Triemaps that match

Technical Report

Simon Peyton Jones
Epic Games
Cambridge, UK
simon.peytonjones@gmail.com

Sebastian Graf
Karlsruhe Institute of Technology
Karlsruhe, Germany
sebastian.graf@kit.edu

Abstract

The *trie* data structure is a good choice for finite maps whose keys are data structures (trees) rather than atomic values. But what if we want the keys to be *patterns*, each of which matches many lookup keys? Efficient matching of this kind is well studied in the theorem prover community, but much less so in the context of statically typed functional programming. Doing so yields an interesting new viewpoint — and a practically useful design pattern, with good runtime performance.

1 Introduction

Many functional languages provide *finite maps* either as a built-in data type, or as a mature, well-optimised library. Generally the keys of such a map will be small: an integer, a string, or perhaps a pair of integers. But in some applications the key is large: an entire tree structure. For example, consider the Haskell expression

let $x = a + b$ **in** ... (**let** $y = a + b$ **in** $x + y$) ....

We might hope that the compiler will recognise the repeated sub-expression $(a + b)$ and transform to

let $x = a + b$ **in** ... $(x + x)$ ....

An easy way to do so is to build a finite map that maps the expression $(a + b)$ to x. Then, when encountering the inner **let**, we can look up the right hand side in the map, and replace y by x. All we need is a finite map keyed by syntax trees.

Traditional finite-map implementations tend to do badly in such applications, because they are often based on balanced trees, and make the assumption that comparing two keys is a fast, constant-time operation. That assumption is false for large, tree-structured keys.

Another time that a compiler may want to look up a tree-structured key is when rewriting expressions: it wants to see if any rewrite rule matches the sub-expression in hand, and if so rewrite with the instantiated right-hand side of the rule. To do this we need a fast way to see if a target expression matches one of the patterns in a set of (*pattern*, *rhs*) pairs. If there is a large number of such (*pattern*, *rhs*) entries to check, we would like to do so faster than checking them one by one. Several parts of GHC, a Haskell compiler, need matching lookup, and currently use an inefficient linear algorithm to do so.

In principle it is well known how to build a finite map for a deeply-structured key: use a *trie*. The matching task is also well studied but, surprisingly, only in the automated reasoning community (Section 7.1): they use so-called *discrimination trees*. In this paper we apply these ideas in the context of a statically-typed functional programming language, Haskell. This shift of context is surprisingly fruitful, and we make the following contributions:

- Following Hinze [2000a], we develop a standard pattern for a *statically-typed triemap* for an arbitrary algebraic data type (Section 3.2). In contrast, most of the literature describes untyped tries for a fixed, generic tree type. In particular:
  - Supported by type classes, we can make good use of polymorphism to build triemaps for polymorphic data types, such as lists (Section 3.6).
  - We cover the full range of operations expected for finite maps: not only *insert*ion and *lookup*, but *alter*, *union*, *fold*, *map* and *filter* (Section 3.2).
  - We develop a generic optimisation for singleton maps that compresses leaf paths. Intriguingly, the resulting triemap *transformer* can be easily mixed into arbitrary triemap definitions (Section 3.7).
- We show how to make our triemaps insensitive to α*-renamings* in keys that include binding forms (Section 4). Accounting for α-equivalence is not hard, but it is crucial for the applications in compilers.
- We extend our triemaps to support *matching* lookups (Section 5). This is an important step, because the only readily-available alternative is linear lookup. Our main contribution is to extend the established idea of tries keyed by arbitrary data types, so that it can handle matching too.
- We present measurements that compare the performance of our triemaps (ignoring their matching capability) with traditional finite-map implementations in Haskell (Section 6).

We discuss related work in Section 7. Our contribution is not so much a clever new idea as an exposition of some

Authors' addresses: Simon Peyton Jones, Epic Games, Cambridge, UK, simon.peytonjones@gmail.com; Sebastian Graf, Karlsruhe Institute of Technology, Karlsruhe, Germany, sebastian.graf@kit.edu.

2022. 2475-1421/2022/5-ART1 $15.00
https://doi.org/

```
type TF v = Maybe v → Maybe v
data Map k v = ...    -- Keys k, values v
Map.empty    :: Map k v
Map.insert   :: Ord k ⇒ k → v → Map k v → Map k v
Map.lookup   :: Ord k ⇒ k → Map k v → Maybe v
Map.alter    :: Ord k ⇒ TF v → k
                → Map k v → Map k v
Map.foldr    :: (v → r → r) → r → Map k v → r
Map.map      :: (v → w) → Map k v → Map k w
Map.unionWith :: Ord k ⇒ (v → v → v)
                → Map k v → Map k v → Map k v
Map.size     :: Map k v → Int
Map.compose  :: Ord b ⇒ Map b c → Map a b → Map a c
infixr 1 >=>          -- Kleisli composition
(>=>) :: Monad m ⇒ (a → m b) → (b → m c)
                 → a → m c
infixr 1 >>>          -- Forward composition
(>>>) :: (a → b) → (b → c) → a → c
infixr 0 |>           -- Reverse function application
(|>) :: a → (a → b) → b
```

Figure 1. API for library functions

old ideas in a new context. Nevertheless, we found it surprisingly tricky to develop the "right" abstractions, such as the *TrieMap* and *Matchable* classes, the singleton-and-empty map data type, and the combinators we use in our instances. These abstractions have been through *many* iterations, and we hope that by laying them out here, as a functional pearl, we may shorten the path for others.

2 The problem we address

Our general task is as follows: *implement an efficient finite mapping from keys to values, in which the key is a tree.* Semantically, such a finite map is just a set of *(key,value)* pairs; we query the map by looking up a *target*. For example, the key might be a data type of syntax trees, defined like this:

```
type Var = String
data Expr = App Expr Expr | Lam Var Expr | Var Var
```

Here *Var* is the type of variables; these can be compared for equality and used as the key of a finite map. Its definition is not important for this paper, but for the sake of concreteness, you may wish to imagine it is simply a string: The data type *Expr* is capable of representing expressions like $(add\ x\ y)$ and $(\lambda x.\ add\ x\ y)$. We will use this data type throughout the paper, because it has all the features that occur in real expression data types: free variables like *add*, represented by a *Var* node; lambdas which can bind variables (*Lam*), and occurrences of those bound variables (*Var*); and nodes with multiple children (*App*). A real-world expression type would have many more constructors, including literals, let-expressions and suchlike.

2.1 Alpha-renaming

In the context of a compiler, where the keys are expressions or types, the keys may contain internal *binders*, such as the binder x in $(\lambda x.x)$. If so, we would expect insertion and lookup to be insensitive to α-renaming, so we could, for example, insert with key $(\lambda x.x)$ and look up with key $(\lambda y.y)$, to find the inserted value.

2.2 Lookup modulo matching

Beyond just the basic finite maps we have described, our practical setting in GHC demands more: we want to do a lookup that does *matching*. GHC supports so-called *rewrite rules* [Peyton Jones et al. 2001], which the user can specify in their source program, like this:

```
{-# RULES "map/map" ∀f g xs. map f (map g xs)
                              = map (f ∘ g) xs #-}
```

This rule asks the compiler to rewrite any target expression that matches the shape of the left-hand side (LHS) of the rule into the right-hand side (RHS). We use the term *pattern* to describe the LHS, and *target* to describe the expression we are looking up in the map. The pattern is explicitly quantified over the *pattern variables* (here f, g, and xs) that can be bound during the matching process. In other words, we seek a substitution for the pattern variables that makes the pattern equal to the target expression. For example, if the program we are compiling contains the expression *map double* (*map square nums*), we would like to produce a substitution $[f \mapsto double, g \mapsto square, xs \mapsto nums]$ so that the substituted RHS becomes *map* (*double* ∘ *square*) *nums*; we would replace the former expression with the latter in the code under consideration.

Of course, the pattern might itself have bound variables, and we would like to be insensitive to α-conversion for those. For example:

```
{-# RULES "map/id" map (λx → x) = λy → y #-}
```

We want to find a successful match if we see a call *map* $(\lambda y \rightarrow y)$, even though the bound variable has a different name.

Now imagine that we have thousands of such rules. Given a target expression, we want to consult the rule database to see if any rule matches. One approach would be to look at the rules one at a time, checking for a match, but that would be slow if there are many rules. Similarly, GHC's lookup for type-class instances and for type-family instances can have thousands of candidates. We would like to find a matching candidate more efficiently than by linear search.

2.3 Non-solutions

At first sight, our task can be done easily: define a total order on *Expr* and use a standard finite map library. Indeed that

works, but it is terribly slow. A finite map is implemented as a binary search tree; at every node of this tree, we compare the key (an *Expr*, remember) with the key stored at the node; if it is smaller, go left; if larger, go right. Each lookup thus must perform a (logarithmic) number of potentially-full-depth comparisons of two expressions.

Another possibility might be to hash the *Expr* and use the hash-code as the lookup key. That would make lookup much faster, but it requires at least two full traversals of the key for every lookup: one to compute its hash code for every lookup, and a full equality comparison on a "hit" because hash-codes can collide.

But the killer is this: *neither binary search trees nor hashing is compatible with matching lookup*. For our purposes they are non-starters.

What other standard solutions to matching lookup are there, apart from linear search? The theorem proving and automated reasoning community has been working with huge sets of rewrite rules, just as we describe, for many years. They have developed term indexing techniques for the job [Sekar et al. 2001, Chapter 26], which attack the same problem from a rather different angle, as we discuss in Section 7.1.

3 Tries

A standard approach to a finite map in which the key has internal structure is to use a *trie*[1]. Generalising tries to handle an arbitrary algebraic data type as the key is a well established, albeit under-used, idea [Connelly and Morris 1995; Hinze 2000a]. We review these ideas in this section. Let us consider a simplified form of expression:

```
data Expr = Var Var | App Expr Expr
```

We omit lambdas for now, so that all *Var* nodes represent free variables, which are treated as constants. We will return to lambdas in Section 4.

3.1 The interface of a finite map

Building on the design of widely used functions in Haskell (see Fig. 1), we seek these basic operations:

```
emptyEM :: ExprMap v
lkEM    :: Expr → ExprMap v → Maybe v
atEM    :: Expr → TF v → ExprMap v → ExprMap v
```

The lookup function *lkEM*[2] has a type that is familiar from every finite map. The update function *atEM*, typically called *alter* in Haskell libraries, changes the value stored at a particular key. The caller provides a *value transformation function* *TF v*, an abbreviation for *Maybe v* → *Maybe v* (see Fig. 1). This function transforms the existing value associated with the key, if any (hence the input *Maybe*), to a new value, if any (hence the output *Maybe*). We can easily define *insertEM* and *deleteEM* from *atEM*:

```
insertEM :: Expr → v → ExprMap v → ExprMap v
insertEM e v = atEM e (\_ → Just v)

deleteEM :: Expr → ExprMap v → ExprMap v
deleteEM e = atEM e (\_ → Nothing)
```

You might wonder whether, for the purposes of this paper, we could just define *insert*, leaving *atEM* for the Appendix[3], but as we will see in Section 3.3, our approach using tries requires the generality of *atEM*.

These fundamental operations on a finite map must obey the following properties:

```
                lookup e empty           ≡ Nothing
                lookup e (alter e xt m)  ≡ xt (lookup e m)
e1 /= e2 ⇒ lookup e1 (alter e2 xt m) ≡ lookup e1 m
```

We also support other standard operations on finite maps, with types analogous to those in Fig. 1, including *unionEM*, *mapEM*, and *foldrEM*.

3.2 Tries: the basic idea

Here is a trie-based implementation for *Expr*:

```
data ExprMap v
  = EM { em_var :: Map Var v, em_app :: ExprMap (ExprMap v) }
```

Here *Map Var v* is any standard finite map (e.g. in *containers*[4]) keyed by *Var*, with values *v*. One way to understand this slightly odd data type is to study its lookup function:

```
lkEM :: Expr → ExprMap v → Maybe v
lkEM e (EM {em_var = var, em_app = app}) = case e of
  Var x      → Map.lookup x var
  App e1 e2 → case lkEM e1 app of
    Nothing → Nothing
    Just m1  → lkEM e2 m1
```

This function pattern-matches on the target *e*. The *Var* alternative says that to look up a variable occurrence, just look that variable up in the *em_var* field. But if the expression is an *App* e_1 e_2 node, we first look up e_1 in the *em_app* field, *which returns an ExprMap*. We then look up e_2 in that map. Each distinct e_1 yields a different *ExprMap* in which to look up e_2.

We can substantially abbreviate this code, at the expense of making it more cryptic, thus:

```
lkEM (Var x)      = em_var ⋙ Map.lookup x
lkEM (App e1 e2) = em_app ⋙ lkEM e1 ⋙= lkEM e2
```

The function *em_var* :: *ExprMap v* → *Map Var v* is the auto-generated selector that picks the *em_var* field from an *EM* record, and similarly *em_app*. The functions (⋙) and (⋙=) are right-associative forward composition operators,

[1] https://en.wikipedia.org/wiki/Trie

[2] We use short names *lkEM* and *atEM* consistently in this paper to reflect the single-column format.

[3] In the supplemental file `TrieMap.hs`

[4] https://hackage.haskell.org/package/containers

respectively monadic and non-monadic, that chain the individual operations together (see Fig. 1). Finally, we have η-reduced the definition, by omitting the *m* parameter. These abbreviations become quite worthwhile when we add more constructors, each with more fields, to the key data type.

Notice that in contrast to the approach of Section 2.3, *we never compare two expressions for equality or ordering*. We simply walk down the *ExprMap* structure, guided at each step by the next node in the target.

This definition is extremely short and natural. But it embodies a hidden complexity: *it requires polymorphic recursion*. The recursive call to *lkEM* e_1 instantiates *v* to a different type than the parent function definition. Haskell supports polymorphic recursion readily, provided you give type signature to *lkEM*, but not all languages do.

3.3 Modifying tries

It is not enough to look up in a trie – we need to *build* them too. First, we need an empty trie. Here is one way to define it:

```
emptyEM :: ExprMap v
emptyEM = EM {em_var = Map.empty, em_app = emptyEM}
```

It is interesting to note that *emptyEM* is an infinite, recursive structure: the *em_app* field refers back to *emptyEM*. We will change this definition in Section 3.5, but it works perfectly well for now. Next, we need to *alter* a triemap:

```
atEM :: Expr → TF v → ExprMap v → ExprMap v
atEM e tf m@(EM {em_var = var, em_app = app}) = case e of
  Var x      → m {em_var = Map.alter tf x var}
  App e1 e2 → m {em_app = atEM e1 (liftTF (atEM e2 tf)) app}
liftTF :: (ExprMap v → ExprMap v) → TF (ExprMap v)
liftTF f Nothing  = Just (f emptyEM)
liftTF f (Just m) = Just (f m)
```

In the *Var* case, we must just update the map stored in the *em_var* field, using the *Map.alter* function from Fig. 1. In the *App* case we look up e_1 in *app*; we should find a *ExprMap* there, which we want to alter with *tf*. We can do that with a recursive call to *atEM*, using *liftTF* for impedance-matching.

The *App* case shows why we need the generality of *alter*. Suppose we attempted to define an apparently-simpler *insert* operation. Its equation for (*App* e_1 e_2) would look up e_1 — and would then need to *alter* that entry (an *ExprMap*, remember) with the result of inserting (e_2, *v*). So we are forced to define *alter* anyway.

We can abbreviate the code for *atEM* using combinators, as we did in the case of lookup, and doing so pays dividends when the key is a data type with many constructors, each with many fields. However, the details are fiddly and not illuminating, so we omit them here. Indeed, for the same reason, in the rest of this paper we will typically omit the code for *alter*, though the full code is available in the Appendix.

3.4 Unions of maps

A common operation on finite maps is to take their union:

```
unionEM :: ExprMap v → ExprMap v → ExprMap v
```

In tree-based implementations of finite maps, such union operations can be tricky. The two trees, which have been built independently, might not have the same left-subtree/right-subtree structure, so some careful rebalancing may be required. But for tries there are no such worries – their structure is identical, and we can simply zip them together. There is one wrinkle: just as we had to generalise *insert* to *alter*, to accommodate the nested map in *em_app*, so we need to generalise *union* to *unionWith*:

```
unionWithEM :: (v → v → v)
            → ExprMap v → ExprMap v → ExprMap v
```

When a key appears on both maps, the combining function is used to combine the two corresponding values. With that generalisation, the code is as follows:

```
unionWithEM f (EM {em_var = var1, em_app = app1})
              (EM {em_var = var2, em_app = app2})
  = EM {em_var = Map.unionWith f var1 var2
       , em_app = unionWithEM (unionWithEM f) app1 app2}
```

It could hardly be simpler.

3.5 Folds and the empty map

The strange, infinite definition of *emptyEM* given in Section 3.3 works fine (in a lazy language at least) for lookup, alteration, and union, but it fails fundamentally when we want to *iterate* over the elements of the trie. For example, suppose we wanted to count the number of elements in the finite map; in *containers* this is the function *Map.size* (Fig. 1). We might attempt:

```
sizeEM :: ExprMap v → Int
sizeEM (EM {em_var = var, em_app = app})
  = Map.size var+???
```

We seem stuck because the size of the *app* map is not what we want: rather, we want to add up the sizes of its *elements*, and we don't have a way to do that yet. The right thing to do is to generalise to a fold:

```
foldrEM :: ∀v. (v → r → r) → r → ExprMap v → r
foldrEM k z (EM {em_var = var, em_app = app})
  = Map.foldr k z1 var
  where
    z1 = foldrEM kapp z (app :: ExprMap (ExprMap v))
    kapp m1 r = foldrEM k r m1
```

In the binding for z_1 we fold over *app*, using *kapp* to combine the map we find with the accumulator, by again folding over the map with *foldrEM*.

But alas, *foldrEM* will never terminate! It always invokes itself immediately (in z_1) on *app*; but that invocation will again recursively invoke *foldrEM*; and so on forever. The

solution is simple: we just need an explicit representation of the empty map. Here is one way to do it (we will see another in Section 3.7):

```
data ExprMap v = EmptyEM | EM { em_var :: ..., em_app :: ... }

emptyEM :: ExprMap v
emptyEM = EmptyEM

foldrEM :: (v → r → r) → r → ExprMap v → r
foldrEM k z EmptyEM = z
foldrEM k z (EM { em_var = var, em_app = app })
  = Map.foldr k z1 var
  where
    z1 = foldrEM kapp z app
    kapp m1 r = foldrEM k r m1
```

Equipped with a fold, we can easily define the size function, and another that returns the range of the map:

```
sizeEM :: ExprMap v → Int
sizeEM = foldrEM (λ_ n → n + 1) 0

elemsEM :: ExprMap v → [v]
elemsEM = foldrEM (:) [ ]
```

3.6 A type class for triemaps

Since all our triemaps share a common interface, it is useful to define a type class for them:

```
class Eq (Key tm) ⇒ TrieMap tm where
  type Key tm :: Type
  emptyTM      :: tm a
  lkTM         :: Key tm → tm a → Maybe a
  atTM         :: Key tm → TF a → tm a → tm a
  foldrTM      :: (a → b → b) → tm a → b → b
  unionWithTM :: (a → a → a) → tm a → tm a → tm a
  ...
```

The class constraint *TrieMap tm* says that the type *tm* is a triemap, with operations *emptyTM*, *lkTM* etc. The class has an *associated type* [Chakravarty et al. 2005], *Key tm*, a type-level function that transforms the type of the triemap into the type of *keys* of that triemap.

Now we can witness the fact that *ExprMap* is a *TrieMap*, like this:

```
instance TrieMap ExprMap where
  type Key ExprMap = Expr
  emptyTM = emptyEM
  lkTM    = lkEM
  atTM    = atEM
  ...
```

Having a class allow us to write helper functions that work for any triemap, such as

```
insertTM :: TrieMap tm ⇒ Key tm → v → tm v → tm v
insertTM k v = atTM k (\_ → Just v)

deleteTM :: TrieMap tm ⇒ Key tm → tm v → tm v
deleteTM k = atTM k (\_ → Nothing)
```

But that is not all. Suppose our expressions had multi-argument apply nodes, *AppV*, thus

```
data Expr = ... | AppV Expr [Expr]
```

Then we would need to build a trie keyed by a *list* of *Expr*. A list is just another algebraic data type, built with nil and cons, so we *could* use exactly the same approach, thus

```
lkLEM :: [Expr] → ListExprMap v → Maybe v
```

But rather than to define a *ListExprMap* for keys of type [*Expr*], and a *ListDeclMap* for keys of type [*Decl*], etc, we would obviously prefer to build a trie for lists of *any type*, like this [Hinze 2000a]:

```
instance TrieMap tm ⇒ TrieMap (ListMap tm) where
  type Key (ListMap tm) = [Key tm]
  emptyTM = emptyLM
  lkTM    = lkLM
  ...

data ListMap tm v = LM { lm_nil :: Maybe v
                       , lm_cons :: tm (ListMap tm v) }

emptyLM :: TrieMap tm ⇒ ListMap tm
emptyLM = LM { lm_nil = Nothing, lm_cons = emptyTM}

lkLM :: TrieMap tm ⇒ [Key tm] → ListMap tm v → Maybe v
lkLM [ ]     = lm_nil
lkLM (k : ks) = lm_cons >>> lkTM k >>> lkLM ks
```

The code for *atLM* and *foldrLM* is routine. Notice that all of these functions are polymorphic in *tm*, the triemap for the list elements.

3.7 Singleton maps, and empty maps revisited

Suppose we start with an empty map, and insert a value with a key (an *Expr*) such as

```
App (App (Var "f") (Var "x")) (Var "y")
```

Looking at the code for *atEM* in Section 3.3, you can see that because there is an *App* at the root, we will build an *EM* record with an empty *em_var*, and an *em_app* field that is... another *EM* record. Again the *em_var* field will contain an empty map, while the *em_app* field is a further *EM* record.

In effect, the key is linearised into a chain of *EM* records. This is great when there are a lot of keys with shared structure, but once we are in a sub-tree that represents a *single* key-value pair it is a rather inefficient way to represent the key. So a simple idea is this: when a *ExprMap* represents a single key-value pair, represent it as directly a key-value pair, like this:

```
data ExprMap v = EmptyEM
               | SingleEM Expr v    -- A single key/value pair
               | EM { em_var :: ..., em_app :: ... }
```

But in the triemap for for each new data type *X*, we will have to tiresomely repeat these extra data constructors, *EmptyX* and *SingleX*. For example we would have to add *EmptyList* and *SingleList* to the *ListMap* data type of Section 3.6. It

is better instead to abstract over the enclosed triemap, as follows[5]:

```
data SEMap tm v = EmptySEM
                | SingleSEM (Key tm) v
                | MultiSEM (tm v)

instance TrieMap tm ⇒ TrieMap (SEMap tm) where
  type Key (SEMap tm) = Key tm
  emptyTM = EmptySEM
  lkTM    = lkSEM
  atTM    = atSEM
  ...
```

The code for lookup practically writes itself. We abstract over *Maybe* with some *MonadPlus* combinators to enjoy forwards compatibility to Section 5:

```
lkSEM :: TrieMap tm ⇒ Key tm → SEMap tm v → Maybe v
lkSEM k m = case m of
  EmptySEM       → mzero
  SingleSEM pk v → guard (k == pk) ≫ pure v
  MultiSEM m     → lkTM k m
```

Where *mzero* means *Nothing* and *pure* means *Just*. The *guard* expression in the *SingleSEM* will return *Nothing* when the key expression *k* doesn't equate to the pattern expression *pk*. To test for said equality we require an *Eq* (*Key tm*) instance, hence it is a superclass of *TrieMap tm* in the **class** declaration in Section 3.6.

The code for alter is more interesting, because it governs the shift from *EmptySEM* to *SingleSEM* and thence to *MultiSEM*:

```
atSEM :: TrieMap tm
      ⇒ Key tm → TF v → SEMap tm v → SEMap tm v
atSEM k tf EmptySEM = case tf Nothing of Nothing → EmptySEM
                                         Just v  → SingleSEM k v
atSEM k1 tf (SingleSEM k2 v2) = if k1 == k2
  then case tf (Just v2) of
    Nothing → EmptySEM
    Just v' → SingleSEM k2 v'
  else case tf Nothing of
    Nothing → SingleSEM k2 v2
    Just v1 → MultiSEM (insertTM k1 v1 (insertTM k2 v2 emptyTM))
atSEM k tf (MultiSEM tm) = MultiSEM (atTM k tf tm)
```

Adding a new item to a triemap can turn *EmptySEM* into *SingleSEM* and *SingleSEM* into *MultiSEM*; and deleting an item from a *SingleSEM* turns it back into *EmptySEM*. You might wonder whether we can shrink a *MultiSEM* back to a *SingleSEM* when it has only one remaining element? Yes we can, but it takes quite a bit of code, and it is far from clear that it is worth doing so.

Finally, we need to re-define *ExprMap* and *ListMap* using *SEMap*:

[5] *SEMap* stands for "singleton or empty map".

```
type DBNum = Int
data DBEnv = DBE {dbe_next :: DBNum, dbe_env :: Map Var DBNum}

emptyDBE :: DBEnv
emptyDBE = DBE {dbe_next = 1, dbe_env = Map.empty}

extendDBE :: Var → DBEnv → DBEnv
extendDBE v (DBE {dbe_next = n, dbe_env = dbe})
  = DBE {dbe_next = n + 1, dbe_env = Map.insert v n dbe}

lookupDBE :: Var → DBEnv → Maybe DBNum
lookupDBE v (DBE {dbe_env = dbe}) = Map.lookup v dbe
```

Figure 2. De Bruijn leveling

```
type ExprMap    = SEMap ExprMap'
data ExprMap' v = EM {em_var :: ..., em_app :: ExprMap (ExprMap v)}

type ListMap    = SEMap ListMap'
data ListMap' tm v = LM {lm_nil :: ..., lm_cons :: tm (ListMap tm v)}
```

The auxiliary data types *ExprMap'* and *ListMap'* have only a single constructor, because the empty and singleton cases are dealt with by *SEMap*. We reserve the original, un-primed, names for the user-visible *ExprMap* and *ListMap* constructors.

3.8 Generic programming

We have not described a triemap *library*; rather we have described a *design pattern*. More precisely, given a new algebraic data type *X*, we have described a systematic way of defining a triemap, *XMap*, keyed by values of type *X*. Such a triemap is represented by a record:

- Each *constructor K* of *X* becomes a *field x_k* in *XMap*.
- Each *field* of a constructor *K* becomes a *nested triemap* in the type of the field *x_k*.
- If *X* is polymorphic then *XMap* becomes a triemap transformer, like *ListMap* above.

Actually writing out all this boilerplate code is tiresome, and it can of course be automated. One way to do so would be to use generic or polytypic programming, and Hinze describes precisely this [Hinze 2000a]. Another approach would be to use Template Haskell.

We do not develop either of these approaches here, because our focus is only the functionality and expressiveness of the triemaps. However, everything we do is compatible with an automated approach to generating boilerplate code.

4 Keys with binders

If our keys are expressions (in a compiler, say) they may contain binders, and we want insert and lookup to be insensitive to α-renaming. That is the challenge we address next. Here is our data type *Expr* from Section 2.1, which brings back binding semantics through the *Lam* constructor:

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

The key idea is simple: we perform De Bruijn numbering on the fly, renaming each binder to a natural number, from outside in. So, when inserting or looking up a key ($\lambda x.\ foo\ (\lambda y.\ x+y)$) we behave as if the key was ($\lambda.\ foo\ (\lambda.\#_1 + \#_2)$), where each $\#_i$ stands for an occurrence of the variable bound by the i'th lambda, counting from the root of the expression. In effect, then, we behave as if the data type was like this:

```
data Expr' = App Expr Expr | Lam Expr | FVar Var | BVar BoundKey
```

Notice (a) the *Lam* node no longer has a binder and (b) there are two sorts of *Var* nodes, one for free variables and one for bound variables, carrying a *BoundKey* (see below). We will not actually build a value of type *Expr'* and look that up in a trie keyed by *Expr'*; rather, we are going to *behave as if we did*. Here is the code (which uses Fig. 2):

```
data ModAlpha a = A DBEnv a
type AlphaExpr = ModAlpha Expr
instance Eq AlphaExpr where ...

type BoundKey = DBNum
type ExprMap = SEMap ExprMap'
data ExprMap' v
  = EM { em_fvar :: Map Var v          -- Free vars
       , em_bvar :: Map BoundKey v     -- Lambda-bound vars
       , em_app  :: ExprMap (ExprMap v)
       , em_lam  :: ExprMap v }

instance TrieMap ExprMap' where
  type Key ExprMap' = AlphaExpr
  lkTM = lkEM
  ...

lkEM :: AlphaExpr → ExprMap' v → Maybe v
lkEM (A bve e) = case e of
  Var v → case lookupDBE v bve of
    Nothing → em_fvar ⋙ Map.lookup v
    Just bv → em_bvar ⋙ Map.lookup bv
  App e1 e2 → em_app ⋙ lkTM (A bve e1) ≫= lkTM (A bve e2)
  Lam v e   → em_lam ⋙ lkTM (A (extendDBE v bve) e)

lookupClosedExpr :: Expr → ExprMap v → Maybe v
lookupClosedExpr e = lkEM (A emptyDBE e)
```

We maintain a *DBEnv* (cf. Fig. 2) that maps each lambda-bound variable to its De Bruijn level[6] [de Bruijn 1972], its *BoundKey*. The expression we look up — the first argument of *lkEM* — becomes an *AlphaExpr*, which is a pair of a *DBEnv* and an *Expr*. At a *Lam* node we extend the *DBEnv*. At a *Var* node we look up the variable in the *DBEnv* to decide whether it is lambda-bound or free, and behave appropriately[7].

[6]The De Bruijn *index* of the occurrence of a variable v counts the number of lambdas between the occurrence of v and its binding site. The De Bruijn *level* of v counts the number of lambdas between the root of the expression and v's binding site. It is convenient for us to use *levels*.

[7]The implementation from the Appendix uses more efficient *IntMap*s for mapping *BoundKey*. *IntMap* is a itself trie data structure, so it could have made a nice "Tries all the way down" argument. But we found it distracting to present here, hence regular ordered *Map*.

The construction of Section 3.7, to handle empty and singleton maps, applies without difficulty to this generalised map. To use it we must define an instance *Eq AlphaExpr* to satisfy the *Eq* super class constraint on the trie key, so that we can instantiate *TrieMap ExprMap'*. That *Eq AlphaExpr* instance simply equates two α-equivalent expressions in the standard way. The code for *alter* and *foldr* holds no new surprises either.

And that is really all there is to it: it is remarkably easy to extend the basic trie idea to be insensitive to α-conversion and even mix in trie transformers such as *SEMap* at no cost other than writing two instance declarations.

5 Tries that match

A key advantage of tries over hash-maps and balanced trees is that we can support *matching* (Section 2.2).

5.1 What "matching" means

Semantically, a matching trie can be thought of as a set of *entries*, each of which is a (pattern, value) pair. What is a pattern? It is a pair (vs, p) where

- vs is a set of *pattern variables*, such as $[a, b, c]$.
- p is a *pattern expression*, such as $f\ a\ (g\ b\ c)$.

A pattern may of course contain free variables (not bound by the pattern), such as f and g in the above example, which are regarded as constants by the algorithm. A pattern (vs, p) *matches* a target expression e iff there is a unique substitution S whose domain is vs, such that $S(p) = e$.

We allow the same variable to occur more than once in the pattern. For example, the pattern $([x], f\ x\ x)$ should match targets like $(f\ 1\ 1)$ or $(f\ (g\ v)\ (g\ v))$, but not $(f\ 1\ (g\ v))$. This ability is important if we are to use matching tries to implement class or type-family look in GHC.

In implementation terms, we can characterise matching by the following type class:

```
class (Eq (Pat k), MonadPlus (Match k)) ⇒ Matchable k where
  type Pat   k :: Type
  type Match k :: Type → Type
  match :: Pat k → k → Match k ()
```

For any key type *k*, the *match* function takes a pattern of type *Pat k*, and a key of type *k*, and returns a monadic match of type *Match k ()*, where *Pat* and *Match* are associated types of *k*. Matching can fail or can return many results, so *Match k* is a *MonadPlus*:

```
mzero :: MonadPlus m ⇒ m a
mplus :: MonadPlus m ⇒ m a → m a → m a
```

To make this more concrete, here is a possible *Matchable* instance for *AlphaExpr*:

```
instance Matchable AlphaExpr where
  type Pat   AlphaExpr = PatExpr
  type Match AlphaExpr = MatchExpr
  match = matchE
```

Let's look at the pieces, one at a time.

5.1.1 Patterns.

A pattern *PatExpr* over *AlphaExpr* can be defined like this:

```
data PatExpr = P PatKeys AlphaExpr
type PatKeys = Map PatVar PatKey
type PatVar  = Var
type PatKey  = DBNum
```

A pattern *PatExpr* is a pair of an *AlphaExpr* and a *PatKeys* that maps each of the quantified pattern variables to a canonical De Bruijn *PatKey*. Just as in Section 4, *PatKeys* make the pattern insensitive to the particular names, and order of quantification, of the pattern variables. We canonicalise the quantified pattern variables before starting a lookup, numbering pattern variables in the order they appear in a left-to-right scan. For example

Original pattern	Canonical *PatExpr*
$([a,b], f\ a\ b\ a)$	$P\ [a \mapsto 1, b \mapsto 2]\ (f\ a\ b\ a)$
$([x,g], f\ (g\ x)$	$P\ [x \mapsto 2, g \mapsto 1]\ (f\ (g\ x))$

5.1.2 The matching monad.

There are many possible implementation of the *MatchExpr* monad, but here is one:

```
type MatchExpr v = MR (StateT Subst [] v)
type Subst = Map PatKey Expr
```

The *MatchExpr* type is isomorphic to $Subst \to [(v, Subst)]$; matching takes a substitution for pattern variables (more precisely, their canonical *PatKey*s), and yields a possibly-empty list of values paired with an extended substitution. Notice that the substitution binds pattern keys to *Expr*, not *AlphaExpr*, because the pattern variables cannot mention lambda-bound variables within the target expression.

The formulation in terms of *StateT* endows us with just the right *Monad* and *MonadPlus* instances, as well as favorable performance because of early failure on contradicting *match*es and the ability to share work done while matching a shared prefix of multiple patterns.

The monad comes with some auxiliary functions that we will need later:

```
runMatchExpr :: MatchExpr v → [(Subst, v)]
liftMaybe    :: Maybe v → MatchExpr v
refineMatch :: (Subst → Maybe Subst) → MatchExpr ()
```

Their semantics should be apparent from their types. For example, *runMatchExpr* runs a *MatchExpr* computation, starting with an empty *Subst*, and returning a list of all the successful (*Subst*, *v*) matches.

5.1.3 Matching summary.

The implementation of *matchE* is entirely straightforward, using simultaneous recursive descent over the pattern and target. The code is given in the Appendix.

The key point is this: nothing in this section is concerned with tries. Here we are simply concerned with the mechanics of matching, and its underlying monad. There is ample room for flexibility. For example, if the key terms had two kinds of variables (say type variables and term variables) we could easily define *Match* to carry two substitutions; or *Match* could return just the first result rather than a list of all of them; and so on.

5.2 The matching trie class

The core of our matching trie is the class *MTrieMap*, which generalises the *TrieMap* class of Section 3.6:

```
class Matchable (MKey tm) ⇒ MTrieMap tm where
  type MKey tm :: Type
  emptyMTM :: tm v
  lkMTM    :: MKey tm → tm v → Match (MKey tm) v
  atMTM    :: Pat (MKey tm) → TF v → tm v → tm v
```

The lookup function takes a key of type *MKey tm* as before, but it returns something in the *Match* monad, rather than the *Maybe* monad. The *atMTM* takes a *pattern* (rather than just a key), of type *Pat* (*MKey tm*), and alters the trie's value at that pattern[8].

We can generalise *SEMap* (Section 3.7) in a similar way:

```
data MSEMap tm v = EmptyMSEM
                 | SingleMSEM (Pat (MKey tm)) v
                 | MultiMSEM (tm v)

instance MTrieMap tm ⇒ MTrieMap (MSEMap tm) where
  type MKey (MSEMap tm) = MKey tm
  emptyMTM = EmptyMSEM
  lkMTM    = lkMSEM
  atMTM    = atMSEM
```

Notice that *SingleMSEM* contains a *pattern*, not merely a *key*, unlike *SingleSEM* in Section 3.7. The code for *lkMSEM* and *atMSEM* is straightforward; we give the former here, leaving the latter for the Appendix

```
lkMSEM :: MTrieMap tm ⇒ MKey tm → MSEMap tm a
                      → Match (MKey tm) a
lkMSEM k EmptyMSEM          = mzero
lkMSEM k (MultiMSEM m)      = lkMTM k m
lkMSEM k (SingleMSEM pat v) = match pat k ≫ pure v
```

Notice the call to *mzero* to make the lookup fail if the map is empty; and, in the *SingleMSEM* case, the call *match* to match the pattern against the key.

5.3 Matching tries for *Expr*

Next, we show how to implement a matching triemap for our running example, *AlphaExpr*. The data type follows closely the pattern we developed for *ExprMap* (Section 4):

```
type MExprMap = MSEMap MExprMap'

data MExprMap' v
  = MM { mm_fvar :: Map Var v          -- Free var
       , mm_bvar :: Map BoundKey v     -- Bound var
       , mm_pvar :: Map PatKey v       -- Pattern var
```

[8] Remember, a matching trie represents a set of (pattern,value) pairs.

```
        , mm_app  :: MExprMap (MExprMap v)
        , mm_lam  :: MExprMap v }

instance MTrieMap MExprMap' where
  type MKey MExprMap' = AlphaExpr
  emptyMTM = ...   -- boring
  lkMTM    = lookupPatMM
  atMTM    = alterPatMM
```

The main difference is that we add an extra field *mm_pvar* to *MExprMap'*, for occurrences of a pattern variable. You can see how this field is used in the lookup code:

```
lookupPatMM :: ∀v. AlphaExpr → MExprMap' v → MatchExpr v
lookupPatMM ae@(A bve e) (MM {..})
  = rigid `mplus` flexi
  where
    rigid = case e of
      Var x     → case lookupDBE x bve of
        Just bv  → mm_bvar ▷ liftMaybe ∘ Map.lookup bv
        Nothing  → mm_fvar ▷ liftMaybe ∘ Map.lookup x
      App e1 e2 → mm_app  ▷ lkMTM (A bve e1)
                          >=> lkMTM (A bve e2)
      Lam x e   → mm_lam  ▷ lkMTM (A (extendDBE x bve) e)

    flexi = mm_pvar ▷ IntMap.toList ▷ map match_one ▷ msum

    match_one :: (PatVar, v) → MatchExpr v
    match_one (pv, v) = matchPatVarE pv ae ≫ pure v
```

Matching lookup on a trie matches the target expression against *all patterns the trie represents*. The *rigid* case is no different from exact lookup; compare the code for *lkEM* in Section 4. The only difference is that we need *liftMaybe* (from Section 5.1.2) to take the *Maybe* returned by *Map.lookup* and lift it into the *MatchExpr* monad.

The *flexi* case handles the triemap entries whose pattern is simply one of the quantified pattern variables; these entries are stored in the new *mm_pvar* field. We enumerate these entries with *toList*, to get a list of (*PatVar*, *v*) pairs, match each such pair against the target with *match_one*, and finally accumulate all the results with *msum*. In turn *match_one* uses *matchParVarE* to match the pattern variable with the target and, if successful, returns corresponding value *v*.

The *matchPatVarE* function does the heavy lifting, using some simple auxiliary functions whose types are given below:

```
matchPatVarE :: PatKey → AlphaExpr → MatchExpr ()
matchPatVarE pv (A bve e) = refineMatch $ λms →
  case Map.lookup pv ms of
    Nothing   -- pv is not bound
      | noCaptured bve e → Just (Map.insert pv e ms)
      | otherwise        → Nothing
    Just sol  -- pv is already bound
      | noCaptured bve e
      , eqExpr e sol     → Just ms
      | otherwise        → Nothing

eqExpr     :: Expr → Expr → Bool
noCaptured :: DBEnv → Expr → Bool
```

To match a pattern variable *pv* against an expression (*A bve e*), we first look up *pv* in the current substitution (obtained from the *MatchExpr* state monad. If *pv* is not bound we simply extend the substitution.

But wait! Consider matching the pattern ($[p], \lambda x \to p$) against the target ($\lambda y \to 3$). That's fine: we should succeed, binding *a* to 3. But suppose we match that same pattern against target ($\lambda y \to y$). It would be nonsense to "succeed" binding *a* to *y*, because *y* is locally bound within the target. Hence the *noCaptured* test, which returns *True* iff the expression does not mention any of the locally-bound variables.

If *pv* is already bound in the substitution, we have a repeated pattern variable (see Section 5.1), and we must check that the target expression is equal (using *eqExpr*) to the the one already bound to *pv*. Once again, however, we must check that the target does not contain any locally-bound variables, hence the *noCaptured* check.

lookupPatMM is the trickiest case. The code for *alterPatMM*, and the other operations of the class, is very straightforward, and is given in the Appendix.

5.4 The external API

The matching tries we have described so far use canonical pattern keys, a matching monad, and other machinery that should be hidden from the client. We seek an external API more like this:

```
type PatMap :: Type → Type
alterPM    :: ([Var], Expr) → TF v → PatMap v → PatMap v
lookupPM   :: Expr → PatMap v → [(PatSubst, v)]
type PatSubst = [(Var, Expr)]
```

When altering a *PatMap* we supply a client-side pattern, which is just a pair ([*Var*], *Expr*) of the quantified pattern variables and the pattern. When looking up in a *PatMap* we supply a target expression, and get back a list of matches, each of which is a pair of the value and the substitution for those original pattern variables that made the pattern equal to the target.

So *alterPM* must canonicalise the client-side pattern variables before altering the trie; that is easy enough. But how can *lookupPM* recover the client-side *PatSubst*? Somehow we must remember the canonicalisation used when *inserting* so that we can invert it when *matching*. For example, suppose we insert the two (pattern, value pairs)

$$(([p], f\ p\ True), v_1) \quad \text{and} \quad (([q], f\ q\ False), v_2)$$

Both patterns will canonicalise their (sole) pattern variable to the De Bruin index 1. So if we look up the target (*f e True*) the *MatchExpr* monad will produce a final *Subst* that maps $[1 \mapsto e]$, paired with the value v_1. But we want to return ([("p", *e*)], v_1) to the client, a *PatSubst* that uses the client variable "p", not the internal index 1.

The solution is simple enough: *we store the mapping in the triemap's domain*, along with the values, thus:

```
type PatMap v = MExprMap (PatKeys, v)
```

Now the code writes itself. Here is *alterPM*:

```
alterPM :: ∀v. ([Var], Expr) → TF v → PatMap v → PatMap v
alterPM (pvars, e) tf pm = atMTM pat ptf pm
  where
    pks :: PatKeys = canonPatKeys pvars e
    pat :: PatExpr = P pks (A emptyDBE e)
    ptf :: TF (PatKeys, v)
    ptf Nothing       = fmap (λv → (pks, v)) (tf Nothing)
    ptf (Just (_, v)) = fmap (λv → (pks, v)) (tf (Just v))
canonPatKeys :: [Var] → Expr → PatKeys
```

The auxiliary function *canonPatKeys* takes the client-side pattern (*pvars*, *e*), and returns a *PatKeys* (Section 5.1.1) that maps each pattern variable to its canonical De Bruijn index. *canonPatKeys* is entirely straightforward: it simply walks the expression, numbering off the pattern variables in left-to-right order.

Then we can simply call the internal *atMTM* function, passing it: a canonical *pat* :: *PatExpr*; and a transformer *ptf* :: *TF* (*PatKeys*, *v*) that will pair the *PatKeys* with the value supplied by the user via *tf* :: *TF v*. Lookup is equally easy:

```
lookupPM :: Expr → PatMap v → [(PatSubst, v)]
lookupPM e pm
  = [(Map.toList (subst `Map.compose` pks), x)
    | (subst, (pks, x)) ← runMatchExpr $
                          lkMTM (A emptyDBE e) pm]
```

We use *runMatchExpr* to get a list of successful matches, and then pre-compose (see Fig. 1) the internal *Subst* with the *PatKeys* mapping that is part of the match result. We turn that into a list to get the client-side *PatSubst*. The only tricky point is what to do with pattern variables that are not substituted. For example, suppose we insert the pattern ([*p*, *q*], *f p*). No lookup will bind *q*, because *q* simply does not appear in the pattern. One could reject this on insertion, but here we simply return a *PatSubst* with no binding for *q*.

5.5 Most specific match, and unification

It is tempting to ask: can we build a lookup that returns only the *most specific* matches? And, can we build a lookup that returns all values whose patterns *unify* with the target. Both would have useful applications, in GHC at least.

However, both seem difficult to achieve. All our attempts became mired in complexity, and we leave this for further work, and as a challenge for the reader. We outline some of the difficulties of unifying lookup in Appendix B.

6 Evaluation

So far, we have seen that trie maps offer a significant advantage over other kinds of maps like ordered maps or hash maps: the ability to do a matching lookup (in Section 5). In this section, we will see that query performance is another advantage. Our implementation of trie maps in Haskell can generally compete in performance with other map data structures, while significantly outperforming traditional map implementations on some operations. Not bad for a data structure that we can also extend to support matching lookup!

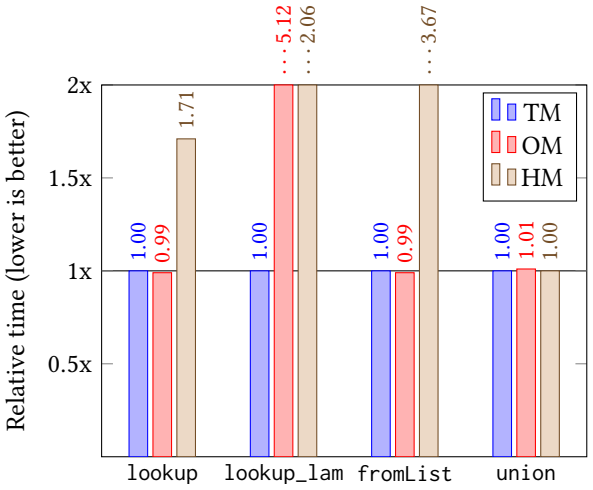

Figure 3. Benchmarks comparing our trie map (TM) to ordered maps (OM) and hash maps (HM)

We took runtime measurements of the (non-matching) *ExprMap* data structure on a selection of workloads, conducted using the *criterion*[9] benchmarking library[10]. Fig. 3 presents a quick overview of the results.

Appendix A is an extended version of this section, featuring a more in-depth analysis, finer runtime as well as space measurements and indicators for statistical significance.

Setup. All benchmarks except `fromList` are handed a prebuilt map containing 10000 expressions, each consisting of roughly 100 *Expr* data constructors drawn from a pseudo-random source with a fixed (and thus deterministic) seed.

We compare three different non-matching map implementations, simply because we were not aware of other map data structures with matching lookup modulo α-equivalence and we wanted to compare apples to apples. The *ExprMap* forms the baseline. Asymptotics are given with respect to map size n and key expression size k:

- *ExprMap* (designated "TM" in Fig. 3) is the trie map implementation from this paper. Insertion and lookup

[9] https://hackage.haskell.org/package/criterion

[10] The benchmark machine runs Ubuntu 18.04 on an Intel Core i5-8500 with 16GB RAM. All programs were compiled on GHC 9.0.2 with `-O2 -fproc-alignment=64` to eliminate code layout flukes and run with `+RTS -A128M -RTS` for 128MB space in generation 0 in order to prevent major GCs from skewing the results.

perform at most one full traversal of the key, so performance should scale with $O(k)$.

- *Map Expr* (designated "OM") is the ordered map implementation from the mature, well-optimised *containers*[11] library. It uses size balanced trees under the hood [Adams 1993]. Thus, lookup and insert operations incur an additional log factor in the map size n, for a total of $O(k \log n)$ factor compared to both other maps.
- *HashMap Expr* (designated "HM") is an implementation of hash array mapped tries [Bagwell 2001] from the *unordered-containers*[12] library. Like *ExprMap*, map access incurs a full traversal of the key to compute a hash and then a $O(\log_{32} n)$ lookup in the array mapped trie, as well as an expected constant number of key comparisons to resolve collisions. The log factor can be treated like a constant for all intents and purposes, so lookup and insert is effectively in $O(k)$.

Some clarification as to what our benchmarks measure:

- The `lookup` benchmark looks up every expression that is part of the map. So for a map of size 10000, we perform 10000 lookups of expressions each of which have approximately size 100.
- `lookup_lam` is like `lookup`, but wraps a shared prefix of 100 layers of (*Lam* `"$"`) around each expression.
- `fromList` benchmarks a naïve *fromList* implementation on *ExprMap* against the tuned *fromList* implementations of the other maps, measuring map creation performance from batches.

Querying. The results suggest that *ExprMap* is about as fast as *Map Expr* for completely random expressions in `lookup`. But in a more realistic scenario, at least some expressions share a common prefix, which is what `lookup_lam` embodies. There we can see that *ExprMap* wins against *Map Expr* by a huge margin: *ExprMap* looks at the shared prefix exactly once on lookup, while *Map* has to traverse the shared prefix of length $O(k)$ on each of its $O(\log n)$ comparisons.

Although *HashMap* loses on most benchmarks compared to *ExprMap* and *Map*, most measurements were consistently at most a factor of two slower than *ExprMap*. We believe that is due to the fact that it is enough to traverse the *Expr* twice during lookup barring any collisions (hash and then equate with the match), thus it is expected to scale similarly as *ExprMap*. Thus, both *ExprMap* and *HashMap* perform much more consistently than *Map*.

Modification. While *ExprMap* consistently wins in query performance, the edge is melting into insignificance for `fromList` and `union`. One reason is the uniform distribution of expressions in these benchmarks, which favors *Map*. Still, it is a surprise that the naïve *fromList* implementations of *ExprMap* and *Map* as list folds beat the one of *HashMap*, although the latter has a tricky, performance-optimised implementation using transient mutability.

What would a non-naïve version of *fromList* for *ExprMap* look like? Perhaps the process could be sped up considerably by partitioning the input list according to the different fields of *ExprMap* and then calling the *fromList* implementations of the individual fields in turn. The process would be very similar to discrimination sort [Henglein 2008], which is a generalisation of radix sort to tree-like data and very close to tries. Indeed, the *discrimination*[13] library provides such an optimised $O(n)$ *toMap* implementation for *Map*.

7 Related work

7.1 Matching triemaps in automated reasoning

Matching triemaps, a kind of *term index*, have been used in the automated reasoning community for decades. An automated reasoning system has hundreds or thousands of axioms, each of which is quantified over some variables (just like the RULEs described in Section 2.2). Each of these axioms might apply at any sub-tree of the term under consideration, so efficient matching of many axioms is absolutely central to the performance of these systems.

This led to a great deal of work on so-called *discrimination trees*, starting in the late 1980's, which is beautifully surveyed in the Handbook of Automated Reasoning [Sekar et al. 2001, Chapter 26]. All of this work typically assumes a single, fixed, data type of "first order terms" like this[14]

```
data MKey = Node Fun [MKey]
```

where *Fun* is a function symbol, and each such function symbol has a fixed arity. Discrimination trees are described by imagining a pre-order traversal that (uniquely, since function symbols have fixed arity) converts the *MKey* to a list of type [*Fun*], and treating that as the key. The map is implemented like this:

```
data DTree v = DVal v | DNode (Map Fun DTree)

lookupDT :: [Fun] → DTree v → Maybe v
lookupDT []       (DVal v)    = Just v
lookupDT (f : fs) (DNode m) = case Map.lookup f m of
                                Just dt → lookupDT fs dt
                                Nothing → Nothing
lookupDT _        _           = Nothing
```

Each layer of the tree branches on the first *Fun*, and looks up the rest of the [*Fun*] in the appropriate child. Extending this basic setup with matching is done by some kind of backtracking.

Discrimination trees are heavily used by theorem provers, such as Coq, Isabelle, and Lean. Moreover, discrimination

[11] https://hackage.haskell.org/package/containers
[12] https://hackage.haskell.org/package/unordered-containers
[13] https://hackage.haskell.org/package/discrimination
[14] Binders in terms do not seem to be important in these works, although they could be handled fairly easily by a De Bruijn pre-pass.

trees have been further developed in a number of ways. Vampire uses *code trees* which are a compiled form of discrimination tree that stores abstract machine instructions, rather than a data structure at each node of the tree [Voronkov 1995]. Spass [Weidenbach et al. 2009] uses *substitution trees* [Graf and Meyer 1996], a refinement of discrimination trees that can share common *sub-trees* not just common *prefixes*. (It is not clear whether the extra complexity of substitution trees pays its way.) Z3 uses *E-matching code trees*, which solve for matching modulo an ever-growing equality relation, useful in saturation-based theorem provers. All of these techniques except E-matching are surveyed in Sekar et al. [2001].

If we applied our ideas to *MKey* we would get a single-field triemap which (just like *lookupDT*) would initially branch on *Fun*, and then go though a chain of *ListMap* constructors (which correspond to the *DNode* above). You have to squint pretty hard — for example, we do the pre-order traversal on the fly — but the net result is very similar, although it is arrived at by entirely different thought process.

Many of the insights of the term indexing world re-appear, in different guise, in our triemaps. For example, when a variable is repeated in a pattern we can eagerly check for equality during the match, or instead gather an equality constraint and check those constraints at the end [Sekar et al. 2001, Section 26.14].

7.2 Haskell triemaps

Trie data structures have found their way into numerous Haskell packages over time. There are trie data structures that are specific to *String*, like the *StringMap*[15] package, or polymorphically, requiring just a type class for trie key extraction, like the *TrieMap*[16] package. None of these libraries describe how to index on expression data structures modulo α-equivalence or how to perform matching lookup.

Memoisation has been a prominent application of tries in Haskell [Elliott 2008a,b; Hinze 2000b]. Given a function f, the idea is to build an *infinite*, lazily-evaluated trie, that maps every possible argument x to (a thunk for) $(f\ x)$. Now, a function call becomes a lookup in the trie. The ideas are implemented in the *MemoTrie*[17] library. For memo tries, operations like alter, insert, union, and fold are all irrelevant: the infinite trie is built once, and then used only for lookup.

A second strand of work concerns data type generic, or polytypic, approaches to generating tries, which nicely complements the design-pattern approach of this paper (Section 3.8). Hinze [2000a] describes the polytypic approach, for possibly parameterised and nested data types in some detail, including the realisation that we need *alter* and *unionWith* in order to define *insert* and *union*. A generalisation of those ideas then led to *functor-combo*[18]. The *representable-tries*[19] library observes that trie maps are representable functors and then vice versa tries to characterise the sub-class of representable functors for which there exists a trie map implementation.

The *twee-lib*[20] library defines a simple term index data structure based on discrimination trees for the *twee* equation theorem prover. We would arrive at a similar data structure in this paper had we started from an expression data type

```
data Expr = App Con [Expr] | Var Var
```

In contrast to our *ExprMap*, *twee*'s *Index* does path compression not only for paths ending in leaves (as we do) but also for internal paths, as is common for radix trees.

It is however unclear how to extend *twee*'s *Index* to support α-equivalence, hence we did not consider it for our benchmarks in Section 6.

Acknowledgments

We warmly thank Leonardo de Moura and Edward Yang for their very helpful feedback.

8 Conclusion

We presented trie maps as an efficient data structure for representing a set of expressions modulo α-equivalence, re-discovering polytypic deriving mechanisms described by Hinze [2000a]. Subsequently, we showed how to extend this data structure to make it aware of pattern variables in order to interpret stored expressions as patterns. The key innovation is that the resulting trie map allows efficient matching lookup of a target expression against stored patterns. This pattern store is quite close to discrimination trees [Sekar et al. 2001], drawing a nice connection to term indexing problems in the automated theorem proving community.

References

Stephen Adams. 1993. Functional Pearls Efficient sets—a balancing act. *Journal of Functional Programming* 3, 4 (1993), 553–561. https://doi.org/10.1017/S0956796800000885

Phil Bagwell. 2001. *Ideal Hash Trees.* Technical Report. Infoscience Department, École Polytechnique Fédérale de Lausanne.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated Type Synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia) *(ICFP '05)*. Association for Computing Machinery, New York, NY, USA, 241–253. https://doi.org/10.1145/1086365.1086397

Richard Connelly and F Lockwood Morris. 1995. A generalization of the trie data structure. *Mathematical structure in computer science* 5 (1995), 381–418. Issue 3.

N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

[15] https://hackage.haskell.org/package/StringMap
[16] https://hackage.haskell.org/package/TrieMap
[17] https://hackage.haskell.org/package/MemoTrie
[18] https://hackage.haskell.org/package/functor-combo
[19] https://hackage.haskell.org/package/representable-tries
[20] https://hackage.haskell.org/package/twee-lib

Conal Elliott. 2008a. Composing memo tries. http://conal.net/blog/posts/composing-memo-tries.

Conal Elliott. 2008b. Elegant memoization with functional memo tries. http://conal.net/blog/posts/elegant-memoization-with-functional-memo-tries.

P Graf and C Meyer. 1996. Advanced indexing operations on substitution trees. In *Proc International Conference on Automated Deduction (CADE'96), LNCS 1104*, MA McRobbie and Slaney JK (Eds.). Springer.

Fritz Henglein. 2008. Generic Discrimination: Sorting and Paritioning Unshared Data in Linear Time. *SIGPLAN Not.* 43, 9 (Sept. 2008), 91–102. https://doi.org/10.1145/1411203.1411220

Ralf Hinze. 2000a. Generalizing Generalized Tries. *Journal of Functional Programming* 10, 4 (2000), 327–351. http://journals.cambridge.org/action/displayAbstract?aid=59745

Ralf Hinze. 2000b. Memo Functions, Polytypically!. In *Proceedings of the 2nd Workshop on Generic Programming, Lima, Portugal*. 17–32.

Shayan Najd and Simon Peyton Jones. 2017. Trees that grow. *Journal of Universal Computer Science (JUCS)* 23 (January 2017), 47–62. https://www.microsoft.com/en-us/research/publication/trees-that-grow/

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop* (2001 haskell workshop ed.). ACM SIGPLAN, ACM. https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/

André Luís De Medeiros Santos. 1995. *Compilation by Transformation in Non-Strict Functional Languages*. Ph. D. Dissertation. University of Glasgow.

R Sekar, IV Ramakrishnan, and A Voronkov. 2001. *Handbook of Automated Reasoning*. Vol. 2. Elsevier.

A Voronkov. 1995. The anatomy of Vampire: Implementing bottom-up procedures with code trees. *Journal of Automated Reasoning* 15 (1995), 237–265. Issue 2.

Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. 2009. SPASS Version 3.5. In *Proc International Conference on Automated Deduction (CADE)*. Springer, 140–145.

A Evaluation

So far, we have seen that trie maps offer a significant advantage over other kinds of maps like ordered maps or hash maps: the ability to do a matching lookup (in Section 5). In this section, we will see that query performance is another advantage. Our implementation of trie maps in Haskell can generally compete in performance with other map data structures, while significantly outperforming traditional map implementations on some operations. Not bad for a data structure that we can also extend to support matching lookup!

A.1 Runtime

We measured the runtime performance of the (non-matching) *ExprMap* data structure on a selection of workloads, conducted using the *criterion*[21] benchmarking library[22]. Table 1 presents an overview of the results, while Table 2 goes into more detail on some configurations.

Setup. All benchmarks except the `fromList*` variants are handed a pre-built map containing N expressions, each consisting of roughly N *Expr* data constructors, and drawn from a pseudo-random source with a fixed (and thus deterministic) seed. N is varied between 10 and 1000.

We compare three different non-matching map implementations, simply because we were not aware of other map data structures with matching lookup modulo α-equivalence and we wanted to compare apples to apples. The *ExprMap* forms the baseline. Asymptotics are given with respect to map size n and key expression size k:

- *ExprMap* (designated "TM" in Table 1) is the trie map implementation from this paper. Insertion and lookup and have to perform a full traversal of the key, so performance should scale with $O(k)$, where k is the key *Expr* that is accessed.
- *Map Expr* (designated "OM") is the ordered map implementation from the mature, well-optimised *containers*[23] library. It uses size balanced trees under the hood [Adams 1993]. Thus, lookup and insert operations incur an additional log factor in the map size n, for a total of $O(k \log n)$ factor compared to both other maps.
- *HashMap Expr* (designated "HM") is an implementation of hash array mapped tries [Bagwell 2001] from the *unordered-containers*[24] library. Like *ExprMap*, map access incurs a full traversal of the key to compute a hash and then a $O(\log_{32} n)$ lookup in the array mapped trie. The log factor can be treated like a constant for all intents and purposes, so lookup and insert is effectively in $O(k)$.

Benchmarks ending in `_lam`, `_app1`, `_app2` add a shared prefix to each of the expressions before building the initial map:

- `_lam` wraps N layers of (*Lam* `"$"`) around each expression
- `_app1` wraps N layers of (*Var* `"$"` `` `App` ``) around each expression
- `_app2` wraps N layers of (`` `App` `` *Var* `"$"`) around each expression

where `"$"` is a name that doesn't otherwise occur in the generated expressions.

- The `lookup*` family of benchmarks looks up every expression that is part of the map. So for a map of size 100, we perform 100 lookups of expressions each of which have approximately size 100. `lookup_one` looks up just one expression that is part of the map.
- `insert_lookup_one` inserts a random expression into the initial map and immediately looks it up afterwards.

[21] https://hackage.haskell.org/package/criterion

[22] The benchmark machine runs Ubuntu 18.04 on an Intel Core i5-8500 with 16GB RAM. All programs were compiled with `-O2 -fproc-alignment=64` to eliminate code layout flukes and run with `+RTS -A128M -RTS` for 128MB space in generation 0 in order to prevent major GCs from skewing the results.

[23] https://hackage.haskell.org/package/containers

[24] https://hackage.haskell.org/package/unordered-containers

Table 1. Benchmarks of different operations over our trie map *ExprMap* (TM), ordered maps *Map Expr* (OM) and hash maps *HashMap Expr* (HM), varying the size parameter N. Each map is of size N (so $M = N$) and the expressions it contains are also each of size N (so $E = N$). We give the measurements of OM and HM relative to absolute runtime measurements for TM. Lower is better. Digits whose order of magnitude is no larger than that of twice the standard deviation are marked by squiggly lines.

N	10			100			1000		
Data structure	TM	OM	HM	TM	OM	HM	TM	OM	HM
`lookup`	**3.49µs**	1.07	1.49	**213µs**	1.08	1.80	**30.4ms**	1.00	1.68
`lookup_app1`	**6.23µs**	1.76	1.51	**481µs**	3.42	1.64	**64.4ms**	4.82	1.65
`lookup_app2`	**6.69µs**	1.12	1.43	**604µs**	1.42	1.42	**100ms**	3.12	1.30
`lookup_lam`	**7.56µs**	2.12	1.83	**539µs**	4.72	2.09	**72.6ms**	6.63	1.94
`lookup_one`	**323ns**	1.00	1.64	**1.55µs**	1.01	1.70	23.0µs	**1.00**	1.73
`fold`	472ns	0.37	**0.30**	6.01µs	**0.30**	0.33	91.9µs	**0.30**	0.36
`insert_lookup_one`	**64.3ns**	2.19	1.45	**2.46µs**	1.15	1.54	**26.2µs**	1.04	1.75
`fromList`	3.71µs	**0.67**	0.98	129µs	**0.91**	1.98	**13.1ms**	1.01	2.80
`fromList_app1`	10.4µs	0.97	**0.64**	730µs	2.03	**0.75**	**99.5ms**	3.33	1.19
`union`	4.90µs	0.87	**0.79**	306µs	0.94	**0.89**	**43.7ms**	1.07	1.07
`union_app1`	**4.19µs**	1.94	1.13	**190µs**	2.39	1.98	**23.6ms**	4.34	5.26

The lookup is to ensure that any work delayed by laziness is indeed forced.

- The `fromList*` family benchmarks a naïve *fromList* implementation on *ExprMap* against the tuned *fromList* implementations of the other maps, measuring map creation performance from batches.
- `fold` simply sums up all values that are stored in the map (which stores *Int*s).

Querying. The results show that lookup in *ExprMap* often wins against *Map Expr* and *HashMap Expr*. The margin is small on the completely random *Expr*s of `lookup`, but realistic applications of *ExprMap* often store *Expr*s with some kind of shared structure. The `_lam` and `_app1` variants show that *ExprMap* can win substantially against an ordered map representation: *ExprMap* looks at the shared prefix exactly once one lookup, while *Map* has to traverse the shared prefix of length $O(N)$ on each of its $O(\log N)$ comparisons. As a result, the gap between *ExprMap* and *Map* widens as N increases, confirming an asymptotic difference. The advantage is less pronounced in the `_app2` variant, presumably because *ExprMap* can't share the common prefix here: it turns into an unsharable suffix in the pre-order serialisation, blowing up the trie map representation compared to its sibling `_app1`.

Although *HashMap* loses on most benchmarks compared to *ExprMap* and *Map*, most measurements were consistently at most a factor of two slower than *ExprMap*. We believe that is due to the fact that it is enough to traverse the *Expr* once to compute the hash, thus it is expected to scale similarly as *ExprMap*.

Comparing the `lookup*` measurements of the same map data structure on different size parameters N reveals a roughly quadratic correlation throughout all implementations, give or take a logarithmic factor. That seems plausible given that N linearly affects expression size and map size (and thus, number of lookups). But realistic workloads tend to have much larger map sizes than expression sizes!

Let us see what happens if we vary map size M and expression size E independently for `lookup`. The results in Table 2 show that *ExprMap* scales better than *Map* when we increase M and leave E constant. The difference is even more pronounced than in Table 1, in which $N = M = E$.

The time measurements for *ExprMap* appear to grow almost linearly with M. Considering that the number of lookups also increases M-fold, it seems the cost of a single lookup remained almost constant, despite the fact that we store varying numbers of expressions in the trie map. That is exactly the strength of a trie implementation: Time for the lookup is in $O(E)$, i.e., linear in E but constant in M. The same does not hold for search trees, where lookup time is in $O(P \log M)$. $P \in O(E)$ here and captures the common short circuiting semantics of the lexicographic order on *Expr*. It denotes the size of the longest shared prefix of all expressions.

By contrast, fixing M but increasing E makes *Map* easily catch up on lookup performance with *ExprMap*, ultimately outpacing it. The shared prefix factor P for *Map* remains essentially constant relative to E: larger expressions still are likely to differ very early because they are random. Increasing M will introduce more clashes and is actually more likely to increase P on completely random expressions. As written above, realistic work loads often have shared prefixes like `lookup_app1`, where we already saw that *ExprMap* outperforms *Map*. The fact that *Map* performance depends on P makes it an extremely workload dependent pick, leading to compiler performance that is difficult to predict. *HashMap* shows performance consistent with *ExprMap* but is a bit

Table 2. Varying expression size E and map size M independently on benchmarks `lookup` and `insert_lookup_one`.

	E \ M	10			100			1000			10000		
		TM	OM	HM	TM	OM	HM	TM	OM	HM	TM	OM	HM
lookup	**10**	**3.49µs**	1.07	1.49	**28.7µs**	1.42	1.53	**343µs**	1.86	1.35	**4.28ms**	2.29	1.15
	100	**19.5µs**	1.02	1.75	**213µs**	1.08	1.80	**2.25ms**	1.22	1.72	**27.3ms**	1.44	1.53
	1000	249µs	**0.98**	1.80	2.66ms	**0.99**	1.78	**30.4ms**	1.00	1.68	353ms	**0.99**	1.50
	10000	2.78ms	**0.98**	1.76	31.7ms	**0.99**	1.71	**331ms**	1.00	1.65	**3.31s**	1.01	1.64
lo_a_app1	**10**	**6.23µs**	1.76	1.51	**56.4µs**	3.32	1.54	**620µs**	4.66	1.44	**7.07ms**	5.67	1.31
	100	**46.1µs**	1.93	1.62	**481µs**	3.42	1.64	**4.89ms**	5.04	1.63	**62.7ms**	6.84	1.41
	1000	**527µs**	1.76	1.73	**5.33ms**	3.15	1.78	**64.4ms**	4.82	1.65	**981ms**	5.08	1.09
	10000	**5.48ms**	1.74	1.77	**73.3ms**	3.50	1.69	**850ms**	5.12	1.47	**7.79s**	8.16	1.57
insert_o_l	**10**	**64.3ns**	2.19	1.45	153ns	2.49	**0.80**	**165ns**	3.56	1.06	**105ns**	6.49	1.86
	100	**2.46µs**	1.05	1.52	**2.46µs**	1.15	1.54	**2.48µs**	1.50	1.54	**2.57µs**	2.02	1.49
	1000	**26.1µs**	1.01	1.76	**26.2µs**	1.02	1.75	**26.2µs**	1.04	1.75	**26.5µs**	1.10	1.74
	10000	253µs	**0.99**	1.81	255µs	**0.99**	1.80	255µs	**0.99**	1.80	**255µs**	1.00	1.80
fromList	**10**	3.71µs	**0.67**	0.98	39.3µs	0.99	**0.83**	588µs	1.26	**0.67**	8.54ms	1.39	**0.52**
	100	8.42µs	**0.81**	2.52	129µs	**0.91**	1.98	**1.55ms**	1.13	1.71	**26.8ms**	1.31	1.39
	1000	82.5µs	**0.96**	3.37	916µs	**0.98**	3.21	**13.1ms**	1.01	2.80	**146ms**	1.13	2.65
	10000	731µs	**1.00**	3.86	9.39ms	**0.99**	3.67	108ms	**1.00**	3.38	**1.10s**	1.02	3.30
union	**10**	4.90µs	0.87	**0.79**	37.6µs	1.02	**0.83**	389µs	1.10	**0.96**	**3.67ms**	1.26	1.06
	100	25.4µs	1.02	**1.00**	306µs	0.94	**0.89**	2.85ms	1.03	**0.96**	**37.0ms**	1.31	1.09
	1000	353µs	0.96	**0.92**	3.54ms	0.97	**0.96**	**43.7ms**	1.07	1.07	**427ms**	1.17	1.16
	10000	**3.38ms**	1.05	1.06	**45.1ms**	1.01	1.00	**450ms**	1.03	1.02	**4.42s**	1.04	1.04

slower, as before. There is no subtle scaling factor like P; just plain predictable $O(E)$ like *ExprMap*.

Returning to Table 1, we see that folding over *ExprMap*s is considerably slower than over *Map* or *HashMap*. The complex tree structure is difficult to traverse and involves quite a few indirections. This is in stark contrast to the situation with *Map*, where it's just a textbook in-order traversal over the search tree. Folding over *HashMap* performs similarly to *Map*.

We think that *ExprMap* folding performance dies by a thousand paper cuts: The lazy fold implementation means that we allocate a lot of thunks for intermediate results that we end up forcing anyway in the case of our folding operator (+). That is a price that *Map* and *HashMap* pay, too, but not nearly as much as the implementation of *foldrEM* does. Furthermore, there's the polymorphic recursion in the head case of *em_app* with a different folding function (*foldrTM f*), which allocates on each call and makes it impossible to specialise *foldrEM* for a fixed folding function like (+) with the static argument transformation [Santos 1995]. Hence we tried to single out the issue by ensuring that *Map* and *ExprMap* in fact don't specialise for (+) when running the benchmarks, by means of a `NOINLINE` pragma. Another possible reason might be that the code generated for *foldrEM* is quite a lot larger than the code for *Map*, say, so we are likely measuring caching effects. We are positive there are numerous ways in which the performance of *foldrEM* can be improved, but believe it is unlikely to exceed or just reach the performance of *Map*.

Building. The `insert_lookup_one` benchmark demonstrates that *ExprMap* also wins on insert performance, although the defeat against *Map* for size parameters beyond 1000 is looming. Again, Table 2 decouples map size M and expression size E. The data suggests that in comparison to *Map*, E indeed affects insert performance of *ExprMap* linearly. By contrast, M does not seem to affect insert performance at all.

The small edge that *ExprMap* seems to have over *Map* and *HashMap* doesn't carry over to its naïve *fromList* implementation, though. *Map* wins the `fromList` benchmark, albeit with *ExprMap* as a close second. That is a bit surprising, given that *Map*'s *fromList* quickly falls back to a list fold like *ExprMap* on unsorted inputs, while *HashMap* has a less

naïve implementation: it makes use of transient mutability and performs destructive inserts on the map data structure during the call to *fromList*, knowing that such mutability can't be observed by the caller. Yet, it still performs worse than *ExprMap* or *Map* for larger E, as can be seen in Table 2.

We expected *ExprMap* to take the lead in `fromList_app1`. And indeed it does, outperforming *Map* for larger N which pays for having to compare the shared prefix repeatedly. But *HashMap* is good for another surprise and keeps on outperforming *ExprMap* for small N.

What would a non-naïve version of *fromList* for *ExprMap* look like? Perhaps the process could be sped up considerably by partitioning the input list according to the different fields of *ExprMap* like *em_lam* and then calling the *fromList* implementations of the individual fields in turn. The process would be very similar to discrimination sort [Henglein 2008], which is a generalisation of radix sort to tree-like data and very close to tries. Indeed, the *discrimination*[25] library provides such an optimised $O(N)$ *toMap* implementation for ordered maps.

The `union*` benchmarks don't reveal anything new; *Map* and *HashMap* win for small N, but *ExprMap* wins in the long run, especially when there's a sharable prefix involved.

A.2 Space

We also measured the memory footprint of *ExprMap* compared to *Map* and *HashMap*. The results are shown in Table 3. All four benchmarks simply measure the size on the heap in bytes of a map consisting of M expressions of size E. They only differ in whether or not the expressions have a shared prefix. As before, `space` is built over completely random expressions, while the other three benchmarks build maps with common prefixes, as discussed in Appendix A.1.

In `space`, prefix sharing is highly unlikely for reasons discussed in the last section: Randomness dictates that most expressions diverge quite early in their prefix. As a result, *ExprMap* consumes slightly more space than both *Map* and *ExprMap*, the latter of which wins every single instance. The difference here is ultimately due to the fact that inner nodes in the trie allocate more space than inner nodes in *Map* or *ExprMap*.

However, in `space_app1` and `space_lam`, we can see that *ExprMap* is able to exploit the shared prefixes to great effect: For big M, the memory footprint of `space_app1` approaches that of `space` because the shared prefix is only stored once. In the other dimension along E, memory footprint still increases by similar factors as in `space`. The `space_lam` family does need a bit more bookkeeping for the de Bruijn numbering, so the results aren't quite as close to `space_app1`, but it's still an easy win over *Map* and *HashMap*.

For `space_app2`, *ExprMap* can't share any prefixes because the shared structure turns into a suffix in the pre-order serialisation. As a result, *Map* and *HashMap* allocate less space, all consistent constant factors apart from each other. *HashMap* wins here again.

B Triemaps that unify?

In effect, the *PatVar*s of the patterns stored in our matching triemaps act like unification variables. The unification problems we solve are always particularly simple, because pattern variables only ever match against are *expression* keys in which no pattern variable can occur.

Another frustrating point is that we had to duplicate the *TrieMap* class in **??** because the key types for lookup and insertion no longer match up. If we managed to generalise the lookup key from expressions to patterns, too, we could continue to extend good old *TrieMap*. All this begs the question: *Can we extend our idiomatic triemaps to facilitate unifying lookup?*

At first blush, the generalisation seems simple. We already carefully confined the matching logic to *Matchable*. It should be possible to generalise to

```
class (Eq k, MonadPlus (Unify k)) ⇒ Unifiable k where
  type Unify k :: Type → Type
  unify :: k → k → Unify k ()
class (Unifiable (Key tm), TrieMap tm) ⇒ UTrieMap tm where
  lookupUniUTM :: Key tm → tm v → Unify (Key tm) v
```

But there are problems:

- We would need all unification variables to be globally unique lest we open ourselves to numerous shadowing issues when reporting unifiers.
- Consider the Unimap for

  $$(([a], T\ a\ A), v1) \quad \text{and} \quad (([b], T\ b\ B), v2)$$

  After canonicalisation, we get

  $$((T\ \$_1\ A), ([(a, \$_1)], v1)) \quad \text{and} \quad (T\ \$_1\ B, ([(b, \$_1)], v2))$$

  and both patterns share a prefix in the trie. Suppose now we uni-lookup the pattern $([c, d], Tcd)$. What should we store in our *UniState* when unifying c with $\$_1$? There simply is no unique pattern variable to "decanonicalise" to! In general, it appears we'd get terms in the range of our substitution that mix *PatVar*s and *PatKey*s. Clearly, the vanilla *Expr* datatype doesn't accomodate such an extension and we'd have to complicate its definition with techniques such as Trees that Grow [Najd and Peyton Jones 2017].

So while embodying full-blown unification into the lookup algorithm seems attractive at first, in the end it appears equally complicated to present.

[25] https://hackage.haskell.org/package/discrimination

Table 3. Varying expression size *E* and map size *M* while measuring the memory footprint of the different map implementions on 4 different expression populations. Measurements of *Map* (OM) and *HashMap* (HM) are displayed as relative multiples of the absolute measurements on *ExprMap* (TM). Lower is better. †indicates heap overflow.

	E \ *M*	**10**			**100**			**1000**			**10000**		
		TM	OM	HM	TM	OM	HM	TM	OM	HM	TM	OM	HM
space	**10**	12.5KB	0.79	**0.78**	87.8KB	0.80	**0.80**	766KB	0.89	**0.88**	6.88MB	0.96	**0.95**
	100	55.5KB	0.93	**0.93**	531KB	0.91	**0.91**	4.94MB	0.91	**0.90**	48.6MB	0.92	**0.92**
	1000	477KB	0.99	**0.99**	4.77MB	0.98	**0.98**	46.6MB	0.98	**0.98**	468MB	0.98	**0.98**
	10000	4.10MB	1.00	**1.00**	42.1MB	1.00	**1.00**	421MB	1.00	**1.00**	4.12GB	1.00	**1.00**
space_app1	**10**	**15.3KB**	1.26	1.25	**90.6KB**	1.78	1.78	**769KB**	1.99	1.99	**6.88MB**	2.12	2.11
	100	**83.6KB**	1.74	1.74	**560KB**	2.54	2.54	**4.97MB**	2.74	2.74	**48.6MB**	2.79	2.79
	1000	**758KB**	1.86	1.86	**5.04MB**	2.74	2.74	**46.9MB**	2.92	2.92	**468MB**	2.93	2.93
	10000	**6.85MB**	1.93	1.93	**44.7MB**	2.98	2.98	**423MB**	3.15	3.15	**4.13GB**	3.16	3.16
space_app2	**10**	29.4KB	0.66	**0.65**	240KB	0.67	**0.67**	2.14MB	0.70	**0.70**	20.2MB	0.72	**0.72**
	100	225KB	0.65	**0.65**	2.06MB	0.68	**0.67**	20.2MB	0.68	**0.67**	200MB	0.68	**0.68**
	1000	2.12MB	0.65	**0.65**	20.1MB	0.69	**0.69**	199MB	0.69	**0.69**	1.95GB	0.69	**0.69**
	10000	20.6MB	0.64	**0.64**	196MB	0.68	**0.68**	1.90GB	0.68	**0.68**	†	†	†
space_lam	**10**	18.3KB	0.97	**0.96**	**118KB**	1.24	1.23	**1.00MB**	1.36	1.35	**9.12MB**	1.45	1.45
	100	**73.3KB**	1.77	1.77	**607KB**	2.08	2.08	**5.56MB**	2.18	2.17	**54.6MB**	2.20	2.20
	1000	**610KB**	2.05	2.05	**4.99MB**	2.47	2.47	**47.8MB**	2.55	2.55	**478MB**	2.55	2.55
	10000	**5.33MB**	2.20	2.20	**43.4MB**	2.73	2.73	**423MB**	2.79	2.79	**4.14GB**	2.79	2.79