Formal foundations for GADTs in Scala

Radosław Waśko

11 maja 2020

Outline

1. Czym są GADT?
2. Enkodowanie GADT w Scali
3. pDOT
4. Enkodowanie GADT w pDOT

Czym są GADT? Przykłady

GADT pozwalają nam definiować dodatkowe więzy na typach zawartych w strukturze.
Na przykład - otypowane drzewa wyrażeń

```
{-# LANGUAGE GADTs #-}
data Expr a where
    Lit  :: Int  -> Expr Int
    Plus :: Expr Int -> Expr Int -> Expr Int
    Pair :: Expr b -> Expr c -> Expr (b, c)

eval :: Expr a -> a
eval (Lit x) = x -- here a =:= Int
eval (Plus lhs rhs) = (eval lhs) + (eval rhs)
eval (Pair lhs rhs) = (eval lhs, eval rhs) -- a =:= (b, c)

eval (Pair (Plus (Lit 40) (Lit 2)) (Lit 23)) // (42, 23)
```

GADT w Scali - przykłady

GADT definiujemy tak jak ADT dodając więzy typowe poprzez klauzulę `extends`.
W Scali 3 (dotty):

```
enum Expr[A] {
  case Lit(x: Int) extends Expr[Int] // A =:= 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)]
}
```

W Scali 2:

```
sealed trait Expr[A] {}
case class Lit(x: Int) extends Expr[Int] // A =:= Int
case class Plus(lhs: Expr[Int], rhs: Expr[Int]) extends Expr[Int]
case class Pair[B,C](lhs: Expr[B], rhs: Expr[C]) extends Expr[(B, C)]
```

GADT w Scali - przykłady

```
enum Expr[A] {
  case Lit(x: 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(x) => x // A =:= Int
  case Plus(lhs, rhs) => eval(lhs) + eval(rhs) // A =:= Int
  case Pair(lhs, rhs) => (eval(lhs), eval(rhs)) // A =:= (B, C)
}

eval(Pair(Plus(Lit(40), Lit(2)), Lit(23))) // (42, 23)
```

Definicja GADT

```
enum Expr[A] {
  case Lit(x: Int) extends Expr[Int] // A =:= 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)]
}
```

GADT cechują się:

- możliwością definiowania więzów na typach
- rekurencyjnymi odwołaniami
- typami egzystencjalnymi

Definicja GADT

Zatem GADT możemy zdefiniować w rachunku z typami egzystancjalnymi i rekurencyjnymi oraz więzami typowymi. Ogólny typ GADT według Xi et al.[1]:

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

[1] w *Guarded recursive datatype constructors* [1]

Definicja GADT

Łatwiej będzie tę ogólną definicję zrozumieć na przykładzie:
Weźmy poprzedni przykład:

```
enum Expr[A] {
  case Lit(x: 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)]
}
```

$$\begin{aligned} Expr \equiv \mu t.\lambda A.(&\exists[A = Int].\ Int \\ &+\exists[A = Int].\ t\ A * t\ A \\ &+\exists[B, C, A = B * C].\ t\ B * t\ C) \end{aligned}$$

Enkodowanie GADT

Scala od dawna wspiera typy egzystencjalne i rekurencyjne.
Spróbujmy zakodować więzy typów.

Enkodowanie więzów - funkcje

Pierwszym przybliżeniem kodowania więzów mogą być funkcje[2]:

```scala
sealed trait Eq[A,B] {
  def to(x: A): B
  def from(y: B): A
}

case class Refl[A]() extends Eq[A,A] {
  def to(x: A): A = x
  def from(y: A): A = y
}
```

[2] Pomysł oparty na pracy *GADTs for the OCaml masses* [2]

Enkodowanie więzów - funkcje

Instancja klasy `Eq` służy nam za świadka, że typy `A` i `B` są równe.

```
trait Eq[A,B] {
  def to(x: A): B
  def from(y: B): A
}

sealed trait Expr[A]
case class LitC[A](x: Int, ev: Eq[A, Int]) extends Expr[A]
case class PlusC[A](lhs: Expr[Int], rhs: Expr[Int], ev: Eq[A, Int]) extends Expr[A]
case class PairC[A,B,C](lhs: Expr[B], rhs: Expr[C], ev: Eq[A, (B,C)]) extends Expr[A]

def Lit(x: Int): Expr[Int] = LitC(x, Refl())
def Plus(lhs: Expr[Int], rhs: Expr[Int]): Expr[Int] = PlusC(lhs, rhs, Refl())
def Pair[B,C](lhs: Expr[B], rhs: Expr[C]): Expr[(B,C)] = PairC(lhs, rhs, Refl())

def eval[A](e: Expr[A]): A = e match {
  case LitC(x, ev) => ev.from(x)
  case PlusC(lhs, rhs, ev) => ev.from(eval(lhs) + eval(rhs))
  case PairC(lhs, rhs, ev) => ev.from((eval(lhs), eval(rhs)))
}
```

W miejscu gdzie korzystamy z poszczególnych równości, wstawiamy wywołania odpowiednich funkcji klasy `Eq`.

Enkodowanie więzów - funkcje

Jedną z wad tego kodowania jest jego słaba integracja z innymi elementami systemu typów, jak np. ze strukturami kowariantnymi.

```
def lista[A,B](lst: List[A], ev: Eq[A, B]): List[B] =
  lst.map(ev.to)
```

Ta metoda realizuje równość nie bezpośrednio w systemie typów, ale niejako tworząc konwersje pomiędzy niezwiązanymi typami, które muszą zostać aplikowane do każdego elementu, na który wpływa dana równość.
W szczególności jak widać na powyższym przykładzie, konwertowanie bardziej złożonych struktur może wiązać się z niepotrzebnym przechodzeniem po całej (potencjalnie dużej) strukturze.

Enkodowanie więzów - type members

Kolejne podejście[3] jest podobne do poprzedniego, ale działa już bliżej samego systemu typów, a nie runtime'u.

```
trait Eq[A,B] {
  type To >: B <: A
  type From >: A <: B
}

case class Refl[A]() extends Eq[A,A] {
  type To = A
  type From = A
}
```

[3] Bazuje na *Towards improved GADT reasoning in Scala* [3]

Enkodowanie więzów - type members

Korzystamy tu z faktu, że konkretne instancje klasy mogą zawierać tylko członki typowe o niesprzecznych więzach.

```
trait Contradiction {
  type C >: String <: Int
}

object X extends Contradiction // Error
// object X cannot be instantiated since
// it has a member C with possibly conflicting bounds
// String <: ... <: Int
```

Enkodowanie więzów - type members

Konstruktory pozostają dokładnie takie same jak w poprzednim podejściu. Zmienia się sposób w jaki korzystamy ze świadków.

```
def eval[A](e: Expr[A]): A = e match {
  case LitC(x, ev) => x: ev.T
  case PlusC(lhs, rhs, ev) => eval(lhs) + eval(rhs): ev.T
  case PairC(lhs, rhs, ev) => (eval(lhs), eval(rhs)): ev.T
}
```

Dodając odpowiednią adnotację typową (nie jest to type-cast) dajemy wskazówkę kompilatorowi. `ev` jest typu `Eq[A, Int]`, zatem `ev.T >: Int <: A`. Kompilator widzi, że skoro `x: Int`, to możemy do niego przypisać typ `ev.T`, który jest nadtypem `Int`, zaś skoro `ev.T` jest podtypem `A`, to możemy użyć wartości `x` tam gdzie oczekujemy wartości typu `A`.

Enkodowanie więzów - type members

To kodowanie jest dobrym punktem wyjścia, w szczególności lepiej radzi sobie z nakreślonym wcześniej problemem złożonych struktur:

```
def lista[A,B](lst: List[A], ev: Eq[A, B]): List[B] =
  lst: List[ev.F]
```

Ta adnotacja typowa jest tylko wskazówką dla kompilatora, nic nas nie kosztuje w runtime.

pDOT

pDOT[4] (Path-dependent Object Types) jest rachunkiem lambda formalizującym podstawy, na których opiera się system typów Scali 3.

Jego najważniejsze cechy:

- type members
- intersection types
- path-dependent types
- singleton types

pozwalają na zakodowanie takich funkcjonalności jak generic classes czy parametric polymorphism.

[4] Opisany w *A path to DOT* [4]

pDOT - wartości

pDOT posiada dwa rodzaje wartości:

1. funkcje

$$\big(\lambda(x : T)\ e\big) : \forall(x : T)\ U$$

```
((x: T) => e) : (T => U)
```

2. obiekty

$$\big(\nu(s : \{A : T..T; x : \texttt{Int}\})\{A = T; x = 2\}\big) : \mu(s : \{A : T..T; x : \texttt{Int}\})$$

```
new { type A; val x: Int } : { type A; val x: Int }
```

Do wartości odwołujemy się przez ścieżki:

$$\texttt{let}\ \ q = \nu(s : \{x : \texttt{Int}\})\{x = 2\}\ \ \texttt{in}\ \ q.x$$

pDOT opiera się na typach strukturalnych, tzn. nie ma rozróżnienia typów ze wzgl. na nazwy (nominality), choć da się ten koncept zakodować.

pDOT - Obiekty

Obiekty składają się z typów i wartości, metody nie są osobną konstrukcją - są to po prostu funkcje przyjmujące jakiś Unit.

```
trait ScalaObj {
  type A
  val x: Int
// x jest obliczony przy konstrukcji i zapamiętany
  def y: Int
// wartość y jest liczona przy każdym odwołaniu
}

trait pDOTObj {
  type A
  val x: Int
  val y: () => Int
}
```

$$T = \{A : \bot..\top;\ x : \mathtt{Int};\ y : \forall(\_ : \top)\ \mathtt{Int}\}$$

pDOT - Type members

```
trait T { type A >: Nothing <: Any }
trait U { type A >: L <: H }
```

$$T = \{A : \bot..\top\}$$

$$U = \{A : L..H\}$$

$$\dfrac{\dfrac{\dfrac{p : \mu(\mathit{self} : \{A : \bot..\mathtt{Int}; x : \mathit{self}.A\})}{p : \{A : \bot..\mathtt{Int}; x : p.A\}}}{p.x : p.A} \qquad p.A <: \mathtt{Int}}{p.x : \mathtt{Int}}$$

pDOT - Singleton types

```
val x = 5
val y: x.type = x // askrypcja jest potrzebna,
                  // bo domyślnie kompilator uogólnia typ
def f(v: x.type) = ...
f(x) // ok
f(y) // ok
f(5) // Type Mismatch
```

$$\begin{array}{l} \texttt{let } x = 5 \texttt{ in} \\ \texttt{let } f = \lambda(v : x.type) \ldots \texttt{ in} \\ \quad f\ x \end{array}$$

Zasada 'wymienności' ścieżek przy typach singletonowych:

$$\frac{\Gamma \vdash p : q.type \qquad q \text{ is typeable}}{\Gamma \vdash T <: T[q/p]}$$

pDOT - Cykliczne odwołania

Umieszczamy definicje wewnątrz modułu

```
object ScalaModule {
  def f(x: Int): Int = g(x)
  def g(x: Int): Int = f(x)
}

object pDOTModule { mod =>
  val f: (Int => Int) = (x: Int) => mod.g(x)
  val g: (Int => Int) = (x: Int) => mod.f(x)
}
```

$$\begin{aligned} \mathit{Module} = \nu(\mathit{mod} : \{ & \\ & f : \forall(x : \mathtt{Int})\,\mathtt{Int};\ g : \forall(x : \mathtt{Int})\,\mathtt{Int} \\ \})\{ & f = \lambda(x : \mathtt{Int})\ \mathit{mod}.g\ x \\ & g = \lambda(x : \mathtt{Int})\ \mathit{mod}.f\ x\} \end{aligned}$$

pDOT - Intersection types

```scala
trait A { def g(v: Int): String }
trait B { val x: Int }
def f(obj: A & B): String = obj.g(obj.x)

f(new A with B {
  def g(v: Int): String = v.toString
  val x: Int = 42
})
```

Subtelna różnica pomiędzy Scalą a pDOT, gdyż Scala ma niepełne wsparcie dla typów strukturalnych.

$$
\begin{aligned}
&A = \{g : \forall(v : Int)\ String\} \\
&B = \{x : Int\} \\
&f : \forall(obj : A \wedge B)\ String \\
&f = \lambda(obj : A \wedge B)\ obj.g\ obj.x \\
&f\ (\nu(s : A \wedge B)\{x = 42; g = ...\})
\end{aligned}
$$

pDOT - Intersection types

$$\frac{S <: T \qquad S <: U}{S <: T \wedge U}$$

$$\frac{S <: A \wedge B}{S <: A}$$

$\{x = 2;\ A = \texttt{Int}\}$ jest lukrem syntaktycznym na $\{x = 2\} \wedge \{A = \texttt{Int}\}$

pDOT - Path-dependent types

```
trait T {
  type A
  val x: A
}
def f(v: T): v.A = v.x
```

$$T = \mu(self : \{A : \bot..\top\} \wedge \{x : self.A\})$$

$$f : \forall(v : T)\ v.A$$

$$f = \lambda(v : T)\ v.x$$

pDOT - przykład: jak zakodować polimorficzną funkcję

```
def id[A](x: A): A = x
id[Int](42)

trait TypeLabel { type Typ }
def id(T: TypeLabel)(x: T.Typ): T.Typ = x
id(new TypeLabel { type Typ = Int })(42)
```

$$TypeLabel = \{Typ : \bot..\top\}$$
$$id : \forall(T : TypeLabel)\ \forall(x : T.Typ)\ T.Typ$$
$$id = \lambda(T : TypeLabel)\ \lambda(x : T.Typ)\ x$$
$$id\ (\nu(\_ : TypeLabel)\{Typ = Int\})\ 42$$

pDOT - przykład: lista

$List \wedge \{A : \bot..T\} = \texttt{List[A]}$, zaś $List = \texttt{List[?]}$

$$
\begin{aligned}
&\nu(\mathbf{sci} \Rightarrow \mathsf{List} = \mu(\mathbf{self}\colon\ \{\mathsf{A}\colon \bot..\top\} \wedge \{\mathsf{head}\colon \forall(\_)\,\mathbf{self}.\mathsf{A}\} \wedge \{\mathsf{tail}\colon \forall(\_)\,(\mathbf{sci}.\mathsf{List} \wedge \{\mathsf{A}\colon \bot..\mathbf{self}.\mathsf{A}\})\});\\
&\quad \mathsf{nil}\colon \forall(x\colon\ \{\mathsf{A}\colon \bot..\top\})\,\mathbf{sci}.\mathsf{List} \wedge \{\mathsf{A}\colon \bot..\bot\}\\
&\qquad = \lambda(x\colon\ \{\mathsf{A}\colon \bot..\top\})\,\mathbf{let}\ \mathsf{result} = \nu(\mathbf{self} \Rightarrow \mathsf{A} = \bot;\\
&\qquad\qquad \mathsf{head}\colon \forall(y\colon \top)\,\mathbf{self}.\mathsf{A} = \lambda(y\colon \top)\,\mathbf{self}.\mathsf{head}\ y;\\
&\qquad\qquad \mathsf{tail}\colon \forall(y\colon \top)\,(\mathbf{sci}.\mathsf{List} \wedge \mathbf{self}.\mathsf{A}) = \lambda(y\colon \top)\,\mathbf{self}.\mathsf{tail}\ y)\\
&\qquad\quad \mathbf{in}\ \mathsf{result};\\
&\quad \mathsf{cons}\colon \forall(x\colon\ \{\mathsf{A}\colon \bot..\top\})\,\forall(\mathsf{hd}\colon x.\mathsf{A})\,\forall(\mathsf{tl}\colon \mathbf{sci}.\mathsf{List} \wedge \{\mathsf{A}\colon \bot..x.\mathsf{A}\})\,\mathbf{sci}.\mathsf{List} \wedge \{\mathsf{A}\colon \bot..x.\mathsf{A}\}\\
&\qquad = \lambda(x\colon\ \{\mathsf{A}\colon \bot..\top\})\,\lambda(\mathsf{hd}\colon x.\mathsf{A})\,\lambda(\mathsf{tl}\colon \mathbf{sci}.\mathsf{List} \wedge \{\mathsf{A}\colon \bot..x.\mathsf{A}\})\\
&\qquad\quad \mathbf{let}\ \mathsf{result} = \nu(\mathbf{self} \Rightarrow \mathsf{A} = x.\mathsf{A};\\
&\qquad\qquad \mathsf{head}\colon \forall(\_)\,\mathbf{self}.\mathsf{A} = \lambda\_.\,\mathsf{hd}\\
&\qquad\qquad \mathsf{tail}\colon \forall(\_)\,(\mathbf{sci}.\mathsf{List} \wedge \mathbf{self}.\mathsf{A}) = \lambda\_.\,\mathsf{tl})\\
&\qquad\quad \mathbf{in}\ \mathsf{result})
\end{aligned}
$$

Przykład zaczerpnięty z [4]

pDOT - przykład: method chaining

```scala
trait C {
  def incr: C
}
trait D extends C {
  def decr: D
}
val x: D = ...
x.decr.incr // ok
x.incr.decr // type error:
// decr is not a member of C
// x.incr: C
```

```scala
trait C { self =>
  def incr: self.type
}
trait D extends C { self =>
  def decr: self.type
}
val x: D = ...
x.decr.incr // ok
x.incr.decr // ok
// x.incr: x.type =:= D
```

Przykład zaczerpnięty z [4]

Enkodowanie ADT w pDOT

Zanim podejmiemy się zakodowania GADT, warto zastanowić się jak możemy kodować zwykłe ADT i pattern matching na nich w rachunku lambda z typami.
Dwa najpopularniejsze podejścia to:

1. Kodowanie Scotta

$$\mathtt{IList} \equiv \mu t.\forall \alpha.(\mathtt{Int} \to t \to \alpha) \to \alpha \to \alpha$$

2. Kodowanie Böhma-Berarducciego

$$\mathtt{IList} \equiv \forall \alpha.(\mathtt{Int} \to \alpha \to \alpha) \to \alpha \to \alpha$$

W obu tych kodowaniach ADT jest reprezentowany przez funkcję, która przyjmuje funkcje określające jak przekształcić poszczególne case'y.

Enkodowanie ADT - kodowanie Scotta

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

```
sum = λlist: IList.
        list
        (λhead: Int. λtail: IList. head + sum tail)
        (0)

tail = λlist: IList.
        list
        (λhead: Int. λtail: IList. tail)
        (nil)
```

Enkodowanie ADT - kodowanie Böhma-Berarducciego

$$\mathtt{IList} \equiv \forall\alpha.(\mathtt{Int} \to \alpha \to \alpha) \to \alpha \to \alpha$$

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

tail = λlist: ∀α.(Int → α → α) → α → α.
          fst (list
               (λh: Int. λacc: list * list.
                              (snd acc, cons h (snd acc)))
               (nil, nil)
             )
```

Enkodowanie GADT w pDOT - typy egzystencjalne

Typy egzystencjalne również można zakodować za pomocą type memberów, jak już widzieliśmy na przykładzie listy.
Weźmy typ $T = \exists \alpha.(\mathtt{Int} \to \alpha) * (\alpha \to \mathtt{String})$.

```
trait T {
  type A
  def f: Int => A
  def g: A => String
}

val x: T = ...
val str: String = x.g(x.f(42))
```

$$T = \mu(\mathit{self} : \{A : \bot..\top;\ f : \forall(\_ : \mathtt{Int})\ \mathit{self}.A;\ g : \forall(\_ : \mathit{self}.A)\ \mathtt{String}\})$$

Enkodowanie GADT w pDOT - więzy

W zakodowaniu więzów pomoże nam wykorzystanie intersection types i singleton types.

```
trait Expr {
  type A
  val x: A
}
trait Lit extends Expr {
  type A = Int
}
trait Pair extends Expr {
  type B; type C
  type A = (B, C)
}
```

$$Expr = \mu(self : \{A : \bot..\top;\ x : self.A\})$$

$$Lit = Expr \wedge \{A : \texttt{Int}..\texttt{Int}\}$$

$$Pair = \mu(s : Expr \wedge \{B : \bot..\top;\ C : \bot..\top;\ A : (s.B * s.C)..(s.B * s.C)\})$$

Weźmy $e : Lit$, wtedy $e.x : e.A$, ale mamy, $e : \{A : \texttt{Int}..\texttt{Int}\}$, zatem $e.A <: \texttt{Int}$, więc $e.x : \texttt{Int}$.

Enkodowanie GADT w pDOT - więzy w pattern matchu

```
trait TL { type T }
trait Expr { self =>
  type A
  def visit(T: TL)(lit: self.type & Lit => T.T): T.T
}
trait Lit extends Expr { self =>
  type A = Int
  val x: Int
  def visit(T: TL)(lit: self.type & Lit => T.T): T.T  =
    lit(self)
}
def eval(R: TL)(e: Expr & { type A = R.T }): R.T =
  e.visit(R)(
    (lit: e.type & Lit) => lit.x
                             // lit.x: Int =:= lit.A =:= e.A =:= R.T
  )
```

Przykład: zakodowanie Expr

```
trait TL { type T }
sealed trait Expr { self =>
  type A
  def visit(R: TL)
          (lit: LitC & self.type => R.T,
           plus: PlusC & self.type => R.T,
           pair: PairC & self.type => R.T): R.T
}
trait LitC extends Expr { self =>
  type A = Int
  def x: Int
  def visit(R: TL)
          (lit: LitC & self.type => R.T,
           plus: PlusC & self.type => R.T,
           pair: PairC & self.type => R.T): R.T =
    lit(this)
}
trait PlusC extends Expr { self =>
  type A = Int
  def lhs: Expr & { type A = Int }
  def rhs: Expr & { type A = Int }
  def visit(R: TL)(...): R.T = plus(this)
}
trait PairC extends Expr { self =>
  type B
  type C
  type A = (B,C)
  def lhs: Expr & { type A = B }
  def rhs: Expr & { type A = C }
  def visit(R: TL)(...): R.T = pair(this)
}
```

```
def Lit(_x: Int): Expr & { type A = Int } =
  new LitC {
    def x = _x
  }
// other constructors analogous

def eval(R: TL)(e: Expr & { type A = R.T }): R.T =
  e.visit(R)(
    (lit: LitC & e.type) =>
      lit.x : lit.A,
    (plus: PlusC & e.type) =>
      val T = new TL { type T = Int }
      eval(T)(plus.lhs) + eval(T)(plus.rhs): plus.A,
    (pair: PairC & e.type) =>
      val TB = new TL { type T = pair.B }
      val TC = new TL { type T = pair.C }
      (eval(TB)(pair.lhs),
       eval(TC)(pair.rhs)) : pair.A
    )
```

Ogólna postać GADT w pDOT

GADT postaci

$$T \equiv \mu t.\lambda\overline{\alpha}.\left(\exists[\overline{\beta_1}, \overline{\sigma_1} = \overline{\alpha}].\tau_1 + \cdots + \exists[\overline{\beta_n}, \overline{\sigma_n} = \overline{\alpha}].\tau_n\right)$$

możemy zakodować następująco:

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

Ogólna postać GADT w pDOT

$$T \equiv \mu t.\lambda\overline{\alpha}.\left(\exists[\overline{\beta_1},\overline{\sigma_1} = \overline{\alpha}].\tau_1 + \cdots + \exists[\overline{\beta_n},\overline{\sigma_n} = \overline{\alpha}].\tau_n\right)$$

Poszczególne warianty kodujemy jako:

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

zaś ich konstruktory:

$$\begin{aligned}
&c_i\mathtt{:}\ \forall\mathtt{(types:\ \{}\beta_{i,1}\mathtt{:}\ \bot\mathtt{..}\top\mathtt{;}\ \mathtt{...}\ \beta_{i,m_i}\mathtt{:}\ \bot\mathtt{..}\top\mathtt{\})}\ \forall\mathtt{(v:}\ \tau_i\mathtt{)}\\
&\qquad \mathtt{env.T}C_i\ \wedge\ \mathtt{\{}\beta_{i,1}\ \mathtt{=}\ \mathtt{types.}\beta_{i,1}\mathtt{;}\ \mathtt{...;}\ \beta_{i,m_i}\ \mathtt{=}\ \mathtt{types.}\beta_{i,m_i}\mathtt{\}}
\end{aligned}$$

$\lambda_{2,G\mu}$

Xi et al. w swojej pracy *Guarded recursive datatype constructors*, która jest jednym z pierwszych opisów GADT zdefiniowali rachunek lambda $\lambda_{2,G\mu}$, który jest rozszerzeniem Systemu F o GADT z pattern matchingiem.
Aby pokazać, że pDOT jest w stanie wyrazić wszystkie własności GADT, chcemy przedstawić schemat kodowania programów w $\lambda_{2,G\mu}$ do pDOTa.

Komplikacje $\lambda_{2,G\mu}$:

- zagnieżdżony pattern matching
- niedeterministyczna ewaluacja
- sprzeczne więzy

$\lambda_{2,G\mu}$ - co już jest zrobione?

Obecnie zdefiniowałem reguły kodowania dla zdeterminizowanego podzbioru $\lambda_{2,G\mu}$ z uproszczonym pattern matchingiem.

Korzystając z tego kodowania, zapisałem w pDOT przytoczony tu przykład Expr oraz prosty interpreter STLC ($\lambda^{\rightarrow}$) oparty na GADT.

Dalsze plany

Dalsze plany:

1. Sformalizować kodowanie $\lambda_{2,G\mu}$ do pDOT
2. Udowodnić, że zakodowane termy się typują
3. Udowodnić, że kodowanie zachowuje semantykę
4. Zgłębić wspomniane ‘komplikacje’
5. Przeanalizować relację GADT z podtypianiem

Bibliografia

Hongwei Xi, Chiyan Chen, and Gang Chen.
Guarded recursive datatype constructors.
SIGPLAN Not., 38(1):224–235, January 2003.

Yitzhak Mandelbaum and Aaron Stump.
Gadts for the ocaml masses.
01 2009.

Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso.
Towards improved gadt reasoning in scala.
In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, Scala '19, pages 12–16, New York, NY, USA, 2019. ACM.

Marianna Rapoport and Ondřej Lhoták.
A path to dot: Formalizing fully path-dependent types.
Proc. ACM Program. Lang., 3(OOPSLA):145:1–145:29, October 2019.

Prezentacja i kod źródłowy dostępne na:
github.com/radeusgd/pDOT-GADT

Dziękuję za uwagę :)