Institut für Programmstrukturen
und Datenorganisation (IPD)

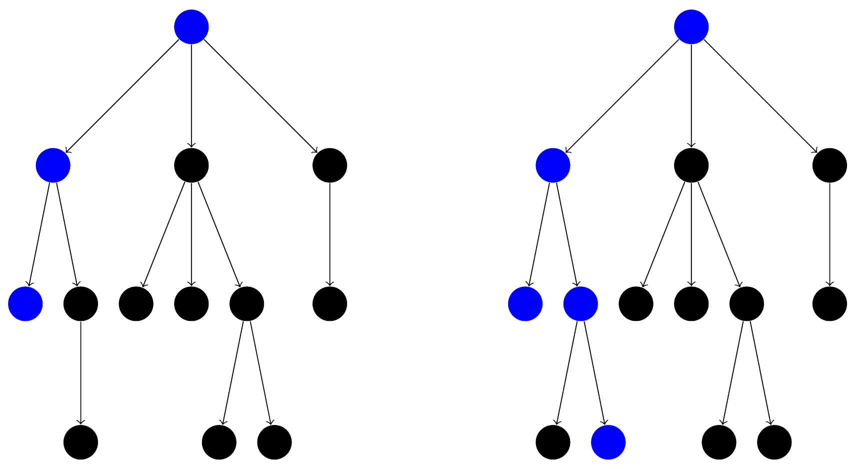
Lehrstuhl Prof. Dr.-Ing. Snelting

Transient Data Structures for Haskell

Bachelorarbeit von

Pascal Ellinger

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	M. Sc. Sebastian Graf
Abgabedatum:	7. November 2022

Zusammenfassung

Unveränderlichkeit ist eine Haupteigenschaft von Haskell, aber veränderbare Datenstrukturen können manchmal schneller modifiziert werden als unveränderbare. Transiente Datenstrukturen versuchen, die Vorteile von unveränderbaren und veränderbaren Datenstrukturen zu verbinden. Wir stellen eine Haskell-Bibliothek für transiente Dictionaries mit ganzzahligen Schlüsseln bereit. Unser transientes Dictionary verwenden wir in GHC, um zu zeigen, dass transiente Datenstrukturen in einer großen Haskell Codebasis die Leistung verbessern können.

Immutability is a main feature of Haskell, but updating mutable data structures can be sometimes faster than updating immutable ones. Transient data structures try to combine the benefits of immutable and mutable data structures. We provide a library for a transient dictionary with integer keys in Haskell. We use our transient dictionary in GHC to show that using transient data structures in large Haskell codebases can improve performance.

Contents

1 Introduction

Programmers not only expect their compilers to use lots of optimizations in order to generate fast code, they also expect compilers to generate that code quickly. The Glasgow Haskell Compiler [1] (GHC) is a Haskell compiler that is also in large parts written in Haskell.

As immutability is a main feature of Haskell, GHC also uses many immutable data structures. One of those immutable data structures used in GHC is a dictionary. Dictionaries map keys to values and a typical operation is the insertion of a key/value pair into the dictionary. As the dictionary is an immutable structure, the insert operation does not modify the dictionary it is given. Instead, the insert operation returns a new dictionary that contains the inserted key/value pair, leaving the original one unchanged. These dictionaries are internally trees. If this insert operation would have to copy the whole tree every time, this would be very inefficient. However, parts of the tree that the insertion does not change can be shared between the original and the new dictionary. Therefore, if one dictionary is passed to multiple functions, those functions can modify the dictionary independently from each other while parts of it are shared. That can be more efficient than cloning the whole dictionary in order to allow those functions independent modifications. Although sharing parts of an immutable structure is great when it is necessary, when it is not necessary, mutating a data structure in place still requires less copying and can therefore be more efficient.

Thus, both immutable and mutable data structures have their advantages and transient data structures try to combine those advantages. Transient data structures consist of a persistent variant and an ephemeral variant and fast conversions between them [2]. When an ephemeral data structure is modified, only the new version of the structure can be used and therefore ephemeral data structures can be mutable [3]. In contrast, persistent data structures allow accessing all versions, similar to immutable data structures [3]. With transient data structures, parts of the code that require persistence can use the persistent variant while performance critical parts that do not require persistence can use the ephemeral variant.

Our contributions are:

- We provide a Haskell library for transient dictionaries with integer keys.
- We explain how using transient data structures can be made safe with linear types.
- In order to show that using transient data structures in a language like Haskell improves performance, we replace an immutable data structure in GHC with

our transient one and evaluate the improvements.

- We discuss challenges that arise from the efficient use of transient data structures.

2 Transient Data Structures

2.1 Dictionaries with integer keys

One of our contributions is the implementation of a transient dictionary with integer keys. For this implementation we need a data structure that implements such a dictionary and is also a tree. The nodes of a transient dictionary sometimes have to be copied. Therefore, in order for the dictionary to be fast, the nodes should be fairly small.

A dictionary with integer keys can be implemented by storing the integer keys in a trie. However, the requirement of fairly small nodes is not fulfilled by a naive trie implementation because it potentially wastes a lot of space. Such a naive implementation splits the integer keys into chunks of $log_2(n)$ bits and treats these chunks as members of an alphabet of size n. Internal nodes of such a trie store a potentially very sparse array of child nodes of size n.

2.1.1 Array Mapped Tries

A more space efficient trie implementation is the Array Mapped Trie (AMT), which was popularised by Phil Bagwell's Hash Array Mapped Tries [4]. The following explanation is based on [4]. AMTs consist of internal nodes and leaves. The leaves contain the key/value pairs. An internal node maps the elements of an alphabet of fixed size n to the corresponding sub-tries. The internal nodes consist of an array of child nodes and a bitmap of size n. For $0 \leq i < n$ bit i in the bitmap indicates whether a corresponding child node exists. The number of set bits in the bitmap up to bit i is the position in the array of child nodes. In contrast to the naive implementation, which stores a pointer for elements of the alphabet that do not have a corresponding child node, AMTs only use one bit for such elements.

Figure 2.1 shows an example AMT with $n = 8$. It illustrates the search for the leaf corresponding to the bit string 010110. In order to search for that bit string, first the three most significant bits, 010, are considered. 010 is the binary representation of 2. The bitmap of the root node is 0001 0100 and therefore the bit at index 2 is set. As the bit is set, a corresponding child node exists. The number of set bits up to bit 2 is 0 because the bits at index 0 and index 1 are cleared. As this number is the position of the corresponding child in the array of child nodes, the position is also 0 and the search continues in the left child node. The bitmap of this node is 0011 0100 and the next three bits of the bit string are 110. As $110_b = 5_d$, the bit at index 5 is checked. Again, the bit is set. In the bitmap of the node the bits at

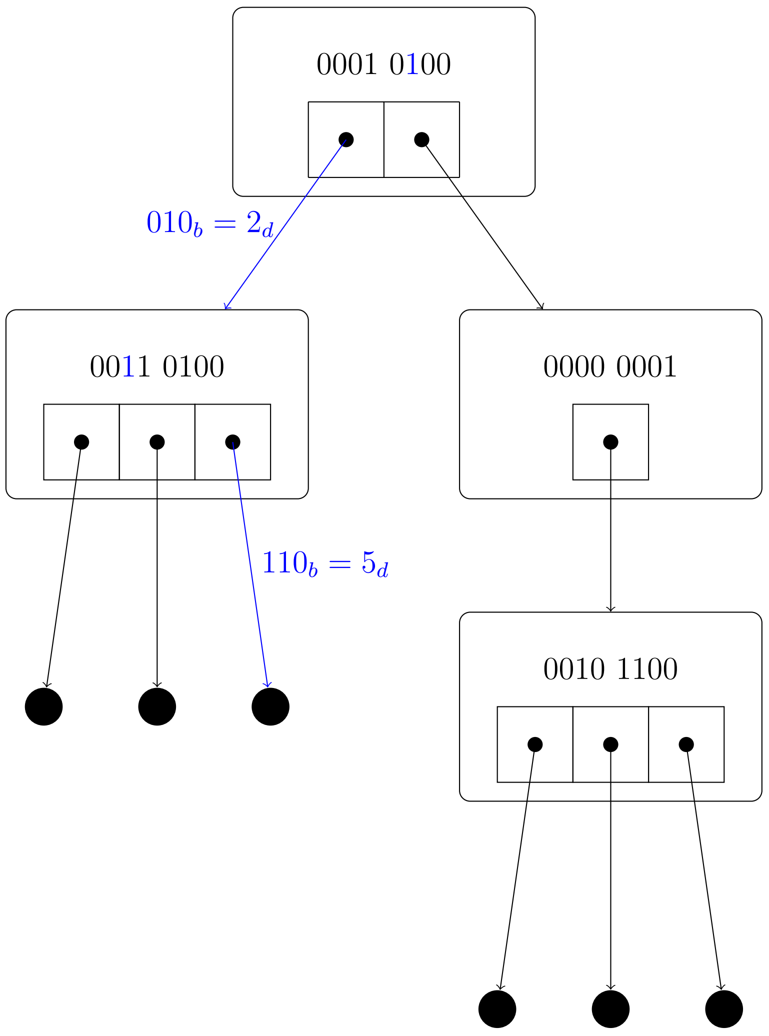


Figure 2.1: An example Array Mapped Trie.

indices 2 and 4 are set whereas the bits at indices 0, 1 and 3 are cleared. Therefore, the number of set bits up to bit 5 and the position in the child node array are 2. The child node at index 2 is a leaf.

As searching for a key should be fast, a search should only need to traverse few nodes. Because the internal nodes of an AMT can have only one child node, long chains of such nodes can exist. Potentially all of those nodes in the chain have to be traversed in a single search. An AMT with such a chain is shown in figure 2.2. When searching for the bit string 000 000 000 000 111, five internal nodes need to be traversed.

2.1.2 Compressed tries

Compressed tries, in contrast to AMTs, compress nodes with only one child into a single one [5]. Therefore, the long chains, which can exist in an AMT, are compressed, and a search in a compressed trie traverses significantly fewer nodes. Figure 2.3 shows a compressed version of the trie in figure 2.2. In the compressed version, the search for the bit string 000 000 000 000 111, only traverses two internal nodes, in contrast to the five internal nodes, which are traversed in case of the AMT.

The concept of compressed tries can also be applied to AMTs. An internal node of such an AMT, in addition to the bitmap and the child node array, stores the offset of the character that differentiates the child nodes and the prefix of the node. As an example, we explain the traversal of an internal node with the key 010 011 000 in an AMT with alphabet size 8. First, the key is shifted right by $offset + log_2(n)$ where n is the alphabet size and the result is compared to the prefix. With offset 3 and prefix 010, the key is shifted right by 6, the result is 010, which is equal to the prefix. If the prefix was not equal to the result, the AMT would not contain the key and the search could stop. As, in this case, they are equal, the chunk of bits at the offset, in this case $011_b = 3_d$ is used as an index into the bitmap. The rest of the traversal is the same as for the AMTs described in section 2.1.1.

2.2 Transient data structures

Transient data structures try to combine the benefits of immutable and mutable data structures. Immutability is an important feature of many functional programming languages. However, updating a mutable data structures can be faster than updating an immutable one. That is because updating an immutable data structure usually involves copying part of the structure, which is not necessary in the mutable case. For instance, when updating the value corresponding to the key 011 000 000 in the AMT shown in figure 2.4, the three nodes on the path to the leaf must be copied if the AMT is immutable. On the other hand, if the AMT is mutable, no node is copied and instead only the first element in the child array that contains the leaf is changed.

Transient data structures consist of an ephemeral variant, a persistent variant and

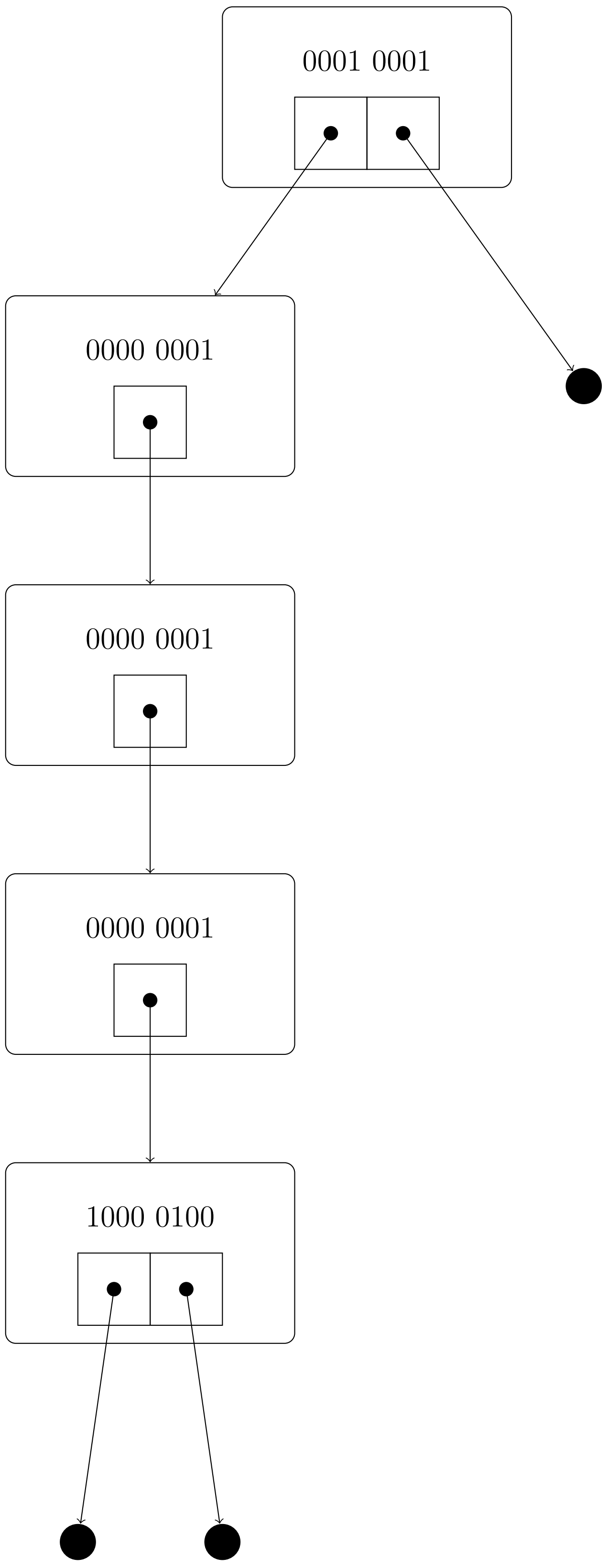


Figure 2.2: An AMT with a chain of internal nodes.

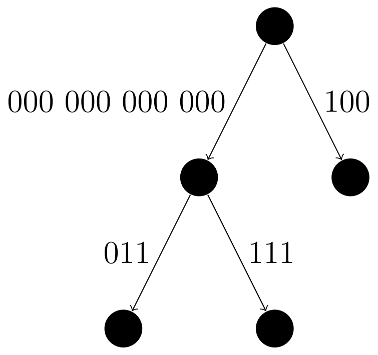


Figure 2.3: A compressed trie.

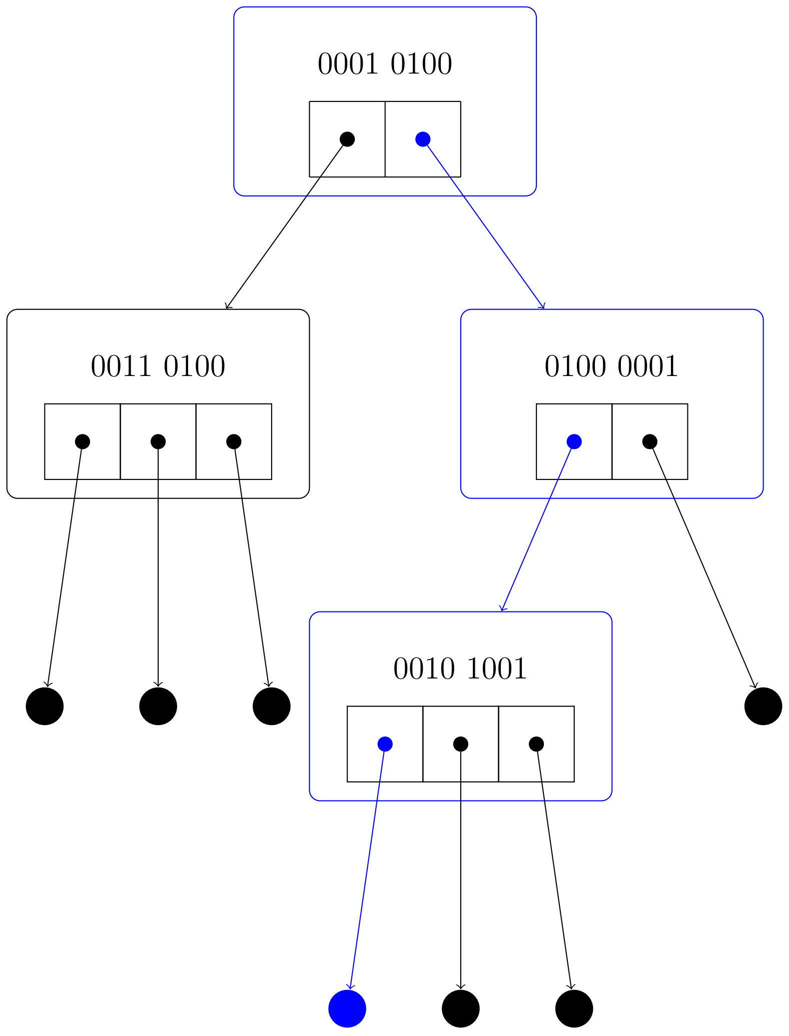


Figure 2.4: An example Array Mapped Trie.

fast conversions between them [2]. When an ephemeral data structure is modified, only the new version of the structure can be used [3]. The old version can no longer be accessed [3]. In contrast, a data structure is persistent if all versions of the structure can be accessed [3].

Ephemeral data structures can be mutable and modifications are therefore usually faster than modifications of persistent data structures. As a transient data structure supports fast conversions between ephemeral and persistent variants, performance critical code that modifies a transient data structure can convert a persistent variant into an ephemeral variant, modify it and convert the result back into the persistent variant.

In the following we will also refer to the ephemeral variant as the transient variant.

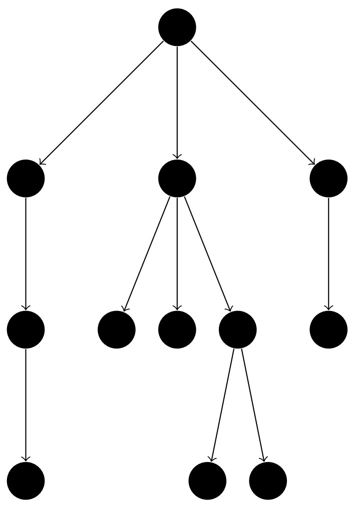
2.2.1 Transient data structures in Clojure

Transient data structures are popular in Clojure [6]. The following explanation is based on [6]. We explain the implementation of transient data structures in Clojure by looking at hash maps. The implementation is based on Hash Array Mapped Tries.

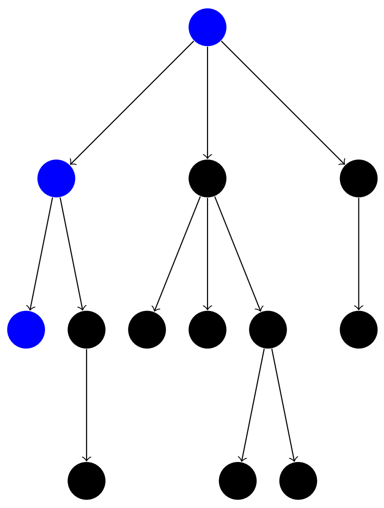
In order to turn such a hash map into a transient data structure, every ephemeral hash map gets a unique identifier and the nodes additionally contain the identifier of the ephemeral hash map in which they were created. A persistent hash map is turned into an ephemeral one by making a new unique identifier and creating an ephemeral hash map with this identifier and a reference to the root node of the persistent hash map. The nodes that have a different identifier than the ephemeral hash map are considered immutable. They are copied before they are modified. When a node is copied or created in the context of an ephemeral hash map it gets the identifier of the ephemeral hash map. Nodes that have the same identifier as the ephemeral hash map can be mutated in place. An ephemeral hash map is turned into a persistent one by throwing away the identifier. This ensures that shared nodes are never mutated and conversions happen in $O(1)$. Figure 2.5 shows two new nodes being inserted into a tree that only contains immutable nodes. The first insertion needs to copy all the nodes on the path from the root node and modify them. These nodes are now mutable. Therefore, the second insertion does not need to copy the root node and its child again.

2.2.2 Transient data structures in Haskell

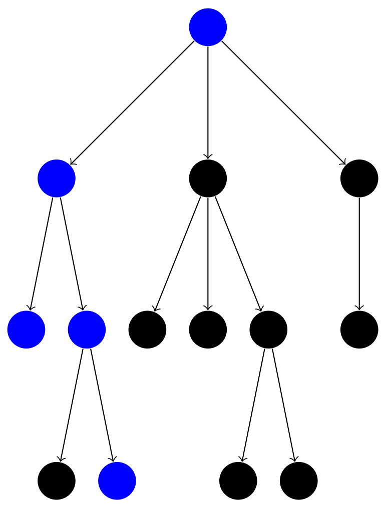
In Haskell most data structures are immutable. Transient data structures could be useful for replacing such existing immutable data structures in a library or an application in order to improve performance. In such a case, performance critical code could be adjusted to use the ephemeral variant and benefit from its mutability. However, changing code to use the ephemeral variant requires some effort because it must be made sure that only the most recent version of the data structure is accessed. Additionally, in Haskell, destructive updates are usually only allowed in monads like `ST`. Parts of the code that are not performance critical or require immutability of the



a A tree that only contains immutable nodes.



b After the first insert, the nodes on the path to the new node are mutable.



c The second insertion can modify mutable nodes in place.

Figure 2.5: The insertion of two nodes into a transient tree.

data structure could use the persistent variant. In contrast to the ephemeral variant, using the persistent variant would likely only require minimal code changes if the persistent variant can implement the same interface as the original immutable data structure.

2.2.3 Transient WordMaps

Edward Kmett started implementing a transient dictionary with integer keys, which he calls a `WordMap`, in Haskell [7]. However, as he never finished the project, we implement our own transient `WordMap` that is based on his work and available at `https://github.com/PascEll/haskell-transients`.

Kmetts transient WordMap

In the following we explain Kmetts implementation of the transient `WordMap` [7].

Kmetts implementation does not require an identifier per node like the Clojure implementation. Instead, each node only requires a bit indicating whether it may be mutated in place. The children of a node are stored in a `SmallArray`. As these `SmallArrays` already contain the information whether they are mutable, this information is also used to determine whether a node is mutable. In a persistent `WordMap` all nodes are immutable while a transient `TWordMap` can contain both mutable and immutable nodes.

Turning a persistent `WordMap` into a transient one is therefore simply a coercion. Turning a transient `TWordMap` into a persistent one relies on the invariant that all nodes that are reachable from an immutable node are also immutable. Thus, to persist a transient `TWordMap`, not every node in the tree must be traversed. Instead, first the root node is checked for mutability. If a node that is checked for mutability is mutable, then it is persisted and all of its child nodes are checked for mutability. However, if the checked node is immutable, all nodes reachable from it are guaranteed to also be immutable and therefore its children are not checked.

Saving space for the identifier per node comes at the cost that persisting a transient node happens only in amortized $O(1)$.

Differences between Kmetts and our WordMap

Both Kmetts `WordMap` and ours use compressed AMTs with $n = 16$ as described in section 2.1.2. However, Kmetts `WordMap` keeps a finger to the previous mutation, which our `WordMap` does not. In Kmetts implementation, when a node is inserted into an array of child nodes, a new array with the increased size is always created. In our implementation, we resize the arrays with a factor of 1.5 so that if the node is mutable and there is unused space in the array, the node can be inserted into the existing array.

The WordMap interface

The public interface of the `WordMap` module includes the persistent `WordMap` type as well as the transient `TWordMap` type. The typical operations on dictionaries like `lookup`, `insert` and `delete` operate on persistent `WordMaps`. Additionally, there are variants of these operations, `lookupT`, `insertT` and `deleteT`, that operate on transient `TWordMaps`. The `persistent` and `transient` functions convert `TWordMaps` into `WordMaps` and vice versa. `WordMaps` and `TWordMaps` can be constructed from lists with the `fromList` and `fromListT` functions. The `fromList` function uses `fromListT` internally, so it can benefit from the mutability of the `TWordMap` nodes.

Listing 2.1: The WordMap interface

```haskell
lookup :: Key -> WordMap a -> Maybe a
insert :: Key -> a -> WordMap a -> WordMap a
delete :: Key -> WordMap a -> WordMap a

lookupT :: Key -> TWordMap s a -> ST s (Maybe a)
insertT :: Key -> a -> TWordMap s a -> ST s (TWordMap s a)
deleteT :: Key -> TWordMap s a -> ST s (TWordMap s a)

persistent :: TWordMap s a -> ST s (WordMap a)
transient :: WordMap a -> TWordMap s a

fromList :: [(Key, a)] -> WordMap a
fromList xs = runST $ fromListT xs >>= persistent

fromListT :: [(Key, a)] -> ST s (TWordMap s a)
```

The linear WordMap interface

Only the most recent version of a transient `TWordMap` may be accessed. In line 3 of the following example, `m1` is not the most recent version and therefore using it is not allowed. However, this requirement of the API is not enforced.

```haskell
1 m1 <- fromListT [(2, 20), (3, 30), (4, 40)]
2 m2 <- insertT 1 10 m1
3 x <- lookupT 2 m1
```

Instead, `lookupT` returns `Nothing`, which might be unexpected, as both the `TWordMap` returned by `fromListT` and the one returned by `insertT` contain the key 2. This behaviour can be explained by looking at the internal representation of the `TWordMaps` `m1` and `m2`. Both maps contain a single internal node, which consists of a prefix, an offset, a bitmap and a `SmallMutableArray` of leaf nodes. In the bitmap of `m1` the bits 2, 3 and 4 are set, whereas in the bitmap of `m2` the bits 1, 2, 3 and 4 are set. However, the `SmallMutableArray` is the same for both maps. In `lookupT`, the given key, 2, is used as an index into the bitmap and because bit 2 is set, the search continues in a child node. The position of this child node in the array of children is calculated as the number of set bits up to bit 2 and is therefore 0. At position 0 of the child array is the leaf with key 1, which was inserted in the call to `insertT`. As the key of the leaf and the key passed to `lookupT` are different, `Nothing` is returned.

In order to catch such errors at compile time, we provide an additional interface that uses Haskells linear types. A function `f` is linear if: when its result is consumed exactly once, its argument is consumed exactly once [1]. Writing `f :: a %1 -> b` means that `f` is a linear function from `a` to `b` [1]. The linear interface is shown in listing 2.2.

Listing 2.2: `The linear interface.`

```haskell
newtype LWordMap a = LWordMap (TWordMap RealWorld a)

lookupL :: Key -> LWordMap a %1 -> (Ur (Maybe a), LWordMap a)
insertL :: Key -> a -> LWordMap a %1 -> LWordMap a
deleteL :: Key -> LWordMap a %1 -> LWordMap a

persistentL :: LWordMap a %1 -> Ur (WordMap a)
transientL :: WordMap a -> (LWordMap a %1 -> Ur b) %1 -> Ur b

fromListL :: [(Key, a)] -> (LWordMap a %1 -> Ur b) %1 -> Ur b
```

With this linear interface, the example from before could be written in the following way, which would however be rejected by the compiler.

```haskell
fromListL [(2, 20), (3, 30), (4, 40)] $ \m1 ->
  insertL 1 10 m1 & \m2 ->
    lookupL 2 m1 & \(x, m3) ->
      consume m3 & \() ->
        x
```

The `linear-base`[1] library includes some type classes for doing non-linear things in linear code. Most notably, the `Dupable` class allows a value to be duplicated via the `dup2 :: a %1 -> (a, a)` method. Mutable containers such as `Array` or `Vector` are instances of this class, but in `dup2` they must be cloned. The `LWordMap` on the other hand can also be an instance of this class, and it does not have to clone the whole underlying `TWordMap`. Instead, the `TWordMap` is persisted and the resulting persistent `WordMap` is converted back into two transient `TWordMaps`.

```haskell
instance Dupable (LWordMap a) where
  dup2 lmap =
    persistentL lmap & \(Ur pmap) ->
      (LWordMap (transient pmap), LWordMap (transient pmap))
```

[1]`https://hackage.haskell.org/package/linear-base`

3 Case Study: Transient WordMaps in GHC

We want to explore whether using transient data structures in Haskell can improve performance. As an example for a large Haskell codebase we look at the Glasgow Haskell Compiler [1] (GHC). In order to show the performance improvement achieved by using a transient data structure, we replace a data structure in GHC with a transient one using our implementation of a transient `WordMap`. Our modifications to the compiler are available at `https://github.com/PascEll/ghc`.

GHC uses dictionaries with integer keys in the form of `IntMaps`. These `IntMaps` are compressed binary tries. While it would be possible to implement the same interface as `IntMap` for our persistent `WordMap`, replacing every `IntMap` in GHC with a persistent `WordMap` would decrease performance. One of the reasons for this performance decrease is that the `WordMap` nodes are much larger than the `IntMap` nodes and copying them is therefore more expensive. Performance is only improved when the nodes are copied much less frequently, which is possible by using the transient `TWordMap` in significant parts of the code. However, as changing code to use the transient `TWordMap` is a lot of manual effort, doing so for significant parts of the whole GHC codebase is not feasible.

A data structure in GHC for which the required code changes are feasible while the performance impact is still significant, is the `InScopeSet`. The `InScopeSet` type wraps a `VarSet`, which eventually is backed by an `IntMap`. We use our transient `WordMap` to replace the `VarSet` in the `InScopeSet` in GHC.

```haskell
newtype InScopeSet = InScope VarSet
```

3.1 The persistent InScopeSet

The first step to using the transient `WordMap` in GHC is to replace the `IntMap` backed `VarSet` in the `InScopeSet` with a persistent `WordMap`.

Instead of replacing the `VarSet` directly, we create a wrapper for the persistent `WordMap`, `FastVarSet`, which has a similar interface to that of `VarSet`. Most functions simply wrap operations on the `WordMap`.

However, the `extendFastVarSetList` function, which inserts a list of `Vars` into the `WordMap`, is a first opportunity to use the transient `TWordMap`. Instead of inserting the `Vars` into persistent `WordMaps`, which would, for each insertion, copy all nodes that are modified, we insert them into transient `TWordMaps`. For such an insertion, in the best case, all nodes that need to be modified are mutable, and no nodes are copied. As the `extendFastVarSetList` function still operates on persistent `FastVarSets`, first the `WordMap` in the `FastVarSet` is converted into a `TWordMap`. Then the `TWordMap` is modified and, finally, the resulting transient `TWordMap` is converted back into a persistent `WordMap`.

Listing 3.1: `extendFastVarSetList`

```haskell
newtype FastVarSet = FastVarSet (WordMap Var)

extendFastVarSetList :: FastVarSet -> [Var] -> FastVarSet
extendFastVarSetList (FastVarSet set) vars
  = FastVarSet $ runST $ do
    tmap <- foldM
              (\map var -> insertT (varToKey var) var map)
              (transient set)
              vars
    persistent tmap
```

In order to replace the `VarSet` in the `InScopeSet` with the `FastVarSet`, the functions using the internal `VarSet` must be adjusted. In most cases this adjustment is simply the replacement of the function operating on `VarSets` with the appropriate function operating on `FastVarSets`. However, sometimes `InScopeSets` need to be converted to and from `VarSets`. As we do not replace all `VarSets` in the whole GHC codebase, in those cases, the `FastVarSet` needs to be converted into a `VarSet` and vice versa.

The `mkInScopeSet` function constructs an `InScopeSet` from a `VarSet`. Before using the `FastVarSet` that was very simple, as listing 3.2 shows. With the `FastVarSet`, the `WordMap` has to be constructed first, as shown in listing 3.3.

Listing 3.2: The original `mkInScopeSet` function.

```haskell
mkInScopeSet :: VarSet -> InScopeSet
mkInScopeSet in_scope = InScope in_scope
```

Listing 3.3: The new `mkInScopeSet` function.

```haskell
mkInScopeSet :: VarSet -> InScopeSet
mkInScopeSet in_scope =
  InScope
    ( FastVarSet
        ( WordMap.fromAscList $
            map intKeyToWordKey $
              varSetToAscList in_scope
        )
    )
  where
    intKeyToWordKey (k, v) = (fromIntegral k, v)
```

At this point all changes are local to the InScopeSet and the code works again.

Obviously, the new `mkInScopeSet` function is much more costly than the old one. However, often the `VarSet` passed to `mkInScopeSet` is only constructed for that purpose from a list. In these cases we eliminate the construction of the intermediate `VarSet` by constructing the `InScopeSet` directly from the list.

```haskell
-- Construction of an InScopeSet from an intermediate VarSet
let in_scope = mkInScopeSet (mkVarSet tvs1)
-- Construction without the intermediate VarSet
let in_scope' = mkInScopeSetList tvs1
```

3.2 The transient TInScopeSet

At this point there is only a persistent `InScopeSet`. This way the usage of the `WordMap` and the transient `TWordMap` are a nicely contained implementation detail of the `InScopeSet`. However, the `TWordMap` is only used in some collective operations such as `extendInScopeSetList`. The next step is to use the transient `TWordMaps` in more places, and to keep them transient between modifications. Keeping the `TWordMap` transient means that nodes that are modified repeatedly do not need to be copied each time, which improves performance.

In order to use the `TWordMap` in more places, we make transient variants of the `FastVarSet` and the `InScopeSet`. Doing so adds a bit of boilerplate code, however, it allows us to use the potentially faster operations on the transient variants without exposing the `WordMap` directly.

The transient `TFastVarSet` type wraps a `TWordMap`, and we add transient variants of the operations on `FastVarSets` that operate on `TFastVarSets`. These operations use the transient interface of the `WordMap`. The transient `TInScopeSet` type wraps a `TFastVarSet` and, similar to the `TFastVarSet`, we add transient variants of the operations on `InScopeSets` that operate on `TInScopeSets`. Listing 3.4 shows operations on persistent `InScopeSets` and `FastVarSets` aswell as the respective transient variants. Additionally, we add `persistentInScopeSet` and `transientInScopeSet` functions, which convert between persistent `InScopeSets` and transient `TInScopeSets`, as shown in listing 3.5. As the `TInScopeSet` is eventually backed by a `TWordMap` and it is only safe to access the most recent version of the `TWordMap`, it is also only safe to access the most recent version of a `TInScopeSet`.

Listing 3.4: Persistent and transient operations on InScopeSets and FastVarSets

```haskell
newtype FastVarSet = FastVarSet (WordMap Var)
newtype TFastVarSet s = TFastVarSet (TWordMap s Var)

extendFastVarSet :: FastVarSet -> Var -> FastVarSet
extendFastVarSet (FastVarSet set) var
  = FastVarSet (insert (varToKey var) var set)

extendTFastVarSet :: TFastVarSet s -> Var -> ST s (TFastVarSet s)
extendTFastVarSet (TFastVarSet set) var
  = TFastVarSet <$> insertT (varToKey var) var set

newtype InScopeSet = InScope FastVarSet
newtype TInScopeSet s = TInScope (TFastVarSet s)

extendInScopeSet :: InScopeSet -> Var -> InScopeSet
extendInScopeSet (InScope in_scope) v
  = InScope (extendFastVarSet in_scope v)

extendTInScopeSet :: TInScopeSet s -> Var -> ST s (TInScopeSet s)
extendTInScopeSet (TInScope in_scope) v
  = TInScope <$> (extendTFastVarSet in_scope v)
```

Listing 3.5: `transientInScopeSet` and `persistentInScopeSet`

```haskell
transientInScopeSet :: InScopeSet -> TInScopeSet s
transientInScopeSet (InScope (FastVarSet in_scope))
  = TInScope (TFastVarSet (transient in_scope))

persistentInScopeSet :: TInScopeSet s -> ST s (InScopeSet)
persistentInScopeSet (TInScope (TFastVarSet in_scope))
  = InScope . FastVarSet <$> persistent in_scope
```

3.3 Using the TInScopeSet

The transient variant of an operation can be faster than the corresponding persistent variant because in the transient variant we can modify nodes, or more specifically the arrays they contain, in place instead of copying them. However, we can only modify nodes in place if such a node is not shared with a different `WordMap` or `TWordMap`. Using a transient variant of an operation is therefore only faster than the corresponding persistent variant if some of the nodes that are modified are marked as mutable. As all nodes in a transient `TWordMap` are marked as immutable when it is converted into a persistent `WordMap`, using `TWordMaps` is beneficial if the `TWordMap` is modified repeatedly, and it can be kept transient between modifications.

Recall the `extendFastVarSetList` function in listing 3.1, which is such an opportunity. The `TWordMap` is modified repeatedly because multiple `Vars` are inserted into it. As the intermediate versions of the `TWordMap` are only passed to the next `insertT` operation, it can be kept transient between them.

Another function that modifies an `InScopeSet` repeatedly is `substExpr`. A simplified implementation of it is shown in listing 3.6. The `substExpr` function applies a `Subst` to an `Expr`. If `substExpr` is called with a lambda expression, the `Subst` is passed to `substBndr`. `substBndr` modifies the `InScopeSet` contained in the `Subst`, and then `substExpr` is called recursively with the resulting modified `Subst`.

Listing 3.6: `substExpr`

```haskell
data Subst = Subst InScopeSet ...

data Expr
  = Var Var
  | App Expr Expr
  | Lam Var Expr

substExpr :: Subst -> Expr -> Expr
substExpr subst expr = go expr
  where
    go (Var v)          = lookupIdSubst subst v
    go (App fun arg)    = App (go fun) (go arg)
    go (Lam bndr body) = Lam bndr' (substExpr subst' body)
      where
        (subst', bndr') = substBndr subst bndr

lookupIdSubst :: Subst -> Var -> Expr
substBndr :: Subst -> Var -> (Subst, Var)
```

The objective is to make a transient variant of `substExpr` that can keep a `TInScopeSet` transient between the modifications in `substBndrT` as often as possible. The first version of `substExprT`, which is shown in listing 3.7, does not quite accomplish this objective. In case of a lambda expression, `substExprT` passes the transient `TSubst` to `substBndrT`, the transient variant of `substBndr`. The resulting modified `TSubst` is then passed to `substExprT` without persisting it. Applying a substitution to an expression like `Lam v1 (Lam v2 (Lam v3 (Var v2)))` might therefore be able to mutate parts of the `TSubst` in place when `substBndrT` is called with `v2` and `v3`. However, consider the expression `App (Lam v1 (Var v1)) (Lam v2 (Var v10))`. In case of an application, the same substitution is applied to the function as well as the argument. In this expression, both the function and the argument are lambda expressions, which result in a modification of the `TSubst` when passed to `substExprT`. Therefore, the `TSubst` must be persisted before it is passed to `substExprT` again. It is not always actually necessary to persist the `TSubst` though. For instance, if the argument of an application is a variable, like in `App (Lam v1 (Var v1)) (Var v2)`, applying the substitution to it does not modify the `TSubst`. In this case, applying the `TSubst` to the argument first and then the function without persisting the `TSubst` would still apply the same substitution to both the function and the argument. Not persisting the `TSubst` would have the advantage that following modifications might need to copy less.

Listing 3.7: `First version of substExprT.`

```
data TSubst s = TSubst (TInScopeSet s) ...

substExpr :: Subst -> Expr -> Expr
substExpr subst expr
  = runST $ substExprT (transientSubst subst) expr

substExprT :: TSubst s -> Expr -> ST s Expr
substExprT subst expr = go expr
  where
    go (Var v) = lookupIdSubstT subst v

    go (App fun arg) = do
      p_subst <- persistentSubst subst
      arg' <- substExprT (transientSubst p_subst) arg
      fun' <- substExprT (transientSubst p_subst) fun
      return (App fun' arg')

    go (Lam bndr body) = do
      (subst', bndr') <- substBndrT subst bndr
      body' <- substExprT subst' body
      return (Lam bndr' body')

lookupIdSubstT :: TSubst s -> Var -> ST s Expr
substBndrT :: TSubst s -> Var -> ST s (TSubst s, Var)
```

We want to only persist the TSubst that is applied to the argument of an application if the TSubst is actually modified. This problem is similar to a problem that can occur in Rust [8]. In Rust, there are shared borrows and mutable borrows. Data cannot be mutated through shared borrows. When only a shared borrow is available, but mutation is required, it is often possible to clone the borrowed data. Of course, cloning is potentially expensive and therefore, if mutation is only required in some cases, the data should only be cloned in those cases. Additionally, the data should not be cloned if it is already owned because that would also be unnecessary.

There are a few differences between the situtation in Rust and our substExprT function in Haskell. In Haskell, there is no concept of borrowing, but we can think of data that is not allowed to be mutated as being borrowed. For instance, in the substExprT function, in the case of an application, when the TSubst is applied to the argument without being persisted first, it may not be mutated because the same TSubst also needs to be applied to the function. Therefore, while being applied to the argument, we can think of the TSubst as borrowed. In Rust, when borrowed data needs to be modified, the data is cloned. Since we are using a transient data structure though, the TSubst does not need to be cloned, it only needs to be persisted.

In Rust, the objective is to only clone the data if mutation is required and the data is not owned. In order to achieve that, Rust has the Cow enum, which is a clone-on-write smart pointer. The enum has a borrowed variant and an owned variant, and immutable access to the data is always possible. However, when mutation is required and the data is only borrowed, it is cloned before it is mutated. On the other hand, if the data is owned, it is not cloned.

In the substExprT function, we want to only persist the TSubst being applied to the argument of an application if the TSubst is modified. The version of substExprT shown in listing 3.8 does that, and it has some similarities to the solution using the Cow enum in Rust. In substExprT, the TSubst is only modified in the Lam case because lookupIdSubstT never modifies the TSubst. Therefore, the TSubst is only persisted in the Lam case and never in the App case directly. However, as the objective is to keep the TSubst transient between modifications, we cannot always persist the TSubst in the Lam case. Instead, in the Lam case, we need to distinguish between TSubsts that need to be persisted before they are modified and TSubsts that do not need to be persisted. In order to make this distinction possible, substExprT is passed an additional argument, owned, which indicates whether the TSubst is owned and therefore may be modified. The `owned` parameter in combination with the `TSubst` is similar to the `Cow` enum in Rust. The owned enum variant in Rust corresponds to `owned` being true and the borrowed enum variant corresponds to `owned` being false. As we do not have separate types for owned and borrowed `TSubsts`, we do not need a persistent-on-write type that can hold the two variants and the `owned` parameter is sufficient. In Rust, if the data is not owned, it is cloned. Here, in the `Lam` case, the `TSubst` is persisted if it is not owned. When `substExprT` is called in `substExpr`, the `TSubst` is only used in this call and may be modified. The same is true for the `TSubst` passed to `substExprT` in the `Lam` case. On the other hand, in the `App` case, the substitution may be used both when applying it to the argument and when applying it to the function. Therefore, when it is applied to the argument, it may not be modified. When applying a substitution to the example expression from before, `App (Lam v1 (Var v1)) (Var v2)`, in contrast to the first version, in this version of `substExprT` the `TSubst` is not persisted. Thus, following modifications to the `TSubst` might be able to happen in place.

Listing 3.8: `Second version of substExprT.`

```haskell
substExpr :: Subst -> Expr -> Expr
substExpr subst expr = runST $ substExprT (transientSubst subst)
   True expr

substExprT :: TSubst s -> Bool -> Expr -> ST s Expr
substExprT subst owned expr
  = go expr
  where
    go (Var v) = lookupIdSubstT subst v

    go (App fun arg) = do
      arg' <- substExprT subst False arg
      fun' <- go fun
      return (App fun' arg')

    go (Lam bndr body) = do
      owned_subst <- toOwned subst
      (subst', bndr') <- substBndrT owned_subst bndr
      body' <- substExprT subst' True body
      return (Lam bndr' body')

    toOwned subst
      | owned = return subst
      | otherwise = transientSubst <$> persistentSubst subst
```

4 Evaluation

In order to evaluate our transient `WordMap` implementation, we benchmark the lookup and insert operations directly. As we also used the transient `WordMap` in GHC, we evaluate the performance of our modified GHC[1], too.

4.1 Lookup and Insert

We compare the performance of the lookup and insert operations of our transient `WordMap` with the strict `IntMap` from the containers package[2]. For this comparison, we adapt a benchmark for dictionary data structures[3] to include our `WordMap`.

For the insert operation, we compare the performance of the strict `IntMap` with the persistent `WordMap` and the transient `TWordMap`. The benchmark inserts the keys from 1 to n into empty dictionaries for $n \in 10, 100, 1000, 10000$. The means of the times that these insertions took, are shown in figure 4.1.

n	`IntMap`	`TWordMap`	`WordMap`
10	230.3 ns	390.8 ns	437.4 ns
100	4.073 µs	5.546 µs	6.991 µs
1000	55.86 µs	66.02 µs	97.38 µs
10000	890.8 µs	804.7 µs	1.569 ms

Figure 4.1: The means of the times the insertion of the keys 1 to n into an empty dictionary took for different dictionaries with integer keys.

As expected, insertions into the `TWordMap` are faster than insertions into the persistent `WordMap` because the `WordMap` needs to copy nodes more often than the `TWordMap`. The `TWordMap` is only faster than the `IntMap` for the highest number of keys, 10000. As the `IntMap` is a compressed binary trie, a node in an `IntMap` only differentiates one bit of the keys compared to the 4 bits in the `WordMap`. Therefore the height of an `IntMap` tree is generally bigger than the height of a `WordMap` tree, but the operations on `IntMaps` are simpler than the ones on `WordMaps`. That suggests that the lower tree height of the `TWordMap` and the reduced number of copies only outweighs the simplicity of the `IntMap` if there are lots of keys.

[1]Git Commit Hash: e7c4eec138a70a933447fe3f9b2edc9922e0569c

[2]`https://hackage.haskell.org/package/containers`

[3]`https://github.com/haskell-perf/dictionaries`

As the lookup operation of persistent `WordMaps` and transient `TWordMaps` are the same, there is no point in comparing those two. Therefore, for the lookup operation, we only compare the strict `IntMap` with the persistent `WordMap`. The benchmark performs a lookup for all the keys in a dictionary with random keys of size n for $n \in 10, 100, 100010000, 100000, 1000000$. The means of the times that these lookups took, are shown in figure 4.2. Lookups are faster for the `WordMap` compared to the `IntMap` for $n \geq 100$, which is probably because of the higher trees in the `IntMap`.

n	`IntMap`	`WordMap`
10	177.5 ns	193.4 ns
100	2.658 µs	2.412 µs
1000	87.00 µs	30.12 µs
10000	1.569 ms	529.4 µs
100000	26.52 ms	9.118 ms
1000000	622.6 ms	235.4 ms

Figure 4.2: The means of the times the insertion of the keys 1 to n into an empty dictionary took for different dictionaries with integer keys.

4.2 GHC

We evaluate the performance of our modified compiler by comparing it to a commit on the GHC master branch, bd9218[4].

4.2.1 Allocations

GHC has a test suite that also includes performance tests. Figure 4.3 shows the bytes allocated in the heap for the performance test cases. Test cases with changes that are smaller than 0.4 percent are omitted.

The change of allocated bytes in the heap ranges from -15.3 percent to +2.9 percent. Given that we only made changes to a small part of the compiler, a reduction of 15.3 percent shows that using transient data structures in Haskell can improve the performance. However, in most test cases, the performance of our modified compiler and the master commit is very similar. In order to explain this spread, we look at two of those test cases in more detail.

When compiling T18223, our modified compiler allocated 15.3 percent less than the master commit. In contrast, when compiling T9630, our modified compiler allocated 1.1 percent more than the master commit. In order to gain more insight into where those changes to the allocations happen, we look at the ticky-ticky profiles of the

[4]Git Commit Hash: bd92182cd56140ffb2f68ec01492e5aa6333a8fc

master commit and our modified compiler, each compiling T18223 and T9630. As we replaced the `IntMap` that eventually backs the `InScopeSet` with a `WordMap`, some of the allocations that are made by `IntMap` operations in the master commit are gone in our modified compiler. In the modified compiler, there are instead allocations made by `WordMap` operations. For T18223, the `IntMap` operation with the biggest reduction in allocations is `insert` with 113,879,272 bytes less allocated. These 113,879,272 bytes are about 12 percent of total heap allocations during compilation with the master commit. For T9630, `insert` is also the `IntMap` operation with the biggest reduction. However, with 52,062,672 bytes, this reduction is only about 3 percent of total heap allocations during the compilation with the master commit. As we want to reduce allocations by replacing the `IntMap` with a `WordMap`, the amount of reduced bytes allocated by `IntMap` operations is sort of an upper bound to the reduction of bytes allocated we can achieve with our changes. Therefore, as for T9630, the reduction of bytes allocated by `IntMap` operations is only 3 percent of total heap allocations, the potential for reducing allocations in T9630 is already very low. An explanation for why the number of bytes allocated was even increased, is that an insertion into our transient `WordMap` does not always allocate less than an insertion into an equivalent `IntMap`. Such an insertion into a `TWordMap` probably only allocates less if some of its nodes are mutable.

4.2.2 Runtime

In addition to comparing the allocations of the performance tests in GHCs test suite, we also evaluate the runtime of our modified compiler. We do that by compiling T18223 and Cabal[5] with the GHC commit on the master branch and our modified compiler. For compiling Cabal, we use this script[6]. In order to determine the runtimes for compiling Cabal, we let hyperfine[7] execute that script ten times with each compiler. Our modified compiler allocated 15.3 percent less than the master commit when compiling T18223 with the -O flag, so we also let hyperfine run the compilation of T18223 ten times with each compiler.

Figure 4.4 shows that it takes our modified compiler about 180 ms less than the master commit to compile T18223, which is about 13 percent of the master commits time. However, the table also shows that the commit on the master branch is about 1 percent faster than our modified compiler at compiling Cabal. In order to understand, why our modifications to the compiler do not result in a shorter runtime when compiling Cabal, we look at the ticky-ticky profiles of the master commit and our modified compiler. As we replaced the `IntMap` that eventually backs the `InScopeSet` with a `WordMap` in the modified compiler, we use the change in allocations by the `insert` operation on `IntMap` as an indicator whether the `InScopeSet` has enough impact on the performance of the compiler. The allocations made by the `insert` operation on `IntMaps` were reduced by 325,831,736 bytes, which

[5]`https://github.com/haskell/cabal/`
[6]`https://gitlab.haskell.org/bgamari/ghc-utils/-/blob/master/build-cabal.sh`
[7]`https://github.com/sharkdp/hyperfine`

is about 0.7 percent of total heap allocations during the compilation with the master commit. Therefore, the `InScopeSet` likely does not have enough impact on the time it takes to compile Cabal.

Test	bd9218	e7c4ee	change
LargeRecord(normal)	6,049,739,192	6,078,551,904	+0.5%
PmSeriesS(normal)	55,046,928	55,287,072	+0.4%
PmSeriesT(normal)	77,475,520	78,027,080	+0.7%
PmSeriesV(normal)	54,326,304	54,565,648	+0.4%
T10421(normal)	114,096,472	114,545,008	+0.4%
T10421a(normal)	80,456,776	80,756,272	+0.4%
T10858(normal)	133,430,864	134,157,656	+0.5%
T11195(normal)	238,318,968	239,538,152	+0.5%
T12150(optasm)	81,753,960	82,332,752	+0.7%
T13056(optasm)	350,455,952	352,972,376	+0.7%
T13253(normal)	345,927,808	348,625,048	+0.8%
T13253-spj(normal)	126,119,808	127,214,104	+0.9%
T14683(normal)	2,830,711,432	2,911,607,904	+2.9%
T15164(normal)	1,298,449,592	1,291,408,096	-0.5%
T17516(normal)	1,814,191,904	1,837,285,800	+1.3%
T18140(normal)	77,604,224	77,997,248	+0.5%
T18223(normal)	916,239,720	776,460,976	-15.3%
T18282(normal)	151,491,680	152,495,464	+0.7%
T18698a(normal)	202,368,224	201,199,792	-0.6%
T18923(normal)	68,562,752	68,821,320	+0.4%
T19695(normal)	1,446,608,872	1,455,410,136	+0.6%
T20049(normal)	92,395,512	91,933,872	-0.5%
T20261(normal)	603,105,184	606,903,544	+0.6%
T3064(normal)	181,464,840	182,107,008	+0.4%
T5631(normal)	533,662,664	536,621,112	+0.6%
T6048(optasm)	103,707,224	104,448,600	+0.7%
T783(normal)	385,375,704	387,498,344	+0.6%
T9233(normal)	721,456,800	672,129,808	-6.8%
T9630(normal)	1,527,790,320	1,543,899,336	+1.1%
T9675(optasm)	440,199,376	402,840,488	-8.5%
WWRec(normal)	622,269,864	619,549,680	-0.4%
geo. mean			-0.1%
minimum			-15.3%
maximum			+2.9%

Figure 4.3: The bytes allocated by a commit on the master branch, bd9218, and our modified compiler, e7c4ee. Tests with changes smaller than 0.4 percent are omitted.

	bd9218	e7c4ee
Cabal	46.242 s	46.781 s
T18223	1.395 s	1.213 s

Figure 4.4: The means of the times it took a commit on the master branch, bd9218, and our modified compiler, e7c4ee, to compile Cabal and T18223.

5 Conclusion

The evaluation showed that replacing an immutable data structure with a transient one in a large codebase like GHC can improve performance in some cases. There are probably two main reasons why the performance could not be improved more. As the evaluation also showed, in some test cases, the `InScopeSet` we replaced was not responsible for enough allocations. Similarly, the `InScopeSet` likely does not have enough impact on the time it takes to compile Cabal. The other reason is that insertions into transient `WordMaps` only require fewer allocations than `IntMaps` if the dictionary contains enough keys and the nodes are mutable.

However, lookups are faster for our `WordMap` than for the `IntMap`. Therefore, the ideal use case for the `WordMap` is probably when the performance of lookups is relevant and most updates happen in bulk. In that case, one could benefit from the fast lookups of the persistent `WordMap` while keeping the performance of updates reasonable by using the transient `TWordMap`.

An open questions that remains is whether replacing a data structure in GHC that is used more than the `InScopeSet` like the `VarSet` would be feasible and improve performance. There is also a GHC proposal for mutable constructor fields[1], which our `WordMap` could benefit from. When a node is inserted into or deleted from the array of child nodes, the bitmap of the node changes. So, even if we can modify the array in place, we need to allocate a new node with the changed bitmap. With mutable constructor fields we would not have to allocate a new node.

[1]`https://github.com/simonmar/ghc-proposals/blob/mutable-fields/proposals/0000-mutable-fields.rst`

Bibliography

[1] “The glasgow haskell compiler.” `https://www.haskell.org/ghc/`. Accessed: 2022-10-20.

[2] A. Moine, A. Charguéraud, and F. Pottier, “Specification and verification of a transient stack,” in *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, (New York, NY, USA), p. 82–99, Association for Computing Machinery, 1 2022.

[3] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, “Making data structures persistent,” in *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, STOC ’86, (New York, NY, USA), p. 109–121, Association for Computing Machinery, 11 1986.

[4] P. Bagwell, “Ideal hash trees,” tech. rep., 2001.

[5] D. R. Morrison, “Patricia—practical algorithm to retrieve information coded in alphanumeric,” p. 514–534, 10 1968.

[6] “The clojure programming language.” `https://clojure.org/`. Accessed: 2022-11-07.

[7] E. Kmett, “Transients.” `https://github.com/ekmett/transients`. Accessed: 2022-11-07.

[8] “Rust programming language.” `https://www.rust-lang.org/`. Accessed: 2022-10-20.

Erklärung

Hiermit erkläre ich, Pascal Ellinger, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift