The Monad.Reader Issue 22

by Anton Dergunov ⟨anton.dergunov@gmail.com⟩
and Matt Fenwick ⟨mfenwick100@gmail.com⟩
and Jay Vyas ⟨jayunit100@gmail.com⟩
and Mike Izbicki ⟨mike@izbicki.me⟩

August 6, 2013

Edward Z. Yang, editor.

Contents

Editorial

by Edward Z. Yang ⟨ezyang@cs.stanford.edu⟩

Summer brings with it the release of three articles talking about three different (and each arguably essential) ideas from the Haskell landscape. The first is an in-depth tutorial about generalized algebraic data types, covering all of the classic use-cases of this useful type system extension. The second is a meditation on the design pattern of monad transformers, and how they can help the develop the design of a parser library. The final shows how even a simple API like that of a monoid can help code achieve dramatic speed-ups. Happy reading!

Generalized Algebraic Data Types in Haskell

by Anton Dergunov ⟨anton.dergunov@gmail.com⟩

Generalized algebraic data types (GADTs) are a feature of functional programming languages which generalize ordinary algebraic data types by permitting value constructors to return specific types. This article is a tutorial about GADTs in Haskell programming language as they are implemented by the Glasgow Haskell Compiler (GHC) as a language extension. GADTs are widely used in practice: for domain-specific embedded languages, for generic programming, for ensuring program correctness and in other cases. The article describes these use cases with small GADT programs and also describes usage of GADTs in Yampa, a Haskell library.

Introduction

The usage of type systems is recognized as the most popular and best established lightweight formal method for ensuring that software behaves correctly [1]. It is used for detecting program errors, for documentation, to enforce abstraction and in other cases. The study of type systems is an active research area and Haskell is considered to be a kind of laboratory in which type-system extensions are designed, implemented and applied [2]. This article is a tutorial about one such extension – generalized algebraic data types (GADTs). The theoretical foundation for GADTs is the notion of dependent types which is extensively described in literature on computer science and logic [1].

GADTs are very useful in practice. Several examples of GADTs usage are described in this article. But these examples are not new. They were taken from many resources [3, 4, 5, 6, 7, 8, 9] and many examples are already a part of Haskell folklore.

The contribution of this article is presenting a concise introductory-level tutorial on the concept and popular use cases of GADTs. The following use cases were described in this article:

- domain-specific embedded languages;
- generic programming;
- ensuring program correctness.

We describe why type signatures are required for functions involving GADTs; the usage of GADTs in Yampa, a domain-specific language for functional reactive programming; an alternative implementation of one of the examples without GADTs usage. Additionally, several other Haskell extensions are covered in passing where necessary, such as type families and data kinds.

All examples in this article were tested with Glasgow Haskell Compiler, version 7.4.1.

GADTs in a Nutshell

Algebraic data types (ADTs) are declared using the data keyword:

```
data Test a = TI Int | TS String a
```

In the example above `Test` is called a *type constructor*. `TI` and `TS` are *value constructors*. If a type has more than one value constructor, they are called *alternatives*: one can use any of these alternatives to create a value of that type. Each alternative can specify zero or more components. For example, `TS` specifies two components: one of them has type `String` and another one – type `a`.

We can construct values of this type this way:

```
ghci> let a = TI 10
ghci> :t a
a :: Test a
ghci> let b = TS "test" 'c'
ghci> :t b
b :: Test Char
```

Value constructors of the type `Test` have the following types:

```
ghci> :t TI
TI :: Int -> Test a
ghci> :t TS
TS :: String -> a -> Test a
```

Using GADT syntax we can define the `Test` data type as:

```
data Test a where
    TI :: Int -> Test a
    TS :: String -> a -> Test a
```

The GADTs feature is a Haskell language extension. Just like other extensions it can be enabled in GHC by:

1. Using command line option `-XGADTs`.

2. Using `LANGUAGE` pragma in source files. This is the recommended way, because it enables this language extension per-file. This pragma must precede the module keyword in the source file and contain the following:

   ```
   {-# LANGUAGE GADTs #-}
   ```

The power of GADTs is not about syntax. In fact, *the single idea of GADTs is to allow arbitrary return types of value constructors*. In this way they generalize ordinary algebraic data types. Of course, this return type must still be an instance of the more general data type that is defined. We can turn the `Test` data type into a full-power GADT for example this way:

```
data Test a where
    TI :: Int -> Test Int
    TS :: String -> a -> Test a
```

We have modified the `TI` value constructor to return value of type `Test Int` and we can test this:

```
ghci> :t TI 10
TI 10 :: Test Int
```

While creating a values with the `TI` results in a value of type `Test Int`, the converse holds too: when a pattern-match succeeds on the `TI` constructor in the example below, we can now conclude that the type of `a` is `Int`. That is to say, pattern matching causes type refinement, which is the key feature of GADTs.

```
f :: Test a -> a
f (TI i) = i + 10
```

Examples in the following sections show some practical applications of GADTs.

Expression Evaluator

This section introduces GADTs with the canonical example of an expression evaluator. At first, we attempt to implement it using ordinary algebraic data types. But as we will see, GADTs allow a more elegant implementation of the evaluator.

We start with the following type of expressions involving addition of integers:

```
data IntExpr = IntVal Int
             | AddInt IntExpr IntExpr
```

`IntVal` value constructor is used to wrap integer literal and `AddInt` is used to represent an addition of two integer expressions. An example of such expression is:

```
ghci> :t AddInt (IntVal 5) (IntVal 7)
AddInt (IntVal 5) (IntVal 7) :: IntExpr
```

Evaluation function for such expressions is easy to write:

```
evaluate :: IntExpr -> Int
evaluate e = case e of
    IntVal i     -> i
    AddInt e1 e2 -> evaluate e1 + evaluate e2
```

Now we extend the type of expressions to support boolean values and add some operations on them:

```
data ExtExpr = IntVal Int
             | BoolVal Bool
             | AddInt ExtExpr ExtExpr
             | IsZero ExtExpr
```

`BoolVal` value constructor wraps Boolean literal. `IsZero` is a unary function that takes an integer and returns a Boolean value. We immediately notice a problem with this type: it is possible to write incorrect expressions that type checker will accept. For example:

```
ghci> :t IsZero (BoolVal True)
IsZero (BoolVal True) :: ExtExpr
ghci> :t AddInt (IntVal 5) (BoolVal True)
AddInt (IntVal 5) (BoolVal True) :: ExtExpr
```

Evaluation function for such expressions is also tricky. The result of evaluation can be either an integer or a Boolean value. The type `ExtExpr` is not parametrized by return value type, so we have to use type `Either Int Bool`. Furthermore, evaluation will fail if the input expression is incorrect, so we have to use type `Maybe`. Finally, the type signature of `evaluate` is the following:

```
evaluate :: ExtExpr -> Maybe (Either Int Bool)
```

Implementing this function is complicated. For example, processing `AddInt` requires usage of a nested `case`:

```
evaluate e = case e of
    AddInt e1 e2 -> case (evaluate e1, evaluate e2) of
        (Just (Left i1), Just (Left i2)) -> Just $ Left $ i1 + i2
        _ -> error "AddInt takes two integers"
```

The conclusion is that we need to represent expressions using values of types parametrized by expression return value type. *Phantom type* is a parametrized type whose parameters do not appear on the right-hand side of its definition. One can use them this way:

```
data PhantomExpr t = IntVal Int
                   | BoolVal Bool
                   | AddInt (PhantomExpr Int) (PhantomExpr Int)
                   | IsZero (PhantomExpr Int)
```

Type `t` in this type corresponds to the expression return value type. For example, integer expression has type `PhantomExpr Int`. But this type definition alone is still not helpful, because it is still possible to write incorrect expressions that type checker will accept:

```
ghci> :t IsZero (BoolVal True)
IsZero (BoolVal True) :: PhantomExpr t
```

The trick is to wrap value constructors with corresponding functions:

```
intVal :: Int -> PhantomExpr Int
intVal = IntVal
boolVal :: Bool -> PhantomExpr Bool
boolVal = BoolVal
isZero :: PhantomExpr Int -> PhantomExpr Bool
isZero = IsZero
```

And now bad expressions are rejected by type checker:

```
ghci> :t isZero (boolVal True)
Couldn't match expected type 'Int' with actual type 'Bool'...
ghci> :t isZero (intVal 5)
isZero (intVal 5) :: PhantomExpr Bool
```

Ideally, we want the following type signature for `evaluate` method:

```
evaluate :: PhantomExpr t -> t
```

But we can't define such function. For example, the following line produces error "`Couldn't match type 't' with 'Int'`":

```
evaluate (IntVal i) = i
```

The reason of this error is that return type of value constructor `IntVal` is `Phantom t` and `t` can be refined to any type. For example:

```
ghci> :t IntVal 5 :: PhantomExpr Bool
IntVal 5 :: PhantomExpr Bool :: PhantomExpr Bool
```

What is really needed here is to specify type signature of value constructors exactly. In this case pattern matching in `evaluate` would cause type refinement for `IntVal`. And this is exactly what GADTs do.

As described in the previous section, GADTs use a different syntax than ordinary algebraic data types. In fact, value constructors specified by the data type `PhantomExpr` can be written as the following functions:

```
IntVal  :: Int -> PhantomExpr t
BoolVal :: Bool -> PhantomExpr t
AddInt  :: PhantomExpr Int -> PhantomExpr Int -> PhantomExpr t
IsZero  :: PhantomExpr Int -> PhantomExpr t
```

Using GADT syntax, the data type `PhantomExpr` type can be declared this way:

```
data PhantomExpr t where
    IntVal  :: Int -> PhantomExpr t
    BoolVal :: Bool -> PhantomExpr t
    AddInt  :: PhantomExpr Int -> PhantomExpr Int -> PhantomExpr t
    IsZero  :: PhantomExpr Int -> PhantomExpr t
```

All value constructors have `PhantomExpr t` as their return type. As noted in the previous section, the distinctive feature of GADTs is the ability to return specific types in value constructors, for example `PhantomExpr Int`. The expression language can be written with GADTs in this fashion:

```
data Expr t where
    IntVal  :: Int -> Expr Int
    BoolVal :: Bool -> Expr Bool
    AddInt  :: Expr Int -> Expr Int -> Expr Int
    IsZero  :: Expr Int -> Expr Bool
    If      :: Expr Bool -> Expr t -> Expr t -> Expr t
```

Note that value constructors of this data type have specific return types. Now bad expressions are rejected by the type checker:

```
ghci> :t IsZero (BoolVal True)
Couldn't match expected type ‘Int’ with actual type ‘Bool’...
ghci> :t IsZero (IntVal 5)
IsZero (IntVal 5) :: Expr Bool
```

Note that the type of value `IsZero (IntVal 5)` is specific: `Expr Bool`.

GADTs allow to write well-defined `evaluate` function:

```
evaluate :: Expr t -> t
evaluate (IntVal i)     = i
evaluate (BoolVal b)    = b
evaluate (AddInt e1 e2) = evaluate e1 + evaluate e2
evaluate (IsZero e)     = evaluate e == 0
evaluate (If e1 e2 e3)  = if evaluate e1 then
    evaluate e2 else evaluate e3
```

Pattern matching causes type refinement: the right-hand side of the following expression `i` has type `Int`:

```
evaluate :: Expr t -> t
evaluate (IntVal i) = ...
```

The type of `AddInt e1 e2` expression is `Expr Int` and the types of `e1` and `e2` must also be `Expr Int`, so we can evaluate recursively the individual expressions and then return the sum (value of type `Int`).

At the end of this article, we describe one more implementation of expression evaluator.

Generic Programming with GADTs

In datatype-generic programming, functions take a type as an argument and change behavior depending on the structure of this type. There are several approaches to such kind of generic programming in Haskell. A paper by Hinze et al. [10] provides an overview of these approaches. In this section, we present an approach which uses GADTs. Ideas for this section were taken from another paper by Hinze at al. [3].

Suppose we would like to write a function to encode data in binary form. This function must be able to work with values of several types. Functions like this one can be implemented using type classes. However, GADTs offer an interesting alternative.

First we need to declare a *representation type* [4], a type whose values represent types:

```
data Type t where
    TInt  :: Type Int
    TChar :: Type Char
    TList :: Type t -> Type [t]
```

This is GADT with value constructors that create a representation of the corresponding type. For example:

```
ghci> let a = TInt
ghci> :t a
a :: Type Int
ghci> let b = TList TInt
ghci> :t b
b :: Type [Int]
```

`String` type is defined in Haskell as a list of `Char` elements, so we can define a value constructor for string type representation this way:

```
tString :: Type String
tString = TList TChar
```

The output of the encoding function is a list of bits where bits are represented using:

```
data Bit = F | T deriving(Show)
```

The encoding function takes a representation of the type, the value of this type and returns a list of bits.

```
encode :: Type t -> t -> [Bit]
encode TInt i = encodeInt i
encode TChar c = encodeChar c
encode (TList _) [] = F : []
encode (TList t) (x : xs) = T :
    (encode t x) ++ encode (TList t) xs
```

We can test this function:

```
ghci> encode TInt 333
[T,F,T,...,F,F,F]
ghci> encode (TList TInt) [1,2,3]
[T,T,F,...,F,F,F]
ghci> encode tString "test"
[T,F,F,...,F,F,F]
```

If we pair the representation type and the value together, we get a *universal data type*, the type `Dynamic` (this code requires using `ExistentialQuantification` extension):

```
data Dynamic = forall t. Dyn (Type t) t
```

Above we have defined an *existential data type* which can also be represented as a GADT:

```
data Dynamic where
    Dyn :: Type t -> t -> Dynamic
```

Now we can declare a variant of `encode` function which gets a `Dynamic` type value as input:

```
encode' :: Dynamic -> [Bit]
encode' (Dyn t v) = encode t v
```

The following session illustrates the usage of this type:

```
ghci> let c = Dyn (TList TInt) [5,4,3]
ghci> :t c
c :: Dynamic
ghci> encode' c
[T,T,F,...,F,F,F]
```

We can now define heterogeneous lists using the `Dynamic` type:

```
ghci> let d = [Dyn TInt 10, Dyn TString "test"]
ghci> :t d
d :: [Dynamic]
```

However, we can't make this list a `Dynamic` value itself. To fix this problem, we need to extend the representation type, adding a value constructor for the `Dynamic` data type.

```
data Type t where
    ...
    TDyn :: Type Dynamic
```

We also need to update `encode` function to handle the `Dynamic` data type:

```
encode :: Type t -> t -> [Bit]
...
encode TDyn (Dyn t v) = encode t v
```

Now we can represent a list of `Dynamic` values as a `Dynamic` value itself and encode it:

```
ghci> let d = [Dyn TInt 10, Dyn tString "test"]
ghci> :t d
d :: [Dynamic]
ghci> let e = Dyn (TList TDyn) d
ghci> :t e
e :: Dynamic
ghic> encode' e
[T,F,T,...,F,F,F]
```

The `Dynamic` data type is useful for communication with the environment when the actual type of the data is not known in advance. In this case a type cast is required to get useful data. A simple way to implement a type cast from `Dynamic` data type to an integer is the following:

```
castInt :: Dynamic -> Maybe Int
castInt (Dyn TInt i) = Just i
castInt (Dyn _ _) = Nothing
```

There is a more generic solution [3] for this problem that works for all types, not just integer, but it is out of scope for this article.

While the presented approach is an important use case of GADTs, the disadvantage of this approach is that we have to extend the representation type whenever we define a new data type.

Proving Correctness of List Operations

An important role of type systems is to ensure that data is manipulated in appropriate ways (for example, to ensure we pass a list to `head` function). But types can be used to express more sophisticated properties. For example, we can define a type of lists of a particular length and then define a `headSafe` function that only accepts non-empty lists. The idea described in this section is to use types to express correctness properties and then use type checker of the programming language to ensure that we can express only those programs that have the desired properties. This idea was developed in Ωmega system [11, 12]. In this section, we use GADTs in the Haskell programming language to prove the correctness of list operations; in the next section, we describe how to use GADTs to prove correctness of the insertion operation in red-black trees.

Lists can be represented using the following algebraic data type:

```
data List t = Nil | Cons t (List t)
```

or using GADT syntax as:

```
data List t where
    Nil :: List t
    Cons :: t -> List t -> List t
```

Now `head` function can be implemented this way:

```
listHead :: List t -> t
listHead (Cons a _) = a
listHead Nil = error "list is empty"
```

The disadvantage of this function is that it can fail: it fails when a list is empty and succeeds otherwise. To address this problem we define a type of non-empty lists. First we define two empty data types (this requires `EmptyDataDecls` extension):

```
data Empty
data NonEmpty
```

Now we define a safe list GADT:

```
data SafeList t f where
    Nil :: SafeList t Empty
    Cons :: t -> SafeList t f -> SafeList t NonEmpty
```

Parameter f takes type Empty when the list is empty and NonEmpty otherwise. The function headSafe is a safe version of listHead function that only accepts non-empty lists as parameter.

```
headSafe :: SafeList t NonEmpty -> t
headSafe (Cons t _) = t
```

For example:

```
ghci> headSafe Nil
Couldn't match expected type ‘NonEmpty’ with actual type ‘Empty’
ghci> headSafe $ Cons 1 $ Cons 2 $ Cons 3 Nil
1
```

However, the implementation of a function to create a list containing an element repeated a given number of times using SafeList data type is problematic: it is not possible to determine return value type of this function.

```
repeatElem :: a -> Int -> SafeList a ???
repeatElem a 0 = Nil
repeatElem a n = Cons a (repeatElem a (n-1))
```

The problem is that empty and non-empty lists have completely different types. To fix this problem we can slightly relax Cons value constructor:

```
data SafeList t f where
    Nil :: SafeList t Empty
    Cons :: t -> SafeList t f -> SafeList t f'
```

Now SafeList t Empty is a type of possibly empty lists, for example:

```
ghci> :t Nil
Nil :: SafeList t Empty
ghci> :t Cons 'a' Nil
Cons 'a' Nil :: SafeList Char f'
ghci> :t Cons 'a' Nil :: SafeList Char Empty
Cons 'a' Nil :: SafeList Char Empty :: SafeList Char Empty
ghci> :t Cons 'a' Nil :: SafeList Char NonEmpty
Cons 'a' Nil :: SafeList Char NonEmpty :: SafeList Char NonEmpty
```

And we can define repeatElem as a function returning possibly empty lists:

```
repeatElem :: a -> Int -> SafeList a Empty
repeatElem a 0 = Nil
repeatElem a n = Cons a (repeatElem a (n-1))
```

It's worth noting that with the current data type definition, a term `Cons 'a' Nil` can even be given the type `SafeList Char Int`. To fix this problem, we need to somehow give the `Empty` and `NonEmpty` types the same kind. This is discussed later for `Nat` data type.

Regardless, `SafeList` data type does not have enough static information to prove stronger list length invariants for many list functions. For example, for the concatenation function we need to show that length of the concatenated list is a sum of source lists lengths. It is not enough to just know if a list is empty or not: we need to encode the length of a list in its type.

The classical way to encode numbers at the type level is Peano numbers:

```
data Zero
data Succ n
```

Zero is encoded as `Zero`, one – as `Succ Zero`, two – as `Succ (Succ Zero)` and so on. Now the list data type is defined as:

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

Function `headSafe` can be defined as:

```
headSafe :: List t (Succ n) -> t
headSafe (Cons t _) = t
```

We can also show that the safe map function does not change the length of a list:

```
mapSafe :: (a -> b) -> List a n -> List b n
mapSafe _ Nil = Nil
mapSafe f (Cons x xs) = Cons (f x) (mapSafe f xs)
```

To implement the concatenation function, we need a type-level function for addition of Peano numbers. A natural way to implement such function is to use *type families* (which in context of this tutorial can be understood as type-level functions). First, we need to declare type family `Plus` (this requires `TypeFamilies` extension):

```
type family Plus a b
```

Then, we need to declare type instances that implement addition of Peano numbers by induction:

```
type instance Plus Zero n = n
type instance Plus (Succ m) n = Succ (Plus m n)
```

The type family `Plus` that we have just defined can be used in the concatenation function signature:

```
concatenate :: List a m -> List a n -> List a (Plus m n)
concatenate Nil ys = ys
concatenate (Cons x xs) ys = Cons x (concatenate xs ys)
```

As mentioned previously, `Succ` has a type parameter of kind `*`, so it possible to write nonsensical terms like `Succ Int` which will be accepted by the type checker. This problem can be addressed using a new *kind*. Just as types classify values, kinds classify types. We can declare the following data type:

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

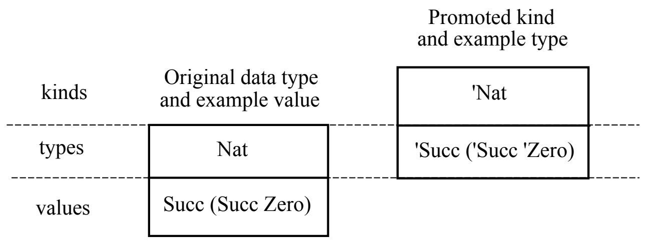

Figure 1: Promotion of a data type to a kind.

Here `Nat` is a type; `Zero` and `Succ` are value constructors. But due to promotion [13] `Nat` also becomes a kind; `Zero` and `Succ` also become types (see Figure 1). Where necessary, a quote must be used to resolve ambiguity. For example, `'Succ` refers to a type, not a value constructor. So, type-level representation of the number two can be written as:

```
type T = 'Succ ('Succ 'Zero)
```

Quotes can be omitted in this case, because there is no ambiguity:

```
type T2 = Succ (Succ Zero)
```

As a result, type checker now rejects wrong terms like `Succ Int`.

The definition of the list data type can also be improved now to clearly specify that the type of its second parameter has kind `Nat` (this requires `DataKinds` extension):

```
data List a (n::Nat) where
    Nil :: List a 'Zero
    Cons :: a -> List a n -> List a ('Succ n)
```

After the changes that we have made to the definition of the `List` data type, the implementation of the `repeatElem` function becomes more involved, because now we can't yet write its return type:

```
repeatElem :: a -> Int -> List a ???
repeatElem a 0 = Nil
repeatElem a n = Cons a (repeatElem a (n-1))
```

On the one hand, the count parameter must be passed as a value to populate the list at run-time. On the other hand, we need a type-level representation of the same number for `List` type. Haskell enforces a phase separation between run-time values and compile-time types. This causes a problem with the type signature of the function `repeatElem`: there is no direct way to specify a type-level representation of list's count in the return value type of the function which will match the count passed in as a value.

The solution to this puzzle is the use of *singleton types*. Singleton types are types that contain only one value (except, of course, the $\perp$ value, which is a member of every type in Haskell).

The singleton for Peano numbers type can be expressed using the following GADT:

```
data NatSing (n::Nat) where
    ZeroSing :: NatSing 'Zero
    SuccSing :: NatSing n -> NatSing ('Succ n)
```

The constructors of the singleton `NatSing` mirror those of the kind `Nat`. As a result, every type of kind `Nat` corresponds to exactly one value (except $\perp$ value) of the singleton data type where parameter `n` has exactly this type (see Figure 2). For example:

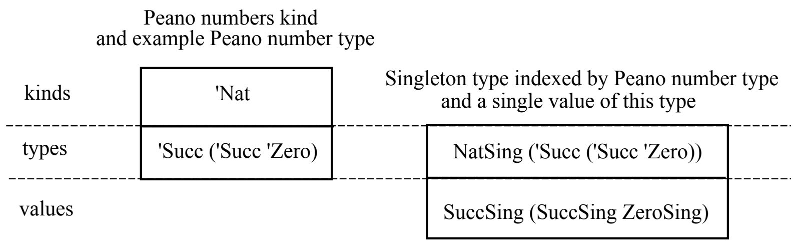
Peano numbers kind
and example Peano number type
Singleton type indexed by Peano number type
and a single value of this type
kinds
types
values
'Nat
'Succ ('Succ 'Zero)
NatSing ('Succ ('Succ 'Zero))
SuccSing (SuccSing ZeroSing)

Figure 2: Singleton type for Peano numbers.

```
ghci> :t ZeroSing
ZeroSing :: NatSing 'Zero
ghci> :t SuccSing $ SuccSing ZeroSing
SuccSing $ SuccSing ZeroSing :: NatSing ('Succ ('Succ 'Zero))
```

Now function `repeatElem` can be defined this way:

```
repeatElem :: a -> NatSing n -> List a n
repeatElem _ ZeroSing     = Nil
repeatElem x (SuccSing n) = Cons x (repeatElem x n)
```

In a function returning an element by index in the list we need to make sure that the index does not exceed the list length. This requires a type-level function to compute whether one number is less than the other. We define the following type family and instances (the `TypeOperators` extension is required to be able to define `:<` operation for types):

```
type family (m::Nat) :< (n::Nat) :: Bool
type instance m         :< 'Zero     = 'False
type instance 'Zero     :< ('Succ n) = 'True
type instance ('Succ m) :< ('Succ n) = m :< n
```

This type-level function is implemented using induction. It returns promoted type `'True` of kind `Bool` when first number is less than the second one.

Now the function can be defined this way:

```
nthElem :: (n :< m) ~ 'True => List a m -> NatSing n -> a
nthElem (Cons x _)  ZeroSing     = x
nthElem (Cons _ xs) (SuccSing n) = nthElem xs n
```

The tilde operation is an *equality constraint*. It asserts that two types are the same in the context. Thus, is it only possible to use this function when the index does not exceed the list length.

We have shown that GADTs provide a way to use Haskell type checker to verify correctness of list operations. To do this, we need to specify necessary properties in the data type. The set of such properties is motivated by the operations that we want to implement. At first we only made a distinction between empty and non-empty lists. This was suitable for the `listHead` function. To implement `repeatElem`, `concatenate` and other functions we extended the list data type with the count of elements that it contains.

Proving Correctness of Red-Black Tree Insert Operation

This section describes another, more involved example of using GADTs to verify correctness of programs: proving correctness of insertion in red-black trees. A red-black tree is a binary search tree where every node has either red or black color and which additionally satisfies several invariants which we describe later. The presense of invariants guarantees that the tree is balanced and thus searching takes $O(\log n)$ time (where n is total number of elements). The insertion operation must maintain these properties, so it is interesting to see how GADTs can be used to enforse these invariants.

We use the implementation of red-black trees described by Okasaki [6]. The source code for the verified red-black tree was originally written by Stephanie Weirich [7] for a university course.

```
data Color = R | B
data Node a = E | N Color (Node a) a (Node a)
type Tree a = Node a
```

`N` is a value constructor of a regular node and `E` is a value constructor for a leaf node. As in all binary search trees, for a particular node `N c l x r` values less than `x` are stored in left sub-tree (in `l`) and values greater than `x` are stored in right sub-tree (in `r`). Membership function implements a recursive search:

```
member :: Ord a => a -> Tree a -> Bool
member _ E  = False
member x (N _ l a r)
    | x < a  = member x l
    | x > a  = member x r
    | otherwise = True
```

Additionally red-black tree satisfies the following invariants:

1. The root is black.
2. Every leaf is black.
3. Red nodes have black children.
4. For each node, all paths from that node to the leaf node contain the same number of black nodes. This number of black nodes is called the *black height* of a node.

These invariants guarantee that tree is balanced. Indeed, the longest path from the root node (containing alternating red-black nodes) can only be twice as long as the shortest path (containing only red nodes). Thus basic operations (such as insertion and search) take $O(\log n)$ time in the worst case.

Insertion operation for red-black trees has the following structure:

```
insert :: Ord a => Tree a -> a -> Tree a
insert t v = blacken (insertInternal t v) where
    insertInternal n@(N c l a r) x
        | x < a = leftBalance  (N c (insertInternal l x) a r)
        | x > a = rightBalance (N c l a (insertInternal r x))
        | otherwise = n
    insertInternal E x = N R E x E
    blacken (N _ l x r) = N B l x r
```

It has the same structure as insertion operation for regular binary search trees which is implemented by recursive descent (down to leaf nodes) until a suitable location for insertion is found. But additionally it must keep the invariants, so there are the following differences:

- The node is inserted with red color. This allows to maintain the 4th invariant, because the black height is not changed.
- To maintain the 1st invariant we call `blacken` at the end of insertion. Again, the 4th invariant remains valid.
- To maintain the 3rd invariant we call `leftBalance` and `rightBalance`.

Figure 3 shows 2 possible cases when the 3rd invariant is violated after insertion in the left branch of the node. To repair this invariant the tree must be restructured as shown on the figure. The following code uses pattern matching to implement the restructuring, otherwise the function returns the sub-tree as is:

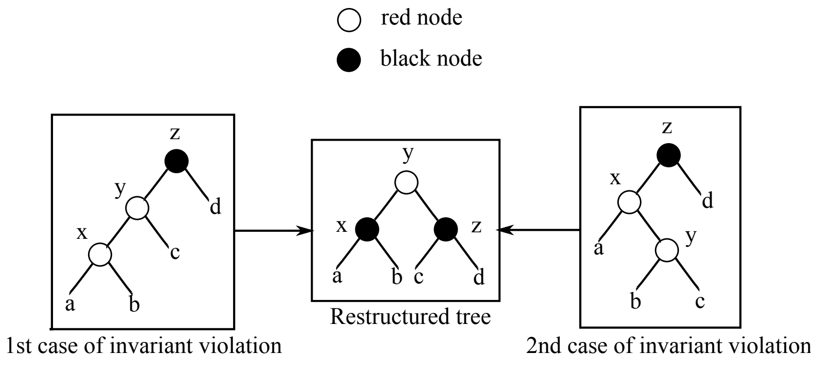
red node
black node
z
y
d
x
c
a
b
1st case of invariant violation
y
x
z
a
b
c
d
Restructured tree
z
x
d
a
y
b
c
2nd case of invariant violation

Figure 3: Possible cases of 3rd invariant violation after insertion in the left branch of the node.

```
leftBalance :: Node a -> Node a
leftBalance (N B (N R (N R a x b) y c) z d) =
    N R (N B a x b) y (N B c z d)
leftBalance (N B (N R a x (N R b y c)) z d) =
    N R (N B a x b) y (N B c z d)
leftBalance n = n
```

The function `rightBalance` is similar. Complete source code is in Appendix A.

Proving that the 4th invariant is maintained by insert

To show that the 4th invariant is maintained by the insertion operation we need to add black height as a parameter of the `Node` data type.

First we define Peano numbers in same way as before:

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

Then we turn Node into a GADT with a black height parameter:

```
data Node a (bh::Nat) where
    E :: Node a 'Zero
    N :: Color -> Node a bh -> a -> Node a bh -> Node a ???
```

The leaf node has black height 0. The definition of internal nodes requires that both children have the same black height. The black height of the node itself must be conditionally incremented based on its color. This is implemented using the following type family which computes the new height based on the color of the node and black height of its children. Both parameters are represented as types (of `Color` and `Nat` kinds correspondingly). This code requires `TypeFamilies` and `DataKinds` extensions.

```
type family   IncBlackHeight (c::Color) (bh::Nat) :: Nat
type instance IncBlackHeight R bh = bh
type instance IncBlackHeight B bh = Succ bh
```

Now we see that a color must be passed as a type (for `IncBlackHeight` type family) and as a value (to the value constructor). Similarly as before, we need to use a singleton type as a bridge:

```
data ColorSingleton (c::Color) where
    SR :: ColorSingleton R
    SB :: ColorSingleton B
```

The value of this singleton type is passed as a parameter to the node value constructor and the color type is used for type family:

```
data Node a (bh::Nat) where
    E :: Node a 'Zero
    N :: ColorSingleton c -> Node a bh -> a
        -> Node a bh -> Node a (IncBlackHeight c bh)
```

After we have added a new parameter for the `Node` data type, it is an error to write:

```
type Tree a = Node a bh
```

since normally when creating a new type in Haskell, every type variable that appears on the right-hand side of the definition must also appear on its left-hand side. One solution to this problem is usage of *existential types* (this definition requires extension `RankNTypes`):

```
type Tree a = forall bh. Node a bh
```

It is also possible to do this with a GADT:

```
data Tree a where
    Root :: Node a bh -> Tree a
```

The implementation of insertion operation never violates the 4th invariant, so the remaining changes are adjustments of type annotations and so on. Complete source code is in Appendix B.

Proving that the 3rd invariant is maintained by insert

Proving the 3rd invariant is more involved. First, we need to specify valid colors for a node on the type level. This can be done using type families as before or using type classes. We choose the latter and define a type class with 3 parameters corresponding to color of the parent and colors of the child nodes (this code requires `MultiParamTypeClasses` extension):

```
class ValidColors (parent::Color) (child1::Color) (child2::Color)
```

We do not need to define any functions in this type class, because our aim is just to declare instances with valid colors (this code requires `FlexibleInstances` extension):

```
instance ValidColors R B B
instance ValidColors B c1 c2
```

The allowed nodes are:

- ► red nodes with black child nodes;
- ► black nodes with child nodes of any color.

We need to add color type as a parameter to the `Node` data type and restrict it to have only correctly-colored nodes using the `ValidColors` type class:

```
data Node a (bh::Nat) (c::Color) where
    E :: Node a 'Zero B
    N :: ValidColors c c1 c1 => ColorSingleton c -> Node a bh c1
        -> a -> Node a bh c2 -> Node a (IncBlackHeight c bh) c
```

With this change we also statically ensure the 2nd invariant: leaf nodes have black color.

We also need to update the definition of the `Tree` data type to specify that root node has black color (this way also ensuring the 1st invariant):

```
data Tree a where
    Root :: Node a bh B -> Tree a
```

The implementation of the insertion operation can temporarily invalidate the 3rd invariant (see Figure 3), so during insertion we are not able to represent the tree using this data type. Thus, we need to declare a data type similar to `Node`, but without the restriction on node colors:

```
data IntNode a (n::Nat) where
    IntNode :: ColorSingleton c -> Node a n c1 -> a
        -> Node a n c2 -> IntNode a (IncBlackHeight c n)
```

As before, we need to make changes in type annotations of the functions implementing insert operation. We also need to change the `leftBalance` function type signature this way:

```
leftBalance :: ColorSingleton c -> IntNode a n -> a
    -> Node a n c' -> IntNode a (IncBlackHeight c n)
```

Earlier, we passed the whole node as a parameter. But we can't do this after the `Node` data type was modified: the 3rd invariant could be violated due to insertion in the left branch of the node. So, we pass all parameters of the parameters of the node and left child is represented using `IntNode` data type.

Previous cases should be rewritten using new types:

```
leftBalance SB (IntNode SR (N SR a x b) y c) z d =
    IntNode SR (N SB a x b) y (N SB c z d)
leftBalance SB (IntNode SR a x (N SR b y c)) z d =
    IntNode SR (N SB a x b) y (N SB c z d)
```

However, now we can't write the same catch-all case as before:

```
leftBalance c (IntNode c1 a x b) d n2 =
    IntNode c (N c1 a x b) d n2
```

This case does not type-check with the following error:

```
Could not deduce (ValidColors c1 c2 c2) ...
```

The reason is that the type of the left node is `IntNode`, so even though we have previously balanced the left sub-tree, technically this is not reflected in the type. We need to explicitly match against the correct cases and reconstruct the node. First, we match against the black nodes where children can have any color:

```
leftBalance c (IntNode SB a x b) z d = IntNode c (N SB a x b) z d
```

Red nodes must have black children:

```
leftBalance c (IntNode SR a@(N SB _ _ _) x b@(N SB _ _ _)) z d =
    IntNode c (N SR a x b) z d
leftBalance c (IntNode SR E x E) z d = IntNode c (N SR E x E) z d
```

Unfortunately, we haven't yet listed all cases. We know that the following cases can't happen, but we do not have enough information in the type to omit them. We can skip them, but this means producing "Non-exhaustive patterns" exception for these impossible cases. To workaround the corresponding warning message from Haskell compiler we are using the `error` function:

```
leftBalance _ (IntNode SR (N SR _ _ _) _ _) _ _ =
    error "can't happen"
leftBalance _ (IntNode SR _ _ (N SR _ _ _)) _ _ =
    error "can't happen"
```

However, the previous code illustrates a general problem with proofs. In fact, in Haskell $\perp$ (bottom) is a member of every type. As a result, we can write:

```
concatenate :: List a m -> List a n -> List a (Plus m n)
concatenate = undefined
```

This code type checks, but, of course, the implementation of the `concatenate` function does not meet our expectations.

At this point, we have specified all cases of the `leftBalance` function (note that the case of one regular node and one leaf node is not valid, because these nodes must have different black heights; so, we do not need to take care of this case). The complete source code of this example is in Appendix C.

To sum up, we have implemented and verified the insertion operation on red-black trees. By using GADTs to express invariants for this data type, we guarantee that the tree is balanced and that searching in such tree will take $O(\log n)$ time (where n is total number of elements in the tree).

Type Signatures for Functions Involving GADTs

Type inference for programs with GADTs is out of scope for this article. But when writing programs we are faced with a problem: typically Haskell functions do not require writing type signatures, but many functions involving GADTs do not compile without specifying type signatures explicitly, as the Haskell compiler can not automatically infer the type. This section describes the reasons of this situation.

Hindley-Milner (HM) is the classic type inference method [14]. One of the most important properties of HM is ability to always deduce the *most general type* (*principle type*) of every term.

However, GADTs pose a difficult problem for type inference, because programs with GADTs lose principle type property [15]. For example, consider the following GADT program:

```
data Test t where
    TInt :: Int -> Test Int
    TString :: String -> Test String

f (TString s) = s
```

There are two possible principal types of the function `f`, but neither of them is an instance of the other:

```
f :: Test t -> String
f :: Test t -> t
```

Also without type signature the following function fails to typecheck:

```
f' (TString s) = s
f' (TInt i) = i
```

Adding type signature fixes the problem:

```
f' :: Test t -> t
```

The paper by Schrijvers et al. [15] provides more information on type inference for programs with GADTs.

Usage of GADTs in Yampa

Yampa [16] is a domain-specific language for functional reactive programming (FRP). FRP is a programming paradigm which involves expressing data flows using functional programming languages. Based on the information from the paper by Nilsson [9], this section describes how GADTs were used to improve performance of Yampa programs.

The signal function is a central abstraction in Yampa. It represents a simple synchronous process that maps an input signal to an output signal (see Figure 4). The type of the signal function is `SF a b` and it can be constructed from an ordinary function using the following function:

```
arr :: (a -> b) -> SF a b
```

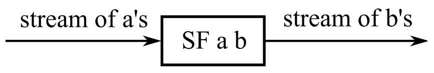

Figure 4: Signal function `SF a b`.

The following function provides a composition of signal functions (as shown in Figure 5):

```
(>>>) :: SF a b -> SF b c -> SF a c
```

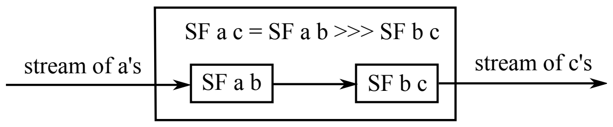

Figure 5: Composition of signal functions.

There is a natural requirement to eliminate the overhead of composition with identity function:

```
arr id >>> f = f
f >>> arr id = f
```

As an attempt to implement this in Yampa we can imagine introducing a special value constructor to represent identity signal functions:

```
data SF a b = ...
            | SFId -- Represents arr id.
```

But the return type of this value constructor is still `SF a b`. We can use the same trick as before with phantom types. We can define a function to construct the value and restrict the type to `SF a a`:

```
identity :: SF a a
identity = SFId
```

Now we can try to use the new value constructor in the definition of the function `>>>` this way:

```
(>>>) :: SF a b -> SF b c -> SF a c
...
SFId >>> sf = sf
sf >>> SFId = sf
```

But this code does not type check, because when we pattern match using `SFId` value constructor, the type is still `SF a b`, not `SF a a`. We have already seen this problem before when we attempted to use phantom types for expression evaluation. The solution is to use GADT to represent the signal function:

```
data SF a b where
    ...
    SFId :: SF a a
```

Benchmark	T_S[s]	T_G[s]
1	0.41	0.00
2	0.74	0.22
3	0.45	0.22
4	1.29	0.07
5	1.95	0.08
6	1.48	0.69
7	2.85	0.72

Table 1: Performance improvements enabled by GADTs in Yampa programs

After this change the function `>>>` as presented above must type check due to type refinement in pattern matching.

There are other performance improvements that are enabled by GADTs in Yampa [9]. The results of performance improvements described by the paper by Nilsson [9] are shown in the table. The table shows execution time of several benchmarks using initial simply-optimized implementation (T_S) and the implementation with GADT-based optimizations (T_G). In addition to performance improvements, GADTs allowed the authors of the paper to write a more concise and cleaner API without the need of pre-composed signal functions (which were defined in Yampa only for performance reasons).

GADTless Programming

Previous sections should convince the reader that GADTs are a very powerful and helpful extension of the language. However, there are cases when this extension is not available (for example, this feature is not implemented in Hugs compiler). For this reason, there is an interest in replacing them with simpler features while not substantially changing programs and their meanings. This is called GADTless programming [17].

Earlier we discussed implementation of expression evaluator using GADT. It can also be implemented using type classes [8]:

```
class Expr e where
    intVal  :: Int -> e Int
    boolVal :: Bool -> e Bool
    add     :: e Int -> e Int -> e Int
    isZero  :: e Int -> e Bool
    if'     :: e Bool -> e t -> e t -> e t
```

Bad expressions are still rejected by the type checker:

```
ghci> :t isZero $ boolVal True
Couldn't match expected type 'Int' with actual type 'Bool'...
ghci> :t isZero $ intVal 5
isZero $ intVal 5 :: Expr e => e Bool
```

Evaluation is implemented by defining a helper data type as an instance of `Expr e` type class:

```
newtype Eval a = Eval {runEval :: a}

instance Expr Eval where
    intVal x  = Eval x
    boolVal x = Eval x
    add x y   = Eval $ runEval x + runEval y
    isZero x  = Eval $ runEval x == 0
    if' x y z = if (runEval x) then y else z

t = runEval $ isZero $ intVal 5
```

Printing expressions is implemented in a similar way:

```
newtype Print a = Print {printExpr :: String}

instance Expr Print where
    intVal x  = Print $ show x
    boolVal x = Print $ show x
    add x y   = Print $ printExpr x ++ "+" ++ printExpr y
    isZero x  = Print $ "isZero(" ++ printExpr x ++ ")"
    if' x y z = Print $ "if (" ++ printExpr x ++ ") then (" ++
        printExpr y ++ ") else (" ++ printExpr z ++ ")"

t' = printExpr $ isZero $ intVal 5
```

Detailed discussion of GADTless programming and comparing advantages of these approaches is out of scope of this paper. However, it is worth pointing out that it is much easier to do deep pattern matching with data types (including GADTs) than with class instances. For example, earlier we have used deep pattern matching in `leftBalance` function:

```
leftBalance :: Node a -> Node a
leftBalance (N B (N R (N R a x b) y c) z d) =
    N R (N B a x b) y (N B c z d)
leftBalance (N B (N R a x (N R b y c)) z d) =
    N R (N B a x b) y (N B c z d)
leftBalance n = n
```

Another important aspect is that GADTs, just like all data types in Haskell, are closed: once they are declared, they can not be extended in the rest of the program. This issue is called the *expression problem* [18]. Wouter Swierstra describes a technique to solve this problem in the article “Data types à la carte” [19].

In contrast, type class instances are open. For example, we could split `Expr e` type class into two separate type classes (this way we improve modularity):

```
class ArithExpr e where
    intVal  :: Int -> e Int
    add     :: e Int -> e Int -> e Int
    isZero  :: e Int -> e Bool

class LogExpr e where
    boolVal :: Bool -> e Bool
    if'     :: e Bool -> e t -> e t -> e t

class (ArithExpr e, LogExpr e) => Expr e
```

Conclusion

This article has demonstrated the use of GADTs in practice:

- We have shown that GADTs are useful for domain-specific embedded languages: they allow to statically type-check valid expressions.
- GADTs allow to express datatype-generic functions using representation types and universal data types.
- GADTs can be used as a lightweight way to ensure program correctness. They allow to encode domain-specific invariants in data type. The programmer can decide which parts of her or his program require verification and add only relevant invariants. Haskell enforces a phase separation between run-time values and compile-time types. Invariants are expressed using types, so there is no additional run-time cost. But on the other hand, we have shown the issue with the $\perp$ value.

- We have also described how GADTs were used to improve performance of Yampa programs.

Acknowledgments

This tutorial article was inspired by Simon Peyton-Jones talk at LASER 2012 summer school. The source code for the red-black tree example was originally written by Stephanie Weirich for a university course. The author is thankful to have ability to use this great example of GADTs usage in this tutorial. Also the author would like to thank Simon Peyton-Jones, Owen Stephens, Alexander Batischev, Denis Kasak, Mann mit Hut, Alexei Alexandrov, Serge Le Huitouze, Eric Mullen, Egor Kazachkov and Vadim Zaytsev for the review of the article draft and providing valuable comments.

References

[1] Benjamin C. Pierce. **Types and programming languages**. MIT Press, Cambridge, MA, USA (2002).

[2] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In **Proceedings of the third ACM SIGPLAN conference on History of programming languages**, pages 12–1–12–55. HOPL III, ACM, New York, NY, USA (2007). `http://doi.acm.org/10.1145/1238844.1238856`.

[3] R. Hinze **et al.** Fun with phantom types. **The fun of programming**, pages 245–262 (2003).

[4] Stephanie Weirich. RepLib: a library for derivable type classes. In **Proceedings of the 2006 ACM SIGPLAN workshop on Haskell**, pages 1–12. Haskell '06, ACM, New York, NY, USA (2006). `http://doi.acm.org/10.1145/1159842.1159844`.

[5] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In **Proceedings of the 2012 symposium on Haskell symposium**, pages 117–130. Haskell '12, ACM, New York, NY, USA (2012). `http://doi.acm.org/10.1145/2364506.2364522`.

[6] Chris Okasaki. Red-black trees in a functional setting. **Journal of Functional Programming**, 9(4):pages 471–477 (July 1999). `http://dx.doi.org/10.1017/S0956796899003494`.

[7] Stephanie Weirich. Dependently-typed programming in GHC. In **Proceedings of the 11th international conference on Functional and Logic Programming**, pages 3–3. FLOPS'12, Springer-Verlag, Berlin, Heidelberg (2012). `http://dx.doi.org/10.1007/978-3-642-29822-6_3`.

[8] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. **Journal of Functional Programming**, 19(5):pages 509–543 (September 2009). `http://dx.doi.org/10.1017/S0956796809007205`.

[9] Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In **Proceedings of the tenth ACM SIGPLAN international conference on Functional programming**, pages 54–65. ICFP '05, ACM, New York, NY, USA (2005). `http://doi.acm.org/10.1145/1086365.1086374`.

[10] Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. In **Proceedings of the 2006 international conference on Datatype-generic programming**, pages 72–149. SSDGP'06, Springer-Verlag, Berlin, Heidelberg (2007). `http://dl.acm.org/citation.cfm?id=1782894.1782896`.

[11] Tim Sheard. Putting Curry-Howard to work. In **Proceedings of the 2005 ACM SIGPLAN workshop on Haskell**, pages 74–85. Haskell '05, ACM, New York, NY, USA (2005). `http://doi.acm.org/10.1145/1088348.1088356`.

[12] Tim Sheard and Nathan Linger. Programming in Omega. In Zoltán Horváth, Rinus Plasmeijer, Anna Soós, and Viktória Zsók (editors), **CEFP**, volume 5161 of **Lecture Notes in Computer Science**, pages 158–227. Springer (2007). `http://dx.doi.org/10.1007/978-3-540-88059-2_5`.

[13] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In **Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation**, pages 53–66. TLDI '12, ACM, New York, NY, USA (2012). `http://doi.acm.org/10.1145/2103786.2103795`.

[14] Luca Cardelli. Basic polymorphic typechecking. **Sci. Comput. Program.**, 8(2):pages 147–172 (April 1987). `http://dx.doi.org/10.1016/0167-6423(87)90019-0`.

[15] Tom Schrijvers, Simon Peyton-Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In **Proceedings of the 14th ACM SIGPLAN international conference on Functional programming**, pages 341–352. ICFP '09, ACM, New York, NY, USA (2009). `http://doi.acm.org/10.1145/1596550.1596599`.

[16] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In **Proceedings of the 2002 ACM SIGPLAN workshop on Haskell**, pages 51–64. Haskell '02, ACM, New York, NY, USA (2002). `http://doi.acm.org/10.1145/581690.581695`.

[17] Martin Sulzmann and Meng Wang. GADTless programming in Haskell 98 (2006). `https://www.cs.ox.ac.uk/files/3060/gadtless.pdf`.

[18] Philip Wadler. The expression problem. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

[19] Wouter Swierstra. Data types à la carte. **Journal of Functional Programming**, 18(4):pages 423–436 (July 2008). http://dx.doi.org/10.1017/S0956796808006758.

Appendix A. Original Red-Black Tree Source Code

```
module RBT1 where

data Color = R | B
data Node a = E | N Color (Node a) a (Node a)
type Tree a = Node a

member :: Ord a => a -> Tree a -> Bool
member _ E  = False
member x (N _ l a r)
    | x < a  = member x l
    | x > a  = member x r
    | otherwise = True

insert :: Ord a => Tree a -> a -> Tree a
insert t v = blacken (insertInternal t v) where
    insertInternal n@(N c l a r) x
        | x < a = leftBalance  (N c (insertInternal l x) a r)
        | x > a = rightBalance (N c l a (insertInternal r x))
        | otherwise = n
    insertInternal E x = N R E x E
    blacken (N _ l x r) = N B l x r

leftBalance :: Node a -> Node a
leftBalance (N B (N R (N R a x b) y c) z d) =
    N R (N B a x b) y (N B c z d)
leftBalance (N B (N R a x (N R b y c)) z d) =
    N R (N B a x b) y (N B c z d)
leftBalance n = n

rightBalance :: Node a -> Node a
rightBalance (N B a x (N R b y (N R c z d))) =
    N R (N B a x b) y (N B c z d)
```

```
rightBalance (N B a x (N R (N R b y c) z d)) =
    N R (N B a x b) y (N B c z d)
rightBalance n = n
```

Appendix B. Red-Black Tree Source Code: Proving the Forth Invariant

```
{-# LANGUAGE TypeFamilies, DataKinds, GADTs #-}

module RBT2 where

data Color = R | B
data ColorSingleton (c::Color) where
    SR :: ColorSingleton R
    SB :: ColorSingleton B

data Nat = Zero | Succ Nat

type family   IncBlackHeight (c::Color) (bh::Nat) :: Nat
type instance IncBlackHeight R bh = bh
type instance IncBlackHeight B bh = Succ bh

data Node a (bh::Nat) where
    E :: Node a 'Zero
    N :: ColorSingleton c -> Node a bh -> a
        -> Node a bh -> Node a (IncBlackHeight c bh)

data Tree a where
    Root :: Node a bh -> Tree a

insert :: Ord a => Tree a -> a -> Tree a
insert (Root t) v = blacken (insertInternal t v) where
    insertInternal :: Ord a => Node a n -> a -> Node a n
    insertInternal n@(N c l a r) x
        | x < a = leftBalance  (N c (insertInternal l x) a r)
        | x > a = rightBalance (N c l a (insertInternal r x))
        | otherwise = n
    insertInternal E x = N SR E x E
    blacken (N _ l x r) = Root (N SB l x r)
```

```
leftBalance :: Node a bh -> Node a bh
leftBalance (N SB (N SR (N SR a x b) y c) z d) =
    N SR (N SB a x b) y (N SB c z d)
leftBalance (N SB (N SR a x (N SR b y c)) z d) =
    N SR (N SB a x b) y (N SB c z d)
leftBalance n = n

rightBalance :: Node a bh -> Node a bh
rightBalance (N SB a x (N SR b y (N SR c z d))) =
    N SR (N SB a x b) y (N SB c z d)
rightBalance (N SB a x (N SR (N SR b y c) z d)) =
    N SR (N SB a x b) y (N SB c z d)
rightBalance n = n
```

Appendix C. Red-Black Tree Source Code: Proving the Third Invariant

```
{-# LANGUAGE TypeFamilies, DataKinds, GADTs, RankNTypes #-}
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}

module RBT3 where

data Color = R | B
data ColorSingleton (c::Color) where
    SR :: ColorSingleton R
    SB :: ColorSingleton B

data Nat = Zero | Succ Nat

type family   IncBlackHeight (c::Color) (bh::Nat) :: Nat
type instance IncBlackHeight R bh = bh
type instance IncBlackHeight B bh = Succ bh

class ValidColors (parent::Color) (child1::Color) (child2::Color)
instance ValidColors R B B
instance ValidColors B c1 c2

data Node a (bh::Nat) (c::Color) where
```

```
    E :: Node a 'Zero B
    N :: ValidColors c c1 c1 => ColorSingleton c -> Node a bh c1
        -> a -> Node a bh c2 -> Node a (IncBlackHeight c bh) c

data Tree a where
    Root :: Node a bh B -> Tree a

data IntNode a (n::Nat) where
    IntNode :: ColorSingleton c -> Node a n c1 -> a
        -> Node a n c2 -> IntNode a (IncBlackHeight c n)

insert :: Ord a => Tree a -> a -> Tree a
insert (Root t) v = blacken (insertInternal t v) where
    insertInternal :: Ord a => Node a n c -> a -> IntNode a n
    insertInternal (N c l a r) x
        | x < a = leftBalance  c (insertInternal l x) a r
        | x > a = rightBalance c l a (insertInternal r x)
        | otherwise = IntNode c l a r
    insertInternal E x = IntNode SR E x E
    blacken (IntNode _ l x r) = Root (N SB l x r)

leftBalance :: ColorSingleton c -> IntNode a n -> a
    -> Node a n c' -> IntNode a (IncBlackHeight c n)
leftBalance SB (IntNode SR (N SR a x b) y c) z d =
    IntNode SR (N SB a x b) y (N SB c z d)
leftBalance SB (IntNode SR a x (N SR b y c)) z d =
    IntNode SR (N SB a x b) y (N SB c z d)
leftBalance c (IntNode SB a x b) z d = IntNode c (N SB a x b) z d
leftBalance c (IntNode SR a@(N SB _ _ _) x b@(N SB _ _ _)) z d =
    IntNode c (N SR a x b) z d
leftBalance c (IntNode SR E x E) z d = IntNode c (N SR E x E) z d
leftBalance _ (IntNode SR (N SR _ _ _) _ _) _ _ =
    error "can't happen"
leftBalance _ (IntNode SR _ _ (N SR _ _ _)) _ _ =
    error "can't happen"

rightBalance :: ColorSingleton c -> Node a n c' -> a
    -> IntNode a n -> IntNode a (IncBlackHeight c n)
rightBalance SB a x (IntNode SR b y (N SR c z d)) =
    IntNode SR (N SB a x b) y (N SB c z d)
rightBalance SB a x (IntNode SR (N SR b y c) z d) =
    IntNode SR (N SB a x b) y (N SB c z d)
rightBalance c a x (IntNode SB b y d) = IntNode c a x (N SB b y d)
rightBalance c a x (IntNode SR b@(N SB _ _ _) y d@(N SB _ _ _)) =
    IntNode c a x (N SR b y d)
rightBalance c a x (IntNode SR E y E) = IntNode c a x (N SR E y E)
rightBalance _ _ _ (IntNode SR (N SR _ _ _) _ _) =
    error "can't happen"
rightBalance _ _ _ (IntNode SR _ _ (N SR _ _ _)) =
    error "can't happen"
```

Error Reporting Parsers: a Monad Transformer Approach

by Matt Fenwick ⟨mfenwick100@gmail.com⟩
and Jay Vyas ⟨jayunit100@gmail.com⟩

Monad transformers are used to deal with combinations of computational effects, including backtracking, errors, and state, in a modular and composable way [1].

Real-world parsers require such computational effects in order to provide key features such as error reporting and context-sensitive results. This article will explore how monad transformers contribute to the design and implementation of parsers.

Parser Combinators

The goals of parsing are to: 1) decide whether a string is part of the language in question, and 2) build a meaningful representation of the structure of the parsed string. In addition, practical parsers must recognize faulty input and accurately report the location and cause of the problem. Thus, a parser must also decide what generates an error, what information is in the error and how error-reporting parsers compose.

Parser combinators are an excellent approach for building expressive, composable and declarative parsers. They also benefit from host-language integration, allowing them to snarf features such as type systems and test frameworks. There are many excellent papers on parser combinators, such as Wadler's classic paper [2] on non-deterministic parser combinators, Hutton's paper which covers factoring parsers into smaller pieces [3], and Leijen's Parsec paper [4]. The inspiration and many of the names used in this article are drawn from those references.

To get a feel for what parser combinators are and how they're used to build parsers, we'll first create a minimal parser combinator library. Next, we'll add monad transformers to the mix. We'll divide the library into four main pieces:

- the Parser datatype,
- a primitive parser,
- type class instances, and
- combinators.

Parser datatype

Parsers operate on token sequences, consuming tokens from the front of the sequence. We could model this with the function type:

```
newtype Parser t =
    Parser {getParser :: [t] -> [t]}
```

We also want parsers to return a value, if they succeed. To do this, we extend the return type to allow for a value, which also requires a second type parameter:

```
newtype Parser t a =
    Parser {getParser :: [t] -> ([t], a)}
```

However, this ignores the possibility of failure. To allow for failure, the return value needs to be wrapped in a Maybe:

```
newtype Parser t a =
    Parser {getParser :: [t] -> Maybe ([t], a)}
```

This gives us a simple, minimal definition of a parser. We've used `newtype` instead of `type` so that we can create type class instances for our parsers later on.

A primitive parser

The simplest parser is `item`; it consumes a single token if available and fails otherwise:

```
item :: Parser t t
item = Parser f
  where
    f [] = Nothing
    f (x:xs) = Just (xs, x)
```

This captures the basic concept that parsers are inherently sequential – they consume tokens left-to-right from a list.

Type class instances

For running multiple parsers in sequence, the combinators from the Monad and Applicative type classes are very useful; our next step is to implement them.

First is the Functor instance, which we need since it's a superclass of Applicative. It just maps a function over the result value, and relies on the Functor instances of Maybe and pairs:

```
instance Functor (Parser t) where
  fmap f (Parser p) = Parser (fmap (fmap f) . p)
```

The `<*>` operator of Applicative runs parsers sequentially and applies the result of the first to the result of the second; it has to make sure that the (possibly modified) output token stream from the first parser is passed to the second:

```
import Control.Applicative  (Applicative(..))

instance Applicative (Parser t) where
  pure x = Parser (\xs -> pure (xs, x))
  Parser p1 <*> Parser p2 = Parser p3
    where p3 xs = case (p1 xs) of
                       Nothing      -> Nothing;
                       Just (ys, f) -> fmap (fmap f) (p2 ys)
```

The Monad instance is similar to the Applicative instance, except that the actual result value of the first parser is used to generate the second parser; similarly to the Functor instance, this depends on instances of its underlying types:

```
instance Monad (Parser t) where
  return x = Parser (\ts -> return (ts, x))
  Parser p >>= f = Parser (\ts ->
                             p ts >>= \(ts', x) ->
                             getParser (f x) ts')
```

The Alternative type class expresses choice; the instance is is defined in terms of the Alternative instance of the underlying result type, `Maybe`.

```
import Control.Applicative  (Alternative(..))

instance Alternative (Parser t) where
  empty = Parser (const empty)
  Parser l <|> Parser r = Parser (\ts -> l ts <|> r ts)
```

Combinators

We can capture common patterns of parser construction in combinators in order to make it more convenient. The `check` combinator, analogous to `Control.Monad.mfilter`, is used to validate a parse result:

```
check :: (Monad f, Alternative f) => (a -> Bool) -> f a -> f a
check p m =
    m >>= \x -> if p x then return x else empty
```

Using `check`, we can build a convenient combinator to parse specific matching tokens:

```
literal :: Eq t => t -> Parser t t
literal x = check (== x) item
```

Examples

First, let's see `item` in action. It fails on empty lists, and succeeds consuming one token otherwise:

```
*Main> getParser item ""
Nothing

*Main> getParser item "abcde"
Just ("bcde",'a')
```

We can run parsers in sequence using the Applicative combinators. First, one parser is run, then the second is run, and the token position is threaded between the two. Note that both parsers must succeed in order for the combined parser to succeed:

```
*Main> getParser (item *> item) "abcde"
Just ("cde",'b')

*Main> getParser (fmap (,) item <*> item) "abcde"
Just ("cde",('a','b'))

*Main> getParser (item *> item) "a"
Nothing
```

Parsers built out of `literal` only succeed when the next token matches the specified one:

```
*Main> getParser (literal 'a') "abcde"
Just ("bcde",'a')

*Main> getParser (literal 'a') "bcde"
Nothing
```

Using the Alternative combinators, we can create parsers that succeed if any of their sub-parsers succeed. This is what allows parsers to backtrack, trying various alternatives if one fails:

```
*Main> getParser (literal 'a' <|> literal 'b') "abcde"
Just ("bcde",'a')

*Main> getParser (literal 'a' <|> literal 'b') "bacde"
Just ("acde",'b')

*Main> getParser (literal 'a' <|> literal 'b') "cdeab"
Nothing
```

Finally, let's parse arbitrarily deeply nested parentheses. This example demonstrates the use of `fmap` to apply a function to a parse result, `check` to make sure that `char` doesn't consume parentheses, `literal` to match specific tokens exactly, `many` to introduce repetition, `<|>` to express choice, and `*>` and `<*` to sequence parsers:

```
data Nesting
    = One Char
    | Many [Nesting]
  deriving (Show, Eq)

char :: Parser Char Nesting
char = fmap One $ check (not . flip elem "()") item

level :: Parser Char Nesting
level = literal '(' *> (fmap Many $ many element) <* literal ')'

element :: Parser Char Nesting
element = char <|> level

parseNest :: Parser Char [Nesting]
parseNest = many element
```

Here are some examples of this code in action:

```
*Main> getParser parseNest "(((())))))"
Just (")",[Many [Many [Many [Many []]]]])

*Main> getParser parseNest "(()abc(def)())zy"
Just ("",[Many [Many [],One 'a',One 'b',One 'c',
                Many [One 'd',One 'e',One 'f'],Many []],
          One 'z',
          One 'y'])

*Main> getParser parseNest "(()abc(def)()"
Just ("(()abc(def)()",[])
```

This concludes the implementation of a simple parser. In the rest of this article, we'll rework and augment these basic parser combinators by generalizing the implementations and extending the result types, while taking advantage of monad transformers to keep them modular.

Monad transformers

How can we use monad transformers to help us create parsers and parser combinators? The parser type we used earlier – `[t] -> Maybe ([t], a)` – can be built out of the StateT monad transformer (from the standard mtl [5] library) applied to Maybe. Here's the StateT implementation:

```
newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }
```

Substituting in [s] for s and Maybe for m in the right-hand side produces (note that the type variable m is no longer needed):

```
newtype StateT s a = StateT { runStateT :: [s] -> Maybe (a,[s]) }
```

This is identical to the Parser type given before modulo the order of the result tuple, so we can redefine the parser type as:

```
import Control.Monad.State       (StateT(..))

type Parser t a = StateT [t] Maybe a
```

The `item` parser is reimplemented using the combinators from MonadState – token lists are the 'state' that `item` must inspect and modify:

```
{-# LANGUAGE   FlexibleContexts  #-}
```

```
import Control.Monad.State       (MonadState (..))

item :: (MonadState [t] m, Alternative m) => m t
item =
    get >>= \xs -> case xs of
                        (t:ts) -> put ts *> pure t;
                        []     -> empty;
```

There are two major benefits to our parsers from using monad transformers:

- compositional implementation with semantic building blocks – the mtl library provides implementations for Maybe, Error, and State building blocks; using these, we can easily describe the semantics of our parsers without writing very much code, and
- type class instances – in addition to providing the building blocks, the mtl provides instances for common type classes, so we only have to implement parser-specific operations.

Introducing Woof: a Simple Lisp

The rest of this article will progressively adding features to a simple initial implementation on our way to creating an error-reporting parser. The simple language we'll use as a motivation for this task is Woof, a simple dialect of Lisp.

The language definition of Woof in pseudo-BNF is:

```
Woof         :=   Form(+)
Form         :=   Symbol  |  Special  |  Application
Symbol       :=   [a-zA-Z](+)
Special      :=   '{'  ( Define  |  Lambda )  '}'
Define       :=   'define'  Symbol  Form
Lambda       :=   'lambda'  '{'  Symbol(*)  '}'  Form(+)
Application  :=   '('  Form(+)  ')'
Whitespace   :=   \s+
Comment      :=   ';'  (not '\n')(*)
```

There is an additional rule that whitespace and comments may appear in any amount before or after tokens. Our tokens are:

- `{`
- `}`
- `(`
- `)`
- `Symbol`

Example 1: Recognition and Tree-Building

Preliminaries

Our first Woof parser will only be responsible for determining whether an input text conforms to the language definition and building an Abstract Syntax Tree (AST) representing the structure of the parsed input. Here's the AST definition we'll use:

```
data AST
    = ASymbol String
    | ALambda [String] [AST]
    | ADefine String AST
    | AApp    AST  [AST]
  deriving (Show, Eq)
```

We'll also want a function for running our parsers, using the type given in the prevous section:

```
runParser :: Parser t a -> [t] -> Maybe (a, [t])
runParser = runStateT
```

Token parsers

Our first parsers are for the most basic syntactic elements: tokens. These include the four braces, symbols, strings, numbers, whitespace, and comments. To recognize braces, we use the `item` and `check` combinators to build a new combinator, `satisfy`, that checks whether the next character meets a condition:

```
satisfy :: (MonadState [t] m, Alternative m) => (t -> Bool) -> m t
satisfy = flip check item
```

Because we used MonadState's combinators to build `item`, `literal` has a slightly different type; we also take advantage of `satisfy` to simplify its implementation:

```
{-# LANGUAGE NoMonomorphismRestriction #-}

literal :: (Eq t, MonadState [t] m, Alternative m) => t -> m t
literal c = satisfy ((==) c)
```

Now we're ready to build parsers for the four bracket tokens. First, a simple bracket parser:

```
opencurly = literal '{'
```

It accepts open curly brackets and rejects everything else:

```
*Main> map (runParser  opencurly) ["{abc", "}abc", "(abc", "abc"]
[Just ('{', "abc"), Nothing, Nothing, Nothing]
```

Recall, however, that our specification said that comments and whitespace can occur anywhere before or after tokens – how should this be handled?

First, we need to define the whitespace and comment patterns – we'll also rename the `some` and `many` functions from Control.Alternative to more clearly indicate their semantics:

```
many0 = many
many1 = some

whitespace = many1 $ satisfy (flip elem " \n\t\r\f")
comment = pure (:) <*> literal ';' <*> many0 (not1 $ literal '\n')
```

We want to throw away comments and whitespace, and this process must occur after every token. We can do this by using a combinator which first parses a token and then throws away junk:

```
junk = many0 (whitespace <|> comment)

tok p = p <* junk
```

The comment parser uses a new combinator, `not1`, built out of a type class, Switch, modeling computations which can be switched from successful to failing and vice versa:

```
class Switch f where
  switch :: f a -> f ()

instance Switch Maybe where
  switch (Just _) = Nothing
  switch Nothing  = Just ()

instance (Functor m, Switch m) => Switch (StateT s m) where
  switch (StateT f) = StateT g
    where
      g s = fmap (const ((), s)) . switch $ f s

not1 :: (MonadState [t] m, Alternative m, Switch m) => m a -> m t
not1 p = switch p *> item
```

With `tok` in hand, we can fix `opencurly`:

```
opencurly = tok $ literal '{'

*Main> runParser opencurly "{;abc\ndef"
Just ('{', "def")

*Main> runParser opencurly "(def"
Nothing
```

Excellent! The `opencurly` parser matches an open curly brace, failing otherwise, and discards trailing whitespace and comments.

It's straightforward to define parsers for the other three brace tokens:

```
closecurly = tok $ literal '}'
openparen  = tok $ literal '('
closeparen = tok $ literal ')'
```

The parser for our last token, Symbol, is more complicated, as any number of alphabetical characters is valid. We'll use `satisfy` and a predicate to check that a character is valid, and `many1` to match one or more valid characters:

```
symbol = tok $ many1 char
  where char = satisfy (flip elem (['a' .. 'z'] ++ ['A' .. 'Z']))

*Main> runParser symbol "abc123"
Just ("abc","123")

*Main> runParser symbol "abc;sdfasdfsa\n  123"
Just ("abc","123")

*Main> runParser symbol "123;sdfasdfsa\n  abc"
Nothing
```

Note that we again use `tok` to throw away trailing junk.

Syntactic structures

Whereas our token parsers dealt with the syntactic primitives of the Woof grammar, the remaining parsers will implement grammar rules that combine the tokens into syntactic structures.

The first combining form is function application. The rule says that it is delimited by matching parentheses, in between which must appear one form as the operator, followed by any number of forms as the arguments to which the operator is applied. We can say that like so:

```
application =
    openparen    *>
    pure AApp   <*>
    form        <*>
    many0 form  <*
    closeparen
```

Note that all the parser sequencing is done using Applicative combinators – we don't need to use the monadic ones. Also, we used `pure` to inject a value – the function `AApp` – into the parser. `pure` has no effect on the tokens and always succeeds:

```
*Main> runParser (pure 3) ""
Just (3,"")

*Main> runParser (pure 3) "abc"
Just (3,"abc")
```

Next, we move on to the special forms. All special forms are delimited with curly braces, and lambda and define are our special forms, so we have:

```
special = opencurly *> (define <|> lambda) <* closecurly
```

A straightforward translation of define from the grammar produces:

```
define =
    check (== "define") symbol   *>
    pure ADefine                <*>
    symbol                      <*>
    form
```

On to Lambda! Not only does Lambda have additional syntax with its parameter list, but we also need to ensure that the parameter names are unique. We can do that using the `check` combinator again:

```
lambda =
    check (== "lambda") symbol      *>
    opencurly                       *>
    pure ALambda                   <*>
    check distinct (many0 symbol)  <*>
    (closecurly                     *>
     many1 form)
  where
    distinct names = length names == length (nub names)
```

If the symbols are distinct, the parameter list subparser will succeed, whereas if there's a repeated symbol, it will fail.

Finishing touches

All we have left to do are `form` (which we've already used recursively to build the previous parsers) and `woof`.

A form is either a symbol, application, or a special form. We could just write:

```
form = symbol <|> application <|> special
```

except that the types don't match – `symbol`'s type parameter is a String, whereas the other two have ASTs. We fix that by mapping the `ASymbol` function over `symbol`:

```
form = fmap ASymbol symbol <|> application <|> special
```

The final parser, `woof`, needs to not only parse all the forms, but also make sure that the entire input is consumed. The end of input check is done with the `end` parser:

```
endCheck = switch item
```

We also need to discard leading comments and whitespace, so the final Woof parser is:

```
woof = junk *> many0 form <* endCheck
```

Examples

Let's look at a few examples of our parser in action. Here are a couple of successful examples:

```
*Main> runParser woof "; \n   {lambda {x y z} a b (c b a)}end"
Just ([ALambda ["x","y","z"]
               [ASymbol "a",
                ASymbol "b",
                AApp (ASymbol "c")
                     [ASymbol "b",
                      ASymbol "a"]],
       ASymbol "end"],
      "")

*Main> runParser woof "a b c {define q (f x)}"
Just ([ASymbol "a",
       ASymbol "b",
       ASymbol "c",
       ADefine "q" (AApp (ASymbol "f") [ASymbol "x"])],"")
```

And a failing one:

```
*Main> runParser woof "a b c {define q (f x)},"
Nothing
```

Note that while the first two parses consume the entire input, while the last one fails because it doesn't recognize the trailing comma.

Example 2: error-reporting

While our first parser worked great on valid input, it wasn't helpful when the input was malformed. When the parser finds bad input, we'd like it to produce an informative error, indicating what and where the problem was.

Let's start with an informal spec of different cases of malformed input, along with the error messages that should be reported:

- application: missing operator:
  `()`
- application: missing close parenthesis:
  `(a b`
- define: missing symbol:
  `{define (a b c)}`
- define: missing form:
  `{define a}`
- lambda: missing parameter list:
  `{lambda (a b c)}`
- lambda: duplicate parameter names:
  `{lambda {a b a} (c d)}`
- lambda: missing parameter list close curly:
  `{lambda {a b (c d)}`
- lambda: missing body form:
  `{lambda {a b}}`
- special form: unable to parse:

  `{defin x y}`
- special form: missing close curly:
  `{define x y`
- woof: unparsed input:
  `a,b`

To allow error reporting in our parsers, we first need to extend our monad stack, adding in a layer for errors. We'll use a second monad transformer type class – MonadError, replace the Maybe layer with its corresponding transformer datatype MaybeT, and add in an Either layer on the bottom of the stack. Our parser stack and `runParser` function are now:

```
type Parse e t a = StateT [t] (MaybeT (Either e)) a

runParser :: Parse e t a -> [t] -> Either e (Maybe (a, [t]))
runParser p xs = runMaybeT (runStateT p xs)
```

While our previous parsers had two different results – Nothing and Just – our new parsers have three possible results: success (Right Just), failure (Right Nothing), and error (Left). A parse failure means that a parser wasn't able to match the input, but nothing bad happened; failures can be recovered from if there is an alternative parse that succeeds. However, an error means that an unrecoverable problem has been found and that parsing should immediately stop; the parser will not be able to recover even if there are alternatives, since backtracking will not be performed. Note that successful and failing parses are wrapped in an additional `Right` constructor:

```
*Main> runParser item "abc"
Right (Just ('a',"bc"))

*Main> runParser item ""
Right Nothing

*Main> runParser (throwError "oops!") ""
Left "oops!"
```

Although the `<|>` combinator can recover from failures, it can't recover from errors. Compare:

```
*Main> runParser (satisfy (== 'a') <|> satisfy (== 'b')) "babc"
Right (Just ('b',"abc"))

*Main> runParser (throwError "no!" <|> satisfy (== 'b')) "babc"
Left "no!"
```

We'll take advantage of this backtracking restriction to produce accurate error messages.

Type Class Preliminaries

Unfortunately, the standard MonadError instance for the `Either` datatype does not fit our needs – it requires an `Error` constraint that we do not want. We'll implement our own MonadError type class. We'll need an instance for each member of our stack – Either, StateT, and MaybeT:

```
{-# LANGUAGE FlexibleContexts, MultiParamTypeClasses,
    FlexibleInstances, TypeFamilies #-}

class Monad m => MonadError m where
  type Error m :: *
  throwError :: Error m -> m a
  catchError :: m a -> (Error m -> m a) -> m a

instance MonadError (Either e) where
  type Error (Either e) = e
  throwError               =  Left
  catchError  (Right x) _  =  Right x
  catchError  (Left e)  f  =  f e

instance MonadError m => MonadError (StateT s m) where
  type Error (StateT s m) = Error m
  throwError      =  lift . throwError
  catchError m f  =  StateT g
    where
      g s = catchError (runStateT m s)
                       (\e -> runStateT (f e) s)

instance MonadError m => MonadError (MaybeT m) where
  type Error (MaybeT m) = Error m
  throwError      =  lift . throwError
  catchError m f  =  MaybeT $ catchError (runMaybeT m)
                                         (runMaybeT . f)
```

Note how the StateT and MaybeT instances don't really do anything with the errors; they just pass the error on through to the next level. That's why they both require that the next lower level supports MonadError. This code uses type families to state what the error type is; explaining type families in more detail is out of scope for this article, but roughly, given some MonadError `m`, you can write down the type of error that monad supports by saying `Error m`.

We'll also need an instance of Switch for MaybeT:

```
instance Functor m => Switch (MaybeT m) where
  switch (MaybeT m) = MaybeT (fmap switch m)
```

Error messages

We must modify the remaining parsers to report meaningful error messages; let's define the messages that will be reported in each case:

```
eAppOper    =  "application: missing operator"
eAppClose   =  "application: missing close parenthesis"
eDefSym     =  "define: missing symbol"
eDefForm    =  "define: missing form"
eLamParam   =  "lambda: missing parameter list"
eLamDupe    =  "lambda: duplicate parameter names"
eLamPClose  =  "lambda: missing parameter list close curly"
eLamBody    =  "lambda: missing body form"
eSpecClose  =  "special form: missing close curly"
eSpecial    =  "special form: unable to parse"
eWoof       =  "woof: unparsed input"
```

The MonadError type class provides two useful combinators: `throwError` for generating errors and `catchError` for dealing with them. We can ignore the second because we don't need to recover from errors, just report them, but when should an error be generated and what data should it include? `throwError` will unconditionally throw an error, but often we will only want to conditionally throw an error. Thus, we introduce the `commit` combinator:

```
commit :: (MonadError m, Alternative m) => Error m -> m a -> m a
commit err p = p <|> throwError err
```

This combinator takes two arguments: an error value and a monadic computation. It first tries to run the computation; if the computation fails, it generates an error. The definition of 'failure' is provided by the Alternative instance of `m`. The reason for its name is that it restricts backtracking and alternatives; it "commits" the parser to a do-or-die situation – either it successfully parses the input, or the parsing stops. It's based on the `nofail` combinator of [6] and also on [7].

What data do we want to include in our errors? A String describing the error and a token reporting the position will do:

```
type Error = String
```

We'll first add error-reporting to `application`:

```
application =
    openparen                       >>
    commit eAppOper form            >>= \op ->
    many0 form                      >>= \args ->
    commit eAppClose closeparen     >>
    return (AApp op args)
```

This parser works by first running the open-parenthesis parser; then, if either `form` or close-parenthesis fails, an appropriate error is generated with a message describing the problem. However, if the open parenthesis is not successfully parsed, it's a failure but not an error. Examples:

```
*Main> runParser application "oops"
Right Nothing

*Main> runParser application "(a b c)"
Right (Just (AApp (ASymbol "a") [ASymbol "b",ASymbol "c"],""))

*Main> runParser application "\n()"
Left "application: missing operator"

*Main> runParser application "(a b"
Left "application: missing close parenthesis"
```

The `define` and `lambda` parsers are upgraded similarly:

```
define =
    check (== "define") symbol    *>
    pure ADefine                 <*>
    commit eDefSym symbol        <*>
    commit eDefForm form

lambda =
    check (== "lambda") symbol        >>
    commit eLamParam opencurly        >>
    many0 symbol                      >>= \params ->
    (if distinct params
        then return ()
        else throwError eLamDupe)     >>
    commit eLamPClose closecurly     >>
    commit eLamBody (many1 form)     >>= \bodies ->
    return (ALambda params bodies)
  where
    distinct names = length names == length (nub names)
```

Note that `commit` isn't used until we can be sure that we're in the right rule: if, in the `define` parser, we committed to parsing the "define" keyword, we wouldn't be able to backtrack and try the `lambda` parser if that failed. Thus, if errors are treated as unrecoverable, it's important not to place a `commit` where backtracking might be necessary to parse a valid alternative. In our example, however, once we see an open curly brace and a define, we're sure that we're in the `define` rule; allowing useless backtracking would only destroy our ability to report errors cleanly and accurately.

Error messages: finishing touches

To complete the error-reporting parser, we need to change a couple more details. First, we want an error reported when an open curly brace is found, but no special form can be parsed. We extend the `special` parser to report an error if it matches an open brace but not define or lambda:

```
special =
    opencurly  *>  commit eSpecial (define <|> lambda)  <*  commit eSpecClose
```

The `form` parser doesn't have to be changed:

```
form = fmap ASymbol symbol <|> application <|> special
```

When we can no longer parse any forms, instead of failing, we'd like to report an error if there's any input left, and otherwise succeed:

```
woof = junk *> many0 form <* commit eWoof endCheck
```

Error messages: examples

While correct input is still parsed successfully:

```
*Main> runParser woof "{define fgh {lambda {f x} (f x)}}"
Right (Just ([ADefine "fgh" (ALambda ["f","x"]
                                     [AApp (ASymbol "f")
                                           [ASymbol "x"]])],
       []))
```

Incorrect input results in an error. For each of the items in our error spec, we can show that our parser correctly reports an error:

```
*Main> runParser woof "(a b"
Left "application: missing close parenthesis"

*Main> runParser woof "{define (a b c)}"
Left "define: missing symbol"
```

Example 3: reporting position

Knowing that a close-parenthesis is missing somewhere deep inside a nested mess of forms isn't very helpful – you also want to know where it happened. Let's add that to our parser.

Parser type

Once again, we extend the monad transformer stack with another layer – a second state layer, in which the line and column position will be kept as simple Ints. The error type is also extended to include a position as well as a message:

```
type Pos = (Int, Int)
type Err = (String, Pos)

type Parser t a = StateT [t] (StateT Pos (MaybeT (Either Err))) a
```

`runParser` is extended to support the new stack; it assumes we always start parsing at line 1, column 1:

```
runParser :: Parser t a -> [t] -> Either Err (Maybe ((a, [t]), Pos))
runParser p ts = runMaybeT $ runStateT (runStateT p ts) (1, 1)
```

Now that we have two separate state layers, we need to be sure not to mix them up (fortunately, because the types of the states are different, Haskell's type system will helpfully complain if I mess up). The `lift` method of the `MonadTrans` type class allows us to access the position state:

```
getState :: Parser Char Pos
getState = lift get

putState :: Pos -> Parser Char ()
putState = lift . put
```

```
updateState :: (Pos -> Pos) -> Parser Char ()
updateState f = getState >>= (putState . f)
```

We also need to update `item`, such that when it consumes a character, it updates the line/column position. We'll do this by splitting it into two parts, the part that consumes a single character, and the part that updates the position:

```
basic_item :: (MonadState [t] m, Alternative m) => m t
basic_item =
    get >>= \xs -> case xs of
                        (t:ts) -> put ts *> pure t;
                        []     -> empty;

item :: Parser Char Char
item = basic_item >>= \x -> updateState (f x) >> pure x
  where f '\n' (ln, c) = (ln + 1, 1)
        f _    (ln, c) = (ln, c + 1)
```

For simplicity, only '
n' characters will count as newlines; every other character, including tabs, will count as a single column on the same line.

Position reporting

To actually report the position when an error is detected, we introduce a new combinator, which I'll call `cut`:

```
cut :: String -> Parser Char a -> Parser Char a
cut message parser =
    getState >>= \p ->
    commit (message, p) parser
```

It checks the current position, runs a parser, and if the parser fails, reports an error consiting of a message and the position.

To take advantage of this combinator, we essentially replace every use of `commit` with `cut`:

```
application =
    openparen                   >>
    cut eAppOper form           >>= \op ->
    many0 form                  >>= \args ->
    cut eAppClose closeparen    >>
    return (AApp op args)

define =
    check (== "define") symbol    *>
    pure ADefine                 <*>
    cut eDefSym symbol           <*>
    cut eDefForm form

lambda =
    check (== "lambda") symbol     >>
    cut eLamParam opencurly        >>
    many0 symbol                   >>= \params ->
    (if distinct params
        then return ()
        else cut eLamDupe empty)      >>
    cut eLamPClose closecurly         >>
    cut eLamBody (many1 form)         >>= \bodies ->
    return (ALambda params bodies)
  where
    distinct names = length names == length (nub names)

special =
    opencurly  *>  cut eSpecial (define <|> lambda)  <*  cut eSpecClose closecurly

form = fmap ASymbol symbol <|> application <|> special

woof = junk *> many0 form <* cut eWoof end
```

Now the parsers report useful errors:

```
*Main> runParser woof " abc 123 "
Left ("woof: unparsed input",(1,6))

*Main> runParser woof " (((abc ())) "
Left ("application: missing operator",(1,10))

*Main> runParser woof "{define const \n  {lambda {x x} \n   x}"
Left ("lambda: duplicate parameter names",(2,15))
```

Stack Order

While the contents of a transformer stack is important, the order of the layers is important as well [8]. Let's revisit our first Woof parser – what if our stack had been Maybe/State instead of State/Maybe?

```
State/Maybe:   s -> Maybe (a, s)

Maybe/State:   s -> (Maybe a, s)
```

given:

```
import Control.Monad.Trans.Maybe (MaybeT(..))
import Data.Functor.Identity     (Identity(..))

type Parser' t a = MaybeT (StateT [t] Identity) a

runParser' :: Parser' t a -> [t] -> (Maybe a, [t])
runParser' p xs = runIdentity $ runStateT (runMaybeT p) xs
```

We can compare it to the earlier parser:

```
*Main> runParser' (literal 'a') "aqrs"
(Just 'a',"qrs")

*Main> runParser' (literal 'a') "qrs"
(Nothing,"rs")
```

Whereas:

```
*Main> runParser (literal 'a') "qrs"
Nothing

*Main> runParser (literal 'a') "aqrs"
Just ('a',"qrs")
```

The difference is that Maybe/State always returns a modified state, regardless of the success of the computation, whereas State/Maybe only returns a modified state if the computation is successful. This means that backtracking doesn't work right with Maybe/State – tokens are consumed even when the parsers don't match, as the example shows.

On the other hand, the order of Error and Maybe relative to each other isn't important. We can show this by losslessly converting values from one stack to the other and back:

```
type Stack1 e a = Either e (Maybe a)
type Stack2 e a = Maybe (Either e a)

forward :: Stack1 e a -> Stack2 e a
forward (Left e)          =  Just $ Left e
forward (Right Nothing)   =  Nothing
forward (Right (Just x))  =  Just $ Right x

backward :: Stack2 e a -> Stack1 e a
backward Nothing           =  Right Nothing
backward (Just (Left e))   =  Left e
backward (Just (Right x))  =  Right $ Just x

identityForward :: Stack1 e a -> Stack1 e a
identityForward = backward . forward

identityBackward :: Stack2 e a -> Stack2 e a
identityBackward = forward . backward
```

No partial functions were used. Now we can use these functions to convert between the two stacks. Note the final result is the same as the input:

```
vals = [Left "an error", Right Nothing, Right $ Just "success"]

*Main> map forward vals
[Just (Left "an error"),Nothing,Just (Right "success")]

*Main> map identityForward vals
[Left "an error",Right Nothing,Right (Just "success")]
```

Another important point is that our parsers don't specify which order – Maybe/State or State/Maybe – they prefer, and will work with either. This means that the semantics are ultimately determined by the actual monad stack we use.

Monad transformer motivation

Let's take a quick look at what we've done from a different perspective that may help motivate why monad transformers were used.

In the first parser, we could parse valid input, but invalid input left us with absolutely no idea what the problem was or where:

```
*Main> runParser woof " {define const {lambda {x x} x}} "
Nothing
```

The second parser reported what the problem was, but not where:

```
*Main> runParser woof " {define const {lambda {x x} x}} "
Left "lambda: duplicate parameter names"
```

And the third parser was able to report both what and where the problem occurred:

```
*Main> runParser woof " {define const {lambda {x x} x}} "
Left ("lambda: duplicate parameter names",(1,25))
```

Thus, a key to the usefulness of the last parser is that it supports monadic effects – backtracking, error reporting, and state. The parsers which supported fewer effects were not as useful.

This emphasizes some major advantages of using monad transformers:

- it's easy to create stacks supporting multiple effects
- type class instances are taken care of by the library, so stacks already support many useful combinators
- it's easy to change the composition and order of stacks

We could see these advantages when many of the parsers didn't have to change at all from one version to the next (although some of their types did change).

Parsec comparison

A quick word about Parsec [4], a popular and battle-tested parser combinator library for Haskell:

The approach I've described differs from Parsec's approach with respect to backtracking and error-reporting. By default, Parsec generates LL(1) parsers – this means it uses predictive parsing with only a single token of lookahead. This is useful for both efficiency and better error reporting. However, often a single token is not enough to unambiguously determine what rule should be tried; for these cases, one must use its `try` combinator. It is up to the programmer to determine when it's necessary to add `try` to a parser implementation.

On the other hand, my approach by default uses unlimited lookahead; it also doesn't automatically generate meaningful errors. It is up to the programmer to generate useful errors using the `throwError` and `commit` combinators.

Conclusion

Monad transformers provide an elegant solution to error reporting in parser combinators. Better yet, they are extensible – we could go back and add more features to our parsers using additional transformers, perhaps extending our parser to save partial results, so that if there's an error, it reports the progress that it had made before the error. We could do that using a Writer transformer. Or we check to make sure that all variables are in scope when they're used using a Reader transformer. We could even capture whitespace and comment tokens, instead of just throwing them away, in case they contained valuable information. And the best part of it is, most of the parsers we already wrote would continue to work fine without needing any modifications. That is the power of monad transformers!

References

[1] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In **Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages**, pages 333–343. ACM (1995).

[2] Philip Wadler. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In Jean-Pierre Jouannaud (editor), **Functional Programming Languages and Computer Architecture**, volume 201 of **Lecture Notes in Computer Science**, pages 113–128. Springer Berlin Heidelberg (1985). `http://dx.doi.org/10.1007/3-540-15975-4_33`.

[3] Graham Hutton and Erik Meijer. Monadic parser combinators (1996).

[4] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world (2001).

[5] Hackage – mtl. `http://hackage.haskell.org/package/mtl-2.1.2`.

[6] Graham Hutton. Higher-order functions for parsing. **Journal of Functional Programming**, 2(3):pages 323–343 (1992).

[7] Kota Mizushima, Atusi Maeda, and Yoshinori Yamaguchi. Packrat parsers can handle practical grammars in mostly constant space. In **Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering**, pages 29–36. PASTE '10, ACM, New York, NY, USA (2010). `http://doi.acm.org/10.1145/1806672.1806679`.

[8] Jeff Newbern. All About Monads. `http://monads.haskell.cz/html/stacking.html`.

Two Monoids for Approximating $\mathbb{NP}$-Complete Problems

by Mike Izbicki ⟨mike@izbicki.me⟩

As a TA, I was confronted with a real life instance of the $\mathbb{NP}$*-complete* SCHEDULING *problem. To solve the problem, I turned to the classic Least Processing Time First (LPTF) approximation algorithm. In this article, we'll see that because LPTF is a monoid homomorphism, we can implement it using HLearn's* `HomTrainer` *type class. This gives us parallel and online versions of LPTF "for free." We'll also be able to use these same techniques to solve a related problem called* BINPACKING*. Hopefully, at the end of the article you'll understand when the* `HomTrainer` *class might be a useful tool, and how to use and build your own instances.*

Framing The Problem

I enjoy TAing the introduction to C++ course at my university. Teaching pointer arithmetic can be immensely frustrating, but it's worth it to see the students when it all finally clicks. Teaching is even better when it causes you to stumble onto an interesting problem. Oddly enough, I found this cool Haskell problem because of my C++ teaching assistanceship.

The professor wanted to assign a group project, and I had to pick the groups. There had to be exactly five groups, and the groups needed to be as fair as possible. In other words, I was supposed to evenly distribute the best and worst students.

After a little thought, I realized this was an instance of the $\mathbb{NP}$-complete SCHEDULING problem in disguise. This problem was first formulated in the context of concurrent computing. In the textbook example of SCHEDULING, we are given p processors and n tasks. Each task t_i has some associated time it will take to complete it. The goal is to assign the tasks to processors so as to complete the tasks as quickly as possible.

The SCHEDULING problem is shown graphically in Figure 1. Processors are drawn as bins, and tasks are drawn as green blocks inside the bins. The height of each task represents the length of time required to process it. Our goal is to minimize the processing time given a fixed number of processors.

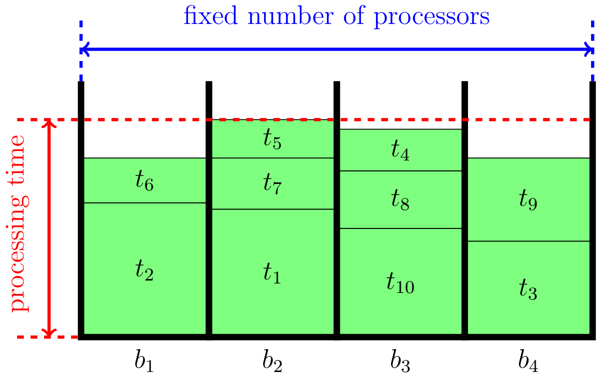

Figure 1: The SCHEDULING problem

What does this have to do with the problem my professor gave me? Well, we can think of the number of groups as the number of processors, each student is a task, and the student's current grade is the task's processing time. Then, the problem is to find a way to divvy up the students so that the sum of grades for the "best" group is as small as possible. We'll see some code for solving this problem in a bit.

There is another closely related problem called BINPACKING that is easily confused with SCHEDULING. In BINPACKING, instead of fixing the number of bins and minimizing the amount in the bins, we fix the bin size and minimize the total number of bins used. Compare Figure 2 below and Figure 1 above to see the difference. The BINPACKING problem was originally studied by shipping companies, although like SHEDULING it occurs in many domains.

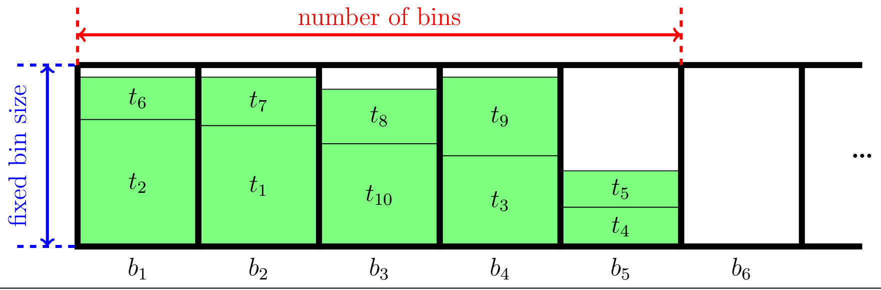

Figure 2: The BINPACKING problem

Both Scheduling and BinPacking are ℕℙ-complete. Therefore, I had no chance of creating an optimal grouping for my professor—the class had 100 students, and 2^{100} is a *big* number! So I turned to approximation algorithms. One popular approximation for Scheduling is called Longest Processing Time First (LPTF). When analyzing these approximation algorithms, it is customary to compare the quality of their result with that of the theoretical optimal result. In this case, we denote the total processing time of the schedule returned by LPTF as *LPTF*, and the processing time of the optimal solution as *OPT*. It can be shown that the following bound holds:

$$LPTF \leq \left(\frac{4}{3} - \frac{1}{3n}\right) OPT$$

This bound was proven in the late 1960's, but the original paper remains quite readable today [1]. By making this small sacrifice in accuracy, we get an algorithm that runs in time $\Theta(n \log n)$ instead of $\Theta(2^n)$. Much better! In the rest of this article, we'll take a detailed look at a Haskell implementation of the LPTF algorithm, and then briefly use similar techniques to solve the BinPacking problem.

The Scheduling HomTrainer

When implementing an algorithm in Haskell, you always start with the type signature. LPTF takes a collection of tasks and produces a schedule, so it's type might look something like:

```
:: [Task] → Schedule
```

Anytime I see see a function of this form, I ask myself, "Can it be implemented using HLearn's `HomTrainer` type class?" `HomTrainers` are useful because the compiler automatically derives online and parallel algorithms for all instances. In this section, we'll get a big picture view of how this class will help us solve the Scheduling problem. We start by looking at the format of a `HomTrainer` instance as shown graphically in Figure 3 below.

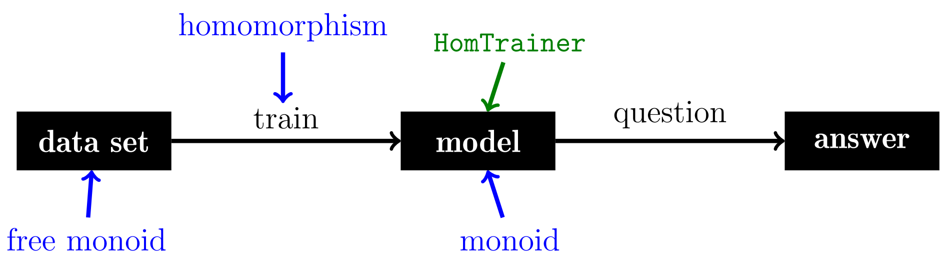

Figure 3: Basic requirements of the HomTrainer type class

There's a lot going on in Figure 3, but we'll look at things piece-by-piece. The black boxes represent data types and the black arrows represent functions. In our case, the data set is the collection of tasks we want scheduled and the model is the schedule. The train function is the LPTF algorithm, which generates a model from the data points. Finally, our model is only useful if we can ask it questions. In this case, we might want to ask, "Which processor is assigned to task t_{10}?" or "What are all the tasks assigned to processor 2?"

The blue arrows in the diagram impose some constraints on the data types and functions. These requirements are a little trickier: they specify that our training algorithm must be a **monoid homomorphism** from the **free monoid**. These are scary sounding words, but they're pretty simple once you're familiar with them. We'll define them and see some examples.

In Haskell, the `Monoid` type class is defined as having an identity called `mempty` and a binary operation called `mappend`:

```
class Monoid m where
    mempty  :: m
    mappend :: m → m → m
```

Sometimes, we use the infix operation `(⋄) = mappend` to make our code easier to read. All instances must obey the identity and associativity laws:

```
mempty ⋄ m = m ⋄ mempty = m
(m1 ⋄ m2) ⋄ m3 = m1 ⋄ (m2 ⋄ m3)
```

Lists are one of the simplest examples of monoids. Their identity element is the empty list, and their binary operation is concatenation:

```
instance Monoid [a] where
    mempty = []
    mappend = ++
```

Lists are an example of free monoids because they can be generated by any underlying type. For the `HomTrainer`, when we say that our data set must be a free monoid, all we mean is that it is a collection of some data points. For the Scheduling problem, it is just a list of tasks.

A homomorphism from the free monoid is a function that "preserves the free monoid's structure." More formally, if the function is called `train`, then it obeys the law that for all `xs` and `ys` of type `[a]`:

```
train (xs ++ ys) = (train xs) ⋄ (train ys)
```

The LPTF algorithm turns out to have this property. Figure 4 shows this in picture form with a commutative diagram. This means that it doesn't matter whether we take the **orange** path (first train schedules from our data sets, then combine the schedules with `mappend`) or the **purple** path (first concatenate our data sets, then train a schedule on the result). Either way, we get the exact same answer.

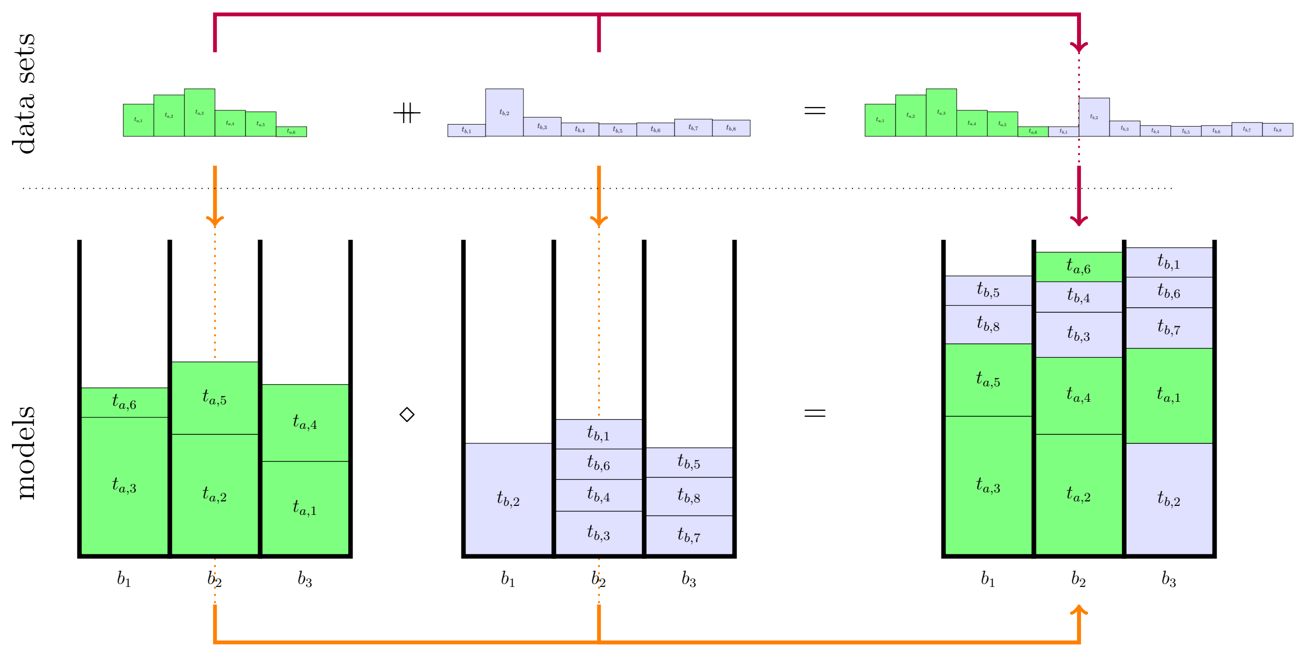

Figure 4: The LPTF is a monoid homomorphism because this diagram commutes

Now that we understand what `HomTrainer` is, we can look at what it gives us. Most importantly, it gives us a simple interface for interacting with our models. This interface is shown in Figure 5. In the class, we associate a specific `Datapoint` type to our model and get four functions for training functions. The most important training function is the batch trainer, called `train`. This is the homomorphism that converts the data set into a model. In our case, it will be the LPTF algorithm. The second most important function is the online trainer `add1dp`. This function takes a model and a datapoint as input, and "adds" the data point to the model. Developing new online functions is an important research area in approximation algorithms. As we will see later, the compiler generates these two functions automatically for all instances of `HomTrainer`.

Finally, HLearn comes with a higher order function for making all batch trainers run efficiently on multiple cores. The function

```
parallel :: (...) ⇒
    (container datapoint → model) → (container datapoint → model)
```

takes a batch trainer as input and returns a parallelized one as output. In the next section, we'll see an example of its use in practice.

```
class (Monoid model) ⇒ HomTrainer model where
    type Datapoint model

    -- The singleton trainer
    train1dp :: Datapoint model → model

    -- The batch trainer
    train :: (Functor container, Foldable container) ⇒
        container (Datapoint model) → model

    -- The online trainer
    add1dp :: model → Datapoint model → model

    -- The online batch trainer
    addBatch :: (Functor container, Foldable container) ⇒
        model → container (Datapoint model) → model
```

Figure 5: The `HomTrainer` type class

Using the Scheduling HomTrainer

Before we look at implementing a `HomTrainer` to solve Scheduling, we'll take a look at how it's used. In particular, we'll look at the Haskell code I used to solve the problem of grouping my students. In order to run the code, you'll need to download the latest `HLearn-approximation` library:

```
cabal install HLearn-approximation-1.0.0
```

Let's begin by doing some experiments in GHCi to get ourselves oriented. The `Scheduling` type is our model, and here we ask GHCi for it's kind signature:

```
ghci> import HLearn.NPHard.Scheduling
ghci> :kind Scheduling
Scheduling :: Nat → ∗ → ∗
```

`Scheduling` takes two type parameters. The first is a type-level natural number that specifies the number of processors in our schedule. In HLearn, any parameters to our training functions must be specified in the model's type signature. In this case, the type-level numbers require the `DataKinds` extension to be enabled. The second parameter to `Scheduling` is the type of the task we are trying to schedule.

Next, let's find out about `Scheduling`'s `HomTrainer` instance:

```
ghci> :info Scheduling
...
```

```
instance (Norm a,...) ⇒ HomTrainer (Scheduling n a) where
...
```

We have a constraint on our task parameter specifying that it must be an instance of `Norm`. What does that mean? In mathematics, a type has a norm if it has a "size" of some sort. In HLearn, the `Norm` type class is defined in two parts. First we associate a specific number type with our model using the `HasRing` type class. Then, the `Norm` type class defines a function `magnitude` that converts a model into the number type.

```
class (Num (Ring m)) ⇒ HasRing m where
    type Ring m

class (HasRing m, Ord (Ring m)) ⇒ Norm m where
    magnitude :: m → Ring m
```

Usually the associated ring will be a `Double`. But we might make it a `Rational` if we need more accuracy or an `Int` if we don't want fractions.

Figure 6 shows the code for solving the student groups problem. In this case, I defined a new data type called `Student` to be the data points and defined the `magnitude` of a `Student` to be its `grade`. Notice that since I am deriving an instance of `Ord` automatically, I must define `grade` before `name` and `section`. This ensures that the ordering over students will be determined by their `grade`s, and so be the same as the ordering over their `magnitude`s.

The `main` function is divided up into three logical units. First, we load a CSV file into a variable `allStudents::[Student]` using the `Cassava` package [2]. The details of how this works aren't too important.

In the middle section, we divide up the students according to which section they are in, and then `train` a `Schedule` model for each section. We use the function:

```
getSchedules :: Scheduling n a → [[a]]
```

to extract a list of schedules from our `Schedule` type, then print them to the terminal. Just for illustration, we train the third section's `Scheduling` model in parallel. With only about 30 students in the section, we don't notice any improvement. But as the data sets grow, more processors provide drastic improvements, as shown in Table 4.1.

In the last section, we combine our section specific models together to get a combined model factoring in all of the sections. Because `Scheduling` is an instance of `HomTrainer`, we don't have to retrain our model from scratch. We can reuse the work we did in training our original models, resulting in a faster computation.

```
{-# LANGUAGE TypeFamilies, DataKinds #-}

import Data.Csv
import qualified Data.ByteString.Lazy.Char8  as BS
import qualified Data.Vector  as V
import HLearn.Algebra
import HLearn.NPHard.Scheduling

-------------------------------------------------------------------

data Student = Student
   { grade   :: Double
   , name    :: String
   , section :: Int
   }
   deriving (Read,Show,Eq,Ord)

instance HasRing Student where
   type Ring (Student) = Double

instance Norm Student where
   magnitude = grade

-------------------------------------------------------------------

main = do
   Right allStudents ←
      fmap (fmap (fmap (λ(n,s,g) → Student g n s) . V.toList) . decode True)
      $ BS.readFile "students.csv" :: IO (Either String [Student])

   let section1 = filter (λs → 1 == section s) allStudents
   let section2 = filter (λs → 2 == section s) allStudents
   let section3 = filter (λs → 3 == section s) allStudents
   let solution1 = train section1 :: Scheduling 5 Student
   let solution2 = train section2 :: Scheduling 5 Student
   let solution3 = parallel train section3 :: Scheduling 5 Student
   print $ map (map name) $ getSchedules solution1

   let solutionAll = solution1 ◇ solution2 ◇ solution3
   print $ map (map name) $ getSchedules solutionAll
```

Figure 6: Solution to my professor's problem

Implementing the LPTF HomTrainer

Now we're ready to dive into the details of how our model, `Scheduling` works under the hood. `Scheduling` is defined as:

```
data Scheduling (p::Nat) a = Scheduling
    { vector   :: !(SortedVector a)
    , schedule :: Map Bin [a]
    }
```

Scheduling has two member variables. The first is a `SortedVector`. This custom data type is a wrapper around the `vector` package's `Vector` type that maintains the invariant that items are always sorted. This vector will be used as an intermediate data structure while performing the LPTF computations. The second member is the actual schedule. It is represented as a `Map` with a `Bin` as the key and a list of tasks as the value. `Bin` is just a type synonym for `Int`:

```
type Bin = Int
```

and it represents the index of the processor in the range of 1 to p.

Before we look at `Scheduling`'s `Monoid` and `HomTrainer` instances, we need to take a more detailed look at the LPTF algorithm. Traditionally, LPTF is described as a two step process. First, sort the list of tasks in descending order. Then, iterate through the sorted list. On each iteration, assign the next task to the processor with the least amount of work. Figure 7 shows a single iteration of this procedure.

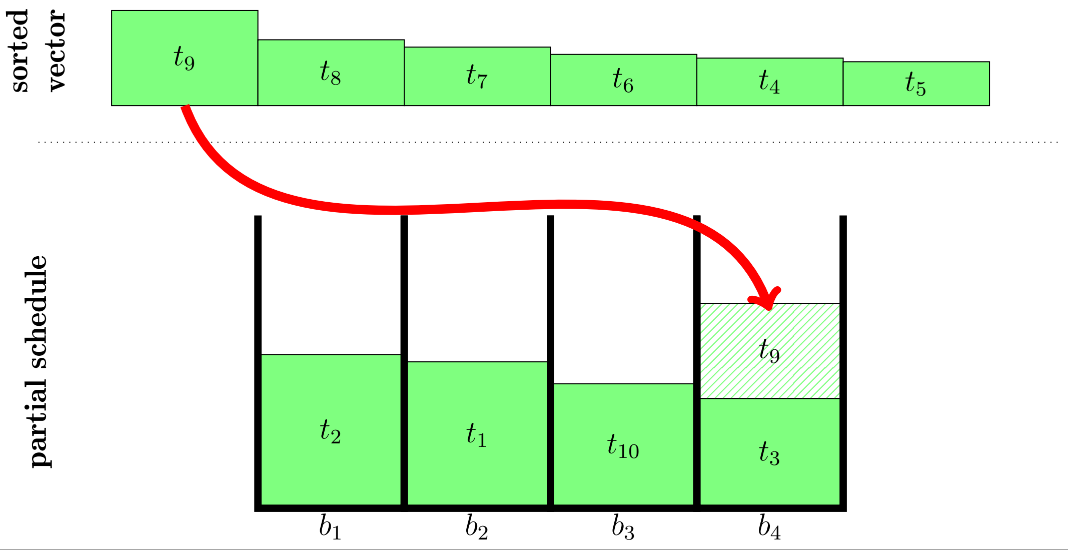

Figure 7: A single iteration of the LPTF algorithm

This conversion process is implemented with the internal function `vector2schedule`, whose code is shown in Figure 8 below. The details of this function aren't particularly important. What is important is that `vector2schedule` runs in time $\Theta(n \log p)$. This will be important when determining the run times of the `mappend` and `train` functions.

```
vector2schedule :: (Norm a) ⇒ Int → SortedVector a → Map.Map Int [a]
vector2schedule p vector = snd $ F.foldr cata (emptyheap p,Map.empty) vector
    where
        emptyheap p = Heap.fromAscList [(0,i) | i←[1..p]]
        cata x (heap,map) =
            let Just top = Heap.viewHead heap
                set = snd top
                prio = (fst top)+magnitude x
                heap' = Heap.insert (prio,set) (Heap.drop 1 heap)
                map' = Map.insertWith (++) set [x] map
            in (heap',map')
```

Figure 8: The `vector2schedule` helper function for LPTF

Our `mappend` operation will implement the LPTF algorithm internally in a way that reuses the results from the input `Schedule`s. We won't be able to reuse the actual schedules, but we can reuse the sorting of the vectors. We do this by taking advantage of the `HomTrainer` instance of the `SortedVector` type. It turns out that merge sort is a monoid homomorphism, and so `SortedVector` can be made an instance of the `HomTrainer` type class. The commutative diagram for `SortedVector` is shown in Figure 9 below.

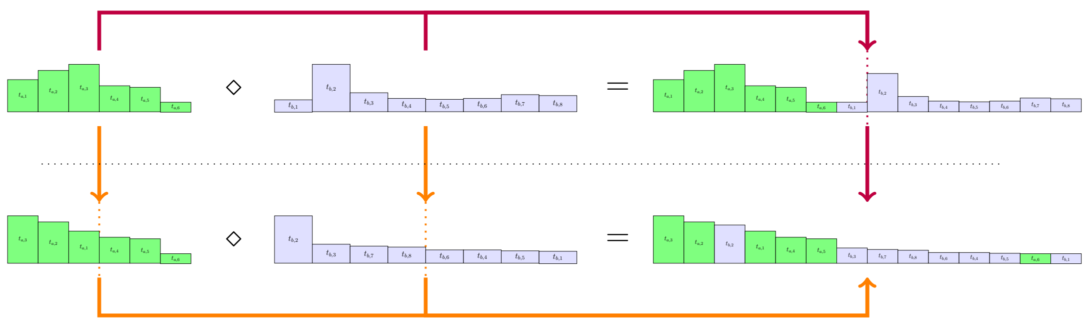

Figure 9: Constructing a `SortedVector` is a monoid homomorphism

It is important to note that `SortedVector`'s `mappend` operation does not take constant time. In fact, it takes $\Theta(n)$ time, where n is the size of both input vectors put together. The `HomTrainer` type class makes reasoning about these non-constant `mappend` operations easy. By looking up in Table 4.1, we can find the run times of the derived algorithms. Notice that if the monoid operation takes time $\Theta(n)$, then our batch trainer will take time $\Theta(n \log n)$, and this is exactly what we would expect for a sorting. Details of how these numbers were derived can be found in my TFP13 submission on the HLearn library [3].

Monoid operation (`mappend`)	Sequential batch trainer (`train`)	Parallel batch trainer (`parallel train`)	Online trainer (`add1dp`)
$\Theta(1)$	$\Theta(n)$	$\Theta\left(\frac{n}{p} + \log p\right)$	$\Theta(1)$
$\Theta(\log n)$	$\Theta(n)$	$\Theta\left(\frac{n}{p} + (\log n)(\log p)\right)$	$\Theta(\log n)$
$\Theta(n)$	$\Theta(n \log n)$	$\Theta\left(\frac{n}{p} \log \frac{n}{p} + n\right)$	$\Theta(n)$
$\Theta(n^b)$, $b > 1$	$\Theta(n^b)$	no improvement	no improvement

Table 4.1: Given a run time for `mappend`, you can calculate the run time of the automatically generated functions using this table. The variable n is the total number of data points being trained on or combined, and the variable p is the number of processors available.

With all of these building blocks in place, the `Monoid` instance for `Scheduling` is relatively simple. The `mempty` and `mappend` operations are exactly the same as they are for `SortedVector`, except that we also call the helper function `lptf`. This function just packages the `SortedVector` into a `Scheduling` type using the `vector2schedule` function we saw earlier.

```
instance (Ord a, Norm a, SingI n) ⇒ Monoid (Scheduling n a) where
    mempty = lptf mempty
    p1 ‘mappend‘ p2 = lptf $ (vector p1) ◇ (vector p2)

lptf :: forall a p. (Norm a, SingI p) ⇒ SortedVector a → Scheduling p a
lptf vector = Scheduling
    { vector = vector
    , schedule = vector2schedule p vector
    }
    where p = fromIntegral $ fromSing (sing :: Sing n))
```

Figure 10: The `Monoid` instance for the `Scheduling` model

Since `vector2schedule` runs in linear time and `SortedVector`'s `mappend` runs in linear time, the `Scheduling`'s `mappend` runs in linear time as well. By Table 4.1 again, we have that the automatically derived batch trainer will take time $\Theta(n \log n)$. This is exactly what the traditional LPTF algorithm takes.

Of course, we still have to implement the `HomTrainer` instance. But this is easy. Inside the `HomTrainer` class is a function called the "singleton trainer":

```
train1dp :: HomTrainer model ⇒ Datapoint model → model
```

All this function does is create a model from a single data point.[1] In practice, such a singleton model is rarely useful by itself. But if we define it, then the compiler can then use this function and `mappend` to build the other functions within the `HomTrainer` class automatically. This is how we get the online and parallel functions "for free."

The resulting `Scheduling` instance looks like:

```
instance (Norm a, SingI n) ⇒ HomTrainer (Scheduling n a) where
    type Datapoint (Scheduling n a) = a
    train1dp dp = lptf $ train1dp dp
```

Figure 11: The `HomTrainer` instance is quite short and mechanical to write

That's all we *need* to do to guarantee correct asymptotic performance, but we've got one last trick that will speed up our `train` function by a constant factor. Recall that when performing the `mappend` operation on `Scheduling` variables, we can only reuse the work contained inside of `vector`. The old `schedules` must be completely discarded. Since `mappend` is called many times in our automatically generated functions, calculating all of these intermediate schedules would give us no benefit but result in a lot of extra work. That is why in the `Scheduling` type, the `vector` member was declared strict, whereas the `schedule` member was declared lazy. The `schedules` won't actually be calculated until someone demands them, and since no one will ever demand a schedule from the intermediate steps, we never calculate them.

Back to Bin Packing

Since BinPacking and Scheduling were such similar problems, it's not too surprising that a similar technique can be used to implement a `BinPacking` model.

[1]The `train1dp` function is analogous to the `pure` function in the `Applicative` class, or the `return` function in the `Monad` class.

The main difference is that we'll replace the LPTF algorithm with another simple algorithm called Best Fit Decreasing (BFD). This gives us the performance guarantee of:

$$BFD \leq \frac{11}{9} OPT + 1$$

There are some slightly better approximations for BinPacking, but we won't look at them here because they are much more complicated. Chapter 2 of *Approximation Algorithms for NP Hard Problems* gives a good overview on the considerable amount of literature for bin packing [4].

BFD is a two stage algorithm in the same vein as LPTF. First, we sort the data points by size. Then, we iteratively take the largest item and find the "best" bin to place it in. The best bin is defined as the bin with the least amount of space that can still hold the item. If no bins can hold the item, then we create a new bin and add the item there. This is shown graphically in Figure 12 below.

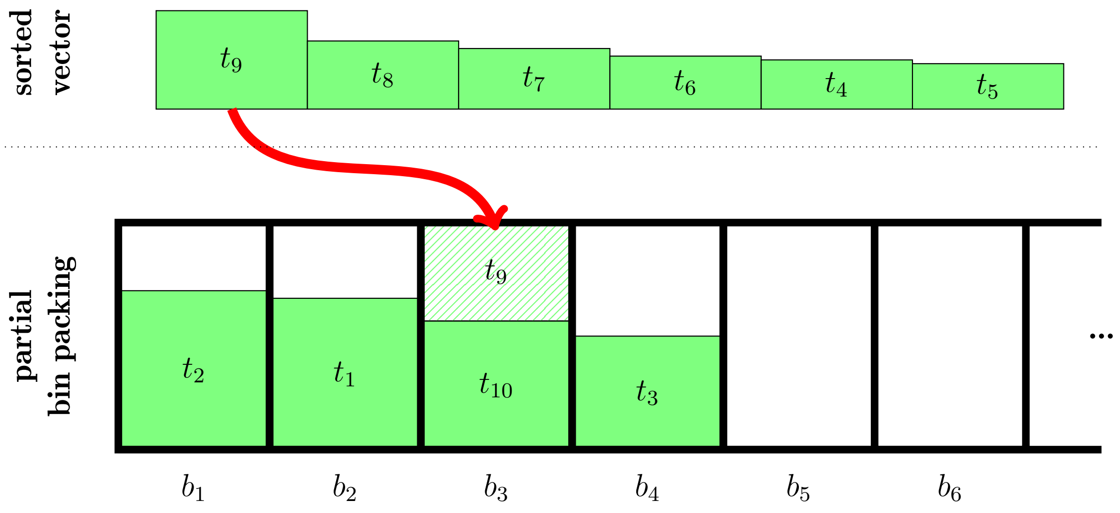

Figure 12: One iteration of the Best First Decreasing (BFD) algorithm

The data type for bin packing is:

```
data BinPacking (n::Nat) a = BinPacking
    { vector  :: !(SortedVector a)
    , packing :: Map.Map Bin [a]
    }
```

This is the exact same form as the Scheduling type had. The only difference is that we will use the BFD strategy to generate our `Map`. Therefore, by similar reasoning, the `BinPacking`'s `mappend` function takes time $\Theta(n)$ and its `train` function takes time $\Theta(n \log n)$. Again, this is exactly what the traditional description of the BFD algorithm requires.

Takeaways

Most instances of the `HomTrainer` type class are related to statistics or machine learning, but the class is much more general than that. For example, we've just seen how to use `HomTrainer`s to approximate two $\mathbb{NP}$-complete problems. So from now on, whenever you see a function that has type:

```
:: [datapoint] → model
```

ask yourself, "Could this algorithm be implemented using a `HomTrainer`?" If yes, you'll get online and parallel versions for free.

Finally, we've looked at the monoid structure for `Scheduling` and `BinPacking`, but these types also have Abelian group, $\mathbb{Z}$-module, functor, and monad structure as well. I'll let you explore the documentation available on the GitHub repository [5] (pull requests are always welcome!) to find creative ways to exploit these structures. If you have any questions or feedback, I'd love to hear it.

References

[1] R. L. Graham. Bounds on multiprocessing timing anomalies. **SIAM Journal on Applied Mathematics**, 17(2):pages 416–429 (1969).

[2] Cassava: A csv parsing and encoding library. `http://hackage.haskell.org/package/cassava`.

[3] Michael Izbicki. Hlearn: A machine learning library for haskell. **Trends in Functional Programming** (2013).

[4] E.G. Coffman Jr., M.R. Garey, and D.S. Johnson. **Approximation algorithms for NP-hard problems**, chapter Approximation Algorithms for Bin Packing: A Survey. PWS Publishing Co., Boston, MA, USA (1997).

[5] Hlearn source repository. `http://github.com/mikeizbicki/hlearn`.