Accelerating Structural Sum Types

Luuk de Graaf

December 2023

Contents

1 Introduction

Data-parallel array languages can operate on a higher abstraction level by distributing instructions over multiple elements. A data-parallel `map` and `fold` function is sufficient to cover a wide range of high-performance applications. A naive intersection algorithm for a raytracer can be defined incredibly concisely compared to a manually vectorized c++ implementation.

```
nearest :: Ray -> Objects -> Distance
nearest ray = fold min 1e30f . map (intersect ray)
```

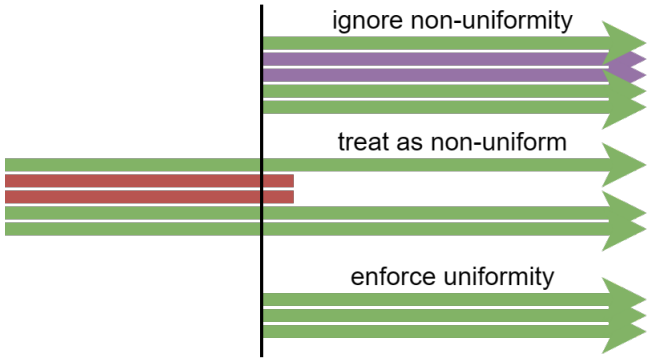
```
float minDistance = 1e30f;
for (int i = 0; i < objects.Length; i++)
{
    float distance = intersect(ray, objects[i]);
    minDistance = min(minDistance, distance);
}
```

An obvious limitation is the lack of low-level control, which is apparent

2 Introduction

Abstractions in programming languages simplify code and allow for implicit exploitation of arising properties. Pure functional array languages utilize the independence between computations to implicitly parallelize array operations. This is complicated in languages that are not inherently thread-safe. Many high-performance applications adopt a data-oriented design, which centres around data transformations and avoiding abstractions around data. Both parallelization opportunities and cache efficiency are focal points of the data-oriented design paradigm. A challenge with data-oriented design is representing non-uniform data efficiently while preserving vectorization opportunities and type safety.

The issue can be illustrated through raytracing, an embarrassingly parallel problem with several instances of non-uniform data. The main instance occurs when recursively intersecting geometry. The exception state of a ray not intersection with any geometry is propagated along, which prevents any vectorization attempts of subsequent rays. In practice this is solved by either unconditionally executing subsequent rays or reordering the rays for each recursive step. These considerations also exist outside exception states, such as handling multiple shapes of geometry efficiently in data-parallel context.



The approaches are all similar in intent, to iterate on a variant of a type, but can have significant runtime performance implications. An ergonomic way to switch between these approaches can be essential for deducing the most performant option through benchmarking. An efficient and type-safe abstraction that encapsulates these approaches is non-trivial, as it involves both the value and type representation. This is apparent when attempting to define functions that can operate on both value-level variants (tagged unions) and type-level variants (types). Unification of these concepts would reformulate operating on non-uniform data to an optimization problem. Within this paper we propose a way to ergonomically switch between efficient representations of non-uniform data. We preserve type-safety by elevating the concept of a mutual exclusive datatype to the type-level, which is achieved through type-level and datatype-generic programming. An implementation can exist without it being natively supported as construct in the implementation language. Which can be utilized by libraries, frameworks and embedded domain-specific languages that exist in languages that facilitate type-level programming and datatype-generic programming, like Haskell.

2.1 Related Work

Initial objective was establishing a computational efficient representations for non-uniform data in data-parallel applications. This induced the need for a flexible and type-safe interface, which has been achieved through type-level and datatype generic programming. Relevance is established by an implementation in the data-parallel language Accelerate, which is deeply embedded within Haskell.

Performance The memory representation of a datatype is often based on the functionality they perform within a language. In data-parallel applications primitive types are distributed over multiple arrays to facilitate vectorization. It is not apparent what the representation of a tagged union should be, as they inherently break vectorization in most cases. The functional data-parallel languages Accelerate[30] and Futhark[27] implicitly distribute primitive types in composite datatypes over multiple arrays. Both implementations have limited deduplication capabilities, but research has been done to integrate a memory efficient tagged union in Accelerate[30]. Game-engines, which deal with many clusters of data, have a fundamentally different approach as they have widely adopted the Entity-Component-System (ECS) pattern. Many implementations incite a collective re-organization of the internal representation when a variant change occurs at runtime. A type-safe and performant implementation is notably hard due to having to statically resolve all interactions between the representations, which means meta-programming and untyped code are extremely prevalent.

Interface Functional languages handle tagged unions safely through Algebraic Data Types (ADTs), where sum types categorize ADTs with multiple variants. Constructors can be local to an unique ADT (nominally typed) or exist as independent types (structurally typed). Deconstructing is done by pattern matching, where functions natively branch on the current active variant. In Haskell sum types are nominally typed, which means variants are not standalone types and cannot exist safely outside the ADT. Structural sum types are often called extensible or open sum types, as they do not have to be explicitly declared before use. In OCaml these are natively supported as polymorphic variants. In Haskell an efficient internal representation can be derived statically through associated types[6]. Datatype generic functions, which depend on the structure of a datatype, can be used to create isomorphic mappings between the computed representations. Both concepts are used in highly generic libraries for a wide-range of applications[25].

Implementation Libraries and domain-specific languages that are embedded construct their user-defined-types through the host-language. A shallow embedding operates directly on types native to the host-language, while deep embeddings construct an abstract syntax tree that is later evaluated. The latter offers flexibility on how user-defined types are implemented, as types exist both on the surface level and as construct within the abstract syntax tree. There are several approaches, which have been subject to research in the domain of circuit design.

- CλaSH is not deeply embedded and operates on user-defined types through generics[2].
- Hydra has the deep embedding constructs nested into the shallow embedding constructs[24].
- Kansas Lava has both embeddings exist in parallel under an encasing type[14].
- ForSyDe has both embeddings exist separately as standalone types[26].

An observation is that user extendability is limited on deeply embedded constructs as execution models must be updated. A proposed approach is to have a small deeply embedded core language that only supports constructs that are relevant for combinatorial optimizations[29]. The shallow embedding can be used to create an extensible and user-friendly interface to this core language. In the context of data-parallel applications and non-uniform data this leans itself to an implementation in the host-language.

3 Performance

There are many components that can influence the performance of a program. This grows the importance of being able to identify *bottlenecks* but also understanding the underlying technological performance considerations[16]. Within the first section fundamental optimizations related to the interaction between data and hardware are introduced. This is used to identify architecture agnostic performance considerations for array operations in the second section.

3.1 Optimizations

In the early days of computing, memory was seen as a way to store data indefinitely. As computational power of processors increased, the importance of main memory increased. Main memory is dependant on the advancements of random-access-memory (RAM), which stagnated due to both cost and physical limitations[8]. This put pressure on the software side to adapt to hardware components for optimizations, rather than merely the computational complexity of algorithms.

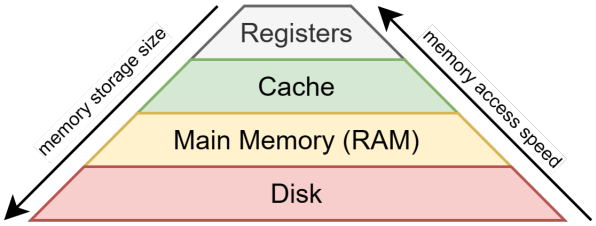


Figure 1: memory hierarchy

One of these hardware optimizations is cache storage, which accelerates memory accesses of predetermined data. The data is decided based on a cache replacement policy, often based on a temporal property. The cache operates independently of the operating system[8], and no direct control can be exercised.

Instructions Interfacing with processors is done through computer instructions. Fetching of an instruction is a memory operation, as it retrieves the instruction at the target of the program counter. Instructions operate on registers, which have distinct sizes depending on the architecture and their respective function. There are many instruction set architectures (ISA) and devices that implement distinct instruction sets. An intermediate representation (IR), such as the LLVM IR[19], can be used to create a uniform interface between these instruction sets[7]. Explicit use of these architecture exclusive instructions can be achieved through compiler intrinsics. Hardware design sometimes allows for executing specialized instructions[1], which are faster than their semantically equivalent instruction(s). This includes sacrificing accuracy for performance (floating-point), combining a sequence of instructions (arithmetic) or by parallel execution on multiple data elements (SIMD). SIMD instructions in particular are often very performant, as several steps within the execution pipeline can be parallelized. This process of instruction parallelization is called *vectorization*.

[1]Note that the term *complex instruction* is avoided, as this concerns a compact *representation* of several instructions.

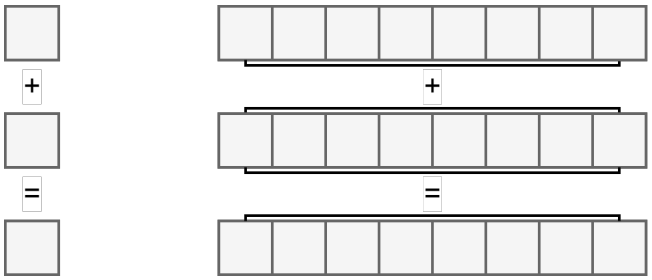


Figure 2: vectorization of a scalar addition operation

Register Pressure Registers can be considered the fastest available memory, as the data is ready to be used by an instruction. Within the context of registers this data is commonly referred to as a variable. Variables existing within registers before execution is a prerequisite for reasonable performance. This scheduling problem is to be considered a NP-complete problem[5]. Programming languages with any form of abstraction delegate this process to the compiler. This simplifies a lot of complexity, as only which data is being used by what instruction is relevant. In some cases there are too many live variables for the available registers, which *spills* the variable. This requires a variable to be stored outside registers, in a slower form of memory, and incites a delay upon use. This can be prevented by reducing the live-time of variables, reordering instructions and diversifying execution units.

Memory Access Time Semantically random-access-memory (RAM) implies that memory operations take around the same amount of time. In practice this does not hold for several reasons.

SRAM/DRAM On a modern system there often exist several different types of RAM, mostly driven by cost differences. The main forms of RAM are static RAM (SRAM) and dynamic RAM (DRAM). SRAM uses six transistors to represent a single bit, while DRAM only uses one transistor with a single capacitor. A capacitor loses electrons over time which means data has to be refreshed repeatedly to preserve its data. A refresh requires both read and write operations, which interferes with other memory operations. This makes DRAM inconsistent and on average significantly slower but much cheaper to produce due to requiring less transistors.[8]

Propagation Data is transferred by using electrical charges through semi-conductors. This creates a physical limitation dictated by physical distance and temperature. This is called propagation delay and a hard limitation to the rate at which components can operate on. SRAM is often located physically closer to the execution units to utilize the faster memory access more effectively.

DMA A processing unit needs to forward the requested data to the targeted location, which takes up processing time. Direct-memory-access (DMA) is an interface for hardware components and allows memory operations to be more organized. This allows for large scale memory operations to be performed efficiently and independently of the main processor. It requires use of several buses which means some processors must idle at seemingly random periods of time. This means that other hardware components can influence the memory access time.

Caching Due to hardware related discrepancies in memory access time, it can be beneficial to organize data according to the memory access time. One way to achieve this is by caching data, that is storing a *copy* of the data in faster accessible medium. A cache is generally made of SRAM and resides close to the processor, which allows memory accesses to be magnitudes faster than the equivalent main memory access[8]. When data already exist in the cache it is referred to as a *cache hit*, otherwise a *cache miss*. Deciding which data is cached and for how long is a cache replacement policy. Adapting to these policies simplifies the scheduling and minimizes the amount of cache misses.

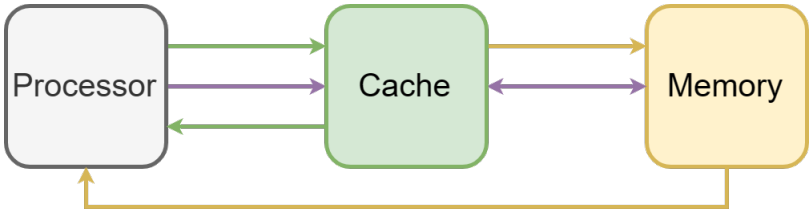


Figure 3: cache hit (green), cache miss (yellow) and replacement (purple)

Caching of data can be done after the data has been retrieved, which means the delay already has occurred. This can be avoided by requesting data in advance and storing it a cache prematurely, so called cache prefetching. This is done by analyzing future instructions (hardware) and instructions that *hint* at the future use of data (software)[3].

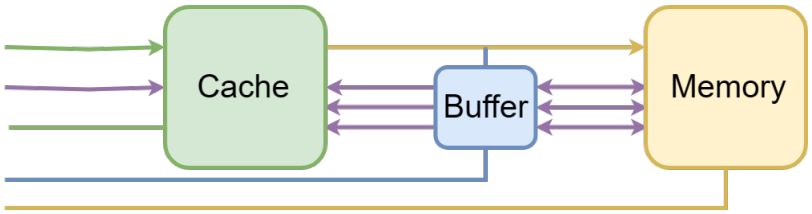


Figure 4: prefetch based on information (blue) from cache miss and processor instructions

This is harder when a branch is encountered, as both the data and the next instructions are uncertain. Speculatively executing these uncertain instructions can be performant if the overhead of redundant work remains small enough. Rather than executing unconditionally, some processors execute the most likely to happen branch based on some parameters (branch prediction)[28].

Parallelism Instruction-level parallelism is the parallel execution of multiple instructions[28]. This can be done by dividing instructions into several steps and outsourcing each step to a distinct processor unit (instruction pipelining). Shuffling the order of instructions can allow more processor units to work in parallel (out-of-order execution). Duplicate units and independent instructions allows for additional parallelism (superscalar execution).

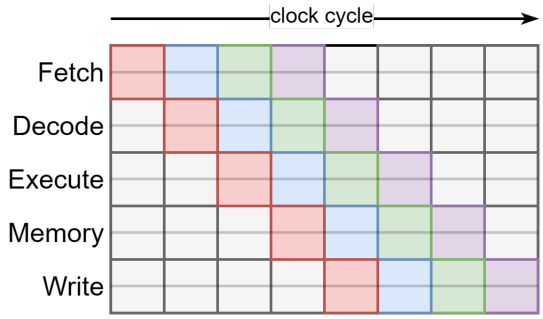


Figure 5: instruction pipelining: each color represents an instruction

Data-level parallelism executes an instruction on several data elements, such as the previously discussed SIMD instructions. Specialized processors sometimes either fully pipeline the data (vector processing) or allow for some form of autonomy (multithreading). Both share instruction fetching and decoding, but threads have their own program counter which allows for an independent sequence of instructions.

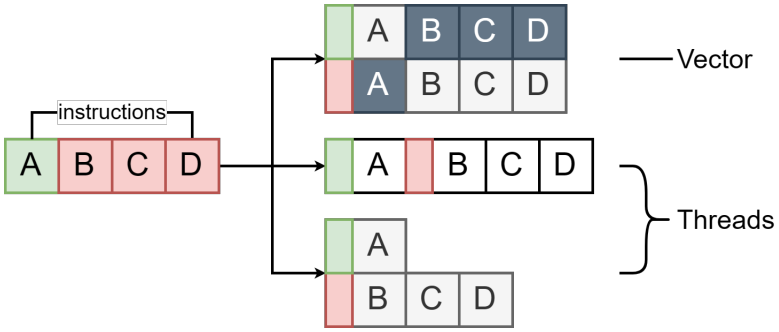


Figure 6: branch instructions: masking (vector) vs independent sequence (threads)

Execution of threads can be done concurrently, which can be useful to hide latencies (context-switching). In parallel is also possible with multi-core processors, which have several processors units (cores) that can support multiple threads. Cores are often not independent processors and might share several components with other cores, such as caches[20].

3.2 Principles

There are many components that can influence the performance of a program, some of which were discussed in previous sections. This makes general statements on optimizations often weak, as the interaction between these components is complex. Focusing on a particular area, such as iterating on data elements, allows for stronger arguments. Within this section previously discussed optimizations will be discussed in the context of iterating on many elements.

Contiguous A rudimentary reason for contiguously allocated data is that it creates structure, which can be used to organize data. Arrays utilize their structure to align elements, such that each element can be identified in constant time[2] through a linear function. This is also used for compound datatypes, where structure and the type can identify the memory location of each field. The structure also simplifies work distribution between threads, as it is a matter of constant offsets. For vectorization contiguous data is a prerequisite as instructions operate on singular contiguous blocks of data. If data is not spatial adjacent in memory, data must aligned temporarily or complex interleaving methods must be used[22]. In the general case compound datatypes interfere with vectorization, as spatial adjacent data is not of the same type. Parallel arrays solve this by creating a distinct array for each primitive type (Struct of Array) so that each field can be vectorized.

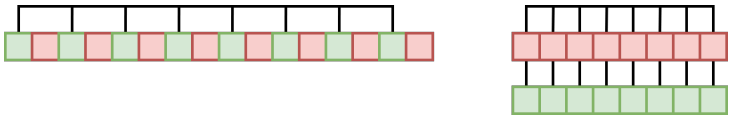


Figure 7: compound datatype array (1) and a parallel array (2)

Caches also operate with contiguous blocks of memory, which means spatial adjacent data within a fixed alignment are stored together. This lends itself well to contiguous allocated data, as it means the least amount of cache blocks are required irrespective of block size and alignment. In addition all memory accesses use the same linear function, a *constant stride* access behavior, which makes it receptive to hardware cache prefetching[3].

Access Patterns As the cache is finite a cache block can be ejected prematurely. This exists for data within the same block but also when the same block is required at multiple times. Increasing temporal locality is done by avoiding random accesses and organizing computations order around data usage. This is non-trivial in iterations where multiple indices are accessed (stencils) or computations that inherently involve random access.

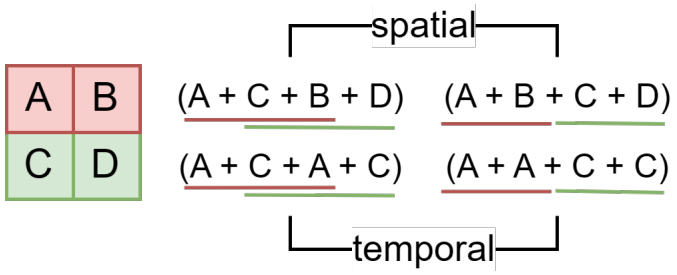


Figure 8: exploitation of spatial and temporal locality

[2]Both in *time complexity* and within *computer architecture* norms, as data access is a single instruction, unlike pointer trees and hash tables.

One way to apply this to iterating on many elements, is to iterate one subset of elements at a time (tiling). This can be further improved by also accounting for shared resources, by grouping elements that use the same resource in their instruction sequence. This is explored in raytracing[1], where spatial locality of rays is used to exploit the cache coherence within the traversal of a tree. These techniques are also important for multi-core processors, as it reduces the need for data to exist in multiple caches.

Branching Pipelining instructions is not possible when the sequence of instructions is dependant on the result of a previous instruction. This limits instruction-level parallelism, which is solved through various unconditional instruction executions[28]. Either by discarding the computed results or by *flushing* the pipeline when the wrong branch is predicted, both of which intuitively have an overhead. A compiler can eliminate[3] branches or move loop-invariant code to facilitate instruction-level parallelism[11]. These optimizations are not absolute, as an increase in instructions can pressure registers and the cache. It is also limited to instructions that cannot fail or overflow, as both can introduce unintentional side-effects.

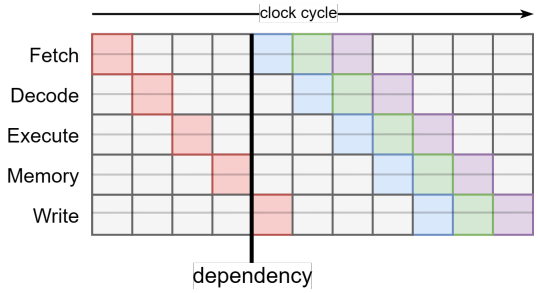


Figure 9: branch introduces dependency on execution of instruction (red)

Branching is also problematic for vectorization, as all data within a certain alignment must follow the same sequence of instructions. This can be resolved through the use of a *bitmask*, which can nullify parts of a result[11]. The additional instructions and computing bit masks can prevents performance from vectorization in certain situations. A notable application is sequential loops, where unrolling creates an opportunity to vectorize the scalar instructions. Automatic vectorization is an active field of research, and limitations have been primarily attributed to the lack of analysis information available to compilers[9]. This means branchless code and simplifying control flow allows the compiler to vectorize in more instances.

Specialized processors where an instruction sequence is distributed over many cores are limited to executing all branches. This is minimized through the use of Streaming Multiprocessors (SM), which contains several cores and fetch their own instructions. Streaming Multiprocessors operate and schedule warps, which often contain 32 threads. When divergence between these threads occurs (*branch divergence*) the instructions will in the general case be executed in lockstep[23].

[3]Either by proofing the branch will never be executed or by replacing the branch with a *conditional move* instruction, which only writes the result on true.

3.3 Data structures

A fundamental aspect of computing is data structures, which is a constant overhead for all computations. For collective operations arrays are essential; as they have a constant access time, are contiguously allocated and access can be parallelized. Composite datatypes within arrays introduce some considerations. One is the *implicit* use of parallel arrays, where each primitive datatype is stored in a distinct array. This enables vectorization opportunities, but a random access pattern might cause additional cache blocks to be cycled between. Since collective operations control the access pattern, parallel arrays are often a natural choice for array languages. The consideration for both structurally and functionally distinct data, now referred to as variant, is often complex. Variants can be represented on an individual basis (element-wise) or collectively (variant-wise). Usage and implementations of these approaches are explored in this chapter. Within this chapter the assumption is made that parallel arrays are used, as they align with the intention to vectorize operations. The example composite datatype has type A, and either has type B or type C.

3.3.1 Element-wise

For each element the choice of variant is represented, which introduces branching and in the general case will break vectorization. As variants are not grouped, functions cannot iterate on a specific variant without iterating on the complete array. The main advantage is that a variant change can be done independent of other elements, and thus can be parallelized. A practical consideration is that each element in an array must be structurally the same, that is they occupy the same memory space. This is a limitation which enforces that each index can determine the location of an element. For parallel arrays this restriction applies for all arrays individually[30].

Tagged Union Multiple variants can be represented through a tag and a fixed size data component with multiple representations, so called *union*. The tag is used to identify the current representation of the union. A naive implementation creates an array for each field of each variant, which means the memory usage is cumulative for each variant. A compact tagged union overlaps fields of variants, as only one representation can be valid at a time. This can in theory reduce the size to the largest variant and the accompanied tag, but this is complicated due to alignment requirements[30].

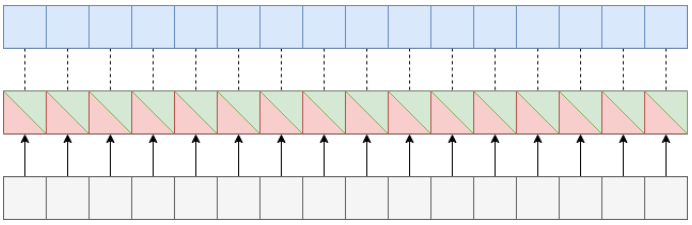


Figure 10: index implicit and tag (gray) identify the data representation

Tagged Pointer Another way to comply with elements being structurally the same is to use a form of indirection, in this case a pointer to memory. The indirection allows variants to escape the uniform size restriction, but there are several notable complications. General complications around pointers, such as being unsafe to operate on and complicating garbage collection apply. In addition, pointers that point to the same data (alias) can prevent parallelization due to possible race conditions. These can be partly solved through language constructs; such as smart pointers, immutable data or abstracting the use of pointers altogether.

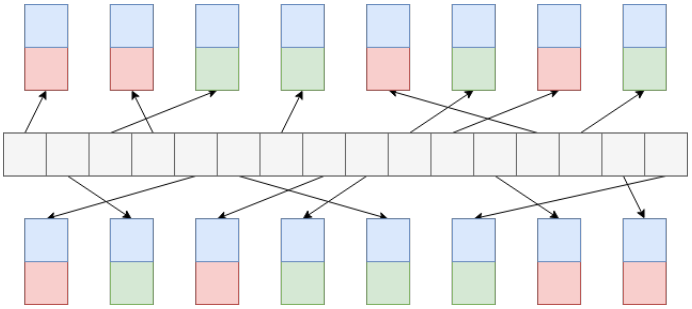
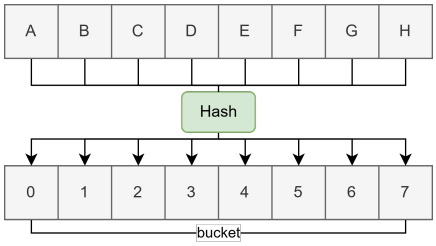


Figure 11: tag and pointer (gray) identify the location and representation.

The key issue is that a change in variant requires new data to be allocated and the pointer to be adjusted. This allocation means there is no guarantee that the data is contiguous, which in addition to the required branching prevents any vectorization efforts. The indirection and fragmented memory is also problematic for cache efficiency, as it is unpredictable and a cache block is not used effectively.

Entity A notable observation is that the re-allocation of a variant change causes the data to be not contiguous, not the indirection in itself. This can be illustrated through a hash table data structure, where a key is mapped to a value within an array (bucket). Any collective operation on the hash table can be vectorized by disregarding the hashing and using the internal array directly, as computations are inherently independent and order is irrelevant.



Entities within the ECS pattern function similarly, a form of indirection that is not used by the collective operations. Represented as a single heterogeneous array, which internally consists of several variant-wise collections. It will mean that variant choice is not *directly* represented on an element basis, which has several implications.

Stable The same entity is not guaranteed to refer to the same data, the entity is no longer stable across structural changes. The reverse also holds, the data is not guaranteed to have the same entity along iterations. This can solved by tracking the location of entities *or* annotating the data with their entity. These approaches can be complementary for performance reasons, but they are collectively isomorphic[4] through gather and scatter operations.

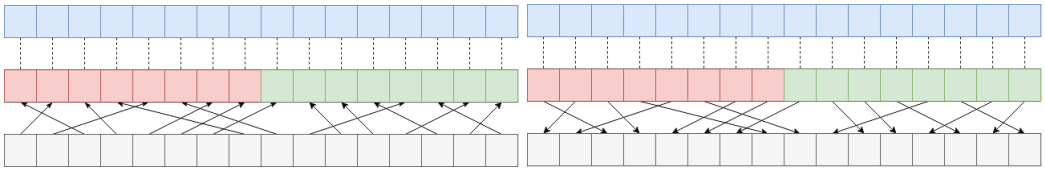


Figure 12: persistent array with indirection (1) or annotate the data with their index (2).

For many array operations stableness is excessive, as it means data is discriminated on the basis of their index. The implicit connection that data with the same index has can be represented through a (temporary) datatype.

Independent Elements can no longer change their variant independently of other elements, which can be problematic for parallelism. Depending on the way variants are grouped, there can also be a significant cost associated with regrouping variants. This can be minimized by delaying structural changes indefinitely, by using a tagged union approach. Regrouping variants is a performance consideration between the cost of regrouping and having to branch for future iterations.

[4]A single entity cannot identify the data in constant time, without an array of pointers. A data element cannot identify the entity in constant time, without the entity as data.

3.3.2 Variant-wise

Grouping variants means all data is uniform, contiguously allocated and there exist no inherit branching within the same grouping of variants. This can be achieved through an array for each variant, but also grouping *within* the same array and using segment descriptors. The latter is effectively an untagged union, where the representation is determined by the index within the array. Both allow instructions to be vectorized, but there exist several other considerations.

grouping As stated in the previous section, regrouping variants to a variant-wise collection is a performance consideration. When variants are stored in separate arrays, the amount of a certain variant must be known before allocation. When this is dependant on a computation, it can be retrieved through an additional scan or atomically[5] counting any structural change, which adds an overhead. This is not required for a singular array if the total remains the same.

immutable An important consideration for purely functional languages is that values, and therefore arrays, are to be considered immutable. This means that *updating* parts of an array efficiently is non-trivial. It must be proven that the array before update will never be used again, otherwise both arrays must co-exist in memory. This is inefficient for small updates and grows the necessity to *destructively update*[18].

automatic Most compilers support automatic vectorization of iterations with flexible bounds, where the final leftover iteration is not vectorized. This overhead can be a significant when the loop is extensively unrolled. This is minimized through epilogue vectorization, which (re-)applies loop vectorization to the remaining scalar code. In practice data must be aligned along specific byte boundaries to be vectorized, which is challenging for (dynamic) regions within an array and not always analyzed by compilers[9].

operable An undiscussed benefit of parallel arrays is that fields can be operated on independent of other fields, as they are distinct arrays. This is also possible for *regions* within an array, but this is less trivial and often requires explicit support in array languages[10].

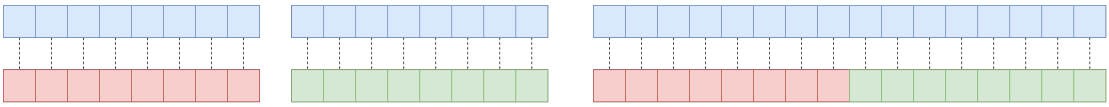


Figure 13: distinct arrays (1) or distributed in a single array (2)

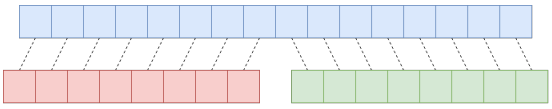


Figure 14: combination of both approaches

[5] Atomic instructions prevent interruptions by other processes and are thread-safe.

3.4 Memory Representation

Memory instructions operate on fixed boundaries, which means operations that overlap these boundaries require additional but strictly unnecessary instructions. A natural alignment of a datatype is achieved by aligning all types according to the instructions that access them. Many compilers introduce *padding* to enforce natural alignment for all the types within the structure. While computational efficient, the alternative of *packing* types together can be preferred for a smaller memory footprint.

```
struct PackedData // 8 bytes
{
    char  name;         // 1 byte
    int   node;         // 4 bytes
    short identifier;   // 2 bytes
    byte  alignment[1]; // 1 byte
};

struct PaddedData // 12 bytes
{
    char  name;         // 1 byte
    byte  padding[3];   // 3 bytes
    int   node;         // 4 bytes
    short identifier;   // 2 bytes
    byte  alignment[2]; // 2 bytes
};
```

Parallel arrays (Struct of Arrays) have a natural alignment by default as they contain primitives types, which are naturally aligned. General purpose languages often default to the Array of Struct, while data-parallel applications use the Struct of Arrays representation. A zero-cost abstraction that can ergonomically switch between these constructs is non-trivial. An interface to index access with an intermediate structure can break automatic vectorization[17]. In addition the different internal representations must be statically definable and able to be handled by the data structures. Many C++ libraries utilize *class templates* to achieve this[17].

4 Interface

A preliminary conclusion of the previous chapter is that a performant internal representation cannot be deducted from a mere theoretical framework. With this in mind it is important for high performance oriented applications to be flexible with the internal representation of datatypes. This is important for both composite data types and data structures. Within this chapter a modular interface is explored around iterating on collections of non-uniform data.

4.1 Paradigm

As discussed in the data structures section, there are two internal representations for collections of non-uniform data. Variants of a particular type are stored either element-wise or variant-wise. There are several considerations for a data structure that is agnostic to if it stores non-uniform data in element-wise or a variant-wise way. A variant-wise collection can operate on only the relevant variants, while an element-wise is forced to discover that at runtime. To avoid redundant iterations for variant-wise collections, a function must be able to be defined on the most specific subset of all variants. For element-wise collection the amount of variants must be finite and no large discrepancies can exist between the size of all the variants.

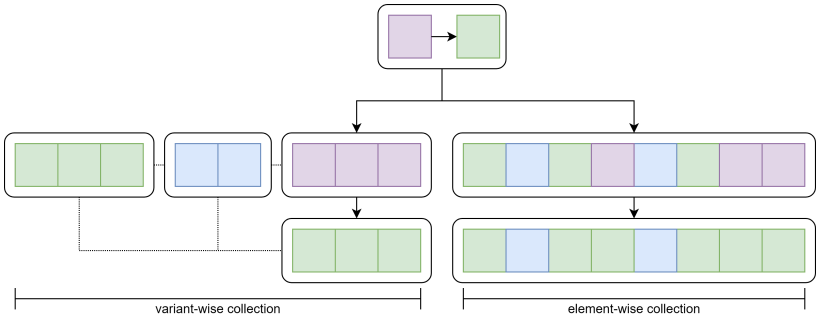


Figure 15: A non-exhaustive function can skip a significant amount of work when non-uniform data is stored in a variant-wise collection, while this is problematic for an element-wise collection.

For variant-wise collections the ECS-pattern will be used as reference. The first section goes into how the ECS-pattern collectively operates on only some variants. For element-wise collections Algebraic Data Types (ADTs) are used to safely discriminate between multiple variants as element. The second section utilizes structurally typed ADTs to create an efficient interface for a variant-wise and element-wise collection.

4.1.1 Entity-Component-System

The ECS pattern is arguably a reaction to the prevalence of object-oriented languages within game engines. The premise is to organizes game-logic within functions rather than data, where the relation between data is flexible[21]. In contrast to inheritance, where relations are statically determined and game-logic is embedded within an predetermined hierarchy. The pattern is often combined with data-oriented design and is used to be able to implicitly exploit data-parallelism in general purpose languages. While there exist many implementations of the pattern in many languages, the principles remain similar.

Components A component is the smallest addressable type within the pattern. Examples of components are `Position` and `Velocity`, which together represent movement. Components are generally value types to avoid race conditions when using data-parallelism. Some implementations allow for a (readonly) reference component that is shared along multiple instances. A shared `Mesh` component prevents redundant geometry to be stored by referencing it.

Entity An entity is idiomatically a set of components. A movable entity has the `Position` and `Velocity` components, while an immovable entity only has the `Position` component. Adding and removing components is done at runtime and there is no predetermined relation between any of the components. Any immovable entity can be made movable by attaching the `Velocity` component at runtime.

Systems A system operates on all entities that match a specific set of components. The `Movement` system will operate on all movable entities, irrespective of any other attached components. Systems are effectively global functions, which operate only on specific variants of the more general entity type. This allows for a variant-wise collection of entities, which most ECS implementations enforce to implicitly create performant vectorized code.

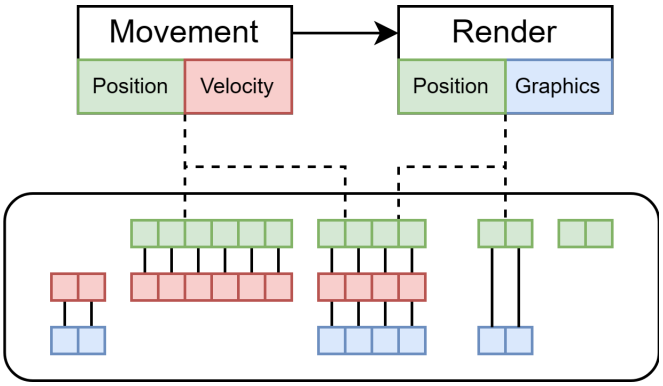


Figure 16: conceptual entity-component-system representation of systems

The ECS design pattern is arguably inherently imperative, as structural changes to entities are done imperatively. Apecs, an ECS library in Haskell, achieves this imperative style through monads[4]. The ECS pattern makes element-wise collections infeasible, as there is no type-safe way to restrict the possible amount of structural changes to an entity. This is inherit to the pattern, as any component can be attached to any entity. Implementations circumvent this limitation by allowing components to be enabled and disabled through a tag. Disabling a component changes the *type* of an entity but the internal representation remains the same. An analogy can be made to a non-deduplicated tagged union that is explicit in the variants it holds.

4.1.2 Algebraic Data Type

Functional languages handle tagged unions safely through algebraic data types (ADTs). A product type (x) is the combination of datatypes, while a sum type (+) is the alternation between datatypes. It is often useful to discriminate between variations of a datatype, which is generally done through a data constructor.

```
data Maybe a = Just a | Nothing
```

Deconstructing an algebraic data type is done by pattern matching on a data constructor. The pattern match can exhaustively match on all variants, as the data constructors of variants are known at compile-time. It can be seen as a native control-flow mechanism that ensures only the operations on the active variant are applied.

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap f (Just a) = Just (f a)
fmap f Nothing  = Nothing
```

In Haskell, algebraic data types are distinguishable by name (nominally typed) and therefore explicitly declared. This means data constructors are local to the declared type and pattern matching happens within the same type. The type signature of the `fmap` function provides no information on the potential structural change of a datatype. It is possible to return `Nothing` for both cases[6]. A function that takes `Just a` and returns `Just b` ensures that the *collective identity* is preserved in a collective operation, irrespective of the data transformation. This exact definition is not possible due to being nominally typed, as `Just a` is not a type but a data constructor under the type `Maybe a`. In some cases, such as a safe division function, the introduced branching is inherit to the function and is now made explicit in the type definition.

```
fmap :: (a -> b) -> Just a -> Just b
fmap f (Just a) = Just (f a)

divide :: Int -> Int -> (Just Int | Nothing)
divide _ 0 = Nothing
divide n m = Just (n `div` m)
```

In this case a function is defined on the structure of a type, the data constructor of algebraic data types. `Maybe a` is now an alias for the mutually exclusive relationship `Just a | Nothing`, rather than a unique and standalone type. This generalizes variance to be between all types. OCaml calls these *polymorphic variants*[7], while other functional languages generally refer to them as *extensible* or *open sum types*. A motivating example is that a collection of `Maybe a` can be an element-wise collection or two variant-wise collections of `Just a` and `Nothing`[8]. While the former involves branching for `fmap`, the latter can ignore the `Nothing` collection and vectorize the `fmap` function. This flexibility aligns with the intention to create a modular system that is agnostic to the internal representation.

[6] `Just b` is not possible as it can only be inferred through `Just a` and the `a -> b` function.
[7] OCaml also implements nominally typed sum types, so called *variants*.
[8] As `Nothing` does not hold data, a size descriptor is sufficient.

4.2 Type-level programming

In the previous section an interface that is agnostic to the variant-wise and element-wise collection is proposed. This demands defining efficient intermediate internal representations for all possible variant combinations, which defeats the purpose of ergonomic switching between representations. Statically deriving an internal representation is impossible or restricted to predetermined parameters in most languages. Some high-performance libraries circumvent this restriction by meta-programming or custom data layouts[15]. A customizable and type-safe solution is type-level programming, which will be explored in this chapter.

4.2.1 Kinds

A value is categorized by types, while a type is categorized by *kinds*. This is relevant when discussing type constructors, where each constructor with a different arity has a distinguishable kind. A well known exposition of type constructors are parametric polymorphic data types, which take type variables as argument. While a lot of languages support polymorphic data types, the concept of kinds is not evident as only concrete types are able to be used for arguments. Haskell supports higher-kinded types, which are analogous to higher-order functions for types, which makes kinds apparent to the user.

```
2.5f         :: Float
Float        :: *
Option a     :: * -> *
Option Float :: *
Apply f x    :: (* -> *) -> * -> *
```

Type constructors can be used to encode data statically, such as Peano numbers. The parametric `Succ a` and `Nil` types are axioms that can be used to construct a natural number on the type-level. By default these exist in a open universe, which means ill-formed expressions can be created. On the type-level this can be resolved by Generalized Algebraic Data Types (GADTs), implemented in Haskell as an extension. It allows the type variables of constructors to diverge from the more general type.

```
// open universe where 'a' can be anything
data Succ a
data Nil

// closed universe under the phantom type 'a'
data Natural a where
    Succ :: Natural b -> Natural (Natural b)
    Nil  :: Natural ()
```

The consequence is that deconstructing a GADT will refine the type. This can be used to construct evidence of certain properties by pattern matching on data constructors. An observation is that this is a categorization of types, similar to how the kind `*` represents all concrete types. The `DataKind` extension promote types to the kind-level and constructors to the type-level. The kind `Natural` includes the types `Succ (a :: Natural)` and `Nil`. It both creates a closed universe and allows other type constructors to expect the more precise kind `Natural`. A limitation is that the construction of the type, which is needed for arithmetic operations, is guarded by the definition of `Natural`.

```
data Natural = Succ Natural | Nil | Add Natural Natural | Minus Natural Natural
```

On the value-level this is solved through functions that transform their input into another output type. Translated to the type-level it means type-level functions that transform an input kind into an output kind.

4.2.2 Type Family

A way to approach type-level functions is to see it as a type dependent on the instantiation of a type variable. This is akin to functions in type classes, where type-indexing allows functions to be overloaded. Haskell reuses this functionality for types, categorizable as *associated types*[6]. This is particularly useful for domain-specific languages, as an instance can have a specialized return type. Accelerate uses the `Elt` class to create a mapping between surface types and the internal representation.

```
class (Elt a) where
    type EltR a :: *
    toElt       :: a -> EltR a
    fromElt     :: EltR a -> a
```

While these integrate well with type classes, type families is the terminology for the standalone concept. A `data family` has unique types associated with the type, while a `type family` is merely the type synonym equivalent. In some cases this is insufficient to represent a function, as the type checker is unsure which instance to use. This is the case when the function has a more general default case which will always match. A closed type family attempts the instances in order of definition, which expresses itself in being able to pattern match on types.

```
type family Elem a (bs :: [b]) :: Bool where
    Elem x '[]       = False        -- no
    Elem x (x : ys)  = True         -- yes
    Elem x (y : ys)  = Elem x ys    -- no, but recurse
```

In this example the variable `y` can be `x`, which means the second and third instance are overlapping with each other. A closed type-family resolves this by attempting the more specific case of `x = x` first. An annoying limitation is that type families in Haskell cannot be partially applied, which means a lot of boilerplate is required to capture more complex functions. It cannot be partially applied due to partially applied type synonyms requiring higher-order unification, which is currently not supported in Haskell.

4.2.3 Interface

Type-level functions allow for computation of types, and as such a way to easily construct multiple representations for a single type. The process of constructing multiple representations can be captured within a single datatype.

```
data Variant (constructor :: [variant] -> *) (variants :: [variant])
```

The data type `Variant` takes two type variables, a higher-kinded construction type and a promoted list type. The constructor takes the promoted list and transforms it into a concrete type. As type families cannot be partially applied, a data family is used to create a constructor.

```
data family Constructor argument :: [variant] -> *

type V argument (variants :: [variant]) = V (Constructor argument) variants
```

An instance of the constructor data family constructs a unique type based on some type argument. An example variant type is `V Compact [Int, Float, Bool]`, where `Compact` is an (empty) descriptive datatype. The `Compact` type is associated with the constructed type within the data family.

4.2.4 Type Structure

With closed type families it is possible to derive compact layouts for multiple variants of a type. As a memory layout only concerns itself with the bit sizes of types, the intermediate structure will be the previously discussed natural number. The `DataKinds` extension natively supports the `Nat` kind with arithmetic expressions and literals for syntax. While mapping of primitive types to their corresponding natural number is trivial, this is not the case for user-defined datatypes.

```
type family BitSize (a :: *) :: Nat where
    BitSize Word8  = 8
    BitSize Custom = ?
```

It is not possible to statically derive the bit size of the `Custom` type within the type family. For this the types of which `Custom` is composed must be known to the type family. This means the structure of the type must be apparent to the type family. One way to approach this is to use a kind more specific then `*` that is explicit in the composition, such as the `Natural` kind but for all datatypes. Another way is to enforce an implicit constraint by ensuring the type can be constructed with a particular GADT. The latter is used by Accelerate where the `TupR` constructor ensures that the type is composed of only units, singles and pairs. The function `eltR` enforces this by requiring the associated type `EltR a` to have a mapping to the `TupR` type[9]. The GADT approach is preferable when access to the structure on the value-level is needed, which is the case for future datatype generic programming endeavors.

```
class (Elt a) where
    type EltR a :: *
    eltR        :: TupR (EltR a)

data TupR v where
    TupRunit   ::                      TupR ()
    TupRsingle :: a                 -> TupR a
    TupRpair   :: TupR a -> TupR b -> TupR (a, b)
```

A simple example to demonstrate the explicit structure is to convert tuples of `Word8` to a single type. In Accelerate each primitive type in a tuple is spread out over multiple arrays, which means this type-family enables an Array of Struct representation for such a tuple.

```
type family ToBitSize (a :: *) :: Nat where
    ToBitSize (a, b) = ToBitSize a + ToBitSize b
    ToBitSize ()     = 0
    ToBitSize Word8  = 8

type family FromBitSize (a :: Nat) :: * where
    FromBitSize 0    = ()
    FromBitSize 8    = Word8
    FromBitSize ...  = Word...
```

In Accelerate the embedded representation is the one relevant for optimizations. All type-functions therefore operate on the embedded representation, the type associated with `EltR`. It is impossible to construct such type outside the `Elt` class. A solution is to automatically derive the `EltR` class for a set of types with a fixed representation, such as tuples. A return type with a fixed representation means that all computed representations will implement the `Elt` class. It is now possible to ergonomically define many internal representations for embedded representations.

[9]Note this can circumvented by merely returning a bottom type such as `undefined`, which will only be detected when using the value.

4.2.5 Layout

While the tools are there to generically construct intermediate representations, it is not trivial to create a single performant solution. Within this section a modular interface is proposed which allows for ergonomic switching between internal representations of multiple variants. The most general intermediate structure of a composite datatype is a collection with the size in bits of each field in the datatype. This removes both hierarchy and type identity, which makes it is easier to reason about the layout of a datatype. The representation does preserve performance critical information about the way data is retrieved from the internal representation.

```
type family FieldSizes (a :: *) :: [Nat] where
    FieldSizes (a, b)   = FieldSizes a ++ FieldSizes b
    FieldSizes ()       = '[]
    FieldSizes a        = ToBitSize a : '[]
```

For simplicity the kind `[]`, the promoted list type, is used to denote all possible variants of a certain type. With closed type-families it is possible to define all relevant operations on lists. The definition of these are similar to their value-level counterparts without any partial application. With these operations it is possible to define a type-level union that creates a compact deduplicated union. The `FieldSizes` function returns a list of natural numbers, the size for each individual field in a tuple. The `BitSizeUnion` recursively adds the unique elements for each datatype, such that all fields map to a distinct element. This is achieved by removing the element from the comparison list once it has been matched.

```
type family Difference (a :: [r]) (b :: [r]) :: [r] where
    Difference '[]      bs = bs
    Difference (a : as) bs = a : Difference as (RemoveOne a bs)

type family BitSizeUnion (a :: [Nat]) (b :: [r]) :: [Nat] where
    BitSizeUnion xs  '[]      = xs
    BitSizeUnion xs  (y : ys) = BitSizeUnion (Difference xs (BitSizes (EltR y))) ys
```

While strictly speaking the representation is compact, it deduplicates when the bit size of the primitive type is of equal size. This is very safe, as there is no inherit performance cost to operating on types with the same size. This is not necessarily the case for types stored in a larger type, or even a type spread out over multiple types. In some memory-bound cases this approach can still be preferable. An efficient implementation requires type-level sorting, to avoid a scenario where the smallest type is inserted into the largest type. A naive sorting algorithm is quite trivial to implement, by inserting all elements into their respective position.

```
type family Sort (types :: [Nat]) :: [Nat] where
    Sort '[]      = '[]
    Sort (x : xs) = Insert x (Sort xs)
```

The derivation of such a compact layout is not particularly complex, as it is similar to the deduplication approach but with multiple stages. Each step increasing the perceived performance cost of the merge and avoiding a local optimum solution by unifying large datatypes first. The complexities of such layout exist with generically operating on such a layout.

4.3 Datatype-Generic programming

In the previous chapter we established a way to compute a wide-range of internal representations for a set of variants. It is not ergonomically viable to write insertion and extraction functions for all the different internal representations. As users can create their own representations there is no closed system, so mapping between all datatypes must be handled. This can be done through datatype-generic programming, which parametrizes on the composition of a datatype[13]. A mapping must be isomorphic, such that all information is preserved between construction and deconstruction of the union. This is not possible for all possible pairings, as the variant must be smaller or equal to the size of the representation. Within the first section a type-level way to prove valid pairings is explored, essential for supporting user-defined datatypes. The second section goes onto the theory of datatype-generic programming, which is used for an implementation of a datatype-generic framework in the third section.

4.3.1 Verifying

A variant must be smaller or equal to the representation. A variant that is larger than its representation loses information when constructing and deconstructing, which means type safety is breached. While it is possible to ensure that the computation of the representation is always larger than all the variants, this is a risky construction. It limits users extensibility and does not catch flaws within the type computation code. A modular implementation requires an independent method that ensures the variant is smaller than its representation. The `Constraint` kind can be used to restrict the construction to larger or equal types. Constraints occur on the left-hand side and are generally constructed through type-classes to enforce a general interface. A relevant example is ensuring that our newly computed representation has a mapping to the embedded representation.

```
class (Elt a) where
    type EltR a :: *

instance (Elt (constructor types)) => Elt (Variant constructor types) where
```

In the type-level programming chapter we already achieved a way to determine the size of any type as kind `Nat`. Fortunately the native `Nat` type already has several comparison operators with the kind `Nat -> Nat -> Constraint`. A simple but functional constraint is the `<=` operator. While it does omit performance considerations, this is required to support a wide-range of representations. The `IsVariant` class has a default implementation but can be extended by the user to support unsafe variants, such as sentinel values. The data-type generic programming part pertains to implementing the default `construct` and `destruct` functions.

```
class (Elt v, Elt t, BitSize v <= BitSize t) => IsVariant v t where

    construct :: Exp v -> Exp t
    construct = ...

    destruct  :: Exp t -> Exp v
    destruct = ...
```

4.3.2 Theory

Before looking into how `construct` and `destruct` are implemented concretely, it is essential to understand the problem datatype-generic programming is attempting to solve. Paradoxically variant types are best to illustrate the problem, coined the *expression problem*[31]. Extending the variants within the `Color` type means all functions that pattern match on `Color` must be changed. This can be resolved by having a general interface, but extending the behavior of this interface requires all types that implement the interface to change.

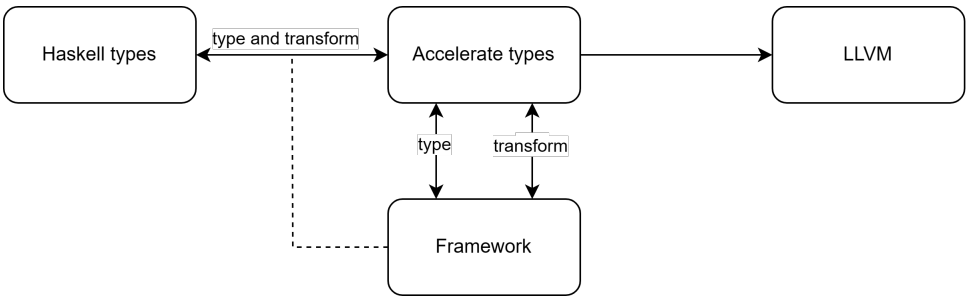
```
data Color = Red | Green | Blue

transform :: Color -> Color
transform Red   = ...
transform Green = ...
transform Blue  = ...
```

```
class ColorInterface a
    transform :: a -> a

instance ColorInterface Red where
    transform :: Red -> Red
    transform = ...
```

The impact of the *expression problem* can be minimized through several methods. Some argue that polymorphic variants fit within this category themselves, as it separates pattern matching with the underlying data[12]. In our case we have chosen for the representations to be extensible, which requires behavior to be defined for each possible representation. Both constructing and deconstructing the variant type into a specific variant. The possible datatypes is an infinite domain, which can be made finite by considering that all composite types can be reduced to primitive types. Datatype-generic programming utilizes the inherit concept of composition in programming languages to operate on any type. This is sufficient to generically operate on the multiple representations, as the semantics of the type are irrelevant for constructing and deconstructing the variant type. It requires access to the composition of a type, which is often not natively supported in programming languages. While Haskell does support datatype-generic programming, we do not operate directly on native Haskell types. An implementation through native Haskell types, which Accelerate currently uses for sum types, is restrictive. This is apparent when attempting to implement structurally typed sum types or more complex representations generically[30]. This is caused due to the `Elt` class requiring a direct mapping to the Haskell equivalent. The translation layer remains essential, but can be implemented with the help of the standalone implementation. Operating directly on embedded types results in a more targeted and adaptable implementation.



4.3.3 Framework

The composition of types is enforced through a GADT, as stated within the type-level programming section. As such a type can only have three cases; the empty type `()`, the primitive type `a` and the composed type `(a, b)`. An initial attempt would be to traverse the structure and apply a function to each primitive type. This means we need a function that discriminates between primitives types at the value level. This is possible in most embedded languages, as the abstract syntax tree itself is represented through a GADT.

Traversing A generic way to apply a function on each primitive type is hard to define. Each pattern match will refine the type further, which means we need a function that operates on multiple types. When passing a polymorphic function to a higher-order function it is not instantiated on the refined type, which means we cannot apply it. A solution is to explicitly limit the scope of type variables, such that it is local to the function.

```
traverse :: Monoid r => (forall v. Type v -> r) -> Structure e -> r
```

A generic traversal of a structure is extremely powerful and the first step to a datatype-generic implementation. The next step is traversing over the values of a structure, which is only a small step up. It simply includes an expression with the same type as the structure, which is also refined when pattern matching on the structure. This allows for functions to operate on fields within the datatype individually, but is restrictive in that it does not account for the hierarchy it exists in. A more involved traverse function makes available how a primitive type can be inserted and retrieved from the structure.

```
type Insert value expression   = value -> expression -> expression
type Retrieve value expression = expression -> value

traverse :: Monoid r => Structure a
                     -> (forall v. Type v -> Insert v e -> Retrieve v e -> r)
                     -> Insert a e
                     -> Retrieve a e
                     -> r
```

The `Retrieve` function recursively accumulates the further we traverse into the structure. The `Insert` function is slightly more involved, as it is recursive on the composed value. Normally this would just be the input expression, but we are operating on the structure and do not have access to the actual expression. Fortunately the `Retrieve` function is available to construct the expression to that point, which make the traversal function quite simple and compact.

Intermediate The traversal function is the foundation for operating on two structures, required to construct and deconstruct variant types. It is possible to create a mapping by traversing the other structure for each primitive value. This is both redundant and highly complex when removing values from the available mappings. The intermediate representation of a list, also used for computing the type representation, only requires a single traversal for each structure. This requires an heterogeneous list with all the different primitive types. Extensional types allow a type variable to be hidden, and thus they can be stored in a singe list.

```
data Field e = forall v. Field (Type v) (Insert v e) (Retrieve v e)

fields :: Structure e -> [Field e]
fields = traverse (\type insert retrieve -> [Field type insert retrieve])
```

Normally extensional types results in functions not being able to discriminate between the hidden types. As the primitive types exist within a GADT, pattern matching on `Type v` will reveal the type to the type system. A structure can now be constructed or destructed by folding over such a list, but more importantly the fields can be compared between structures.

Isomorphism An undiscussed constraint is that the `construct` and `destruct` functions must be isomorphic from each other. Both must map primitive types to the same fields, otherwise we cannot retrieve the same type from the representation. A simple way to achieve this is by creating both within the same function, such that all mapping decisions are inherently isomorphic. This is trivial with access to the `Field` type, as we have access to functions that insert and retrieve the value.

```
decisions :: forall v t. (Elt v, Elt t) => (Exp v -> Exp t, Exp t -> Exp v)
decisions = ...

construct :: forall v t. (Elt v, Elt t) => Exp v -> Exp t
construct = fst (decisions @v @t)

destruct :: forall v t. (Elt v, Elt t) => Exp t -> Exp v
destruct = snd (decisions @v @t)
```

It does mean that the constructor `v -> t` might make different decisions than the constructor `t -> v`, as decisions are made based from the perspective of one side. This is not a problem as constructed types must always be destructed first. The `t` type within the `decisions` function is allowed to have spare fields, as they are the undefined fields within an union.

Steps To avoid a local optimum in the representation several iterations must be done within the `decisions` function. A matching function determines whether there exist a mapping between two fields, and returns the functions that transform between the two primitive types. This makes the decision function extensible, as long as the user specifies the relation between two fields. Matching functions can be provided in order of preference. An implementations can in some cases be made more efficient by sorting before matching, but this reduces the generality of the function. Some cases might require a custom `decisions` function, such as inserting multiple fields into a single field. In general the implementation should be tailored to the implementation language.

5 Implementation

accelerate specific

6 Benchmarks

optional benchmarking of different options

7 Future work

8 Conclusion

References

[1] Daniel Meister 0002, Jakub Boksanský, Michael Guthe, and Jirí Bittner. On ray reordering techniques for faster gpu ray tracing. In Dan Casas, Eric Haines, Sheldon Andrews, Natalya Tatarchuk, and Zdravko Velinov, editors, *I3D '20: Symposium on Interactive 3D Graphics and Games, San Francisco, CA, USA, September 15-17, 2020*, pages 13:1–13:9. ACM, 2020.

[2] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. Clash: Structural descriptions of synchronous hardware using haskell, 2010.

[3] J. L. Baer and T. F. Chen. An effictive on-chip preloading scheme to reduce data access penalty. In *Supercomputing 1991*, pages 179–186, November 1991.

[4] Jonas Carpay. apecs: Fast entity-component-system library for game programming. `https://hackage.haskell.org/package/apecs`.

[5] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[6] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class, 2004.

[7] David Chisnall. The challenge of cross-language interoperability, 2013.

[8] U. Drepper. What every programmer should know about memory. *Red Hat, Inc*, 2007.

[9] Jing Ge Feng, Ye Ping He, and Qiu Ming Tao. Evaluation of compilers' capability of automatic vectorization based on source code analysis. *Scientific Programming*, 2021, 2021.

[10] Martijn Fleuren. Independently computed regions in a data parallel array language, 2020.

[11] A. Fog. Optimizing subroutines in assembly language: An optimization guide for x86 platforms, 2008.

[12] Jacques Garrigue. Code reuse through polymorphic variants, 2000.

[13] Jeremy Gibbons. Datatype generic programming, 2006.

[14] Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing kansas lava, 2010.

[15] Bernhard Manfred Gruber, Guilherme Amadio, Jakob Blomer, Alexander Matthes, Rene Widera, and Michael Bussmann. Llama: The low-level abstraction for memory access, 2022.

[16] Paul Hsieh. Programming optimization, 2016.

[17] Sylvain Jubertie, Ian Masliah, and Joel Falcou. Data layout and simd abstraction layers: Decoupling interfaces from implementations, 2018.

[18] Georgios Korfiatis, Michalis A. Papakyriakou, and Nikolaos Papaspyrou. A type and effect system for implementing functional arrays with destructive updates. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Federated Conference on Computer Science and Information Systems, FedCSIS 2011, Szczecin, Poland, 18-21 September 2011, Proceedings*, pages 879–886, 2011.

[19] Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[20] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.

[21] Adam Martin. Entity systems are the future of mmog development, 2007.

[22] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. *PLDI'06*, pages 132–143, June 2006.

[23] Nvidia. Nvidia tesla v100 gpu architecture.

[24] John T O'Donnell. Overview of hydra: A concurrent language for synchronous digital circuit design, 2002.

[25] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruna C. d. S. Oliveira. Comparing libraries for generic programming in haskell, 2008.

[26] Ingo Sander. System modeling and design refinment in forsyde, 2003.

[27] Robert Schenck. Sum types in futhark, 2019.

[28] Michael S. Schlansker, B. Ramakrishna Rau, Scott Mahlke, Vinod Kathail, Richard Johnson, Sadun Anik, and Santosh G. Abraham. Achieving high levels of instruction-level parallelism with reduced hardware complexity. Technical Report HPL-96-120, Hewlett–Packard Corporation, 2000.

[29] Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding of domain-specific-languages, 2015.

[30] Rick van Hoef. Accelerating sum types, 2022.

[31] Philip Wadler. The expression problem, 1998.