a parameter list we also speak of the binding being *read*. Is the binding present in the result bindings list we also speak of the binding being *written* or *created*[4].

Any IR graph can be easily serialised into a Clojure `let` form preserving the semantics of the graph. The serialisation of the dataflow IR into a human readable form makes it easier to reason about semantics of the program, which in turn makes it easier for a programmer to debug the program.

3.3.1 Implementation

The actual implementation of the graph is simply a `Vector` of functions, see Figure 3.5. Each function is a Clojure record with fields for a unique id, the name of the function, a vector of input bindings and a binding or a vector of bindings as result values. By moving result lists to the left and a putting a Clojure function call to the right this graph can be represented as a Clojure `let` form, see Figure 3.6.

[2]`a > b` means `a` is executed before `b`
[3]This may be subject of future work
[4]Due to the uniqueness constraint *writing* and *creating* are equivalent.

```
(defrecord IRFunc [id name args return])

(def graph
        [(->IRFunc 1 "f" ['a 'b] 'c)
         (->IRFunc 2 "g" ['c] ['d 'e])])
```

FIGURE 3.5: Concrete IR snippet

```
(let [c (f a b)
      [d e] (g c)]
  [d e])
```

FIGURE 3.6: IR representation as Clojure program

Chapter 4

Ÿauhau

Ÿauhau is a plugin for the Ohua compiler. It was first introduced in the paper "Ÿauhau: Concise Code and Efficient I/O Straight from Dataflow." [5], where a more detailed account of the motivation, implementation and results may be found. Following I will briefly outline the basic transformation in Ÿauhau which is necessary to understand the arising problems which I am solving in the Chapters following.

Ÿauhau hooks into the compiler after the Clojure source has been parsed and transformed into the dataflow IR. It is a pure IR to IR transformation. In the flow graph in Figure 3.1 Ÿauhau would be run as part of the IR optimisation step. Additionally a map of context information is given to the plugin. The precise nature of this context information will be explained in section 7 its information is used in the transformations of Chapter 6 and Chapter 5. It is irrelevant to the basic Ÿauhau idea and transformation.

Fundamentally Ÿauhau performs a graph transformation on the dataflow graph it has been given in form of the IR. The goal behind this transformation is to find sets of I/O operations which are data independent from each other. This is functionally similar to the approach taken by Haxl[12], see Chapter 2. Haxl leverages the Monad and Applicative abstractions provided by the Haskell language and base library to propagate a list of parallel executable I/O actions through the program path, blocking each path as soon as an I/O action is encountered. This is a dynamic way of finding parallel I/O actions, i.e. at runtime a data structure (Data.Sequence.Seq) is populated with pending requests. Ÿauhau takes a different approach and statically finds parallel I/O actions at compile time.

You can see the full Ÿauhau transformation flow in Figure 4.1. We will discuss the individual parts of this flow graph in the upcoming sections.

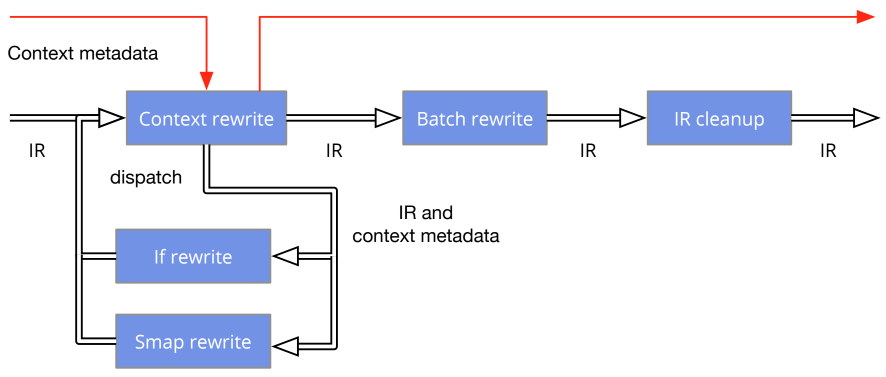


FIGURE 4.1: Rewrite order and flow in Ÿauhau

4.1 Runtime execution

In Ÿauhau an accumulative fetch node, called __accum-fetch statically replaces a set of fetches. This is equivalent to a **round** in Haxl. The accumulator is statically wired into the call sites of the fetches it replaces. It has one input and one output for each fetch node it replaces. Once a request for each input is present at runtime the accumulator executes, retrieving all of the requests simultaneously.

The parallel fetches are grouped into sequences of fetches on the same data source. Each sequence is handed to its respective data source which has to perform the fetch. In Ÿauhau this process is inherently parallel and each data source performs its request simultaneously. In Haxl the programmer can choose whether to perform an asynchronous or synchronous request. When all data sources are finished the results are distributed back through the corresponding outputs of the accumulator. A visual representation of this can be seen in Figure 4.7.

A new experimental and optional feature in Ÿauhau is to do dynamic dispatching of the finished requests. This means once a certain data source returns with the data, computations only depending on that data source, can continue immediately.

4.2 Data sources

Similar to Haxl, compare Section 2.1.3, all data sources a user wants to access in Ÿauhau must be defined externally.

Ÿauhau has a more loose way of working with data sources than Haxl. I/O actions in Ÿauhau are instances of the Request class.

```
class Request<Payload, Return>{
  Paylod payload;
  IDataSource<Payload, Return> dataSource;
  Request(Payload payload,
          IDataSource<Payload, Return> dataSource)
    { /* implementation omitted */ }
  /* omitted code */
}
```

FIGURE 4.2: Implementation of the Request class

Each Request has two fields, an arbitrary Payload and a data source, compare Figure 4.2 . Here the data source, similarly to Haxl implements a Java interface IDataSource, see Figure 4.3, which provides a method for executing batched requests, hence the type Iterable<Payload> for the fetch method on the IDataSource interface (Figure 4.3).

```
interface IDataSource<Payload, Return> {
  Object getIdentifier();
  Iterable<Return> fetch(Iterable<Payload> requests);
}
```

FIGURE 4.3: Implementation of the IDataSource interface

At runtime requests are grouped by the data source they are paired with and then handed to the respective source as a single sequence of many requests.

4.3 Compile time transformation

The basic Ÿauhau transformation, as mentioned, operates entirely on a dataflow graph. The transformation does not concern itself with the kinds of nodes in the graph except for one, the `fetch` node/function which is a special node provided by the Ÿauhau library. Beyond that, to simplify the transformation, it operates exclusively on the structure of the graph. The `fetch` node itself is implemented as a regular stateful function (See Figure 4.4) and works without the Ÿauhau transformation, although using it without the batching optimisation makes it much less useful. Since data sources expect sequences of requests the `fetch` operator builds one and then returns the first element from the sequence of results returned by the data source. If the source works as expected this result sequence should hold results of performing the requests in the same order as the requests were originally.

```
class FetchOp<P, R> {
  @defsfn
  R fetch(Request<P, R> request) {
    List<P> l = Collections.singletonList(
                  request.getPayload()
                );
    Iterable<R> res = request.getDataSource().fetch(l);
    return res.iterator().next();
  }
}
```

FIGURE 4.4: implementation of the `fetch` stateful function

4.3.1 Round detection

The goal of the round detection is to find a minimal set of rounds. A 'round' is a set of `fetch` nodes with no data dependencies between the nodes. Rounds must not overlap. Hence the round detection finds a minimal set of non overlapping sets of data independent `fetch` nodes.

Algorithm description

To find the rounds in a given program I simulate the execution of the program and block code paths when a `fetch` node is encountered. The simulation is done by topologically visiting the nodes of the graph while tracking three sets of items.

created The set of created bindings tracks which pieces of data that flow between the nodes of the program have been created prior to a particular point in the program.

visited The set of visited nodes tracks which nodes were part of a previous stage.

round The current round consists of all pending `fetch` nodes. When a new fetch round is finalised it will consist of these `fetch` nodes and the current round will be newly initialised empty.

Since bindings are unique in the dataflow IR we can track produced data by simply adding the respective binding to the set of created bindings. The set of created bindings is initialised with the input parameters of our program. From this point we traverse the graph in stages. Each stage consists of the set of graph nodes where all inflowing data has previously been created and which was not in a previous stage.

For the former condition we check, for each input binding, whether the respective binding is contained in the set of *created* bindings. To ensure the latter condition a separate set of *visited* nodes is maintained and only nodes which are not members of this set are allowed in a new stage. Once the stage has been computed all functions in the stage are added to the *visited* set. Then all `fetch` nodes are filtered from the stage and added to the current *round*. All remaining nodes are

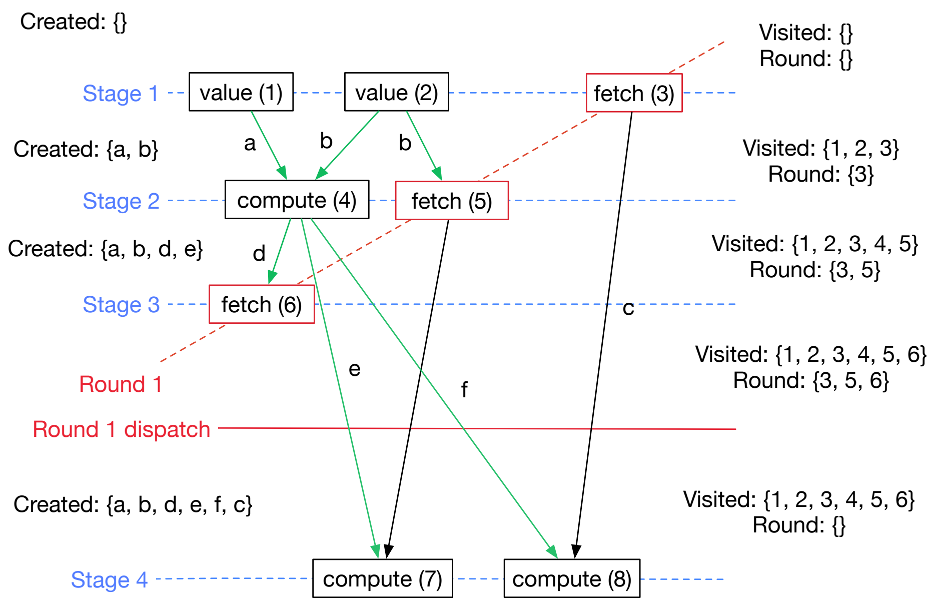


FIGURE 4.5: Example of the round detection algorithm

being executed. Simulating to executing the nodes is done by simply adding all produced bindings (return bindings) of the node to the *created* set. If there are no remaining nodes to execute in a stage we have reached a point where the remainder of the dataflow graph depends on I/O actions, including the `fetch` nodes. Therefore the current round is maximal and we simulate executing all `fetch` nodes of the current round and dispatch a finalised fetch round. Afterwards the current *round* set is initialised empty again. This is done until we get an empty stage, at which point we dispatch the current round again, unless it is empty, and we have reached the end of the round detection.

As an example you can see a typical dataflow IR graph in Figure 4.5. The number after the node name is the unique id for that particular node and used for in the *Visited* and *Round* sets. Blue dotted lines mark the stages as we traverse the graph. In each stage only nodes for whom all incoming edges are created, symbolised, for up to the first round, by the green arrows, can be part of the stage. You can also follow this algorithm along in Figure 4.6. After each stage the outgoing nodes are added to the created set as seen on the lefthand side. The red `fetch` nodes block the propagation through the graph and they are added to the *Round* set on the righthand side. After stage three we have no more nodes for whom all incoming edges are green, and therefore we dispatch a round of fetches, indicated by the dotted red line. When the round has been dispatched, indicated by the solid red line, its created bindings are added to the *Created*, visible after the dispatch on the lefthand side.

Result guarantees

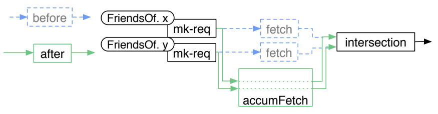
Adding the `fetch` nodes to the set of visited nodes guarantees that rounds do not overlap. Not simulating fetch execution before the round finishes guarantees that no round contains two fetches with data dependencies between them, since the dependent fetch would never have been in a stage before the round was finished since the required binding from the first fetch would not be created before the round finishes. Finally since we only dispatch a round once no more executable nodes are found all other nodes have to be fully data dependent and this guarantees the set of rounds to be minimal.

Remarks:

- `set/union` is the set-union ($\cup$), `set/difference` is the set-difference ($\setminus$) and `#{}` is an empty set.
- `filter` filters the collection where the predicate is true, `remove` where the predicate is false.
- `loop` and `recur` is tail recursion in clojure. `recur` restarts the body of `loop` with variables bound to the arguments of `recur`.
- `ir` is the ir graph give to the algorithm

```
(loop [visited #{}
       created #{}
       round #{}
       rounds []]
  (let [stage (set/difference
                 (get-callable-fns created ir)
                 visited)
        new-visited (set/union visited stage)]
    (if (empty? stage)
      ; end algorithm
      (if (empty? round)
        rounds
        (conj rounds round))
      ; continue
      (let [fetches (filter is-fetch? stage)
            non-fetches (remove is-fetch? stage)
            new-round (set/union round fetches)]
        (if (empty? non-fetches)
          ; dispatch round
          (recur new-visited
                 (set/union
                   created
                   (all-return-bindings new-round))
                 ; next round set to empty again
                 #{}
                 ; current round appended to rounds
                 (conj rounds new-round))
         ; continue to next stage
         (recur new-visited
                ; non-fetch bindings are now created
                (set/union
                  created
                  (all-return-bindings non-fetches))
                new-round
                rounds))))))
```

FIGURE 4.6: Round detection algorithm in Clojure



h

FIGURE 4.7: Base transformation (taken from our paper[5])

```
(ohua
  (let [free-var (add 4 5)
        list [1 2 3 4]]
    (smap
      (algo [item]
        (add item free-var))
      list)))
```

FIGURE 4.8: Free variables in smap

4.3.2 Graph rewrite

Once the rounds have been computed an accumulated fetch is inserted for every round. For each round all fetch nodes in the round are removed from the graph. An accumulator is inserted with the combined inputs of all fetch nodes and the combined outputs of those nodes in the same order. See Figure 4.7.

4.3.3 Control flow considerations

At runtime a node in the dataflow graph waits for inputs to arrive in channels. There is one channel per function argument. When the runtime scheduler activates the operator one input packet is removed from each channel and the stateful function is called with these arguments. This process repeats until the scheduler decides to deactivate the node or if one of the input channels is empty.

These semantics are the same as they would be in a typical program, where every argument to a function has to be present for its call. In dataflow this is not as straight forward as in typical code. For instance free variables in algorithms inside an smap have to be specially handled during execution. For clarification, smap is a mapping operation over a list. It takes two arguments, an algorithm and a list of items and for each item in the list the algorithm is executed once with the item as argument collecting the results into a new list. This is analogous to the `map` function in Haskell or Clojure, with the difference that there are some guarantees about execution order. For a usage reference see Figure 4.8.

Normally variable bindings are simply translated into a set of arcs from where the value in the binding was created to each site where the variable is used. When the source emits a value it is sent down each arc once. This poses a problem in the case of smap where the target of the arc is a free variable inside the smap, like in Figure 4.8, and other inputs may receive many packets due to the mapping, whereas this input would naively only receive one. What Ohua does to solve this is replicate sending the packet once for each element in the mapped collection.

In Yauhau we run into a similar problem. We do not have free variables but perform an operation which is the complete opposite, where we rewrite connections to lead to an accumulator which is positioned outside the current control flow context (`smap`). What we do not consider during our graph transformation is how often any of the fetch operators we are merging are called in the program. If we have fetches in different contexts and naively accumulate them we create a similar situation to the one described above, where the inputs to the accumulator receive different amounts of data. An example can be seen in Figure 4.9.

In the example provided we can see two issues. When a collection, which is longer than 1 is given to smap the respective input to the accumulator would be

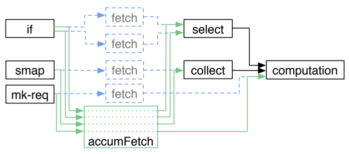


FIGURE 4.9: Naive batching

provided with a piece of data for each item in the collection whereas the bottom-most input from `mk-req` would at the same time only receive a single piece of data. At the same time the topmost two inputs comes from inside an if which means that only **one** of those two accumulator inputs would receive data during any execution of this program snippet, whereas the other receives nothing.

The structures which introduce issues like this are control flow structures, because they change how often or whether a particular section of code is executed. Unfortunately the flow of control in a program is not structurally visible in our dataflow graph. Therefore our accumulation transformation is unable to deal with this problem.

Instead of hammering support into this transformation I decided to add additional preparation steps before the batching transformation to produce an intermediate graph with certain properties such that the batching transformation is possible. The properties of the graph we want are that any fetch in the same round is control flow wise in the same context, i.e. gets called the same amount of times. Ensuring this particular property is easier if we simple pull every fetch out of any control flow context it is contained in so that it ends up being called exactly once in a given execution of the program. Therefore our desired property of every fetch being executed the same amount of times holds and we can proceed with the transformation.

In the following Chapters I will explain how we handle `smap` (Chapter 5) and `if` (Chapter 6) and how we generalise the two to define a pluggable, general transformation which ensures we handle nesting correctly (Chapter 7).

Chapter 5

Smap Transformation

5.1 Simplified

Aim of the smap transformation step is to merge a bounded and unknown number of `fetch`es into a single `fetch`. How many times the `fetch` would be executed is not known at compile time but only given at runtime by the length of the collection which is being mapped over. Basis for the transformation itself is that a mapping operation over two composed functions is semantically identical to the composition of one mapping operation with one function each, in the same order, see Figure 5.1.

The basic transformation steps can be described as follows. The function inside the map is separated into a computation before the fetch, the fetch, and a computation after the fetch, see `decomposed` in Figure 5.3. Since there may be data flowing from `before` to `after` which does not pass through the `fetch` we choose `before` and `after` such that `before` returns a tuple where the first part of the tuple contains only the data our fetch can operate on and the second part of the tuple contains any additional data which needs to flow between `before` and `after`. `after` then accepts a tuple where the first part is the result of executing the fetch and the second part is the untouched additional data flowing between `before` and `after`. In the middle we apply the fetch function to only the first part of the tuple with the higher order function `first`.

Subsequently we can transform the single map into multiple according to the rule in Figure 5.1. Then we restructure the data pulling the tuple out of the sequence using `unzip` and repackage later with `zip`. Lastly we can replace `map fetch` with `fetchMany`.

In Figure 5.3 you can see this rewrite described in a first attempt at denotational semantics in Haskell as a typed lambda calculus, which should indicate correctness in so far as each stage would typecheck and the top level type signature of each stage does not change. It is no proof of semantic correctness but provided as an indicator. In the future we hope to formalise these rewrite rules based on lambda calculus and prove correct semantics as well, see Chapter 12.

5.2 Implementation detail

On the dataflow IR we don't work with expressions as above but with graph nodes, which could also be interpreted as operators. The splitting is therefore not done with maps, but special nodes which start and end a mapping operation in Ohua. The Ohua mapping operation is called `smap` and consists at IR level of several operators, the two interesting ones being `smap` (internally called `smap-fun`) and `collect`. `smap` is the starting operator and provides the mapping functionality by continuously emitting items from a collection until the collection is empty.

```
map (f . g) == map f . map g
```

FIGURE 5.1: Equivalence of map composition/decomposition

- `::` and `->` denote a type signatures. Types between arrows are input types and the last type is the return type. For instance `f :: Int -> Int -> Bool` equates to the C type signature `bool f(int ..., int ...)` and `val :: Int` would equate to `int val`. These signatures can be placed either above function definitions, but also inline after a value, to fix its type.
- Lowercase words in type signatures are type variables. Can be thought of as being like the type variables in generics in java or templates in C++ but bound for only one type signature.
- `forall <vars> .` binds the type variables `<vars>` for the entire function. Therefore any reference to a name from `<vars>` in a type signature in the function body now references the same type.
- `where` blocks define local values and functions.
- `(a, b)` denotes a 2-tuple with fields of type `a` and type `b`

FIGURE 5.2: Remarks for denotational code

`collect` is the opposite and only receives items until the collection is rebuilt. Both also obtain information such as the size of the collection in order to function correctly.

The `smap` transformation as implemented in Ÿauhau simply inserts a collect operator before the fetch to rebuild the collection and a tree builder node to wrap it. The collect operator is also fed the size of the original collection. A new `smap` operator is inserted after the fetch to break the list into items again and continue the mapping. A graphical example of this can be seen in Figure 5.

5.3 Encoding

The structure of the nested smap is saved in a tree structure. We use a rose tree[11] which is only labeled on the leaves. This particular tree structure is flexible both in height and width but restricted to one type of contained item. We can therefore support arbitrary deep nestings of `smaps` (tree height) and arbitrarily large collections to `smap` over (Number of children $\rightarrow$ tree width) so long as all our items are requests only. Our rose tree only holds one type of data and therefore can easily be flattened into a list of data items and later be reconstructed into a tree. The Java programmer will recognise the Composite Pattern[6] in the definition of our tree, which comprises an abstract tree base class, Figure 5.5 and two concrete implementations with a branch (Figure 5.6) and a leaf (Figure 5.7).

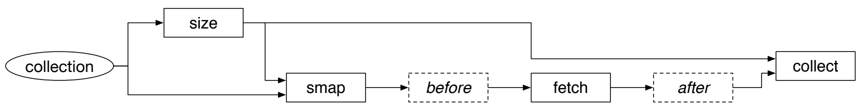
The basic tree offers methods to access all contained requests as a flat stream as well as a method for constructing a nested tree structure of results mirroring the structure of the request tree. Our tree is composed of unlabeled branches which hold a sequence of subtrees as seen in Figure 5.6. In this case the flat request stream simply comprises the concatenation of all requests from the subtrees. We don't need any further knowledge of the structure of those subtrees to obtain the requests here. Although this structure is technically capable of handling a heterogeneous list of subtrees, in practice, as a result of how these trees are created sibling branches have the same height and width.

Lastly our leaf nodes are simply requests, as the basic `Request` also extends the `RequestTree`, Figure 5.7.

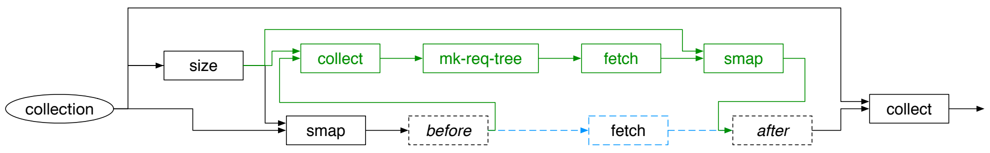
```haskell
fetchMany :: [FetchData] -> [FetchResult]
fetchMany = -- omitted

original :: forall a b. [a] -> [b]
original = map f
    where f :: a -> b
          f = -- omitted

-- chosen such that: f = after . first fetch . before
decomposed :: forall a b free. [a] -> [b]
decomposed = map (after . first fetch . before)
    where before :: a -> (FetchData, free)
          before = -- omitted

          fetch :: FetchData -> FetchResult
          fetch = -- omitted

          after :: (FetchResult, free) -> b
          after = -- omitted

fission :: forall a b free. [a] -> [b]
fission =
    (map after :: [(FetchResult, free)] -> [b])
    . (map (first fetch) :: [(FetchData, c)] -> [(FetchResult, c)])
    . (map before :: [a] -> [(FetchData, free)])
    where ...

unzipped :: forall a b free. [a] -> [b]
unzipped =
    map after
    . (uncurry zip :: ([c], [d]) -> [(c, d)])
    . (first
        (map fetch :: [FetchData] -> [FetchResult])
        :: ([FetchData], g) -> ([FetchResult], g))
    . (unzip :: [(e, f)] -> ([e], [f]))
    . map before
    where ...

replaced :: forall a b free. [a] -> [b]
replaced =
    map after
    . uncurry zip
    . first fetchMany
    . unzip
    . map before
```

See remarks in Figure 5.2

FIGURE 5.3: Denotational decomposition



(A) Smap original



(B) Smap after rewrite

FIGURE 5.4: Smap transformation

```java
abstract class RequestTree {
  abstract Stream<Request>
  getRequestsStream();

  abstract Iterable<Object>
  buildResult(Map<Request, Object> responses);
}
```

FIGURE 5.5: Abstract base class

```java
class RequestTreeBranch
      extends RequestTree {
  Iterable<RequestTree> subtrees;

  Stream<Request> getRequestsStream() {
    return StreamSupport
              .stream(subtrees.spliterator(), false)
              .flatMap(RequestTree::getRequestsStream);
  }

  Iterable<Object>
  buildResult(Map<Request, Object> responses) {
    return StreamSupport
              .stream(subtrees.spliterator(), false)
              .map(t -> t.buildResult(responses))
              .collect(Collectors.toList());
  }
  /* omitted code */
}
```

FIGURE 5.6: Concrete branch

```
class Request<P, R>
      extends RequestTree {
  Stream<Request> getRequestsStream() {
    return Collections
             .singletonList((Request) this)
             .stream();
  }

  Iterable<Object>
  buildResult(Map<Request, Object> responses) {
    return Collections.singletonList(responses.get(this));
  }
  /* omitted code */
}
```

FIGURE 5.7: Request class

Chapter 6

Conditionals Transformation

6.1 Simplified

The if construct in the Ohua comprises essentially of three parts. A visual representation can be seen in Figure 6.1. First an operator called `ifThenElse`[1] which evaluates the condition. Output of this operator are two so called *context arcs*, which is a dataflow device to activate, or not activate a branch of the graph at runtime, represented in the Figure 6.1 by dotted lines. Second there are two subgraphs, one representing the code to execute if the condition is `true` and one to execute if the condition evaluates to false. And lastly an operator called `select` which combines the output of the two subgraphs. The two *context arcs* from the `ifThenElse` operator are each connected to one of the subgraphs. Depending on whether the condition evaluates to `true` or `false` one of the subgraphs is activated by sending a packet down the respective *context arc*. The `select` operator then propagates the result from the activated subgraph to the rest of the program.

6.1.1 Splitting branches

In code

We can apply a similar method of decomposition as we did with smap. To keep the transformation simple we assume both branches to have an equal number of fetches such that always two fetches, one from each branch, can be merged. If we merge one fetch from each branch we know that no matter the condition, exactly one of them is going to have input present. Hence if we select whichever input is present and feed it to our single merged fetch it is guaranteed to have input no matter the condition. Afterwards we will ensure the output from `fetch` is fed back into the continuation of the branch the input came from.

In Figure 6.2 we can see an exemplary version of this rewrite in Haskell, again this is provided as a visual aid and an indicator for correctness as a result of the types lining up. Again the idea is to decompose the original function into a part before and after the `fetch`, as well as the `fetch` itself. Other data flowing between the before and after functions is packaged into a tuple with `FetchData` and `FetchResult` and the fetch is only applied to the data it concerns.

Intermediary we unify this free flowing data into an `Either` structure which is a sum data type, capable of holding one of of two arbitrary pieces of data. This is necessary since the free data flowing between `fBefore` → `fAfter` and `gBefore` → `gAfter` does not have to have the same type. Additionally the choice between the `Left` and `Right` constructor of the `Either` type also encodes our condition.

[1] I often abbreviate this by just calling it `if`

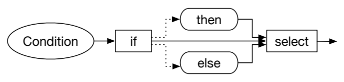


FIGURE 6.1: If represented in operators

Remarks for denotational code:

- `Either a b` encodes a choice of type. It is either inhabited by `a` (`Left` constructor) or the type `b` (`Right` constructor).
- `either f g val` is a function which applies one of two functions `f` or `g` to the value within an `Either` (`val`).

```
original :: Bool -> a -> a
original cond val =
    if cond
        then f :: a -> a
        else g :: a -> a

decomposed :: forall a b c => Bool -> a -> a
decomposed cond val =
    if cond
        then fAfter . first fetch . fBefore
        else gAfter . first fetch . gBefore
  where
    fBefore :: a -> (FetchData, b)
    fBefore = -- omitted

    fAfter :: (FetchResult, b) -> a
    fAfter = -- omitted

    gBefore :: a -> (FetchData, c)
    gBefore = -- omitted

    gAfter :: (FetchResult, c) -> a
    gAfter = -- omitted

data Either a b = Left a | Right b

either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left v)  = f v
either f g (Right v)  = g v

splitIf :: forall a b c => Bool -> a -> a
splitIf cond =
    (either
      fAfter
      gAfter
      :: Either (FetchResult, b) (FetchResult, c) -> a)
    . liftEither
    . (first fetch
      :: (FetchData, Either b c) -> (FetchResult, Either b c))
    ((if cond
        then second Left . fBefore
        else second Right . gBefore)
        :: a -> (FetchData, Either b c))
  where
    liftEither :: (FetchResult, Either b c)
               -> Either (FetchResult, b) (FetchResult, c)
    liftEither (fr, t) = either (Left . (fr,)) (Right . (fr,)) t
```

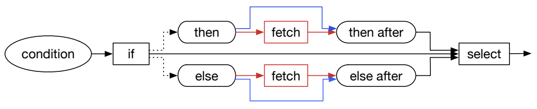
FIGURE 6.2: Denotational conditional rewrite

If the condition was true, we will have a `Left` value and if it was false, we will have a `Right` value.

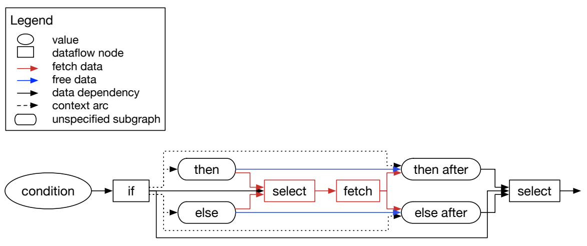
After the fetch was performed, we pull the Fetch result inside the `Either` value. We can do this, since it requires no knowledge about the structure of the type inside the `Either`. Finally we use the `either` function to conditionally apply either `fAfter` or `gAfter` to this `Either`.

As dataflow graph

As an exemplary graph in Figure 6.3a we see two branches, each with a fetch somewhere in them. We cannot be sure which fetch gets executed when this code is run, however we know for certain that exactly one of them gets executed. Therefore we insert a `select` operator into the graph which returns whichever input data was present and fetch that, see the red path in Figure 6.3a and 6.3b. After that we wire the result into all places of the rest of **both** branches wherever one of the fetches was used. We insert context arcs from the `if` operator to those places to make sure they run only if the respective branch was selected. Any other data flow between the sections before and after the `fetch` are left unchanged, see the blue arrows in Figure 6.3a and 6.3b.



(A) Source graph



(B) Graph after transformation

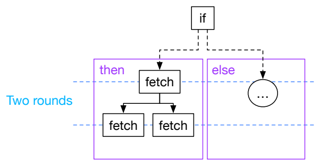
FIGURE 6.3: Visual graph transformation

6.1.2 Fetch imbalance

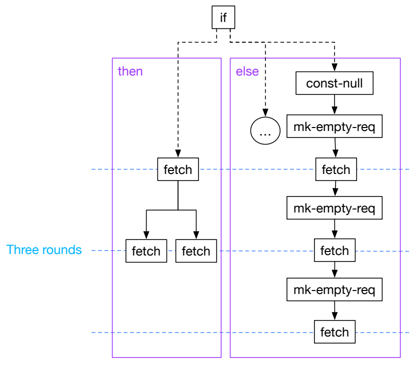
We previously assumed that both branches held an equal number of fetches and that these could be paired up neatly in twos. However in real programs this may not be the case at all, see Figure 6.4a. There is a simple solution to this problem. We solve the imbalance of fetches by inserting empty (NoOp) fetches at the front of the branch with fewer fetches, see Figure 6.4b.

Originally I would create the empty fetches by collecting an graph of fetches and data dependencies between them on each branch. From this I obtained a topologically sorted list of fetches. After that I would calculate the difference in length between both lists to obtain a number of necessary empty fetches. Then I would generate a sequence of request-fetch pairs as long as the calculated difference. These would then be connected in sequence. The first one would receive input from the last fetch of the shorter of the two previously calculated sequences of preexisting fetches. An example of this can be seen in Figure 6.4b.

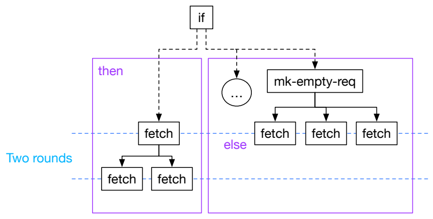
Since these are NoOp they ignore their inputs, we use the inputs only to create data dependencies. If the shorter sequence of preexisting fetches was empty a new operator `const-null` would be created and inserted. This operator would not get input at all, except for a context arc from the `if` and emit a data packet containing `null` used to activating the first created fetch. The second request-fetch pair would receive as input the result from the first fetch and the third one the result from the second and so on. The final result was discarded.



(A) Graph of a graph with imbalanced fetches in conditional



(B) Insertion of empty requests and fetches



(C) Better approach to empty requests and fetches

FIGURE 6.4: Two approaches to inserting empty fetches

There is a significant problem with this approach, which is, and I only realised this late, that this created sequence is fully dependent. Meaning each fetch depends on **all** of its predecessor fetches. We have created a branch which contains a sequence of fully dependent fetches. However there is no reason why the other, longer branch should be fully dependent also. The result is that, once we merge both branches around the fetches the resulting combined fetches inherit both data dependencies. Since you cannot depend on more than all predecessors the resulting merge inherits its dependence from the fully dependent second branch. In the the second branch each fetch was dependent on all previous fetches in the branch, and from this follows that it will have to be in a separate fetch round to all its predecessors. And since this is true for all fetches in the branch each of those will be in a separate round. They can still be batched with other, parallel parts of the program, however we have lost the opportunity for some batching here, because we have forced sequentiality, even if it may not have existed originally.

There is a simple solution to this problem, which, coincidentally, also simplified the implementation of the algorithm as a whole. Now I only generate a single operator called `mk-empty-req` which creates one empty request, with no dependency other than the `if` operator. Note that the `mk-empty-req` implementation changed as well to combine the functionality of `const-null` and the old `mk-empty-req`. This empty request is the only argument to each of the NoOp fetches I create. Therefore we only create one request, which reduces the number of operators, and the fetches depend only on the `mk-empty-req` and the `if` operator, not on one another. Therefore they could in theory all be batched. If we now merge the branches our empty requests will create no new dependencies, since they have the least dependency possible in the branch.

Now we can apply the merge transformation as described before.

Chapter 7

Context generalisation

Structures such as `smap` and `if`, which alter control flow are not compatible with the Ÿauhau base transformation. In the preceding Chapters we have learned how both these structures may be altered individually to extract a particular node from them. However these structures do not occur exclusively independent, rather they may be nested into one another in arbitrary fashion. The transformations as described in the previous chapters are only able to extract selected nodes from one particular, single layer of one of these structures. I.e. the `smap` transformation[1] moves all fetch nodes from one `smap` into the next outer layer.

When designing the unified control flow rewrite I realised however that as long as one possesses a rewrite for each type structure which extracts nodes from exactly one level we can build an algorithm which uses these individual rewrites to extract *all* nodes from *all* contexts in a semantics preserving way.

7.1 Definition

In order to talk about structures such as `smap` and `if`, in particular for the purposes of our unified extraction algorithm we introduce a new concept to Ohua called *context*. *Contexts* are a much broader concept than just `if` and `smap` but this is where the idea first emerged. Contexts now solve an issue emerging from the internal representation of the program. The information in a dataflow graphs is limited to data dependencies and derivable properties. However real programs have properties which do not emerge structurally in a dataflow graph. Control flow structures such as `smap` and `if` for instance change the properties of the contained subgraph but this is not structurally visible in the graph. In order to determine whether a certain node has such properties we need to look and interpret the actual labels of the nodes. In the example in Figure 7.2 these properties are marked with dotted circles, because they are not inherently visible from the structure of the graph itself.

Context is the general name we give these not structurally emergent properties of a subgraph. The context type is a label which refers to the behaviour of that particular context, i.e. `if` or `smap`. An instance of a context is a concrete subgraph of the dataflow graph, as seen in Figure 7.2. In this figure you can also see the typical enter and exit nodes of a context, which are located at the context boundary. The type of enter and exit nodes determines the behaviour of the context itself.

[1]A slightly modified version of the transformation

```
(defalgo algo1 [param]
  (smap
    (algo [val]
      (if (is-iterable? val)
        (smap (algo [item] (compute-something (fetch item)))
          val)
        (some-operation val))))
```

FIGURE 7.1: Nested context example

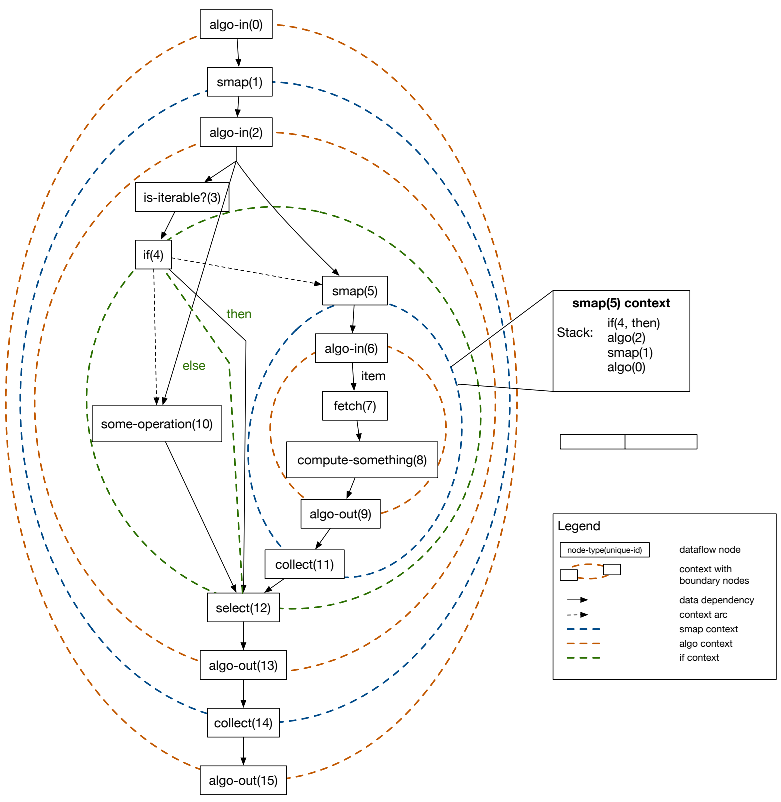


FIGURE 7.2: Visual example of context nesting with exploded stack

7.1.1 The stack

A context stack $S(n)$ is a ordered list of nested contexts a node is contained in. In the following paragraphs I will sometimes speak of a context and sometimes of a context stack. Both are equivalent, since every context can be uniquely associated with a context stack and vice versa. The context associated with a stack s is the topmost context of s. The stack associated with a context c is the context stack of its exit node. See also Figure 7.2.

Notation

Symbol	Semantics
$c \rightarrow_{pop} c'$	Removing the topmost element from the stack c yields c'
$c \rightarrow_{pop}^{n} c'$	Removing the topmost n elements from the stack c yields c'
$c \rightarrow_{pop}^{*} c'$	Removing an unspecified amount of entries from the top of c yields c'

Definitions

If there are two context stacks s_1 and s_2 and the successive removal of elements from the top of s_1 yields s_2, then we say s_1 is *nested* in s_2 or a *child context* of s_2 and s_2 is a *subcontext* or *parent context* of s_1.

$$s_1 \rightarrow_{pop}^{n} s_2 \mid n > 0$$

A root context r is the context of the empty context stack $r = []$. The amount of elements $D(s)$ which need to be removed from a stack s to reach the root context is its *nesting depth*, which equates to the height of the stack.

$$D(s) := s \rightarrow_{pop}^{d} r$$

Truncating $T_n(s)$ a stack s to n means to remove elements from the top until a nesting depth of n is reached. If the nesting of the original stack was already lower than n the truncated stack is the original stack itself.

$$T_n(s) := \begin{cases} s' \mid s \rightarrow^*_{pop} s', D(s') = n & D(s) > n \\ s & D(s) \leq n \end{cases}$$

Context effects aggregate in a context stack.

Properties

There are a few general properties which can be asserted about contexts in the Ohua sense.

- **Contexts are naturally fully enclosing.** This means that in a valid program[2] two contexts never partially overlap. Truncating both to the length of the smaller stack either yields the same stack for both or one were no element occurs in both stacks. Formally

  $$\forall s_1, s_2 | n = \min(D(s_1), D(s_2)) := \begin{cases} T_n(s_1) = T_n(s_2) \\ e_{s_1} \neq e_{s_2} \mid e_{s_1} \in T_n(s_1), e_{s_2} \in T_n(s_2) \end{cases}$$

  An example of this property can be seen in Figure 7.2. Every context is completely embedded within any of its parent contexts.

- **Contexts are an inherited property.** Nodes which are not special, context changing nodes such as `smap` or `ifThenElse` inherit their context from their preceding nodes. Briefly said a node N is in every context any of its preceding nodes is in. Since, as mentioned above, contexts do not partially overlap, contexts of preceding nodes are always subcontexts of one another.

Contexts are special in Ohua because as of yet they are exclusively builtin and cannot be user defined.

7.2 Detection

7.2.1 IR Based

There are mutliple ways of detecting the context at various stages of the compiler This algorithm employes a strategy whereby the IR graph would be traversed in topological order. Each nodes context would be calculated from the maximal context out of the contexts from the preceding nodes (see inheritance property). Maximal context here means the most deeply nested (see full enclosure property). Then depending on the type of node this context stack would be transformed.

1. If the node opens a new context (`smap`, `if`) a new frame for that context is pushed onto the context stack.
2. If the node closes the current topmost context (`collect`, `select`) the topmost frame is popped off the stack.
3. If the node neither closes nor opens a new context stack remains the same.
4. If the node closes a context which is not the current topmost context an error os thrown.

The node would then be labeled with the transformed context stack.

[2] On the level of dataflow operators one might be able to construct graphs which contain partially overlapping contexts, however the resulting program would be invalid in that it breaks some invariants and most likely either not execute or never terminate.

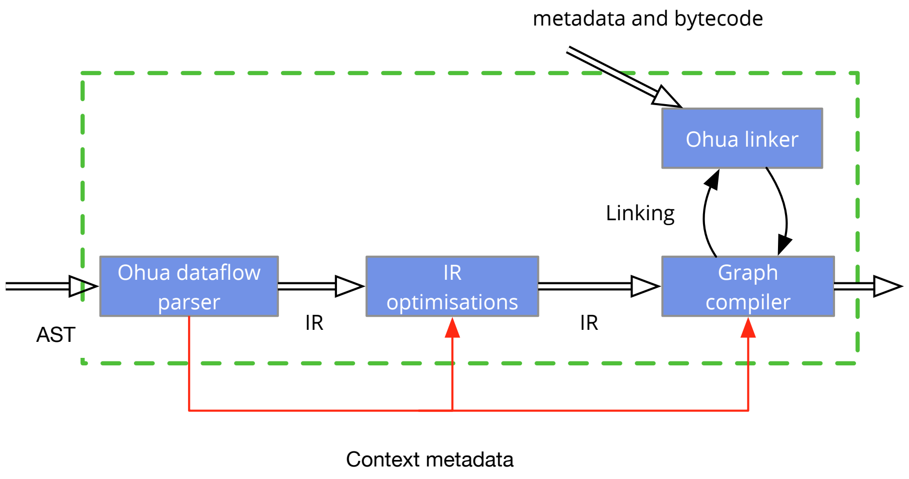


FIGURE 7.3: Context information in the Ohua compiler flow

7.2.2 Source based

Alternatively context detection can be performed during the AST traversals on the source code. This subgraphs of the context are now easier to find as they are just a subtree in the AST. Currently in Ohua context detection is done on the AST, when the Clojure code is being transformed into the IR, see Figure 7.3.

At this point the subgraphs enclosed by the context are even more easily visible. When we traverse the AST and we encounter a context introducing function like `smap`, we mark the enclosed function such that when it is being traversed all nodes inside are annotated with a reference to the `smap` starting node. This overwrites any previous marker. What we end up with, as a result, is an IR, where every node has, in its metadata, a reference to its closest context starting node, a visual representation from our example of the nested contexts from earlier can be found in Figure 7.4. The context starting node then has a reference to its nearest outer context and thus we can rebuild the entire context stack for each node afterwards. For this we use dynamic programming by populating a HashMap with the stacks for each node, reusing previously computed results. This can be done safely because Clojure standard library data structures like `Vector` are immutable and therefore safe to reuse. After this process is completed we obtain a Map from node ids to completely resolved context stacks which is handed to subsequent transformations.

7.3 Unwinding nested context

Contexts like `if` and `smap` can be arbitrarily nested and interleaved in a program. The detection algorithm, as described above, is able to correctly identify stacks of contexts for each node of the graph. Subsequently we would like to ‘undo’ each layer of context around each fetch to guarantee the node is only executed once per program run, which allows us to batch it. This ‘undoing’ operation should be reasonably efficient and yet easy to understand.

My context unwinding algorithm, which combines the `if` and `smap` transformations to deal with arbitrary nesting, uses the properties of contexts as mentioned above. It unwinds one context at a time, dispatching to the appropriate handler, see Figure 4.1, starting with the deepest nesting level. This unifying unwinding algorithm is not limited to `if` and `smap` contexts, but can be configured as to the type of contexts it dispatches on. The reason this results in a correct unwinding is that if a particular context has been unwound around a particular fetch, this fetch now lies in what was formerly the parent context. Since this parent context cannot

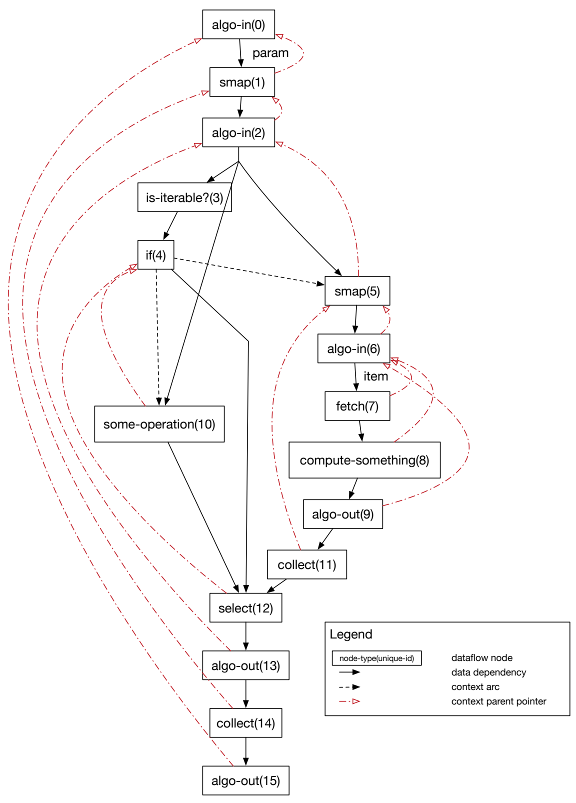


FIGURE 7.4: Visualisation of context pointers after the AST traversal

have been as deeply nested as the now unwound child context, it itself has not been unwound yet since we go from high nesting to low nesting. At the same time since all contexts are eventually being unwound the parent context has already been scheduled for unwinding. Through this iterative unwinding all contexts around fetches are eventually unwound completely.

The correct order is achieved by using a work list of contexts to unwind. The list is obtained by collecting the set of all context stacks on fetches, adding all substacks, ordering by length and finally mapping to the topmost element of the stack. This last list is guaranteed to have no duplicates, due to the “fully enclosed” property of contexts.

In order to be as efficient as possible transformations are performed in a stateful way. Individual transformation algorithms insert and delete nodes from the graph, insert and delete edges, as well as alter contexts for existing nodes or label new nodes with appropriate contexts. Currently no sanity checks for the validity of those alterations are present, therefore the individual transformations themselves are responsible for preserving the integrity of the graph and its associated metadata structures. This is not a hard limitation and might change in the future if we decide to allow more or user defined rewrites.

Chapter 8

Transformation Implementation

8.1 Guidelines

In general, when designing the Ÿauhau graph transformation, I have opted for a simplicity focused approach. The following sections describe some properties of the algorithms which I have tried to adhere to as much as possible.

8.1.1 Single Concern

Each transformation should be as self contained as possible. The less overlap there is between any two graph transformations the easier they are to implement and the easier to debug.

Therefore both context rewrites `if` and `smap` are separate transformations which are combined using the generalised context rewrite algorithm. None of those rewrites know of the other. If and smap have no overlap in the types of nodes they insert or delete (apart from the `fetch` node of course for which both of them change the associated context stack). Also the combining context rewrite has no knowledge of the inner workings of any of the individual unwinding transformations. Its only concern is to correctly handle nesting. Therefore it simply dispatches rewrites based on type of context encountered. Rewrites are basically just implementations of an interface.

8.1.2 No complicated edge case optimisation

No regards for special cases unless necessary for semantic correctness. Algorithms should handle the problem as generic as possible. This makes the implementation simpler, as less special cases have to be distinguished. The generic implementation of course is required to be semantic preserving for all cases, which often means inserting redundant operators.

In general this rule introduces a fair number of redundant operators. The worst perpetrator of this is an operator called `identity` which is used to preserve destructuring in fetches on if branches. When two fetches which were on different branches of the same if are merged it may be that one of the fetches had its return be destructured differently than the other. If this is the case the (IR) returns of those ifs cannot be unified. Therefore a new binding is added leading from the combined fetch to two `identity` operators. Each of the `identity` operators is destructured like one of the former two fetches and wired to the same subsequent nodes respectively. A context arc from the `ifThenElse` operator selects one of the `identity` operators at runtime depending of the active branch.

The reason this is so inefficient is that, other than destructuring data, the `identity` operator does literally nothing, thus we have to schedule a bunch of operators at runtime which are essentially NoOp's.

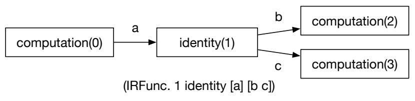
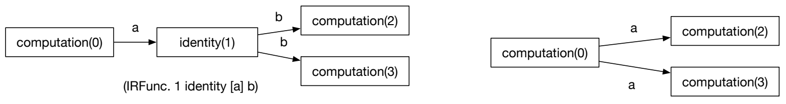


FIGURE 8.1: Identity use example



(A) Redundant identity example (B) Optimised flow

FIGURE 8.2: Optimising `identity` operators

8.2 Separate Optimisations

To get back the lost efficiency from the naive and simple transformations Ÿauhau separately introduces an optimisation pass. This means the overall transformation is divided into three steps

1. **Context unwinding.** Rewriting all contexts around fetches such that every fetch is only called once.
2. **Batching.** Replacing all fetches with rounds of accumulated fetches.
3. **Optimisations.** Clean up the graph, remove redundant operators.

Optimisations of this kind are high level. They operate entirely on the dataflow IR and only concern themselves with Ÿauhau and Ohua internal operators. Optimisations are repeated until the graph does not change anymore.

8.2.1 Identities

As a result of, in particular the if rewrite, a lot of `identity` operators get added to the graph and not all of them are strictly necessary. The typical use of an `identity` operator is to destructure a piece of data, see Figure 8.1. This is used after the if rewrite to destructure the data differently depending on the chosen branch. Since the identity operator itself doesn't actually perform any calculation flow of the form seen in Figure 8.2a has no practical use. This optimisation removes those kinds of identities. Specifically if the output of an identity operator is not destructured then there is no need for it, hence it can be removed and each occurrence of the return binding is replaced by the input binding and a potential context arc is connected to the recipient as well, see Figure 8.2b.

A second `identity` optimisation is the removal of duplicate identities. If two identities with the same input are destructured into the same output they are redundant. Both are removed from the graph and data is destructured at the source. Both context arcs are connected to the respective recipients.

Chapter 9

Side Effects

In terms of Ÿauhau, side effects are actions which modify a remote resource that Ÿauhau is configured to access. They are called side effects because they violate the rules of referential transparency. Referential transparency is a property of pure functions and means that the result of the overall computation does not change when the call sites of the referentially transparent functions are replaced by their result. In Ÿauhau we make the assumption that, even though we do not control the implementation of the fetch performed by the data source or have control over the remote source, read requests are pure actions. This allows us to cache results and remove duplicate requests. In practice the remote resource may of course change as the program runs which is why we allow the programmer to choose an appropriate caching strategy with the desired tradeoff between performance and out-of-date errors.

9.1 Current implementation

This behaviour is similar to the behaviour of Haxl, which also caches fetch results and removes duplicate requests. We however wanted to go one step further and provide the user with the ability to perform transformative actions on the remote resource during execution. Effectful actions are of course already supported by Ohua, however as mentioned before it breaks the assumptions made by Ÿauhau. Therefore the programmer would either have to accept the possibility of errors or relinquish the use of a cache, which has profound impacts on performance.

We solve this issue in Ÿauhau by specifically implementing write requests. These special `write` operators are not only able to access the network in a user defined way, but it also informs the cache of the performed request. When the user defines a cache he can also provide a function for how to manipulate the cache as a result of this request. Two simple predefined actions are available by default `dropOne` and `dropAll`. The former drops a request from the cache matching the current one and the latter clears the entire cache.

9.2 Enforcing intra algorithm order

The mentioned cache handling allows programmers to obtain correct read-write semantics. However there are still issues due to ordering. In Ohua order of execution is only guaranteed for operators with explicit data dependencies. In the example in Figure 9.2 for instance it is entirely indeterministic whether the read or write is going to be executed first. This problem can be avoided by using an operator called `seq`. `seq` enforces its first argument to be fully evaluated before allowing its second argument to be evaluated. The inspiration for this operator comes from Haskell, which has a function called `seq`, however the semantics are subtly different. Whereas Haskell's `seq` only guarantees that then it returns its second arguments both arguments have been evaluated to WHNF, Ohua's `seq` guarantees that *before* the latter argument is being evaluated the first has been *fully* evaluated. In the example Figure 9.3a we enforce the write to happen first using seq.

```
/* Caches should implement both interfaces */
interface Dropable<A> {
  Dropable<A> dropOne(A a);

  Dropable<A> dropAll();
}

interface Cache {
  BiFunction<Dropable<Request>, Request, Dropable<Request>>
  getRemoveAction();
  Object get(Request r);
  Object set(Request r, Object resp);
  void handleStore(Request r);
  int cacheSize();
}
```

FIGURE 9.1: The cache interface

```
(defalgo reqs [req]
  (let [_ (write req)
        r (read req)]
    r))
```

FIGURE 9.2: Example for if independent actions

We could also achieve the same semantics by introducing data dependencies between the two function calls as seen in Figure 9.3b. These are typical semantics as would be expected of a program written in a non sequential language such as Ohua or Haskell due to laziness. However one problem remains.

9.3 Enforcing order inter algorithm

What if we have a similar situation as above, but the read and write requests are in two algorithms? An example of this may be seen in Figure 9.4. Here we have basically the same situation as above, but this time the programmer tried to introduce the data dependency by making one algorithm depend on the result of the other. Semantics of most languages, such as, again, Haskell for instance, this should work. However in Ohua all algorithms are directly spliced into the main program except for two boundary operators called `algo-in` and `algo-out`. The boundary operators however only track in and output to the alorithm. An independent fetch like the one we have here in `does-read` has no data dependency to the algorithm boundary. As the result of our write request is only connected to the algorithm boundary, via a parameter, there is no data dependency between the read and the write request after splicing.

If as a result the order of the `read` and `write` request was nondeterministic it would be counterintuitive to the programmer. He would expect regular program semantics to hold in this case, which they do not do. There is of course still the `seq` operator, which we could use to enforce ordering and that would work even across algorithms, but require additional work from the programmer. Therefore we would like to automate the addition of `seq` operators in the program where there are independent reads or writes that may cause unexpected indeterminism.

```
(defalgo reqs [req]
  (let [w (write req)
        r (seq w (read req))]
    r))
```

(A) Enforcing order using seq

```
(defalgo req [req]
  (read (write req)))
```

(B) Enforcing order with data dependencies

```
(defalgo does-read [req some-data]
  (let [res (read some-constant)]
    (computation res req some-data)))

(defalgo does-write [req]
  (write req))

(defalgo reqs [req]
  (does-read req (does-write req)))
```

FIGURE 9.4: Reads and writes in algorithms

9.4 Rewrite

Fundamentally we only need to `seq` algorithms together when we have one of the typical read after write, write after write or read before write conflicts and the latter operator is data independent from the former. Therefore we must first identify the dependency relationships between the algorithms.

We build a directed graph of data dependencies between algorithms by first finding all distinct algorithm context frames which have been assigned in the dataflow graph of our program. These context frames become the nodes of our dependency graph, which is a directed graph where edges represent data flowing from the source algorithm to the target algorithm. Edges are calculated by first finding algo exit nodes, called `algo-out`. From an `algo-out` node we traverse the data dependencies of the program graph downwards until we either

- encounter an `algo-in` node, in which case we insert an edge from the algorithm belonging to the `algo-out` node we started from to the algorithm belonging to the `algo-in` node we just encountered, into the dependency graph. We do not explore this path further since following dependencies are guaranteed to be covered by transitive relationships.
- encounter an `algo-out` node, in which case we are nested in another algorithm and we stop exploring this path since subsequent dependencies are covered by the enclosing algorithm.

Thus we obtain a version of a dataflow graph which only shows data dependencies between algorithms.

Following that we assign labels to each algorithm indicating whether the algorithm in question performs reads and, or writes. This is done by compiling sets of algos with reads and writes. I filter the IR function list for `fetch` and `write` nodes. Map each node to its associated algo contexts, concatenate these and finally remove duplicate entries. Now we have a set of read and write performing algos respectively. Each algorithm in the graph is then labeled with `:does-read` and `:does-write` depending on which set it is member of.

Finally for each algorithm which writes the algorithm gets `seq`'ed to its algorithm boundary. All subsequent reading algorithms, which are not dependent on another write get `seq`'ed to the writing algorithms return value.

Chapter 10

Extending the code generator

10.1 Maps

Adding meaningful mapping operations to the code generator is not trivial. Meaningful mappings should be done over a large collection. However at the present the code we generate does not produce large mappable structures. It deals mostly with small numeric values to avoid having to deal with the type system too much when generating code.

10.1.1 Current status

The code generated by our generator is determined by two things

1. **The label of the node** determines the type of function call the node will translate to. It holds potential extra information necessary for this type of function.

2. **The successors** become the arguments to the function. Hence they control how many and which arguments a function receives.

Return values of functions are never large mappable structures, as mentioned above. Therefore we have to build mappable structures specially for map operations. However the only data available to us to do that with are the successors to our node. Therefore we could only build mappable structures as large as the number of successors to our node. In order to attain reasonably sized programs however the number of successors a node can have is limited to a maximum of 6. Lastly Ohua is currently unable to handle mapping over empty collections and also sometimes issues arise when mapping over collections of only one element. This is not an issue for our experiments per se, since mappings over empty or one element structures provide no meaningful data to us anyways. However we have to ensure that the generator tool does not generate maps over empty collections and for safety reasons also none for collections with only one element.

When the appropriate command line parameters are given the generator randomly assigns the `Map` computation type to nodes. I tweaked the serialiser, which turns the graphs into actual source code to wrap the results from the successor nodes into a collection and generate a new independent function with which to map over the collection. Furthermore the generator has been tweaked to fix the arity for independent functions which are used for mapping to 1[1]. See produced code in Figure 10.1.

10.2 Conditionals

For our conditional experiments we need random programs in two different versions. Explanation for why this is necessary can be found in Section 11.2.2.

[1]2 for Haxl since I also track an index to circumvent the implicit cache in Haxl which cannot be deactivated.

```
(defalgo ifnlocal3 [param1]
    ...)
(defn main []
  (ohua
    (length (smap ifnlocal3 (vector local4 local5 local6)))))

ifnlocal3 param1 = ...
main = do
    a <- fmap length (mapM ifnlocal3 [local4, local5, local6])
    return a
```

FIGURE 10.1: Example for generated mapping code in Ohua and Haskell

```
(defn main []
  (ohua
    (let [local1 (...)
          local2 (...)
          local3 (...)]
      (if local1 local2 local3))))
```

FIGURE 10.2: Old serialisation for conditionals

In one version computations on `then` and `else` branch of an `if` are left in place, and in another the value of each branch is moved out of the branches and computed *before* the `if` itself is evaluated. In order to make this work with the current generator setup I changes how the branches of an if are handled. Previously a conditional node would always have three successors. The first one would become (part of) the condition. Depending on the evaluation of the condition one of two branches would be selected, and each branch would contain a reference to the result of one of the other two successors respectively, see a schematic example in Figure 10.2.

In order to get a full subgraph rather than just a reference onto these branches I changed the generator to generate two new functions per conditional node, one for the *else*, one for the *then* branch, see Figure 10.3. The first successor still becomes (part of) the condition but now each branch contains a function invocation to respectively one of the newly generated functions where the input parameters are all of the results from the successor nodes.

Now I can decide to either precompute the values for the branches by moving the function invocation out of the conditional statement, into a `let` binding, see Figure 10.3. Or alternatively put the function invocation in the conditional expression, see Figure 10.4. In terms of batching opportunities calling functions on these branches is equivalent splicing a subgraph there because that basically how Ohua handles calling functions/`algo`s. A new command line parameter was added to the generator which is used to select one of the two styles.

Apart from this difference (Figure 10.3 and Figure 10.4) the generated graphs and functions are identical.

```
(defalgo ifnthenlocal4 [param1 param2 param3]
    ...)
(defalgo ifnelselocal4 [param1 param2 param3]
    ...)
(defn main []
  (ohua
    (let [local1 (...)
          local2 (...)
          local3 (...)]
      (let [thenlocal4 (ifnthenlocal4 local1 local2 local3)
            elselocal4 (ifnelselocal4 local1 local2 local3)]
        (if local1 thenlocal4 elselocal4)))))
```

FIGURE 10.3: New precomputed serialisation for conditionals

```
(defalgo ifnthenlocal4 [param1 param2 param3]
    ...)
(defalgo ifnelselocal4 [param1 param2 param3]
    ...)
(defn main []
  (ohua
    (let [local1 (...)
          local2 (...)
          local3 (...)]
      (if local1
        (ifnthenlocal4 local1 local2 local3)
        (ifnelselocal4 local1 local2 local3)))))
```

FIGURE 10.4: New inline serialisation for conditionals

Chapter 11

Experiments

In this chapter I will show some experimental evidence of the effectiveness of the system which we have implemented. Experiments are usually run against the competing systems Haxl[12]. As mentioned in previous Chapters the code used for testing all systems in equal conditions is generated with our random code generator[7]. Additional boilerplate implementations for test data sources etc can be found in the Ÿauhau repository[1] for the Ÿauhau experiment code and for Haxl in the haxl-test-generated-graph[1] repository.

11.1 Smap

This experiment show the quality of transformation for program graphs containing operations which map over a collection. Our experiment setup for this experiment operates on several graphs of constant depth (number of levels is constant). We gradually increase the relative amount of mapping operations in the graph. Additionally we run the series against multiple seeds to get an average result.

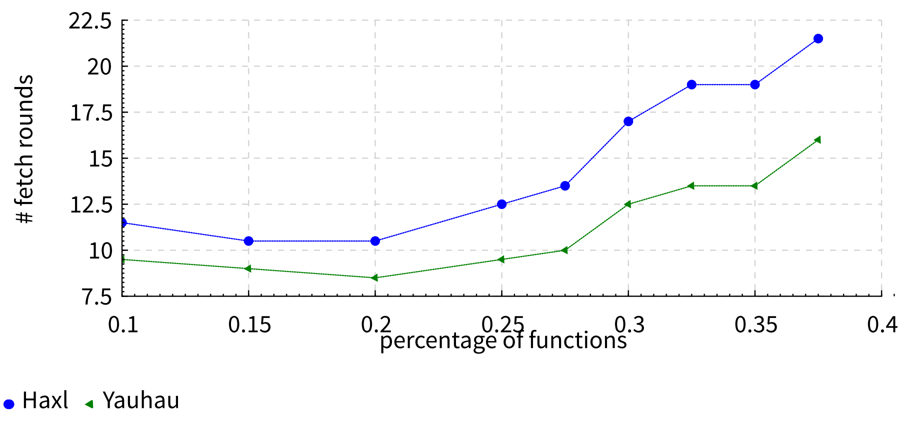
In order to show that we in fact achieve the same performance with maps as Haxl does we run this experiment in two configurations. For both configurations we operate on the same code graphs. After the graphs have been generated we randomly select a number of nodes, with probabilities as listed on the x-axis in the plots. We then change the type, and therefore the behaviour of those nodes, in two ways.

1. The designated nodes become function invocations. Figure 11.1a This is equivalent to attaching a small random subgraph to the node which is run once.
2. The designated nodes become a an invocation of a mapping operation, `mapM` in Haxl and `smap` in Ÿauhau, over a function. Figure 11.1b

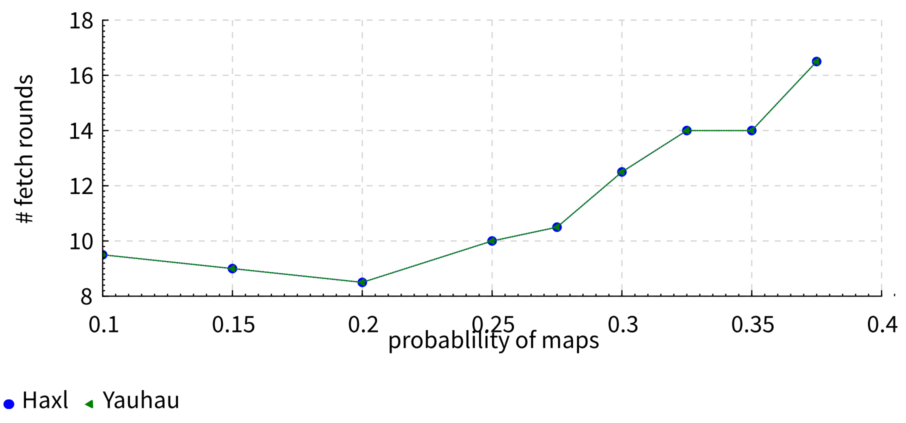
For each node the invoked function and the mapped function are identical in both configurations, only the number of invocations changes. In the first configuration, Figure 11.1a, we can then see the flat performance advantage of Ÿauhau over Haxl independent from any `smap` or `mapM`. We use this data to correct the data from the second experiment by this advantage to only get performance with respect to mapping operations.

In the second experiment, Figure 11.1b, we run the same graph as in the fist experiment, but here the nodes which used to be function invocations are now mappings. The raw result data is then corrected by the difference from the previous experiment. For each percentage of function/mapping nodes the difference in rounds created in the first experiment (Figure 11.1a) is subtracted from the amount of rounds created for Haxl in the second experiment (Figure 11.1b). This accounts for the flat benefit which Ÿauhau gains from the code style independence. As is visible in the plot in Figure 11.1b the performance of Ÿauhau and Haxl with respect to mapping operations is identical to that of Haxl.

[1]https://github.com/JustusAdam/haxl-test-generated-graph



(A) Rounds created with functions



(B) Number of rounds performed with maps enabled

FIGURE 11.1: Influence of mapping operations

11.2 Conditionals

Understanding exactly how the different frameworks handle conditionals is of particular interest because it may holds potential opportunity for optimisation.

11.2.1 Semantics of generated code

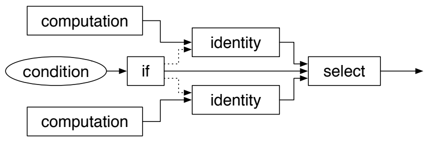
Currently the code generator produces conditional code as seen in Figure 11.3a for Yauhau and Figure 11.2 for Haxl. From the Haxl example it is particularly obvious that values on both branches of the conditional are precomputed, that is the (monadic) operations necessary to compute both values are performed before the condition is being evaluated. This has the consequence that any I/O operation which is necessary to produce this value is performed before the condition for the `if` is evaluated and therefore might render the I/O actions redundant if the value is not selected. Of course the value might also be used in other, subsequent calculations, in which case precomputing is sensible. In fact that is the reason why the graph serialisation in the code generator uses this particular style of code. It does not inquire how many other nodes depend on the result of this calculation and hence obtains no knowledge of whether it is safe to move the computation onto the if branch itself.

```
do
  (val1, val2) <- (,) <$> computation ... <*> computation ...
  return $ if condition then val1 else val2
```

FIGURE 11.2: Generated conditional code for Haxl

```
(let [val1 (computation ...)
      val2 (computation ...)]
  (if condition val1 val2))
```

(A) Clojure representation



(B) Graph representation

FIGURE 11.3: Generated conditional code for Yauhau

The code generated for Ÿauhau, see Figure 11.3a gets translated into a dataflow graph with similar semantics to those of Haxl. A visual representation of this can be seen in Figure 11.3b.

As an example we will take the program from Figure 11.3a. However a simple change to the Ÿauhau code, see Figure 11.4a can produce a graph with a different structure, see Figure 11.4b.

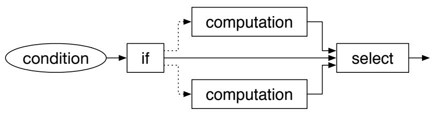
In the second version both computations are dependent on the `if` via a *context arc* (dotted arrow) and therefore would be computed *after* the condition had been evaluated. In the first version the computations do not depend on the `if` and there is no context arc, instead the context arc goes to the `identity` operator which itself does nothing but return its value. What this does as a result is select one of the two computed values, depending on the condition. In summary: In version one the `if` selects one one *computation* out of two, and in version two one *binding* out of two.

11.2.2 Evaluation of precomputed conditional branches

Let us assume that, in the system using Haxl or Ÿauhau computations are pure and the only stateful actions are the I/O actions performed using the framework. This view is of course not entirely consistent with reality, since nothing prevents you from doing effectful things in Ÿauhau, however in the domain where this system may be used it is a reasonable assumption to make for the purposes of the following deliberations. Furthermore we may assume our fetches/reads to be *pure* in so far as that they do not mutate the resource they request data from and serving a cached copy of a request is equivalent to actually performing the request. How Ÿauhau deals with cases in which these assumptions do not hold is explained in detail in Chapter 9.

```
(let [data1 (get-data request0 source1)]
  (if (computation data1)
    (get-data request1 source1)
    (get-data request2 source2)))
```

(A) Requests on branches

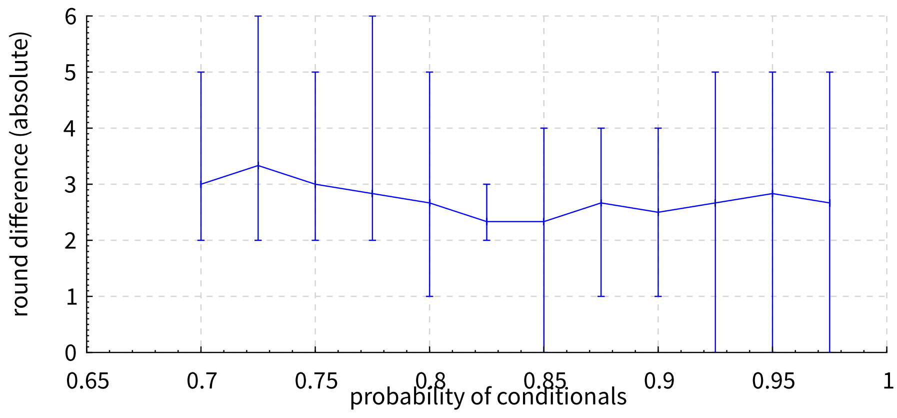


(B) Conditionals as hypothesised

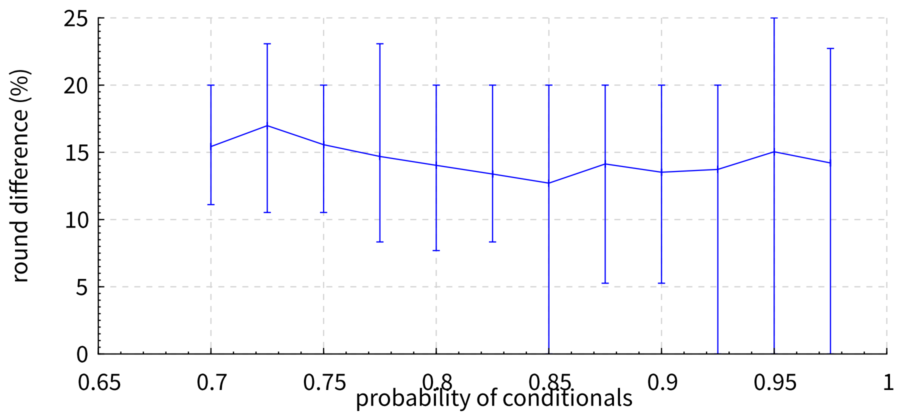
FIGURE 11.4: Representation of conditionals

If we assume the mentioned, lets call it *near purity*, we can start to consider code reordering as an optimisation. In particular reordering code around conditional statements is a possible code optimisation. It turns out that, when performing batching transformations, like we do, moving requests out of, or into conditional branches has more intricate and interesting effects than it would in a regular program.

New batching opportunities



(A) Absolute



(B) In percent

FIGURE 11.5: Round difference with precomputed conditionals

We have made amendments to the code generator to enable us to produce both types of graph, see Section 10.2. The experiments in Figure 11.5a show that an absolute difference in the number of rounds which are performed in the program between when requests are precomputed and when they are not. The figure shows an average difference in the number of performed rounds, as well as a highest and lowest value for the number of rounds saved. The formula is always $precomputed - inline$. Since the minimum value is never below 0 we can conclude that the precomputed version never performs more rounds than the inlined one and on average we save three rounds of I/O when precomputing the branches. Figure 11.5b shows the average, minimum and maximum percentage of rounds which were saved my precomputing the branches.

By precomputing branches we can decrease the number of rounds in a program on average by about 15% and up to 25%. Simultaneously we can observe a significant increase in the number of performed fetches compared to the inlined version, see Figure 11.6.

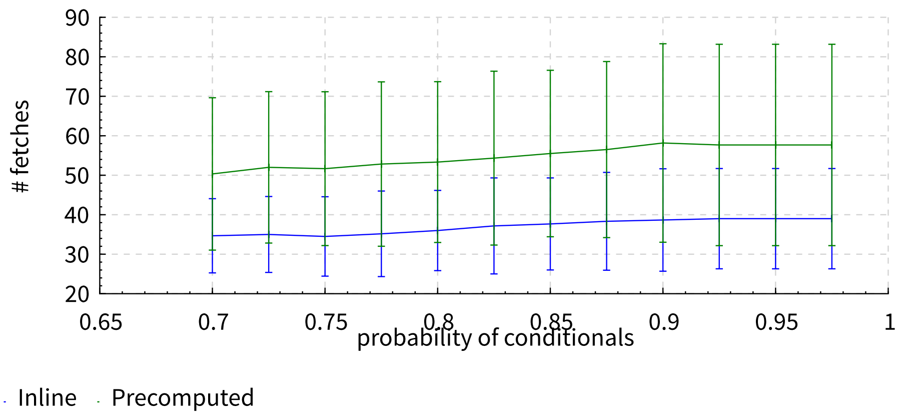


FIGURE 11.6: Number of fetches performed

In Ÿauhau we generally operate under the assumption that I/O actions are expensive. The goal of our system is to do as few actual I/O actions as possible. Fewer rounds of I/O generally reduces the execution time for the program.

Explanations

Consider a simple example (Figure 11.4a) where we have a regular conditional statement and depending on the result of the computation one request. Consider how this program would be batched. The requests on the branches depend on the the evaluation of the condition which in turn depends on the data from `request0`, see also Figure 11.7c. Fetch rounds are indicated by the blue, dashed rectangle. As a result we would end up with two fetch rounds. One for the request before the conditional and one for either the request from the true branch or the false branch.

In this case we see that we are performing two actions where we might only have to perform one action. If we can, as previously mentioned, consider the fetches and computations as pure, we can rewrite the program to perform the conditional fetches **before** evaluating the condition (See Figure 11.7a), which allows it to be batched with the first request, since the two fetches now do not depend on the condition anymore, See Figure 11.7b. The round again indicated by the blue, dashed rectangle. In particular if the fetch round before already includes a request to `source2` as well, then after our rewrite we get all the data from the entire second round in the first. For pure computations this will be semantically identical even though it performs redundant computation.

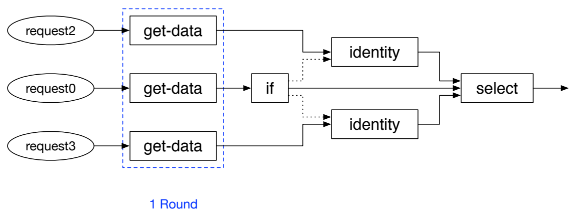
Findings and explanation

We can save up to 25% of rounds and an average of about 15% of rounds by precomputing conditional branches. However these numbers were achieved in programs with a large number of conditional nodes. Up to 95% (of nodes with three children) were tagged and serialised as conditional nodes. Considering this strong bias for conditionals the results seem pretty low.

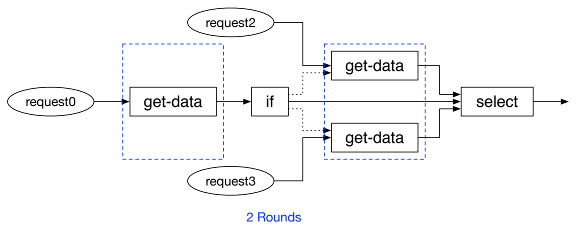
The explanation for these low numbers seem to be that pulling fetches up into an earlier round only reduces the overall number of rounds if said fetch was part of the critical fetch path. Like the critical path in a program we can construct a graph of data dependencies where we only include `fetch` nodes. A visual representation of this can be seen in Figure 11.8 The number of fetches along the longest path

```
(let [data1 (get-data request0 source1)
      data2 (get-data request1 source1)
      data3 (get-data request2 source2)]
  (if (computation data1)
    data2
    data3))
```

(A) Requests precomputed



(B) Precomputed requests as graph



(C) Request on branches as graph

FIGURE 11.7: Precompute transformation

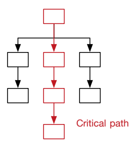
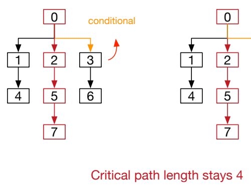


FIGURE 11.8: Critical path of fetches

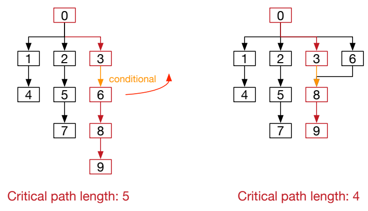
through this graph is the critical path with respects to fetches and marks a lower bound to the number of rounds we have to perform. If the fetches we pulled from their branches were not part of the critical path pulling them up wont change the overall number of rounds at all, compare Figure 11.9a, since there would have been a necessary later round which we cannot avoid performing and which could have incorporated the fetches from our branch. Conversely if the fetches were part of the critical path, pulling them up reduces the length of the critical path, see Figure 11.9b, ergo the optimal solution to minimal rounds decreases and our algorithm finds it. Note that this is not limited to fetches on the critical fetch path of the original program but also to fetches on the new critical path resulting from a rewrite.

In our experiments all fetches had the same execution time, however if we consider a system with fetches of heterogeneous latency we find another potential source of increased performance. If the fetches we pulled into earlier rounds have a particularly long latency we might observe decreased execution time, even if they were not part of the critical path, since we would be increasing the amount and length of program paths able to run concurrently with the fetch. Simultaneously however we might significantly increase runtime if we precompute a particularly expensive fetch which in the end turns out would not have been selected by its conditional.

Overall precomputing request seems a promising area for optimisation, however it would require a smart heuristic for selecting sites which are likely to yield a



(A) Moving fetches not on the critical path



(B) Moving fetches on the critical path

FIGURE 11.9: Effect of the critical path

performance benefit and perhaps additional information about the data sources to decide whether a certain rewrite is beneficial. This goes beyond the scope of this thesis.

Chapter 12

Conclusion and Future Work

With the introduction of contexts and exploration of their properties I was able to create a generalised and extedable unifying context unwinding algorithm for preparing graphs for the Ÿauhau transformation. Should a new context structure be added to Ohua the unified unwinding algorithm can be extended by providing a rewrite for only level of the new context.

A new graph rewrite for automatically introducing necessary `seq` operators into a program with side effects allows Ÿauhau programs with intuitive side effects in the form of writes to a datasource.

Code generator extensions enable new experiments, which show how Ÿauhau delivers comparable, if not better, performance compared to the similar system Haxl while allowing modularity.

An evaluation of the effect of precomputing branches of conditionals lays a foundation for perhaps future optimisations by the compiler. Precomputing conditional branches can noticeably decrease the number of performed rounds in a program and thus potentially significantly decrease overall execution time. However an implementation would require a sophisticated selection algorithm for finding sites where this optimisation is beneficial.

12.1 Formalised rewrite rules

Rewrite rules as applied by both the context unrolling and the batching transformation are not formally verified to be correct. An aim of future work in this direction, not just from Ÿauhau but the Ohua system in general could be the formulation of a core set of program rewrites, primitives, which are then verified to be correct. Subsequently the transformations as described in Chapters 6, 5 and 4 could be described in those abstract terms and verified to be correct and semantics preserving. We have found inspiration for this in a paper "Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code"[17], where formally verified rules are used to decompose and transform high-level functional expressions.

12.2 Subgraph batching

Because the context rewrites introduce so many additional operators for each unwound node it may be beneficial to batch some of those nodes ahead of time to save unwinding nodes. The principal approach here is to apply the round detection algorithm to the subgraphs of a context and accumulate many independent, parallel, single fetches into a multi-fetch so then we only have to unwind the fewer multi-fetches and not each individual fetch. This subgraph batching could also be used to help Ÿauhau to deal with cyclic graphs, which as of yet do not exist.

12.3 Pattern based code generation

In the process of generating the code for the experiments, especially `if`, it became apparent that it is desirable to create random code with specific characteristics and patterns. In the particular case of the if experiment it would be desirable to create nodes where branches are either precomputed or on the branch. Currently this needs to be hardcoded in the generator and switched via a command line parameter. For future work there should be a way to define ad hoc patterns for how a node, or perhaps even whole subgraphs matching certain requirements should be serialised. Combine this flexibility with a probability for the pattern to be applied and most of what is now a hard coded transformation of graph nodes could be defined with those rules.

Bibliography

[1] Justus Adam, Sebastian Ertel, and Andrés Goens. *Ÿauhau*. 2016. URL: https://bitbucket.org/ohuadevelopment/yauhau.

[2] Lennart Augustsson et al. *The Haskell programming language*. 2010. URL: https://haskell.org.

[3] Sebastian Ertel. *Ohua*. URL: https://bitbucket.org/sertel/ohua.

[4] Sebastian Ertel, Christof Fetzer, and Pascal Felber. "Ohua: Implicit Dataflow Programming for Concurrent Systems". In: *Proceedings of the Principles and Practices of Programming on The Java Platform*. PPPJ '15. Melbourne, FL, USA: ACM, 2015, pp. 51–64. ISBN: 978-1-4503-3712-0. DOI: 10.1145/2807426.2807431. URL: http://doi.acm.org/10.1145/2807426.2807431.

[5] Sebastian Ertel et al. "Ÿauhau: Concise Code and Efficient I/O Straight from Dataflow." In: *POPL '17. In submission.* (2016).

[6] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[7] Andrés Goens. *Random code generator using level graphs*. 2015. URL: https://github.com/goens/rand-code-graph.

[8] James Gosling and Sun Microsystems. *The Java programming language*. 2014. URL: http://java.net.

[9] Rich Hickey. *The Clojure programming language*. 2016. URL: https://clojure.org.

[10] Alexey Kachayev. *Muse*. URL: https://github.com/kachayev/muse.

[11] Grant Malcolm. "Data structures and program transformation". In: *Science of Computer Programming* 14.2 (1990), pp. 255 –279. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/0167-6423(90)90023-7. URL: http://www.sciencedirect.com/science/article/pii/0167642390900237.

[12] Simon Marlow. *Haxl*. URL: http://hackage.haskell.org/package/haxl.

[13] Simon Marlow et al. "There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access". In: *SIGPLAN Not.* 49.9 (Aug. 2014), pp. 325–337. ISSN: 0362-1340. DOI: 10.1145/2692915.2628144. URL: http://doi.acm.org/10.1145/2692915.2628144.

[14] Conor McBride and Ross Paterson. "Applicative programming with effects". In: *Journal of functional programming* 18.01 (2008), pp. 1–13.

[15] Martin Odersky. *The Scala programming language*. 2016. URL: http://scala-lang.org.

[16] Simon Peyton Jones et al. "Simple Unification-based Type Inference for GADTs". In: *SIGPLAN Not.* 41.9 (Sept. 2006), pp. 50–61. ISSN: 0362-1340. DOI: 10.1145/1160074.1159811. URL: http://doi.acm.org/10.1145/1160074.1159811.

[17] Michel Steuwer et al. "Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code". In: *SIGPLAN Not.* 50.9 (Aug. 2015), pp. 205–217. ISSN: 0362-1340. DOI: 10.1145/2858949.2784754. URL: http://doi.acm.org/10.1145/2858949.2784754.

[18] Wouter Swierstra. “Data Types à La Carte”. In: *J. Funct. Program.* 18.4 (July 2008), pp. 423–436. ISSN: 0956-7968. DOI: 10.1017/S0956796808006758. URL: http://dx.doi.org/10.1017/S0956796808006758.

[19] Philip Wadler and Stephen Blott. “How to Make ad-hoc Polymorphism Less ad-hoc”. In: *POPL*. 1989, pp. 60–76.