Master's Thesis
Computing Science

Taking a functional datastructure out of context

Raymon Zutekouw

Supervisor
dr. W.S. Swierstra

Second examiner
prof. dr. G.K. Keller

July 2025

UTRECHT UNIVERSITY

DEPARTMENT OF INFORMATION AND COMPUTING SCIENCES

Taking a functional datastructure out of context

Abstract

In purely functional programming languages, operations on pointers are ruled out to guarantee referential transparency. There are language extensions that allow the programmer to write purely functional code, which can be compiled to efficient pointer operations internally. Constructor contexts in Koka are such a language extension, which has introduced a limited API that allows creating values of types which may be extended in O(1) time at the end of the structure by making use of an extra internal pointer. Such constructs remediate the inadequacy of purely functional programming to express code that is cheap and effective within imperative programming.

Currently Koka only allows creating and filling these so called constructor contexts, as the implementation is quite new to the language. But one can imagine that this kind of structure can also be pattern matched on, so that you have access to the constructor and arguments like you would with regular pattern matching. What does the language design of that feature look like? And how does it allow defining highly efficient functional datastructures?

This research specifies this new feature, and contributes the following:

- Demonstrate how to implement queues with O(1) operations with them;
- Define pattern matching on constructor contexts as a specification;
- Prototype the implementation in a recent Koka compiler version.

Acknowledgements

The guidance and expertise offered by Wouter Swierstra has been invaluable to completing this thesis. At each point during the thesis it remained challenging to work on without the task seeming impossible. The support of friends and family has kept me going throughout the entire project. Their unwavering support kept my hopes up that it would all work out at the end.

The guidance provided by the writing coaches at the university skills lab will not be forgotten, they gave professional advice and personal feedback based on my previous writing endeavors.

Regarding the work on the Koka compiler, special thanks go out to Daan Leijen for guiding me through the Koka compiler, which allowed me to build an actual prototype besides just doing theoretical work for this thesis.

Attribution and Rights

Contents

List of Figures

1 Introduction

This thesis fulfills a specific need in the language design of the programming language Koka[6]. According to the documentation of the language, Koka v3 is a research language that is currently under development and is not quite ready for production use. Koka is described as a strongly typed functional style language with effect types and handlers, but there is much more to it that will instead be the focus of this thesis.

1.1 Compilation to efficient code

The language analyzes when code can be made fully in-place, Section 2.1 elaborates on how this works in Koka and discusses the relevance for programming in general. As Koka chooses to be a functional style language, it begs the question of how Koka compares to other types of languages like C++ and Java in terms of performance. According to the benchmarks that the Koka development team provides, their technique for memory reclamation is viable and can be competitive to other programming languges. That motivates further work on the Koka in this area of the compiler, to enable the language to make as much use of these techniques as possible.

One of the ways to optimize for memory usage is not having to allocate

the memory in the first place, and instead re-use the memory whenever possible. Koka has a sense of memory re-use as part of the Perceus[14] reference counting algorithm. With how Koka is designed, the user writes code in a functional style and the language makes in-place updates in a safe way. Lorenzen et al. [8] recently explored a wide class of purely functional programs, and how these in-place updates can be applied.

Starting from the perspective of in-place updates, the representation of the data types is important to make the operations efficient. The follow up research of Lorenzen et al. [9] added support for so called constructor contexts in the Koka language, to capture the imperative notion of algorithms with accumulators efficiently. Their work will be introduced in Section 1.3 as part of the introduction, as they are central to understanding the contribution of this thesis.

Compared to other functional style languages, Koka takes a different approach than just supporting features like IORefs[1]. The languages provides a suite of constructs aimed at writing certain fully-in-place functional programs. The main idea for improvements on the functional side here is to provide new constructs as language extensions. These constructs allow the user to write functional code, while retaining enough guarrantees to make it safe to use mutation behind the scenes.

All the code examples in this thesis are written in the Koka language [6] (v3.1.2). There are examples to introduce the language, along with its concept of fully in-place programming.

1.2 Revisiting list reversal

This section will go back to the basics of reversing a list with functional programming. This section identifies code properties to look for, and gives an example to work with to understand constructor contexts.

The motivation for this research runs all the way back to the work of Okasaki [12] on purely functional datastructures. That work has a collection of various techniques, one of which comes from Hood and Melville [4] to establish amortized O(1) queues.

Those FIFO queues are immutable datastructures, using two lists with one being reversed. And while it is great that it can be done without mutation, there seems to be a puzzle piece missing from the functional paradigm.

[1] as they are present in the programming language Haskell

It feels like it can not be the case that the functional paradigm has to resort to IORefs to express this form of mutability. The mutability has a fixed structure based on extension, and is not just arbitrary pointer arithmetic. To get a feel for this structure, the reversal of a list is used as an example to introduce the concept of constructor contexts. Their current limitations are made explicit in Section 1.6, motivating the work of this thesis to address them.

Reversing a linked list can be thought of as reversing a stack. The list has an easy access to the first element, which should become the last element in the new list that will be constructed. An easy way is to start with an empty second list, which is called the accumulator, and move elements one by one. Figure 1 shows the reversal the three elements `[3,2,1]`, as if it were a stack. It results in `[1,2,3]` at the end, which indeed is the reverse of the original list.

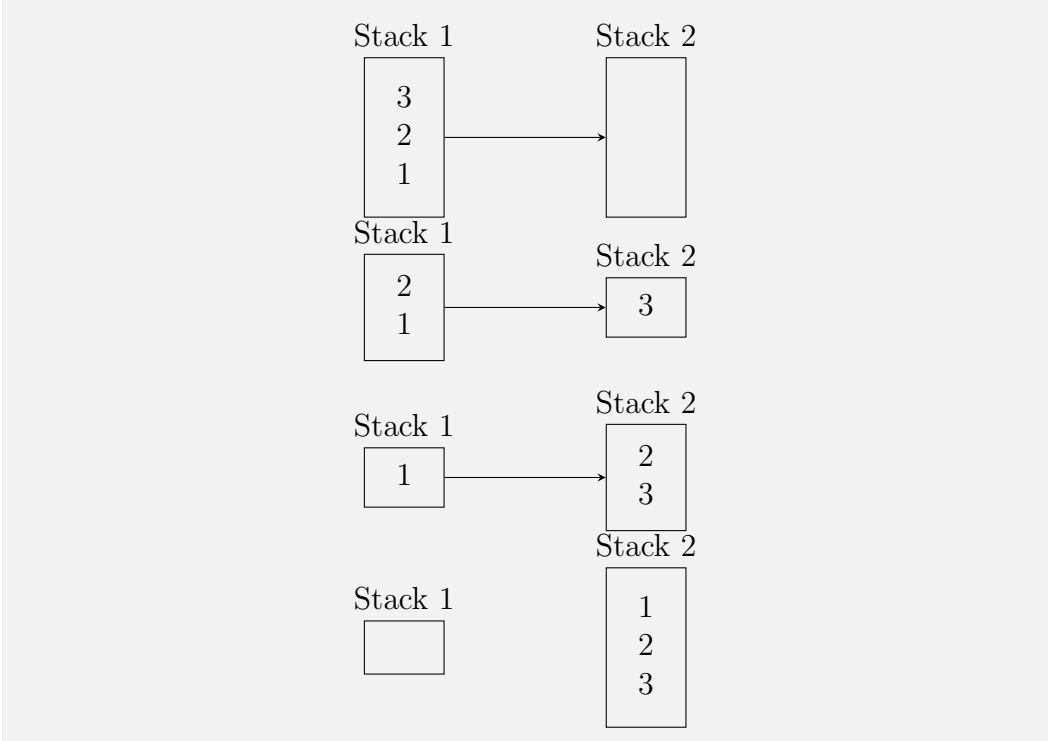

Figure 1: Reversing a stack by transferring elements to another stack

The functional code for this list reversal algorithm will look something like the code in Figure 2. It starts with the original list and an empty list as the accumulator. The `Nil` case is hit when the original list runs out of elements, meaning that all encountered values have been accumulated and can be returned. The `Cons` case is hit when the original list still has elements to be reversed, which is where we move the next element to the accumulator.

There are a couple of interesting properties that can be observed for this piece of functional code. The first thing is the number of allocations. In the `go` function, there are no allocations made in the `Nil` case, and in the `Cons` case there was one allocation on the left (the pattern) and one new allocation on the right (to add to the accumulator). Those allocations cancel out, in Koka that memory can be re-used (assuming that there is no further sharing). This re-use mechanism is explained in more detail in Section 2.1.

```
fun reverse_list(input : list<a>) : list<a>
  go(input, [])


fun go(rest : list<a>, acc : list<a>) : list<a>
  match rest
    Nil -> acc
    Cons(x, xs) -> go(xs, Cons(x, acc))
```

Figure 2: Reversing a linked list with an accumulator

The second thing is about the nature of accumulators like these in functional languages. Notice the way the accumulator is used, it is passed as the recursive argument to build a larger accumulator with the element `x` added. The accumulator remains of the same type, `list<a>`, and grows from the front with the addition of the `Cons` constructor. The previous value of the accumulator is not inspected, it is only wrapped by this new constructor around it.

And this is the point where the functional paradigm starts to create friction. The paradigm does not expect an extension to the datastructure at another position than the front. For that you would have to define a traversal, to match against the various contructors the nested accumulator has, or store

the datastructure differently like with multiple lists. As a reminder, this is why the function `map` returns the arguments in reverse, when tail recursively defined. To adhere to proper mapping, it requires a final reverse at the end.

```
fun map_list(input : list<a>, f : a -> b) : list<b>
  go(input, [], f)

fun go(rest : list<a>, acc : list<b>, f : a -> b) : list<b>
  match rest
    Nil -> reverse(acc)
    Cons(x, xs) -> go(xs, Cons(f(x), acc), f)
```

Figure 3: Mapping a linked list with an accumulator

The third option envisioned here is a list accumulator that should grow from the end, acting like a queue. It functions similar to how imperative languages allow this with mutable pointers, without summoning the hydra pointers lead to. That is where a new concept introduced to Koka comes in, called constructor contexts, which will be introduced in the next section.

1.3 First-class constructor contexts

Koka has introduced constructor contexts as first-class citizens in Koka[9], but what are they? They implement what is called a context in lambda calculus; a term with a single hole in it. The functional representation of such a term has been given by Minamide [11]. Usually when you call a constructor, you provide all the values the constructor asks for as arguments, so that it can construct its type. But in the case of contexts, you should omit one of the constructors. By leaving one of those values open as a hole, you allow extending the datastructure from the end like a queue. What is returned is not yet the type, but an intermediate structure that has a hole in it, which internally has pointer to that position. This construct has strict requirements to make it type safe, and offers a way to express this operation functionally. The current API Koka provides is limited to only two operations.

The operator to extend this intermediate value is `(++)`, which can extend the hole by another constructor that fits and has a hole of the same type.

The intermediate structure can be closed with the `(++.)` operator, which just expects a value of the type of the hole to replace the hole with. Both operators run in O(1) time complexity. Section 1.5 will go over the typing of all these values with an example. Lorenzen et al. [9] have used constructor contexts to efficiently implement top-down algorithms for binary search tree restructuring, by using the constructor contexts as the accumulators.

1.4 Accumulating constructor contexts

Lorenzen et al. [9, sec 2.3] show a top-down version of the function `map`, by using an accumulating constructor context. What will follow is an explanation of the steps, the code is also shown in Figure 4 for reference. It should give a better idea of the operations that are currently available in Koka to work with constructor contexts.

```
fip fun map-td(xs : list<a>, ^f : a -> b
  , acc : ctx<list<b>>) : list<b>
  match xs
   Cons(x,xx) -> map-td(xx, f, acc ++ ctx Cons(f(x),_))
   Nil -> acc ++. Nil

fip fun map(xs : list<a>, ^f : a -> b) : list<b>
  map-td(xs, f, ctx _)
```

Figure 4: The map function for lists using contexts as accumulators seems more complicated at first, this section steps goes through the implementation

From the start of the list, it will apply the function f to each value it comes across, and the accumulator leaves a hole for the next value in the list. All the mapped values of the list are connected back together with the `(++)` operator which maintains the correct order of the list. After the new value is calculated, the context for it is constructed, and it can be filled in for the hole in the accumulator. At the end there are no values left in the list, so the large structure that has been accumulated can be closed; the hole at the end is no longer needed, which returns the whole structure as a list. The last

hole can be filled in with the `(++.)` operator, the last step shows that the final hole is filled with `Nil`. Figure 5 visually shows these steps.

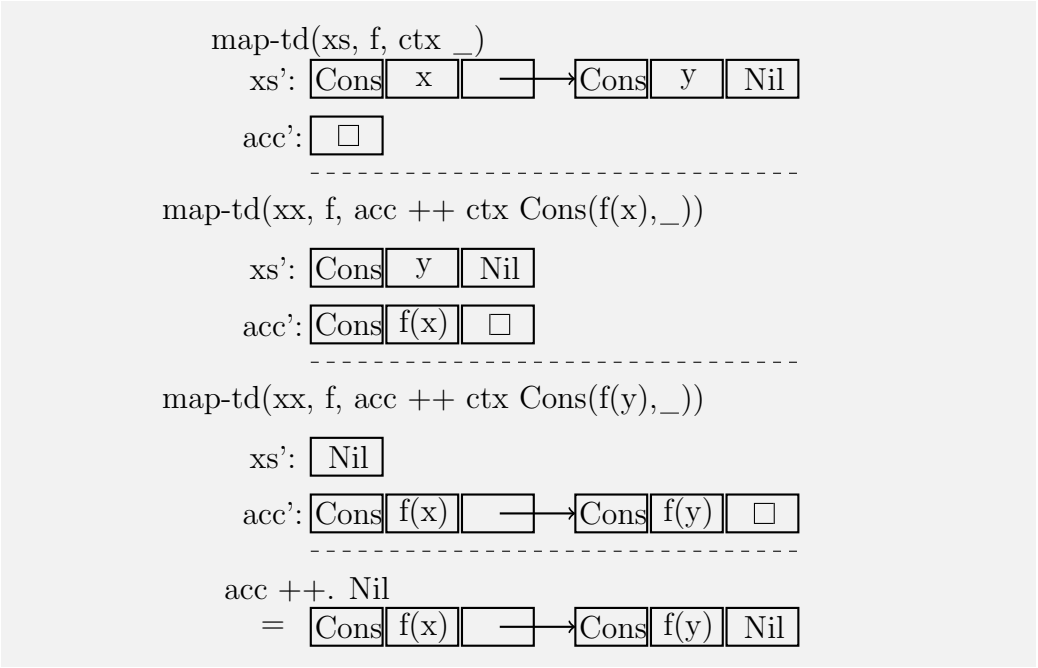

Figure 5: Accumulating a list while mapping makes good use of the accumulator that can can grow from the back, so that a final reverse is not needed

The top-down implementation starts with constructing an empty context, which is a context with just a hole and no values yet. The main idea is that the map operation can be done in one traversal, the context allows moving through the list and making changes in-place. It works like an accumulator, but is more efficient as the compiler also knows how to make this implementation in-place. Using a regular list as the accumulator would have required to reverse the list at the end, which is not needed here.

This forms a function that the compiler optimizes into an in-place algorithm, as it keeps growing the context for every mapped `x`. The mapped value added to the context is stored in a re-used piece of memory, which was previously memory of the original list. The context is only used intermediately, it is not returned as a context but as the full type again. With the

current limited API Koka provides, there is not much utility in the context construct itself as the representation for a datastructure.

1.5 Creating a constructor context in Koka

To help navigating the idea of constructor contexts, a simple example on lists might help.

As seen before, a list is defined with two constructors, `Nil` and `Cons`, where `Nil` is the end of the list and `Cons` allows adding an element `x` to that list. As the second argument to `Cons` is recursive, we can do this any number of times.

```
type list<a>
  Nil
  Cons(head : a, tail : list<a>)
```

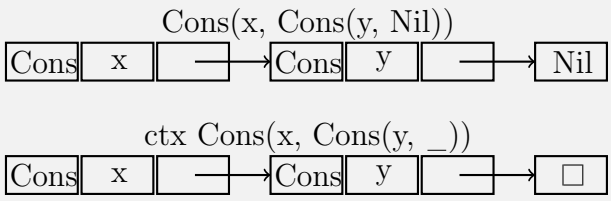

Figure 6: Example list with two elements, x and y, which has a hole at the end to allow further extension to the back of the list

So if there are two elements to store, that constructor `Cons` is used twice and closed off with a `Nil`, as shown in Figure 6. The simple example is drawn out to illustrate a new point; we can add a new perspective to data types, the perspective of extending at the end of the datastructure in a functional way.

A hole (later also shown as _or □) can be introduced to the list type, and placed on the `tail` argument of the last `Cons` constructor, as that is where you would extend the list. In Koka, the `ctx` keyword is used to create constructor contexts, also with constructors in them. With the `ctx` keyword,

it is not an option to use the `Nil` constructor directly because it has zero arguments. There would be no place to place the hole, so the `Cons` constructor is used with the hole on the `tail` argument position. It is also possible to define an empty context, which will be used for when there are no elements in the list. It is created by writing `ctx _` in Koka.

It has the type `list<a>` for the hole, as that is the type of the `tail` argument, and the entire structure is still a `list<a>` if the hole is filled in. The type of this intermediate structure would be `ctx<list<a>>`, which is short for `cctx<list<a>, list<a>>`. In Koka, the type `cctx<a, b>` means the following: `a` is the type it will produce if you provide the value that fits into the hole of type `b`. So that means that for lists, you can extend them with other lists that has a hole `(++)`, or close them with a full list type `(++.)`. It will replace the hole with the value you provide it, without the need to traverse the datastructure. As it is common that the types `a` and `b` are the same, Koka defines a type alias `ctx<a>` equal to `cctx<a,a>`. Those types are always the same when a hole is placed on a recursive argument of a constructor.

1.6 Current limitations

So far, the construct has mainly been used as an accumulator; as an accumulating constructor context. There are currently not many operations you can do on an existing context. These contexts are still a relatively new construct in the Koka language, and still have a limited API built for them. Pattern matching is not yet supported on these contexts, which is the main contribution of this thesis.

Research question: Although the current constructor contexts can be created, combined and filled, there is no mechanism to pattern match on constructor contexts. The aim of this thesis is to support pattern matching on constructor contexts, making it possible to deconstruct constructor contexts without filling them prematurely.

The limits of the current API can be seen when implementing queues in Koka with lists as constructor contexts. It is able to define an efficient `enqueue` in O(1) time, but can not define the `dequeue` operations as it does not have a way to pattern match on a constructor context (yet). Other

functional solutions go into different directions to solve this problem, but the original problem of not utilizing pointers remains.

This thesis prototypes pattern matching on constructor context, so that the `dequeue` operation can be defined. The pattern matching works alongside the existing two operations. Section 4.8 demonstrates that these two existing operations can also be implemented by pattern matching on constructor contexts.

2
Background

This chapter elaborates on the related work for constructor contexts, and how they have been used so far. It will mainly be about the related features that Koka currently supports and about limitations the features currently face. The remainder of this chapter provides motivating examples to further develop the API on constructor contexts.

2.1 Fully in-place programming in Koka

The `fip` keyword on a function in Koka indicates that a static check guarantees the function is fully in-place (FIP). This section explains the relevance of FIP in Koka, as introduced by Lorenzen et al. [8]. An example would be the reversal of a linked list, as seen in Section 1.2. Both functions are fully in-place, as the number of allocations is net zero. Does it also work for other examples? Yes, a typical function that can be done in-place is the map function on a list, where a function is applied to every element in the list.

```
fip fun map(xs : list<a>, ^f : a -> b) : list<b>
  match xs
    Cons(x,xx) -> Cons(f(x), map(xx,f) )
    Nil -> Nil
```

Figure 7: Mapping fully in-place has a near identical defintion to the standard definition of map, the Koka compiler does a FIP analysis of this function

The match on the list constructors `Cons` or `Nil` is destructive, which allows the memory of the `Cons` pattern to be re-used by the `Cons` allocation after the match. As there are no additional allocations, this function is considered fully in-place by Koka.

Sometimes it is inevitable to do additional allocations, but you still want to have some kind of static check. For functions that must do these allocations, you can still add a static check to see if the allocations are bounded. It is possible to specify an amount of additional allocation credits, so for instance `fip(1)` as a keyword would allow up to one additional allocation. Figure 8 shows an example, the function makes one additional allocation for the list, which Koka can verify.

```
fip(1) fun snoc(xs : list<a>, x : a) : list<a>
  match xs
    Cons(y,ys) -> Cons(y, snoc(ys,x))
    Nil -> Cons(x, Nil)
```

Figure 8: Adding an element to the end of a list does not use more than one allocation in total, which allocates only in the Nil case to the value x

2.2 Motivating examples

This section goes over examples that have a use for a pattern matching mechanism in Koka, which are the motivating examples for this work. This section explores what might be possible from a theoretical standpoint, it does

not yet define the technology to make the examples go through. As will be illustrated in the following sections, constructor contexts allow creating top-down algorithms in a functional style, like Lorenzen et al. [9] describe. The benefit to the programmer is that it will provide them with a structure they can efficiently extend and pattern match on in a functional style. These two operations are not exclusive to each other, and by improving Koka it will no longer be needed to convert contexts to do pattern matching on them.

Different patterns

The example of the `map` function shows that constructor contexts can be used, but there does not seem to be a clear benefit at first glance. It looks like a different way of writing the same algorithm, because it is already possible to define `map` top-down as a fully in-place algorithm in Koka. But that is not the case, as it is currently not possible to write it in a tail-recursive way without reverse, in a single pass.

The main idea is that Koka should see the different hole patterns, which can be matched with new the `match-ctx` construct. This should be integrated with the existing `match` construct in Koka, to not add a redundant keyword.[2] The patterns are different than the patterns on a regular data type. The language feature is formalized in Chapter 3. The example on trees is included to demonstrate this, as it can see multiple patterns per constructor. Binary trees are excellent as they have multiple recursive arguments, which the pattern generation for contexts is based on.

It opens up a new paradigm for functional programming, where datastructures themselves can be representated as contexts. The key contribution of this work is that the API for contructor contexts becomes more expressive, so that constructor contexts can be more than just an accumulator to some calculation. This extension allows defining a queue as a list context in Koka, which makes it highly efficient while maintaining memory usage properties. It improves the top-down implementation of `snoc` to the end of the list, as it is not needed to traverse the list at all to extend the context. The proposed syntax is used in the examples, along with an explanation to what it should translate to.

[2] the prototype temporaraly uses the different keyword, to not break existing behavior

2.3 Queues

Recall the two operations on a queue: `enqueue` and `dequeue`. The map operation on lists (as explained in Section 2.1)) can get away with a simple API for constructor contexts, as it returns the list as its original datastructure at the end. The rules used to map the list matches on the original list structure. The approach with the current constructor context operations would be slow for queues, as it would require traversing the queue for at least one of the operations.

Okasaki [12] lists a faster implementation with Catenable Double-Ended Queues, essentially having two lists to represent both the front and the back of the queue. With a performance of O(1) amortized time, it may seem like that is the limit of performance for functional languages. That solution is explained in more detail in Section 2.3.1. At the same time this does not seem optimal to represent a queue, given the imperative solution can use pointers. Could the same solution be produced in a functional style, by utilizing constructor contexts? This section will discuss this different approach to queues, that internally is like the imperative solution, written in a functional style.

2.3.1 Double-ended queues

It is functionally fast to add to the front of a list, as adding `Cons` is just a constant time operation. Therefore, it makes sense to use that operation for adding to the front of a list. Now if you also want to quickly add to the back of a list, you can keep a separate list for the end of the list, instead of traversing the first list. Essentially you have two lists, one to represent the front of the list and one to represent the back.

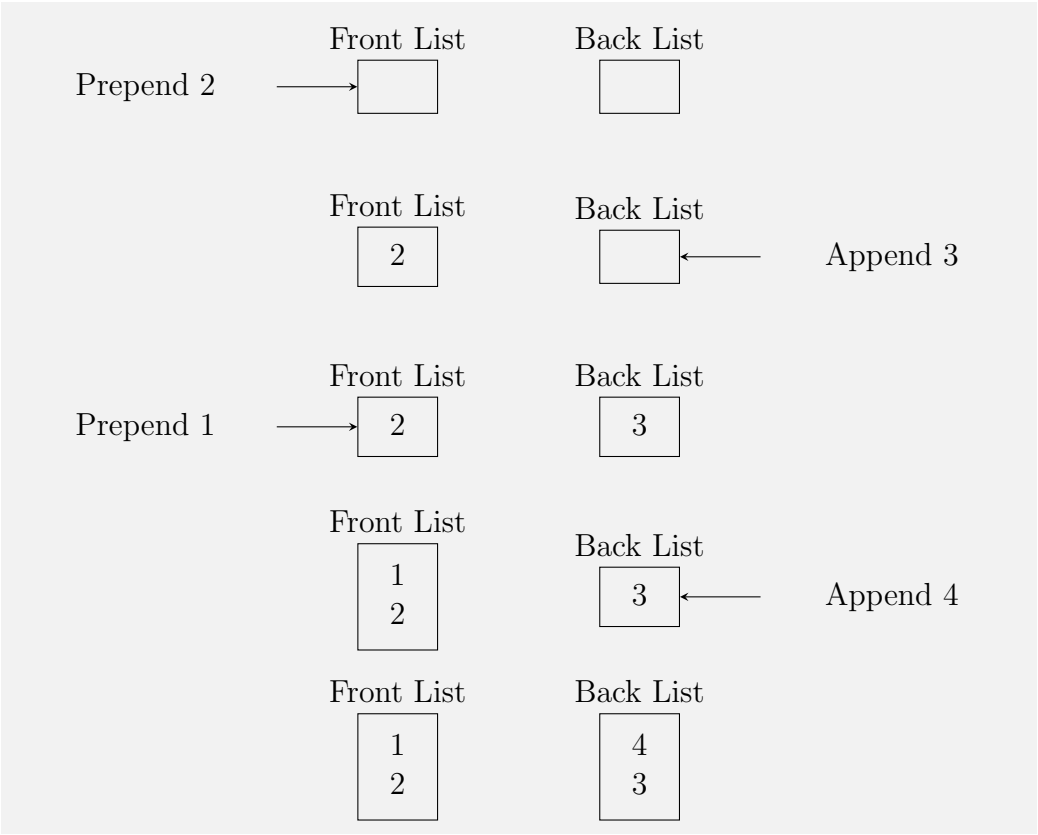

Figure 9: Adding elements to both ends of a list, one of the lists is reversed to represent the back of the full list, which is called the back list here

Figure 9 shows an example of a lists that starts empty, and has several elements added to it. Elements added to the front list are in order, while the elements added to the back list are in reverse. That means that `[1, 2]` in the front and `[4, 3]` results in `[1, 2, 3, 4]` when zipped back together.

You can move elements between the two lists, and still have the same result when zipped back together. For instance if `([1, 2], [4, 3])` and `([1], [4, 3, 2])` both represent the list `[1, 2, 3, 4]`.

The issue of this representation is that you have to rebalance, and can not always do the operation directly. If the list is `([], [4, 3, 2, 1])`, removing from the front list first requires rebalancing to `([1], [4, 3, 2])` first.

It is a clever technique, and it also allows to add to the back in O(1) time by adding the element to the second list. The next section will explore taking a different route: by not using two lists, but by using constructor contexts. They are suited as they can directly add elements to the back, but require the work of this thesis to implement `dequeue`.

2.3.2 Contexts functioning as queues

The naive functional implementation would use a cons list, which is slow because the operations require a modification to both the start and end of the list. But contexts can solve this problem, the `enqueue` just has to add an element to the back. Then the `dequeue` can happen from the start of the list, currently Koka does not allow to inspect the start of the list as a constructor context[3], that is not part of the limited API on constructor contexts.

Koka already understands this `enqueue` operation, and even recognizes that it is a `fip(1)` operation. To also make the `dequeue` compile, pattern matching on constructor contexts should be added to the Koka language. What will follow is a specification of how this pattern matching should work work with the queue operations as the concrete example.

The desired code is shown in Figure 10, which serves as the specification. Currently the compiler has no way to express these patterns, as there are several nuances that need to be expressed in the patterns. To start, there needs to be a blank pattern to express when the constructor context is just a hole. Further, for all other patterns that involve a constructor, the constructor argument with the continuation needs to be marked.

[3] without conversions to and from the regular type behind the constructor context

```
alias queue<a> = ctx<list<a>>

fip(1) fun enqueue(x : a, q : queue<a>) : queue<a>
  return q ++ ctx Cons(x, _)

fun dequeue(q : queue<a>) : maybe<(a, queue<a>)>
  match-ctx q
    // the context is just a hole
    ctx hole -> Nothing
    // the context has a Cons constructor
    // x is just a value,
    // xs is the rest of the queue
    Cons(x, ctx xs) -> Just((x, xs))
```

Figure 10: Enqueue and dequeue are both possible, the dequeue operation requires a top-down matching to eliminate the case that the context is empty

Patterns

The queue is specified to be a context of `list<a>`, and the type for the hole is `list<a>`. That means that there are two possible cases when you receive a `queue`: it has zero elements, `Nil`, or it has an element followed by another `queue`, `Cons`. The hole can not be in the first argument of the `Cons` constructor. Then the type of `queue` should have been `cctx<list<a>, a>`, as the hole would have been of the generic type `a`. These patterns can also be expressed in Koka.

```
type queuePatterns
  Hole
  Cons( value : a, xs : ctx<list<a>> )
```

Figure 11: The patterns on the context expressed as types

Figure 11 shows the types that will be exposed for the `match` construct on the context. The `ctx` keyword is used inside the `Cons`-pattern on the xs ar-

gument, to create a pattern analogous to `ctx Cons(x, _)` used earlier. The position of this keyword matters, other data types with multiple recursive constructor arguments have multiple patterns that need to be distinguished.

Specifying this language feature is one contribution that this thesis delivers. Chapter 3 provides the details of a proper definition. That definition is shaped by the examples in the following sections.

2.3.3 Visualizing the operations

To further demonstrate how this queue works, the operations will be shown with a small example in this section. Figure 12 shows the starting setup for a list with a single element `x`. For clarity, the internal pointers that the constructor context has are added in the visualizations. These are not exposed to the outside construct, the programmer can only use the built-in operations on the contexts. The element `x` is the last value in the queue, so the first to go out, and its location is the head of the list. The hole of the queue is where the next cons cell would be allocated, and its location is the end of the list.

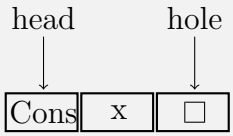

Figure 12: A queue as a context with a single element in it. The pointer to the hole is the extra internal pointer that the constructor context type brings

On this queue there are two operations performed in sequence; an `enqueue` of element `y` and a `dequeue`. Notice how the context changes after applying the operations. After each operation, the datastructure remains in a state with a hole, as a context. Figure 13 shows the `enqueue`, which creates a new cell with a hole and replaces the existing hole context with a pointer to the new cell. Figure 14 shows the `dequeue` operation, it frees the memory of element `x` and moves the head context pointer to the next element. The garbage collector can deallocate the memory as there are no references to the value anymore.

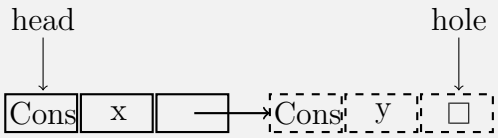

Figure 13: Enqueue element y

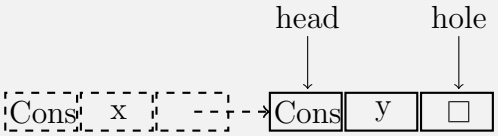

Figure 14: Dequeue element x

The `dequeue` should not assume the queue has a value. The return type of the `dequeue` operation is a `maybe`, to make the operation safe in case there is not value left in the queue. That is why it is implemented with a pattern match, against the various cases of the queue. This is one of the main motivations for adding pattern matches on constructor contexts, so that these cases can be distinguished on the context itself.

2.3.4 Currently unsupported in Koka

The pre-existing `(++)` and `(++.)` operators were not sufficient to implement the dequeue operation efficiently. While it is not recommended, it is technically possible to do a dequeue with the current operators Koka provides. As matching was not possible on the constructor context itself, the matching has to be done on the `list<a>` type itself. The way to get around it is to plug the context before matching. But the `dequeue` operation must still return the `queue<a>` type, not the `list<a>` type the the native implementation would return.

```
fun dequeue(q : queue<a>) : maybe<(int, queue<a>)>
  match (q ++. Nil)
    Nil -> Nothing
    Cons(v, n) -> Just((v, n))
     • inferred type: list<a>
     • expected type: queue<a>

fun re-ctx(l : list<a>) : queue<a>
  match l
    Nil -> ctx _
    Cons(v, n) -> (ctx Cons(v, _)) ++ (re-ctx(n))
```

Figure 15: Type mismatch for the return type, as this filling and then matching results in regular types and not the expected constructor context

The native implementation is shown in Figure 15, along with a `re-ctx` operator to ensure the correct type is returned. It can be applied to `n`, which would make the implementation work. But that is quite slow, every time a dequeue happens it would go in and out of a context, which is not how contexts are intended to be used. Besides, the match plugs the context with the hole constructor, which is quite a hack to make it work. That solution is as slow as the O(n) queues, the lists with the snoc operation.

Previously it was only theoretically conceivable to do, without plugging it first, and then converting it back to a context; the `re-ctx` operation should not be needed and is costly. But there was no way in Koka to express this, the solution was limited by the constructor context API in Koka.

2.3.5 Conclusion

There needs to be a way to inspect a constructor context, to re-context natively as part of the programming language. This is what this thesis prototypes with the pattern matching feature, which allows inspection besides operations focused on creation.

For queues it is possible to define the cases the hole can be in, and define the types of the destructured variables exposed in the patterns. The syntax should support selecting a variant, as the same constructor can have several positions the hole can be in. The addition of these pattern matches allows

defining an efficient `dequeue` implementation, similar to the fastest imperative solution. The `enqueue` operation can directly make use of the hole provided by the constructor context, and use the existing `(++)` operator to insert a new cons cell.

2.4 Trees

Consider the `match-ctx` construct proposed in Section 2.3. Like the example on queues, first the data type is defined. Along with that definition the patterns type for the `match-ctx` construct will been given.

The derivation meets the description of what Ralf Hinze et al. [13] call a *type-indexed data type*; a data type that is constructed in a generic way from an argument data type. They describe the procedure for defining a zipper[4] for an arbitrary data type [13, sec. 5], similar to what the pattern match construct should do over an arbitrary data type. It has an example on binary trees, which naturally gave the idea to do the same for constructor context matching. The *Context* they mention constructs the context parameterized by an original type; it is the same as the `ctx<>` type wrapper in Koka. They are aware of McBride [10] work that makes this derivation formal, which they mention they recently discovered.

[4] explained in Section 2.5, the next related works will be explained in depth later on

```
type tree
  Empty
  Node( left : tree, value : int, right : tree )

type treePatterns
  Hole
  NodeL( left  : ctx<tree>
       , value : int
       , right : tree)
  NodeR( left  : tree
       , value : int
       , right : ctx<tree>)
```

Figure 16: Zipper definition of a binary tree in Koka. Notice how there are two patterns for the Node constructor, and that the hole location information is encoded by marking as either NodeL or NodeR to distinguish these cases

2.4.1 Functional binary search trees

Lorenzen et al. [9] implemented red-black tree rebalancing. Constructor contexts are used to keep areas in the tree; the area that has already been locally sorted and the part which still requires to be rebalanced. Figure 16 shows the definition for binary search trees, which the red-black tree data type is based on. The additional information that is stored on a node is the color and the key, which allows it to behave as a red-black tree. By implementing the pattern matching, it can be researched if it can also help out with constructing efficient tree algorithms in a functional style.[5]

2.5 Zippers

The name zipper comes from the work of Huet [5]. It shows how a datastructure can be updated efficiently by reversing the way it is stored, such that it can always be "zipped" together to the original the datastructure. To clarify,

[5]the broader implications belong to future work, see Chapter 5

constructor contexts do not have this reversal, which is why a conversion is needed when directly translating from or to zippers.

While the datastructure is in this inside-out zipper form, there is a so called focus and context. Calculation is done on this focus, which is a nested part of the datastructure brought to the front. The data that was on top of the focus is stored in the context in reverse, which adjusts accordingly when moving around the nested structure. Moving around is efficient, typically a constant time operation, and works on immutable datastructures.

The moving around becomes more interesting when demonstrated for binary trees. The zipper can be constructed by taking the original tree, called the focus, with an empty list, called the contexts. At the start the focus is on the root node of the tree, and there are no values above it, so there is no further context to store. More interesting is going down the tree structure, which moves the focus to a subtree. For instance you can go left, which replaces the focus the left subtree of the original focus. The other parts of data, which are the value and right subtree of the original focus, can be stored in the list as a context.

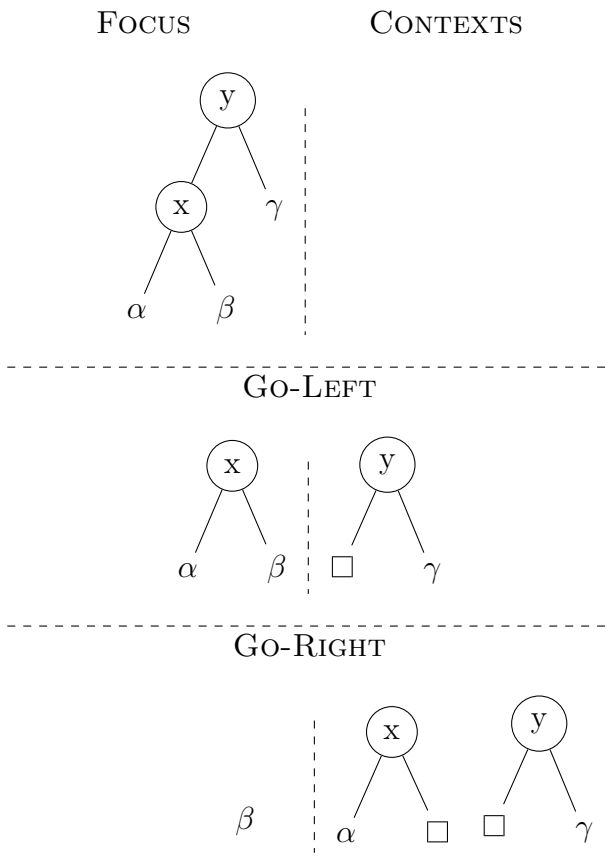

Figure 17: Moving around a zipper of a tree allows to focus in on the element β. At any point the original tree can be constructed by concatenating the focus with the contexts again

Figure 17 shows the start just described, which goes left to focus on the subtree with `x` as the root. The subtree with `y` as the root is stored in the list, as it surrounds the focus and should be kept in order to be able to reconstruct the entire datastructure.

After drilling down to the desired focus point, which is β in this case, the tree is smaller and the list is filled with two contexts. This small subtree is more efficient to do modifications on, and data is not lost as the entire tree can be reconstructed by applying the contexts from the list again. The last element that has been added to the list is applied first to surround the focus, until the contexts list is empty. For zippers, this operation is called going up. After going left and right, going up once results in the same zipper as only

going left from the start. After going left and right, going up twice results in the same zipper as the zipper from the start. This shows an approach were the focus is of type `tree`, and the contexts are a list, each element of type `ctx<tree>`.

Although Lorenzen et al. [9, sec 2.2] could have represented zippers as such, they did it slightly differently. They instead defined a type called `zipper` for binary trees, see Figure 18.

```
type zipper
  Done
  NodeL(up : zipper, key : key, right : tree )
  NodeR(left : tree, key : key, up : zipper )
```

Figure 18: Representing the tree zipper as a regular data type

It requires writing the rebuild function specifically for the `tree` type, when instead a list of context derives this procedure automatically from the data type. Those are exactly the same patterns that can be used with this new pattern matching construct, as seen in the previous section. A list of constructor contexts allows encoding the path taken by the position of the holes along a series of contexts. The downside of it is that it takes an extra allocation for the list constructors. The implementation of Lorenzen et al. [9] is able to be fully in-place by not making this allocation, and instead rely on the type itself to nest the multiple layers.

Perhaps this rebuild function could also be derived automatically, without allocating more memory like the list of contexts representation would do. They seem to be an alternative, which do not even require to have pattern matching for constructor contexts implemented, as the focus can just be the regular type and the rebuild is just a fold of `(++.)` over the contexts.

2.5.1 Conversions between zippers and constructor contexts

It is also possible to represent the whole data type as a constructor context, and convert it to a zipper. That does require pattern matching to be implemented, before it could only be done the other way around.

```
fun toZacc(tree: ctx<tree>, acc : zipper) : zipper
  match-ctx tree
    ctx hole -> Empty
    Node(ctx l, x, r) -> toZacc(l, (NodeL(acc, x, r)))
    Node(l, x, ctx r) -> toZacc(r, (NodeR(l, x, acc)))

fun toZ(tree : ctx<tree>) : zipper
  toZacc(tree, Empty)
```

Figure 19: Constructor context to zipper is a procedure that turns the structure inside out, which needs to know the location of the hole to encode it with either NodeL or NodeR

Figure 19 shows this conversion for a tree zipper. From a data type zipper, an accumulating constructor context can be used to take layers of the zipper and build a constructor context.

3 Defining match-ctx

The motivating examples seen so far sketched the usage of pattern matching, without the feature being formally defined yet. This chapter formalizes the ideas so far as a single and proper language feature for the Koka programming language. The `match-ctx` construct works on constructor contexts derived from regular types. Koka already does this derivation through the `ctx<>` and `cctx<>` type wrappers, which are based on the derivation by McBride [10]. The same derivation will be used for the patterns on constructor contexts. It will illustrate how pattern matching on constructor contexts over regular types works.

3.1 Recursive regular types

Contexts have the type `cctx<a, b>` in Koka. The type `a` is a recursive regular type, which are types following language T such that:

T ::= 0 | 1 | τ | T + T | T * T | I
τ ::= int | float | bool | ...

The most familiar types are the primitive types, which are characterized by τ, like ints, floats, booleans and so on. To form more complex types,

constructors are modelled as a choice, `+`, and if a constructor has multiple arguments the types are composed with `*`. These are also called algebraic data type. As type recursion is allowed, the `I` is used when a constructor argument is recursive. There are different ways to encode the recursive argument `I`. In the original derivation, McBride [10, sec 2.5] describes it as μ to indicate that it calculates a least fixed point. The example on natural numbers states the type as $nat := \mu x.1 + x$, which simply translates to $nat = 1 + nat$ or 1 + I in the language T. The initial implementation of the pattern matching will keep it simple, and only support direct recursion.

3.1.1 Typing

The baseline for the typing will be the typing rules for the pattern matching on regular non-constructor context types: the `match` rule. A pattern match is performed on a scrutinee, *expr* of type a, matched against the constructors of that same type with their respective constructor parameters $\overline{x}$.

For any type that is of type `cctx<a, b>`, the pattern matching will be handled by the `match-ctx` rule. So in this case it is known that scrutinee type is `cctx<a, b>`, and the type a is the type the constructor context will return once completed. The constructors have the index j for the index of the constructor context argument where the hole is. That index is used to make an exception on the typing of the constructor argument types, $\overline{x_i} : \tau_i \mid x_j : cctx\langle a, b\rangle$ will denote this type exception. All arguments have the usual types, as τ_i, except the argument matching the constructor context argument index x_j. For x_j, because it is the continuation, the typing is a constructor context with the same result type as the scrutinee: type a. This unwrapping of the scrutinee will be defined and explained in more detail in the next section. Because here the C_j's are not specified further, they will be formalized in the next section. It includes the notion that there can be multiple arms for a single constructor, each with a different position of the continuation across the constructor arguments.

$$\frac{\Gamma \vdash expr : a \quad \Gamma \vdash C_i : \tau_i \to a \quad \Gamma, \overline{x_i} : \tau_i \vdash e_i : \tau}{\Gamma \vdash \text{match } expr \text{ with } \overline{C_i\ \overline{x} \to e_i} : \tau}\text{MATCH}$$

$$\frac{\begin{array}{c}\Gamma \vdash expr : cctx\langle a, b\rangle \quad \tau_j = b \\ \Gamma \vdash C_i : \tau_i \to a \quad \Gamma, (\overline{x_i} : \tau_i | x_j : cctx\langle a, b\rangle) \vdash e_i : \tau\end{array}}{\Gamma \vdash \text{match-ctx } expr \text{ with } \overline{C_i\ \overline{x} \to e_i} : \tau}\text{MATCH-CTX}$$

Figure 20: Typing of the scrutinee and constructor arguments for matching

3.1.2 Patterns for all recursive arguments

With the type language defined, it is now the question which patterns are possible on the types. The pattern match will unwrap one layer of the recursive regular type, exposing a constructor with its arguments. A good place to start is recursive arguments, as each of them could be the place where a hole is located. At this time it is fine to ignore all other simple argument types, the only two other interesting cases are `+` and `*`, as they allow forming the more complex types.

Ralf Hinze et al. [13, sec 5.1] have a definition for *Ctx*, which in Koka corresponds to the alias `ctx<a>`. It is equal to `cctx<a,a>`, meaning that the type of the hole is the same as the result type itself. In the type language T, that is the case when the recusive type argument, `I` which they call *Id*, is used. They also restrict themselves to only recursive arguments. They also refer to the derivation of McBride [10, sec 3.1], but use a different syntax. There is common ground between those definitions, which will be established here as the function δ in the syntax McBride [10] used. The result 0 is used to denote that a type is empty, and cannot contain a hole when unwrapped with a pattern match, which are all the simple argument types said to be ignored. The result 1 is used to indicate an argument that could be context when unwrapped, which is the case for all recursive arguments.

The only piece of the puzzle left is how to define `+` and `*`, which has an intuition behind it. If given a type that is either `F` or `G`, you do not know which it is, so either one you get back is the context; so it translates to $\delta(F)$ or $(+)$ $\delta(G)$. If given a type that is a combination of the types `F` and `G`, the hole as to be in either one of them, as there can only be one hole across the entire type. You do not know inside which the hole resides, so it also has a

coproduct (+). It translates to F and (*) $\delta(G)$ if the hole resides in G, or (+) to $\delta(F)$ and (*) G if the hole resides in F. McBride [10] and Ralf Hinze et al. [13] define this derivation in the same way, in summary that gives the following common ground definition for $\delta(T)$:

To bring it back to what it means for Koka, is that this common definition defines the way to see the patterns on a constructor context, by enumerating the constructor argument positions that are recursive by the positions the hole could be in. This allows unwrapping a single constructor off a constructor context, as if is a regular type, except for the constructor argument that has the next layer constructor context.

$$\delta :: T \to Type$$

$$\begin{aligned}
\delta(0) &= 0 \\
\delta(1) &= 0 \\
\delta(\tau) &= 0 \\
\delta(F + G) &= \delta(F) + \delta(G) \\
\delta(F * G) &= F * \delta(G) + \delta(F) * G \\
\delta(I) &= 1
\end{aligned}$$

Coming back to natural numbers, `nat` is `zero + succ nat` which is equivalent to `1 + I` in the type language T.

$$\delta(1 + I) = \delta(1) + \delta(I) = 0 + 1 = 1$$

So that means that, beside the hole pattern, there is one other possible pattern corresponding to `succ`.

The next section will go over a small extension to the definition by McBride [10], which can also be used for the pattern matching on constructor contexts. It comes with a few caveats, but allows defining the hole type `b` as seen in the Koka type `cctx<a, b>` as a parameter, `x`, to the δ function as δ_x. This knowledge is sufficient to skip ahead to the examples, which start at Section 3.2.

3.1.3 Patterns for all type matching arguments

A recursive argument type can be seen as an specific instance of the hole type. It has the same type for both what is returned the context itself and what type needs to be provided to the context to fill it. To open up the possibility that these types are not the same, there need to be two additions. One, the δ function needs to know for which type the derivation is done, which is added as a subscript `x`. Two, the original type needs to be passed along, so that the original type is available for comparison on recursive arguments, which is the additional Y passed along. These additions make the δ function more generic, and allows the various other positions a hole could be in besides direct recursive arguments.

When the hole type is not known, all types defined by τ in the type language T are possible. McBride [10, sec 3.1] defined δ_x with the parameter `x`, which coincides with this type constaint. The hole type `b` can reduce the amount of patterns that are possible. Because a context carries the type of its hole, not all patterns across the constructor arguments will be needed to provide a total pattern match. Only the patterns with the hole at a constructor argument of the same type qualify, the other patterns can be discarded as they will never occur.

$$\delta_x :: Type \to T \to Type \to Type$$

$$\begin{aligned}
\delta_x(0, Y) &= 0 \\
\delta_x(1, Y) &= 0 \\
x \nsim \tau, \delta_x(\tau, Y) &= 0 \\
x \sim \tau, \delta_x(\tau, Y) &= 1 \\
\delta_x(F + G, Y) &= \delta_x(F, Y) + \delta_x(G, Y) \\
\delta_x(F * G, Y) &= F * \delta_x(G, Y) + \delta_x(F, Y) * G \\
x \nsim Y, \delta_x(I, Y) &= 0 \\
x \sim Y, \delta_x(I, Y) &= 1
\end{aligned}$$

The adjusted derivation is shown above. The original type that is passed along, Y, is used in the comparison to recursive constructor arguments denoted by I. In further examples of the δ function the Y is omitted, as it would be repetitive and verbose to constantly pass it along.

The programmer is given a choice to constrain the scrutinee, or not by using a free type variable as the scrutinee type; that would still make patterns for all possible hole locations. But if this type `b` is specified for the context passed to the `match-ctx` construct, the patterns will be reduced accordingly. For instance when `a` $\sim$ `b`, as is the case when the hole is a recursive argument, then only the recursive arguments make patterns to match against. To accommodate for constructor contexts where `a` $\nsim$ `b`, the derivation is parameterized by `b`, henceforward `x`, to find all valid patterns on `a`.

Any type that is not equal to the parameter `x` will result in a 0 as the result; not generating patterns. So if the `x` is a free variable and not constrained, it will make all patterns for all constructor arguments.

3.1.4 Comparison of types

The previous section briefly mentioned `a` $\sim$ `b`, but what exactly is this type comparison operation? It can for instance be syntactic equality or unifcation. It is expected that unification has much more utility in detecting the correct types, potentially eliminating more impossible patterns. But unification does add a significant complexity to the procedure.

Is unification needed to implement the core pattern matching on constructor context? This thesis will assume syntactic equality of types, and leaves it open which technique is used to compare types. Throughout the rest of this thesis, this comparison of types will be interchangably refered to as unification. The core implementation presented in this thesis will focus on an implementation that works based on syntactic equality.

3.2 Patterns on lists

As a simple example to start with, this section elaborates on the patterns for lists, with the same type definition as seen in the introduction. The list type can be defined as `1 + (`τ `* I)`, as it is `Nil` (1) or (+) `Cons` with a value (τ) and (*) recursively a list (I). The τ is set to integers, so `int`, but the example would also work with generic lists with a type parameter a.

How many patterns are possible? That depends on which type context it is asked. There are three possible cases; no constraint, constrained to `int` and constrained to `list<int>`. Constructor names are added in subscript to distinguish the different constructors when calculating $\delta_x(F + G)$.

$$\delta_x(list\langle int\rangle) :: T$$

$$\delta_x(() + (int * list\langle int\rangle)) =$$
$$\delta_{x_{nil}}(()) + \delta_{x_{cons}}(int * list\langle int\rangle) =$$
$$\delta_{x_{nil}}(()) + int * \delta_{x_{cons}}(list\langle int\rangle) + \delta_{x_{cons}}(int) * list\langle int\rangle$$

With the original constructor names added, it shows that this derivation yields one pattern for `Nil` and two patterns for `Cons`. Now depending on what is filled in for `x`, there will be a different number of patterns that are valid. For instance if you fill in `x ~ int`, only $\delta_{x_{cons}}(int) * list\langle int\rangle$ becomes a valid pattern as that δ_x has type `int` inside of it, as $\delta_x(x) = 1$ will hold. As said before, leaving `x` unconstrained will allows all the patterns, as the types will unify with any of them.

The types that still have a δ around them after this process become the holes in a specific pattern. The constructor that tags along with this δ function is the constructor that becomes available. The hole will have `ctx` keyword in front of it, to indicate it is of type `ctx<..>`. Figure 21 shows how this information is exposed to the programmer.

```
match-ctx list // list : cctx<list<int>, b>
  ctx hole -> // : ()
  Cons(x, ctx xs) -> // x : int, xs : ctx<list<int>>
  Cons(ctx x, xs) -> // x : ctx<int>, xs : list<int>
```

Figure 21: Patterns for integer lists, all types are possible for the hole as the b type is not constrained further. Having the context on x is quite useless, practically speaking, but that is not the point here

3.3 Dealing with units

The `Nil` constructor is a special case, as it has no constructor arguments. Up to this point it has been handled like it can be constructed as a context. However, constructor contexts in Koka strictly require each context to have

a hole, and not have zero holes. It is therefore impossible to encounter this pattern within a constructor context as the constructor pattern itself. The `Nil` constructor can occur within the matched `Cons` constructor, but only in one of the non-hole positions of another constructor that has arguments.

These patterns on zero argument constructors can be ommitted entirely. The solution used in the examples adds an extra empty pattern `ctx hole` for each pattern match on a constructor context.

3.4 Non-recursive types

The current version of Koka does permit constructing a constructor context with a non-recursive types as the hole, as long it can resolve its type. If you use polymorphism, it should be clear what the concrete type eventually is. The type itself is allowed as an argument; for example with `tree<a>` the type `cctx<tree<a>,a>` is allowed. There is one catch, and that is when you provide an empty context. As it is empty, it needs a hole of the same type, so `ctx<a>`, as it needs to be able to construct an `a` when filled in. Figure 22 shows these two points in one example.

```
fun require_int(input : cctx<tree, int>) : int
  match (input ++. 0)
    TEmpty -> 0
    TNode(_, _, _) -> 1

// not allowed
require_int(ctx _)
 • type error: types do not match
     context       :   require_int(ctx _)
     term          :                 ctx _
     inferred type: ctx<contexts/tree>
     expected type: cctx<contexts/tree,int>
// allowed, returns 1
require_int(ctx TNode(TEmpty, _, TEmpty))
```

Figure 22: Errors on calling the function

3.5 Non-inhabited types

For functions it is possible to require an argument that is not constructable as a value. That can occur when the hole type is not an constructor argument of any of the constructors of the return type.

For the tree with int nodes, you can instead make an argument with the type `cctx<tree, bool>`. As long as you do not call the function, it will compile. And you can not call this function, as you would have to provide this impossible type.

3.6 Patterns on binary trees

The same can be done for binary search trees, as they can also be defined in the type language T. The tree type can be defined as `1 + (I * τ * I)`, as it is `Empty` (1) or (+) `Node` with recursively a tree (I) and (*) a value (τ) and (*) recursively a tree (I). To make this example more interesting, the tree type will remain polymorphic with type variable a.

$$\delta_x(tree\langle a\rangle) :: T$$

$$\delta_x(() + (tree\langle a\rangle * a * tree\langle a\rangle))$$

$$\begin{aligned} = \quad & \delta_{x_{empty}}(()) \\ + \; & \delta_{x_{node}}(tree\langle a\rangle * a * tree\langle a\rangle) \end{aligned}$$

$$\begin{aligned} = \quad & \delta_{x_{empty}}(()) \\ + \; & tree\langle a\rangle * \delta_{x_{node}}(a * tree\langle a\rangle) \\ + \; & \delta_{x_{node}}(tree\langle a\rangle) * a * tree\langle a\rangle \end{aligned}$$

$$\begin{aligned} = \quad & \delta_{x_{empty}}(()) \\ + \; & tree\langle a\rangle * a * \delta_{x_{node}}(tree\langle a\rangle) \\ + \; & tree\langle a\rangle * \delta_{x_{node}}(a) * tree\langle a\rangle \\ + \; & \delta_{x_{node}}(tree\langle a\rangle) * a * tree\langle a\rangle \end{aligned}$$

As can be seen, the patterns for a tree context can also be derived with the same procedure. The tree type has two constructor arguments of type `tree<a>` for the `Node` constructor, which means that there will be two patterns when `x ~ tree<a>` for the δ_x derivation. Like before, it is clear what the types are of all of the variables that the pattern matching will expose. Figure 23 shows how this information is exposed to the programmer.

```
match-ctx tree // tree : cctx<tree<a>, b>
  ctx hole -> // : ()
  Node(l, v, ctx r) -> // tree<a>, a, ctx<tree<a>>
  Node(l, ctx v, r) -> // tree<a>, ctx<a>, tree<a>
  Node(ctx l, v, r) -> // ctx<tree<a>>, a, tree<a>
```

Figure 23: Patterns for trees, as expected there are three patterns for the node constructor as it has three arguments. Each could have the hole in it, and the types change accordingly

It seems like this type language is sufficient, but it is still quite primitive. The next sections consider possible extensions to the type language T, each with examples to motivate why more complex types can be beneficial. The type language up to this point is considered the core typing language for the pattern matching feature.

3.7 Compositional type parameters

An additional operator that is worth mentioning is the composition $(\cdot)$ operator. It can be used to define more complex types, that can enforce constraints via types. The common example for binary trees is the constraint that the tree is perfect. A perfect binary tree is a binary tree with all leaf nodes at the same depth, and all internal nodes have degree 2, according to Yuming Zou and Paul E. Black [18]. It is not to be confused with a complete binary tree, where the deepest layer is not necessarily entirely filled. As a sidenote, the property will no longer hold when a constructor context is made out of it. The goal of this section is to illustrate the shortcomings of the current Koka compiler for these types with compositional type parameters.

The constructors are slightly different in Figure 24, because the `Node` constructor now has the composite type `(a,a)`.

```
type tree<a>
  Empty
  Node( value : a, children : tree<(a,a)> )
```

Figure 24: The binary tree is constained via the type placed on children, Each time children is provided a Node instead of Empty, the type forces the field value to double the amount values it requires: a; (a,a); ((a,a),(a,a)); ...

It ensures that each new layer of nodes provides the full layer of the tree. For example, it is possible to construct `Node(1, Empty)`. The next layer will be `Node(1, Node(??, Empty))`. The type of this new value is `(a, a)`, so it would be valid to fill in `(2, 3)`.

The pre-existing implementation of Koka understands these types, but reports type errors for constructor contexts in a number of instances. The base case works just fine, as it is just a hole for the type `a`. Issues arrise when the hole is placed on one of the value arguments of `Node`. Even though the type is clearly `(a, a)`, it is currently not allowed to place a hole there. It is even not allowed to place a hole of type `a` within this tuple. But as seen before, it is allowed to place the hole at the very start, when it simply requires a non-composite `a`. And it is also allowed to place the hole at the very end, which Koka can correctly identify the composite type of. Figure 25 shows these instances.

```
ctx Node(_, Node((2, 3), Empty))
// :type cctx<tree<int>, int>

ctx Node(1, Node(_, Empty))
 • type error: the hole in a constructor context
   cannot be a value type

ctx Node(1, Node((2, _), Empty))
 • type error: ill-formed context

ctx Node(1, Node((2, 3), _))
// :type cctx<tree<int>, tree<((int, int), (int, int))>>
```

Figure 25: Unable to form contexts on composed type holes

So now we have also seen some cases not supported by the current Koka compiler, which provide future work as supporting these cases was not crucial for this thesis. There seems to be enough type information to construct contexts in all four cases, but would probably still not be trivial to implement.

3.8 Mutually recursive types

In Koka, mutual recursion is allowed, as can be seen by the example in Figure 26. Consider the `tree` type definition. It should see two patterns on the `Node` constructor; one for the `value` argument and one for the `trees` argument. So far so good, what happens with the patterns when the type of the hole is specified?

```
type tree
  Node(value : int, trees : forest)

type forest
  Forest(trees : list<tree>)
```

Figure 26: Mutual recursion in Koka, it is mutual because tree has a forest, which is goes back to tree by the list of trees forest contains

Figure 27 shows how you can start with a `forest` and end up with two `trees` nested in them. If this structure were to become a constructor context, with a hole of type `tree`, it would be expected that there are two valid hole positions in the last example; one for each `Node` constructor.

```
Forest([])
Forest([Node(0, Forest([]))])
Forest([Node(0, Forest([Node(1, Forest([]))]))])
```

Figure 27: Forming mutually recursive trees

Koka can already form the correct constructor contexts for these mutually recursive types. Figure 28 shows this cycle through the contexts that are possible. As Koka also identifies the correct hole types on these mutually recursive types, pattern matching on them should also be possible.

```
ctx Forest(_)
// :type cctx<forest,list<tree>>
ctx Forest([_])
// :type cctx<forest,tree>
ctx Forest([Node(_, Forest([]))])
// :type cctx<forest,int>
ctx Forest([Node(0, Forest(_))])
// :type cctx<forest,list<tree>>
ctx Forest([Node(0, Forest([_]))])
// :type cctx<forest,tree>
```

Figure 28: Able to form contexts on mutually recursive types

Are there any adjustments needed to support these types? It does not seem to be the case, Koka recognizes the hole types correctly. Where it might be a challenge is the implementation of the fully in-place analysis, to detect if memory can be re-used if it is of another type.

4

Implementing the specification in Koka

This chapter takes on the practical side of this thesis, by diving into the pre-existing Koka compiler and making the necessary changes to implement the core specification. The existing compiler was forked on Koka version 3.1.2, and built with the provided instructions.[6] The testing suite that was present was used to validate that the new implementation did not break existing language constructs. To practically introduce the context matching to Koka, the matching was introduced under a new `match-ctx` keyword, which may be merged with the existing `match` construct in the future. Further, as this paper is a proof of concept and experimental, the existing code was copied over for this construct to allow freely editing it, without the need to immediately merge it into coherent code that also implements the `match` construct at the same time.

The following sections will go over the implemented changes in the various modules in the compiler, and explain how the specification translates to changes. To verify that translation, specific instances of the specification were made along the way, which have also been provided as runnable tests in the compiler test suite. Each time one of those instances failed, the required change to make it pass was made to the compiler, as is typically done with

[6] see http://koka-lang.org/

test-driven-development.

The better way, turned out, to be leave the type inference as is, and propose a method do desugar. This method has been layed out as a proposal so that it can be integrated into the Koka compiler.

4.1 Syntax

The `Syntax.Syntax` module defines the type for expressions in the Koka language with the `Expr` type. It has a variant for the match expression, called `Case`, which has been duplicated into a variant called `CaseCtx` for the case expression on constructor contexts. It is not needed to make a variant of the `Branch` type which the match statements contain, as both can have the same type, as will be explained in the changes to the Parser. It is just the pattern part of the individual branches, which is the first part, that needs to add the new syntax.

Looking at the specification, the constructors can now have the `ctx` keyword in front of one of their arguments in the patterns, which if present should interpret the entire constructor that is matched as a constructor context. So for simplicity, the syntax will mark such a constructor as the variant `PatConCtx` besides the existing `PatCon` for regular constructors. The variable that is marked with the `ctx` keyword should be treated differently, as that individual type should also be interpreted as a constructor context. So `PatVarCtx` is introduced as a variant of `PatVar`. As a sidenote, it seems optional to define `PatHole` for the wildcards inside a constructor context pattern, the existing `PatWild` can also be used for that.

4.2 Parser

The branch type consists of three distinct syntactical parts: the pattern itself, the guard, and the expression.

```
    match a
      Cons(a) | empty(a) -> 1 + 1
  -- ^^^^^^^                                  the pattern
  --           ^^^^^^^^                       the guard
  --                       ^^^^^              the expression
```

Figure 29: Syntax parts with a concrete example

In the `Syntax.Parse` module, the first two parts are clearly separated parser components. In later parts of the compiler there are also checks for the types the branches produce, as they have to fit with the general type the match expression produces. One aspect to note is that, as far as the parsing goes, the construct is the same, it is only the pattern part that should be interpreted differently if it is written for a constructor context. Therefore, it is enough to flag that part as different with a variant on the pattern for the parser, and leave the rest of the code in the syntax definition the same.

The use of this keyword is already somewhat restricted in the parser, to avoid applying the effects multiple times later on in the compiler. So for instance, it is not allow to stack the keywords multiple times on the same argument:

```
fun main(): int
  val a = ctx _

  matchctx a
    ctx ctx hole -> 0 // duplicate ctx keyword
```

Figure 30: Syntax restriction for the ctx keyword

There is also the mention of the variable `hole` here, which seem to come out of nowhere. It has to do with the way wildcards are implemented in the parser, that the underscore character `_` is used both for a wildcard and for denoting a constructor context hole. That was not an issue up to this point in the compiler, but with constructor context patterns they need to be

```
type listInt
  NilInt
  ConsInt(head : int, tail : listInt)

fun main() : int
  val a = ctx ConsInt(0, _)

  matchctx a
    ConsInt(x, ctx xs) -> x
```

Figure 31: Accessing values out of the top-most constructor

distiguished. So the underscore will keep denoting wildcards, as that is the most intuitive meaning of it.[7] Wildcards still work without a `ctx` keyword, to denote patterns of any type in general and not specific to constructor contexts; which is how the wildcard on the `match` construct works.[8]

The matching on constructors are easier for types that have no type parameters, which have the name concrete in their test cases. If the arguments of the matched constructor do not have the `ctx` keyword in front of them, they will be interpreted as regular types:

Compared regular constructors that can be defined as patterns for matching, there is one additional piece of data that is important. The argument index of where the `ctx` keyword was used can be inferred, as it is the index where a `PatVarCtx` is used. As there will be additional tests for how the entire pattern is formed, the parser is kept permissive and the additional checks are performed later. For convenience, the parser will also emit the constructor argument index of the variable that is marked with the `ctx` keyword.

4.3 Type inference

After the Koka program gets parsed, it will get type checked by the `Type.Infer` module. This section will explain why the current type inference can not han-

[7] the test cases for those are "test/pattern-ctx/match-cctx-empty-wildcard.kk" and "test/pattern-ctx/match-cctx-empty-hole.kk".

[8] see test case "test/pattern-ctx/match-cctx-empty-any-wildcard.kk"

dle the proper type assignments. The adjustment in this section is just one method of implementing the specification. Section 4.5 has an alternative implementation that allows leaving the type inference intact. That implementation is the core proposal of this thesis, to be implemented for the Koka the compiler.

As seen in the typing rules of Section 3.1.1, the `match-ctx` rule requires different typing based on the argument index. All constructor arguments get their regular type assigned, except for the argument with the `ctx` keyword infront.[9] The type information of that argument is read and upgraded to a constructor context type, before it is passed along to the regular type inference logic. The field is the type for the hole part, and the overall type is the type the constructor belongs to.

There is a catch to this adjustment, as the regular type inference logic matches the procedure to destructure a regular constructor. Although the exception of the typing rules can be put into the type inference, the code itself is still not handled. This adjusted type inference expects the backend to destructure the constructor context according to the assigned types. Without this consideration the compiler will error on incompatible types for the arguments, as it would not destructure properly.

There needs to be a reduction to core code that the compiler understands. The type inference can still check proper typing of the used arguments before this embedding. This ensures that the expressions can be bound with the confidence of being well typed.

```
matchctx list
  ConsInt(x, ctx xs)
  // 1. type inference:
  //   x : Int
  //   xs : cctx<ListInt, ListInt>
  // 2. embedding into core
  ConsInt(x, xs) | @ctx-index(1)
```

Figure 32: Typing and embedding a pattern match on integer lists as constructor contexts

[9] this can be done by changing the inferpatterns function of this module, or via the desugar method explained later on in Section 4.5

After embedding the context with the guard for the index, it can be treated regularly again. It is important that after the type inference, the entire expression is still treated as a constructor context. The type inference merely takes a look into the underlying constructor.

While developing the feature however, it was useful to pass along the regular constructor to the backend of the compiler, as it already understood how to deal with it. When the type checks were properly implemented, the whole constructor context was passed along to the backend. The current Koka backend errors does not know how to handle this further, which is what the next section is about.

4.4 Backend

If you try to pass a contructor context to the C from core backend, it errors stating it can not generate the next pattern.

The main part to this existing code is the generation of the code that does the checking of the match in C. It generates an if-statement with a condition of all tests, combined by conjunction. Constructor patterns can have multiple tests, it is not only the variant of the constructor that is included in the test. If certain fields are set to certain values, like `Cons(10, Nil)` instead of just `Cons(x, xs)`, there will also be tests generated for `x = 10` and `xs = Nil`.

For the passed down constructor context that is being matched on, the typing information of the individual fields is known. The effect of the context field index has already been applied to these types during the type inference. So as long as the types match, the pattern itself is also a match.

So now after discussing what the current Koka implementation does, and determining that the needed information is available, the remaining changes to implement the specification will be discussed.

Implementation

The backend receives a well-typed pattern match on a constructor context. The pattern match follows the context path of the constructor context when the match is made on a recursive argument, allowing the programmer to traverse the entire constructor context via this pattern matching. The pattern matching is a view operation introduced to the constructor context API, and

this section will discuss the primitives that are needed to make this construct work.

For the implementation of the backend, there need to be a few internal helpers that can be called. Those helpers do the heavy lifting, and are used in the generated code to do the matching on contexts.

- @Cctx
- cctx/field-is-ctx
- cctx/field-is-hole
- cctx/unsafe-cast-ctx

`@Cctx` is a new primitive that allows access into the underlying constructor of a constructor context, when used as a pattern. In that way, it can expose the fields to pattern variables, and provide the holepointer for the other new predicates that will be discussed shortly. So `@Cctx(parent as Cons(x, xs), holeptr)` would be a valid pattern that matches any constructor context with the `Cons` constructor.

There are two more parts to the implementation, which is where the other three new primitives come in. First there is determining what the fields are, two new guards are introduced: `field-is-ctx` and `field-is-hole`. The last addition is the `unsafe-cast-ctx`, which allows creating the new context for the variable that is the continuation context. It can only be used after the proper guards are in place, hence why it has unsafe in the name.

4.5 From prototype to feature

This adjustment of both the type inference and the backend is quite intrusive, error prone, and makes keeping the compiler sound more complicated. For the prototype of this thesis, the parts of the Koka compiler that needed adjustment were copied to not break existing behavior. This way of working ensured that the prototype could be made, but raised code maintanance concerns that needed to be addressed.

This section addresses how the thesis went from a prototype implementation to a compiler proposal to be implemented in the Koka compiler. This proposal boils down to a specific set of desugar operations, syntax and parser extensions and new primitives. It will also explain how it relates to more intricate code examples, like nested pattern matching.

Desugar method

Now that a prototype has been layed out and new primitives have been introduced, it is time to piece them together to create the language feature that adheres to the specification. For that, this section will show to what expressions the newly proposed pattern matching syntax should be desugared.

Executing pattern matching comes down to generating conditions to check the scrutinee against. The existing matching procedure in Koka can be formalized as generating a boolean condition.

So expressed as a function ϵ, it is of type $Pattern \to Bool$. For variables, it always holds so it just returns `True`. For constructors, it matches if the specific constructor matches, and the arguments also match. Lastly there are wildcards, which just always match, they are just like variables with no assignment later.

$$\epsilon :: Pattern \to Bool$$

$$\epsilon(x) \mapsto True$$

$$\epsilon(C(\overline{p})) \mapsto isCon(C) \wedge \epsilon(\overline{p})$$

$$\epsilon(\_) \mapsto True$$

$$\epsilon(ctx\ ...) \mapsto ???$$

Now there finally is sufficient work to answer what constructor contexts should translate to. Besides the checks for a regular constructor ($\epsilon(C(\overline{p}))$), there are additional checks that will now be listed.

Given the field index i of the constructor context and the constructor context itself, one additional condition checks the position of the next context on the context path. So for $\epsilon_i(c_{ctx})$, that means adding $field\_is\_ctx(c_{ctx}[C(p_i)], i)$ to the guards.

For the pattern itself that has the constructor name, it should desugar to an expression with the `@Cctx` internal. The `@Cctx` constructor exposes the internal datastructure. Earlier there were plenty of examples on lists, here is how the pattern for `ConsInt(x, ctx hole)` can be expressed:

```
match xs
  @Cctx(parent as ConsInt(x, _hole), holeptr)
    | cctx/field-is-hole(parent,1.int32, holeptr) ->
    x
```

Figure 33: The type is accessed using @Cctx, and the hole should be on index 1, which is the second constructor argument of the ConsInt constructor

The `ConsInt` constructor is placed within the `@Cctx` internal. The check for the continuation is done with the new `field-is-hole` guard.

If instead the pattern was `ConsInt(x, ctx rest)`, the new `field-is-ctx` would be used. With the `ConsInt(x, ctx hole)` pattern, the hole variable is not assigned: `_hole`. So the final piece of the puzzle is assigning the correct value to `rest` in the `ConsInt(x, ctx rest)` pattern.

That rest variable needs to be rebound, as it should come from a cast of the original datastructure. The cast is performed on both the original matched structure, as well as the bound structure for the specific pattern. That can be done using a generated let statement, so that references to the variable *rest* come from this cast.

```
match xs
  @Cctx(parent as ConsInt(x, _valrest), holeptr)
    | cctx/field-is-ctx(parent,1.int32)
    val rest = cctx/unsafe-cast-ctx(xs,valrest)
    // value rest can be used here
    // here comes the rest of the provided user code
```

Figure 34: The index is the same as in the previous example, but now the value is used so the structure must be cast to allow access to the rest value for the inner code

4.6 Nested pattern matching

The translation scheme from the previous section is doable, but be cautious to do one step at a time. The user might write a pattern like this, which involves multiple layers of constructors inside a pattern:

```
match
  ConsInt(_, ConsInt(_, xxs)) -> ...
```

Figure 35: Multiple layers means a constructor inside another constructor

Doing the translation at once, by combining these generated steps gets complicated, which is beyond the scope of this thesis. There are also constraints like nested patterns on certain values that are possible. As said earlier, those constraints should still be added, this translation scheme describes the additional checks besides further guards that the user may write. The additional checks by the user can also be concrete value equality on the fields, the user might write:

```
match
  ConsInt(1, ConsInt(_, xxs)) -> ...
  ConsInt(2, ConsInt(_, xxs)) -> ...
  ConsInt(_, ConsInt(_, xxs)) -> ...
```

Figure 36: There can be equality checks to numbers or wildcards, or both

This can initially just generate all the different cases. If there are nested patterns, classic methods like that of Augustsson [1] can be applied for efficiency.

4.7 Optimizations

Speaking of optimizations, there is another optimization that can be done on the generated code. To not overcomplicate it, the code should always generate the conditions for the positions.

After it, there can be an analysis to check how many hole positions are possible with the matched type. If it is only one, it can drop all the position check guards, as they will always be *True*.

4.8 Context operations

The following section will define the operations `(++)` and `(++.)` in terms of the new pattern matching on constructor context. This serves as a reference implementation, it is slow compared to the current implementation but should give identical results.

Joining contexts

Given two constructor contexts, add them back to back. The following pseudocode sketches such an implementation generically. With the pattern matching this thesis prototypes, such an function would need to be written for each type.

```
fun ++(arg1 : cctx<a, b>, arg2 : cctx<b, c>) : cctx<a, c>
  match-ctx arg1
    hole -> arg2
    _@C(...args, argNextCtx) ->
      ctx C(...args, argNextCtx ++ arg2)
    // argNextCtx is next on the context path of C
    // C is the constructor of arg1,
    // which is exposed by the matching
```

Figure 37: New reference implementation for the join operation

The entire first constructor context is traversed, and all constructor arguments should be maintained. The result is a reconstruction, replacing the hole at the end with the second given constructor context.

Filling a context

The function `(++.)` can be defined in the same way, albeit that the type of arg2 is just `b` and that it returns type `a`. It returns the type the first constructor context promises.

5 Future work

Chapter 2 already talked about the related work with regards to constructor contexts, but what are the broader techniques that have also been tried so far? How does this work relate to the vast amount of work done so far, when it comes to optimizing functional programs? The notable techniques and research directions will be highlighted in this chapter.

Relevant research is still being done by combining this persistent question with new insights. Selsam et al. [15] combined the question with the insights into dependent types, and how they can be used in a proofing setting for high-performance systems for automated reasoning. They implemented it for Lean v4[3].

As Koka is not based on dependent type theory, this new method can not be integrated in the Koka compiler. What is left for Koka are at least the optional features that are mentioned in the definition of this thesis, Chapter 3.

So that are features like polymorphism, mutual recursion, etc. This proposal also established a back-and-forward between zippers and constructor context representations. The broader implications of this was beyond the scope of this research.

Matching on constructor contexts also allows implementing the `(++)` and `(++.)` operators themselves. This is slower than the pointer operations it

is currently defined as, so it would not be practical. Still, it makes for an interesting case as a reference implementation.

5.1 Discussion

Pattern matching on constructor contexts makes sense for a language that already implements constructor contexts. For Koka specifically, it can work together with the memory re-use algorithm. Up to this point, the tone has been quite optimistic in favor of adding and using this feature.

That it is possible to implement efficient functional queues with this construct, shows that the core of purely functional programming is still evolving. Introducing pattern matching is fitting for Koka, and is a natural continuation of the API Koka has introduced by making constructor contexts first-class.

For actual algorithmic work, this work is deliberately limited to focus on the core implementation. Although the supported types are still quite simplistic, it does answer the research question if this method is beneficial for queues with an affirmitive yes. Polymorphic types were not a requirement for this implementation, which is why the implementation examples use types like `list<int>` instead of `list<a>`.

If the programming language does not support FIP features, then it is much harder to motivate pattern matching on constructor contexts. But it is still possible to apply elsewhere; Bagrel and Spiwack [2] have recently shown how the developments of Koka allow for advancements in formalizing destination passing under a pure functional calculus. Destination passing comes with other challenges, like the requirement for lifetimes. But their calculus seems to work well for tree traversals, which is a question that this thesis left open for other approaches.

While implementing the proposed feature in Koka, there was one annoyance that would make the pattern matching more verbose. The type system could not rule out the case where the matched on context was empty by type, as dependent types are not present in Koka. If this feature would be implemented in a programming language with dependent types, the context path length should be encoded in the user exposed type. That way, it is practically sound with the type system, and prevents users from writing unreachable code.

Koka has challenges due to its design to work with effects and reuse of

memory. Koka uses Perceus[14] for implementing reference counting with reuse. There are also new type and effect systems being made, like Affect by Van Rooij and Krebbers [16]. This work is seperate from other compiler techniques, like inlining, tail-call-optimization or other optimizations.

In the earlier research by Šefl [19], similar connections for related work were made. The comparison in that work involved zippers as a solution, which were formed in according to the derivation of McBride [10]. There were two tests done on the effectiveness of zippers, comparing them with the imperative counterpart. The difference between the test is whether or not the imperative counterpart needed to do allocation. The speculation is that such a comparison could yield better results with Koka as the language for the solution due to re-use. As the tests were on trees, constructor contexts would not help, unless future works would find a way to make them useful on trees.

5.2 Multiple holes

Throughout this work, the focus has been mainly on one-holed contexts. One-holed contexts have a strong use in allowing data structures to extend, and to finally be filled. Earlier work in compiler theory has also focused on many-holed contexts, which have a slightly different focus and application.

In terms of compiler theory, Lindley [7] has shown that a Hindley-Milner type system with only abstract types as an extension is sufficient to statically type many-holed contexts. So with minimal compiler support, it is possible embed structures with multiple holes, that can each be statically typechecked.

The usage of many-holed contexts is more focused on filling, with a static type check guarrantee, and that every hole occurs exactly once in the context. Lindley [7] focuses on XHTML contexts, and a construct called a formlet which has its own validation logic. Think of templates in web programming, they require multiple pieces of data to be filled in, and you want to know the types of the pieces of data to be filled in. For a more in depth dive into this topic, see the work of Yallop [17]. It is also beyond the scope of this thesis due to the vast advances of modern web frameworks and what tools they provide to the programmer.

This section will dig a bit deeper into just the theoretic comparison, to scratch the surface to see that many-holed contexts are tough to extend. The prefered way is to use zippers and the existing research, and this thesis com-

pletes the bi-directional translation between one-holed constructor contexts and zippers.

Difference

One-holed contexts allow travelling from the root of the datastucture to the hole. With multiple holes there is not a singular path, so extensionality would require specifying which hole to extend.

What is interesting however, is that the splitting process would still work. The surrounding context will just increase the amount of holes with one. It does not matter if it already has holes, it will just have one more at the position the subtree is cut-out. The subtree that is cut-out remains with the amount of holes it has. There is one subtle piece of information that also gets produced. The position to the parent tree of the sub-tree is now a piece of information that needs to be kept. Because that position needs to be known to the `(++)` operation to merge the two trees back together, as the parent tree can have multiple holes. Here is a small example on tree with two holes:

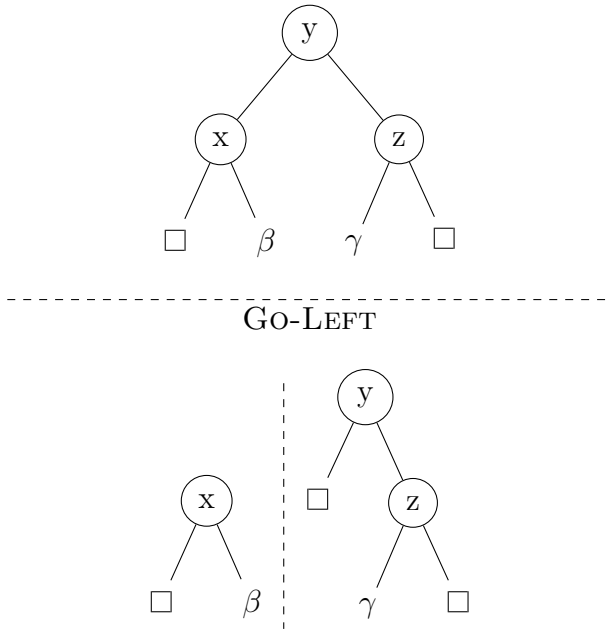

Figure 38: Navigating trees with multiple holes requires more care

It does not seem that useful, as it will just be a structure with arbitrary

holes. Even when you start from the root node `y` and deconstruct to both subtrees, with all subtrees only having one hole, there is no operation to look around from the end upwards. So that is why zippers are should be used.

6 Conclusion

The pattern matching on constructor contexts is an additional tool to express purely functional programs. The API Koka provided for constructor was limited (Section 1.6), pattern matching is a desired feature to have to further progress fully in-place programming (Chapter 2). Pattern matching on constructor contexts allows the conversion to zippers (Section 2.5), opening a world of possibilities in terms of algorithms that can be explored further (Chapter 5). The pattern matching also allows queues to be implemented as constructor contexts of lists, allowing O(1) operations natively in purely functional programming (Section 2.3). This pattern matching is defined for regular types, which has patterns for every constructor and every constructor argument the hole can be in (Chapter 3). The thesis made a prototype of this language feature in the Koka compiler, which describes how a set of compiler internals can be used to implement pattern matching (Chapter 4).

6.1 Limitations

It is quite plausible that there is no broader use, besides queues and zippers, for this pattern matching construct. Time and future research will tell if this addition to the Koka API is considered practical. This work is only a tiny

step, and has no solid answer to representing multiple holes as a meaningful language construct (Section 5.2).

The discussion (Section 5.1) also touches upon the many other ways compiler techniques that have been used instead, to produce efficient code. Those are more developed and likely preferable. But it is great that this work establishes O(1) queues and broadens the theoretic connection to the well-studied concept of zippers.

6.2 Ethics and Privacy

The Ethics and Privacy Quick Scan of the Utrecht University Research Institute of Information and Computing Sciences classified this research as low-risk with no fuller ethics review or privacy assessment required. The results of that assessment are also provided along with this thesis.

Bibliography

[1] Lennart Augustsson. 1985. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture* (Berlin, Heidelberg, 1985), Jean-Pierre Jouannaud (Ed.). Springer, 368–381. https://doi.org/10.1007/3-540-15975-4_48

[2] Thomas Bagrel and Arnaud Spiwack. 2025. Destination Calculus: A Linear Lambda-Calculus for Purely Functional Memory Writes. 9 (2025), 253–279. Issue OOPSLA1. https://doi.org/10.1145/3720423

[3] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25* (Cham, 2015), Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[4] Robert Hood and Robert Melville. 1981. Real-time queue operations in pure LISP. 13, 2 (1981), 50–54. https://doi.org/10.1016/0020-0190(81)90030-2

[5] Gérard Huet. 1997. The Zipper. 7, 5 (1997), 549–554. https://doi.org/10.1017/S0956796897002864

[6] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. https://doi.org/10.48550/arXiv.1406.2061 arXiv:1406.2061

[7] Sam Lindley. 2008. Many holes in hindley-milner. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML* (Victoria BC Canada, 2008-09-21). ACM, 59–68. https://doi.org/10.1145/1411304.1411313

[8] Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP²: Fully in-Place Functional Programming. 7 (2023), 275–304. Issue ICFP. https://doi.org/10.1145/3607840

[9] Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. 2024. The Functional Essence of Imperative Binary Search Trees. 8 (2024), 518–542. Issue PLDI. https://doi.org/10.1145/3656398

[10] Conor McBride. 2001. The Derivative of a Regular Type is its Type of One-Hole Contexts. (2001). http://strictlypositive.org/diff.pdf

[11] Yasuhiko Minamide. 1998. A functional representation of data structures with a hole. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1998-01-21) *(POPL '98)*. Association for Computing Machinery, 75–84. https://doi.org/10.1145/268946.268953

[12] Chris Okasaki. 1996. Purely Functional Data Structures. (1996).

[13] Ralf Hinze, Johan Jeuring, and Andres Löh. 2004. Type-indexed data types. https://www.cs.ox.ac.uk/people/ralf.hinze/publications/MPC02a.pdf

[14] Alex Reinking, Ningning Xie, Leonardo De Moura, and Daan Leijen. 2021. Perceus: garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual Canada, 2021-06-19). ACM, 96–111. https://doi.org/10.1145/3453483.3454032

[15] Daniel Selsam, Simon Hudon, and Leonardo de Moura. 2020. Sealing pointer-based optimizations behind pure functions. 4 (2020), 115:1–115:20. Issue ICFP. https://doi.org/10.1145/3408997

[16] Orpheas Van Rooij and Robbert Krebbers. 2025. Affect: An Affine Type and Effect System. 9 (2025), 126–154. Issue POPL. https://doi.org/10.1145/3704841

[17] Jeremy Yallop. 2010. Abstraction for web programming. The University of Edinburgh. https://era.ed.ac.uk/handle/1842/4683

[18] Yuming Zou and Paul E. Black. 1998. *Dictionary of Algorithms and Data Structures* (2019 ed.). NIST. `https://www.nist.gov/publications/dictionary-algorithms-and-data-structures`

[19] Vít Šefl. 2019. Performance Analysis of Zippers. `https://doi.org/10.48550/arXiv.1908.10926` arXiv:1908.10926 [cs]