Generalized Algebraic Data Types in Haskell

Anton Dergunov

Intel Corporation

Abstract. Generalized algebraic data types (GADTs) is a feature of functional programming languages that generalizes 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, etc. The article describes these use cases with small GADT programs and also describes usage of GADTs in Yampa programs.

Keywords: Haskell, functional programming, GADT

1 Introduction

Type systems are recognized as the most popular and best established lightweight formal method for ensuring that software behaves correctly [1]. They are used for detecting program errors, for documentation, to enforce abstraction, etc. Type systems are 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 single idea of GADTs is to allow specific return types of value constructors. In this way they generalize ordinary algebraic data types. 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 [3–9] and many of them 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. A quick introduction to GADTs is provided in section 2. The following use cases were described in this article[1]:

- domain-specific embedded languages (section 3);
- generic programming (section 4);
- ensuring program correctness (section 5).

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

Section 7 describes why type signatures are required for functions involving GADTs. Section 8 describes an alternative implementation of one of the examples without GADTs usage. Section 9 describes usage of GADTs in Yampa, a domain-specific language for functional reactive programming. Finally the conclusions are summarized in section 10.

2 GADTs in a Nutshell

Algebraic data types 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 component: one of them has type `String` and another one - type `a`.

We can construct values of this type this way[2]:

```
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:

- Using command line option -XGADTs.
- Using LANGUAGE pragma in source file. 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:

[2] We use `"ghci"` to denote a prompt of GHCi session.

```
{-# 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
```

The key feature of GADTs is that pattern matching causes type refinement. In the right-hand side of the following equation the type of `a` is refined to `Int`:

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

Examples in the following sections show the real practical value of GADTs.

3 Expression Evaluator

This section introduces GADTs with a canonical example of 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 some operations on them:

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

`BoolVal` value constructor wraps Boolean literal. `IsZero` is an unary function that takes an integer and returns a Boolean value. One can 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`. Also, evaluation can fail, because the input expression is incorrect. Finally, type signature is the following:

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

And implementation of 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 it is required to represent the 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 value constructor `IntVal` return type 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 what 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 ability to return specific types in value constructors, for example `PhantomExpr Int`. GADT for the expression language looks this way:

```
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
```

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, so for example in the right-hand side of the following expression `i` has type `Int`:

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

Section 8 describes one more implementation of expression evaluator.

4 Generic Programming with GADTs

Suppose it is required to write a function to encode data in binary form[3]. This function must be able to work with several types. Functions like this one can be implemented using type classes. But GADTs offer an interesting alternative.

First we need to declare a *representation type* [4]:

```
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:

[3] Ideas for this section were taken from [3].

```
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`[4]:

```
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 only 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: add a value constructor for the `Dynamic` data type.

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

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

[4] This code requires using `ExistentialQuantification` extension.

```
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 we are not sure about the actual type of the data. In this case it is required to use a type cast to get the 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
```

A more generic solution is provided in [3].

5 Proving Correctness of List Operations

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 if list is null. To address this problem it is possible to define a type of non-empty lists. First it is required to define two empty data types[5]:

[5] This requires `EmptyDataDecls` extension.

```
data Empty
data NonEmpty
```

Now it is possible to define a 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 its 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
```

But implementation of a function to repeat an element 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 root cause 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[6]:

```
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
```

[6] Actually, with the current data type definition a term `Cons 'a' Nil` can even be given type `SafeList Char Int`. To fix this problem it is required to give same kind for `Empty` and `NonEmpty` types. This is discussed later for `Nat` data type.

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))
```

But `SafeList` data type does not have enough static information to prove list length invariants for list functions. For example, for concatenation function we need to show that length of the concatenated list is a sum of source lists lengths. So 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)`, etc. Now 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 reverse function does not change the length of a list:

```
reverseSafe :: List a n -> List a n
reverseSafe Nil = Nil
reverseSafe (Cons x xs) = Cons x (reverseSafe xs)
```

To implement 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 [?]. First we need to declare type family `Plus`[7]:

```
type family Plus a b
```

Than 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 concatenation function signature:

[7] This requires `TypeFamilies` extension.

```
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)
```

In fact, at the moment `Succ` has type parameter of kind `*`, so it possible to write nonsensical terms like `Succ Int`, and they will be accepted by the type checker. This problem can be addressed by promotion [10]. We can declare the following data type:

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

Here `Nat` is a type and `Zero` and `Succ` are value constructors. But due to promotion `Nat` also becomes a kind and `Zero` and `Succ` - also become types. 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 Peano number 2 can be written as:

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

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

```
type T' = 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`[8].

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

The implementation of the `repeatElem` function is 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.

The solution to this puzzle is the use of *singleton types* which allow to express dependency between values and types [5]. The singleton for Peano numbers type is the following GADT:

[8] This requires `DataKinds` extension.

```
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 ⊥ value) of the singleton data type where parameter `n` has exactly this type. For example:

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

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[9]:

```
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 in the context are the same. Thus, is it only possible to use this function when the index does not exceed the list length.

6 Proving Correctness of Red-Black Tree Insert Operation

A red-black tree is a binary search tree where every node has either red or black color[10]:

[9] `TypeOperators` extension is required to be able to define `:<` operation for types.

[10] The implementation of a red-black tree presented here is taken from [6].

```
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 greater than `x` are stored in left sub-tree (in `l`) and values less 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) 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.

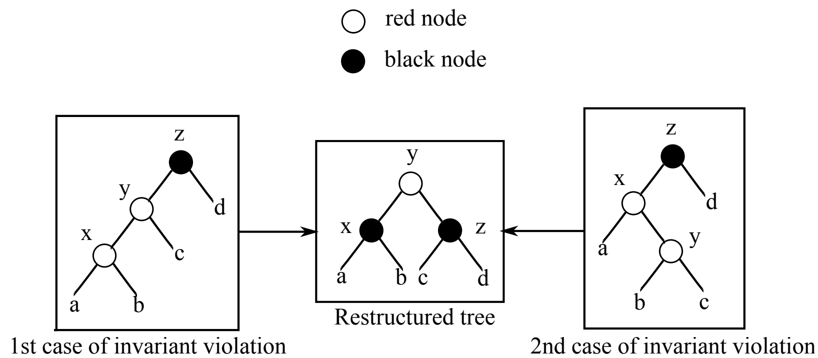

Fig. 1. Possible cases of 3rd invariant violation after insertion in the left branch of the node.

- To maintain the first 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 1 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:

```
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.

We need to show that the 4th invariant is maintained by the insertion operation[11]. This requires adding 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 ???
```

[11] The source code for the verified red-black tree was originally written by Stephanie Weirich for a university course and was described in the talk [7].

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)[12]:

```
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 color must be passed as a type (for `IncBlackHeight` type family) and as a value (to the value constructor). So, similarly as in section 5, 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
```

Because 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*[13]:

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

Or it is also possible to do this with GADT:

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

[12] This code requires `TypeFamilies` and `DataKinds` extensions.
[13] This definition requires extension `RankNTypes`.

The implementation of insertion operation never violates the 4th invariant, so the remaining changes are adjustments of type annotations, etc.

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. First we define a type class with 3 parameters corresponding to color of the parent and colors of the child nodes[14]:

```
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[15]:

```
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 1), so we will not be able to represent the tree using this data type. Thus it is required 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)
```

[14] This requires `MultiParamTypeClasses` extension.
[15] This requires `FlexibleInstances` extension.

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 it 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 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[16]:

```
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.

```
leftBalance c (IntNode SR a@(N SR _ _ _) x b) z d =
    error "can't happen"
leftBalance c (IntNode SR a x b@(N SR _ _ _)) z d =
    error "can't happen"
```

[16] Note that the case of one regular node and one leaf node is not valid, because these nodes must have different black heights.

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
```

Of course, the implementation of the `concatenate` function does not meet our expectations. But this code still type checks.

7 Type Signatures for Functions Involving GADTs

Hindley-Milner (HM) is a classical type inference method [11]. One of the most important properties of HM is ability to always deduce the most general (principle) type of every term.

However, GADTs pose a difficult problem for type inference, because programs with GADTs lose principle type property [12]. 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, 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
```

More information on type inference for programs with GADTs is provided in [12].

8 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 [13].

For example, expression evaluator from section 3 can 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 expression is implemented in a similar way:

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

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
```

9 Usage of GADTs in Yampa

Yampa is a domain-specific language for functional reactive programming (FRP) [14]. FRP is a programming paradigm of expressing data flows using the building blocks of functional programming.

Based on the information from [9], this section describes how GADTs were used to improve performance of Yampa programs.

Fig. 2. Signal function `SF a b`.

Signal function is a central abstraction in Yampa. It represents a simple synchronous process mapping an input signal to an output signal (see figure 2). 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
```

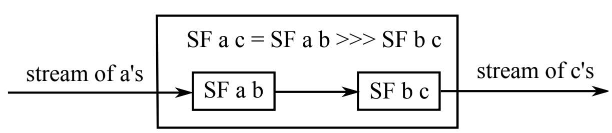

Fig. 3. Composition of signal functions.

The following function provides a composition of signal functions (as shown with figure 3):

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

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 tricks as in section 3 for 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 does not type check, because when we pattern match using `SFId` value constructor, the type is still `SF a b`, not `SF a a`. This situation is similar to the case with phantom types described in section 3. The solution is to use GADT to represent the signal function:

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

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

Table 1. Performance improvements enabled by GADTs in Yampa programs

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

There are other performance improvements that are enabled by GADTs in Yampa [9]. The results of performance improvements are shown in table 1 that was taken as is from [9]. The table shows execution time of several benchmarks using initial simply-optimized implementation (T_S) and the implementation with GADT-based optimizations (T_G). It is also noted in [9] that GADTs allowed to write a more concise and cleaner API without the need of pre-composed signal function (that were defined only for performance reasons).

10 Conclusion

This article has demonstrated the usefulness of GADTs in practice with several use cases:

- We have shown that GADTs are useful for domain-specific embedded languages: they allow to statically type-check valid expressions.
- GADTs allow to express 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.

As a result, we have shown that the notion of GADTs is a very valuable extension of the Haskell language.

11 Acknowledgments

This tutorial article was inspired by Simon Peyton-Jones talk on 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 grateful to have ability to use this great example of GADTs usage in this tutorial.

References

1. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA, USA (2002)
2. Hudak, P., Hughes, J., Peyton Jones, S., Wadler, P.: A history of haskell: being lazy with class. In: Proceedings of the third ACM SIGPLAN conference on History of programming languages. HOPL III, New York, NY, USA, ACM (2007) 12–1–12–55
3. Hinze, R., et al.: Fun with phantom types. The fun of programming (2003) 245–262
4. Weirich, S.: RepLib: a library for derivable type classes. In: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell. Haskell '06, New York, NY, USA, ACM (2006) 1–12

5. Eisenberg, R.A., Weirich, S.: Dependently typed programming with singletons. In: Proceedings of the 2012 symposium on Haskell symposium. Haskell '12, New York, NY, USA, ACM (2012) 117–130
6. Okasaki, C.: Red-black trees in a functional setting. J. Funct. Program. **9**(4) (July 1999) 471–477
7. Weirich, S.: Dependently-typed programming in GHC. In: Proceedings of the 11th international conference on Functional and Logic Programming. FLOPS'12, Berlin, Heidelberg, Springer-Verlag (2012) 3–3
8. Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(5) (September 2009) 509–543
9. Nilsson, H.: Dynamic optimization for functional reactive programming using generalized algebraic data types. In: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming. ICFP '05, New York, NY, USA, ACM (2005) 54–65
10. Yorgey, B.A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Giving Haskell a promotion. In: Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation. TLDI '12, New York, NY, USA, ACM (2012) 53–66
11. Cardelli, L.: Basic polymorphic typechecking. Sci. Comput. Program. **8**(2) (April 1987) 147–172
12. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming. ICFP '09, New York, NY, USA, ACM (2009) 341–352
13. Sulzmann, M., Wang, M.: GADTless programming in Haskell 98. (2006)
14. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. Haskell '02, New York, NY, USA, ACM (2002) 51–64