Investigation of GADT applications and usage

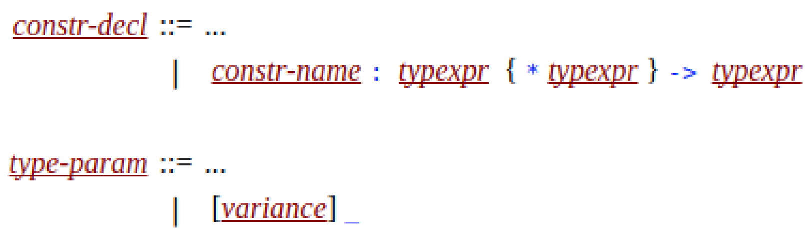
PARTH SHAH, Stanford University, USA

Generalized Algebraic Datatypes, or GADTs, extend algebraic datatypes by allowing an explicit relation between type parameters and constructor definitions. They can be used in numerous applications like modeling programming languages, maintaining invariants in data structures and in implementing constraints in domain-specific languages. In our project, we discuss GADTS and provide some of their use cases. We also mention our experience while using them in practice and discuss both advantages and disadvantages associated with them.

ACM Reference Format:
Parth Shah. 2017. Investigation of GADT applications and usage. 1, 1 (December 2017), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Generalized Algebraic Datatypes(GADTs), extend usual sum types in two ways: constraint on type parameter may change depending on the value constructor and some type variables may be existentially quantified. They were introduced in OCaml in version 4.00 and as mentioned in the OCaml-manual[10], we can write a constructor for GADT in the following way:



```
constr-decl ::= ...
             |  constr-name : typexpr { * typexpr } -> typexpr


type-param ::= ...
             |  [variance] _
```

The rightmost *typexpr* in the above *constr-decl* is the return type and needs to be explicitly mentioned. This return type must use the same type constructor as defined and have the same number of parameters as in the definition. The underlying information can be retrieved by using pattern matching on the constructor types. GADT advantages are two folds: they allow to incorporate invariants in type definitions and to use these invariants to reduce the dynamic checks needed.

The idea that we can incorporate types within constructor definitions, makes using GADTs a very attractive option while implementing an evaluator for a language. The blog post[3] describes an evaluator built for a very simple language using this idea. We have already built an evaluator for typed lambda calculus as part of assignments during the course. We used ADTs in the assignment and had to implement a complex type checking module to ensure that the terms are well-formed. GADTs provide an alternative solution, which is much cleaner, intuitive and completely remove the need for this separate module. This aspect was our prime motivation to further explore this language extension for our project. During the course, we had explored the *Sum*, *Product* and the *Existential* types and the concept of GADTs builds over those ideas to provide extra flexibility and features.

Author's address: Parth Shah, Stanford University, 21 Comstock Circle, Stanford, CA, 94305, USA.

Algorithm 1 Definition using ADTs

```
module Term = struct
    type binop = Add | Sub | Mul | Div
    type t =
        | Int of int
        | Var of string
        | Lam of string * Type.t * t
        | App of t * t
        | Binop of binop * t * t
        | Tuple of t * t
        | Project of t * direction
        | Inject of t * direction * Type.t
        | Case of t * (string * t) * (string * t)
end
```

The rest of the paper in organized as follows. In section 2, we discuss our implementation of an evaluator for typed lambda calculus and compare with our experience with the implementation using ADTs. In section 3, we discuss a data structure implemented using GADTS, a red-black tree, and the challenges faced while implementing it. In section 3.4, we provide the conclusion for our project.

2 USING GADTS FOR EVALUATING TYPED LAMBDA CALCULUS

One of the canonical example to showcase use of GADTs is to use them for representing higher-order abstract syntax trees. As the first step of our project, we also implemented an interpreter for a typed lambda calculus. We decided to pick the same language model that we used in Assignment 3 and 4 during the course. There were two major reasons for this. Firstly, we already had our experience during the course while using ADTs to solve the problem, so it provided a reference for our GADT experience. Secondly, the language model after the extensions of Assignment 4 was non trivial and had several complex constructs. We also added the boolean type and if statement to show how GADTs could be easily used to extend the language model.

2.1 Advantages of GADTs

Definitions in 1 and 2 provide the *Term* definition for implementation using ADTs and GADTs respectively. We now mention the advantages of using GADTs over ADTs for implementation.

2.1.1 Type Safety. Using GADTs, we are able to associate types with the expression. This information is extremely valuable when we want to implement type safety. *expr* in definition 2 is associated with a type and we can define the return types for different constructors like the *GInt* returns *int expr* and *GBool* returns a *bool expression*. We can use these to define other constructors and provide *compile-time type checks* rather than doing it at runtime as in the case of ADTs. Taking the example of **GBinop:** *binop * int expr * int expr -> int expr*, we have defined that it would take two int expressions as input and return an int expression. This is not possible in the case of ADTs where the corresponding expression is *Binop* of *binop * t * t* which provides no restriction on the type of *t*. Similarly, we see it is extremely simple to denote these relations for other expressions as well like *GIf, GApp and GCase*. We did not need the type checking module at all for our implementation which used GADTs. We were able to express all the constraints and relations using the typed expression notation.

2.1.2 Compiler Optimization. Mentioning types with the expression and constraining typed expressions as input also allows the compiler to optimize the code. Again taking the example of *Binop*, since we have mentioned

Algorithm 2 Definition using GADTs

```
module Term = struct
    type binop = Add | Sub | Mul | Div
    type ('a, 'b) sum = Left of 'a | Right of 'b
    type ('a, 'b) tuple = Pair of 'a * 'b
    type _ expr =
        | GBool : bool -> bool expr
        | GIf : bool expr * 'a expr * 'a expr -> 'a expr
        | GEq : 'a expr * 'a expr -> bool expr
        | GLt : int expr * int expr -> bool expr
        | GInt : int -> int expr
        | GBinop : binop * int expr * int expr -> int expr
        | GApp: ('a -> 'b expr) * 'a expr -> 'b expr
        | GTuple: 'a expr * 'b expr -> ('a, 'b) tuple expr
        | GProjectLeft: ('a, 'b) tuple expr -> 'a expr
        | GProjectRight: ('a, 'b) tuple expr -> 'b expr
        | GInjectLeft: 'a expr -> ('a, 'b) sum expr
        | GInjectRight: 'b expr -> ('a, 'b) sum expr
        | GCase: ('a, 'b) sum expr * ('a -> 'c expr) * ('b -> 'c expr) -> 'c expr
end
```

that the input expressions are restricted to type int, the compiler now does not flag warning when we match the inputs only to ints and further can produce a better switch case code after compilation, getting rid of some unnecessary check conditions. This is further covered in more detail in 8.1.3 in[2]. To validate our theory about compiler optimization we decided to run tests to **benchmark** *Binop* operation. The GADT implementation was approximately 4 times faster compared to the earlier implementation. 10^8 operations took around 1 second in the GADTized version and 4 seconds in the ADT version. No typechecker was invoked while benchmarking ADT, so the actual time taken would be even more in the case of ADTs.

2.1.3 Readability. Terms, as defined in GADTs, provide more information about their working, inputs, and outputs than the corresponding terms in ADT. Again considering the example of *Binop*, we know that it would take two int expressions and return an int expression. From the ADT definition, we get no such information. It takes in two terms and returns a term. GADT expressions, in our opinion, seem to provide more intuition about the function than the corresponding ADT expressions.

2.2 Disadvantages of using the GADT approach

While using GADT provide several advantages in type safety and optimizations, there are certain downsides to using this approach as well.

2.2.1 Separate projection and inject functions. As seen in Definition 2 we have separate functions for projections and injections instead of just one in Definition 1. The addition of type parameter in the expressions means that ProjectLeft and ProjectRight no longer return the same expression(type $'a$ expr and $'b$ expr are not considered same) and hence cannot be returned from different branches of the same match expression. Similarly, injectleft and injectright returns expression with different types and cannot be merged together.

Algorithm 3 Evaluator on GADT definition

```
let rec eval : type a. a Term.expr -> a = fun x ->
    match x with
    | Term.GBool (b) -> b
    | Term.GInt (i) -> i
    | Term.GIf (b, l, r) -> if eval b then eval l else eval r
    | Term.GEq (a, b) -> (eval a) = (eval b)
    | Term.GLt (a,b) -> eval a < eval b
    | Term.GBinop(binop, l, r) ->
    (match binop with
        | Term.ADD -> eval l + eval r
        | Term.SUB -> eval l - eval r
        | Term.MUL -> eval l * eval r
        | Term.DIV -> eval l / eval r
    )
    | Term.GApp(a, b) -> eval(a(eval(b)))
    | Term.GProjectLeft(t) ->
    (match t with
        | Term.GTuple(l, _) -> eval(l)
    )
    | Term.GProjectRight(t) ->
    (match t with
        | Term.GTuple(_, r) -> eval(r)
    )
    | Term.GCase(a, lf, rf) ->
    (match a with
        | Term.GInjectLeft(l) -> eval(Term.GApp(lf, l))
        | Term.GInjectRight(r) -> eval(Term.GApp(rf, r))
    )
```

2.2.2 Absence of Lam function. Our new implementation does not provide support for *Lam* function. *Lam* function takes in a variable, the type of the variable and the body. The type of the body is not trivial to determine and would require analysis of the body which can not be described with a simple constructor. Therefore, the type of the operation can not be determined by the constructor and including it would break the whole type safety of the implementation and we would have to reintroduce the type-checking module.

One round-about for this problem is to use OCaml built in functions. Using them instead of Lam in our implementation provides the same type of flexibility and the expressiveness as the original language and also makes type inference easier. Below is an example usage of GApp in our new implementation

```
Term.GApp((fun (x: int)->Term.GInt(x)), Term.GInt(1))
```

2.2.3 Complex function definitions. Using GADTs make the function definitions more verbose, cumbersome and difficult to manage. Because of the introduction of types in definition of *expr*, we need to bind these types before we can use them in the expression. The recursive implementation in 3 provides an example for this where *Type a.a Term.expr* binds type variable *a* and which can be then used. This is very much like the existential types definitions. Also, since expressions now involve types, we need to provide them with every instantiations.

Algorithm 4 Type Definitions for RBTree

```
type z = Z : z
type ’n s = S: ’n -> ’n s
module Node = struct
    type black = Black : black
    type red = Red : red
    type (_,_, _) node =
        | Null : (black, z, ’a) node
        | BlackN : (_, ’n, ’a) node * ’a * (_, ’n, ’a) node -> (black, ’n s, ’a) node
        | RedN: (black, ’n, ’a) node * ’a * (black, ’n, ’a) node -> (red, ’n, ’a) node
end
module RBTree = struct
    type _ root =
        | Root : (Node.black, ’l, ’a) Node.node-> ’a root
end
```

3 BUILDING RED-BLACK TREES USING GADTS

We mentioned in Section 1 that we can use GADTs to incorporate invariants in the type and we decided to implement a data structure to further explore this aspect of relation. A red-black tree follows the given constraints:

- Every leaf node(Null) is black
- If a node is red then both the children are black.
- Every path from a node to any descendant leaf node contains the same number of black nodes
- The root node is always black

3.1 Type Design

We would like our type definition using GADTs to naturally inherit all these constraints. Our type design is described in 4. The type *z* and *s* are singleton types and used to represent the black-height of a node. The type *node* takes three parameters: color, black height and the type of value. We have described three different nodes based on the data-structure invariants:

- Null: Leaf nodes. Color is set to black and black height is set to *z*
- BlackN: Black nodes which are not the leaves. It takes two children with same black-height($'n$) and returns a node with $'n\ s$ height.
- RedN: Red nodes. Both the children have the color black and same black height

The Tree root is defined to take a node with black color and return a root with the same type as the underlying data in the node. As seen in the definition, it is very simple and intuitive to encode the data-structure rules in the definition.

The major advantages of this scheme are that if we have a Tree root, we know for certain that the insert and other operations would always work in the $\log(n)$ time and the tree is a Red-Black tree indeed as the rules would not allow for anything else. These kind of guarantees are very useful from the user perspective and we get all these just from defining our data in the right way.

However, this type definition is significantly difficult to maintain compared to a *Sum* based type definition. We now describe the challenges faced during implementation.

Algorithm 5 Insert Function Definition

```
let rec insert: type c d l m a.(c, l, a)Node.node -> a -> (d, m, a)Node.node
```

Algorithm 6 Indirection

```
type 'a type_node =
    T : ('c, 'n, 'a) Node.node -> 'a type_node;;
```

3.2 Challenges faced

The presence of three parameters in the node definition makes it very complex to define functions over the node type. Following are few of the major challenges that we faced followed by our way of countering them:

3.2.1 Base Algorithm cannot be used. The major concern that we realized as soon as we began our implementation is that the base pseudo-code which is widely available is not valid for our implementation. All the pseudo codes always insert a red-node, which might make the tree invalid for a while and then the algorithm proceeds to balance the tree. This implementation is not valid for us because of the stringent type definition. The type-definition makes sure that all the nodes in the tree are valid nodes and we can not have the inconsistency even for a while.

We tried researching the available solutions for this problem and found several implementations in Haskell. Apart from being in a different language, the implementations used a separate context information or a lot of helper functions making the code unintuitive and very divergent from the original pseudo code. We initially thought to maintain a list of nodes on the path to the insertion nodes and then handle the conflicts in the list of nodes, modifying them on the go. But this solution can not be used because all the nodes had different types and can not be part of the same list. After a lot of thought, we decided to introduce black nodes in case of conflicts and handle the conflicts while tracking back. It also involved introducing another level of abstraction to convey the information to parent nodes about the potential conflicts. The final implementation is extremely verbose, but is still able to maintain a resemblance to the original pseudo-code.

3.2.2 Inserts cannot return a node type. An insert function to add a new value to a tree rooted at $Node\ n$ would normally have a return type of $Node$ but that is not straight-forward in the case of our type definition. Consider the $insert$ function defined in 5. The definition takes $(c, l, a)Node.node$ and value of type a and return a Node of type $(d, m, a)Node.node$. The return type node can have different color and also different black-height than the original node, we, therefore, have to use different types and these types cannot be determined beforehand. OCaml interpreter does not allow this kind of representation and this issue is further explained more in Section 8.4.2 in[2] and in the blog[6]. The main reason is that the above definition states that this is true for all possible values of d and m while what we want to represent is that these values are true some values of d and m. We, therefore, have to use a level of indirection to avoid this conflict. We used several different indirections in our implementation, one of which is shown in 6.

3.2.3 More complex pattern matching. As mentioned in the above section, we have to use an indirection for doing inserts on the node. This means that we now need more pattern matching to retrieve the information we encoded earlier. This practice makes implementation a lot more verbose and difficult to understand. Further, whenever the Nodes are now instantiated we need to ensure the subnodes are of the matching types, this includes more explicit type checking and pattern matching to ensure this, even in the case when the user knows the type is going to be correct based on the algorithm.

3.3 Final Implementation

The final implementation of the insert and find operations red-black tree took a little over 170 lines. Although the type definitions were extremely concise and intuitive, same cannot be said about the functions built over those definitions. For the sake of convenience the code is made available on github[9].

We tested our code for correctness by inserting 10^5 values and then asserting the insertion using a find function. The final black-height of the tree was 16 which is logarithmic in scale. The time taken for insertion for 10^5 values was around 5 ms which validated that the insertion were indeed $O(\log n)$. The time taken in case of $O(n)$ insertion for even 10^4 values is 30 ms.

3.4 Conclusion

General Algebraic Datatypes provides an important language extension over the *Sum* type. While it is extremely intuitive to encode constraints and types in the constructor definition, in our experience the functions that are built over such types were more verbose and complex than their counterparts using ADTs. Writing the interpreter for GADTized language was very convenient and concise. The use of types provide for compile-time type safety which is a huge improvement and at the same time allows compiler to optimize the pattern matching leading to faster execution, up to 4 times in the case of *Binop* as seen in 2.1.2.

Our experience while designing the data structure was completely different. While implementing the Red-black tree we had to manipulate the underlying algorithm to use it in our setting. Even after that, the indirection imposed by the use of GADT made the code unreadable and very inefficient to manage and debug. We found that such indirection is a common disadvantage associated with GADT usage[6]. The problem in the case of evaluator involved only one parameter and was easy to manage but the data structure implementation involved three parameters, greatly increasing the complexity for the subsequent operations and things got ugly pretty quickly. The blog[6] also sheds light on this aspect.

Another major holdback for using GADTs is the lack of proper documentation for its usage. The official implementation page[1] is very concise and does not provide reasonable examples. The manual[10] is very short and does not explain the full complexity involved in the concept. The type inference is also very complex in the case of GADTs and the given that the type is very important in constructor definitions and is needed for instantiations, it compounds the problem. The workshop article[4] sheds some light on how the type inference in implemented in OCaml.[8] and[5] provide a more formal treatment for the type theory involved in GADTs.

Overall, GADTs look like an interesting language construct and they provide significant optimization over the simplified ADT implementation by allowing for compiler optimizations. The fact that they are being used in Jane Street[7] to optimize performance further strengthens the claim. However, lack of proper documentation and complex example use cases in domains other than modeling languages has led to low public interest in the domain. This project was meant to explore the topic and provide examples in modeling languages and other domains(red-black tree). Through our experience, we were able to highlight both the advantages and disadvantages associated with the concept. The list of resources compiled along the way can also provide a good base for person who wants to learn about GADTs.

REFERENCES

[1] caml.inria.fr. 2011. GADTs in OCaml. (2011). https://sites.google.com/site/ocamlgadt/

[2] Dr Jeremy Yallop Dr Anil Madhavapeddy. 2015. Programming using GADTs. (Jan. 2015). https://www.cl.cam.ac.uk/teaching/1415/L28/gadts.pdf

[3] Mads Hartmann. 2015. Detecting use-cases for GADTs in OCaml. (Jan. 2015). http://mads-hartmann.com/ocaml/2015/01/05/gadt-ocaml.html

[4] Jacques Le Normand Jacques Garrigue. 2011. Adding gadts to ocaml: the direct approach. In *Workshop on ML*.

[5] Jacques Le Normand Jacques Garrigue. 2011. Adding GADTs to OCaml the direct approach. (2011). http://www.math.nagoya-u.ac.jp/~garrigue/papers/inria2011.pdf
[6] Anders Fugmann Jonas B. Jensen and Mads Hartmann Jensen. 2015. Practicalities and trade-offs in programming with GADTs. (Sept. 2015). http://engineering.issuu.com/2015/09/17/gadt-practicalities.html
[7] Yaron Minsky. 2015. Why GADTs matter for performance. (March 2015). https://blog.janestreet.com/why-gadts-matter-for-performance/
[8] Yann RÃĺgis-Gianas. 2015. Towards proved programs. Part I: Introducing GADTs. (Jan. 2015). http://yann.regis-gianas.org/mpri/01-gadts-checking.pdf
[9] Parth Shah. 2017. CS 242 project code. (Dec. 2017). https://github.com/shahparth95/CS242project
[10] Alain Frisch Jacques Garrigue Didier RÃĺmy Xavier Leroy, Damien Doligez and JÃĺrÃťme Vouillon. 2017. OCaml Manual. (Nov. 2017). https://caml.inria.fr/pub/docs/manual-ocaml-4.06/extn.html#sec251