Triemaps that match

SIMON PEYTON JONES, Microsoft Research, UK
RICHARD A. EISENBERG, Tweag I/O, UK
SEBASTIAN GRAF, Karlsruhe Institute of Technology, Germany

In applications such as compilers and theorem provers, we often want to match a target term against multiple patterns (representing rewrite rules or axioms) simultaneously. 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.

Note: this draft is under review. Please do give us feedback!

ACM Reference Format:
Simon Peyton Jones, Richard A. Eisenberg, and Sebastian Graf. 2021. Triemaps that match. 1, 1 (July 2021), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

$$\textbf{let } x = a + b \textbf{ in} \ldots (\textbf{let } y = a + b \textbf{ in } x + y) \ldots.$$

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

$$\textbf{let } x = a + b \textbf{ in} \ldots (x + x) \ldots.$$

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, get a hit, and replace y by x. All we need is a finite map in 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 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 subsequently rewrite with the instantiated right-hand side (RHS) of the rule. For a compiler developer to accommodate such a feature, we need an extended version of a finite map in which we can insert a collection of rewrite rules, expressed as $(\mathit{pattern}, \mathit{rhs})$ pairs, and then look up an expression in the map, getting a hit if one or more of the patterns *match* the expression. If there is a large number of such $(\mathit{pattern}, \mathit{rhs})$ 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.

Authors' addresses: Simon Peyton Jones, Microsoft Research, Cambridge, UK, simonpj@microsoft.com; Richard A. Eisenberg, Tweag I/O, Cambridge, UK, rae@richarde.dev; Sebastian Graf, Karlsruhe Institute of Technology, Karlsruhe, Germany, sebastian.graf@kit.edu.

In principle it is well known how to build a finite map for a deeply-structured key: use a *trie*. For the matching task, use *discrimination trees*, a variant of tries that are heavily used by the automated reasoning community (Section 7.1). 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 new algebraic data type (Section 3.1). In contrast, most of the literature describes untyped tries for a fixed, generic tree type. In particular:
  - Supported by type class, we can make good use of polymorphism to build triemaps for polymorphic data types, such as lists (Section 3.5).
  - We cover the full range of operations expected for finite maps: not only *insert*ion and *lookup*, but *alter*, *union*, and *fold* (Section 3.1). Other operations like *map* and *filter* are easily implemented, too.
  - 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 3.6).
- 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. The code is short, but surprisingly tricky.
- We present measurements that compare the performance of our triemaps (ignoring their matching capability) with traditional finite-map implementations in Haskell (Section 6).

Our contribution is not so much a clever new idea as an exposition of some old ideas in a new context, perhaps providing some new perspective on those old ideas. We discuss related work in Section 7.

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.* For example, an *Expr* data type might be defined like this:

```
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:

```
type Var = 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.

A finite map keyed by such expressions is extremely useful. The Introduction gave the example of a simple common sub-expression elimination pass. GHC also does many lookups based on *types* rather than *expressions*. For example, when implementing type-class instance lookup, or doing

```
type XT v = Maybe v → Maybe v
data Map k v = . . .    -- a finite map from keys of type k to values of type 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 ⇒ XT v → k → Map k v → Map k v
Map.unionWith :: Ord k ⇒ (v → v → v) → Map k v → Map k v → Map k v
Map.size       :: Map k v → Int
Map.foldr      :: (v → r → r) → r → Map k v → r
data Bag v    -- An unordered collection of values v
Bag.empty      :: Bag v
Bag.single     :: v → Bag v
Bag.union      :: Bag v → Bag v → Bag v
Bag.map        :: (v1 → v2) → Bag v1 → Bag v2
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
```

Fig. 1. API for library functions

type-family reduction, GHC needs a map whose key is a type. Both types and expressions are simply trees, and so are particular instances of the general task.

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.

2.1 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 $map\ (double \circ square)\ nums$; 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" $map\ (\lambda x \rightarrow x) = \lambda y \rightarrow y$ #-}

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.

2.2 The interface of of a finite map

What API might such a map have? Building on the design of widely used functions in Haskell (see Fig. 1), we seek these basic operations:

$$\begin{aligned}
&emptyEM &&:: ExprMap\ v\\
&lookupEM &&:: Expr \rightarrow ExprMap\ v \rightarrow Maybe\ v\\
&alterEM &&:: Expr \rightarrow XT\ v \rightarrow ExprMap\ v \rightarrow ExprMap\ v
\end{aligned}$$

The functions $emptyEM$ and $lookupEM$ should be self-explanatory. The function $alterTM$ is a standard generalisation of $insertEM$: instead of providing just a new element to be inserted, the caller provides a *transformation* $XT\ v$, an abbreviation for $Maybe\ v \rightarrow Maybe\ v$ (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 $Maybe$). These fundamental operations on a finite map must obey the following properties:

$$\begin{aligned}
&\forall e, && lookup\ e\ empty && \equiv Nothing\\
&\forall e\ m\ xt, && lookup\ e\ (alter\ e\ xt\ m) && \equiv xt\ (lookup\ e\ m)\\
&\forall e_1\ e_2\ m\ xt, e_1 \neq e_2 \Rightarrow && lookup\ e_1\ (alter\ e_2\ xt\ m) && \equiv lookup\ e_1\ m
\end{aligned}$$

We can easily define $insertEM$ and $deleteEM$ from $alterEM$:

$$\begin{aligned}
&insertEM :: Expr \rightarrow v \rightarrow ExprMap\ v \rightarrow ExprMap\ v\\
&insertEM\ e\ v = alterEM\ e\ (\backslash\_ \rightarrow Just\ v)
\end{aligned}$$

$$\begin{aligned}
&deleteEM :: Expr \rightarrow ExprMap\ v \rightarrow ExprMap\ v\\
&deleteEM\ e = alterEM\ e\ (\backslash\_ \rightarrow Nothing)
\end{aligned}$$

You might wonder whether, for the purposes of this paper, we could just define $insert$, leaving $alter$ for the Supplemental[1], but as we will see in Section 3.2, our approach using tries fundamentally requires the generality of $alter$.

We would also like to support other standard operations on finite maps, including

- An efficient union operation to combine two finite maps into one:

  $unionEM :: ExprMap\ v \rightarrow ExprMap\ v \rightarrow ExprMap\ v$

- A map operation to apply a function to the range of the finite map:

  $mapEM :: (a \rightarrow b) \rightarrow ExprMap\ a \rightarrow ExprMap\ b$

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

- A fold operation to combine together the elements of the range:

```
foldEM :: (a → b → b) → ExprMap a → b → b
```

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. While this double-check is not so terrible, we will see that the naive approach described here does not extend well to support the extra features we require in our finite maps.

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 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*[2]. Let us consider a simplified form of expression:

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

We leave lambdas out 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 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, existing finite map, such as the *containers*[3] library keyed by *Var*, with values *v*. One way to understand this slightly odd data type is to study its lookup function:

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

[2]https://en.wikipedia.org/wiki/Trie
[3]https://hackage.haskell.org/package/containers

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:

```
lookupExpr (Var x)       = em_var >.> Map.lookup x
lookupExpr (App e₁ e₂) = em_app >.> lookupExpr e₁ >=> lookupExpr 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, 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. (We typically use the term "target" for the key we are looking up in the finite map.)

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

3.2 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:

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

It is interesting to note that *emptyExpr* is an infinite, recursive structure: the *em_app* field refers back to *emptyExpr*. We will change this definition in Section 3.4, but it works perfectly well for now.

Next, we need to *alter* a triemap:

```
alterExpr :: Expr → XT v → ExprMap v → ExprMap v
alterExpr e xt m@(EM {em_var = m_var, em_app = m_app})
   = case e of
       Var x     → m {em_var = Map.alter xt x m_var}
       App e₁ e₂ → m {em_app = alterExpr e₁ (liftXT (alterExpr e₂ xt)) m_app}
liftXT :: (ExprMap v → ExprMap v) → XT (ExprMap v)
liftXT f Nothing  = Just (f emptyExpr)
liftXT 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 Haskell the notation "*m* {*fld* = *e*}" means the result of updating the *fld* field of record *m* with new value *e*. In the *App* case we look up e_1 in *m_app*; we should find a *ExprMap* there, which we want to alter with *xt*. We can do that with a recursive call to *alterExpr*, using *liftXT* for impedance-matching.

The *App* case shows why we need the generality of *alter*. Suppose we attempted to define an apparently-simpler *insert* operations. 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 *alterExpr* 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 Supplemental.

3.3 Unions of maps

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

```
unionExpr :: 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*:

```
unionWithExpr :: (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:

```
unionWithExpr f (EM {em_var = m1_var, em_app = m1_app})
                (EM {em_var = m2_var, em_app = m2_app})
  = EM {em_var = Map.unionWith f m1_var m2_var
       , em_app = unionWithExpr (unionWithExpr f) m1_app m2_app}
```

It could hardly be simpler.

3.4 Folds and the empty map

This strange, infinite definition of *emptyExpr* given in Section 3.2 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:

```
sizeExpr :: ExprMap v → Int
sizeExpr (EM {em_var = m_var, em_app = m_app}) = Map.size m_var+???
```

We seem stuck because the size of the *m_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:

```
foldrExpr :: (v → r → r) → r → ExprMap v → r
foldrExpr k z (EM {em_var = m_var, em_app = m_app})
  = Map.foldr k z1 m_var
  where
    z1 = foldrExpr kapp z m_app
    kapp m₁ r = foldrExpr k r m₁
```

In the binding for *z1* we fold over *m_app* :: *ExprMap* (*ExprMap* *v*). The function *kapp* is combines the map we find with the accumulator, by again folding over the map with *foldrExpr*.

But alas, *foldrExpr* will never terminate! It always invokes itself immediately (in *z1*) on *m_app*; but that invocation will again recursively invoke *foldrExpr*; 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 3.5):

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

emptyExpr :: ExprMap v
emptyExpr = EmptyEM

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

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

```
sizeExpr :: ExprMap v → Int
sizeExpr = foldrExpr (λ_ n → n + 1) 0

elemsExpr :: ExprMap v → [v]
elemsExpr = foldrExpr (:) [ ]
```

3.5 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 (TrieKey tm) ⇒ TrieMap tm where
  type TrieKey tm :: Type
  emptyTM  :: tm a
  lookupTM :: TrieKey tm → tm a → Maybe a
  alterTM  :: TrieKey tm → XT a → tm a → tm a
  foldTM   :: (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*, *lookupTM* etc. The class has an *associated type* [Chakravarty et al. 2005], *TrieKey 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 TrieKey ExprMap = Expr
  emptyTM  = emptyExpr
  lookupTM = lookupExpr
```

```
    alterTM    = alterExpr
    ...
```

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

```
insertTM :: TrieMap tm ⇒ TrieKey tm → v → tm v → tm v
insertTM k v = alterTM k (\_ → Just v)

deleteEM :: TrieMap tm ⇒ TrieKey tm → tm v → tm v
deleteEM k = alterEM 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 built 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

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

But rather than 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]:

```
lookupList :: TrieMap tm ⇒ [TrieKey tm] → ListMap tm v → Maybe v
lookupList []       = lm_nil
lookupList (k : ks) = lm_cons >.> lookupTM k >=> lookupList ks

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

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

The code for *alterList* and *foldList* is routine. Notice that all of these functions are polymorphic in *tm*, the triemap for the list elements. So *ListMap* is a *triemap-transformer*; and if *tm* is a *TrieMap* then so is *ListMap tm*:

```
instance TrieMap tm ⇒ TrieMap (ListMap tm) where
    type TrieKey (ListMap tm) = [TrieKey tm]
    emptyTM = emptyList
    lookupTM = lookupList
    ...
```

3.6 Singleton maps, and empty maps revisited

Suppose we start with an empty map, and insert a value with a key (an *Expr*) that is large, say

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

Looking at the code for *alterExpr* in Section 3.2, 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 :: Map Var v, em_app :: ExprMap (ExprMap v) }
```

But we will have to tiresomely repeat these extra data constructors, *EmptyX* and *SingleX* for each new data type *X* for which we want a triemap. For example we would have to add *EmptyList* and *SingleList* to the *ListMap* data type of Section 3.5. It is better instead to abstract over the enclosed triemap, like this:

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

The code for lookup practically writes itself:

```
lookupSEMap :: TrieMap tm ⇒ TrieKey tm → SEMap tm v → Maybe v
lookupSEMap _ EmptySEM                           = Nothing
lookupSEMap tk (SingleSEM pk v) | tk == pk       = Just v
                                | otherwise      = Nothing
lookupSEMap tk (MultiSEM tm)                     = lookupTM tk tm
```

Notice that in the *SingleSEM* case we need equality on the key type *TrieKey tm*, to tell if the key being looked up, *tk* is the same as the key in the *SingleEM*, namely *pk*. That is why we made *Eq* (*TrieKey tm*) a superclass of *TrieMap tm* in the **class** declaration in Section 3.5.

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

```
alterSEM :: TrieMap tm ⇒ TrieKey tm → XT v → SEMap tm v → SEMap tm v
alterSEM k xt EmptySEM = case xt Nothing of Nothing → EmptySEM
                                            Just v   → SingleSEM k v
alterSEM k₁ xt (SingleSEM k₂ v2)
  | k₁ == k₂  = case xt (Just v2) of
                  Nothing → EmptySEM
                  Just v'  → SingleSEM k₂ v'
  | otherwise = case xt Nothing of
                  Nothing → SingleSEM k₂ v2
                  Just v1  → MultiSEM (insertTM k₁ v1 (insertTM k₂ v2 emptyTM))
alterSEM k xt (MultiSEM tm) = MultiSEM (alterTM k xt tm)
```

Now, of course, we can make *SEMap* itself an instance of *TrieMap*:

```
instance TrieMap tm ⇒ TrieMap (SEMap tm) where
  type TrieKey (SEMap tm) = TrieKey tm
  emptyTM = EmptySEM
  lookupTM = lookupSEM
  alterTM  = alterSEM
  ...
```

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*. But you might wonder whether we can shrink a *MultiSEM* back to a *SingleSEM* when it has only one remaining element? Yes, of course 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*:

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

type ListMap        = SEMap ListMap'
data ListMap' tm v = LM { lm_nil :: Maybe v, 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.

The singleton-map optimisation makes a big difference in practice.

3.7 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 (Section 2). That is the challenge we address next. Here is our data type, *ExprL*, where the "L" connotes the new *Lam* constructor:

```
data ExprL = AppL ExprL ExprL | Lam Var ExprL | VarL 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' = AppL ExprL ExprL | Lam ExprL | FreeVar Var | BoundVar Int
```

Notice (a) the *Lam* node no longer has a binder and (b) there are two sorts of *VarL* nodes, one for free variables and one for bound variables. 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

```
data ExprLMap v = ELM { elm_app :: ExprLMap (ExprLMap v)
                      , elm_lam :: ExprLMap v
                      , elm_fv  :: Map Var v              -- Free variables
                      , elm_bv  :: Map BoundVarKey v }    -- Lambda-bound variables
```

```
lookupExprL :: ExprL → ExprLMap v → Maybe v
lookupExprL e = lkExprL (DB emptyBVM e)

data DBExprL = DB {edb_bvm :: BoundVarMap, edb_expr :: ExprL}

lkExprL :: DBExprL → ExprLMap v → Maybe v
lkExprL (DB bvm (AppL e1 e2)) = elm_app >.> lkExprL (DB bvm e1) >=> lkExprL (DB bvm e2)
lkExprL (DB bvm (Lam v e)) = elm_lam >.> lkExprL (DB (extendBVM v bvm) e)
lkExprL (DB bvm (VarL v)) = case lookupBVM v bvm of
                                Nothing → elm_fv >.> Map.lookup v     -- Free
                                Just bv → elm_bv  >.> Map.lookup bv   -- Lambda-bound

data BoundVarMap = BVM {bvm_next :: BoundVarKey, bvm_map :: Map Var BoundVarKey}
type BoundVarKey = Int

emptyBVM :: BoundVarMap
emptyBVM = BVM {bvm_next = 1, bvm_map = Map.empty}

extendBVM :: Var → BoundVarMap → BoundVarMap
extendBVM v (BVM {bvm_next = n, bvm_map = bvm})
  = BVM {bvm_next = n + 1, bvm_map = Map.insert v n bvm}

lookupBVM :: Var → BoundVarMap → Maybe BoundVarKey
lookupBVM v (BVM {bvm_map = bvm}) = Map.lookup v bvm
```

We maintain a *BoundVarMap* that maps each lambda-bound variable to its de-Bruijn level[4] [de Bruijn 1972], of type *BoundVarKey*. The key we look up — the first argument of *lkExprL* — becomes a *DBExprL*, which is a pair of a *BoundVarMap* and an *ExprL*. At a *Lam* node we extend the *BoundVarMap*. At a *Var* node we look up the variable in the *BoundVarMap* to decide whether it is lambda-bound (within the key) or free, and behave appropriately. The code for *alter* and *fold* holds no new surprises. The construction of Section 3.5, to handle empty and singleton maps, applies without difficulty to this generalised map.

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.

5 TRIES THAT MATCH

A key advantage of tries over hash-maps and balanced trees is that they can naturally extend to support *matching* (Section 2.1). In this section we explain how.

5.1 What "matching" means

First, we have to ask what the API should be. Our overall goal is to build a *matching trie* into which we can:

- *Insert* (pattern, value) pairs
- *Look up* a target expression, and return all the values whose pattern *matches* that expression.

Semantically, then, 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 $f\ a\ (g\ b\ c)$.

[4]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*.

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, suppose we wanted to encode the rewrite rule

```
{-# RULES "foo" ∀x. f x x = f_2 x #-}
```

Here the pattern $([x], f\ x\ x)$ has a repeated variable x, and 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.

5.2 The API of a matching trie

Here are the signatures of the lookup and insertion[5] functions for our new matching triemap, *MExprMap*:

```
type ExprPat = ([PatVar], Expr)
type PatVar = Var
type Match v = ([(PatVar, Expr)], v)
type MExprMap v = ...    -- in Section 5.5
insertMExpr :: ExprPat → v → MExprMap v → MExprMap v
lookupMExpr :: Expr → MExprMap v → Bag (Match v)
```

A *MExprMap* is a trie, keyed by *ExprPat* *patterns*. A pattern variable, of type *PatVar* is just a *Var*; we use the type synonym just for documentation purposes. When inserting into a *MExprMap* we supply a pattern expression paired with the [*PatVar*] over which the pattern is quantified. When looking up in the map we return a *bag* of results (because more than one pattern might match). Each item in this bag is a *Match* that includes the (*PatVar*, *Expr*) pairs obtained by matching the pattern, plus the value in the map (which presumably mentions those pattern variables).

A *Bag* is a standard un-ordered collection of values, with a union operation; see Fig. 1. We need to be able to return a bag because there may be multiple matches. Even if we are returning the most-specific matches, there may be multiple incomparable ones.

5.3 Canonical patterns and pattern keys

In Section 4 we saw how we could use de-Bruijn levels to make two lambda expressions that differ only superficially (in the name of their bound variable) look the same. Clearly, we want to do the same for pattern variables. After all, consider these two patterns:

$$([a, b], f\ a\ b\ True) \qquad \text{and} \qquad ([p, q], f\ q\ p\ False)$$

The two pattern expressions share a common prefix, but differ both in the *names* of the pattern variable and in their *order*. We might hope to suppress the accidental difference of names by using numbers instead – we will use the term *pattern keys* for these numbers. But from the set of pattern variables alone, we cannot know *a priori* which key to assign to which variable.

Our solution is to number the pattern variables *in order of their first occurrence in a left-to-right scan of the expression*[6]. As in Section 4 we will imagine that we canonicalise the pattern, although in reality we will do so on-the-fly, without ever constructing the canonicalised pattern. Be that as it

[5]We begin with *insert* because it is simpler than *alter*

[6]As we shall see, this is very convenient in implementation terms.

may, the canonicalised patterns become:

$$f\ \$_1\ \$_2\ True \qquad \text{and} \qquad f\ \$_1\ \$_2\ False$$

By numbering the variables left-to-right, we ensure that they "line up". In fact, since the pattern variables are numbered left-to-right we don't even need the subscripts (just as we don't need a subscript on the lambda in de-Bruijn notation), so the canonicalised patterns become

$$f\ \$\ \$\ True \qquad \text{and} \qquad f\ \$\ \$\ False$$

What if the variable occurs more than once? For example, suppose we are matching the pattern $([x],\ f\,x\,x\,x)$ against the target expression $(f\ e_1\ e_2\ e_3)$. At the first occurrence of the pattern variable x we succeed in matching, binding x to e_1; but at the second occurrence we must note that x has already been bound, and instead check that e_1 is equal to e_2; and similarly at the third occurrence. These are very different actions, so it is helpful to distinguish the first occurrence from subsequent ones when canonicalising. So our pattern $([x],\ f\,x\,x\,x)$ might be canonicalised to $(f\ \$\ \%_1\ \%_1)$, where the first (or binding) occurrence is denoted \$ and subsequent (bound) occurrences of pattern variable i are denoted $\%_i$.

For pattern-variable occurrences we really do need the subscript! Consider the patterns

$$([x, y], f\ x\ y\ y\ x) \qquad \text{and} \qquad ([p, q], f\ q\ p\ q\ p)$$

which differ not only in the names of their pattern variables, but also in the order in which they occur in the pattern. They canonicalise to

$$(f\ \$\ \$\ \%_2\ \%_1) \qquad \textit{and} \qquad (f\ \$\ \$\ \%_1\ \%_2)$$

respectively. The subscripts are essential to keep these two patterns distinct.

5.4 Undoing the pattern keys

The trouble with canonicalising our patterns (to share the structure of the patterns) is that matching will produce a substitution mapping pattern *keys* to expressions, rather that mapping pattern *variables* to expressions. For example, suppose we start with the pattern $([x, y], f\ x\ y\ y\ x)$ from the end of the last section. Its canonical form is $(f\ \$\ \$\ \%_2\ \%_1)$. If we match that against a target $(f\ e_1\ e_2\ e_2\ e_1)$ we will produce a substitution $[\%_1 \mapsto e_1, \%_2 \mapsto e_2]$. But what we *want* is a *Match* (Section 5.2), that gives a list of (pattern-variable, expression) pairs $[(x, e_1), (y, e_2)]$.

Somehow we must accumulate a *pattern-key map* that, for each individual entry in triemap, maps its pattern keys back to the corresponding pattern variables for that entry. The pattern-key map is just a list of (pattern-variable, pattern-key) pairs. For our example the pattern key map would be $[(x, \$_1), (y, \$_2)]$. We can store the pattern key map paired with the value, in the triemap itself, so that once we find a successful match we can use the pattern key map and the pattern-key substitution to recover the pattern-variable substitution that we want.

To summarise, suppose we want to build a matching trie for the following (pattern, value) pairs:

$$(([x, y],\ f\ y\ (g\ y\ x)),\ v_1) \qquad \text{and} \qquad (([a],\ f\ a\ True),\ v_2)$$

Then we will build a trie with the following entries (key-value pairs):

$$((f\ \$\ (g\ \%_1\ \$)),\ ([(x, \$_2), (y, \$_1)], v_1)) \qquad \text{and} \qquad ((f\ \$\ True),\ ([(a, \$_1)],\ v_2))$$

5.5 Implementation: lookup

We are finally ready to give an implementation of matching tries. We begin with *Expr* (defined in Section 3) as our key type; that is we will not deal with lambdas and lambda-bound variables for now. Section 4 will apply with no difficulty, but we can add that back in after we have dealt with matching. With these thoughts in mind, our matching trie has this definition:

```
type PatKeys = [(PatVar, PatKey)]
type MExprLMap v = MExprLMapX (PatKeys, v)
data MExprLMapX v
  = MM {mm_app  :: MExprLMap (MExprLMap v)
       , mm_fvar :: Map Var v
       , mm_pvar :: Maybe v      -- First occurrence of a pattern var
       , mm_xvar :: PatOccs v    -- Subsequent occurrence of a pattern var
       }
  | EmptyMM
type PatOccs v = Map PatKey v
```

The client-visible *MExprLMap* with values of type *v* is a matching trie *MExprLMapX* with values of type (*PatKeys*, *v*), as described in Section 5.4. The trie *MExprLMapX* has four fields, one for each case in the pattern. The first two fields deal with free variables and applications, just as before. The third deals with the *binding* occurrence of a pattern variable \$, and the fourth with a *bound* occurrence of a pattern variable $\%_i$.

The core lookup function looks like this:

```
lkMExpr :: ∀v. Expr → (PatSubst, MExprLMapX v) → Bag (PatSubst, v)
```

As well as the target expression *Expr* and the trie, the lookup function also takes a *PatSubst* that gives the bindings for pattern variable bound so far. It returns a bag of results, since more than one entry in the trie may match, each paired with the *PatSubst* that binds the pattern variables. A *PatSubst* carries not only the current substitution, but also (like a *BoundVarMap*, Section 4) the next free pattern key:

```
data PatSubst = PS {ps_next :: PatKey, ps_subst :: Map PatKey Expr}
type PatKey = Int
emptyPatSubst :: PatSubst
emptyPatSubst = PS {ps_next = 0, ps_subst = Map.empty}
extendPatSubst :: Expr → PatSubst → PatSubst
extendPatSubst e (PS {ps_next = next, ps_subst = subst})
  = PS {ps_next = next + 1, ps_subst = Map.insert next e subst}
lookupPatSubst :: PatKey → PatSubst → Expr
lookupPatSubst pat_key (PS {ps_subst = subst})
  = case Map.lookup pat_key subst of
      Just expr → expr
      Nothing → error "Unbound key"
```

Given *lkMExpr* we can write *lookupMExpr*, the externally-callable lookup function:

```
lookupMExpr :: Expr → MExprLMap v → Bag (Match v)
lookupMExpr e m = fmap rejig (lkMExpr e (emptyPatSubst, m))
   where
      rejig :: (PatSubst, (PatKeys, v)) → Match v
      rejig (ps, (pkmk, v)) = (map (lookupPatKey ps) pkmk, v)
lookupPatKey :: PatSubst → (PatVar, PatKey) → (PatVar, Expr)
lookupPatKey subst (pat_var, pat_key) = (pat_var, lookupPatSubst pat_key subst)
```

Here *lookupMExpr* is just an impedance-matching shim around a call to *lkMExpr* that does all the work. Notice that the input changed. The latter returns a bag of (*PatSubst*, (*PatKeys*, *v*)) values, which the function *rejig* converts into the the *Match v* results that we want. The "unbound key" failure case in *lookupPatSubst* means that *PatKeys* in a looked-up value asks for a key that is not bound in the pattern. The insertion function will ensure that this never occurs.

Now we can return to the recursive function that does all the work: *lkMExpr*:

```
lkMExpr :: ∀v. Expr → (PatSubst, MExprLMapX v) → Bag (PatSubst, v)
lkMExpr e (psubst, mt)
  = pat_var_bndr `Bag.union` pat_var_occs `Bag.union` look_at_e
  where
    pat_var_bndr :: Bag (PatSubst, v)
    pat_var_bndr = case mm_pvar mt of
      Just v → Bag.single (extendPatSubst e psubst, v)
      Nothing → Bag.empty

    pat_var_occs :: Bag (PatSubst, v)
    pat_var_occs = Bag.fromList [ (psubst, v)
                                | (pat_var, v) ← Map.toList (mm_xvar mt)
                                , e == lookupPatSubst pat_var psubst]

    look_at_e :: Bag (PatSubst, v)
    look_at_e = case e of
      Var x → case Map.lookup x (mm_fvar mt) of
                Just v → Bag.single (psubst, v)
                Nothing → Bag.empty
      App e₁ e₂ → Bag.concatMap (lkT (D dbe t2)) $
        lkT (D dbe t1) (tsubst, mem_fun mt)
```

The bag of results is the union of three possibilities, as follows. (Keep in mind that a *MExprLMap* represents *many* patterns simultaneously.)

- *pat_var_bndr*: we consult the *mm_pvar*, if it contains *Just v* then at least one of the patterns in this trie has a pattern binder \$ at this spot. In that case we can simply bind the next free pattern variable (*ps_next*) to *e*, and return a singleton bag.
- *pat_var_occs*: any of the bound pattern variables might have an occurrence $\%_i$ at this spot, and a list of such bindings is held in *pat_var_occs*. For each, we must do an equality check between the target *e* and the expression bound to that pattern variable (found via *lookupPatSubst*). We return a bag of all values for which the equality check succeeds.
- *look_at_e* corresponds exactly to the cases we saw before in Section 3. The only subtlety is that we are are returning a *bag* of results, but happily the Kleisli composition operator (>=>) (Fig. 1) works for any monad, including bags.

5.6 Altering a matching trie

How did the entries in our map get their *PatKeys*? That is, of course, the business of *insert*, or more generally *alter*. The key, recursive function must carry inwards a mapping from pattern variables to pattern keys; we can simply re-use *BoundVarMap* from Section 4 for this purpose. The exact signature for the function takes a bit of puzzling out, and is worth comparing with its predecessor in Section 3.2:

```
type PatKeyMap = BoundVarMap    -- We re-use BoundVarMap
xtMExpr :: Set PatVar → Expr → (PatKeyMap → XT a)
   → PatKeyMap → MExprLMapX v → MExprLMapX v
```

It is unsurprising that the function is given the set of pattern variables, so that it can distinguish pattern variables from free variables. It also takes a *PatKeyMap*, the current binding of already-encountered pattern variables to their pattern keys; when it completes the lookup it passes that completed binding map to the "alter" function.

Given this workhorse, we can build the client-visible *insert* function[7]:

```
insertMExpr :: ∀v. [Var]   -- Pattern variables
            → Expr        -- Pattern
            → v → MExprLMap v → MExprLMap v
insertMExpr pat_vs e v mm
   = xtMExpr (Set.fromList pat_vs) e xt emptyBVM mm
  where
    xt :: PatKeyMap → XT (PatKeys, v)
    xt pkm _ = Just (map inst_key pat_vs, v)
        -- The "_" means just overwrite previous value
      where
        inst_key :: PatVar → (PatVar, PatKey)
        inst_key x = case lookupBVM x pkm of
          Nothing → error ("Unbound pattern variable " ++ x)
          Just pk → (x, pk)
```

This is the code that builds the *PatKeys* in the range of the map. It does so using the *PatKeyMap* accumulated by *xtMExpr* and finally passed to the local function *xt*.

Now we can define the workhorse, *xtMExpr*:

```
xtMExpr pvs e xt pkm mm
   = case e of
    App e1 e2 → mm {mm_app = xtMExpr pvs e1 (liftXTS (xtMExpr pvs e2 xt))
                                           pkm (mm_app mm)}
    Var x | Just xv ← lookupBVM x pkm
      →    -- Second or subsequent occurrence of a pattern variable
        mm {mm_xvar = Map.alter (xt pkm) xv (mm_xvar mm)}
      | x `Set.member` pvs
      →    -- First occurrence of a pattern variable
        mm {mm_pvar = xt (extendBVM x pkm) (mm_pvar mm)}
      | otherwise
      →    -- A free variable
        mm {mm_fvar = Map.alter (xt pkm) x (mm_fvar mm)}
```

[7] *alter* is not much harder.

Table 1. Benchmarks of different operations over our trie map *ExprLMap* (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_all`	**1.90̃µs**	1.00̃	1.43̃	**130µs**	1.09	1.72	**19.3̃ms**	1.06̃	1.64̃
`lookup_all_app1`	**3.63̃µs**	1.55̃	1.40̃	**293µs**	2.70̃	1.51̃	**42.9ms**	4.96	1.53̃
`lookup_all_app2`	**3.56̃µs**	1.16̃	1.41̃	**387̃µs**	1.42̃	1.26̃	84.8̃ms	3.30̃	**0.94̃**
`lookup_all_lam`	**3.61µs**	2.72̃	1.96̃	**322µs**	5.57	2.17̃	**49.9ms**	7.82̃	1.95̃
`lookup_one`	177ns	**0.74**	1.38̃	**811̃ns**	1.0̃7̃	1.64	13.5µs	**0.98**	1.64
`fold`	357̃ns	0.37	**0.30**	4.03̃µs	0.34	**0.33**	68.4̃µs	**0.30**	0.36
`insert_lookup_one`	**42.0̃ns**	1.98̃	1.86̃	**1.39µs**	1.24	2.07	**15.6µs**	1.03̃	2.26
`fromList`	2.69̃µs	**0.65**	0.84	104µs	**0.94̃**	1.65	11.8ms	**1.00**	2.26̃
`fromList_app1`	6.44̃µs	0.86̃	**0.65**	457̃µs	1.71̃	**0.77̃**	**63̃.9̃ms**	3.86	1.40̃
`union`	3.40µs	**0.70**	0.72	211µs	0.92	**0.89**	**32.0̃ms**	1.09	1.06̃
`union_app1`	**3.13̃µs**	1.77̃	1.25̃	**153µs**	2.21	1.96	**22.0ms**	3.46̃	3.43̃

5.7 Further developments: most specific match, and unification

It is sometimes desirable to be able to look up the *most specific match* in the matching triemap. For example, suppose the matching trie contains the following two (pattern,value) pairs:

$$\{([a],\ f\ a),\quad ([p,q],\ f\ (p+q))\}$$

and suppose we look up $(f\ (2+x))$ in the trie. The first entry matches, but the second also matches (with $S = [p \mapsto 2, q \mapsto x]$), and *the second pattern is a substitution instance of the first.* In some applications we may want to return just the second match. We call this *most-specific matching*.

The implementation we have shown returns *all* matches, leaving it to a post-pass to pick only the most-specific ones. It seems plausible that some modification to the lookup algorithm might suffice to identify the most-specific matches, but it turns out to be hard to achieve this, because each case only has a local view of the overall match.

Sometimes one wants to find all patterns that *unify* with the target, assuming we have some notion of "unifiable variable" in the target. Embodying full-blown unification into the lookup algorithm seems hard, but it is relatively easy to return a set of *candidates* that then be post-processed with a full unifier to see if the candidate does indeed unify with the target.

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!

6.1 Runtime

We measured the runtime performance of the (non-matching) *ExprLMap* data structure[8] on a selection of workloads, conducted using the *criterion*[9] benchmarking library[10]. 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 *ExprLMap* forms the baseline. Asymptotics are given with respect to map size n and key expression size k:

- *ExprLMap* (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*[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 *ExprLMap*, 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 an 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* `"$"`) around each expression
- `_app1` wraps N layers of (*Lit* `"$"` ‘*App*‘) around each expression[13]
- `_app2` wraps N layers of (‘*App*‘ *Lit* `"$"`) around each expression

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

- The `lookup_all*` 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. The lookup is to ensure that any work delayed by laziness is indeed forced.
- The `fromList*` family benchmarks a naïve *fromList* implementation on *ExprLMap* 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).

[8]Called just *ExprMap* in the supplemental, also supporting an additional *Lit Var* constructor in *Expr*.
[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 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.
[11]https://hackage.haskell.org/package/containers
[12]https://hackage.haskell.org/package/unordered-containers
[13]Recall that *Lit* is only present in the Supplemental and works like a constant occurrence of a free variable.

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

	E \ M	**10**			**100**			**1000**			**10000**		
		TM	OM	HM	TM	OM	HM	TM	OM	HM	TM	OM	HM
`lookup_all`	**10**	**1.90̃µs**	1.00̃	1.43̃	**17.0µs**	1.43	1.43	**224µs**	1.77̃	1.26	**2.94̃ms**	2.17̃	1.05̃
	100	10.9µs	**1.00**	1.66̃	**130µs**	1.09	1.72	**1.38ms**	1.25	1.63	**18.3̃ms**	1.48̃	1.38̃
	1000	151µs	**0.98**	1.76̃	1.60ms	**1.00**	1.76	**19.3̃ms**	1.06̃	1.64̃	**214ms**	1.07	1.51
	10000	1.66̃ms	**0.98**	1.76̃	19.7̃ms	**1.00̃**	1.69̃	**211ms**	1.01	1.58	**2.16s**	1.00	1.54
`lo_a_app1`	**10**	**3.63̃µs**	1.55̃	1.40̃	**34.1̃µs**	2.8̃2̃	1.41̃	**389̃µs**	3.84̃	1.32̃	**4.66̃ms**	4.60̃	1.18̃
	100	**26.7µs**	1.61̃	1.47̃	**293µs**	2.70̃	1.51̃	**2.98̃ms**	4.17	1.53̃	**42.6̃ms**	6.55̃	1.26̃
	1000	**322̃µs**	1.46̃	1.63̃	**3.18ms**	2.91̃	1.88̃	**42.9ms**	4.96	1.53̃	**714ms**	4.98	1.06
	10000	**3.50ms**	1.39̃	1.67̃	**57.4̃ms**	2.79̃	1.35̃	**487ms**	7.19	2.09	**5.69s**	9.04	1.73
`insert_o_1`	**10**	**42.0̃ns**	1.98̃	1.86̃	102ns	1.78̃	**0.80̃**	109ns	2.58	**0.98**	**69.7̃ns**	4.74	1.99
	100	**1.39µs**	1.05	2.11	**1.39µs**	1.24	2.07	**1.47µs**	1.65	2.02	**1.52µs**	2.35̃	1.98̃
	1000	15.5µs	**0.97**	2.27	15.6µs	**0.99**	2.27̃	**15.6µs**	1.03̃	2.26	**15.7µs**	1.07	2.26
	10000	152µs	**0.99**	2.52̃	152µs	**0.99**	2.52̃	153µs	**0.99**	2.50̃	**153µs**	1.00	2.50̃
`fromList`	**10**	2.69̃µs	**0.65**	0.84	28.5µs	0.97	**0.73**	443̃µs	1.18̃	**0.62**	6.72̃ms	1.25̃	**0.47̃**
	100	6.64̃µs	**0.85̃**	1.95	104µs	**0.94̃**	1.65	**1.24ms**	1.11	1.44	**22.7̃ms**	1.24	1.21̃
	1000	70.1̃µs	**0.98̃**	2.66	789̃µs	**0.99̃**	2.51	11.8ms	**1.00**	2.26̃	**132ms**	1.11	2.15
	10000	648̃µs	**1.00̃**	2.91	8.51̃ms	**0.9̃9̃**	2.89̃	98.6̃ms	**1.00̃**	2.66̃	**1.00s**	1.02	2.61
`union`	**10**	3.40µs	**0.70**	0.72	25.8̃µs	0.89	**0.80**	279µs	1.03	**0.95**	2.88̃ms	1.06̃	**0.98**
	100	16.5µs	0.97	**0.96**	211µs	0.92	**0.89**	2.03ms	1.00	**0.92**	**30.8ms**	1.19̃	1.06̃
	1000	238µs	0.96	**0.93**	2.38ms	0.98	**0.96**	**32.0̃ms**	1.09	1.06̃	**327ms**	1.13	1.08
	10000	**2.24ms**	1.05	1.06	**32.4̃ms**	1.02̃	1.00̃	**323ms**	1.04	1.04	**3.16s**	1.07	1.05

Querying. The results show that lookup in *ExprLMap* often wins against *Map Expr* and *HashMap Expr*. The margin is small on the completely random *Expr*s of `lookup_all`, but realistic applications of *ExprLMap* often store *Expr*s with some kind of shared structure. The `_lam` and `_app1` variants show that *ExprLMap* can win substantially against an ordered map representation: *ExprLMap* 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 *ExprLMap* and *Map* widens as N increases, confirming an asymptotic difference. The advantage is less pronounced in the `_app2` variant, presumably because *ExprLMap* 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 *ExprLMap* and *Map*, most measurements were consistently at most a factor of two slower than *ExprLMap*. 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 *ExprLMap*.

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

Let us look at what happens if we vary map size M and expression size E independently for `lookup_all`. The results in Table 2 show that *ExprLMap* 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 *ExprLMap* 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 $\mathcal{O}(E)$, i.e., linear in E but constant in M. The same does not hold for search trees, where lookup time is in $\mathcal{O}(P \log M)$. $P \in \mathcal{O}(E)$ 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 *ExprLMap*, 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_all_app1`, where we already saw that *ExprLMap* 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 *ExprLMap* but is a bit slower, as before. There is no subtle scaling factor like P; just plain predictable $\mathcal{O}(E)$ like *ExprLMap*.

Returning to Table 1, we see that folding over *ExprLMap*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 *ExprLMap* 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 *foldExpr* does. Furthermore, there's the polymorphic recursion in the head case of *em_app* with a different folding function (*foldTM f*), which allocates on each call and makes it impossible to specialise *foldExpr* 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 *ExprLMap* 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 *foldExpr* 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 *foldExpr* 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 *ExprLMap* 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 *ExprLMap* linearly. By contrast, M does not seem to affect insert performance at all.

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

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 *ExprLMap* (TM). Lower is better. †indicates heap overflow.

	M \ E	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

fromList, knowing that such mutability can't be observed by the caller. Yet, it still performs worse than *ExprLMap* or *Map* for larger E, as can be seen in Table 2.

We expected *ExprLMap* 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 *ExprLMap* for small N.

What would a non-naïve version of *fromList* for *ExprLMap* look like? Perhaps the process could be sped up considerably by partitioning the input list according to the different fields of *ExprLMap* 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*[14] 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 *ExprLMap* wins in the long run, especially when there's a sharable prefix involved.

6.2 Space

We also measured the memory footprint of *ExprLMap* 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 Section 6.1.

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, *ExprLMap* consumes

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

slightly more space than both *Map* and *ExprLMap*, 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 *ExprLMap*.

However, in `space_app1` and `space_lam`, we can see that *ExprLMap* 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`, *ExprLMap* 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.

7 RELATED WORK

7.1 Matching triemaps in automated reasoning

Matching triemaps, also called *term indexing*, 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 2.1). 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[15]

```
data Term = Node Fun [Term]
```

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 *Term* 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 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

[15]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.

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 *Term* 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*[16] package, or polymorphically, requiring just a type class for trie key extraction, like the *TrieMap*[17] 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*[18] 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 3.7). 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*[19]. The *representable-tries*[20] 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*[21] 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 *ExprLMap*, *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. That is an interesting optimisation that could decrease space usage in benchmarks such as `space_app1`.

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.

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

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

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

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.

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.