Utrecht University

Faculty of Science
Computing Science
Master's Thesis

June 2022

Accelerating Sum Types

Optimizing the memory usage of sum data types
in parallel array languages

Supervisors:
dr. T.L. McDonell
dr. W.S. Swierstra

Student:
Rick van Hoef
5665299

Abstract

Parallel array languages are programming languages that make it possible to write highly parallel programs without knowing the intricacies of parallel programming and hardware. In this thesis, we present an optimization for the memory usage of parallel array languages, specifically geared towards sum types. Sum types are algebraic data types with more than one constructor, which are used to denote a choice between multiple different variants. Examples of applications of sum types in parallel programs are material choices in a ray tracer and parameters to a fluid simulation. We introduce a new representation of sum types that aligns with the struct-of-arrays memory layout that parallel array languages commonly use. We show that this representation is close to optimal with respect to memory usage, and that it is an improvement over existing representations that are suited for a struct-of-arrays memory layout. This representation is implemented in the *POSable* library, which generically converts non-recursive Haskell 98 data types to this representation. *Accelerate* is a functional parallel array language embedded in Haskell that supports sum types. The *POSable* library has been integrated in *Accelerate*, which shows the viability of the approach.

Acknowledgements

I want to thank Trevor McDonell and Wouter Swierstra for being there from the very conception to the end result of this thesis, answering my questions, and guiding me to this result; the Accelerate team for listening to my somewhat incoherent ramblings every week during the team meetings; Niels Kwadijk and Matthijs Ham for working next to me for months; the board of study association Sticky for providing me with coffee and *koenkies*; and all the kind people that have proofread (parts of) my thesis for their invaluable feedback.

Contents

1 Introduction and motivation **3**

2 Background **5**
2.1 Algebraic data types . . . 5
2.2 Accelerate . . . 5
2.3 SIMD and struct-of-arrays . . . 7
2.4 Representations of sum types . . . 8
2.5 Type level programming . . . 10
2.6 Data type generic programming . . . 12

3 Representing Sum Types Efficiently **15**
3.1 Recursive Tagged Union . . . 15
3.2 Theoretical efficiency . . . 16
3.3 Comparison . . . 16

4 The POSable library **18**
4.1 Goal and constraints . . . 18
4.2 Representing types . . . 18
4.2.1 Choices . . . 18
4.2.2 Fields . . . 19
4.2.3 The POSable class . . . 20
4.3 Mapping a simple type . . . 20
4.4 Mapping polymorphic types . . . 20
4.4.1 Choices . . . 21
4.4.2 Fields . . . 21
4.5 Generics . . . 22
4.6 Limitations . . . 23

5 Implementation in Accelerate **25**
5.1 Mapping to Elt . . . 25
5.2 Type hierarchy . . . 26
5.3 Pattern matching . . . 26
5.4 Changes to the AST . . . 27
5.5 Limitations . . . 28

6 Future work **29**
6.1 Implementation in Accelerate . . . 29
6.2 Optimizations . . . 29
6.3 Sorting types . . . 30
6.4 Fighting type erasure . . . 30

7 Conclusion **31**

8 References **32**

A Calculations: memory usage of different representations of sum types. **34**

1 Introduction and motivation

Parallel array languages make it possible to write highly parallel programs without knowing the intricacies of parallel programming and hardware. This makes it possible for developers to write high performance software with relative ease. Parallel array languages offer operations that are performed on an entire array of data, like *map*, *fold* or *stencil*. These operations are executed in parallel on a GPU or multicore CPU.

In this thesis, we present an optimization for the memory usage of parallel array languages. This has the potential to improve the performance of memory bound programs. The optimization we introduce is specific to sum types, algebraic data types with more than one constructor.

Sum types are common in functional programming languages like Haskell and ML. They represent a choice between multiple alternatives, each potentially of different types. A commonly used sum type is the `Maybe` type (also known as an `Option` type in other languages). It has two constructors, `Just` and `Nothing`. The former takes a single value, the latter takes no value. The `Maybe` type can, for example, be used as the result of a search function. The result `Just x` than signifies that a result has been found, and the result value is brought in scope as x, while `Nothing` signifies that no result has been found.

Functional parallel array languages have only recently gained support for sum types [11, 23]. A drawback of current implementations of sum types in parallel array languages is the memory usage. In this thesis, we propose a new, more efficient memory layout for sum types in parallel array languages. This optimization has been applied to Accelerate, a parallel array language embedded in Haskell.

Being a parallel array language, Accelerate is mainly concerned with arrays. Instead of applying a function to a single value, a typical Accelerate program applies a function to an array of values. Accelerate stores these arrays in a struct-of-arrays fashion. This means that for arrays of types with multiple fields, multiple arrays are allocated, each containing values of a single type, corresponding to the fields. For example, when a array of tuples is created by the programmer, this internally leads to the creation of two arrays, one with the first element of each tuple, and one with the second element of each tuple.

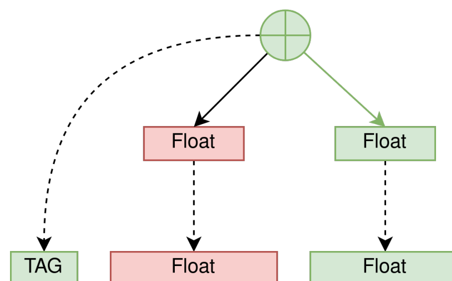

Figure 1: *Memory layout of a value of type Float + Float in Accelerate. Depicted in the top is the structure tree of type Float + Float. The sum is displayed as a circle with a plus symbol in it. Displayed in green is one of the possible values of this type, where the right constructor has been chosen. Depicted below are the allocated arrays and their types. Illustrated in green are the arrays that hold values at the index where the aforementioned value is stored. At each index only one of the arrays of Floats contains a value. The other array contains an undefined and unused value at this index.*

Arrays of sum types are represented similarly. For example, consider an array of `Either`s, another common sum type. When an array with values of type `Either` is created, the fields of the `Either`s are spread out over two arrays. The constructor choices are stored as an integer, called the tag, in a separate array. This means that the creation of an array of `Either Float Float` internally leads to the creation of 3 arrays, one with tags and two with floats. This is shown visually in figure 1.

This representation is not optimal. At each index of the arrays, either the second or the third array contains an undefined value that is never accessed, depending on the value of the tag. It

would be far more efficient to only create 2 internal arrays, one with tags, and one with floats. A visual example of this is shown in figure 2.

Another limitation of the current approach becomes visible when nesting sum types. In nested data-types, the TAG values are spread out over different arrays. To perform a nested pattern match on such a value, the hardware has to perform multiple memory accesses. It would be more efficient to instead have a single TAG describing the whole nested structure.

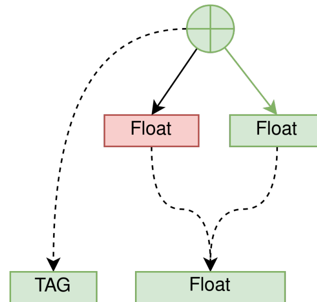

Figure 2: *Compact layout for a value of type Float + Float. Depicted in the top is the structure tree of type Float + Float. The sum is displayed as a circle with a plus symbol in it. Displayed in green is one of the possible values of this type, where the right constructor has been chosen. Depicted below are the allocated arrays and their types. Two arrays are allocated for this type, one of with tags, the other with Floats. Both arrays contain a value at each index, no matter the value that is represented at that index.*

In this thesis, we propose a memory layout for algebraic data types based on these concepts. We compare this layout to existing and proposed memory layouts in section 3, and conclude that our approach is close to optimal with respect to memory usage. This layout has been implemented in a library, which is described in section 4. This library has been integrated in the *Accelerate* compiler, as described in section 5. Finally, we discuss limitations of the library and the compiler integration, and directions for future research.

In summary, we answer the following research question:

> *How can we efficiently represent sum data types in memory, in the context of data-parallel array applications?*

Our contributions are the following:

- A new representation of sum data types in memory, that is close to optimal with respect to memory usage. This representation is specifically designed to align with a struct-of-arrays representation of data. The representation is presented in section 3.
- A library, *POSable*, which generically converts non-recursive Haskell 98 data types to this representation. This library is presented in section 4.
- An integration of this library in *Accelerate*. Although the integration is not yet complete, it proves that it is possible to use the library in an array parallel language. The integration is described is section 5.

2 Background

2.1 Algebraic data types

Algebraic data types are types that can be defined by the programmer, in contrast to machine types like `Int` and `Float`. These types have proven to be a powerful, convenient, and expressive way of creating and organizing data, and have become a defining characteristic of functional programming languages in particular. Algebraic data types can be divided in two categories: sums and products. Common examples of the former are `Either` and `Maybe`. In listing 1 the definition of `Either` is shown. It represents a choice between two different values, possibly of different types. In a sum type only one of the values is present at the same time. Sum types are denoted as $a + b$.

```
data Either l r = Left l
                | Right r
```

Listing 1: *The definition of the `Either` data type. The type has two type variables, l and r. `Either` has two variants, called constructors: `Left` and `Right`. Both constructors have a single type argument. The type of the `Left` constructor is `l -> Either l r`, and the type of the `Right` constructor is `r -> Either l r`.*

The other variety of an algebraic data type is a product type. The `Point` type shown in listing 2 is an example of a product type. In a product type the values of all fields are present at the same time. Product types are denoted as $a \times b$. Most programming languages offer good support for these types, in the form of tuples or structs.

```
data Point = Point Float Float
```

Listing 2: *The definition of the `Point` data type. The `Point` type, in contrast to the `Either` type, has only a single constructor (or variant). This constructor in turn has two values, both of type `Float`.*

A product type really is a specialization of a sum type, containing only one choice instead of multiple. Conversely, sum types can contain products in each variant. In conclusion, algebraic data types are types that are composed of other types. There are two ways of composing these types; products and sums. A sum represents a choice between multiple variants, while a product holds multiple fields at the same time.

2.2 Accelerate

Accelerate [5] is a parallel array language that is deeply embedded in Haskell. Accelerate provides the programmer with a set of parallel array functions, like *map*, *stencil* and *fold*. By optimizing these array functions for SIMD hardware, Accelerate achieves good parallel performance.

A deeply embedded language is a language that overloads features of the host language to produce an abstract syntax tree (AST) of the the embedded language instead of directly executing the code [13]. This AST can then be analyzed, optimized and compiled. This means that the embedded language can leverage the parser, lexer and type checking facilities of the host language. This in turn means programmers can use existing tooling, like IDE's and language servers, to aid in programming in the embedded language. The code of the embedded language can also be interleaved with host language code, offloading certain tasks to the embedded language.

In the case of Accelerate, this means that parallizable tasks can be offloaded to be executed on a parallel computing device (e.g. a GPU). This approach greatly reduces the amount of (boilerplate) code that has to be written for typical data-parallel programs, compared to C-based languages like OpenGL or CUDA. An example of a function written in Accelerate is shown in listing 3. Except for its type, this function is equivalent to the Haskell implementation of the same operation. Accelerate compiles this program to a GPU kernel that performs the `zipWith`

operation in parallel and `fold` operation as a parallel tree reduction. It then loads both `Vectors` into GPU memory and executes the kernel.

```
dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Listing 3: *Computing the dot product of two vectors in Accelerate. Excepting the type annotation, this is equivalent to the implementation of this function in plain Haskell. The type however makes that the Accelerate version of the overloaded functions and operators are selected by the Haskell compiler. Instead of directly executing the function, this code creates an Accelerate AST that can be fed to the `run` function from one of the backends. This backend then in turn executes the AST on the respective hardware.*

In the previous example the Accelerate `fold` function is used. Its type is given in listing 4. The function takes a function on `Exps`, a default value `Exp a`, and an Accelerate `Array`. It returns another `Array` of one dimension less. The difference between `Exp` and `Acc` shows that there are two levels of data in Accelerate. The outer level is that of `Arrays`, n-dimensional lists. Functions that operate on these live in the `Acc` AST. The `fold`, `zipWith` and `map` are examples of this. The inner level is that of the `Exp` AST, which contains scalar expressions and values. Arithmetic functions list `+` and `*` are examples of this. No recursion is allowed in `Exp` expressions. Types have to implement the `Elt` class to be admissable in `Exp`. This class constrains types to be non-recursive and composed of machine types that Accelerate supports. Instead, recursive operations or loops have to be performed on the outer level.

```
fold :: (Shape sh, Elt a)
     => (Exp a -> Exp a -> Exp a)
     -> Exp a
     -> Acc (Array (sh :. Int) a)
     -> Acc (Array sh a)
```

Listing 4: *The type of the Accelerate `fold` function. As most array functions in Accelerate, it is shape-polymorphic. That is, it works on vectors, matrices, and arrays of higher dimensions. The function takes an array of some dimensionality, applies a function on the inner list, and returns an array of one dimension less.*

Accelerate supports differently sized floating point and integer types as scalar values, together with tuples and vectors of these. It also supports typical Haskell sum types [23], such as `Maybe` and `Either`. User-defined sum and product types are supported too, as long they are non-recursive.

The supported scalar values in Accelerate are captured in the `Elt` class. This class can be automatically derived for non-recursive Haskell 98 types as shown in listing 5.

```
data Option a = None
              | Some a
              deriving (Generic, Elt)
```

Listing 5: *Deriving the Accelerate `Elt` class for a user-defined sum data type. `Generic` has to be derived to make this possible, as the default implementation of the `Elt` class relies on the `Generic` class.*

Listing 6 shows a Accelerate function using sum data types. In this example, the Haskell `find` function is implemented in Accelerate. This function takes a predicate function and return the first element that matches the predicate. The `Maybe` sum type is used here to discern matching from non-matching values. By implementing this function as a parallel map and fold, it is much faster than the equivalent in plain Haskell for large arrays. Real-life examples of sum data types used in Accelerate code are common Haskell sum types like Either and Maybe, but also user-defined sum data types to represent parameters to fluid simulations [9] and ray tracers [8].

```
find :: (Elt a, Elt b)
     => (Exp a -> Exp (Maybe b))
     -> Acc (Vector a)
     -> Acc (Scalar (Maybe b))
find f xs = fold firstJust Nothing_ $ map f xs
```

Listing 6: *The implementation of the Haskell `find` function in Accelerate. The function maps a predicate over an array, and then finds the first matching element with a fold. The implementation of the `firstJust` function is not shown here, but follows directly from its definition is Haskell.*

Accelerate has been extended with a Foreign Function Interface [6], chunking operators [18] and kernel fusion [22]. The compiler targets an intermediate language; LLVM [21], which in turn targets multiple backends, including GPU's and multicore CPU's.

2.3 SIMD and struct-of-arrays

Accelerate is designed to be run on SIMD hardware. SIMD, or Single Instruction, Multiple Data, is a computer chip architecture in which multiple compute units load their own data, but perform the same instruction on this data. Modern CPU's support SIMD operations with the SSE and AVX instruction sets and can perform a single instruction on up to 16 *32-bit* numbers at the same time [14]. Modern GPU's however can perform a single operation on thousands of numbers at the same time. For CPU SIMD instructions, it is required that the input data is stored in a contiguous area of memory. In GPU's this is not a requirement, but to get maximum performance, this is still preferred.

Accelerate, and similar parallel array languages, store data in a struct-of-arrays (SoA) representation [5, 11]. An arrays-of-structs (AoS) representation is more common in other languages. Storing a list of records in a AoS representation means storing each record back-to-back in memory. Conversely, storing a list of records in a SoA representation means storing the first field of each record in a contiguous area of memory, the second field of each record in another contiguous area of memory, et cetera.

As an example, consider an array of `Point`s, as defined in section 2.1. In the AoS layout, the x and y coordinate of each point are stored together. In the SoA layout however, all the x coordinates are stored together, and all y coordinates are stored together. Listing 7 and listing 8 show the definition of this array of `Point`s in the AoS and SoA layouts respectively, as defined in C.

```
struct {
  float x, y;
} AoS[N];
```

Listing 7: *A list of N points in the AoS representation, defined in C. In memory, x values and y values are alternated.*

```
struct {
  float x[N];
  float y[N];
} SoA;
```

Listing 8: *A list of N points in the SoA representation, defined in C. In memory, all x values are stored together and all y values are stored together.*

A SoA representation generally leads to better performance in a SIMD environment. For example, assume that a program performs some operation only on the x element of each point. When the first x is loaded by an instruction, x and the values directly next to it are stored in the

cache. The amount of values that are stored in cache depend on the size of a cache line, which is typically 64 or 128 bytes wide. In a AoS representation, each cache line then contains both xs and ys. In a SoA representation however, each cache line is filled with just xs, and twice the amount of data can stay cached. This makes memory access cheaper, leading to better performance.

2.4 Representations of sum types

The previous section discusses how *product* types such as `Point` can be represented in memory. In this section the represention of *sum* data-types is discussed.

Functional languages, like Haskell and ML, have traditionally had support for sum types. Those are generally stored as a tag and a pointer, the former representing the constructor choice, while the latter points to the fields of this constructor. In a lazily evaluated language, using pointers is necessary, as the value might still have to be evaluated. Instead a thunk, an unevaluated function, might reside at the pointer location.

In GHC, the tag is present in the info table (a heap object containing information about values and thunks). Heap objects have a header that points into this info table. For types with a small amount of constructors the tag is also stored in the unused bits of the pointer [20]. This is not a memory optimization (the tag is still present in the info table), but rather a performance optimization.

Support for unboxed sum types was added to GHC (Glasgow Haskell Compiler) in version 8.2.1 [24]. Unboxed types are types that are represented in memory without the use of a pointer [15]. As a consequence, unboxed types can not be undefined nor point to a thunk, which means that functions that produce these types are evaluated strictly. Unboxed types are primarily used to prevent pointer-chasing in Haskell, which optimizes application run time. This also means the values are allocated on the stack rather than the heap. Unboxed sum types are more space-efficient than tagged pointers, but the existing implementation in GHC does not pack alternatives.

Futhark [11], a functional parallel array programming language in the ML family, supports sum types [28]. Like Accelerate, it uses a struct-of-arrays representation, and spreads out the fields of sum types over multiple arrays. Support for packing alternatives is not yet present, although an approach has been proposed [10]. The proposed idea is to sort the fields of the different constructors by type, in order to pack equal types in a single array.

In the imperative world, sum data types are often defined as tagged unions. Languages like C and C++ give the programmer the freedom to construct such data types in a memory efficient manner, but without the compile time safety guarantees that functional languages provide. Representing arrays of sum types in a struct-of-arrays layout, which is more suited for data-parallel applications, is possible, but with the same drawback.

Rust has good support for sum types, which are called enums in the language [27]. Enums are stored in a similar fashion to tagged unions. Both the tag and the fields are efficiently packed into a single struct, and the compiler provides strong type safety guarantees. However, Rust does not have support for representing arrays of these types in a struct-of-arrays format.

The different representations can be summarized and divided up in four different categories:

Tag and Pointer

This is the common approach taken by most functional programming languages. A sum data type is represented by a tag, representing the constructor choice, and a pointer to the fields of the chosen constructor. In a lazy evaluated language, this layout is necessary, because instead of a value, a thunk (an unevaluated function) can lie behind a pointer. Following pointers is a somewhat costly operation, especially on GPU's, making this representation suboptimal for use in parallel array languages. On the upside, the fields can be packed efficiently, only taking up the minimal amount of space required for the fields of the chosen constructor. Because Accelerate lacks pointers, this approach cannot directly be used in the language. Instead of using a pointer, an index into an array could be used however.

Unboxed

Functional array-based languages with sum type support, like Accelerate and Futhark, spread the constructors of sums over multiple arrays. That means that multiple arrays are allocated for a sum type, each representing a separate part of the sums. The tags, representing the constructor choices, are put in the first array, and other arrays represent a single field of a single constructor each.

This approach is not very memory-efficient, as at each index of the arrays at least part of the fields are unused. The unboxed sum types in Haskell also belong to this category, as all fields are aligned next to each other in memory, without pointers involved. Haskell does not use a struct-of-arrays representation however. The Unboxed layout is shown in Figure 3.

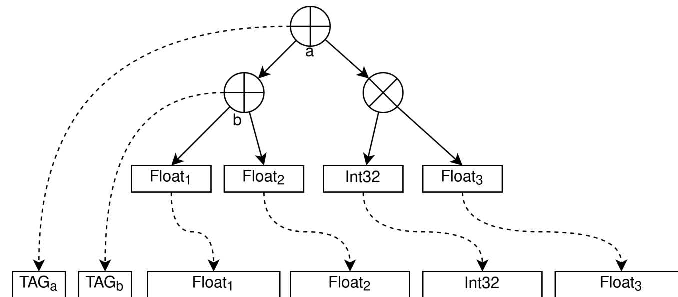

Figure 3: *The memory layout of the type* $(Float_1 + Float_2) + (Int32 \times Float_3)$ *in the unboxed sums representation. Sums are displayed as a circle with a plus symbol in it, products as a circle with a cross in it. The sums are transformed into a product-like structure. The choices of nested sums* $(+_b)$ *are represented with a separate array of tags.*

Tagged Union

This is the common approach taken by C-like languages, representing the sum type as a tag, and directly adjacent in memory, unions of the fields of each constructor. This approach does not use pointers, preserving some memory. The width of a tagged union in memory depends on memory padding (alignment to memory addresses) or packing (no alignment) and the target architecture.

In a struct-of-arrays memory layout the fields of a single constructor have to be mapped to separate arrays. This means that per array types have to be padded to the width of the largest inhabitant. An example of this layout is shown in Figure 4.

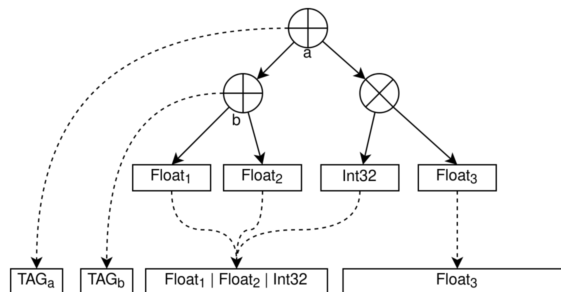

Figure 4: *The memory layout of* $(Float_1 +_b Float_2) +_a (Int32 \times Float_3)$ *in the tagged unions representation. Sums are displayed as a circle with a plus symbol in it, products as a circle with a cross in it. The sums are transformed into unions. The choices of nested sums* $(+_b)$ *are represented with a separate array of tags.*

Sorted Fields

This approach, which has been proposed by Troels Hendriksen [10] but (as far as we know) never implemented, sorts the fields by type, and can be used in a struct-of-arrays representation. This is more efficient than the Unboxed approach, but is generally less efficient than a tagged union. An example of this layout is shown in Figure 5.

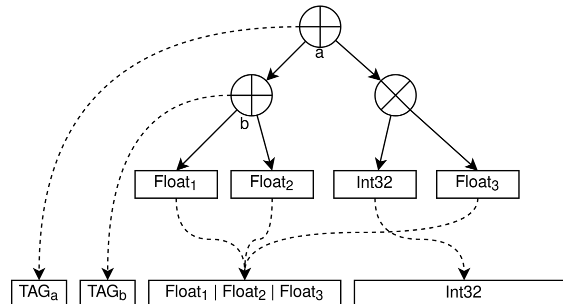

Figure 5: *The memory layout of* $(Float_1 +_b Float_2) +_a (Int32 \times Float_3)$ *in the sorted fields representation. Sums are displayed as a circle with a plus symbol in it, products as a circle with a cross in it. The choices of nested sums* ($+_b$) *are represented with a separate array of tags. The* $Float_3$ *and* $Int32$ *in the second constructor of* $+_a$ *have swapped places compared to the order in which they are defined. The chosen ordering* ($TAG > Float > Int32$) *is arbitrary, and any other ordering could have been possible.*

The different approaches are compared in section 3.

2.5 Type level programming

In Haskell, it is possible to write functions not only on the value, but also on the type level. This makes it possible to express dependencies between the types of a function. In turn, this makes it possible to enforce stronger static guarantees on a program. Type functions are used frequently in the library presented in section 4.

In order to understand how type level functions work, it is important to understand *kinds*. Each value in Haskell has a type. The value `1` has the type `Int` [1], the value `"Hello World!` has type `String`. Each function in Haskell also has a type. The function `&&` has the type `Bool -> Bool -> Bool`, the function `map` has the type `(a -> b) -> [a] -> [b]`. All of the types in Haskell also have a *kind*, the type of a type. The kind of `Int`, `String`, `&&` and `map` is `Type`, also denoted with `*`.

Data types can have more complex kinds. The kind of the `Maybe` type is `Type -> Type`. It takes a type, and returns a type. When a type is applied to `Maybe`, for example the `Int` type, it returns the type `Maybe Int`. The kind of `Maybe Int` is just `Type`.

Haskell, with the right language extensions enabled, supports kinds that are not just combinations of `Type` and arrow (`->`). Listing 9 shows the definition of the natural numbers, Peano style. By enabling the `DataKinds` [30] language extension, the constructors and types are lifted to types and kinds respectively. This means there is a kind `Nat` of which the inhabitants are the the types `Zero`, `Succ Zero`, et cetera.

```
data Nat = Succ Nat
         | Zero
```

Listing 9: *The `Nat` data type defines the natural numbers, Peano style. The `Nat` type is lifted to the `Nat` kind by enabling the `DataKinds` language extension. Similarly, its constructors, `Zero` and `Succ`, are lifted from values to types.*

[1] This is a simplification. The type of `1` can also be `Float`, or any other number type, depending on the context. This is called *overloading*.

In order to use types of this `Nat` kind, functions have to be defined that do not compute values, but types. There are different sorts of type functions in Haskell, two of which are used in the library presented in section 4. The first is closed type families [29], which are defined analogous to normal functions. An example of a type family is the `Sum` type family, defined in listing 10. It adds together two type level natural numbers, analogous to the `+` operator.

```
type family Sum (a :: Nat) (b :: Nat) :: Nat where
  Sum Zero y = y
  Sum (Succ x) y = Succ (Sum x y)
```

Listing 10: *The `Sum` type family adds two natural numbers of kind `Nat` together, to produce a new `Nat`.*

Haskell also allows for the definition of types that depend on a type of which the kind is not `Type`. This is enabled by the *GADT* [25] language extension. Listing 11 shows a *GADT*, a vector that depends on a type of kind `Nat`. This natural number encodes the length of the vector, which makes the length of the vector known statically.

```
data Vector n a where
  Cons :: a -> Vector n a -> Vector (Succ n) a
  Nil :: Vector Zero a
```

Listing 11: *The `Vector` type, defined as a Generalized Abstract Datatype, or GADT. The type is indexed by two variables. `n` encodes the lenght of the vector, `a` encodes the type of the values in the vector.*

In order to make use of the `Sum` type family, it has to be used in the type of some function. As an example, with the ingredients defined above, it is possible to define a function to append two vectors, shown in listing 12. In order to return a `Vector` type with the right length, the sum of the lengths of the input types has to be calculated. The `Sum` type establishes this relation between the input types and the return type of the `append` function.

```
append :: Vector n a -> Vector m a -> Vector (Sum n m) a
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Listing 12: *A function that appends two `Vectors`. In order to write the return type, a relation has to be established between the input types. This is done with the `Sum` type family.*

Haskell also supports the definition of infix type operators. Instead of the `Sum` type defined above, the addition of two naturals can also be defined with the `+` operator, as shown in listing 13. The `TypeOperators` language extension has to be enabled to define and use this function.

```
type family (+) (a :: Nat) (b :: Nat) :: Nat where
  Zero + y = y
  (Succ x) + y = Succ (Sum x y)

infixl 6 +
```

Listing 13: *A type function that adds together two natural numbers, as an infix type operator. In order to define and use this function, the `TypeOperators` language extension has to be enabled.*

It is not necessary to define type level naturals and addition. Instead, this functionality can be imported from the various GHC libraries. This also makes it possible to write the type `Succ Zero` as just `1`. The GHC libraries also support type level lists, including a list syntax that looks like the value-level syntax.

It is sometimes useful to convert a type of kind `Nat` to its equivalent value. That is: convert the type `1 :: Nat` to the value `1 :: Integer`. The `natVal` function, part of the `KnownNat` class, is able to do exactly this. The `KnownNat` constraint is automatically derived for simple `Nats` by GHC. This however is not done for compound types, which is where the GHC plugin `KnownNat Solver` [2] comes in. This plugin can generate `KnownNat` constraints for compounded `Nats`. For the type function shown in listing 14 it generates the constraint `KnownNat (Length xs)`. This works because the plugin knows how to add up `Nats`, and is able to perform recursive calculations. To prevent infinite calculations at compile time, a maximum recursion depth is set. The plugin makes working with `Nats` more ergonomic, as there is not need to pass complex constraints around or prove properties of natural numbers to the compiler.

```
type family Length (xs :: [a]) :: Nat where
  Length '[] = 0
  Length (x ': xs) = Length xs + 1
```

Listing 14: *A type function that calculates the length of a type list. The kind of the single argument `xs` is a list of `a`'s. The result kind is `Nat`, a kind containing all natural numbers as type. The `'[]` and `':` types are the type equivalent of `[]` and `:`. The `+` type adds two natural numbers together.*

The *Finite* library [17] makes use of the type level `Nats`, by restricting the values of the `Finite` type to be lower than the value of its `Nat` type argument. This makes it possible to, for example, limit the index to a vector of length `n` to `Finite n`, which can statically prevents out-of-bounds errors.

Haskell also supports associated types [4], which make it possible to add a type (or type function) to a class. Each instance of this class has to give a definition of this type. This makes it possible to express class functions in terms of the associated type.

2.6 Data type generic programming

Haskell supports datatype generic programming [3], a way of abstracting over the structure of a data type. This makes it possible to write functions that are polymorphic over the structure of a data type. The schoolbook example for such a function is `show`, which returns a string representation of some value. A programmer can instantiate this function for their own datatype by using the `deriving` statement, as shown in listing 15.

```
data Maybe a = Nothing
             | Just a
             deriving (Show)
```

Listing 15: *Deriving the `Show` class for the `Maybe` type. The `Show` class is one of the stock derivable classes, which exist since Haskell 98. The `show` function from this class is commonly used for debugging, as it creates a string that shows the structure of some value.*

In Haskell 98, only a select amount of classes can be derived (`Bounded`, `Enum`, `Eq`, `Ord`, `Ix`, `Read` and `Show`). Implementation of these is left to the compiler. In order to let programmers use the deriving mechanism for their own classes, the language has been extended [19]. This makes it possible to define a default implementation for the functions of a class. The deriving statement for user defined classes selects the default implementation of these functions.

Creating a default implementation of a class however is not really useful if nothing is known about the derived type. To write a default implementation for the `show` function, for example, it is important to know the amount and names of the constructors, and the amount of fields they contain. The `Generics` class was introduced to make this information available to the programmer.

The `Generics` class can be derived from a datatype by the compiler, and gives structural and nominal information about the type. Its `from` function returns a structure based on the `:*:` and `:+:` types, for product and sum types respectively. Writing separate implementations for

the different types make it possible to 'pattern match' on the structure of the type. This makes writing default implementations of classes more powerful. The GHC Generics representation of the type `Maybe Int` is shown in listing 16.

```
D1 (C1 U1 :+: C1 (S1 (Rec0 Int))) x
```

Listing 16: *Simplified GHC Generics representation of `Maybe Int`. It is simplified in the sense that all meta data (like constructor names) have been removed from the representation. The `D1` type represents data types, the `C1` type represents constructors, `:+:` represents a binary sum, `U1` represents unit, `S1` represents a 'record selector'.*

Representing a type as binary sums and products however makes it relatively hard to perform some operations. Something relatively simple, like counting the amount of constructors of a data type, can lead to quite a lot of (boilerplate) code. The sums-of-products (SOP) approach instead captures the structure of data types as a sum-of-products. This means that each type is represented as a list of lists, where the outer list represents the constructors, and the inner list the fields of this constructor. This makes it more intuitive to write generic programs. The `generics-sop` library provides a set of combinators on the SOP structure that capture idiomatic generic code. Additionally, the library separates metadata (like constructor names) from the structure of the datatype, enabling generic programs to ignore meta-data, or instead provide their own. The *generics-sop* representation of the type `Maybe Int` is shown in listing 17.

```
NS (NP I) '[ '[], '[Int]]
```

Listing 17: *generics-sop representation of `Maybe Int`. The `SOP` type represents a sum of products, `I` is the type level identity function.*

Instead of writing instances for each type, the *generics-sop* library makes it possible to use pattern matching to bring the structure of a type. The outer type list, the product, is captured by the `NP` type. The inner type lists, the sum, is captured by the `NS` type. The definition of these types is shown in listing 18. For example, `Just 1`, a value of the `Maybe Int` type as shown above, is represented with the value `S (Z (I 1 :* Nil))`.

```
data NP :: (k -> Type) -> [k] -> Type where
  Nil  :: NP f '[]
  (:*) :: f x -> NP f xs -> NP f (x ': xs)

data NS :: (k -> Type) -> [k] -> Type where
  Z :: f x -> NS f (x ': xs)
  S :: NS f xs -> NS f (x ': xs)
```

Listing 18: *The definition of the `NP` and `NS` types in the generics-sop library. Both type have the same kind, taking a type function that maps a k to a type and a type lists of ks, and returning a type. `NP` is analogous to a list, with a `Nil` and a `:*` constructor, the latter being the cons operator. `NS` is defined as an index to the type level list, with a the zero and successor constructor, `Z` and `S` respectively.*

An example of a SOP program is shown in listing 19. This function is an implementation of the equality operator (`==`). The `eq` function creates a `SOP` structure from the two values, and applies the `EqS` function to the `SOP` structures. The `EqS` function in turn compares if the same constructor is chosen. The chosen constructor is represented by applications of `Z` and `S`. If an equal amount of `S`'s is applied, the same constructor was chosen. If this is the case, it applies the `EqP` function to the inner product, otherwise it returns False. The `eqP` function compares the values of the inner products to each other. While the heterogeneous lists containing the fields of the product is not empty, the head elements are compared, and the function recurses over the tail. When the heterogeneous lists are empty, `True` is returned. The *generics-sop* library

contains many functions that capture typical generic programming idioms, making it possible to write very concise functions. These are not used in this example in order to show the usage of the constructors of `NP` and `NS`.

```
eq :: (All2 Eq (Code x), Generic x) => x -> x -> Bool
eq x y = eqS (unSOP $ from x) (unSOP $ from y)

eqS :: (All2 Eq xss) => NS (NP I) xss -> NS (NP I) xss -> Bool
eqS (Z xs)  (Z ys)  = eqP xs ys
eqS (S xss) (S yss) = eqS xss yss
eqS _       _       = False

eqP :: All Eq xs => NP I xs -> NP I xs -> Bool
eqP Nil         Nil         = True
eqP (I x :* xs) (I y :* ys) = x == y && (eqP xs ys)
```

Listing 19: *An implementation of* `==` *with use of generics-sop.*

3 Representing Sum Types Efficiently

In this section we present a new memory representation for sum data types, suited for a struct-of-arrays layout. Next, the theoretical optimal size of a data type is discussed. Finally, we compare the different existing and proposed representations (see section 2.4) against the theoretical optimum and show that our representation is memory efficient.

3.1 Recursive Tagged Union

In this thesis we propose and implement a recursive variant of tagged unions. Tagged unions are described in more details in section 2.4. A limitation of Accelerate is that no loops or recursion are allowed in scalar expressions. This limitation can be used to represent nested sum types more efficiently. Because recursion is disallowed, each data type is of finite size, and recursing into fields is thus a finite operation. By recursing over a datatype and gathering all constructor choices, a single tag can be created which describes the whole structure. Other representations create a new tag for a nested sum instead. In the same vein all fields can be gathered, which can be represented as a compact product of unions. Each union in the product then becomes the width of the largest inhabitant. Each of these unions can be translated to a separate array, with a union type, in an struct-of-array memory representation.

A simple sum type like `Either Int Float` is represented with a single tag with two inhabitants, and a single union of `Int` and `Float`. A more complex, nested, sum type like `Either (Maybe Int) (Maybe Float)` is represented with a single tag with four inhabitants (`Left Nothing`, `Left Just`, `Right Nothing`, `Right Just`). Again, its fields are represented with a single union of `Int` and `Float`. A product type containing sum types also has a single tag. The type `(Maybe Int, Maybe Float)` is represented with a single tag with four inhabitants (`(Nothing, Nothing)`, `(Nothing, Just)`, `(Just, Nothing)`, `(Just, Just)`). Its fields are represented with two unions of a single element (`Int` and `Float`). An visual example of this layout is shown in Figure 6.

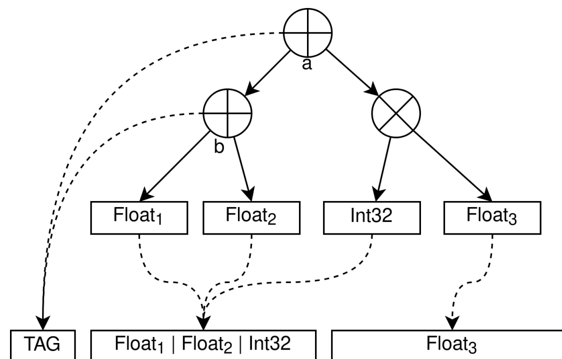

Figure 6: *The memory layout of* $(Float_1 +_b Float_2) +_a (Int32 \times Float_3)$ *in the Recursive Tagged Union representation. Sums are displayed as a circle with a plus symbol in it, products as a circle with a cross in it. The first union contains the first machine type in each constructor, the second union contains the second machine type in each constructor. The choices of nested sums do not require an extra array of tags.*

Shallow pattern matching on this representation can be a little less efficient compared to tagged unions, because each pattern can match multiple tags, and the tags of nested sum types have to be rebuild after a pattern match succeeds. This problem can be partially eliminated by using reachability analysis (see section 6.2). In contrast, nested pattern matching on this representation is more efficient than on other representations, because only a single memory access is needed to retrieve the types of the fields of the whole data structure.

3.2 Theoretical efficiency

We compare the space usage of these approaches with the minimal amount of memory that should be reserved for a type. The minimal amount of bits needed to store a data type can be calculated by adding the amount of bits needed to represent the constructor choices in the datatype to the amount of bits needed to store the fields. The minimal amount of bits needed to store the constructor choice of some sum type can be calculated by taking the the log_2 of the amount of constructors. This number then has to be rounded up to a whole number. The minimal width of a product is equal to the sum of the widths of its fields, while the minimal width of a sum is equal to the maximum of the widths of its fields. This can be summarized as:

$$width(x) = \left\lceil log_2(|x|) + \max_{i=0}^{|x|} \left\{ \sum_{j=0}^{|x[i]|} width(x[i][j]) \right\} \right\rceil \tag{1}$$

This definition is recursive, which means base cases are needed. These can be found in machine types, which have a statically known width. Recursive data types such as lists do not have a statically known width.

3.3 Comparison

	Float?	Float + Float	Float + Double	Float + Double × Double + Float	Double? + Double?
Tag and Pointer	72 or 104	104	104 or 136	168	144 or 208
Unboxed	40	72	104	216	152
Tagged Union	40	40	72	136	80
Sorted Fields	40	40	104	104	80
Recursive Tagged Union	40	40	72	136	72
Minimum bits needed	33	33	65	97	66

Table 1: *The amount of bits reserved for some common sum data types in different representations, compared to the minimum amount of bits needed. Tags are assumed to be one byte wide, as most hardware does not permit addressing smaller offsets. The calculations hold for a struct-of-arrays representation. See Appendix A for the full calculations. 'Float?' is notation for the type 'Float + undefined', commonly referred to as an option type (`Maybe` in Haskell).*

Table 1 shows the amount of bits that have to be reserved for some common sum data types in different representations. The minimum amount of required bits per type is also shown. From the small selection of types some trends emerge. Using pointers has significant memory overhead compared for smaller types. For larger sum types the Unboxed approach takes up the most space. The Tagged Union, Sorted Fields and Recursive Tagged Union approaches all perform quite well. For most of the compared types, these approaches are within byte rounding difference from the minimum width. The Sorted Fields approach shines when the fields of the different constructors have the same types, no matter the order. The Tagged Union and Recursive Tagged Union perform very similar, although there is an edge to the recursive approach for recursive types.

The theoretical results in table 1 show that the Recursive Tagged Union approach performs quite well compared to other representations. It scales better with larger sum types that have

different types in the constructors compared to the Sorted Fields approach. A combination of the approaches would be even more memory efficient, as described in section 6.3.

4 The POSable library

The Recursive Tagged Union approach has been implemented in the `POSable` library [12], which generically calculates this memory layout for non-recursive Haskell-98 data types. `POSable` uses `generics-sop` [7] as a generics library, which represents data types as Sums of Products (as discussed in section 2.6). The `POSable` library represents data types as a (tagged) Product of Sums instead, hence its name. The library is available under the BSD-3-Clause license on GitHub [12].

This section first describes the goal of the library, and the constraints that influenced the design of it. It then discusses the `POSable` class, which captures the Recursive Tagged Union layout in a Haskell class. An example implementation of this class for a non-polymorphic data type is given in section 4.3. In section 4.4 this implementation is generalized to a polymorphic data type. In the next section, the generic implementation is explained, building on the polymorphic implementation. In the last section, limitations of the library are outlined.

4.1 Goal and constraints

The goal of the `POSable` library is to calculate, for some type, the information a compiler needs in order to store that type in the Recursive Tagged Union layout. The library encodes the Recursive Tagged Union layout in types, making this information available statically. By using strict types, the library has strong correctness guarantees.

Embedded compilers need to be able to lift values from plain Haskell to their internal language, and back. Since the `POSable` library is used by an embedded compiler, it also needs to be able to lift values to its representation type, and back. The library provides functions to perform these operations.

The library provides its representation type and the conversion function as part of a type class. This makes it possible to write a generic implementation for it, while allowing users to override the default behavior.

It is important that this class is derivable by the user. This makes it possible for a programmer to create their own types, and use them in Accelerate, without having to know how the representation works. A deriving statement cannot bring compound constraints into scope. Simple superclass constraints, of the form `Superclass a => Class a`, are fine however. This limitation on the class constraints come with some additional complexity, explained in section 4.5. Only non-recursive types need to be derivable, as recursive types cannot be represented in the Recursive Tagged Union layout.

The library is built in a reusable way, which makes it possible to integrate it in other projects. This means that the code in the library is not specialized to Accelerate. Because different languages can have different sets of machine representable types, the set of these types is provided in a class. A library user can extend this class with types of their liking. This means no assumptions are made about which types are machine representable.

4.2 Representing types

The POSable library captures the Recursive Tagged Union representation of a type, and the functions to convert values from and to this representation, in the `POSable` class. This class contains all information Accelerate needs to transform a list of some type into an efficient SoA representation. This section explains the associated types and functions of this class.

4.2.1 Choices

In the Recursive Tagged Union representation all constructor choices in a type are represented with a single natural number. The value of this number is bounded by the amount of choices in the the data structure, which can be calculated at compile time. The POSable library captures this maximum value in the associated type `Choices`. This type is of kind `Nat`. To constraint values under this bound, the *Finite* library [17] is used. Its main type, also named `Finite`, takes a type argument of kind `Nat`, and bounds its possible values to be less than this type. As an illustration, the possible values of type `Finite 3` are 0, 1 and 2.

To get the tag belonging to some value, the `choices` function is used. This function has the type type `x -> Finite (Choices x)`.

4.2.2 Fields

The fields of a type are captured at the type level as a list of lists, similar to how `generics-sop` represents types. The outer list represents a product, while the inner lists represent sums. This list of lists of types is stored in the `Fields` associated type. Its kind is `[[Type]]`. For example, the fields of `Either Int Float` is represented by `[[Int, Float]]`. Since the outer list represent a product, this is only a product of one element. This makes sense, as both constructors of `Either` have a single field, which in turn are machine types. The inner list, the sum, contains both of these fields. This also makes sense, as only one of the fields can be present at the same time. Note the different embedding order of the list-of-lists compared to the *generics-sop* library. In that library, `Either Int Float` is represented as `[[Int], [Float]]`.

To show the recursive aspect of the representation, consider the type `Either (Int, Float) (Maybe Double)`. By recursing into the nested types, all fields are gathered in a single structure. The type is represented by `[[Int, Undef, Double], [Float, Undef]]`. The `Undef` type here is used when a Sum potentially holds no value, for example when the `Nothing` constructor from the `Maybe` type is chosen. The `Int` and `Float` fields of the left constructor end up in different sums. This is necessary, as the values are present at the same time. When instead the right constructor of `Either` is chosen, the second sum does not contain a value. This is denoted with `Undef`. Depending on the chosen constructor of `Maybe` the first sum contains a `Double` or an `Undef`.

To capture the product-of-sums structure at the value level, the `Product` and `Sum` types are used. `Product`s contain a value for each type in the type list, while `Sum`s only contain one value from the type list. This is similar to `NP` and `NS` from the `generics-sop` library. The `Product` type is of kind `[[Type]] -> Type`, while the `Sum` type is of kind `[Type] -> Type`. The `Product` type constrains its values to the `Sum` type. See listing 20 for the definitions of the `Product` and `Sum` types.

```
data Sum :: [Type] -> Type where
  Pick :: Ground x => x -> Sum (x ': xs)
  Skip :: Ground x => Sum xs -> Sum (x ': xs)

data Product :: [[Type]] -> Type where
  Nil :: Product '[]
  Cons :: Sum x -> Product xs -> Product (x ': xs)
```

Listing 20: *Sums and Products*

To capture the set of types with a statically known width, ground types, the `Ground` class is introduced (see listing 21). The library only adds one inhabitant to this class (`Undef`), giving the users of the library the freedom to choose their own set of ground types. `Sum`s can only contain types that implement `Ground`. This constraint makes it necessary to introduce the `Product` and `Sum` types, instead of reusing the `NP` and `NS` types.

```
class (Typeable a) => Ground a
```

Listing 21: *The Ground class, which captures the set of types that can occur as a type in the `Fields` of an instance of the `POSable` class. This set is extensible by the user of the `POSable` library. The class requires `Typeable` as a superclass, which makes it possible for library users to inspect the type at runtime (see section 6.4).*

4.2.3 The POSable class

The choices and fields as defined above are captured in the `POSable` class. This class contains two associated types, one for `Choices` and one for `Fields`. These are then used as the result of the `choices` and `fields` functions respectively. It is also possible to convert back to the original type, which is captured in the `fromPOSable` function. See listing 22 for the definition of this class.

```
class POSable x where
  type Choices x :: Nat
  type Fields x :: [[Type]]

  choices :: x -> Finite (Choices x)
  fields :: x -> Product (Fields x)

  fromPOSable :: Finite (Choices x) -> Product (Finite x) -> x
```

Listing 22: *The POSable class*

4.3 Mapping a simple type

Implementation of the `POSable` class for a non-polymorphic data type is fairly easy. Consider the datatype `Either Int (Float, Double)`. The type contains two possible choices, `Left` and `Right`. This means that the `Choices` type should be 2. The corresponding `choices` function should return 0 when the `Left` constructor is chosen, en 1 when the `Right` constructor is chosen. The `Finite` type makes sure that only these values are admissible as the values of `choices`.

The definition of the fields is a little more involved. The type `Either Int (Float, Double)` contains a product of two types. This means that the outer list is of length two. In the first sum (the first element of the outer list), the first field of each constructor is stored. These elements are `Int` and `Float`, from respectively the `Left` and `Right` constructor. In the second sum, the second field of each constructor is stored. The `Left` constructor however does not have a field in the second position. Instead, the `Undef` type is used here. The second sum thus contains `Undef` and `Double`. Taking this altogether, the `Fields` type becomes `[[Int, Float], [Undef, Double]]`.

The corresponding `fields` function should return a `Product` corresponding to the type. If the `Left` constructor is chosen, the `Int` it contains is put in the first element of the product. The `Int` is also the first element of this sum, which is denoted by not skipping any element, but immediately using `Pick`. To give a value to the second element of the `Product`, the `Undef` constructor has to be used. The `Undef` type is again the first element of the `Sum`, and again the `Pick` constructor can be used immediately. If instead the `Right` constructor is chosen, there are two values in scope. The first, of type `Float`, is stored in the first sum, while the second, of type `Double`, is stored in the second sum. Both are in the second place of their respective sums, which means the `Skip` constructor is applied once to put them in the second position.

The implementation of `POSable` for the `Either Int (Float, Double)` type is summarized in 23. This implementation assumes that `Int`, `Float` and `Double` are part of the `Ground` class. The implementation of the `fromPOSable` function is a straightforward inverse of the `choices` and `fields` functions.

4.4 Mapping polymorphic types

To map all data types to the POS representation, polymorphic types need to be considered. For example, it should be possible to define an instance for `Either a b`, where a and b are variables. This instance can then be used by all concrete types of the form `Either a b` for which a and b have an instance for `POSable`. This means no separate instances need to be defined for `Either Int Float` and `Either Float Int`, but just one instance suffices. This section explains the implementation of `POSable` for polymorphic types.

```
instance POSable (Either Int (Float, Double)) where
  type Choices (Either Int (Float, Double)) = 2
  type Fields (Either Int (Float, Double)) = '[ '[Int, Float], '[Undef, Double]]

  choices (Left _) = 0
  choices (Right _) = 1

  fields (Left x) = Cons (Pick x) (Cons (Pick Undef) Nil)
  fields (Right (x,y)) = Cons (Skip (Pick x)) (Cons (Skip (Pick y)) Nil)

  fromPOSable 0 (Cons (Pick x) (Cons (Pick Undef) Nil)) = Left x
  fromPOSable 1 (Cons (Skip (Pick x)) (Cons (Skip (Pick y)) Nil)) = Right (x,y)
```

Listing 23: *An instance of the POSable class for `Either Int (Float, Double)`. This type consist out of a sum and a product, showing the construction of both. The `fromPOSable` function is the inverse of the `choices` and `fields` functions.*

4.4.1 Choices

In order to implement `POSable` for a polymorphic type, the `Choices` type and `choices` function have to be implemented. Both have to be implemented in terms of the nested type(s).

To implement `Choices` for a type like `Either a b`, the sum of `Choices a` and `Choices b` has to be calculated. On the value level, when the `Left` constructor is chosen, the value of `choices` is equal to the `choices` value of the nested type. When the right constructor is chosen however, the amount of `Choices` in the left constructor has to be added. This functionality is provided by the `Finite` library as the `combineSum` function, displayed in listing 24.

```
-- | 'Left'-biased (left values come first) disjoint union of finite sets.
combineSum :: KnownNat n => Either (Finite n) (Finite m) -> Finite (n + m)
combineSum (Left (Finite x)) = Finite x
combineSum (Right (Finite x)) = Finite $ x + natVal (Proxy @n)

-- | Take a 'Left'-biased disjoint union apart.
separateSum :: KnownNat n => Finite (n + m) -> Either (Finite n) (Finite m)
separateSum (Finite x) = if x >= (Proxy @n)
                            then Right $ Finite $ x - (Proxy @n)
                            else Left $ Finite x
```

Listing 24: *The `combineSum` and `separateSum` functions from the `Finite` library. The former combines two `Finite`'s by calculating the index in the disjoint union between the two sets. The latter reverses this operation.*

For product types, instead of summing, the `Choices` have to be multiplied. The values should not be multiplied however. For example, in the type $a + b \times c + d$, there are two tags in the left operand of the product, and two tags in the right operand. The values of these tags are 0 and 1. The product should contain tags 0, 1, 2 and 3. We achieve this result by taking the index in the cartesian product of the two finite sets of the operands. Simpler put, we use the left operand as the outer iteratee, and the right operand as the inner iteratee. This means that in the previous example tags $0,0$ becomes 1, $0,1$ become 1, $1,0$ becomes 2 and $1,1$ becomes 3. This is calculated by the `combineProduct` function from the `Finite` library, displayed in listing 25.

4.4.2 Fields

In order to implement `POSable` for a polymorphic type, its `Fields` type and `fields` function also have to be implemented. Both have to be implemented in terms of the nested type(s).

```haskell
-- | 'fst'-biased (fst is the inner, and snd is the outer iteratee) product of
--   finite sets.
combineProduct :: KnownNat n => (Finite n, Finite m) -> Finite (n * m)
combineProduct (Finite x, Finite y) = Finite $ x + y * natVal (Proxy @n)

-- | Take a 'fst'-biased product apart.
separateProduct :: KnownNat n => Finite (n * m) -> (Finite n, Finite m)
separateProduct (Finite x) =
  (Finite $ x `mod` (Proxy @n), Finite $ x `div` (Proxy @n))
```

Listing 25: *The `combineProduct` and `separateProduct` functions from the `Finite` library. The former combines two `Finite`'s by calculating their index in the cartesian product of the two sets. The latter reverses this operation. `separateProduct` is a rather expensive operation, use of which should be prevented in hot code paths if possible.*

For a product type, this means concatenating the `Fields` of the nested types. Consider the following polymorphic type: $a + b \times c + d$. The fields of the inner sums are represented as `[[a, b]]` and `[[c, d]]`. The fields of this type should be mapped to `[[a, b], [c, d]]`. This means this `Products` should be concatenated. To perform type level concatenation the type function `++` is used. The value equivalent of this is simply also a concatenation.

For a sum type however, the `Fields` should not be concatenated, but zipped instead. Consider the polymorphic type $(a \times b) + (c \times d)$. The fields of the inner products are represented as `[[a],[b]]` and `[[c], [d]]` These fields should be mapped to `[[a, c], [b, d]]`. This is achieved by `zip`ping sum concatenation over the outer products. This is done by the type function `Merge`. This type level version of `zip` keeps the length of the longest list, and appends the shorter list with `Undef`s to match the length. The definition of `Merge` is shown in listing 26. On the value level, the same operation is achieved with the `merge` function.

To zip fields on the value level, it is needed to know on the structure of the type level lists at runtime. This is problematic, as only one of the components of the sum is in scope (that of the chosen constructor). Either an implicit (constraint) or explicit (value) representation of the other constructors needs to be in scope. In the `POSable` library this is achieved by the adding the `emptyFields` function to the `POSable` class, which returns a `ProductType` value. The `ProductType` value represents the type of a `Product`, and is build up from `SumType`s, which in turn represent the type of the `Sums`. In the *generics-sop* library a similar problem is solved with an implicit representation (in the form of the `SListI` class). This approach however does not work in this context, as this brings in scope compound superclass constraints, which have to be avoided in order to make the class derivable.

```haskell
type family Merge (xs :: [[Type]]) (ys :: [[Type]]) :: [[Type]] where
  Merge '[] '[] = '[]
  Merge '[] (b ': bs) = (Undef ': b) ': Merge '[] bs
  Merge (a ': as) '[] = (a ++ '[Undef]) ': Merge as '[]
  Merge (a ': as) (b ': bs) = (a ++ b) ': Merge as bs
```

Listing 26: *The type function `Merge`. This function `zip`s two type lists, keeping the length of the longest list, while filling the shorter list with `Undef`s to match the length.*

With the tools described above, an instance of `POSable` for polymorphic `Either` can be created. The implementation of `POSable` as shown in listing 27. The implementation of `fromPOSable` is not discussed above, but follows directly from the implementation of `choices` and `fields`.

4.5 Generics

The `POSable` class has default definitions for each associated type and function, which are implemented in the `GPOSable` class. Default definitions make a class derivable by a user (if the the

```haskell
instance (KnownNat (Choices l), POSable r, POSable l)
    => POSable(Either l r) where
  type Choices (Either l r) = Choices l + Choices r
  type Fields (Either l r) = Merge (Fields l) (Fields r)

  choices (Left x)  = combineSum (Left (choices x))
  choices (Right x) = combineSum (Right (choices x))

  fields (Left x)  = merge (Left (fields x, emptyFields @r))
  fields (Right x) = merge (Right (emptyFields @l, fields x))

  fromPOSable n x = case separateSum n of
                    Left n' -> Left (fromPOSable n')
                      (splitLeft x (emptyFields @l) (emptyFields @r))
                    Right n' -> Right (fromPOSable n')
                      (splitRight x (emptyFields @l) (emptyFields @r))
```

Listing 27: *An instance of POSable for the polymorphic `Either` type. The `Choices` of the argument types are summed together, and the `choices` function is implemented in terms of `combineSum`. The equivalent for `Fields` are the `Merge` type and `merge` function. The `fromPOSable` function is implemented in terms of the `separateSum` and `split` functions. These perform the reverse of `combineSum` and `merge` respectively.*

language extension `DeriveAnyClass` is enabled).

The implementation of the `GPOSable` class relies mostly on the functions (both value- and type-level) that are built for polymorphic types. Most of these functions are binary, and are generalized to operate on lists. This is because types with all possible amounts of constructors and amounts of fields in these constructors need to be handled. Because it is not possible to create a higher order type family in Haskell, mapping these functions over lists is done by creating a new type function that performs the mapping. For example, to map the `Choices` type over a list, the `MapChoices` type is created. If higher order type level functions were supported, this could be written as `Map Choices` instead.

A derivable statement cannot rely on complex constraints being in scope, but only constraints brought in scope by other deriving statements. The derivable class should thus only have simple superclass constraints of the form `Superclass a => Class a`. The `POSable` class however, has constraints on the inhabitants of the type lists in `Fields`, and on the `Choices`.

By capturing the `Ground` constraint on the inhabitants of `Fields` in the `Product` and `Sum types`, the first problem is circumvented. The `KnownNat` constraint on `Choices` however is brought in scope in a different manner. Although the `Finite` class could have been adapted to bring in scope the `KnownNat` constraint, this would have brought along some boilerplate. Instead, the GHC `KnownNatSolver` plugin [2] is used. This plugin is discussed in more detail in section 2.5. A similar solver would make it possible to use this trick for type lists, but, as far as we known, this does not exist yet. Instead, the `emptyFields` function is used to bring in scope the structure of a type list.

Not all types can derive `POSable`. Examples of types that cannot are recursive types like lists and machine types. Recursive types are not supported by design, as the Recursive Tagged Union representation does not allow recursive types. Machine types however should be supported, as these are the obvious candidates for `Ground` types. These types however do not have an instance of `Generic`, which is needed to derive `POSable`. Instead, the instances for these types are generated with Template Haskell [26]. The `mkPOSableGround` can be called by a `POSable` user to generate the `POSable` instance for their `Ground` types.

4.6 Limitations

In the current implementation of the library, product types are represented with a single tag. This is the most memory efficient representation, but can be less ideal for performance, because it relies

on the `seperateProducts` function (see listing 25). This function uses the relatively expensive *div* and *mod* operations. Whether this is relevant depends on the use case. Possible solutions to this problem are discussed in section 5.5 and 6.2.

Haskell does not support multiple default implementations of a class. This makes it necessary to use Template Haskell to create `POSable` implementations for `Ground` types (the `mkPOSable Ground` function). Having multiple default default implementations (and way to disambiguate between them) would prevent the need for Template Haskell here.

Another limitation is that the `Ground` class cannot be extended by the library user with extra functions. This makes it hard to integrate it in the type hierarchy of Accelerate (see section 5.2). This problem presumably extends itself to other usages of the library. Sadly, this problem is not solvable in Haskell, as it does not support parameterized libraries. A solution to this problem has been proposed, namely Backpack [16], but this requires support from the package manager. *Stack*, the package manager used by Accelerate, does not support Backpack. Currently, the problem is circumvented by making the `Typeable` class an superclass of `Ground`, and using the runtime type information this provides to convert the `Ground` types to Accelerate types. A more permanent solution might be to integrate the library in Accelerate.

5 Implementation in Accelerate

The POSable library has been partially integrated in Accelerate. This integration shows the viability of using the approach in a parallel array language. A successful integration means that the LLVM code Accelerate generates reserves less memory for sum types. To achieve this goal, several parts of the compiler have to be changed.

The parts that are updated are the following: The default implementation of the `Elt` class, which is used to represent data types, are replaced. Unions are added to the type hierarchy, as are `Undef`s. The pattern matching code is updated to handle combined tags. New constructors are added to the AST to handle unions.

Finally, the backends need to be updated to handle the new AST constructors. This part still has to be implemented.

5.1 Mapping to Elt

Accelerate, unlike the POSable library, does not use the list kind to represent heterogeneous vectors. Instead such structures are build up from binary tuples, with unit denoting the start or end of the list. Such a tuple-list is called a cons or snoc lists, depending on the position of unit. The kind of such a list is `Type`, which does not put any constraints on the type. The heterogeneous list representing a type is stored in the `EltR` associated type of the `Elt` class. It is a list, not a list-of-lists, as there are no unions in the Unboxed representation Accelerate uses.

The `Elt` class also contains a set of functions that convert a type from and to `EltR`. The class has a generic implementation in terms of `GElt`, which is based on GHC generics.

In the existing version of Accelerate, singleton types, product types and sum types are all represented differently. Singleton types, like `Int` and `Float`, are represented as their own type: `EltR Int == Int`. Product types are represented as a snoc list: `EltR (Int, Int) == (((), Int), Int)`. Sum types are represented as a tuple of a `TAG` and a snoc list: `EltR (Either Int Int) == (TAG, (((), Int), Int))`.

The functions and associated types of the `POSable` class are mapped to the `Elt` class. The alternative, replacing `Elt` by `POSable`, was considered to be too invasive of an operation, as lots of the compiler code relies on the `Elt` class. This mapping consist of both type and value level mappings. `EltR` is defined in terms of `Fields` and `Choices` and `toElt` in terms of `choices` and `fields`. This replaces the default `GElt` implementation of `Elt`.

Defining `EltR` in terms of `Choices` and `Fields` means converting these higher-kinded types to the `Type` kind. In the case of `Choices` this is done by converting it to the existing `TAG` type, which is a type synonym for a byte-sized word. In the case of `Fields`, the type level outer list has to be mapped to a cons list. The inner lists, containing the unions, are captured with a type list by the `UnionScalar` type. This type is shown in listing 28.

```
data UnionScalar a where
  PickScalar  :: x -> UnionScalar (x ': xs)
  SkipScalar  :: UnionScalar xs -> UnionScalar (x ': xs)
```

Listing 28: *The `UnionScalar` type, which captures a union of types as a type lists. The union only has one (scalar) value at a time, with a type from the type list.*

The list-of-lists structure is transformed to a cons list with `UnionScalar` containing the inner lists by the `FlattenProduct` type family. This type family is shown in listing 29.

```
type family FlattenProduct (xss :: [[a]]) :: Type where
  FlattenProduct '[] = ()
  FlattenProduct (x ': xs) = (UnionScalar x, FlattenProduct xs)
```

Listing 29: *The `FlattenProduct` which maps the `Product` type from the POSable library to tuple lists, as used in Accelerate.*

Because singleton types and product types are represented differently in Accelerate, those are handled separately. Using the product-of-sums representation for all types is infeasible because of the many places in the Accelerate source code where the existing representation of singleton and product types are used. The conversion from the `POSable` representation and the `EltR` representation is done with the `POStoEltR` type family, as shown in listing 30.

```
type family POStoEltR (cs :: Nat) fs :: Type where
  POStoEltR 1 '[ '[x]] = x -- singleton types
  POStoEltR 1 x = FlattenProduct x -- product types
  POStoEltR n x = (TAG, FlattenProduct x) -- sum types
```

Listing 30: *The `POStoEltR` type function, which maps the `Choices` and `Fields` from the POSable library to the snoc lists of Accelerate. Singleton types in Accelerate are not represented as a list, but just as a type. Similarly, product types in Accelerate are represented without bringing along their tag. The `Finite` type is replaced the by Accelerates `TAG` type. This means some type safety is lost, but makes the integration less invasive.*

The default implementations of the `toElt` and `fromElt` functions are implemented in terms of POSable functions. The definitions of these functions follow the types. This means the functions split on the form of the type, either *singleton*, *tagless* or *tagged*. A helper function is used to make this distinction, while bringing the correct constraints in scope.

5.2 Type hierarchy

To use a type as a scalar value in Accelerate, it must be part of the type hierarchy. Both an implicit (classes) and explicit (values) version of this type hierarchy exist, in this section we focus on the explicit hierarchy. This hierarchy consist of representation types, values representing a single type. The top of this hierarchy is named `ScalarType`. In the existing version of Accelerate, this can be either a `SingleType` or `VectorType`. `VectorType` contains n elements of the same `SingleType`. `SingleType` in turn contains all floating and integral types. `TAG`s and `Undef` are handled separately, and are not part of the type hierarchy.

To integrate the POSable library, the representation types of `TAG`s and `Undef` needs to be added as a type in the hierarchy. `TAGType` is added to the hierarchy as an integral type, as simple mathematical operations have to be applied to `TAG`s, like comparisons, additions and multiplications. `UndefType` is added directly to `SingleType`, as there are no operations that should be applied to `Undef`. Last but not least, `UnionScalarType`, the representation type of `UnionScalar` (see listing 28), is added to `ScalarType`. This type fulfills a similar function as `SumType` in POSable, but the types are represented as cons lists instead of type lists. The resulting type hierarchy is shown in figure 7.

5.3 Pattern matching

The existing version of Accelerate generates an AST with Template Haskell for each constructor. The generated AST has to closely follow the representation of the type in memory. As the generated AST depends on the memory layout of the types, the pattern matching code has to be rewritten completely. Instead of using Template Haskell, we approached this problem by creating a new class, named `Matchable`, which has a default implementation that uses the constructs offered by POSable. The resulting code is more readable and maintainable than the Template Haskell original. The `Matchable` class is shown in listing 31.

The `Matchable` class contains two functions, `build` and `match`. The `build` function takes a heterogeneous list of `Exp`s, the types of which correspond to the types of the chosen constructor. It builds a new `Exp` from the input `Exp`s. The `match` function takes an expression and returns a heterogeneous list of `Exp`s, the types of which correspond to the types of the matched constructor. The result is wrapped in a `Maybe`. This is necessary to make embedded pattern matching work in Accelerate. This is a side effect of the fact that pattern matching, unlike other language features like list and conditionals, are not yet overloadable in Haskell [31, 23].

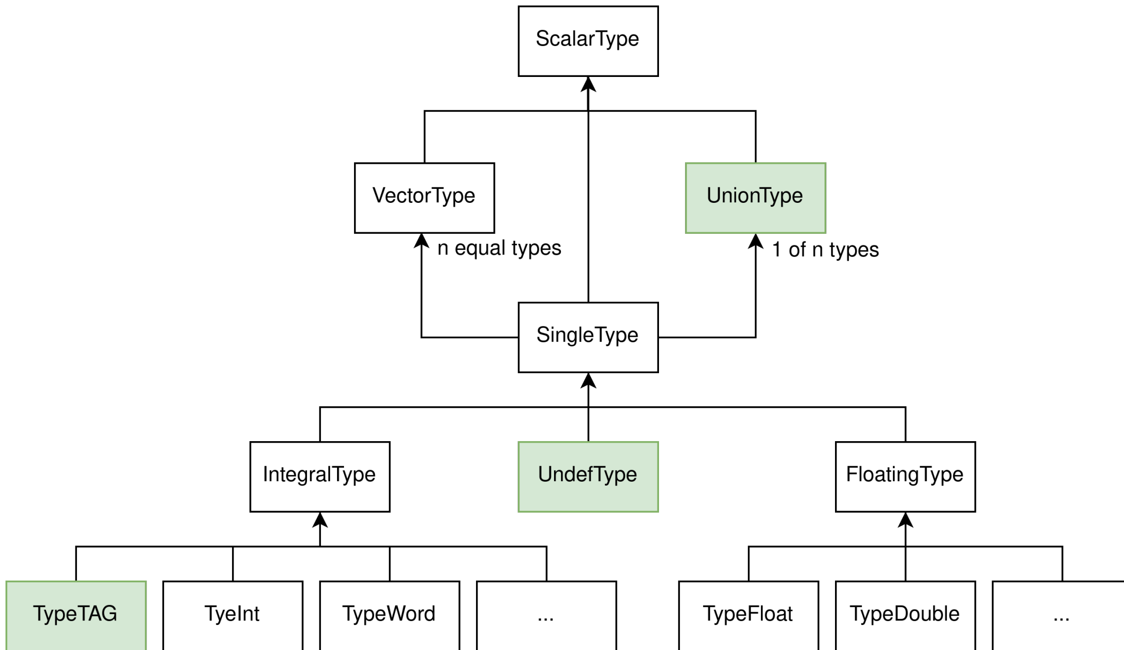

Figure 7: *The type hierarchy of scalar types in Accelerate. Annotated in green are the additions made to integrate POSable.*

```
class Matchable a where
  type SOPCode a :: [[Type]]

  build :: KnownNat n => Proxy n -> NP Exp (SOPCode a !! n) -> Exp a

  match :: KnownNat n => Proxy n -> Exp a -> Maybe (NP Exp (SOPCode a !! n))
```

Listing 31: *The `Matchable` class. The associated `SOPCode` type is an alias for `Code`, from the generics-sop library. To keep the `Matchable` class open for implementation, the `SOP.Generics` class is not a superclass of `Matchable`, making aliasing `Code` necessary. The `!!` type operator is the type level equivalent of the `!!` operator in plain Haskell, that is, indexing in a list.*

In order to implement this class generically, the functions on type lists and `Finite`'s have to be lifted to the AST. For example, the `combineSum` and `combineProduct` function from the *Finite* library have been reimplemented as a function on `Exp`s. Because this implementation relies on existing AST constructors, the backends do not need to understand `Finite`. Type functions like `Concat` and `Merge` need to be lifted to tuple lists instead of type lists. The current version of the implementation supports `Bool`, `Either` and `Maybe`, the latter being polymorphic types. This gives confidence that a generic implementation is possible, although this is not yet finished due to time constraints.

5.4 Changes to the AST

To make use of union types in the AST, both creating and destructing unions should be possible. Next to those, coercions of these unions also have to be possible, as both pattern matching and applying constructors change the possible types in a union. The following constructors have been added to the AST: `LiftUnion`, `Union` and `PrjUnion`. `LiftUnion` lifts a scalar type to a singleton union type. `Union` coerces between unions of different types. `PrjUnion` extracts a type from a singleton union. All three of those operations change the type (interpretation) of some value, but do not change the underlying value. This means that all of these operations can be implemented

as casts in the backends. These casts should be type safe by construction of the AST.

The `Match` and `Case` expressions have been updated to match a range of tags instead of a single tag. This is because a shallow pattern match can match multiple concrete tags. Take for example the type `Either (Maybe Int) Float`. A shallow pattern match on `Left` should match on both `Left Nothing` and `Left (Just _)`. These have tags 0 and 1 respectively. This means we have to match on a range of tags. The `Match` and `Case` expressions have been changed accordingly.

5.5 Limitations

Although the Recursive Tagged Union representation is memory efficient, there are reasons it might not always be the right choice for product types. When pattern matching on a sum type, rebuilding the tags is a fairly cheap operation, only consisting out of a comparison and the minus operator (see listing 24). Rebuilding the tags in a product type however is an operation build from the modulo and divisions operators (see listing 25), which are relatively expensive. Another problem is that the amount of possible values in a tag of a product type with nested sum types grows much faster than that of a sum with nested products. For example, a sum of four quadruples needs a two-bit tag. A quadruple of 4-sums however needs an eight-bit tag instead. This makes that tags wider than a byte might be needed for larger data types, especially those that contain sums inside products.

There are two simple solutions to these problems. The first is to not represent pure product types with a single tag, but with separate tags for each nested sum. This is not unlike the current representation in Accelerate. If this approach is chosen, tuples should not derive `POSable`, but instead a separate implementation should be provided.

The second problem can be simply solved by adding larger tag sizes to Accelerate. The amount of choices in a type is stored in the `Choices` type of the `POSable` class. The size of the tag in bits is the log_2 of this value. This value can be statically known, and can be converted to the right tag type, whether it is a byte, int16, int32 or int64. This however does not solve the first problem of expensive pattern matches. More involved solutions to this problem are explained in section 6.2.

6 Future work

In this section, we identify areas of improvement and future research. Most importantly, we describe how the implementation in Accelerate can be finished and which possible optimizations can be implemented in the compiler.

6.1 Implementation in Accelerate

The integration of the POSable library in Accelerate has not been finished due to time constraints. The remaining work consists of two parts: finishing the pattern matching code and updating the backends.

To finish the pattern matching code, the generic implementation of the `Matchable` class has to be finished. Next to that, a bit of Template Haskell should be written to generically create the pattern synonyms, the implementation of which can use the `Matchable` class.

Accelerate code can be executed on multiple different platforms. To make this possible, it supports multiple backends, which compile the AST down to the target hardware instructions. To accommodate the AST changes, these backends, including the interpreter, have to updated. Implementation of the new `LiftUnion`, `Union` and `PrjUnion` constructors can be done in terms of typecasts. To implement the updated, ranged, `Case` constructor the generated code needs to perform two comparisons instead of a single equality check. Backends should preferably implement a ranged pattern match containing just a single value as a simple equality check.

6.2 Optimizations

Pattern matching on unified tags comes with a drawback: the tags of embedded sum types have to be rebuild after a pattern match. Although building tags and dissecting tags of sum types are a fairly cheap operations (consisting of applications of $+$, $-$ and $*$), dissecting tags of product types involves the more expensive *mod* en *div* operations. Luckily, nested pattern matches can often be optimized into a single pattern match, making rebuilding tags unnecessary. For example, consider matching on the type `Maybe (Float, Maybe Int)`. Matching on this type means doing two, nested, pattern matches. The first pattern match, on the outer `Maybe`, can bring in scope a `Float`, which does not have a tag that needs to be rebuild. It then also brings into scope a value of type `Maybe Int`. Another pattern match is needed to bring the inner `Int` into scope. There are three scenarios possible:

- The value of type `Maybe Int` is not used. This can be figured out by doing a variable usage analysis. The tag does not need to be rebuild in this case.
- A pattern match is done on the inner `Maybe`. The pattern match is done irrespective of the value of the `Float`. In this case, the pattern matches can be combined. Since the combined pattern match does only reveal singleton types, no tags needs to be rebuild. This also means that each pattern does not cover a range of tags, but instead the match can be performed as simple equality checks.
- The pattern match is done conditionally, depending on the value of the `Float`. In this uncommon case, the pattern matches cannot be combined, and rebuilding tags is necessary.

To support the first and second scenario, an extra optimization step has to be build in the compiler. With this optimization applied, matching on a combined tag is faster than matching on separate tags, because less memory accesses are needed. In the third, uncommon, case, the performance hit is unavoidable when using a combined tag.

Another solution to this problem would be to not store the combined tag of a product as a index in a cartesian product. Instead each of the nested tags of a product can be stored strictly next to each other in the product tag. This means that the tag of the first nested sum is stored in the first n bits, the tag of the second nested sum in the next m bits, et cetera. For this, the width of the tag of each nested sum has to be rounded up to a whole number. Instead of *mod* and *div*, cheap bitmasking operations can then be used to get the tags out. These operations can be build from the binary *and* and binary $shift$ operations.

Which of the proposed solutions is preferable performance-wise is only knowable by running a set of benchmarks on the different implementations. These benchmarks can also verify the performance impact of using the Recursive Tagged Union representation.

6.3 Sorting types

In section 3, the Sorted fields approach, proposed by Troels Hendriksen, is briefly discussed. This approach can be combined with the Recursive Tagged Union approach, to do an even better job in terms of memory usage. This can be achieved by adding the size (in bytes) of a data type to the `GroundType` class, and sorting on this, on the type level. Type-level sort is not an operation that can be performed easily in current Haskell, which leaves this approach open for the future.

6.4 Fighting type erasure

To be able to construct a value of a type that depends on a type list or type natural number, it is useful to know something about this type list or type natural number at runtime. Haskell however erases types at runtime, which means this is not as simple as calling a `typeof` function. There are two ways to bring type knowledge in scope statically. The first way is using type classes. Consider a type class that has a separate implementation for the empty list type and the non-empty list type. In functions of a type class, this knowledge about the type is in scope, and can thus be used. The other possibility is to bring the type in scope by passing a helper value. Pattern matching on this helper value then brings into scope the type associated to that constructor.

Both ways are used in the POSable library. For the type level naturals that represent `Choices`, the `KnownNat` type class is used. By using the `KnownNatSolver`, this constraint is brought in scope automatically for most simple use cases (see also section 2.5). For the type level lists however, helper values are used. The type lists are represented by the `ProductType` and `SumType` types. These helper values can be summoned by the `emptyFields` function.

Using constraints for the type lists would have been possible, but this would come with a huge drawback. The type class would have to be a superclass of the `POSable` class. This makes deriving a lot less ergonomic, as shown in listing 32.

```
deriving instance
  ( POSable l
  , POSable r
  , All SListI (Merge (Fields l ++ '[]) (Fields r ++ '[]))
  ) => POSable (Either l r)
```

Listing 32: *Deriving an adapted version of the `POSable` class that has a dependency on the `SListI` class, a type class used to 'pattern match' on type lists. The example shows that this exposes the innards of the `POSable` class.*

The `POSable` class however does have a dependency on the `KnownNat` class, without the same drawback. This is due to the `KnownNatSolver`. A similar solver for type lists would make the POSable library a lot more compact. This would be achieved by removing need for the `ProductType` and `SumType` types, and all functions that handle these types.

7 Conclusion

This thesis answers the research question:

> *How can we efficiently represent sum data types in memory, in the context of data-parallel array applications?*

In order to answer this question, we have compared approaches of sum type representations in different languages. We have identified the strengths and weaknesses of each approach in the context of a struct-of-arrays layout, the typical memory layout for a parallel array languages. Based on these strengths and weaknesses, we have presented our own representation, the Recursive Tagged Union representation. This layout uses a single tag to represent the choices in a data type, and a product of unions to compactly represent its fields. By calculating the minimal memory usage of a set of types, and comparing the different representations, we show that this representation is efficient.

We have implemented Recursive Tagged Union representation in the *POSable* library. This library represents the layout in the `POSable` class. The default implementation of this class generically transforms any non-recursive Haskell 98 data type to the representation. This makes the class user-derivable, while the default can be overridden by the user. The library calculates this layout at the type level, which means this information is available statically.

The usage of this library in a parallel array language has been demonstrated by a (partial) integration in Accelerate, a parallel array language embedded in Haskell. The changes involve adding unions as a type to the AST and the internal type hierarchy. Tags are also handled differently, as some mathematical operators have to be applied to tags now. The implementation shows that the Recursive Tagged Union approach is feasible to implement in a parallel array language.

Altogether, these results answer our research question.

8 References

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi:10.1109/IEEESTD.2019.8766229.

[2] Christiaan Baaij. ghc-typelits-knownnat, 2021. URL: https://github.com/clash-lang/ghc-typelits-knownnat.

[3] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming. In *Advanced Functional Programming*, pages 28–115, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/10704973_2.

[4] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 1–13, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1040305.1040306.

[5] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011*, pages 3–14. ACM, 2011. doi:10.1145/1926354.1926358.

[6] Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. Embedding foreign code. In *Practical Aspects of Declarative Languages - 16th International Symposium, PADL 2014, San Diego, CA, USA, January 20-21, 2014. Proceedings*, volume 8324 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 2014. doi:10.1007/978-3-319-04132-2_10.

[7] Edsko de Vries and Andres Löh. True sums of products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, WGP '14, page 83–94, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2633628.2633634.

[8] Trevor McDonell et al. accelerate-ray. URL: https://github.com/AccelerateHS/accelerate-examples/tree/master/examples/ray.

[9] Trevor McDonell et al. A stable fluid simulation. URL: https://github.com/AccelerateHS/accelerate-examples/tree/master/examples/fluid.

[10] Troels Henriksen. Futhark 0.12.1 released, August 21, 2019. URL: https://futhark-lang.org/blog/2019-08-21-futhark-0.12.1-released.html.

[11] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. *SIGPLAN Not.*, 52(6):556–571, jun 2017. doi:10.1145/3140587.3062354.

[12] Rick van Hoef. POSable, 2022. URL: https://github.com/Riscky/posable.

[13] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196–es, dec 1996. doi:10.1145/242224.242477.

[14] Intel. Intel® AVX-512 instructions, 2013. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html.

[15] Simon L Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Conference on Functional Programming Languages and Computer Architecture*, pages 636–666. Springer, 1991.

[16] Scott Kilpatrick, Derek Dreyer, Simon Peyton Jones, and Simon Marlow. Backpack: Retrofitting Haskell with interfaces. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 19–31, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2535838.2535884.

[17] Felix Klein. finite: Finite ranges via types. URL: https://hackage.haskell.org/package/finite.

[18] Frederik M. Madsen, Robert Clifton-Everest, Manuel M. T. Chakravarty, and Gabriele Keller. Functional array streams. In *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing*, FHPC 2015, page 23–34, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2808091.2808094.

[19] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, page 37–48, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1863523.1863529.

[20] Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. Faster laziness using dynamic pointer tagging. *Acm sigplan notices*, 42(9):277–288, 2007. doi:10.1145/1291151.1291194.

[21] Trevor L. McDonell, Manuel M. T. Chakravarty, Vinod Grover, and Ryan R. Newton. Type-safe runtime code generation: accelerate to LLVM. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 201–212. ACM, 2015. doi:10.1145/2804302.2804313.

[22] Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 49–60. ACM, 2013. doi:10.1145/2500365.2500595.

[23] Trevor L. McDonell, Joshua D. Meredith, and Gabriele Keller. Embedded pattern matching, 2021. doi:10.48550/ARXIV.2108.13114.

[24] Simon Peyton Jones, Tobias Dammers, Ryan Scott, Ömer Sinan Ağacan, Ben Gamari, and Takenobu Tani. Unpacked sum types. URL: https://gitlab.haskell.org/ghc/ghc/-/wikis/unpacked-sum-types.

[25] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. *SIGPLAN Not.*, 41(9):50–61, sep 2006. doi:10.1145/1160074.1159811.

[26] Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. Pattern synonyms. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, page 80–91, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976002.2976013.

[27] Rust by example: Enums. URL: https://doc.rust-lang.org/stable/rust-by-example/custom_types/enum.html.

[28] Robert Schenck. Sum types in Futhark, 2019. URL: https://futhark-lang.org/student-projects/robert-msc-thesis.pdf.

[29] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, page 51–62, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1411204.1411215.

[30] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, page 53–66, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2103786.2103795.

[31] David Young, Mark Grebe, and Andy Gill. On adding pattern matching to Haskell-based deeply embedded domain specific languages. In *International Symposium on Practical Aspects of Declarative Languages*, pages 20–36. Springer, 2021. doi:10.1007/978-3-030-67438-0_2.

A Calculations: memory usage of different representations of sum types.

We have to make some assumptions to be able to compare different representations of sum types. The first assumption is that tags are saved in a single byte. This is plenty of space to represent the choices of the types analyzed here. Using more than a byte might be necessary for larger types. Using less than a byte to represent a tag is not feasible, as indexing on most hardware is limited to byte offsets. The second assumption is that pointers are 64bit wide, which is correct for most contemporary hardware.

We use data types for which the width is clearly defined, like `Float` and `Double` [1]. We then get the following calculations:

Tag and pointer

When this representation is used in a struct-of-arrays approach, arrays have to created of which the length is only known at runtime. Instead of using a pointer, an index into this array could be used, the width of which could depend on the array length. The array length is a variable we don't want to take into account here. We assume the pointer to be 64 bits wide, which is the common pointer width on current hardware. Random access into an array is a rather expensive operation on a GPU, making this representation suboptimal for other reasons than the memory usage.

$Float?$: 8 bit tag + 64 bit pointer + optionally a 32 bit Float = 72 or 104 bits

$Float + Float$: 8 bit tag + 64 bit pointer + 32 bit Float = 104 bits

$Float + Double$: 8 bit tag + 64 bit pointer + either 64 bit Double or 32 bit Float = 104 or 136 bits

$Float \times Double + Double \times Float$: 8 bit tag + 64 bit pointer + 96 bit data (Double + Float) = 168 bits

$Double? + Double?$: 8 bit tag + 64 bit pointer for the Either + 8 bit tag + 64 bit pointer for the Maybe + optionally 64 bit for the Double = 144 or 208 bits

Unboxed

$Float?$: 8 bit tag + 32 bit Float = 40 bit

$Float + Float$: 8 bit tag + 2 times 32 bit Float = 72 bit

$Float + Double$: 8 bit tag + 64 bit Double + 32 bit Float = 104 bit

$Float \times Double + Double \times Float$: 3 times 8 bit tag + 2 times 96 bit data (Float + Double) = 216 bit

$Double? + Double?$: 3 times 8 bit tag + 2 times 64 bit Double = 152 bit

Tagged Union

$Float?$: 8 bit tag + 32 bit Float = 40 bit

$Float + Float$: 8 bit tag + 32 bit union of Float and Float = 40 bit

$Float + Double$: 8 bit tag + 64 bit union of Double and Float = 72 bit

$Float \times Double + Double \times Float$: 8 bit tag + 2 times 64 bit union of Double and Float = 136 bit

$Double? + Double?$: 8 bit tag + 8 bit union of tags + 64 bit union of Doubles = 80 bit

Sorted Fields

$Float?$**:** 8 bit tag + 32 bit Float = 40 bit

$Float + Float$**:** 8 bit tag + 32 bit Float = 40 bit

$Float + Double$**:** 8 bit tag + 32 bit Float + 64 bit Double = 104 bit

$Float \times Double + Double \times Float$**:** 8 bit tag + 64 bit Double + 32 bit Float = 104 bit

$Double? + Double?$**:** 2 times 8 bit tag + 64 bit Double = 80 bit

Recursive Tagged Union

$Float?$**:** 8 bit tag + 32 bit Float = 40 bit

$Float + Float$**:** 8 bit tag + 32 bit union of Float and Float = 40 bit

$Float + Double$**:** 8 bit tag + 64 bit union of Double and Float = 72 bit

$Float \times Double + Double \times Float$**:** 8 bit tag + 2 times 64 bit union of Double and Float = 136 bit

$Double? + Double?$**:** 8 bit tag + 64 bit union of Doubles = 72 bit