Applications of Generalized Algebraic Data Types in OCaml

Junjie Ke
Department of Computer Science
Stanford University
junjiek@stanford.edu

Abstract

The introduction of generalized algebraic data types (GADTs) makes it syntactically possible to constrain type parameters for the return type of the constructors of a data type, which results in more efficient compilation, better memory representation and higher performance. In this project, I look into various applications of GADTs in order to understand its usage and benefits. In particular, I use GADTs in OCaml for implementing a lambda calculus-like language, an efficient, well-typed LR parser and an efficient AVL tree. Results show that the GADTs LR parser can be well-typed, efficient without runtime checks and that the GADTs AVL tree is almost twice as efficient as the traditional non-GADTs implementations.

1. Introduction

Generalized algebraic data types, or GADTs, is a way to restrict the return types of constructors for providing a variety of non-regular types. Previously, it's sometimes hard for the compiler to figure out the return types for each branch of pattern matching which results in valid but non-typecheck codes. GADTs overcomes the limitation through an explicit local specification of the typing context in each branch.

Appropriate use of GADTs allows the typechecker to know more about the valid states of the program which can be verified during compilation instead of runtime. Moreover, GADTs allow the programmer to track more information about their datatypes which make it possible to write safer, well-typed and more efficient code. As a result, both existing paradigms (e.g. indexed lists) and previously uncheckable paradigms (e.g. typed printf) can be improved.

In this project, I investigate three different applications of GADTs in order to gain a better understanding of its usage, performance and benefits of using them.

Firstly, I familiarize myself with GADTs in OCaml by implementing a lambda-calculus like language in both GADTs and non-GADTs way. This is a canonical example to demonstrate how generalized algebraic data types can improve the efficiency and expressiveness of the program.

Secondly, I build a well-typed GADTs LR parser in [3] as well as its optimized variant. Compared to the non-GADTs version, the GADTs LR parser successfully get rid of redundant runtime check while remaining well-typed and improving the performance.

In order to have a better understanding of GADTs' impact on the performance of real-world applications, I choose to re-implement a classic data structure, the AVL tree, using GADTs. I compare the performance between non-GADTs and GADTs AVL tree, showing that the GADTs implementation can improve the insertion speed by $\times 1.7$.

Lastly, I formalize the notion of GADTs in a lambda calculus type system and establish the type soundness of the type system.

In section 2, I describe related works on the GADTs theory and real-world applications. In section 3, I would present the design and techniques of the GADTs applications I implement. In section 4, I would present the experiment results as well as the formal semantics for using GADTs in the lambda calculus and a proof of progress, preservation.

2. Related Works

2.1. GADTs Theory

Generalized algebraic data types were introduced by Xi, Chen and Chen under the name *guarded recursive datatype constructors* [5]. A guarded recursive datatype is a datatype with local (i.e. existentially quantified) type variables associated with each data constructor. Each type argument for constructing the datatype can be specialized within each datatype case. They formalized the type theory of GADTs by introducing an explicitly typed calculus λ_{2,G_μ} language and also an elaboration process from an implicitly typed source language to λ_{2,G_μ} for supporting unobstrusive programming.

Independently, Cheney and Hinze formulated a very similar system named as *first-class phantom types* [1]. Phantom types, first introduced by Leijen and Meijer [2] to embed

domain specific languages into Haskell type safely, are parameterized types that do not use their type arguments at runtime. Instead, the arguments are only checked statically at compile time. However, the original phantom type encodings can only enforce type constraints when constructing values, but not when decomposing a value. Cheney and Hinze filled the gap by putting the formerly unused type variables in type equations to refine the argument types. They showed that their first-class phantom types can be used to define type representations and generic functions in a more efficient and more expressive way.

2.2. GADTs Applications

More recently, generalized algebraic data types have been put to a large variety of uses. Those use cases demonstrate that GADTs enables functional programmers to code any dependently-typed program by using some simple encoding tricks.

Well-typed Parsers

Parsing is a classic and extremely well-studied topic in programming language. The most straightforward way to showcase the expressiveness and efficiency of GADTs is writing a fast and type-safe parser which does not require values to carry run-time tags. Pottier and Regis-Gianas [3] showed that, for a fixed LR(1) automaton, the inductive invariant that describes the stack and guarantees safety can be encoded as GADTs. In addition to a safety guarantee, this approach has better performance than non-GADTs ML implementation because it gets rid of redundant tags and dynamic checks (e.g. stack cells must be redundantly tagged with *nil* or *cons*). However, Pottier et al. mainly focused on the theoretical safety guarantees and did not report any performance figures. In this project, I would put the effort in implementing both the GADTs and non-GADTs LR parsers using OCaml as well as running performance comparison experiments.

More Efficient Functions

With the power of GADTs, it is possible to re-implement real-world applications in a more efficient and type-safe form. For example, Vaugon [4] proposed to use GADTs instead of strings to represent printf/scanf formats in OCaml, which not only improves the performance but also fixes potential bugs and stabilizes the code. This new implementation of formats has a positive impact on OCaml community and is a real usage of GADTs to elegantly solve real-world problems.

However, since GADTs is relatively new to OCaml programmers and the syntax is somewhat complicated at first glance, there is little information and resources in how people utilize GADTs in building and improving real-world applications. In order to understand the potential benefit of using GADTs, I want to use GADTs to improve a classic data structure, AVL tree, and compare the performance.

3. Methods

In this section, I describe the design and techniques of three GADTs applications I look into.

3.1. Simple Lambda Calculus

A canonical example of demonstrating how to use GADTs is re-implementing a simple lambda calculus language.

3.1.1 Non-GADTs Approach

Without GADTs, we can define the language as below:

```
type typ = | Boolean | Integer
| Arrow of typ * typ
type exp = | And of exp * exp
| Add of exp * exp
| App of exp * exp
| Lam of string * typ * exp
| Var of string
| Int of int
| Bool of bool

let e1=Add(Int 0, Add(Int 1, Int 2))
let e2=And(Bool false, Bool true)
let e3=App(Lam("x",Integer,
           Add(Var "x",Var "x")), Int 1)
let e4=Add(Int 1, Bool true) (* Error *)
```

In this language, e_1, e_2, e_3 are syntactically correct expressions, but e_2 should be invalid because we want `And` to be applied to two `Bool` and `Add` to be applied to two `Int`. Therefore, when writing the `eval` function, we need to check the types of the operants and throw error with unexpected types.

```
let rec eval env e =
match e with
| Add(e1, e2) ->
    let v1 = eval env e1 in
    let v2 = eval env e2 in (
    match (v1, v2) with
    | (Integer i1, Integer i2)
     -> Integer (i1 + i2)
    | _ -> failwith "Integer Type Error"
    )
......
```

3.1.2 GADTs Approach

The problem with non-GADTs is that, since both `Int` and `Bool` are of type `exp`, we don't know how to distinguish the two before run time. GADTs solves the probelm by restricting the type parameter of a type constructor.

```
type (_,_) exp =
| Int : int -> ('e, int) exp
| Bool : bool -> ('e, bool) exp
| Add : ('e, int) exp * ('e, int) exp
-> ('e, int) exp
| And : ('e, bool) exp * ('e, bool) exp
```

```
 -> ('e, bool) exp
| App : ('e, ('a->'b)) exp * ('e, 'a) exp
 -> ('e, 'b) exp
| Lam : (('a * 'e), 'b) exp
 -> ('e, ('a -> 'b)) exp
| Var0 : (('a * 'e), 'a) exp
| VarS : ('e, 'a) exp -> (('b*'e), 'a) exp
```

By simply listing the type signatures of all the constructors, we can let the compiler figure out the correct arguments for us. For example, when an `Add` type appears, we know that the two arguments would be an integer instead of boolean.

```
let rec eval: type e t.e->(e,t)exp->t
= fun env e ->
match e with
| Add (e1,e2) ->
 let v1 = eval env e1 in
 let v2 = eval env e2 in
 v1 + v2
......
```

We can implement an evaluation function as above that takes advantage of the type marker. Notice that, this time, we don't need to check the types of e_1 and e_2 when evaluating `Add`.

3.2. Typed LR Parser

In this section I will discuss the setup of LR parser and show how the GADTs design can improve the non-GADTs approach.

3.2.1 Grammar

$$
\begin{array}{lrcl}
(1) & E\{x\} + T\{y\} & \rightarrow & E\{x+y\} \\
(2) & T\{x\} & \rightarrow & E\{x\} \\
(3) & T\{x\} * F\{y\} & \rightarrow & T\{x \times y\} \\
(4) & F\{x\} & \rightarrow & T\{x\} \\
(5) & (E\{x\}) & \rightarrow & F\{x\} \\
(6) & \mathbf{int}\{x\} & \rightarrow & F\{x\}
\end{array}
$$

Figure 1: LR parser grammar with semantic actions

As listed in Figure 1, the grammar's tokens are **int, +, *, (** and **)**. The input stream ends by the token **$**. The grammar's nonterminal symbols are E, T and F, which represents expression, term and factor. E is also the starting symbol.

Figure 2 shows the finite deterministic pushdown automation for the grammar. It maintains a current state $s_i, i \in 0...11$ and a operator stack $\sigma ::= \epsilon | \sigma s v$ where ϵ is the empty stack and $\sigma s v$ denotes pushing new state s and value v on to the stack.

State	*action*						*goto*		
	int	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				**acc**			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Figure 2: Pushdown automation table for the expression grammar

3.2.2 Non-GADTs LR Parser

As a simple ML implementation of the automation, we can encode the stack as follows.

$$
\begin{array}{l}
\text{type } stack = \\
\quad | \ SEmpty \\
\quad | \ SP \text{ of } stack \times state \\
\quad | \ SS \text{ of } stack \times state \\
\quad | \ SL \text{ of } stack \times state \\
\quad | \ SR \text{ of } stack \times state \\
\quad | \ SI \text{ of } stack \times state \times int \\
\quad | \ SE \text{ of } stack \times state \times int \\
\quad | \ ST \text{ of } stack \times state \times int \\
\quad | \ SF \text{ of } stack \times state \times int
\end{array}
$$

P, S, L, R and I are short for Plus, Star, Left bracket, Right bracket and Int. The type stack carries a tag. The pushdown automation is parameterized by the current state and stack. It may end up returning the final integer result or raising a Syntax Error exception.

For example, if we're currently at state 9 and the next token is **+**. According to Figure 2 we want to reduce by grammar rule (1) $E\{x\} + T\{y\} \rightarrow E\{x+y\}$. The syntactically correct stack must look like {ST, SP, SE}. We can get the operators x, y from the stack, produce a new semantic value $x + y$ and a new stack cell.

This non-GADTs approach models the stack as arbitrary sequences of (state, value) pairs. We can not detect syntax error without actually running the program because we know nothing about what's going to be on the stack. Therefore, in the parser evaluation code, we need to use "match s with" everywhere to dynamically check the stack and raise error when the stack is invalid. We also need to write a lot of error checking code to ensure the matching constructs are *exhaustive*. This is a waste of time and code because by the

design of the automation rules, the valid stacks that do arise at runtime are not arbitrary.

In the next section, I'll show how we can make use of GADTs to get rid of superfluous runtime checks.

3.2.3 GADTs LR Parser

The idea is to associate the state with the expected stack shape. To do so, we associate the type of state with a type variable α which is also the type of the current stack. Figure 3 shows the encoding.

```
(* Types for stack cells . *)
type empty = SEmpty
type α cP = SP of α × α state
type α cS = SS of α × α state
type α cL = SL of α × α state
type α cR = SR of α × α state
type α cI = SI of α × α state × int
type α cE = SE of α × α state × int
type α cT = ST of α × α state × int
type α cF = SF of α × α state × int

(* The type of states . *)
type state : ⋆ → ⋆ where
| S0 : empty state
| S1 : empty cE state
| S2 : ∀α.α cT state
| S3 : ∀α.α cF state
| S4 : ∀α.α cL state
| S5 : ∀α.α cI state
| S6 : ∀α.α cE cP state
| S7 : ∀α.α cT cS state
| S8 : ∀α.α cL cE state
| S9 : ∀α.α cE cP cT state
| S10 : ∀α.α cT cS cF state
| S11 : ∀α.α cL cE cR state
```

Figure 3: GADTs LR parser stack and state encoding

The generalized state type allows the compiler to check the dependency between the current state and current stack. For example, if we are at state $S9$, matching the stack with $ST(SP(SE(stack, s, x), \_), \_, y)$ would always succeed because the compiler knows the shape of the state.

The reduce action now doesn't need to perform runtime check and the parser is well-typed.

3.2.4 GADTs LR Parser Optimization

As mentioned in [3], we can further optimize the code by associating the specific state number with the stack. The reason is that the stacks that do arise at runtime are not random but range over a strict subset, which is given in Figure4. For example, when the automation is in state 5, then we know that the top cell holds a state in subset {0, 4, 6, 7} and a semantic value **int**. The optimized stack and state encodings are shown in Figure5

Stack shape							State
ϵ							0
ϵ	$\{0\}$	E					1
σ	$\{0,4\}$	T					2
σ	$\{0,4,6\}$	F					3
σ	$\{0,4,6,7\}$	(					4
σ	$\{0,4,6,7\}$	**int**					5
σ	$\{0,4\}$	E	$\{1,8\}$	+			6
σ	$\{0,4,6\}$	T	$\{2,9\}$	*			7
σ	$\{0,4,6,7\}$	(	$\{4\}$	E			8
σ	$\{0,4\}$	E	$\{1,8\}$	+	$\{6\}$	T	9
σ	$\{0,4,6\}$	T	$\{2,9\}$	*	$\{7\}$	F	10
σ	$\{0,4,6,7\}$	(	$\{4\}$	E	$\{8\}$	)	11

Figure 4: The LR parser automation invariant

```
type empty = SEmpty
type (α, ρ) cP = SP of α × (α, ρ) state
type (α, ρ) cS = SS of α × (α, ρ) state
type (α, ρ) cL = SL of α × (α, ρ) state
type (α, ρ) cR = SR of α × (α, ρ) state
type (α, ρ) cI = SI of α × (α, ρ) state × int
type (α, ρ) cE = SE of α × (α, ρ) state × int
type (α, ρ) cT = ST of α × (α, ρ) state × int
type (α, ρ) cF = SF of α × (α, ρ) state × int

type state : (⋆, row) → ⋆ where
| S0 : (empty, {0}) state
| S1 : ∀γ̄.((empty, ⟨0⟩) cE, {1}) state
| S2 : ∀αγ̄.((α, ⟨0, 4⟩) cT, {2}) state
| S3 : ∀αγ̄.((α, ⟨0, 4, 6⟩) cF, {3}) state
| S4 : ∀αγ̄.((α, ⟨0, 4, 6, 7⟩) cL, {4}) state
| S5 : ∀αγ̄.((α, ⟨0, 4, 6, 7⟩) cI, {5}) state
| S6 : ∀αγ̄.(((α, ⟨0, 4⟩) cE, ⟨1, 8⟩) cP, {6}) state
| S7 : ∀αγ̄.(((α, ⟨0, 4, 6⟩) cT, ⟨2, 9⟩) cS, {7}) state
| S8 : ∀αγ̄.(((α, ⟨0, 4, 6, 7⟩) cL, ⟨4⟩) cE, {8}) state
| S9 : ∀αγ̄.((((α, ⟨0, 4⟩) cE, ⟨1, 8⟩) cP, ⟨6⟩) cT, {9}) state
| S10 : ∀αγ̄.((((α, ⟨0, 4, 6⟩) cT, ⟨2, 9⟩) cS, ⟨7⟩) cF, {10}) state
| S11 : ∀αγ̄.((((α, ⟨0, 4, 6, 7⟩) cL, ⟨4⟩) cE, ⟨8⟩) cR, {11}) state

val run : ∀αρ.(α, ρ) state → α → int
val gotoE : ∀αγ̄.(α, ρ) state → (α, ρ) cE → int
                                  where ρ = ⟨0, 4⟩
```

Figure 5: Optimized GADTs LR parser stack and state encoding

3.2.5 Comparison Experiment Setup

I implement all three versions of LR parser: non-GADTs, GADTs without optimization, GADTs with optimization. The evaluation programs for those three parsers are almost identical. Only the type annotations carried by the function changes. Therefore, it will be a fair comparison between GADTs and non-GADTs.

Firstly I compare the verboseness and number of "non-exhaustive" warnings of three approaches. Then I randomly generate expressions with around 4000, 5000, 6000, 6500, 7600, 9000 tokens (Int, Add, Multiply, Left/Right Brackets). The tokens and the integer values are randomly chosen. For each expression I run each of the program 10 times and record the average running time. The comparison re-

sults are shown in section 4.1.

3.3. AVL Tree

In this section, I'll discuss how to use GADTs to improve the performance of the classic AVL Tree.

3.3.1 Non-GADTs AVL Tree

The most important feature of an AVL is that the height of left subtree and right subtree differs at most by 1. To maintain this feature, we need to keep track of the balance factor (height of right subtree minus height of left subtree) whenever we insert a node.

In a typical non-GADTs implementation, each node is associated with the height. Then in each insertion, we compute the balance factor and if imbalanced, we do the re-balancing by rotating left or rotating right.

```
type t = Leaf
 | Node of t * int * t * height
let height = function
| Leaf -> 0
| Node (_, _, _, h) -> h
```

In this non-GADTs implementation, we assume that the height of left and right subtree can be any non-negative integer. However, since we are re-balancing the AVL tree for each insertion, we know that there are only three cases for the height of left/right subtree: $h_L = h_R - 1$, $h_L = h_R$ or $h_L = h_R + 1$.

We can utilize this information to do smarter re-balancing without frequently computing the difference of the heights.

3.3.2 GADTs AVL Tree

Using GADTs, we can define the nodes in AVL tree as follows:

```
type z = Z : z
type 'd s = S : 'd -> 'd s
type (_, _, _) diff =
| Less : ('d, 'd s, 'd s) diff
| Same : ('d, 'd, 'd) diff
| More : ('d s, 'd, 'd s) diff

type ('a,'d) tree =
| Empty : ('a,z) tree
| Tree  : ('a,'m) tree * 'a * ('a,'n) tree
* ('m,'n,'o) diff -> ('a,'o s) tree
```

In line 8, 'a is the type of the element (int, bool etc), 'd is the depth of the tree which is represented by a sequence S. With the new depth representation, we can easily define the three cases $h_L = h_R - 1$, $h_L = h_R$ or $h_L = h_R + 1$ as `Less`, `Same` and `More` respectively (line 4-6). The three arguments for `diff` represents height for left subtree, right subtree and current node. Lastly, for each tree node, we maintain a `diff` for left/right subtree height difference.

```
let rec run: (* Function signature *) =
fun s l stack ->
match s, (peek l) with
| ...
| S2, KPlus ->
 let ST(stack,s,x)=stack in
    gotoE s l (SE(stack, s, x))
| S2, KStar ->
 run S7 (discard l) (SS(stack, s))
| ...
```

Figure 6: Implementation for the LR Parser Pushdown Automation

Thanks to GADTs, during insertion, we can have a good understanding of how the local tree looks like without calculating the balancing factor. This means we can directly match on the `diff` for each node and do smarter rotation based on the `diff` for left and right subtrees.

3.3.3 Comparison Experiment Setup

I implement the insertion function for both non-GADTs AVL and GADTs AVL. To compare the performance, I generate lists of 1000, 10000, 100000, 150000, 200000, 250000 random integers and insert them into the AVL tree. I run each setup 10 times and the average total insertion time is reported. The results are listed in section 4.2.

4. Results

In this section, I'll report the experiment results as well as present the formal semantics for using GADTs in the lambda calculus and a proof of progress and preservation.

4.1. LR Parser Performance Comparison

I compare the performance of non-GADTs, GATDs, optimized GATDs LR parsers. To make this a fair comparison, the function for running the automation is almost the same except for the function type signatures. The evaluation code looks like Figure 6.

4.1.1 Verbosity

Firstly, I compare the number of "non-exhaustive" match warnings in three programs.

In non-GADTs LR parser, whenever we want to use `let` to match the shape of the stack (like line 6 in Figure 6), the compiler will complain that "the pattern-matching is not exhaustive".

In fact, using the same block of code, non-GADTs method generates 26 warnings when compiling. GADTs without optimization generates only 3 non-exhaustive warnings in the `gotoE`, `gotoT`, `gotoF` functions because we do not know the states that are associated with E, T and F

as listed in Figure 4. With the optimization mentioned in section 3.2.4, there will be 0 warning, which means all the pattern-matchings are exhaustive.

This shows that GADTs is able to help programmers get rid of many superfluous runtime checks and unnecessary warnings. By associating the stack cell with type variables, we can keep track of the stack's structure by just examining the current state. As a result, the compiler knows that the "match" and "let" cases could not fail.

Thanks to the new type information, those formerly non-exhaustive pattern matching construct has become exhaustive and this will allow the compiler to produce better machine code without a runtime check, even though the source code is almost the same. The GADTs method yields better efficiency for programming, and more importantly, stronger correctness is guaranteed.

4.1.2 Performance

To compare the runtime performance, I randomly generate lists of expressions with around 4000, 5000, 6000, 6500, 7600, 9000 tokens (Int, Add, Multiply, Left/Right Brackets). The average running time for non-GADTs, GADTs and GADTs with optimization are shown in Figure 7.

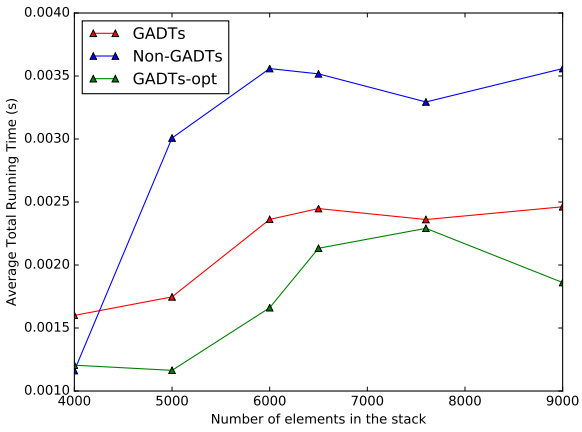

Figure 7: Performance Comparison Between non-GADTs and GADTs LR Parser

From the graph we can see that when the number of tokens is large, GADTs methods outperform non-GADTs methods. The speed up is around $\times 1.5$ from non-GADTs to GADTs. The optimized GADTs methods is able to run even faster, although the gap between optimized and non-optimized GADTs is small.

4.2. AVL Tree Performance Comparison

In this section, I will show the results for comparing the insertion performance of GADTs and non-GADTs AVL trees. I generate lists of 1000, 10000, 100000, 150000, 200000, 250000 random integers and insert them into those AVL trees. The average total runtime is shown in Figure 8.

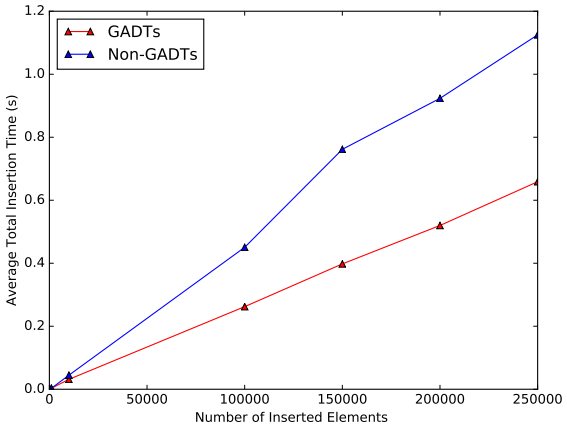

Figure 8: Performance Comparison Between non-GADTs and GADTs AVL Tree Node Insertion

From the graph we can see that GADTs consistently outperforms non-GADTs AVL tree. With the increase in number of inserted elements, the gap between GADTs and non-GADTs is widening. The average speedup is around $\times 1.7$.

As discussed in section 3.3.2, the reason why GADTs is able to run much faster is that we can avoid computing the balance factor for each insertion. GADTs enables us to associate `diff` with every node and `diff` is exactly the information we need to know while re-balancing.

This result shows that GADTs method can be very powerful in improving the performance of real-world applications, especially for large-scale systems.

4.3. Semantics of GADTs in the lambda calculus

In this section I present the formal semantics of using GADTs for explicitly typed lambda calculus.

4.3.1 Syntax

types	$\sigma ::= \tau \mid Int \mid \sigma_1 \times \sigma_2 \mid \sigma_1 \to \sigma_2 \mid T(\vec{\sigma}) \mid \forall \alpha{:}\kappa.\sigma$
expressions	$e ::= i \mid x \mid \lambda x{:}\sigma.e \mid e[\tau] \mid \langle e_1, e_2\rangle \mid e_1 e_2 \mid \textbf{fst}(e) \mid \textbf{snd}(e) \mid \Lambda\alpha{:}\kappa.e \mid C[\overline{\tau}]\overline{e}$
values	$v ::= i \mid \lambda x{:}\sigma.e \mid \langle e_1, e_2\rangle \mid \Lambda\alpha{:}\kappa.v \mid C[\overline{\tau}]\overline{e}$
kinds	$\kappa ::= \star \mid \kappa \to \kappa$
constructors	$\tau ::= \alpha \mid \tau_1\ \tau_2 \mid Int \mid\ \to\ \mid\ \times\ \mid T$
equations	$\epsilon ::= \tau_1 \equiv \tau_2$
type contexts	$\Delta ::= \cdot \mid \Delta, \alpha{:}\kappa$
expression contexts	$\Gamma ::= \cdot \mid \Gamma, x{:}\sigma$
equation contexts	$\Psi ::= \cdot \mid \Psi, \epsilon{:}\kappa$
substitutions	$\Theta ::= \cdot \mid \Theta, \alpha \equiv \tau$
datatype contexts	$\Sigma ::= \cdot \mid \Sigma; \textbf{type}\ T\ \overline{\alpha{:}\kappa} = \Sigma_T$
datatype signatures	$\Sigma_T ::= \cdot \mid \Sigma_T \mid \exists\overline{\beta{:}\kappa}.C\overline{\sigma}\ \textbf{with}\ \overline{\epsilon{:}\kappa}$

4.3.2 Typing Rules

$\boxed{\Delta; \Psi; \Lambda \vdash e : \sigma}$

$$\frac{}{\Delta;\Psi;\Gamma \vdash i : Int} \qquad \frac{}{\Delta;\Psi;\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Delta;\Psi;\Gamma, x{:}\sigma \vdash e : \sigma'}{\Delta;\Psi;\Gamma \vdash \lambda x{:}\sigma.e : \sigma \to \sigma'}$$

$$\frac{\Delta;\Psi;\Gamma \vdash e : \sigma_1 \quad \Delta;\Psi \vdash \sigma_1 \equiv \sigma_2}{\Delta;\Psi;\Gamma \vdash e : \sigma_2}$$

$$\frac{\Delta;\Psi;\Gamma \vdash e_1 : \sigma \to \sigma' \quad \Delta;\Psi;\Gamma \vdash e_2 : \sigma}{\Delta;\Psi;\Gamma \vdash e_1 e_2 : \sigma'}$$

$$\frac{\Delta, \alpha{:}\kappa;\Psi;\Gamma \vdash e : \sigma}{\Delta;\Psi;\Gamma \vdash \Lambda\alpha{:}\kappa.e : \forall\alpha{:}\kappa.\sigma}$$

$$\frac{\Delta;\Psi;\Gamma \vdash e : \forall\alpha{:}\kappa \quad \Delta \vdash \tau : \kappa}{\Delta;\Psi;\Gamma \vdash e[\tau] : \sigma[\tau/\alpha]}$$

$$\frac{\Delta;\Psi;\Gamma \vdash e_1 : \sigma_1 \quad \Delta;\Psi;\Gamma \vdash e_2 : \sigma_2}{\Delta;\Psi;\Gamma \vdash \langle e_1, e_2\rangle : \sigma_1 \times \sigma_2}$$

$$\frac{\Delta;\Psi;\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Delta;\Psi;\Gamma \vdash \textbf{fst}(e) : \sigma_1} \qquad \frac{\Delta;\Psi;\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Delta;\Psi;\Gamma \vdash \textbf{snd}(e) : \sigma_2}$$

$$\frac{\Sigma_{T.C\overline{\alpha}} = \exists\overline{\beta{:}\kappa}.C\overline{\sigma}\ \textbf{with}\ \overline{\epsilon{:}\kappa'} \quad \Delta;\Psi;\Gamma \vdash e_i : \sigma_i[\overline{\tau}'/\overline{\alpha}, \overline{\tau}/\overline{\beta}]}{\Delta;\Psi \vdash \epsilon_i[\overline{\tau}'/\overline{\alpha}, \overline{\tau}/\overline{\beta}] : \kappa_i' \qquad \Delta \vdash \tau_i : \kappa_i}{\Delta;\Psi;\Gamma \vdash C[\overline{\tau}]\overline{e} : T\overline{\tau}'}$$

4.3.3 Type Soundness

Progress *If* $\cdot;\cdot;\cdot \vdash e : \tau$ *then either* e *is a value or there exists* e' *such that* $e \mapsto e'$

Proof. Let $\gamma : \Gamma$ represent that γ is a function mapping variables bound in Γ to values such that $\cdot;\cdot;\cdot \vdash \gamma(x) : \Gamma(x)$, $\delta : \Delta; \Psi$ represent that δ maps type variables bound in Δ to type constructors of appropriate kinds so that Ψ is satisfied: $\cdot \vdash \delta(\alpha) : \Delta(\alpha)$ and $\cdot;\cdot \vdash \delta(\tau) \equiv \tau' \in \Psi$.

Let $\delta(\sigma)$ and $\gamma\delta(e)$ represent the result of substituting all type or term variables in a type σ or expression e with their values in γ or δ.

Then the induction hypothesis is generalized to: If $\delta; \Psi; \Gamma \vdash e : \tau$ and $\gamma : \Gamma, \delta : \Delta; \Psi$ then either $\gamma\delta(e)$ is a value or there exists e' such that $\gamma\delta(e) \mapsto \gamma\delta(e')$. Proof is by induction on the derivation of $\Delta; \Psi; \Gamma \vdash e : \sigma$.

case e is i or $\lambda x{:}\sigma.e_1$ **or** $\langle p_1, p_2\rangle$ **or** $C[\overline{\tau}]\overline{e}$**:** Then e is a value, as expected.

case e is fst(e) *or* **snd**(e): Then we can follow the typing rules to find $e \to e'$

case e is x: This is not possible because we would have $\vdash x : \tau$ and from the empty environment we cannot assign any type to x.

case e is $e_1[\tau]$**:** By the typing rules, the normal derivation ends in a use of $\frac{\Delta;\Psi;\Gamma \vdash e : \forall\alpha{:}\kappa \quad \Delta \vdash \tau : \kappa}{\Delta;\Psi;\Gamma \vdash e[\tau] : \sigma[\tau/\alpha]}$. Hence, $\vdash e_1 : \forall\alpha{:}\kappa$ and $\vdash \tau : \kappa$. Since type application is an evaluation context, e_1 is irreducible. Hence by induction hypothesis, e_1 is a value. Since e is irreducible, e_1 is not a type abstraction. Then e_1 has a polymorphic type which is an application to types. Therefore e is a value.

case e is $e_1 e_2$ By the typing rules, the normal derivation ends in a use of $\frac{\Delta;\Psi;\Gamma \vdash e_1 : \sigma \to \sigma' \quad \Delta;\Psi;\Gamma \vdash e_2 : \sigma}{\Delta;\Psi;\Gamma \vdash e_1 e_2 : \sigma'}$. Hence $\vdash e_1 : \sigma \to \sigma'$ and $\vdash e_2 : \sigma$. By induction hypothesis, e_1 is value or $\exists e_1'.e_1 \to e_1'$, e_2 is value or $\exists e_2'.e_2 \to e_2'$. If e_1 or e_2 is not a value, we would have $e \to e_1'\ e_2$ or $e \to e_1\ e_2'$. Suppose both e_1 and e_2 are values. Since e_1 has an arrow, it has to be a type abstraction. Say e_1 is $\lambda x \in \sigma'.e_3$ and e_2 is some value v. Then $e = (\lambda x \in \sigma'.e_3)v \to e_3[v/x]$

case e is $\Lambda\alpha : \kappa.e_1$**:** By the typing rules, the derivation ends in $\frac{\Delta,\alpha{:}\kappa;\Psi;\Gamma \vdash e : \sigma}{\Delta;\Psi;\Gamma \vdash \Lambda\alpha{:}\kappa.e : \forall\alpha{:}\kappa.\sigma}$. Therefore, $\vdash e_1 : \sigma$. Since type abstraction is an evaluation context, e_1 is irreducible. Hence, by induction hypothesis, e_1 is a value and e is also a value.

Preservation *If* $\cdot;\cdot;\cdot \vdash e : \sigma$ *and* $e \mapsto e'$ *then* $\cdot;\cdot;\cdot \vdash e' : \sigma$

Proof. The induction hypothesis is strengthened to: If $\Delta; \Psi; \Gamma \vdash e : \sigma$ and $\delta : \Delta; \Psi$ and $\gamma : \Gamma$, and $\gamma\delta(e) \mapsto \gamma\delta(e')$ then $\Delta; \Psi; \Gamma \mapsto e' : \sigma$. Since evaluation contexts preserve typing, it suffices to consider only the reducible expressions.

case e is $(\lambda x{:}\sigma.e_1)v$: Then e' is $e_1[v/x]$, The typing derivation of $e : \sigma$ must look like

$$\frac{\frac{x{:}\tau' \vdash e_1 : \sigma}{(\lambda x{:}\sigma.e_1) : \sigma' \to \sigma} \quad \vdash v : \sigma'}{\vdash (\lambda x{:}\sigma.e_1)v : \sigma}$$

we need to show that $e_1[v/x] : \sigma$ using the facts that $x :$

$\sigma' \vdash e_1 : \sigma$ and $\vdash v : \sigma'$. This is proved by the substitution lemma.

case e is $e_1 e_2$: According to the typing rules, we must have $\vdash e_1 : \sigma' \to \sigma$ and $\vdash e_2 : \sigma'$ for some type σ'.

If e_1 is reducible. Because the typing derivation for e_1 is a subderivation of the typing derivation, by induction hypothesis, $e_1 \mapsto e_1'$ also preserves type. So we know $\vdash e_1' : \sigma' \to \sigma$ and therefore $e' = e_1' e_2$ has the desired type σ. The proof is the same when e_2 is reducible, where we would have $e' = e_1 e_2' : \sigma$.

case e is $(\Lambda\alpha{:}\kappa.e_1)[\tau]$: Then e' is $e_1[\tau/\alpha]$. From the typing derivation for $\vdash e : \sigma$, we have $\vdash \Lambda\alpha{:}\kappa.e_1 : \forall\alpha{:}\kappa.\sigma'$ and $\sigma = \sigma'[\tau/\alpha]$.

By the type substitution lemma (If $\Delta; \alpha{:}\kappa; \Psi; \Gamma : e : \sigma$ then $\Delta[\tau/\alpha]; \Psi[\tau/\alpha]; \Gamma[\tau/\alpha] : e[\tau/\alpha] : \sigma[\tau/\alpha]$), we would have $e_1[\tau/\alpha] : \sigma'[\tau/\alpha]$. Therefore, $e' : \sigma$

5. Conclusion

In this paper, I discuss how generalized algebraic data types can improve the efficiency of LR parser, AVL tree and more importantly, maintain a stronger correctness guarantee. Experiments show that GADTs not only helps programmer get rid of redundant dynamic checks, but also improves the speed of the program. Moreover, the code is safe and well-typed. In short, this project is a good illustration of the new expressiveness offered by generalized algebraic data types.

References

[1] J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[2] D. Leijen and E. Meijer. Domain specific embedded compilers. In *ACM Sigplan Notices*, volume 35, pages 109–122. ACM, 1999.

[3] F. Pottier and Y. Régis-Gianas. Towards efficient, typed lr parsers. *Electronic Notes in Theoretical Computer Science*, 148(2):155–180, 2006.

[4] B. Vaugon. A new implementation of ocaml formats based on gadts, 2013.

[5] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *ACM SIGPLAN Notices*, volume 38, pages 224–235. ACM, 2003.