Open Problems

Release 1.0

Pierre-Évariste Dagand

Jan 22, 2019

CONTENTS

In this course, we have seen that there may be such a thing as "dependently-typed programming" after all. We saw that it can be:

- effectufl (lecture 1)
- correct-by-construction (lecture 2)
- structurally recursive or not (lecture 3)
- generic (lecture 4)

And it is not just "as good as" languages of the ML family: it enables genuinely novel and interesting forms of programming.

However, some would argue that programming with dependent type is unbearably difficult (watch out for disgruntled Coq users). They may not be absolutely right but perhaps not absolutely wrong either. For instance, we never had to perform a type cast along a propositional equality: this is too good to be true. In this last lecture, we explain why we were able to write these programs so elegantly, we dissect a few *bad* programs and highlight some very *ugly* (or impossible) ones.

This time, we shall not try to come to a conclusion: we are touching upon research questions, only the future will tell whether the AI robot take over the planet will be programmed with Haskell type families, C++ template, or ANSI C.

CHAPTER

ONE

THE GOOD

We review a few, salient definitions of each lecture. For the definitions I had to come up with myself or had to reconstruct from a paper, I provide an estimate of the time it took me to find the "right" definition (Conor McBride talks about the act of "engineering coincidences"). This measure must be interpreted in relative terms as I am, after all, only a moderately clever person with only one PhD (whose topic was on programming in the inductive fragment of type theory).

1.1 Lecture 0: The Evolution of a Typechecker

The tricky definition is the typechecker itself

```
_⊢?_∋_ : (Γ : context)(T : type)(t : term ⇓) → Γ ⊢[ ⇓ ]-view T ∋ t
_⊢?_∈  : (Γ : context)(t : term ⇑) → Γ ⊢[ ⇑ ]-view t ∈
```

that uses `_⊢[_]-view_` to ensure that the type-checking or type-failing derivation produced corresponds to the original term through the forgetful function `⌊_⌋`:

```
data _⊢[_]-view_ (Γ : context)(d : dir) : Dir d → Set where
  yes : ∀ {T} (Δ : Γ ⊢[ d ] T)   → Γ ⊢[ d ]-view ⌊ Δ ⌋
  no  : ∀ {T} (¬Δ : Γ ⊢/[ d ] T)  → Γ ⊢[ d ]-view ⌊ ¬Δ ⌋
```

The trick is to notice that not only is the function `⌊_⌋` injective but it is also merely mapping each **constructor** of the source datatype to a constructor of the target datatype. Therefore, as we pattern match on derivations, the type becomes more precise by revealing a constructor and therefore the forgetful function mechanically unfolds. If the forgetful map was recursively non trivial, it would have been another story.

Estimated time: It took me about 3 weeks of full-time work to converge on the final implementation of the type-checker. I initially thought that I could exploit a monad-generic implementation of the type-checker to use it for type-directed parsing & pretty-printing but this was incompatible with the dependently-typed implementation for which it is not clear how to write it monadically, let alone monad-generically.

1.2 Lecture 1: Effects

The programs we wrote were simply typed so we had it easy. Nonetheless, we were not able to write the generic definition of the free monad, which would be:

```
data Free (F : Set → Set)(X : Set) : Set where
     return : X → Free F X
     op : F (Free F X) → Free F X
```

because the positivity-checker would reject this definition: `F` provides no guarantee that it will use `Free F X` in a strictly-positive position. This is one of those instances were relying exclusively on oracles as a way of

introducing definitions impedes compositionality. We have seen in Lecture 4 that it could be addressed by reflecting inductive types in the type theory itself.

1.3 Lecture 2: Indices

Good news, using inductive families for typing `sprintf` seems to qualify as industrial-strength, being faster and safer that the untyped alternative.

The expression and machine code are indexed by a type-level tag, which is almost literally transferred by the compiler

```
compile : ∀ {T σ} → exp T → code σ (T :: σ)
```

If significantly distinct types were used on either side of the compiler, a non-trivial translation function would be necessary to state the type of the compiler

```
compile : ∀ {T σ} → exp T → code σ (non-trivial T σ)
```

which would probably make for a few sleepless nights and type-casting proofs.

The difficulty of the normalization-by-evaluation program is mostly conceptual: to deal with binders, one must construct the right model, which was an issue for category theorists before being one for dependently-typed programmers.

As part of this exercise, we saw a structural definition of substitution: this is a folklore trick (that I learned from Conor McBride), meaning that we can assume that countless hours were spent by the first person that tried implementing it. Note that it is not a particularly efficient implementation.

Estimated time: It took me about 2 days of full-time work, mostly studying the literature, to extract the right abstractions to write the normalization-by-evaluation program. However, dependent types were instrumental in forcing me to identify these abstractions: in a simply-typed setting, there is no incentive to do this effort (or one end up doing pure category theory).

1.4 Lecture 3: Recursion

The representation of terms for the MGU relies once again on the folkore representation of well-scoped terms.

The recursion scheme justifying the proof search was found and explained by McBride. This is indeed hard work but that needs to be done anyway. It could be done in a proof-irrelevant manner (using an accessibility predicate to justify the general-recursive definition, for example) or in a proof-relevant manner (which is what we did here).

Estimated time: It took me about 5 days of full-time work to translate the ideas set forward in McBride's Djinn Monotonic into a workable implementation. Knowing which datatype to define (it is given by the paper) and the overall recursion scheme is but one small step: writing a (correct) implementation handling ordered formuli was far from trivial. For instance, the straightforward zipper is gone, replaced by a head-scratching search procedure.

1.5 Lecture 4: Generic programming

Programming with algebraic structures was made possible by the availability of instance arguments (ie. type-classes) in Agda. However, not all objects lend themselves to being instances. Attaching structures to (inductive) type formers works like a charm, since the unifier can rely on a concrete object (the type former) to drive the instance search.

Attaching structure to a compute type did not work at all. For instance, I was not able to define any (useful) instance for the functor `Hyper Fs`:

```
Hyper : List (Set → Set) → Set → Set
Hyper [] A = A
Hyper (F :: Fs) A = Hyper Fs (F A)
```

As a poor man's alternative, I reified the identity and composition of functors as inductive type formers:

```
data Id (A : Set) : Set where
  I : A → Id A

data Seq (G : Set → Set)(F : Set → Set)(A : Set) : Set where
  S : F (G A) → Seq G F A
```

and attached structure to those: when `Hyper` computes, it will reveal a chain of `Seq`, potentially closed by an `Id`, all of which have the structures wwe are interested in. However, this is not a silver bullet: on `Hyper Fs`, with `Fs` a quantified variable, we cannot access any of its structure. We cannot rely on ad-hoc polymorphism and must manually instantiate the dictionaries.

Also, the paper becomes really interesting in Section 6, in which alignment between matrices of different dimensions is automatically computed through the following (simplified) type family:

```
instance Alignable (F :: Fs) (F :: Gs) where
  (...)
instance Alignable [] (F :: Fs) where
  (...)
```

From a type-theoretic point of view, the first instance is meaningless: what does it mean for `F : Set → Set` to be "equal" on both sides? Equality of functions is notoriously ill-behaved in type theory. I therefore stopped right before Section 6, retreating from a doomed attempt: the very definition of `hyper` as `List (Set → Set)` was already an admission of defeat.

The definition of the `Desc` was suspiciously redundant: the sum product (`_`+_`) codes could have been simulated by Π `Bool` λ `{ true → _ ; false → _ }`. However, the former encoding enables a purely first-order implementation (as an inductively-defined sum) whereas the latter involves a function type. First, intensional equality is not well-behaved (understand: useless) on functions, hence a general tendency to avoid them at all cost. Second, having sums allows us to levitate the description of descriptions as a purely first-order object (no function lying around).

1.6 Other examples

The Mathcomp library defines tuples, which are equivalent to our very own vector. However, rather than defining an inductive family, a tuple of size `n` is implemented as "a list whose length is n". Crucially, the constraint that the length is equal to n does **not** use propositional equality but decidable equality over natural numbers:

```
Structure tuple_of : Type := Tuple {tval :> seq T; _ : size tval == n}.
```

Hedberg theorem tells us that this implies that two tuples are equal if and only if they package the same list (irrespectively of the proof that established that these lists are of the right size). This is not an issue in Vanilla Agda, for which all proofs of equality are equal ("uniqueness of identity proofs", UIP). It is in Coq or in Agda with the flag "–without-k".

Aside from allowing us to avoid relying on UIP, this style also enables the definition of operations on vectors (concatenation, reversal, etc.) to directly re-use the existing operations on lists: one just need to prove that the constraints on lengths are respected.

CHAPTER

TWO

THE BAD

Gibbons definition of hyper-matrices is given as a (nested) datatype:

```
data Hyper' (A : Set) : List (Set → Set) → Set where
  Scalar : A → Hyper' A []
  Prism  : ∀ {F Fs} → Hyper' (F A) Fs → Hyper' A (F :: Fs)
```

Once again, this definition would not be manageable in type theory because it implicitly introduces propositional equality to enforce the constraint `Hyper A (F :: Fs)` in the target of the Prism constructor, which is meaningless for the function `F`.

Sometimes, even apparently good definitions, such as the pervasive vectors, are behaving frustratingly. In each of the following examples, we shall give a working definition with lists and a computationally-equivalent but non-working version on vectors. The point is not that the desired functionality *cannot* be implemented (most of the time, one can rewrite them to side-step the problem) but rather that a "morally correct" definition or property is forbidden.

The first example is a failure of compositionality, first between two functions:

```
module NCons-List (A : Set) where

  ncons : ℕ → A → List A → List A
  ncons zero a vs = vs
  ncons (suc m) a vs = a :: ncons m a vs

  nseq : ℕ → A → List A
  nseq m a = ncons m a []

module NCons-Vec (A : Set) where

  ncons : ∀ {n} (m : ℕ) → A → Vec A n → Vec A (m + n)
  ncons zero a vs = vs
  ncons (suc m) a vs = a :: ncons m a vs

  nseq : (m : ℕ) → A → Vec A m
  nseq m a = {!ncons m a []!}
  -- m + zero != m of type ℕ
  -- when checking that the expression ncons m a [] has type Vec A m
```

and, second, between a single (recursive) function:

```
module RCons-List (A : Set) where

  _++r_ : ∀ {A : Set}→ List A → List A → List A
  [] ++r ys = ys
```

(continues on next page)

(continued from previous page)

```
  (x :: xs) ++r ys = xs ++r (x :: ys)

module RCons-Vec (A : Set) where

  _++r_ : ∀ {A : Set}{m n} → Vec A m → Vec A n → Vec A (n + m)
  [] ++r ys = {!ys!}
  -- .n != .n + 0 of type ℕ
  -- when checking that the expression ys has type Vec .A (.n + 0)
  (x :: xs) ++r ys = {!xs ++r (x :: ys)!}
  -- suc (.n₁ + .n) != .n₁ + suc .n of type ℕ
  -- when checking that the expression xs ++r (x :: ys) has type Vec .A (.n₁ + suc .n)
```

This sort of annoyance also happens when stating properties:

```
module Cats-List (A : Set) where
  cats0 : ∀ (v : List A) → v ++L [] ≡ v
  cats0 [] = refl
  cats0 (x :: v) rewrite cats0 v = refl

module Cats-Vec (A : Set) where

  cats0 : ∀ {n} (v : Vec A n) → v ++ [] ≡ {!v!}
                    -- n != n + zero of type ℕ
                    -- when checking that the expression v has type Vec A (n + zero)
  cats0 = {!!}
```

which means that some proofs are themselves impossible to do compositionally since the necessary lemmas are simply not expressible:

```
module Rcons-List (A : Set) where

  rcons : List A → A → List A
  rcons [] a = a :: []
  rcons (x :: xs) a = x :: rcons xs a

  last' : (a : A) → List A → A
  last' a [] = a
  last' _ (a :: xs) = last' a xs

  last-rcons : ∀ x s z → last' x (rcons s z) ≡ z
  last-rcons x s z = trans (cong (last' x) (cat1 s z)) (last-cat x s (z :: []))
    where postulate
            last-cat : ∀ x (s₁ : List A)(s₂ : List A) → last' x (s₁ ++L s₂) ≡ last' (last' x s₁) s₂
            cat1 : ∀ s z → rcons s z ≡ s ++L (z :: [])

module Rcons-Vec (A : Set) where

  rcons : ∀ {n} → Vec A n → A → Vec A (suc n)
  rcons [] a = a :: []
  rcons (x :: xs) a = x :: rcons xs a

  last' : ∀ {n} → (a : A) → Vec A n → A
  last' a [] = a
  last' _ (a :: xs) = last' a xs

  postulate
```

(continues on next page)

(continued from previous page)

```
    last-cat : ∀ {m n} x (s₁ : Vec A m)(s₂ : Vec A n) → last' x (s₁ ++ s₂) ≡ last' (last' x s₁) s₂
    cat1 : ∀ {n} (s : Vec A n) z → rcons s z ≡ {!s ++ (z ∷ [])!}
      -- n + suc zero != suc n of type ℕ
      -- when checking that the expression s ++ z ∷ [] has type Vec A (suc n)

  last-rcons : ∀ {n} x (s : Vec A n) z → last' x (rcons s z) ≡ z
  last-rcons {n} x s z = trans {!cong (last' {suc n / n + 1}) ?!} (last-cat x s (z ∷ []))
     -- Heterogeneous equality necessary
```

An elegant solution (due to Conor McBride) to the problem of stating and using the above equalities consists in generalizing slightly the definition of equality so that it becomes **heterogeneous**:

```
data _≅_ {A : Set} (x : A) : {B : Set} → B → Set where
  refl : x ≅ x
```

However, we are still at a loss to write programs such as `nseq` or `_++r_` in a compositional, proof-free manner.

CHAPTER

THREE

THE UGLY

Nested datatypes, such as:

```
data Bush (A : Set) : Set where
  leaf : A → Bush A
  node : Bush (A × A) → Bush A

size : ∀ {A} → Bush A → ℕ
size (leaf x) = 1
size (node b) = let n = size b in n + n
```

are slightly suspicious in the sense that the constructors are actually encoding the *size* of the container, the constructor `leaf` containing a tuple whose shape is determined by the previous `node` constructors. First, programming with such a definition is not going to be pleasant. Second, to encode structural properties, we have a better tool at hand: indices and inductive families!

Addressing the second point, one could write:

```
data Bush' (A : Set)(n : ℕ) : Set where
  leaf : Vec A n → Bush' A n
  node : Bush' A (n + n) → Bush' A n
```

Addressing the first point, one realizes that Peano numbers are not the right fit for indexing this structure:

```
data Bin𝕟 : Set where
  #0 : Bin𝕟
  #1 : Bin𝕟 → Bin𝕟

to𝕟 : Bin𝕟 → ℕ
to𝕟 #0 = 1
to𝕟 (#1 n) = let x = to𝕟 n in x + x

data Bush'' (A : Set) : Bin𝕟 → Set where
  leaf : A → Bush'' A #0
  node : ∀ {n} → Bush'' A n → Bush'' A n → Bush'' A (#1 n)
```