# Formal foundations for GADTs in Scala

Radosław Waśko

January 10, 2020

**Abstract**

GADTs are a very useful language feature that allow encoding some invariants in types. GADT reasoning is currently implemented in Scala and Dotty, but it's plagued with soundness issues. To get a better understanding of GADTs in Scala, we explore how they can be encoded in pDOT, a calculus that is the formal foundation of the Scala programming language, and show a sketch of encoding a calculus containing GADTs to pDOT.

## 1 Introduction

Generalized abstract data types allow to encode expressive invariants in the type system [1]. Both Scala and Dotty support GADT reasoning, but its foundations have not been deeply explored. The aim of this semester project, supervised by Aleksander Boruch-Gruszecki, was to extend the work desribed in the recent paper *Towards Improved GADT Reasoning in Scala* [2]. We want to get more insight of how GADT reasoning can be implemented in Scala's foundations and if what is required to express all aspects of GADT reasoning.

In section 2, we first describe what are GADTs and why they are useful. In section 3, we analyse what is required to encode GADTs in a language; we argue that instead of extending DOT [4] which would require redoing its soundness proof, we can encode the necessary constructs in a well-described dialect, pDOT [8]. In section 4, we show two GADT examples in Scala and how they can be encoded in pDOT; we also sketch an encoding of a general GADT type. In section 5, we argue that full GADT functionality can be expressed in pDOT by sketching an encoding of a calculus containing GADTs into pDOT. For now we do not prove formally that the encoding is valid. In section 6, we mention some pecularities found while working with pDOT. Finally, in section 7, we describe what is necessary to formally show validity of the encoding described in section 5 and suggest further extensions.

## 2 What is a GADT?

In this section I will describe what GADTs are and why they are useful.

Generalized Abstract Data Types are an extension of ADTs that allows the constructors to have additional type parameter and to arbitrarily instatiate the type parameter of their result - this allows to derive additional type equalities and type programs that would not be typable without such an extension.

One of the first description of the higher kinded type representing a GADT is described in a paper called *Guarded recursive datatype constructors* [3].

Languages like Haskell or OCaml have a clear difference between simple ADTs and GADTs. Scala, however, doesn't have a separate mechanism for (G)ADTs - instead what we call ADTs in Scala are actually classes structured so that they behave like ADTs.

We can define ADTs in Scala using the enum syntax:

Listing 1: Simple ADT definition of a List in Scala

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

This creates a sealed trait and case classes for each case under the hood. So the code from listing 1 would actually be equivalent to the following Scala code:

Listing 2: Simple ADT in Scala defined using the basic constructs

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

As the base trait is sealed, the compiler can check for exhaustiveness of the pattern matching, like in languages with native ADT support like Haskell.

The difference between a plain ADT and a Generalized ADT in Scala is that at least one of the constructors instantiates the type parameter in a nontrivial way (using the `extends` clause), like in the example below:

Listing 3: Compact container GADT example

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

In the above example, the $A$ type parameter is instantiated to `Char` in the `Characters` case. This means that when we pattern match over an instance of `Container[A]`, when handling the `Characters` we can infer that $A$ = `Char` and use that information somehow in the pattern-match.

For example the following function would compile:

Listing 4: Example of pattern matching with type equality inference

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

where if not for the type equality inference, we would have to do an explicit cast to $A$ in the first branch. This is a nice example where the GADT reasoning allows us to safely type a module for compact containers (that uses a more compact representation of character lists) that would otherwise require type coercion.

A GADT should have at least one type parameter, otherwise it has no parameters to instantiate and is indistinguishable from a simple ADT.

Below we show an example GADT encoding very simple typed expressions:

Listing 5: Simple expression GADT definition in Scala

```
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)]
}
```

This example shows another feature of GADTs - existential types. In the `Pair` case there are types $B$ and $C$ which are later used in the type parameter. We can treat them as 'existentially quantified', because when we match the `Pair` case we know that there exist some types $B$ and $C$ for which the type parameter $A \equiv (B, C)$.

We can convert the cases containing multiple member values so that they contain just one value of a complex type (for example a tuple). This will make the encoding much simpler, so for further reference we will use this version of the `Expr` type:

Listing 6: Expression GADT transformed so that each case has exactly one member value

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

Which is of course functionally equivalent to the previous definition.

# 3 How to encode GADTs in DOT?

In this section we will provide a high-level overview of what is required to be able to encode GADTs in DOT (the lambda calculus that is the foundation of the Scala language as described in [4]).

As explained in the previous section, GADTs are inherently connected with pattern matching and derivation of additional type equalities in respective cases. That is why to be able to encode GADTs in DOT, we need a way to encode pattern matching and the type equalities introduced by each of the constructors.

## 3.1 Pattern matching

One way to handle pattern matching is to just extend the DOT calculus. However adding new features to the calculus is highly non-trivial and would require modifying the type-safety proof which is a very complex and time consuming task.

Instead it is posible to encode pattern matching by reusing facilities already available in DOT and that's what we have done.

There are numerous ways to encode ADTs with pattern matching in calculi that don't support them out of the box.

Two well-known encodings are the Böhm-Berarducci (introduced in the paper *Automatic Synthesis of Typed Lambda-Programs on Term Algebras* [5]) and the Scott encoding (described for example in section 5.2 of *Directly reflective meta-programming* [6]). They both encode the ADT as functions that are similar to elimination rules in logic, but they differ in the details.

An instance of an ADT value is encoded as a function which takes functions desribing how to handle various cases and return some result. The difference between the two encodings is in how these functions behave.

A short characterization of both encodings will follow. In both cases let us assume we want to encode a datatype $T$ with constructors $C_1$, ..., $C_n$ where each constructor is unary and it's type is $C_i : \tau_i \to T$ (it is trivial to extend this to more arguments for example by accepting tuples).

### 3.1.1 The Böhm-Berarducci Encoding

Böhm-Berarducci Encoding is the typed version of Church encoding. It essentially creates a term that accepts iterators to be applied according to the structure of the data.

In this encoding $T$ becomes $\forall\alpha.(\tau_1 \to \alpha) \to \cdots \to (\tau_n \to \alpha) \to \alpha$ where every occurence of $T$ in $\tau_i$ is replaced by $\alpha$.

Thanks to the fact that the functions are applied to the whole data structure, it is possible to encode complex functions working on the data even without direct support for recursion in the calculus. That is because the encoding itself emulates a recursor for the encoded type (in a similar manner to the recursor for natural numbers in Gödel's System T as shown for example in *Gödel's System T Revisited* [7]).

In this encoding we could a list of integers as $\forall\alpha.(Int \to \alpha \to \alpha) \to \alpha \to \alpha$.

As the encoding resembles iteration over the data structure, it is easy to write functions like summing up elements of a list:

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

However writing some other simple functions can become unexpectedly complicated. For example the tail function (where we assume a tail of an empty list is also an empty list) would become:

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

Where `fst` and `snd` are standard tuple operations and `nil` and `cons` are constructors for the `list` datatype. As we can see the function is complex and computationally expensive (it has to iterate over the whole list even though tail can usually be returned in constant time).

### 3.1.2 The Scott Encoding

Scott encoding is similar to the previously described one, but instead of accepting iterators it can be thought of as accepting continuations standing for case statements of a pattern match.

To be typable, the encoding requires the calculus to support recursive types (as seen in the list example, the function that handles the `Cons` case takes tail which is of the type we are currently defining).

The encoded $T$ becomes $\mu t.\forall\alpha.(\tau_1 \to \alpha) \to \cdots \to (\tau_n \to \alpha) \to \alpha$ where every occurence of $T$ in $\tau_i$ is replaced by $t$ (the recursive self-reference).

In this encoding a list of integers would be $\texttt{List} = \mu t.\forall\alpha.(\texttt{Int} \to t \to \alpha) \to \alpha \to \alpha$.

Writing an analogous tail function in this encoding is straightforward:

```
tail = λlist: List. list [List] (λhead: Int. λtail: List. tail) (nil)
```

The code is much simpler but it requires recursive types.

To write a similar sum function as in the previous encoding, this time we need to use recursion:

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

### 3.1.3 Summary

The main difference between the two is that Böhm-Berarducci's 'visits' the whole data structure while Scott's only unpacks the outermost constructors. Both of these approaches have their advantages.

The Böhm-Berarducci is good for iteration (like folding over a list) and can work in simpler calculi (without recursion).

The Scott encoding on the other hand does essentialy a case-by-case analysis whcih much more closely resembles what pattern matching looks like in most languages.

As DOT has both recursive types and recursion, we choose Scott encoding as it best fits our usecase of emulating pattern matching.

## 3.2 Type equalities

Based on the idea presented in the paper *Towards Improved GADT Reasoning in Scala* [2] we encode type equalities with the help of type members and singleton types.

Each type parameter is encoded by an abstract type member which is then refined in specific cases according to the way it is instantiated.

Looking at the example from Listing 6, we define an abstract type member $A$ in the type `Expr` and we define a subtype `Pair` <: `Expr` in which additional type members $B$ and $C$ are defined and $A$ is refined to $(B, C)$.

When we pattern match an instance `e` of type `Expr[A]` with the case `Pair[B,C]` which is of type `Expr[(B,C)]`, we want to be able to treat types $A$ and $(B, C)$ as equivalent and use them interchangeably.

Listing 7: Encoding type equalities

```
abstract class Expr { self =>
    type A
    def pmatch[R](pair: self.type & Pair => R, ...other cases): R
}
class Pair extends Expr { self =>
    type B
    type C
    type A = (B,C)
    val data: (Expr[B], Expr[C])
    def pmatch[R](pairCase: (self.type & Pair) => R, ...other cases): R = pair(self)
}
```

The visitor function `pmatch` in the base type (which is the Scott encoding of the ADT), takes arguments for each of the cases. Each argument is the function handling the respective case. The important thing is the type of the parameter of that function is an intersection of the specific case's class and the self type. This allows us to derive the needed type equalities.

For example

```
val e: Expr
val x: e.A = e.pmatch[e.A](
    (pair: e.type & Pair) => { val r: (pair.B, pair.C) = ???; r }
    ... other cases
)
```

In the passed lambda function, we can derive `e.A` <: `pair.A` <: (`pair.B`, `pair.C`) and vice-versa which makes both types equivalent. For example we can type `r: (pair.B, pair.C)` as `r: e.A` (using the `SUB` rule).

# 4 GADTs encoded in pDOT

In this section we will show an example encoding of the `Expr` type and a function using this datatype in pDOT. Than we will show an encoding of a more complex example - a typesafe STLC interpreter. Later we will draft a set of general rules of encoding arbitrary GADTs into pDOT.

pDOT (introduced in *A Path to DOT: Formalizing Fully Path-dependent Types* [8]) is a sound extension / dialect of DOT that amongst others, adds singleton type support.

As we need singleton types for our encoding of type equalities, from now on we will be working with pDOT to be able to take advantage of this feature.

## 4.1 Example pattern match

We will now show an encoding type Expr defined in Listing 6 and the following `eval` function that uses this datatype - it evaluates the encoded expression and returns the computation result which depends on expression's type. Thanks to GADT reasoning, the function is typesafe and doesn't need any casts.

Listing 8: Pattern matching Expr in Scala

```
def eval[A](e: Expr[A]): A = e match {
  case Lit(n) => n
  case Plus(data) => eval(data.fst) + eval(data.snd)
  case Pair(data) => (eval(data.fst), eval(data.snd))
}
```

For a pattern match as the one presented in listing 8 to typecheck we need special typing support.

If `Expr` was a normal ADT, the first two cases would need a (possibly unsafe) typecast to be typable - we want to return an integer, but the function is expected to return an instance of type $A$. Only thanks to the type equalities introduced by the GADT (namely $A$ = `Int`), it is possible to derive that `n: Int` can also by typed as `n: A`.

Similar reasoning is also required in the third case where we need to be able to treat a value of type $(B, C)$ as $A$.

## 4.2 Encoding conventions

As described above, we encode the type parameters as abstract type members, instantiation of type parameters in GADT constructors becomes refinement of these members. Encoding generic types using abstract type members is described for example in the paper *The Essence of Dependent Object Types* [4].

To encode pattern matching the base type exports a visitor function `pmatch` which is the elimination rule from the Scott encoding.

As `pmatch` has a generic result type we need to be able to encode polymorphic functions. We encode the type parameters by adding additional arguments called *typelabels*. For example a function `def id[A](x: A): A = x` would be encoded as $\lambda(A : \{T : \bot..\top\})\ \lambda(x : A.T)\ x$.

For readability we use some simplifications for the pDOT syntax:

1. $\{A = \tau\}$ in type declaration as shorthand for $\{A : \tau..\tau\}$, because duplicating big types greatly reduces readability.
2. $[\tau]$ stands for $\nu(s : \{T : \tau..\tau\})\{T = \tau\}$ as we often need to pass typetags to functions, so a compact notation makes it more readable and easier to understand.
3. When we need to declare a complex object declaration or type, instead of writing $\{a\} \wedge \{b\}$ we write $\{a; b\}$ or even

   ```
   {
     a
     b
   }
   ```

   to reduce clutter.
4. It is allowed to apply arbitrary expressions to arbitrary expressions (instead of only stable paths). To achieve that, every such application $e_1$ $(e_2)$ is transformed into `let` $p = e_1$ `in let` $q = e_2$ `in` $p$ $q$ (where $p$ and $q$ are fresh names).
5. Instead of writing $\nu(s : \{A : T..T; x : U\})\{A = T; x = u\}$ we just write

   ```
   ν(s: {
       A = T
       x: U
       x = u
   })
   ```

   and skip the inline type altogether if it is obvious from the context.

The encoding also uses some primitives that are not built-in in pDOT, so we first define an object called `lib` that defines them:

```
let lib = ν(lib: {
    Unit = {U: ⊥..⊤}
    unit = ν(s: {U=⊤})
    Tuple = μ(s: {T1: ⊥..⊤; T2: ⊥..⊤; fst: s.T1; snd: s.T2})
    tuple = λ(tl: {T1: ⊥..⊤; T2: ⊥..⊤})
            λ(x1: tl.T1) λ(x2: tl.T2)
            ν(s: {T1 = tl.T1; T2 = tl.T2; fst = x1; snd = x2})
}) in ...
```

We also don't define, but assume there exists an `Int` type and a + function that will be written using the infix notation. So te be absolutely precise we'd convert expressions $e_1 + e_2$ into `int.plus` $e_1$ $e_2$ where `int.plus` : $\forall(a$ : `Int`$)$ $\forall(b$ : `Int`$)$ `Int`.

## 4.3 Encoding the datatype

Below we show an encoding of the `Expr` type. When encoding the type and its constructors we put all generated definitions into a toplevel object `env`. However for readability we skip the object and some of the declarations - the full version is available in the appendix A.

First we define the base GADT type with its visitor `pmatch` function and type parameters (here only one: `A1`):

```
Expr = μ(s: {
        A1: ⊥..⊤
        pmatch: ∀(r: {R: ⊥..⊤})
                        ∀(litcase: ∀(arg: s.type ∧ env.ExprLit) r.R)
                        ∀(pluscase: ∀(arg: s.type ∧ env.ExprPlus) r.R)
                        ∀(paircase: ∀(arg: s.type ∧ env.ExprPair) r.R)
                        r.R
})
```

Then, for each case we define its type - this type helps to encode the type equalities and the data that the case holds. Below we show the type for the `Pair` case, other types are defined analogously.

```
ExprPair = μ(s: env.Expr
        ∧ {a: ⊥..⊤}
        ∧ {b: ⊥..⊤}
        ∧ {A1 = lib.Tuple ∧ {T1 = s.a; T2 = s.b}}
        ∧ {data: lib.Tuple ∧ {T1 = env.Expr ∧ {A1=s.a}; T2 = env.Expr ∧ {A1=s.b}}}
)
```

Finally we need a way to create instances of the GADT - for each case we create a constructor, as earlier we show the `Pair` case below:

```
pair: ∀(types: {a: ⊥..⊤; b: ⊥..⊤})
      ∀(t: lib.Tuple ∧ {T1 = env.Expr ∧ {A1=types.a}; T2 = env.Expr ∧ {A1=types.b}})
         env.ExprPair ∧ {a = types.a; b = types.b}
pair =
  λ(types: {a: ⊥..⊤; b: ⊥..⊤})
  λ(t: lib.Tuple ∧ {T1 = env.Expr ∧ {A1=types.a}; T2 = env.Expr ∧ {A1=types.b}})
       ν(s: env.ExprPair ∧ {a = types.a; b = types.b}) {
               a = types.a
               b = types.b
               A1 = lib.Tuple ∧ {T1 = s.a; T2 = s.b}
               data = t
               pmatch = λ(r: {R: ⊥..⊤})
                        λ(litcase: ∀(arg: s.type ∧ env.ExprLit) r.R)
                        λ(pluscase: ∀(arg: s.type ∧ env.ExprPlus) r.R)
                        λ(paircase: ∀(arg: s.type ∧ env.ExprPair) r.R)
                           let h={z=s} in paircase h.z
        }
}
```

It might seem strange that we write `let h={z=s} in paircase h.z` instead of just `paircase s`. This is because the second expression is not actually well-typed. We describe the issue in more detail in section 6.1

## 4.4 Encoding a simple pattern match

Below we will show an encoding of the function `eval` as defined in listing 8:

Listing 9: Encoding eval. Full version in appendix A.

```
let eval =
  let hlp = ν(f: {
    fix =
      λ(u: lib.Unit)
      λ(a: {T: ⊥..⊤})
      λ(e: env.Expr ∧ {A1=a.T})
       let tl = ν{R=a.T} in
       let lit = λ(arg: e.type ∧ env.ExprLit) let data = arg.data in
           data
       in
       let plus = λ(arg: e.type ∧ env.ExprPlus) let data = arg.data in
           let lhs = f.fix lib.unit [Int] data.fst in
           let rhs = f.fix lib.unit [Int] data.snd in
             lhs + rhs
       in
       let pair = λ(arg: e.type ∧ env.ExprPair) let data = arg.data in
           let lhs = f.fix lib.unit [arg.a] data.fst in
           let rhs = f.fix lib.unit [arg.b] data.snd in
           lib.tuple ν{T1=arg.a; T2=arg.b} lhs rhs
       in
```

```
      e.pmatch tl lit plus pair
  })
  in hlp.fix lib.unit
```

The helper object `hlp` is used as a fixpoint to encode the recursive function call.

All of the cases need to return values of type `a.T`.

For the `lit` case to typecheck, we need to be able to type `data: a.T`. As `e: {A1=a.T}`, we have `e.A1 <: a.T`. Since `arg: e.type` we can get `arg.A1 <: e.A1` by the $\text{SNGL}_{pq}$ rule. As `arg: env.ExprLit` (which implies `arg: {A1=Int}`), we know that `Int <: arg.A1` (by the rule `SEL`). We can connect all the inequalities to get `Int <: ... <: a.T`, we also know that `data: Int`. So by the rules `TRANS` and `SUB`, we get `data: a.T` which we wanted to prove.

For the `pair` case, we return a `Tuple` $\wedge$ `{T1=arg.a; T2=arg.b}` so we want to prove `Tuple` $\wedge$ `{T1=arg.a; T2=arg.b} <: a.T`.
Analogously as in previous cases, we get `Tuple` $\wedge$ `{T1=arg.a; T2=arg.b} <: arg.A1 <: e.A1 <: a.T` and can use the `SUB` rule.

## 4.5 Encoding STLC with De Bruijn indices

In the following section we will encode a more complex and interesting example - a simply typed lambda calculus interpreter. For simplicity, instead of using variable names which require using some kind of a dictionary for mapping variable names to their types and handling shadowing, the variables will be encoded by the DeBruijn indices - the index $i$ binds the variable of the $i$th closest lambda that is above the current term in the derivation tree (counting from 0). For example $id = \lambda.\#0$ and $\lambda x.\lambda y.x$ would become $\lambda.\lambda.\#1$. A good description of the concept of DeBruijn indices can be found in *Stitch : The Sound Type-Indexed Type Checker* [9, section 2.3].

### 4.5.1 STLC in Scala

We define two GADTs - `Var` representing a variable encoded by its DeBruijn index and `Expr` which encodes STLC expressions.

The helper functions `fst` and `snd` are to aid Scala's type inference. In the pDOT encoding we will use for example `x.fst` instead of `fst[a,b](x)`.

Listing 10: STLC interpreter in Scala. Credits to Paolo G. Giarrusso. The original code can be found at https://gist.github.com/Blaisorblade/89d433e81001f72dca864f4b6fcc0b6a

```
enum Var[G, A] {
  case Z[G, A](nodata: Unit) extends Var[(A, G), A]
  case S[G, A, B](x: Var[G, A]) extends Var[(B, G), A]
}

def fst[A, B](x: (A, B)): A = x._1
def snd[A, B](x: (A, B)): B = x._2

import Var._
def evalVar[G, A](x: Var[G, A])(rho: G): A = x match {
  case _: Z[g, a] =>
    fst[a, g](rho)
  case s: S[g, a, b] =>
    evalVar(s.x)(snd[b, g](rho))
}

enum Expr[G, A] {
  case Lit[G](n: Int) extends Expr[G, Int]
  case V[G, A](x: Var[G, A]) extends Expr[G, A]
  case App[G, A, B](data: (Expr[G, A => B], Expr[G, A])) extends Expr[G, B]
  case Fun[G, A, B](body: Expr[(A, G), B]) extends Expr[G, A => B]
}
```

```
import Expr._
def eval[A, G](e: Expr[G, A])(rho: G): A = e match {
  case Lit(n) => n
  case V(x) => evalVar(x)(rho)
  case App(data) =>
    val f = data._1
    val a = data._2
    val fdenot = eval(f)(rho)
    val adenot = eval(a)(rho)
    fdenot(adenot)
  case f: Fun[g, a, b] =>
    (x: a) => eval(f.body)(x, rho)
}
```

This example is interesting because it requires advanced GADT reasoning to typecheck.

To typecheck the first match case in `evalVar` we need to use the type equalities for an argument that is not directly a GADT value - `rho`. We need `rho: (a, g)` while we only have `rho: G`, so we need to infer the equality `G = (a, g)`. This equality is introduced by the `Z` constructor.

### 4.5.2 STLC in pDOT

Below we show the most interesting parts of the encoding of the above interpreter in pDOT. The full code is available in the appendix.

Encoding the datatypes and their constructors is done analogously as in the previous example, so we will just show the base types and one example constructor.

Listing 11: Encoding Var. Full version in appendix B.

```
Var = μ(s: {
  A1: ⊥..⊤
  A2: ⊥..⊤
  pmatch: ∀(r: {R: ⊥..⊤})
          ∀(Zcase: ∀(arg: s.type ∧ env.GADTZ) r.R)
          ∀(Scase: ∀(arg: s.type ∧ env.GADTS) r.R)
          r.R
})

VarZ = μ(s: env.Var ∧ {
  G: ⊥..⊤
  A: ⊥..⊤
  A1=lib.Tuple ∧ {T1=s.A; T2=s.G}
  A2=s.A
  data: lib.Unit
})

VarS = μ(s: env.Var ∧ {
  G:⊥..⊤
  A: ⊥..⊤
  B: ⊥..⊤
  A1=lib.Tuple ∧ {T1=s.B; T2=s.G}
  A2=s.A
  data: env.Var ∧ {A1=s.G; A2=s.A}
})
```

Listing 12: Encoding Expr. Full version in appendix B.

```
Expr = μ(s: {
  A1: ⊥..⊤
  A2: ⊥..⊤
```

```
  pmatch: ∀(r: {R: ⊥..⊤})
          ∀(Litcase: ∀(arg: s.type ∧ env.ExprLit) r.R)
          ∀(Vcase: ∀(arg: s.type ∧ env.ExprV) r.R)
          ∀(Appcase: ∀(arg: s.type ∧ env.ExprApp) r.R)
          ∀(Funcase: ∀(arg: s.type ∧ env.ExprFun) r.R)
          r.R
})

ExprLit = μ(s: env.Expr ∧ {
  G: ⊥..⊤
  A1 = s.G
  A2 = Int
  data: Int
})

ExprV = μ(s: env.Expr ∧ {
  G: ⊥..⊤
  A: ⊥..⊤
  A1 = s.G
  A2 = s.A
  data: env.Var ∧ {A1=s.G} ∧ {A2=s.A}
})

ExprApp = μ(s: env.Expr ∧ {
  G: ⊥..⊤
  A: ⊥..⊤
  B: ⊥..⊤
  A1 = s.G
  A2 = s.B
  data: lib.Tuple
        ∧ {T1 = env.Expr ∧ {A1=s.G} ∧ {A2=∀(arg: s.A) s.B}}
        ∧ {T2 = env.Expr ∧ {A1=s.G} ∧ {A2=s.A}}
})

ExprFun = μ(s: env.Expr ∧ {
  G:⊥..⊤
  A:⊥..⊤
  B:⊥..⊤
  A1=s.G
  A2=∀(arg: s.A) s.B
  data: env.Expr ∧ {A1 = lib.Tuple ∧ {T1=s.A} ∧ {T2=s.G}} ∧ {A2 = s.B}
})

// example constructor, others are analogous
V = λ(types: {G: ⊥..⊤; A: ⊥..⊤})
    λ(t: env.Var ∧ {A1=types.G} ∧ {A2=types.A})
      ν(s: env.ExprV ∧ {G=types.G} ∧ {A=types.A}) {
        G = types.G
        A = types.A
        A1 = s.G
        A2 = s.A
        data = t
        pmatch = λ(r: {R: ⊥..⊤})
                 λ(Litcase: ∀(arg: s.type ∧ env.ExprLit) r.R)
                 λ(Vcase: ∀(arg: s.type ∧ env.ExprV) r.R)
                 λ(Appcase: ∀(arg: s.type ∧ env.ExprApp) r.R)
                 λ(Funcase: ∀(arg: s.type ∧ env.ExprFun) r.R)
                 let h={z=s} in Vcase h.z
      }
```

All of the above definitions are placed in the `env` object. Now we will show encoding of the `eval` and `evalVar` functions.

Listing 13: Encoding evalVar. Full version in appendix B.

```
evalVar =
  let hlp = ν(self: {
    fix = λ(u: lib.Unit)
          λ(G: {T: ⊥..⊤})
          λ(A: {T: ⊥..⊤})
          λ(x: env.Var ∧ {A1=G.T} ∧ {A2=A.T})
          λ(rho: G.T)
          let tl = ν{R=A.T} in
          let Z = λ(arg: x.type ∧ env.VarZ) let u = arg.data in
            rho.fst
          in
          let S = λ(arg: x.type ∧ env.VarS) let v = arg.data in
            self.fix lib.unit [arg.G] [arg.A] v rho.snd
          in
          x.pmatch tl Z S
  }
  in hlp.fix lib.unit
```

We know that `rho` is of some arbitrary type `G.T`. Without type equalites we couldn't even show that the path `rho.fst` is well-formed.

To typecheck the `Z` case we need to show that `rho` is a tuple and it's first element is of type `A.T`. We know that `G.T <: x.A1` (by rule `SEL` for `x: {A1=G.T}`) and `x.A1 <: arg.A1` (by rule $\text{SNGL}_{pq}$ and the fact that `arg: x.type`), furthermore as `arg: VarZ`, we have `arg.A1 <: lib.Tuple∧{T1=arg.A; T2=arg.G}` (by rule `SEL` and playing with `REC-E`). By connecting these inequalities we can derive `rho: lib.Tuple∧{T1=arg.A; T2=arg.G}` which in turn allows us to conclude `rho.fst: arg.A`. We want the result type to be `A.T`. However we know that `x.A2 <: A.T` (by `SEL` as earlier) and analogously `arg.A2 <: x.A2` (by $\text{SNGL}_{pq}$). `arg.A <: arg.A2` (because `arg: μ(s: {A2 = s.A})`). By connecting these inequalities we get `arg.A <: ... <: A.T` which allows us to derive `rho.fst: A.T` which proves this case correct.

The `S` case is analogous - we prove that `G.T <: lib.Tuple∧{T1=arg.B; T2=arg.G}`, so `rho.snd: arg.G`. It can now be easily seen that the arguments to `self.fix` are of correct types. The result of `self.fix` is of type `arg.A`, but we can show (analogously as in the `Z` case) that `arg.A <: arg.A2 <: x.A2 <: A.T` so we can correctly treat the result as `A.T`.

Listing 14: Encoding evalVar. Full version in appendix B.

```
eval =
  let hlp = ν(self: {
    fix = λ(u: lib.Unit)
          λ(G: {T: ⊥..⊤})
          λ(A: {T: ⊥..⊤})
          λ(e: env.Expr ∧ {A1=G.T} ∧ {A2=A.T})
          λ(rho: G.T)
          let tl = ν{R=A.T} in
          let Lit = λ(arg: e.type ∧ env.ExprLit) let n = arg.data in
            n
          in
          let V = λ(arg: e.type ∧ env.ExprV) let x = arg.data in
            evalVar [G.T] [A.T] x rho
          in
          let App = λ(arg: e.type ∧ env.ExprApp) let s = arg.data in
            let f = s.fst in
            let a = s.snd in
            let fdenot = self.fix lib.unit [G.T] [∀(x: arg.A) arg.B] f rho in
            let adenot = self.fix lib.unit [G.T] [arg.A] a rho in
            fdenot adenot
```

```
          in
          let Fun = λ(arg: e.type ∧ env.ExprFun) let body = arg.data in
            λ(x: arg.A)
            let e2 = lib.tuple (ν({T1=arg.A; T2=G.T})) x rho in
            self.fix lib.unit [lib.Tuple ∧ {T1=arg.A} ∧ {T2=G.T}}] [arg.B] body e2
          in
          e.pmatch tl Lit V App Fun
  }
  in hlp.fix lib.unit
```

The first two cases are simple - we just need to prove `Int <: A.T` in the first case and `arg.A <: A.T` in the second one. Both proofs are similar to ones shown previously.

In the `App` case we see that `f: env.Expr ∧ {A1=arg.G} ∧ {A2=∀(arg: arg.A) arg.B}` and `a: env.Expr ∧ {A1=arg.G} ∧ {A2=arg.A}`. We need to use, in a similar manner as earlier, the equality `G.T ≡ arg.G` and `A.T ≡ arg.B`. With these equalities it's clear that both `self.fix` applications typecheck and we get `fdenot: ∀(x: arg.A) arg.B` and `adenot: arg.A`, so the result is `arg.B` which is the same as `A.T`.

In the `Fun` case, we return a function of type `∀(x: arg.A) arg.B`, but we can derive the equality `∀(x: arg.A) arg.B ≡ A.T`. To type the inner function application we need to notice that `body: env.Expr ∧ {A1 = lib.Tuple ∧ {T1=arg.A} ∧ {T2=arg.G}} ∧ {A2 = arg.B}`.

## 4.6 General GADT encoding

Xi et al. [3] define a general GADT type as:

$$(\overline{\alpha})T \equiv \mu t.\lambda\overline{\alpha}.\,(\exists[\overline{\alpha_1}, \overline{\sigma_1} = \overline{\alpha}].\tau_1 + \cdots + \exists[\overline{\alpha_n}, \overline{\sigma_n} = \overline{\alpha}].\tau_n) \tag{1}$$

In this section we will describe a general draft of encoding a GADT. We will assume some source language that can express the type defined in equation 1 and will show only a translation for the interesting part - the GADT type itself.

We will create a base type `T` that will be corresponding to our $(\overline{\alpha})T$ and the constructors for each case.

For each case in the inner sum-type, we will create a type which will be used to encode type equalities and the existential type parameters of that type (for clarity, $k$th case will be called $C_k$) and it's constructor. As the base type and these helper types need a way to refer to each other, we put all of the types in an object `env`, so that references to `T` become `env.T` inside of the definition etc.

To be able to destruct our base type, it will have a `pmatch` visitor function that will take functions dealing with each case which is the Scott encoding of the underlying ADT.

As the GADT type contains other types we need a way to encode these as well. But as in this section we focus only on encoding the GADT, we assume that we have some `EncodeType` function that takes the type in the source language and a substitution (mapping type variables to pDOT types) and returns a pDOT type that is an encoding of the first argument with the variables substituted according to the second argument. A more concrete example of what this function could look like can be found in section 5.3.1.

First we rewrite the original definition to a form where we can more clearly see how many type parameters there are:

$$(\overline{\alpha})T \equiv \mu t.\lambda\overline{\alpha}.\left(\exists\left[\overline{\beta_1}, \overline{\alpha} = \overline{\sigma_1}\right].\tau_1 + \cdots + \exists\left[\overline{\beta_n}, \overline{\alpha} = \overline{\sigma_n}\right].\tau_n\right) \tag{2}$$

$n$ is the number of cases, $m$ is the number of type parameters (length of the vector $\overline{\alpha}$), $m_i$ is the number of existential types in the $i$th case (the length of the vector $\overline{\beta_i}$). $\alpha_i$ and $\beta_{i,j}$ are type variable names while $\sigma_{i,j}$ and $\tau_i$ may be arbitrary types, also referencing $\overline{\alpha}$ and $\overline{\beta_i}$.

The base type becomes:

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

```
        ...
        ∀(c_n: ∀(arg: env.TC_n ∧ s.type) r.R)
        r.R
})
```

As $\sigma_{i,j}$ and $\tau_i$ are arbitrary types that can reference $\alpha_i$ and $\beta_{i,j}$, we need to be able to refer to these types in pDOT as well. In the recursive type definition using $\mu$ we can refer to type $\alpha_i$ as $s.\alpha_i$. So we define a helper operation $^*$ (defined separately for each case) that converts types that we are referring to into pDOT, let $\tau^* = \texttt{EncodeType}(\tau, \emptyset[\alpha_1 \mapsto \texttt{s}.\alpha_1] \ldots [\alpha_m \mapsto \texttt{s}.\alpha_m][\beta_{i,1} \mapsto \texttt{s}.\beta_{i,1}] \ldots [\beta_{i,m_i} \mapsto \texttt{s}.\beta_{i,m_i}])$.

We can define the type for the $i$th case:

```
TC_i = μ(s: env.T ∧ {
  β_{i,1}: ⊥..⊤; ...; β_{i,m_i}: ⊥..⊤
  α_1 = σ*_{i,1}; ...; α_m = σ*_{i,m}
  data: τ*_i
})
```

And we define a constructor term corresponding to that case is (analogously, let $\tau^{\#} = \texttt{EncodeType}(\tau, \emptyset[\alpha_1 \mapsto \texttt{types}.\alpha_1] \ldots [\alpha_m \mapsto \texttt{types}.\alpha_m][\beta_{i,1} \mapsto \texttt{types}.\beta_{i,1}] \ldots [\beta_{i,m_i} \mapsto \texttt{types}.\beta_{i,m_i}])$):

```
c_i: ∀(types: {β_{i,1}: ⊥..⊤; ... β_{i,m_i}: ⊥..⊤}) ∀(v: τ#_i)
    env.TC_i ∧ {β_{i,1} = types.β_{i,1}; ...; β_{i,m_i} = types.β_{i,m_i}}
c_i = λ(types: {β_{i,1}: ⊥..⊤; ... β_{i,m_i}: ⊥..⊤}) λ(v: τ#_i)
      ν(s: {
        β_{i,1} = types.β_{i,1}; ...; β_{i,m_i} = types.β_{i,m_i}
        α_1 = σ*_{i,1}; ...; α_m = σ*_{i,m}
        value = v
        pmatch = λ(r: {R:⊥..⊤})
                 λ(c_1: ∀(arg: env.TC_1 ∧ s.type) r.R)
                 ...
                 λ(c_n: ∀(arg: env.TC_n ∧ s.type) r.R)
                   let h={z=s} in c_i h.z
      })
```

It is important to note, that we need to actually introduce additional terms to encode a GADT type. That is because the constructors for GADT types can be built-in language constructs, but as we are emulating them, we need additional terms that can be used to create new instances of the GADT.

As pDOT has no notion of ill-formed types, there is no need to prove that the type definitions themselves are correct. To make sure the encoding is valid, we can check that the term for the constructor is well-typed (the proof is analogous to the one in section 4.3).

# 5 Moving towards encoding GADT calculus in pDOT

We want to make sure that we can express all necessary properties of GADTs in pDOT. One way to achieve that goal is to encode another lambda calculus, one that is well-known to support GADTs, in pDOT.

We have chosen the $\lambda_{2,G\mu}$ calculus, described in the *Guarded Recursive Datatype Constructors* paper [3] which is one of the papers that defined the GADT concept in the first place.

In the following sections we will describe some challenges that arise when creating such an encoding. Then we will outline how the encoding could look.

## 5.1 Pattern matching

The $\lambda_{2,G\mu}$ calculus supports a complex variant of pattern matching.

It allows for nested patterns, non-exhaustive sets of clauses (the semantics is defined in such a way that if no pattern is matched the program is 'stuck') and multiple patterns matching the same thing (the actual evaluated pattern is chosen non-deterministically).

On the other hand, the Scott encoding of pattern matching that we use only supports having exactly one case for each of the matched constructors and no pattern nesting.

From now on we will assume a restricted dialect of $\lambda_{2,G\mu}$ with what we will from now on call *simple pattern matching* where each pattern match has exactly one case for each constructor of the datatype is being matched (or exactly one case for matching a tuple or a unit) and the patterns are not nested.

Some ideas on how to deal with unrestricted pattern matching are described in section 7.2.

## 5.2 Nominality

The GADT types in $\lambda_{2,G\mu}$ are nominal [10] - two types with the same structure but different names cannot be unified.

pDOT is by default structurally typed, so special steps have to be taken to encode nominal types in it, although it is possible as described in *The Essence of Dependent Object Types* [4].

Our encoding will skip this issue for simplicity, because even if our encoded types are structural, all programs that are typable in $\lambda_{2,G\mu}$ will also be typable in pDOT after being encoded (because all programs that typecheck under nominal typing, also typecheck under structural typing). The described encoding however can be easily modified to make the types nominal if that was needed.

## 5.3 Encoding sketch

In this section we will describe a sketch of encoding the $\lambda_{2,G\mu}$ calculus in pDOT.

The complete encoding would convert a type derivation of a term in $\lambda_{2,G\mu}$ into a type derivation of an encoded term in pDOT - this way the encoding would also be a proof that all well-typed source terms are encoded to well-typed terms in pDOT. As this has exceeded the scope of this report, we present an encoding that converts a type derivation of a term in the source language into a pDOT syntax tree. Additionally we include an informal argument for why the resulting terms are well-typed.

We define the whole $\lambda_{2,G\mu}$ program to be a type derivation for the root term and the constant $\Sigma$. The whole encoding is then defined by a function $\texttt{EncodeProgram}(\texttt{expr derivation}, \Sigma)$.

We assume a supply of fresh names that do not appear anywhere in the whole converted program (so that it will be simple to show that they will not introduce any name aliasing). We mark the first occurrence of a fresh name with a box, for example $\boxed{\mathit{name}}$.

To keep track of converting $\lambda_{2,G\mu}$ names to pDOT names or types / values we will use substitutions $\Theta_t$ which maps $\lambda_{2,G\mu}$ type names to pDOT types and $\Theta_f$ that maps $\lambda_{2,G\mu}$ fix variable names to pDOT paths. $\Theta[a \mapsto b]$ is a substitution that returns $b$ for $a$ and $\Theta(x)$ for other arguments. $\emptyset$ stands for the empty substitution (a function that is nowhere defined).

### 5.3.1 Encoding types

As some terms require to write their types, we also need a scheme for encoding terms. We will define a function $\texttt{EncodeType}(\tau, \Theta_t)$ which takes a $\lambda_{2,G\mu}$ type and a substitution environment (mapping type names to pDOT types) and returns a pDOT type. The function is defined recursively with every invokation being on a smaller type.

$$\texttt{EncodeType}(\alpha, \Theta_t) = \Theta_t(\alpha)$$

$$\texttt{EncodeType}(\mathbf{1}, \Theta_t) = \texttt{lib.Unit}$$

$$\texttt{EncodeType}(\tau_1 * \tau_2, \Theta_t) = \texttt{lib.Tuple} \wedge \{T1 = \texttt{EncodeType}(\tau_1, \Theta_t)\} \wedge \{T2 = \texttt{EncodeType}(\tau_2, \Theta_t)\}$$

$$\texttt{EncodeType}(\tau_1 \to \tau_2, \Theta_t) = \forall \left( \boxed{\mathit{arg}} : \texttt{EncodeType}(\tau_1, \Theta_t) \right). \texttt{EncodeType}(\tau_2, \Theta_t)$$

$$\texttt{EncodeType}((\tau_1, \ldots, \tau_n)\, T, \Theta_t) = \texttt{env.T} \wedge \{A_1 = \texttt{EncodeType}(\tau_1, \Theta_t)\} \cdots \wedge \{A_n = \texttt{EncodeType}(\tau_n, \Theta_t)\}$$

$$\texttt{EncodeType}(\forall \alpha.\tau, \Theta_t) = \forall \left( \boxed{A} : \{T : \bot..\top\} \right). \texttt{EncodeType}(\tau, \Theta_t[\alpha \mapsto A.T])$$

### 5.3.2 Encoding $\Sigma$

Before encoding terms, we have to encode the environment which consists of the $\Sigma$ constant that specifies the constructors. The following encoding is based on the sketch described in section 4.6.

We assume that the provided $\Sigma$ is described by the concrete syntax as described in section 2.2 of the paper [3], and consists of a list of definitions ($\Sigma = \mathbb{T}_1, ..., \mathbb{T}_n$). So the function `EncodeSigma` that we define below takes a list of type definitions ($\mathbb{T}_i$), for each type generates its base type and constructors and returns an env object consisting of these definitions.

```
EncodeSigma(𝕋₁,...,𝕋ₙ) =
  ν(env: {
    EncodeGADT(𝕋₁)
    ...
    EncodeGADT(𝕋ₙ)
  })
```

`EncodeGADT` is a helper function that returns a list of definitions for one type. As each case $c_i$ refers to types $\tau_i$ and $\overline{\sigma_i}$, they will also need to be converted into pDOT. To make the definitions more concise, we define two helper substitutions for each case $i$:

$$\theta_i^* = \emptyset[\beta_{i,1} \mapsto \mathsf{s}.\beta_{i,1}] \ldots [\beta_{i,m_i} \mapsto \mathsf{s}.\beta_{i,m_i}] \tag{3}$$

$$\theta_i^\# = \emptyset[\beta_{i,1} \mapsto \mathtt{types}.\beta_{i,1}] \ldots [\beta_{i,m_i} \mapsto \mathtt{types}.\beta_{i,m_i}] \tag{4}$$

Contrary to the definition in section 4.6, we don't have to add $A_j$ to the substitution, because the concrete syntax cannot refer to these type parameters directly. The only references are through the type equations $A_j \equiv \sigma_{i,j}$.

With these tools, we can now define the `EncodeGADT` function:

```
EncodeGADT(
    typecon (type₁, ..., typeₘ) T = {β̄₁}. (σ̄₁) c₁ of τ₁]
                                  | ...
                                  | {β̄ₙ}. (σ̄ₙ) cₙ of τₙ])
) =
  T = μ(s: {
    A₁: ⊥..⊤; ... Aₘ: ⊥..⊤
    pmatch: ∀(r: {R:⊥..⊤})
            ∀(c₁: ∀(arg: env.Tc₁ ∧ s.type) r.R)
            ...
            ∀(cₙ: ∀(arg: env.Tcₙ ∧ s.type) r.R)
            r.R
  })

  // and for each constructor from c₁ to cₙ
  Tcᵢ = μ(s: env.T ∧ {
    βᵢ,₁: ⊥..⊤
    ..
    βᵢ,ₘᵢ: ⊥..⊤
    A₁ = EncodeType(σᵢ,₁, θᵢ*)
    ...
    Aₘ = EncodeType(σᵢ,ₘ, θᵢ*)
    data: EncodeType(τᵢ, θᵢ*)
  })

  cᵢ: ∀(types: {βᵢ,₁: ⊥..⊤; ... βᵢ,ₘᵢ: ⊥..⊤}) ∀(v: EncodeType(τᵢ, θᵢ#))
        env.Tcᵢ ∧ {βᵢ,₁ = types.βᵢ,₁; ...; βᵢ,ₘᵢ = types.βᵢ,ₘᵢ}
```

```
c_i = λ(types: {β_{i,1}: ⊥..⊤; ... β_{i,m_i}: ⊥..⊤}) λ(v: EncodeType(τ_i, θ_i^#))
  ν(s: {
    β_{i,1} = types.β_{i,1}
    ...
    β_{i,m_i} = types.β_{i,m_i}
    A_1 = EncodeType(σ_{i,1}, θ_i^*)
    ...
    A_m = EncodeType(σ_{i,m}, θ_i^*)
    value = v
    pmatch = λ(r: {R:⊥..⊤})
             λ(c_1: ∀(arg: env.Tc_1 ∧ s.type) r.R)
             ...
             λ(c_n: ∀(arg: env.Tc_n ∧ s.type) r.R)
               let h={z=s} in c_i h.z
  })
```

In the listing above we assume that variables with a line above are vectors of variables. We assume there are $n$ constructors, the base type has $m$ parameters (so each $\overline{\sigma_i}$ has length $m$) and the $i$th constructor has $m_i$ existential types (so $\beta_i$ has length $m_i$).

Names of the defined type and constructors are the same as the names in source language. Names for the helper types for each case are made by concatenating the base type and constructor name's together. To avoid name clashes we just assume that $\Sigma$ doesn't contain types or constructors which names are the same as one of the helper types names. In case there were any conflicts, the types in $\Sigma$ and references to them in the term to be encoded should be renamed beforehand.

### 5.3.3 Encoding terms

We will define a function `EncodeExpr` which takes a $\lambda_{2,G\mu}$ proof tree (and a context variable $\Theta$ described below) and returns a pDOT AST, we will define it recursively, case by case (each case corresponds to a typing rule from figure 4 of the paper mentioned earlier), in the following form: $\texttt{EncodeExpr}(e : \tau, \Theta) = e'$.

To simplify notation, $\Theta$ stands for a pair of substitutions - $\Theta_t$ for types and $\Theta_f$ for fix variables. When updating the substitution, we specify which one of the two is updateted by adding a its name before a colon. For example $\Theta[f : a \mapsto b]$ means that in the updated context, the mapping $\Theta_f$ now maps $a$ to $b$.

As we are returning only a syntax tree, we can skip $\Delta$ and $\Gamma$. However to make sure we generate pDOT code that preserves original variable names and changed typenames are handled by the mapping $\Theta_t$.

To simplify notation, we will not show explicitly how we pass the proof trees around, instead for each recursive invocation we will just provide the last judgement from the respective proof tree and assume the whole tree is passed implicitly. The function is well-defined as the structures passed to the recursive invokations will always be proper subtrees of the inputs.

We also use all the pDOT notational conventions mentioned earlier (notably the ability to apply expressions to arbitrary expressions that in pure pDOT would get rewritten into let statements).

**(ty-eq)** $\texttt{EncodeExpr}(e : \tau_2, \Theta) = \texttt{EncodeExpr}(e : \tau_1, \Theta)$

(because on syntactic level nothing has to be done to encode using the equality - the type derivation in pDOT will use the fact that $\tau_1 <: \tau_2$ and rule **SUB** to prove the corresponding equality)

**(ty-var)** $\texttt{EncodeExpr}(x : \tau,\ \Theta) = x$

$\texttt{EncodeExpr}(f : \tau,\ \Theta) = \Theta_f(f)$

**(ty-cons)** $\texttt{EncodeExpr}(c[\overline{\tau'}](e) : \tau_2[\overline{\alpha} \mapsto \overline{\tau'}],\ \Theta) =$

$\texttt{env}.c\ \ \nu(A_1 = \texttt{EncodeType}(\tau'_1, \Theta_f); ...; A_m = \texttt{EncodeType}(\tau'_m, \Theta_f))\ \ \texttt{EncodeExpr}(e : \tau_1[\overline{\alpha} \mapsto \overline{\tau'}], \Theta)$

**(ty-unit)** $\texttt{EncodeExpr}(\langle\rangle : \mathbf{1},\ \Theta) = \texttt{lib.unit}$

**(ty-tup)** $\texttt{EncodeExpr}(\langle e_1, e_2\rangle : \tau_1 * \tau_2,\ \Theta) =$

$$\begin{aligned}&\texttt{let } \boxed{v_1} = \texttt{EncodeExpr}(e_1 : \tau_1, \Theta) \texttt{ in}\\ &\texttt{let } \boxed{v_2} = \texttt{EncodeExpr}(e_2 : \tau_2, \Theta) \texttt{ in}\\ &\texttt{lib.tuple } \nu\{T1 = \texttt{EncodeType}(\tau_1, \Theta_t); T2 = \texttt{EncodeType}(\tau_2, \Theta_t)\}\ v_1\ v_2\end{aligned}$$

**(ty-fst)** $\texttt{EncodeExpr}(fst(e) : \tau_1,\ \Theta) = \texttt{let } \boxed{\texttt{v}} = \texttt{EncodeExpr}(e : \tau_1 * \tau_2, \Theta) \texttt{ in v.fst}$

**(ty-snd)** $\texttt{EncodeExpr}(snd(e) : \tau_1,\ \Theta) = \texttt{let } \boxed{\texttt{v}} = \texttt{EncodeExpr}(e : \tau_1 * \tau_2, \Theta) \texttt{ in v.snd}$

**(ty-lam)** $\texttt{EncodeExpr}((\lambda x : \tau_1.e) : \tau_1 \to \tau_2,\ \Theta) = \lambda(x : \texttt{EncodeType}(\tau_1, \Theta_t))\ \texttt{EncodeExpr}(e : \tau_2, \Theta)$

**(ty-app)** $\texttt{EncodeExpr}(e_1(e_2) : \tau_2,\ \Theta) = \texttt{EncodeExpr}(e_1 : \tau_1 \to \tau_2, \Theta)\quad \texttt{EncodeExpr}(e_2 : \tau_1, \Theta)$

**(ty-tlam)** $\texttt{EncodeExpr}(\Lambda\alpha.e : \forall\alpha.\tau,\ \Theta) = \lambda\left(\boxed{A} : \{T : \bot..\top\}\right)\ \texttt{EncodeExpr}(e : \tau_2, \Theta[t : \alpha \mapsto A.T])$

**(ty-tapp)** $\texttt{EncodeExpr}(e[\tau_1] : \tau[\alpha \mapsto \tau_1],\ \Theta) = \texttt{EncodeExpr}(e : \forall\alpha.\tau, \Theta)\quad \nu(T = \texttt{EncodeType}(\tau_1, \Theta_t))$

**(ty-fix)** $\texttt{EncodeExpr}((\textbf{fix } f : \tau.e) : \tau,\ \Theta) =$

$$\begin{aligned}&\texttt{let } \boxed{hlpObj} = \nu(\boxed{self} : \{\\ &\quad fix = \lambda\left(\boxed{unused} : \texttt{lib.Unit}\right)\ \texttt{EncodeExpr}(e : \tau, \Theta\left[f : f \mapsto self.fix\ \texttt{lib.unit}\right])\\ &\})\ \texttt{in } hlpObj.fix\ \texttt{lib.unit}\end{aligned}$$

**(ty-let)** $\texttt{EncodeExpr}(\textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} : \tau_2,\ \Theta) = \textbf{let } x = \texttt{EncodeExpr}(e_1 : \tau_1, \Theta) \textbf{ in } \texttt{EncodeExpr}(e_2 : \tau_2, \Theta)$

**(ty-case)** $\texttt{EncodeExpr}(\textbf{case } e \textbf{ of } ms : \tau_2,\ \Theta) = \texttt{EncodeMatches}(e : \tau_1, ms : \tau_1 \to \tau_2, \Theta)$

### 5.3.4 Encoding pattern matching

Now we will describe the `EncodeMatches` function that was mentioned in the previous section.

First we make an important observation - as we have restricted the pattern matching as described in section, we don't have nested patterns, there has to be exactly one case for each constructor and all cases have to be for the same type. This narrows down the combinations of allowed terms to 4 possibilities:

1. one case matching a unit (not really useful)
2. one case matching a variable (effectively equating to a **let** statement)
3. one case deconstructing a tuple
4. $n$ cases for each of $n$ constructors of some type `T`

We will now show how to handle each case.

$$\texttt{EncodeMatches}(e : \mathbf{1},\ \langle\rangle \Rightarrow e_1 : \mathbf{1} \Rightarrow \tau_2\ ,\ \Theta) = \texttt{let}\ \boxed{\textit{unused}} = \texttt{EncodeExpr}(e : \mathbf{1}, \Theta)\ \texttt{in}\ \texttt{EncodeExpr}(e_1 : \tau_2, \Theta)$$

$$\texttt{EncodeMatches}(e : \tau_1,\ x \Rightarrow e_1 : \tau_1 \Rightarrow \tau_2\ ,\ \Theta) = \texttt{let}\ x = \texttt{EncodeExpr}(e : \tau_1, \Theta)\ \texttt{in}\ \texttt{EncodeExpr}(e_1 : \tau_2, \Theta)$$

$$\begin{aligned}\texttt{EncodeMatches}(e : \tau_1 * \tau_2,\ \langle a, b\rangle \Rightarrow e_1 : \tau_1 * \tau_2 \Rightarrow \tau_3\ ,\ \Theta) = \\ \texttt{let}\ \boxed{\texttt{tup}} = \texttt{EncodeExpr}(e : \tau_1 * \tau_2, \Theta) \\ \texttt{let}\ a = \texttt{tup.fst}\ \texttt{in} \\ \texttt{let}\ b = \texttt{tup.snd}\ \texttt{in} \\ \texttt{EncodeExpr}(e_1 : \tau_3, \Theta)\end{aligned}$$

$$\begin{aligned}&\texttt{EncodeMatches}(e : \texttt{T}, \\ &\quad (c_1[\beta_{1,1}, ..., \beta_{1,m_1}](x_1) \Rightarrow e_1 \mid ... \mid c_n[\beta_{n,1}, ..., \beta_{n,m_n}](x_n) \Rightarrow e_n)) : \texttt{T} \Rightarrow \tau_2, \\ &\quad \Theta) = \\ &\quad \texttt{let}\ \boxed{v} = \texttt{EncodeExpr}(e : \texttt{T}, \Theta) \\ &\quad \texttt{let}\ \boxed{c_1\texttt{case}} = \lambda\left(\boxed{\textit{arg}} : \texttt{env.T}C_1\right)\ \texttt{let}\ x_1 = \textit{arg}.\texttt{data}\ \texttt{in} \\ &\qquad \texttt{EncodeExpr}(e_1 : \tau_2, \Theta[t : \beta_{1,1} \mapsto \textit{arg}.\beta_{1,1},\ ...,\ \beta_{1,m_1} \mapsto \textit{arg}.\beta_{1,m_1}]) \\ &\quad \texttt{in} \\ &\quad ... \\ &\quad \texttt{let}\ \boxed{c_n\texttt{case}} = \lambda\left(\boxed{\textit{arg}} : \texttt{env.T}C_n\right)\ \texttt{let}\ x_n = \textit{arg}.\texttt{data}\ \texttt{in} \\ &\qquad \texttt{EncodeExpr}(e_n : \tau_2, \Theta[t : \beta_{n,1} \mapsto \textit{arg}.\beta_{n,1},\ ...,\ \beta_{n,m_n} \mapsto \textit{arg}.\beta_{1,m_n}]) \\ &\quad \texttt{in} \\ &\quad v.\texttt{pmatch}\ \nu(\{R = \texttt{EncodeType}(\tau_2, \Theta_t)\})\ c_1\texttt{case}\ ...\ c_n\texttt{case}\end{aligned}$$

In the last case we assume that the cases are in the same order as in the type definition.

### 5.3.5 Encoding the program

With all the modules in place, we define the final function as

```
EncodeProgram(e, Σ) =
  let lib =
    ν(lib: {
      Unit = {U: ⊥..⊤}
      unit = ν(s: {U=⊤})
      Tuple = μ(s: {T1: ⊥..⊤; T2: ⊥..⊤; fst: s.T1; snd: s.T2})
      tuple = λ(tl: {T1: ⊥..⊤; T2: ⊥..⊤})
              λ(x1: tl.T1) λ(x2: tl.T2)
              ν(s: {T1 = tl.T1; T2 = tl.T2; fst = x1; snd = x2})
    })
  in let env =
    EncodeSigma(Σ)
  in
    EncodeExpr(e, ∅)
```

To avoid name clashes, we assume that `lib` and `env` are fresh names that are not present in the original term `e` to be encoded.

# 6 pDOT peculiarities

In the following section we describe some peculiarities of pDOT that we observed while working with the calculus.

## 6.1 No $p : p.\mathbf{type}$ judgement

One thing we have found is that pDOT typing rules do not include a proof rule that allows to prove $\vdash p : p.\mathbf{type}$. We needed this kind of judgement to be able to type the `pmatch` function as described in section 4.3 - we need to provide an object of type $s.\mathbf{type}$ and we want this object to be the current instance (the self type $s$). It seems very counter intuitive that there is no such judgement in the typing rules, because it looks reasonable, that an object that is the only instance of its own singleton type cannot be directly typed to it. It turns out that this is by-design, because reflexivity of singleton typing would be an issue for pDOT's safety proof [8, section 7.1, p. 27].

This is can be however overcome with a workaround - in every place where one needs an element equal to $p$ that should be typable to $p.\mathbf{type}$, a let binding using a helper object can be used in it's place, namely $\texttt{let}\ helper = \nu(s : \{alias : p.\mathbf{type}\})\{alias = p\}\ \texttt{in}\ helper.alias$. By using the rule DEF-PATH we can type $helper : \{alias : p.\mathbf{type}\}$ and then by FLD-E it is possible to get $helper.alias : p.\mathbf{type}$. Moreover, the value of $helper.alias$ reduces to $p$ by LOOKUP-STEP-VAL so it stands for the original value.

## 6.2 Notation

As pDOT's abstract syntax is mostly focused on simplicity, more complex programs become very long and difficult to read. To alleviate this, we introduce some notation simplifications as described in section 4.2, similarily to what is done in the original pDOT paper [8].

# 7 Future work

In the following section we describe what still needs to be done regarding formalizing GADTs' foundations in Scala.

## 7.1 Proof of validity of the encoding

To completely show that all GADTs features can be correctly encoded, the validity of the encoding described in section 5 has to be proven.

The proof would have to consist of two parts:

### 7.1.1 Semantics

It has to be proven that the encoded term has the same semantics as the original one, ie. evaluating it yields the same result (up to some equivalence).

### 7.1.2 Typing correctness

It has to be proven that every well-typed $\lambda_{2,G\mu}$ closed term is translated into a term that is well-typed in pDOT. One way to approach the proof could be to transform $\lambda_{2,G\mu}$ typing derivation trees into pDOT's typing derivation trees (instead of just pDOT's syntax trees). We conjecture that the method of encoding type equalities with singleton types that is described in section 3.2 is enough to encode type equalities present in $\lambda_{2,G\mu}$ (with the exception of equalities derived from a contradiction that is described below).

**Contradictory equalities** The rules for solving type constraints in $\lambda_{2,G\mu}$ [3, figure 7] allow to derive arbitrary type equalities given a contradiction in the assumptions (essentialy they emulate an *ex falso* rule). In pDOT we currently don't know if it is possible to derive arbitrary equalities given a contradiction in the assumptions. So we currently don't know how to encode equalities arising from contradictory assumptions.

However, contradictory assumptions are introduced only in case branches that will never be executed anyway. For example:

```
enum Expr[A] {
  case EStr(str: String) extends Expr[String]
  case EInt(int: Int) extends Expr[Int]
}
def add(e: Expr[Int], x: Int): Expr[Int] = e match {
  case EStr(_) => ??? // this can prove String = Int type equality, but it is never executed
  case EInt(v) => EInt(v + x)
}
```

The simplest solution is to further restrict the calculus to disallow deriving arbitrary equalities from contradiction.

Another possible solution could be to detect the introduction of contradictory statements in a branch and invalidating that branch by replacing it with an infinite loop. It is however unclear how to detect the contradictions and if this is a decidable problem.

A third solution is to detect expressions which rely on contradiction to derive their typability. This should be possible as the encoding algorithm has access to the type derivation of expression that is being encoded and deriving arbitrary equalities based on a contradiction is achieved by rules that resemble *ex falso* in figure 7 of the paper [3], so the encoding could check if expression's type derivation uses one of these rules (inside of a **ty-eq** judgement). Such expressions relying on a contradiction could be then replaced with an infinite loop that is typable with the desired type.

The last two solutions don't restrict the source language, but they would require a proof that the transofrmations described don't alter the semantics of the original program. The proof would likely consist of showing that a branch that introduces a contradiction is never executed and so it can be replaced with an infinite loop without altering the overall program semantics.

## 7.2 Pattern matching simplifications

The described encoding assumes only simple pattern matches are present in the input term, as described in section 5.1. The original calculus can have much more complex patterns, so it is needed to show how to deal with issues introduced by complex patterns.

Listing 15 shows an example function that uses nested and non-exhaustive patterns.

Listing 15: Function that returns the second element of a list or gets stuck if the list is too short.

```
// (α) list is a GADT type with 2 constructors:
// cons: ∀α.(α ∗ (α)list) → (α)list
// nil: ∀α.1 → (α)list
second = Λα. λl:list[α].
  case l of
    cons[α1](<first, cons[α2](<second, tail>)>) ⇒ second
```

### 7.2.1 Nested patterns

Nested patterns often make it possible to write complex functions in a concise manner, however in the end the computer has to check if the provided object matches the complicated pattern. One approach to check multiple nested patterns in an efficient way is to prepare optimized decision trees, for example as described in *Compiling Pattern Matching to Good Decision Trees* [11]. The mentioned paper describes how to transform

a pattern match of a very similar structure to the one employed in $\lambda_{2,G\mu}$ and generate a decision tree which resembles applying the simplified pattern match (exactly one case for each constructor[1]).

### 7.2.2 Nonexhaustivity

The semantics of the source language are defined in such a way that if a value doesn't match any of the patterns, the evaluation gets stuck. This can also be handled by the decision trees method mentioned earlier. The nonexhaustive matches are transformed to exhaustive matches, however a branch that was missing in the original code is executing a `Fail` command which is the equivalent of stuck evaluation. We can emulate `Fail` in pDOT by running an infinite loop.

The function from listing 15 could be converted to the following code:

Listing 16: The same function as earlier using only the simple pattern matching

```
// Fail is a shorthand for an infinite loop
second = Λα. λl:list[α].
  case l of
    cons[α1](x) ⇒
      case x of
        <first, y> ⇒
          case y of
            cons[α2](z) ⇒
              case z of
                <second, tail> ⇒ second
          | nil[α2] ⇒ Fail
  | nil[α1] ⇒ Fail
```

It would also be needed to prove that the simplified pattern matches preserve original semantics .

### 7.2.3 Nondeterminism

The biggest issue is that semantics of $\lambda_{2,G\mu}$ allow for nondeterministic execution in case multiple patterns are matched. This is a rather unusual design choice for a calculus.

The simplest solution would be to modify the semantics of $\lambda_{2,G\mu}$ to always choose the first matching pattern (which would be in line with the semantics assumed by the decision trees method) and argue that this does not limit the expressivity of the calculus. One way to show that would be to encode the original nondeterministic $\lambda_{2,G\mu}$ in the determinized dialect.

Another solution could be to encode nondeterministic execution in pDOT (by using some kind of backtracking).

However, nondeterministic execution isn't related to GADTs, so this is a lower priority concern for the project.

## 7.3 Subtyping and variance

We describe the GADT functionality basing on $\lambda_{2,G\mu}$ which does not have subtyping. However as pDOT allows for subtyping, interactions of these two language features could be explored.

Firstly, it could be possible to support what is called Open GADTs [12] - GADTs which do not necessarily have a flat hierarchy and can be extended.

Another thing to consider is extending the idea of type equalities that GADTs express with type inequalities. Instead of asserting that $A$ = `Int` in a branch, it could be possible to only assume $A$ <: `Int` etc. The idea of interaction between GADTs and subtyping is explored in *GADTs Meet Subtyping* [13].

[1] Actually the match can include only a subset of all constructors, but then it is required to include a default case which handles the other constructors. This can be handled by replacing the default case with copies for each of the missing constructors.

# 8 Acknowledgements

We thank Paolo G. Giarrusso for taking the time to discuss the encoding and allowing us to reuse his implementation of STLC with De Bruijn indices in Scala which we have encoded into pDOT.

# References

[1] Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. *SIGPLAN Not.*, 40(10):21–40, October 2005.

[2] 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.

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

[4] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In *A List of Successes That Can Change the World*, 2016.

[5] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39:135–154, 1985.

[6] Aaron Stump. Directly reflective meta-programming. *Higher-Order and Symbolic Computation*, 22:115–144, 2009.

[7] Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. Gödel's system t revisited. *Theoretical Computer Science*, 411:1484–1500, 03 2010.

[8] 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.

[9] Stitch : The sound type-indexed type checker ( author ' s cut ). 2018.

[10] Moez A. AbdelGawad. An overview of nominal-typing versus structural-typing in object-oriented programming. *ArXiv*, abs/1309.2348, 2013.

[11] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 35–46, New York, NY, USA, 2008. ACM.

[12] Paolo G. Giarrusso. Open gadts and declaration-site variance: A problem statement. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 5:1–5:4, New York, NY, USA, 2013. ACM.

[13] Gabriel Scherer and Didier Rémy. Gadts meet subtyping. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 554–573, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

# A Appendix: Expr example

Below we show the full pDOT term that encodes the `Expr` type and `eval` function. We again use the notational conventions defined in section 4.2.

Listing 17: Encoding Expr with evalAdd.

```
let lib = ν(lib: {
  Unit = {U: ⊥..⊤}
  unit: lib.Unit
  unit = ν(s: {U=⊤})
```

```
  Tuple = μ(s: {T1: ⊥..⊤; T2: ⊥..⊤; fst: s.T1; snd: s.T2})
  tuple: ∀(tl: {T1: ⊥..⊤; T2: ⊥..⊤}) ∀(x1: tl.T1) ∀(x2: tl.T2) lib.Tuple∧{T1=tl.T1; T2=tl.T2}
  tuple = λ(tl: {T1: ⊥..⊤; T2: ⊥..⊤})
          λ(x1: tl.T1) λ(x2: tl.T2)
            ν(s: {T1 = tl.T1; T2 = tl.T2; fst = x1; snd = x2})
}) in
let env = ν(env: {
  // encoding of Expr
  Expr = μ(s: {
    A1: ⊥..⊤
    pmatch: ∀(r: {R: ⊥..⊤})
            ∀(litcase: ∀(arg: s.type ∧ env.ExprLit) r.R)
            ∀(pluscase: ∀(arg: s.type ∧ env.ExprPlus) r.R)
            ∀(paircase: ∀(arg: s.type ∧ env.ExprPair) r.R)
            r.R
  })

  ExprLit = μ(s: env.Expr
    ∧ {A1=Int}
    ∧ {data: Int}
  )

  ExprPlus = μ(s: env.Expr
    ∧ {A1=Int}
    ∧ {data: lib.Tuple ∧ {T1 = env.Expr ∧ {A1=Int}; T2 = env.Expr ∧ {A1=Int}}}
  )

  ExprPair = μ(s: env.Expr
    ∧ {a: ⊥..⊤}
    ∧ {b: ⊥..⊤}
    ∧ {A1 = lib.Tuple ∧ {T1 = s.a; T2 = s.b}}
    ∧ {data: lib.Tuple ∧ {T1 = env.Expr ∧ {A1=s.a}; T2 = env.Expr ∧ {A1=s.b}}}
  )

  lit: ∀(t: Int) env.ExprLit
  lit =
    λ(t: Int)
      ν(s: env.ExprLit) {
            A1 = Int
            data = t
            pmatch = λ(r: {R: ⊥..⊤})
                     λ(litcase: ∀(arg: s.type ∧ env.ExprLit) r.R)
                     λ(pluscase: ∀(arg: s.type ∧ env.ExprPlus) r.R)
                     λ(paircase: ∀(arg: s.type ∧ env.ExprPair) r.R)
                       let h={z=s} in litcase h.z
      }

  plus: ∀(t: lib.Tuple ∧ {T1 = env.Expr ∧ {A1=Int}; T2 = env.Expr ∧ {A1=Int}}) env.ExprPlus
  plus =
    λ(t: lib.Tuple ∧ {T1 = env.Expr ∧ {A1=Int}; T2 = env.Expr ∧ {A1=Int}})
      ν(s: env.ExprPlus) {
            A1 = Int
            data = t
            pmatch = λ(r: {R: ⊥..⊤})
                     λ(litcase: ∀(arg: s.type ∧ env.ExprLit) r.R)
                     λ(pluscase: ∀(arg: s.type ∧ env.ExprPlus) r.R)
                     λ(paircase: ∀(arg: s.type ∧ env.ExprPair) r.R)
                        let h={z=s} in pluscase h.z
      }
```

```
  pair: ∀(types: {a: ⊥..⊤; b: ⊥..⊤})
        ∀(t: lib.Tuple ∧ {T1 = env.Expr ∧ {A1=types.a}; T2 = env.Expr ∧ {A1=types.b}})
          env.ExprPair ∧ {a = types.a; b = types.b}
  pair =
    λ(types: {a: ⊥..⊤; b: ⊥..⊤})
    λ(t: lib.Tuple ∧ {T1 = env.Expr ∧ {A1=types.a}; T2 = env.Expr ∧ {A1=types.b}})
      ν(s: env.ExprPair ∧ {a = types.a; b = types.b}) {
        a = types.a
        b = types.b
        A1 = lib.Tuple ∧ {T1 = s.a; T2 = s.b}
        data = t
        pmatch = λ(r: {R: ⊥..⊤})
                 λ(litcase: ∀(arg: s.type ∧ env.ExprLit) r.R)
                 λ(pluscase: ∀(arg: s.type ∧ env.ExprPlus) r.R)
                 λ(paircase: ∀(arg: s.type ∧ env.ExprPair) r.R)
                   let h={z=s} in paircase h.z
      }
}) in
let eval =
  let hlp = ν(f: {
    fix =
      λ(u: lib.Unit)
      λ(a: {T: ⊥..⊤})
      λ(e: env.Expr ∧ {A1=a.T})
        let tl = ν{R=a.T} in
        let lit = λ(arg: e.type ∧ env.ExprLit) let data = arg.data in
          data
        in
        let plus = λ(arg: e.type ∧ env.ExprPlus) let data = arg.data in
          let lhs = f.fix lib.unit [Int] data.fst in
          let rhs = f.fix lib.unit [Int] data.snd in
            lhs + rhs
        in
        let pair = λ(arg: e.type ∧ env.ExprPair) let data = arg.data in
          let lhs = f.fix lib.unit [arg.a] data.fst in
          let rhs = f.fix lib.unit [arg.b] data.snd in
            lib.tuple ν{T1=arg.a; T2=arg.b} lhs rhs
        in
        e.pmatch tl lit plus pair
  })
  in hlp.fix lib.unit
in eval [Int] (env.lit 42) // returns 42
```

# B Appendix: STLC example

Listing 18: Encoding Simply typed lambda calculus interpreter in pDOT.

```
let lib = ν(lib: {
  Unit = {U: ⊥..⊤}
  unit: lib.Unit
  unit = ν(s: {U=⊤})
  Tuple = μ(s: {T1: ⊥..⊤; T2: ⊥..⊤; fst: s.T1; snd: s.T2})
  tuple: ∀(tl: {T1: ⊥..⊤; T2: ⊥..⊤}) ∀(x1: tl.T1) ∀(x2: tl.T2) lib.Tuple∧{T1=tl.T1; T2=tl.T2}
  tuple = λ(tl: {T1: ⊥..⊤; T2: ⊥..⊤})
          λ(x1: tl.T1) λ(x2: tl.T2)
```

```
            ν(s: {T1 = tl.T1; T2 = tl.T2; fst = x1; snd = x2})
}) in
let env = ν(env: {
  // encoding of Var
  Var = μ(s: {
    A1: ⊥..⊤
    A2: ⊥..⊤
    pmatch: ∀(r: {R: ⊥..⊤})
            ∀(Zcase: ∀(arg: s.type ∧ env.GADTZ) r.R)
            ∀(Scase: ∀(arg: s.type ∧ env.GADTS) r.R)
            r.R
  })

  VarZ = μ(s: env.Var ∧ {
    G: ⊥..⊤
    A: ⊥..⊤
    A1=lib.Tuple ∧ {T1=s.A; T2=s.G}
    A2=s.A
    data: lib.Unit
  })

  VarS = μ(s: env.Var ∧ {
    G:⊥..⊤
    A: ⊥..⊤
    B: ⊥..⊤
    A1=lib.Tuple ∧ {T1=s.B; T2=s.G}
    A2=s.A
    data: env.Var ∧ {A1=s.G; A2=s.A}
  })

  Z = λ(types: {G: ⊥..⊤; A: ⊥..⊤}) λ(t: lib.Unit)
        ν(s: env.VarZ ∧ {G=types.G} ∧ {A=types.A}) {
          G = types.G
          A = types.A
          A1 = lib.Tuple ∧ {T1=s.A; T2=s.G}
          A2 = s.A
          data = t
          pmatch = λ(r: {R: ⊥..⊤})
                   λ(Zcase: ∀(arg: s.type ∧ env.VarZ) r.R)
                   λ(Scase: ∀(arg: s.type ∧ env.VarS) r.R)
                      let h={z=s} in Zcase h.z
        }

  S = λ(types: {G: ⊥..⊤; A: ⊥..⊤; B: ⊥..⊤}) λ(t: env.Var ∧ {A1=types.G} ∧ {A2=types.A})
        ν(s: env.VarS ∧ {G=types.G} ∧ {A=types.A} ∧ {B=types.B}) {
          G = types.G
          A = types.A
          B = types.B
          A1 = lib.Tuple ∧ {T1=s.B; T2=s.G}
          A2 = s.A
          data = t
          pmatch = λ(r: {R: ⊥..⊤})
                   λ(Zcase: ∀(arg: s.type ∧ env.VarZ) r.R)
                   λ(Scase: ∀(arg: s.type ∧ env.VarS) r.R)
                      let h={z=s} in Scase h.z
        }

  Expr = μ(s: {
    A1: ⊥..⊤
```

```
  A2: ⊥..⊤
  pmatch: ∀(r: {R: ⊥..⊤})
          ∀(Litcase: ∀(arg: s.type ∧ env.ExprLit) r.R)
          ∀(Vcase: ∀(arg: s.type ∧ env.ExprV) r.R)
          ∀(Appcase: ∀(arg: s.type ∧ env.ExprApp) r.R)
          ∀(Funcase: ∀(arg: s.type ∧ env.ExprFun) r.R)
          r.R
})

ExprLit = μ(s: env.Expr ∧ {
  G: ⊥..⊤
  A1 = s.G
  A2 = Int
  data: Int
})

ExprV = μ(s: env.Expr ∧ {
  G: ⊥..⊤
  A: ⊥..⊤
  A1 = s.G
  A2 = s.A
  data: env.Var ∧ {A1=s.G} ∧ {A2=s.A}
})

ExprApp = μ(s: env.Expr ∧ {
  G: ⊥..⊤
  A: ⊥..⊤
  B: ⊥..⊤
  A1 = s.G
  A2 = s.B
  data: lib.Tuple
        ∧ {T1 = env.Expr ∧ {A1=s.G} ∧ {A2=∀(arg: s.A) s.B}}
        ∧ {T2 = env.Expr ∧ {A1=s.G} ∧ {A2=s.A}}
})

ExprFun = μ(s: env.Expr ∧ {
  G:⊥..⊤
  A:⊥..⊤
  B:⊥..⊤
  A1=s.G
  A2=∀(arg: s.A) s.B
  data: env.Expr ∧ {A1 = lib.Tuple ∧ {T1=s.A} ∧ {T2=s.G}} ∧ {A2 = s.B}
})

Lit = λ(types: {G: ⊥..⊤}) λ(t: Int)
        ν(s: env.ExprLit ∧ {G=types.G}) {
              G = types.G
              A1 = s.G
              A2 = Int
              data = t
              pmatch = λ(r: {R: ⊥..⊤})
                       λ(Litcase: ∀(arg: s.type ∧ env.ExprLit) r.R)
                       λ(Vcase: ∀(arg: s.type ∧ env.ExprV) r.R)
                       λ(Appcase: ∀(arg: s.type ∧ env.ExprApp) r.R)
                       λ(Funcase: ∀(arg: s.type ∧ env.ExprFun) r.R)
                          let h={z=s} in Litcase h.z
        }

V = λ(types: {G: ⊥..⊤; A: ⊥..⊤}) λ(t: env.Var ∧ {A1=types.G} ∧ {A2=types.A})
```

```
        ν(s: env.ExprV ∧ {G=types.G} ∧ {A=types.A}) {
              G = types.G
              A = types.A
              A1 = s.G
              A2 = s.A
              data = t
              pmatch = λ(r: {R: ⊥..⊤})
                       λ(Litcase: ∀(arg: s.type ∧ env.ExprLit) r.R)
                       λ(Vcase: ∀(arg: s.type ∧ env.ExprV) r.R)
                       λ(Appcase: ∀(arg: s.type ∧ env.ExprApp) r.R)
                       λ(Funcase: ∀(arg: s.type ∧ env.ExprFun) r.R)
                          let h={z=s} in Vcase h.z
        }

  App = λ(types: {G: ⊥..⊤; A: ⊥..⊤; B: ⊥..⊤}) λ(t: lib.Tuple ∧ {T1 = env.Expr ∧ {A1=types.G}
      ↪ ∧ {A2=∀(arg: types.A) types.B}} ∧ {T2 = env.Expr ∧ {A1=types.G} ∧ {A2=types.A}})
        ν(s: env.ExprApp ∧ {G=types.G} ∧ {A=types.A} ∧ {B=types.B}) {
              G = types.G
              A = types.A
              B = types.B
              A1 = s.G
              A2 = s.B
              data = t
              pmatch = λ(r: {R: ⊥..⊤})
                       λ(Litcase: ∀(arg: s.type ∧ env.ExprLit) r.R)
                       λ(Vcase: ∀(arg: s.type ∧ env.ExprV) r.R)
                       λ(Appcase: ∀(arg: s.type ∧ env.ExprApp) r.R)
                       λ(Funcase: ∀(arg: s.type ∧ env.ExprFun) r.R)
                          let h={z=s} in Appcase h.z
        }

  Fun = λ(types: {G: ⊥..⊤; A: ⊥..⊤; B: ⊥..⊤}) λ(t: env.Expr ∧ {A1 = lib.Tuple ∧ {T1=s.A} ∧ {
      ↪ T2=s.G}} ∧ {A2 = s.B})
        ν(s: env.ExprFun ∧ {G=types.G} ∧ {A=types.A} ∧ {B=types.B}) {
              G = types.G
              A = types.A
              B = types.B
              A1 = s.G
              A2 = ∀(arg: s.A) s.B
              data = t
              pmatch = λ(r: {R: ⊥..⊤})
                       λ(Litcase: ∀(arg: s.type ∧ env.ExprLit) r.R)
                       λ(Vcase: ∀(arg: s.type ∧ env.ExprV) r.R)
                       λ(Appcase: ∀(arg: s.type ∧ env.ExprApp) r.R)
                       λ(Funcase: ∀(arg: s.type ∧ env.ExprFun) r.R)
                          let h={z=s} in Funcase h.z
        }

} in
let evalVar =
  let hlp = ν(self: {
    fix = λ(u: lib.Unit)
          λ(G: {T: ⊥..⊤})
          λ(A: {T: ⊥..⊤})
          λ(x: env.Var ∧ {A1=G.T} ∧ {A2=A.T})
          λ(rho: G.T)
            let tl = ν{R=A.T} in
            let Z = λ(arg: x.type ∧ env.VarZ) let u = arg.data in
                rho.fst
```

```
            in
            let S = λ(arg: x.type ∧ env.VarS) let v = arg.data in
                self.fix lib.unit [arg.G] [arg.A] v rho.snd
            in
            x.pmatch tl Z S
  }
  in hlp.fix lib.unit
in
let eval =
  let hlp = ν(self: {
    fix = λ(u: lib.Unit)
          λ(G: {T: ⊥..⊤})
          λ(A: {T: ⊥..⊤})
          λ(e: env.Expr ∧ {A1=G.T} ∧ {A2=A.T})
          λ(rho: G.T)
            let tl = ν{R=A.T} in
            let Lit = λ(arg: e.type ∧ env.ExprLit) let n = arg.data in
                n
            in
            let V = λ(arg: e.type ∧ env.ExprV) let x = arg.data in
                evalVar [G.T] [A.T] x rho
            in
            let App = λ(arg: e.type ∧ env.ExprApp) let s = arg.data in
                let f = s.fst in
                let a = s.snd in
                let fdenot = self.fix lib.unit [G.T] [∀(x: arg.A) arg.B] f rho in
                let adenot = self.fix lib.unit [G.T] [arg.A] a rho in
                fdenot adenot
            in
            let Fun = λ(arg: e.type ∧ env.ExprFun) let body = arg.data in
                λ(x: arg.A)
                  let e2 = lib.tuple (ν({T1=arg.A; T2=G.T})) x rho in
                  self.fix lib.unit [lib.Tuple ∧ {T1=arg.A} ∧ {T2=G.T}}] [arg.B] body e2
            in
            e.pmatch tl Lit V App Fun
  }
  in hlp.fix lib.unit
```