Formal foundations for GADTs in Scala

Radosław Waśko

January 16, 2020

Overview

What is a GADT? ADTs in Scala

One can define an ADT using the enum syntax, for example:

```
enum List[A] {
  case Nil()
  case Cons(head: A, tail: List[A])
}
```

which is roughly equivalent to:

```
sealed trait List[A]
case class Nil[A]() extends List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
```

What do the GADTs add?

We can add additional type constraints for particular cases

```
enum Container[A] {
  case Characters(data: String) extends Container[Char]
  case Other(data: List[A])
}

def head[A](c: Container[A]): A = c match {
  case Container.Characters(data) =>
    data.head // typechecks
  case Container.Other(data) => data.head
}

Characters("abc") : Container[Char]
```

What do the GADTs add?

Another feature is existential types: see `Pair`

```
enum Expr[A] {
  case Lit(n: Int) extends Expr[Int]
  case Plus(lhs: Expr[Int], rhs: Expr[Int])
       extends Expr[Int]
  case Pair[B,C](lhs: Expr[B], rhs: Expr[C])
       extends Expr[(B,C)]
}

def eval[A](e: Expr[A]): A = e match {
  case Lit(n) => n
  case Plus(lhs, rhs) => eval(lhs) + eval(rhs)
  case Pair(lhs, rhs) => (eval(lhs), eval(rhs))
}
```

How to encode GADTs in DOT?

We need to figure out how to encode:

- Pattern Matching
- Type Constraints

We could extend DOT with these features but this would require redoing the soundness proof.
Another approach is to encode them using existing features.

Pattern matching

Most common ways to encode an ADT and pattern matching it in a lambda calculus are:

- Böhm-Berarducci Encoding
- Scott Encoding

Pattern matching - Böhm-Berarducci Encoding

$$\texttt{List} \equiv \forall\alpha.(\texttt{Int} \to \alpha \to \alpha) \to \alpha \to \alpha$$

```
sum = λlist: ∀α.(Int → α → α) → α → α.
  list
    [Int]
    (λelem: Int. λaccumulator: Int.
      elem + accumulator)
    0
```

Pattern matching - Scott Encoding

$$\texttt{List} \equiv \mu t.\forall\alpha.(\texttt{Int} \to t \to \alpha) \to \alpha \to \alpha.$$

```
sum = fix f: List → Int. λlist: List.
  list
    (λhead: Int. λtail: List. head + f tail)
    (0)
```

- requires recursive types
- is much closer to simple pattern matching

Type equalities

Looking at the previously mentioned example

```
def eval[A](e: Expr[A]): A = e match {
  case Lit(n) => n
  case Plus(lhs, rhs) => eval(lhs) + eval(rhs)
  case Pair(lhs, rhs) => (eval(lhs), eval(rhs))
}
```

In the last case, we return `(B,C)` and need to show it is equal to `A`.

Type equalities

We use singleton types to encode the type equalities.

```
abstract class Expr { self =>
  type A
  def pmatch[R](pair: self.type & Pair => R, ...): R
}
type Expr'[T] = Expr { type A = T }
class Pair extends Expr { self =>
  type B
  type C
  type A = (B,C)
  val data: (Expr'[B], Expr'[C])
  def pmatch[R](pair: (self.type & Pair) => R, ...): R =
     pair(self)
}
```

Type equalities

```
def eval(e: Expr): e.A =
  e.pmatch[e.A](
    (pair: e.type & Pair) =>
      val lhs = eval(pair.data.fst) // pair.B
      val rhs = eval(pair.data.snd) // pair.C
      (lhs, rhs) // (pair.B,pair.C) = pair.A = e.A
    ... other cases
  )
```

We know that `pair.A` is equal to `(pair.B,pair.C)` from defintion of `Pair`.
As `pair: e.type` we can use `e` and `pair` interchangeably in types, so we also get that `pair.A` is the same as `e.A`.

Encoding examples

We show two examples of encoding a GADT programs in pDOT:

- the mentioned Expr type and its eval function
- a type-safe STLC interpreter using DeBruijn indices

Sketch of a general encoding

Xi et al. define a general GADT type as:

$$T \equiv \mu t.\lambda\overline{\alpha}.\,(\exists[\overline{\alpha_1}, \overline{\sigma_1} = \overline{\alpha}].\tau_1 + \cdots + \exists[\overline{\alpha_n}, \overline{\sigma_n} = \overline{\alpha}].\tau_n)$$

Such types can be encoded in pDOT as:

```
T = μ(s: {
  α1: ⊥..⊤; ... αm: ⊥..⊤
  pmatch: ∀(r: {R:⊥..⊤})
            ∀(c1: ∀(arg: env.TC1 ∧ s.type) r.R)
            ...
            ∀(cn: ∀(arg: env.TCn ∧ s.type) r.R)
            r.R
})
```

Sketch of a general encoding

Each case is then encoded as:

$$
\begin{aligned}
&TC_i = \mu(s: \mathsf{env}.T \wedge \{ \\
&\quad \beta_{i,1}: \bot..\top; \ \ldots; \ \beta_{i,m_i}: \bot..\top \\
&\quad \alpha_1 = \sigma_{i,1}; \ \ldots; \ \alpha_m = \sigma_{i,m} \\
&\quad \mathsf{data}: \tau_i \\
&\})
\end{aligned}
$$

With a constructor of type:

$$
\begin{aligned}
&c_i: \forall(\mathsf{types}: \{\beta_{i,1}: \bot..\top; \ \ldots \ \beta_{i,m_i}: \bot..\top\}) \ \forall(v: \tau_i) \\
&\quad \mathsf{env}.TC_i \wedge \{\beta_{i,1} = \mathsf{types}.\beta_{i,1}; \ \ldots; \ \beta_{i,m_i} = \mathsf{types}.\beta_{i,m_i}\}
\end{aligned}
$$

(we skip the constructor term here because it's very big)

Moving towards encoding a GADT calculus in pDOT

To show that we can fully encode GADT reasoning in pDOT, we can encode some existing calculus that is well-known to support GADT reasoning.
We show how to encode types and terms of $\lambda_{2,G\mu}$ calculus defined in the paper *Guarded Recursive Datatype Constructors* by Xi et al.
The encoding takes as input type derivations of terms in $\lambda_{2,G\mu}$ and returns pDOT terms.

Challenges encoding $\lambda_{2,G\mu}$

- pattern matching
  - nested patterns
  - non-exhaustivity
  - non-determinism
- nominality
- contradictory type equalities

Project outcomes

- encoding $\lambda_{2,G\mu}$ calculus in pDOT (but without proofs)
- side results:
  - comparing methods of encoding pattern matching
  - encoding selected GADT program examples in pDOT
  - sketching a general encoding scheme of a GADT type

Future work

- Formalizing the encoding
  - Proving that the encoded terms are well-typed in pDOT
  - Proving that semantics of the encoded terms is analogous to original semantics
  - Dealing with simplifications of pattern matching
  - Dealing with contradictions
- Subtyping and variance

Thank you :)